WEBSITE BOOK EDITION

Pixels → Perception

From Pixels to Understanding

Foundations • Vision • Language • Learning • Theory

Volume 2 of 4

Language and Pattern Recognition

English Edition • Volume 2

2026-11-05

About this edition

This is volume 2 of 4 of the Pixels → Perception course, *Language and Pattern Recognition*. It contains 22 published tutorials and 0 blog posts, in the order of the website's learning tracks. This is a typeset edition of the website, not a separately peer-reviewed publication.

The prose, code, equations, and references retain the website's content. Exercise hints are expanded for print. Static figures remain in the book, image comparisons show their input and output images, and interactive demonstrations link to the website. Reference numbers are local to each tutorial.

The website is still under review. AI tools assisted its production, and errors or omissions may remain. Read it alongside the original sources and consult the website for subsequent corrections.

English and Traditional Chinese, and each volume, are separate PDFs. Links to tutorials in this volume navigate within the PDF. Links to tutorials in other volumes, to specific web sections, interactive demonstrations, notebooks, and external sources remain online. Content snapshot: 2026-11-05

Source revision: `2b3c30d`

https://blog.shyandram.dev/

Contents

I Natural Language Processing 1

NLP 01 ·Introduction to Natural Language Processing 2

NLP 02 ·Linguistic Essentials 17

NLP 03 ·Neural Networks for NLP 44

NLP 04 ·Lexical Semantics and Word Embeddings 70

NLP 05 ·Recurrent Neural Networks for Language 99

NLP 06 ·Sequence Labeling with RNNs (POS Tagging, Information Extraction) 125

NLP 07 ·Attention Mechanism and Transformers 153

NLP 08 ·Sequence-to-Sequence Models with Transformers (Machine Translation, Chatbots) 172

NLP 09 ·Language Modeling with Transformers (Natural Language Generation) 198

NLP 10 ·Pretrained Models: BERT 237

NLP 11 ·Classification with BERT (Sentiment Analysis, Natural Language Inference) 256

NLP 12 ·Syntactic and Semantic Tasks 279

II Pattern Recognition 309

PR 01 ·Patterns, Probability and Bayes Decisions 310

PR 02 ·Density Estimation and Bayesian Learning 328

PR 03 ·Linear Models for Regression and Classification 347

PR 04 ·Neighbors, Kernels and Nonlinear Classifiers 366

PR 05 ·Feature Selection, Transforms and Dimensionality 387

PR 06 ·Template Matching and Structured Recognition 408

PR 07 ·Clustering, Mixtures and EM 427

PR 08 ·Approximate Inference and Bayesian Model Complexity 452

PR 09 ·Ensembles, Validation and Cluster Validity 475

PR 10 ·Recognition Capstone: Rejection, Shift and Learned Features 499

Part I

Natural Language Processing

NLP 01 • Introduction to Natural Language Processing

What NLP is, why language is hard for computers, how the field moved from rules to statistics to pretrained models, the main tasks, Chinese-specific issues, and a practical pipeline from task definition to error diagnosis.

Online edition: https://blog.shyandram.dev/en/learn/nlp/introduction/

What you'll learn

- Explain why language is hard for computers, with an example of ambiguity at each linguistic level.
- Trace the main eras of NLP — rules, statistical learning, neural networks and pretrained models — through the papers that marked them.
- Classify common NLP tasks by the shape of their output and find the tutorial that covers each.
- Recognize the issues specific to Chinese text: word segmentation, and the Traditional and Simplified character sets.
- Define an NLP task through its input, output and required linguistic evidence, and preserve that evidence through normalization and tokenization.
- Build data splits and diagnose errors at the stage where they originate.

The big picture

Natural language processing (NLP) is the study of computer systems that analyze, interpret or produce human language. An NLP pipeline converts a particular language problem into a computation. Consider "Pump B: not ready." A maintenance application might need the equipment name and its stated readiness. A message-routing application might need only a category. A conversational assistant might need to ask what remains unfinished. These are different tasks even though the input string is identical.

The first job is to decide which answer the application needs and what evidence supports it. Then preserve that evidence while turning text into model input. A powerful model cannot recover a distinction that preprocessing has already erased, and a plausible label is useless if it describes the wrong target.

> **Explain like I'm five**
>
> Imagine sorting letters into trays. You first decide what each tray means. Then you read the letters without rubbing out important words. Turning "the parcel did not arrive" into "the parcel arrived" makes the sorting problem wrong before you even choose a sorter.

Hagiwara's application-first treatment and McRoy's treatment of language and its data structures provide the two anchors for this tutorial [1, 2]. We first look at why language is difficult, how the field arrived at today's methods, which tasks it solves and what changes for Chinese text. Then we follow one practical route: define the task, preserve text, choose computational units, construct data and evaluate the result. NLP 02 develops the language structures behind those decisions, and the roadmap at the end links the whole track.

1. Why language is hard for computers

Plain version. The same string of characters can mean different things, and what it means depends on structure, context and the speaker's purpose. A computer sees only the characters unless we give it the rest.

Ambiguity appears at every level of linguistic analysis. Each row below is developed with examples and code in NLP 02:

Level	Example	Two readings
Word boundaries	學生會說話	"students can speak" / "the student council speaks"
Word category	"the pilots can land" / "the metal can fell"	"can" as an auxiliary / as a noun
Word sense	"window"	an opening in a wall / an area on a screen
Attachment	"The technician inspected the panel with the camera."	the camera is the tool / the camera is on the panel
Scope	"Every technician uses a camera."	possibly different cameras / one shared camera
Reference	"Mei put a camera beside a lamp. It was overheating."	"it" is the camera / the lamp
Intention	"Can you lower the gain?"	a question about ability / a request to act

People resolve most of these without noticing, using the surrounding sentence, the situation and knowledge of the world. A program must either be given that context or learn to approximate it from data. Three further properties compound the difficulty.

- **Productivity.** Speakers build new words and sentences from familiar parts all the time, so no finite list of inputs covers the language.
- **Skewed frequency.** A few words are extremely common and most words are rare. NLP 02 measures this with Zipf's law: in one corpus, a third of all word types occur exactly once. Rare words are where models have the least evidence.
- **Variation.** Dialects, registers, spelling conventions, mixed-language messages and writing systems all differ. A system trained on one variety can fail quietly on another.

None of this means NLP is impossible. It means every system makes choices about which distinctions it preserves, and good engineering makes those choices explicit.

2. A short history: rules, statistics, neural networks and pretraining

Plain version. Early systems followed rules that people wrote by hand. Later systems learned probabilities from large collections of text. Neural networks then learned their own features, and today most systems start from a large model pretrained on unlabeled text and adapt it to the task.



Figure 1 — Milestones cited in this tutorial, grouped into four approximate eras. Events are equally spaced, not to scale; in practice the approaches overlap.

Rules and hand-built knowledge. Weizenbaum's ELIZA (1966) responded to typed English by matching keywords and applying decomposition and reassembly rules from a script; its best-known script imitated a psychotherapist's conversational style [6]. Systems of this era encoded grammars, lexicons and patterns by hand. They could be precise inside their design, but every new phrasing or domain required new rules.

Statistical learning from corpora. Over the following decades, researchers increasingly estimated models from data instead of writing every rule. Brown and colleagues (1993) described a series of statistical translation models whose parameters are estimated from pairs of sentences that are translations of each other [7]. In the same year the Penn Treebank released a large English corpus annotated with part-of-speech tags and syntactic structure [8], giving taggers and parsers shared training data and a shared test. The key change was that the evidence came from counts in corpora, and evaluation on held-out data became standard practice.

Neural networks. Bengio and colleagues (2003) trained a neural language model that learned a distributed representation for each word together with the probability of word sequences [9]. Collobert and colleagues (2011) used one neural architecture for part-of-speech tagging, chunking, named-entity recognition and semantic role labeling, learning internal representations largely from unlabeled text instead of hand-engineered features [10]. Mikolov and colleagues (2013) made word vectors cheap to train on large corpora [11]. Models still tended to be built for one task at a time.

Pretrained models and large language models. The Transformer (2017) replaced recurrence with attention [12]. ELMo (2018) supplied context-dependent word representations from a pretrained bidirectional language model [13], and BERT (2019) pretrained a deep bidirectional Transformer on unlabeled text and then fine-tuned it for each task with a small output layer [14]. GPT-3 (2020) showed that a large autoregressive model could perform many tasks from a few examples in its prompt, without gradient updates [15], and InstructGPT (2022) fine-tuned such a model with human feedback to follow instructions [16]. Zhao and colleagues' survey organizes the large-language-model literature around pretraining, adaptation, utilization and capacity evaluation [17].

The eras overlap: rule-based components, counting methods and neural models all remain in use. What carries over is the pipeline in Sections 4–9. Whatever the model, someone still has to define the task, preserve the evidence, choose units, build data and evaluate.

3. The main tasks and applications

Plain version. Most NLP tasks can be grouped by what they output: a label for the whole text, a label for each word, a structure, a passage from a collection, or new text.

Output shape	Example tasks	Applications	Covered in
One label per text or text pair	topic classification, sentiment, spam detection, natural language inference	routing, moderation, review analysis	NLP 03, NLP 11
One label per token or span	part-of-speech tagging, named-entity recognition, information extraction	turning reports into records	NLP 06
A linguistic structure	parsing, semantic roles, coreference	analysis of who did what to whom	NLP 12
Evidence from a collection	search, extractive question answering	document search, help desks	NLP 11, NLP 09
New text conditioned on input	translation, summarization, dialogue	translators, assistants, chatbots	NLP 08, NLP 09
A reusable representation	word and sentence embeddings, pretrained encoders	similarity, clustering, transfer to new tasks	NLP 04, NLP 10

The same application often combines several rows. An assistant that answers maintenance questions may classify the question, retrieve passages, extract an equipment name and generate a reply. Each step has its own output and its own errors, which is why the rest of this tutorial treats the task definition as the first design decision.

4. Specify the task before choosing a model

Plain version. "Understand this message" is too broad to train or evaluate. Specify which part of its meaning must become an output.

For the invented message "Pump B: not ready," three possible outputs are:

Task	Output	Evidence that matters
Topic classification	`maintenance`	The message concerns equipment status
Entity extraction	`Pump B`, with its source span	The exact equipment-name characters
Status extraction	Equipment `Pump B`; readiness `false`	The name and the negated readiness statement

The first answer does not establish the third. A classifier can recognize a topic from "pump" while ignoring "not." A status extractor must preserve that negation. A system that returns equipment names also needs boundary rules: should `B` be included, and should the colon be excluded? Write these decisions into the annotation instructions.

Now compare "Pump B was not ready yesterday" with "Pump B is ready now." The equipment is unchanged, but time and polarity differ. A current-status application needs a rule for multiple dated statements; a historical extractor may keep both. The correct answer depends on the task definition and available context, rather than on one universal interpretation of the string.

For "It still isn't ready," the equipment is absent from the message. If the application supplies earlier conversation, reference resolution can use it. If it supplies only this isolated sentence, identifying `Pump B` would add information that was never available. This is an input-design issue, developed further in NLP 12 (coreference) and NLP 08 (dialogue state).

Separate a prediction from an action. Predicting a request to reschedule does not determine which booking to change. Start with an output that can be checked against the text: an intent label, an equipment span or a stated event. Later tutorials add the contextual reasoning needed to resolve missing details.

5. Preserve the original text and the evidence

Plain version. The text used for model input and the text shown as evidence can be different representations of the same message. Keep a way to connect them.

A pipeline can be written as:

$$r \xrightarrow{g} x \xrightarrow{f_\theta} \hat{y}.$$

Here r is the raw input record; g is the preprocessing and numerical encoding function; x is model input; f_θ is a predictor with parameters θ; and $\hat{y}$ is its output. A record can include text and permitted context. Preprocessing includes decisions about normalization, token boundaries and vocabulary IDs.

If two different records become the same x, a deterministic predictor must return the same result for both:

$$g(r_1) = g(r_2) \quad \Longrightarrow \quad f_\theta(g(r_1)) = f_\theta(g(r_2)).$$

Here r_1, r_2 are two raw records. This simple identity explains why preprocessing is part of the model. If removing "not" makes affirmative and negative statements identical, changing the neural architecture cannot separate them afterward. The same reasoning applies when lowercasing merges `US` with `us` or punctuation removal collapses a decimal number into another string.

Some equivalences are useful. Unicode NFC normalization maps canonically equivalent character sequences to a common form [3]. This is different from choosing to remove accents, punctuation or case. The first concerns a specified character equivalence; the others may remove distinctions that the task needs. Decide what should remain equivalent for this application, rather than treating every transformation as harmless cleaning.

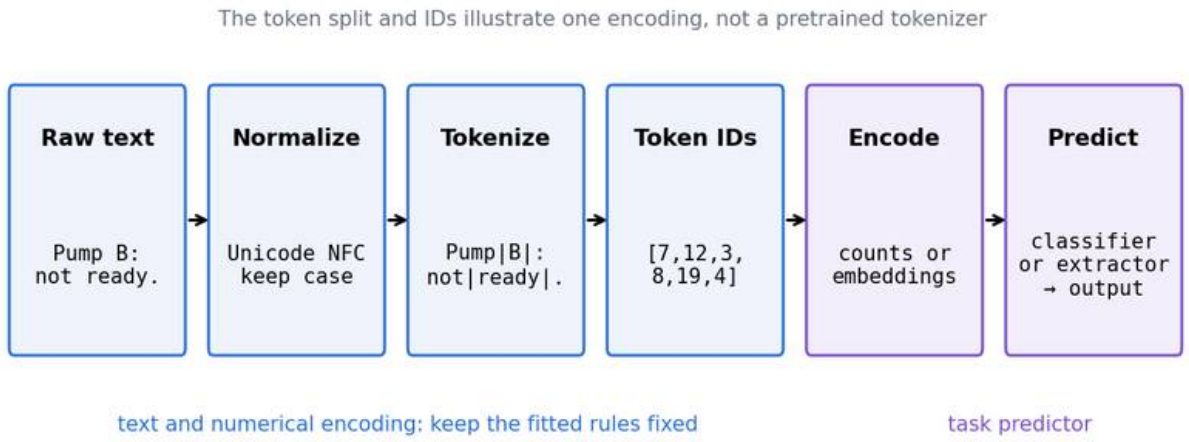


Figure 2 — An illustrative text-processing pipeline. Preserve a connection to the original text, and use the same fitted encoding during training and evaluation.

A worked normalization and span example

The word `Café` can use a precomposed accented letter or a base `e` followed by a combining accent. They look similar but occupy different numbers of code points [3]. If an extractor reports character positions, normalization can therefore shift a later span.

A span $[a, b)$ means the characters starting at offset a and ending just before b in a **specified string**. Both offsets are nonnegative integers, with $a \leq b$. They are not meaningful without knowing which text representation and indexing unit they refer to.

```
import re
import unicodedata

raw = 'Café Pump B: not ready.'
normalized = unicodedata.normalize('NFC', raw)
needle = 'Pump B'
raw_start = raw.index(needle)
```

```
normal_start = normalized.index(needle)
raw_span = (raw_start, raw_start + len(needle))
normal_span = (normal_start, normal_start + len(needle))
tokens = [(m.group(), m.start(), m.end())
          for m in re.finditer(r'\w+|[^\w\s]', normalized)]
assert normalized != raw
assert unicodedata.normalize('NFC', normalized) == normalized
assert raw[raw_span[0]:raw_span[1]] == needle
assert raw[normal_span[0]:normal_span[1]] != needle
assert all(normalized[start:end] == token for token, start, end in tokens)
print('code points:', len(raw), '->', len(normalized))
print('raw / normalized evidence spans:', raw_span, normal_span)
print('normalized tokens:', [token for token, _, _ in tokens])
```

The code prints `24 -> 23` code points. The raw equipment span is `(6, 12)` and the normalized span is `(5, 11)`. Both select `Pump B` in their own strings; applying the second span to the first string selects the wrong characters. The final assertion checks the normalized tokens against their own source offsets.

For extraction, either maintain an alignment from transformed positions back to the original text or retain annotations directly on the raw string. This example demonstrates the failure; it does not implement a general alignment algorithm. NLP 06 explains how word, token and character spans become structured outputs.

6. Decide which units the computation will use

Plain version. A word, a character and a model token answer different questions. Choose computational units without confusing them with all the units of language.

In "a cat and a dog," whitespace splitting produces five token **occurrences** and four distinct **types**: `a`, `cat`, `and`, `dog`. A vocabulary assigns an ID to each retained type. IDs are names for table entries: ID 7 does not represent a larger meaning than ID 3.

The string "Pump B: not ready." raises different choices. Whitespace splitting attaches punctuation to words. A punctuation-aware splitter may produce `Pump`, `B`, `:`, `not`, `ready`, `.`. A subword tokenizer can split a word further. All can be valid computational encodings, but their token positions are different. Word labels must be aligned to whichever encoding the model actually uses.

Subword tokenizers learn from data which pieces to keep. NLP 02 trains a small byte-pair encoding (BPE) tokenizer step by step and compares it with WordPiece, the unigram model and SentencePiece. Its central lesson is that learned pieces reflect frequency, not morphology, and that a tokenizer is a complete fitted procedure, not just a vocabulary list.

Vocabulary coverage and unknown inputs

A word vocabulary contains only its retained entries. Map an unseen word to an unknown marker, and all such words share that ID. This can be acceptable when the task depends on familiar surrounding words, but it removes the difference between an unseen equipment name and an unseen account name.

Define an illustrative unknown-token rate:

$$u = \frac{\sum_{i=1}^{N} \mathbf{1}[t_i \notin V]}{N}.$$

Here u is the unknown-token rate; t_i is token occurrence i under a declared tokenizer; V is its fitted vocabulary; $N > 0$ is the number of evaluated occurrences; and $\mathbf{1}[\cdot]$ equals one when the condition is true. A repeated unknown word counts repeatedly. This measures missing vocabulary entries, not lost meaning or task accuracy.

7. Chinese text: segmentation and two character sets

Plain version. Chinese is written without spaces between words, so deciding where words begin and end is itself an analysis. Chinese is also written in two standard character sets, Traditional and Simplified, and a model fitted on one sees many unfamiliar characters in the other.

Word segmentation

Chinese text often lacks spaces between words. "學生會說話" permits 學生 / 會 / 說話 (students can speak) or 學生會 / 說話 (the student council speaks); context helps choose the intended reading. A model tokenizer may simply use character or subword pieces. Its segmentation need not be a complete morphological or syntactic analysis. NLP 02 enumerates both readings with a small program and distinguishes those linguistic structures.

Segmentation is also a matter of convention. The Second International Chinese Word Segmentation Bakeoff used four corpora, two in Traditional characters (Academia Sinica in Taiwan and City University of Hong Kong) and two in Simplified characters (Peking University and Microsoft Research), each segmented under its provider's own standard [18]. When a dataset has word-level labels, the standard it follows is part of the task definition, and a character-based model must align its outputs to those words before evaluation.

Traditional and Simplified characters

Text from Taiwan and Hong Kong is mainly written in Traditional characters and text from mainland China in Simplified characters; the bakeoff corpora above follow that division [18]. Many characters are the same in both, but many are different Unicode code points: Traditional 機 is U+6A5F while its Simplified counterpart is U+673A. Unicode normalization does not merge them, because they are distinct characters, not alternative encodings of one character.

Conversion is not a one-to-one character swap. The Unihan database records Simplified and Traditional variants for each character and notes that context can decide the mapping: when converting Simplified to Traditional, U+810F becomes 臟 when it means "viscera" and 髒 when it means "dirty" [19]. Word choice can also differ between regions, so even a perfect character converter does not turn one variety's text into the other's.

The script mismatch shows up directly as vocabulary coverage. The program below fits a character vocabulary on one Traditional sentence and measures the unknown-token rate u on a new sentence written both ways. The Simplified version is written with escape codes so it is visible which characters differ.

```
vocab = set("這台相機已經校正好了")              # character vocabulary fitted on Traditional
 ↪  text
tests = {"Traditional": " 那台相機已經過熱了",
         # The same sentence in Simplified characters, written as escapes:
         "Simplified": " 那台相\u673a 已\u7ecf\u8fc7\u70ed 了"}
for name, text in tests.items():
    unknown = [f"U+{ord(ch):04X}" for ch in text if ch not in vocab]
    print(f"{name:11s} u = {len(unknown)}/{len(text)}  unknown: {unknown}")
```

The Traditional sentence has $u = 3/9$: 那, 過 and 熱 were not in the tiny training text. The Simplified sentence has $u = 5/9$: 那 (identical in both scripts) and the Simplified forms of 過 and 熱 are unknown as before, plus the Simplified counterparts of 機 and 經, which the vocabulary contains only in Traditional form. Real vocabularies are far larger, but the effect is the same in kind: a model trained mainly on one script has less evidence for the other. Decide whether to train on both, convert with a context-aware tool, or evaluate each script separately, and report which you chose.

8. Build examples and labels that answer the task

Plain version. A corpus is a collection of observations. Training examples are constructed from those observations according to a task. The construction determines what the model learns.

For classification, an example might be a whole message paired with one category. For extraction, it might be a message paired with character spans and their labels. For language modeling, the text supplies prediction targets without category annotations. These distinctions explain why the same corpus can support different objectives [1, 2].

Suppose a maintenance message contains "the camera is fine, but the delivery was late." A product-sentiment label, a delivery-sentiment label and a whole-message label need not agree. Include the target in the instructions. If annotators choose different targets, a disagreement is evidence about the task definition before it is evidence about model quality.

Define the unit of independence for your split. Two paraphrases of one complaint are distinct strings but may describe the same event. If one appears in training and one in testing, the test can reward recognition of that event rather than transfer to a new one. Split related messages,

conversations or document revisions together when their shared information would otherwise leak.

There are two kinds of fitted state to track. The predictor learns weights from training labels. A tokenizer or vocabulary may learn from unlabeled training text. Keeping labels out of testing is not enough if the feature pipeline still uses test text to decide its vocabulary. Freeze both according to the experiment's stated rules.

Use a small baseline to diagnose the representation

Begin with a representation whose behavior is easy to inspect. A count-based message representation preserves which words appear while ignoring their order. "The dog bit the man" and "The man bit the dog" then produce the same counts despite reversing who did what. Every deterministic classifier using only those counts must give the same prediction for both.

This is a precise representation limitation. More examples cannot solve this pair while the feature map remains identical. Local phrase features can separate some order differences; contextual models can use richer input interactions. NLP 03 develops and tests those choices. Choose added structure in response to an observed failure, so the experiment identifies what changed.

9. Evaluate the output and diagnose the cause

Plain version. Evaluation should tell you what went wrong and where to improve it. One aggregate score rarely identifies the cause.

For classification, inspect errors for each category, especially rare ones; accuracy can conceal a category that is never predicted. Precision asks how often returned items are correct, while recall asks how many required items were returned. Both depend on what counts as an item [5]. For extraction, finding the right word with the wrong span is a boundary error. For a status record, the correct equipment paired with the wrong polarity is a meaning error.

Keep a few challenge pairs where one linguistic difference matters: "ready" versus "not ready," `Pump A` versus `Pump B`, and "yesterday" versus "now." Hold the intended task fixed while changing that difference. A correct answer on an ordinary message does not explain whether the system uses the distinction needed by these cases.

Observation	Inspect first	What an improvement must establish
Unseen equipment names collapse together	Vocabulary and fallback	Names remain distinguishable in the input
Correct words, incorrect evidence highlight	Normalization and offset alignment	Returned offsets refer to the displayed source
Negation reverses the true label but not the prediction	Feature map and supervision	The relevant distinction survives encoding and affects the decision
Strong validation result, weak new-customer result	Split independence and domain differences	Performance transfers to the intended population
Good results on one Chinese script, poor on the other	Character coverage and conversion	Both scripts are represented, or the gap is reported

Change one stage at a time when possible. If a preprocessing change and a model change happen together, a better score does not tell you which caused the improvement. Inspect transformed inputs before interpreting weights or outputs. This turns error analysis into a testable explanation instead of a list of model names.

10. Roadmap of the NLP track

The track follows the eras in Section 2: language structure first, then neural building blocks, sequence models, attention and Transformers, and finally pretrained models and structured tasks.

Tutorial	What it adds
NLP 01 ·Introduction to Natural Language Processing	Why language is hard, history, tasks, Chinese issues and the task-to-evaluation pipeline
NLP 02 ·Linguistic Essentials	Morphology, tokens and subwords, parts of speech, syntax, meaning, context, corpora and Zipf's law
NLP 03 ·Neural Networks for NLP	Count features, TF–IDF, logistic and neural text classifiers
NLP 04 ·Lexical Semantics and Word Embeddings	Distributional meaning, count-based vectors, word2vec
NLP 05 ·Recurrent Neural Networks for Language	n-gram language models, perplexity, recurrent language models
NLP 06 ·Sequence Labeling with RNNs	Part-of-speech tagging, entity recognition and span evaluation
NLP 07 ·Attention Mechanism and Transformers	Attention, contextual representations and position
NLP 08 ·Sequence-to-Sequence Models with Transformers	Encoder–decoder models, decoding, machine translation and chatbots
NLP 09 ·Language Modeling with Transformers	Causal language models, text generation and retrieval-augmented generation
NLP 10 ·Pretrained Models: BERT	Masked-language pretraining and the BERT encoder
NLP 11 ·Classification with BERT	Sentiment analysis, natural language inference and extractive QA heads
NLP 12 ·Syntactic and Semantic Tasks	Parsing, semantic roles, coreference and discourse

Related tutorials on this site cover the general machinery: ML 02 for neural networks, ML 04 for recurrent networks, ML 05 for attention and the Transformer block, and IT 01 for entropy and cross-entropy.

A first route through the papers. Start with the main chapters; these twelve guides are optional checkpoints, not extra prerequisites. Each guide opens with what to know first and ends with a next step. The route groups questions rather than publication dates: within attention, read Seq2Seq → Bahdanau → Transformer to follow the architectural motivation.

Paper route 1 • Units and word vectors

Know first: Tokenization, vector dot products, cross-entropy

Read: BPE → word2vec → GloVe

Check your understanding: What is a token, and what evidence gives its vector meaning?

Paper route 2 • Local and sequential decisions

Know first: Convolution, recurrent states, conditional probability

Read: TextCNN → Seq2Seq → BiLSTM-CRF

Check your understanding: How do local features become a class, a sequence or a constrained label path?

Paper route 3 • Attention and pretraining

Know first: Encoder–decoder flow, softmax, masks

Read: Bahdanau attention → Transformer → BERT

Check your understanding: Which context can the model read, and what is it trained to predict?

Paper route 4 • Task objectives and structure

Know first: Fine-tuning, sentence pairs, span evaluation

Read: Sentence-BERT → InstructGPT → Coreference

Check your understanding: How do similarity, human preference and entity links require different evidence?

For each paper, explain one arrow in its original figure, define the symbols in its objective, and identify the split and model setting behind one reported result. If any step is unclear, return to the linked parent chapter before continuing.

Modern view

Chang and colleagues' evaluation survey organizes what, where and how to evaluate language models [4]. For a pipeline, that motivates separate checks of encoding, task output and application behavior. A fluent generated explanation is not an alternative measurement of equipment-name recall; the two outputs answer different questions.

Zhao and colleagues' survey of large language models describes the current default: pretrain one large model on broad text, then adapt it (for example by instruction tuning or alignment with human preferences), use it through prompting, and evaluate its capacities across many tasks [17]. Its organization mirrors the history in Section 2, where each era kept the task definitions of the previous one and changed how the mapping from text to output was obtained.

For current systems, the transferable lesson is to inspect interfaces before comparing model families. A larger input budget is useful only if the relevant text survives truncation. A multilingual tokenizer is useful only if its representations support the distinctions the task needs, including both Chinese character sets when users write in both. A pretrained representation can reduce

labeled-data requirements, but the application's targets still require clear supervision and evaluation. Later tutorials analyze the respective mechanisms rather than treating these as automatic benefits.

Open questions include how to measure linguistic coverage beyond character coverage, how segmentation affects different languages and registers, and how to separate representation failures from insufficient labels or missing context. A useful small study compares two encodings with one fixed task, independently split data and a few diagnostic linguistic cases. Report what each encoding preserves or loses, then explain the observed errors in those terms.

Key takeaways

- Language is ambiguous at every level, from word boundaries to intention; context and world knowledge resolve what the string alone cannot.
- NLP moved from hand-written rules to corpus statistics, then to neural networks and pretrained models; the task-definition and evaluation problems carried over unchanged.
- Group tasks by output shape: text labels, token labels, structures, retrieved evidence, generated text or representations.
- Chinese adds word segmentation under competing standards and two character sets whose conversion is not one-to-one.
- Define the desired output and its linguistic evidence before choosing a model, and keep raw text distinct from transformed model input.
- A predictor cannot recover distinctions erased by its preprocessing function; tokens and vocabulary IDs are computational units, not complete analyses of language.
- Fit learned representations on permitted training text, keep related observations together in splits, and diagnose vocabulary, boundary, meaning and transfer errors before changing architecture.

Exercises

1. For "Pump B was not ready yesterday, but it is ready now," define labels for topic classification, historical status extraction and current-status extraction. Which labels require a time policy?

   Hint

   The equipment name can stay fixed while status values and their validity times differ. Decide whether you return one state or two dated events.

2. Replace the decomposed accent in the normalization example with a precomposed one. Explain which offsets change and which assertions remain valid.

   Hint

   Compare raw and normalized strings separately. An offset is a location in one representation, not in all equivalent-looking strings.

3. Make a preprocessing rule remove “not.” Construct two messages with different required status labels but identical transformed inputs, and prove why a deterministic predictor cannot separate them.

Hint

Apply the equality involving $g(r_1)$ and $g(r_2)$. The contradiction arises before training.

4. In the script-coverage program, refit the vocabulary on the union of the Traditional training sentence and its Simplified version. Recompute u for both test sentences. Then explain why converting every Simplified test character to Traditional before lookup is not always possible with a character table alone.

Hint

The union removes the script-specific unknowns but not 那, 過 and 熱. For conversion, use the Unihan example: one Simplified code point maps to two Traditional characters depending on meaning.

5. Group three rewritten messages from one incident into a train/test dataset. Design a split and two diagnostics that distinguish event memorization from transfer to new incidents.

Hint

Split incident groups together, then examine unseen names and unseen wording separately. Keep the label policy fixed.

References

1. Masato Hagiwara, *Real-World Natural Language Processing: Practical Applications with Deep Learning*, Chapters 1–3 and 10–11, Manning, 2021. Publisher
2. Susan McRoy, *Principles of Natural Language Processing*, open textbook, 2021. Book
3. Unicode Consortium, “Unicode Normalization Forms,” Unicode Standard Annex #15, revision 58, 2026. Specification
4. Yupeng Chang et al., “A Survey on Evaluation of Large Language Models,” *ACM Transactions on Intelligent Systems and Technology*, 2024. Survey
5. Scikit-learn developers, “Metrics and Scoring: Quantifying the Quality of Predictions,” documentation, current online edition. Documentation
6. Joseph Weizenbaum, “ELIZA—A Computer Program for the Study of Natural Language Communication Between Man and Machine,” *Communications of the ACM*, 9(1), pp. 36–45, 1966. DOI
7. Peter F. Brown, Stephen A. Della Pietra, Vincent J. Della Pietra and Robert L. Mercer, “The Mathematics of Statistical Machine Translation: Parameter Estimation,” *Computational Linguistics*, 19(2), 1993. Paper

8. Mitchell P. Marcus, Beatrice Santorini and Mary Ann Marcinkiewicz, “Building a Large Annotated Corpus of English: The Penn Treebank,” *Computational Linguistics*, 19(2), 1993. Paper
9. Yoshua Bengio, Réjean Ducharme, Pascal Vincent and Christian Jauvin, “A Neural Probabilistic Language Model,” *Journal of Machine Learning Research*, 3, pp. 1137–1155, 2003. Paper
10. Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu and Pavel Kuksa, “Natural Language Processing (Almost) from Scratch,” *Journal of Machine Learning Research*, 12, pp. 2493–2537, 2011. Paper
11. Tomas Mikolov, Kai Chen, Greg Corrado and Jeffrey Dean, “Efficient Estimation of Word Representations in Vector Space,” arXiv:1301.3781, 2013. Paper
12. Ashish Vaswani et al., “Attention Is All You Need,” *NeurIPS*, 2017. Paper
13. Matthew E. Peters et al., “Deep Contextualized Word Representations,” *NAACL-HLT*, 2018. Paper
14. Jacob Devlin, Ming-Wei Chang, Kenton Lee and Kristina Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” *NAACL-HLT*, 2019. Paper
15. Tom B. Brown et al., “Language Models are Few-Shot Learners,” *NeurIPS*, 2020. Paper
16. Long Ouyang et al., “Training Language Models to Follow Instructions with Human Feedback,” *NeurIPS*, 2022. Paper
17. Wayne Xin Zhao et al., “A Survey of Large Language Models,” arXiv:2303.18223, 2023 (continuously updated). Survey
18. Thomas Emerson, “The Second International Chinese Word Segmentation Bakeoff,” *Proceedings of the Fourth SIGHAN Workshop on Chinese Language Processing*, 2005. Paper
19. Unicode Consortium, “Unicode Han Database (Unihan),” Unicode Standard Annex #38, revision 41, 2026. Specification

NLP 02 • Linguistic Essentials

The language structure every NLP system works with: units and morphology, tokens and subwords, parts of speech, phrase and dependency structure, meaning, context and discourse, corpora, annotation and Zipf's law.

Online edition: https://blog.shyandram.dev/en/learn/nlp/linguistic-essentials/

What you'll learn

- Distinguish language from its spoken, signed and written forms, and distinguish linguistic units from computational tokens.
- Analyze word structure, compare stemming with lemmatization, and enumerate ambiguous Chinese segmentations.
- Train a small BPE tokenizer and relate it to WordPiece, the unigram model and SentencePiece.
- Use Universal Dependencies part-of-speech tags and explain why grouping, grammatical relations and agreement carry information that word counts omit.
- Work out lexical meaning, composition, negation and quantifier scope, and separate assertion, presupposition, implicature and discourse relations.
- Read corpora and annotations critically: agreement between annotators, Zipf's law and the long tail of rare words.

The big picture

Language lets people describe situations, ask questions, coordinate action and change what others know. The visible sentence is only part of that activity. Its internal structure constrains interpretation; speakers also rely on shared objects, earlier conversation and expectations about why someone spoke.

> **Explain like I'm five**
>
> Imagine someone hands you building blocks and says, "Build that one again." Knowing the blocks' names does not tell you which building "that one" means, what happened last time, or whether the speaker wants the same colors. Language works similarly: words supply pieces, grammar connects them, and the situation supplies information the speaker leaves unsaid.

Consider "The technician inspected the panel with the camera." Did the technician use a camera, or inspect a panel that had a camera attached? Knowing every dictionary entry does not settle the question. Consider "Can you inspect it tomorrow?" Now the object, reference date and intended request depend on context. A useful analysis identifies these alternatives before choosing a representation or label.

This tutorial supplies the language foundation for the practical route in Hagiwara's *Real-World Natural Language Processing* [1] and the symbolic and computational route in McRoy's *Principles of Natural Language Processing* [2]. *Essentials of Linguistics*, an open textbook, supplies additional linguistic grounding [3]. The examples here are original. NLP 01 introduced the task-to-evaluation pipeline; this tutorial supplies the units and structures that pipeline must preserve. Later tutorials turn them into models: NLP 06 tags parts of speech and entities, and NLP 12 parses sentences, labels semantic roles and resolves coreference. These levels interact rather than forming a compulsory sequence of separate software modules.

1. Language, modality and units

Plain version. Speech, signing and writing are different ways of expressing language, and each records different information. A computer's "token" is yet another unit, chosen for processing.

Language is a system of structured expression and interpretation. Speech and writing are ways to express it; they are not interchangeable containers. Signed languages use systematic visual-manual expression, and tactile signing provides another modality [4]. A written transcript preserves selected information from an interaction, while leaving out timing, gaze, gesture and much of its physical setting.

Sounds and written symbols answer different questions

Phonetics studies the physical production and perception of speech sounds. **Phonology** studies their patterning within a language. A **phoneme** is a contrastive sound unit in a particular linguistic system; different physical realizations can belong to the same phoneme [5]. A **grapheme** is a unit of a writing system. These units need not map one to one.

English "ship" has four letters, yet its initial "sh" represents one consonant sound. Conversely, a letter can contribute differently in different words. Counting letters therefore cannot directly answer a question about sound structure. A speech application and a text application may need different annotations even when their users intend the same words.

Tone and intonation also differ [6]. Mandarin mā, má, mǎ and mà distinguish the ordinary written words 媽, 麻, 馬 and 罵. Their tone patterns contribute to lexical contrasts. The broader contour of an utterance can also convey questioning or emphasis. In English, "Really." and "Really?" suggest different interpretations in writing, but punctuation only approximates how an actual speaker expresses disbelief, curiosity or confirmation. The written question mark does not recover the original voice.

Characters, words and tokens

A **word form** is a particular realized form; a **lexeme** groups related forms as one vocabulary item. For a simplified analysis, "repair," "repairs" and "repaired" are forms of the verb lexeme REPAIR.

A **lemma** is a chosen citation form used to index that item. Lemmatization is thus an annotation decision, not a claim that the surface forms are interchangeable in a sentence [2, 3].

The distinctions matter in "The technician repairs cameras" versus "The technician repaired cameras." Grouping both verbs under REPAIR helps search for the activity, but discarding the forms loses a present-versus-past contrast. An application can preserve both the original span and its lemma rather than deciding that one representation must replace the other.

A computational **token** is an input unit selected by a processing procedure. It might be a word, part of a word, punctuation or a character sequence. Its boundaries need not coincide with phonemes, morphemes or lexemes. Likewise, a Chinese character can participate in a multi-character word: the written sequence 相機 denotes a camera. A character-based system receives two units where a word-oriented analysis may use one. Neither count independently defines the number of meanings in the phrase.

2. Morphology: structure inside words

Plain version. Words have parts. Some parts carry vocabulary meaning, others carry grammar such as tense or number, and the way the parts are grouped can change the meaning.

Word structure can convey both vocabulary meaning and grammatical information. **Morphology** studies that structure; a **morpheme** is a minimal recurring pairing of form with meaning or grammatical function [7]. "Minimal" does not mean "one letter," and a meaningful element need not stand alone. Morphological evidence such as a plural ending, tense marker or derivational prefix is useful, but not every language expresses these properties with the same pieces or word boundaries.

Inflection, derivation and productive formation

An **inflectional** contrast changes a word's grammatical realization, as in repair/repaired. A **derivational** process can create a new vocabulary item, as in repair/repairable. A **compound** combines bases, as in camera mount [3, 7]. These are analytical distinctions: identifying an ending's spelling is not enough to decide its function.

Compare "repairs" in "She repairs the camera" and "The repairs took a day." In the first sentence, it is a finite verb form; in the second, a plural noun. A rule that strips every final "s" loses both grammatical category and the reason for the ending. It would also mishandle irregular pairs such as mouse/mice. The task is to explain recurring form–function patterns, with exceptions represented explicitly.

Productivity means that a pattern can support new combinations, rather than merely describing a memorized list. If a workshop introduces a new "sensor cradle," listeners can interpret the compound from familiar pieces and the setting. Yet composition does not specify every detail: the cradle might hold a sensor during assembly, transport or operation. Word formation can supply a general relation while context supplies its exact instance.

Chinese also has internal word structure and productive combinations. In 三台相機, 三 expresses the count, 台 is a classifier associated with the counted object, and 相機 names the object. English "three cameras" places plural marking on the noun in this example. The contrast identifies different grammatical choices; it does not establish that Chinese words lack structure or that English nouns always carry an explicit count.

Stemming versus lemmatization

Plain version. A stemmer chops endings off by rule; a lemmatizer looks up the dictionary form, using the word's grammatical category. The stemmer is cheap and crude; the lemmatizer needs more knowledge and makes fewer blunders.

A **stemmer** maps word forms to a shared string, the **stem**, by rewriting suffixes. Porter's algorithm is the classic English example: it applies ordered groups of suffix rules, each guarded by a condition on the remaining stem, and was designed to improve information retrieval by grouping related terms [16]. The stem need not be a real word; it only has to be consistent. A **lemmatizer** returns the lemma, which requires a lexicon and usually the part of speech: "saw" is the lemma of a noun but a form of SEE as a verb.

The toy stripper below uses four ordered rules and a minimum-stem-length guard, in the spirit of (but far simpler than) Porter's algorithm. The lookup table stands in for a dictionary.

```
# A toy suffix stripper (a few Porter-style rules) versus a lookup lemmatizer.
RULES = [("ies", "y"), ("ing", ""), ("ed", ""), ("s", "")]   # tried in order; first match
 ↪  wins

def stem(word):
    for suffix, replacement in RULES:
        if word.endswith(suffix) and len(word) - len(suffix) >= 3:
            return word[: -len(suffix)] + replacement
    return word

# A lemmatizer needs the part of speech; this tiny table stands in for a dictionary.
LEMMAS = {("repairs", "VERB"): "repair", ("repairs", "NOUN"): "repair",
          ("repaired", "VERB"): "repair", ("mice", "NOUN"): "mouse",
          ("was", "AUX"): "be", ("batteries", "NOUN"): "battery",
          ("news", "NOUN"): "news", ("this", "DET"): "this"}

def lemmatize(word, pos):
    return LEMMAS.get((word, pos), word)

for word, pos in [("repairs", "VERB"), ("repaired", "VERB"), ("batteries", "NOUN"),
                  ("mice", "NOUN"), ("was", "AUX"), ("news", "NOUN"), ("this", "DET")]:
    print(f"{word:10s} stem={stem(word):9s} lemma={lemmatize(word, pos)}")
```

The stripper handles the regular forms (`repairs`, `repaired`, `batteries` all reach a sensible stem) but returns `mice` and `was` unchanged and damages `news` → `new` and `this` → `thi`. The first two failures are irregular morphology; the last two are **over-stemming**, where an ending that only

looks like a suffix is removed. The lemmatizer gets all seven right, but only because its table already contains them; an unseen word passes through unchanged. Neither tool decides the grammatical category: the caller supplied `VERB` or `NOUN`, which is the job of the tagger in NLP 06.

Which one to use depends on the task. For search, merging `repairs` and `repaired` can raise recall even if a few unrelated words collide. For a status extractor, the tense the stemmer throws away may be exactly the answer. Modern neural pipelines rarely stem their input; they keep the original form and let subword tokens and context carry the morphology, which is one reason the next section matters.

The same pieces can combine in different orders

An invented description "an unsealable case" permits two useful structural analyses:

- `un + (seal + able)`: a case that cannot be sealed.
- `(un + seal) + able`: a case that can be unsealed.

The first treats "un" as negating an adjective; the second treats it as reversing the sealing action before "able" forms an adjective. The readings have opposite practical implications. Their availability and naturalness depend on context, so the example demonstrates possible analyses rather than claiming that every speaker assigns both equally often.

A bag of these pieces is identical in both analyses. What changes is the intermediate grouping and the kind of object each operation combines with. The same problem will reappear in sentence attachment and logical scope: a representation must retain structure if the distinction matters to the task.

Chinese segmentation is an interpretation problem

Written Chinese does not mark word boundaries with spaces. The original string 學生會說話 supports "學生 / 會 / 說話" ("students can speak") and "學生會 / 說話" ("the student council speaks"). The first reading treats 會 as expressing ability. The second treats 學生會 as a compound noun. Spaces inserted after processing therefore record an analysis; the unspaced string does not supply them directly.

The following program enumerates paths through a small, explicit lexicon. It is not a general Chinese segmenter. It shows why choosing the longest matching entry cannot establish the speaker's intended reading.

```
from functools import lru_cache

text = " 學生會說話"
lexicon = (" 學生", " 學生會", " 會", " 說話")

@lru_cache(None)
def segment(start):
    if start == len(text):
```

```
        return ((),)
    paths = []
    for word in lexicon:
        if text.startswith(word, start):
            for rest in segment(start + len(word)):
                paths.append((word,) + rest)
    return tuple(paths)

for path in segment(0):
    print(" / ".join(path))
assert len(segment(0)) == 2
```

The two printed paths are the readings above. Every recursive step consumes a known word, and the base case accepts only complete coverage. Enlarging the lexicon may add alternatives; it does not add evidence about a particular conversation. A syntax analysis, the surrounding sentence or knowledge that the message concerns a council meeting could favor one path.

Even the gold standard is a choice. The Second International Chinese Word Segmentation Bakeoff evaluated systems on four corpora, two in Traditional characters (from Academia Sinica in Taiwan and City University of Hong Kong) and two in Simplified characters (from Peking University and Microsoft Research), and each provider segmented its data under its own standard [27]. A system can therefore be "wrong" on one corpus and "right" on another for the same split. Report which segmentation standard your labels follow.

3. Tokens and subwords

Plain version. A model needs a fixed list of units. Whole words leave too many unknown words; single characters make sequences long. Subword methods learn a middle ground from data: frequent strings become single tokens, rare ones are spelled out from smaller pieces.

NLP 01 defined the unknown-token rate and showed why mapping every unseen word to one marker loses distinctions. A character vocabulary reduces whole-word unknowns if the necessary characters are covered, at the cost of longer sequences. A byte-based encoding can cover arbitrary input bytes with a complete base byte alphabet, but individual units then need not correspond to readable characters. Subword methods trade vocabulary size against sequence length; Mielke and colleagues survey these choices [22]. There is no universal segmentation that is best for every language, task and resource budget.

A learned tokenizer typically fits statistical patterns in a corpus. These patterns can help reuse frequent pieces, but a frequent piece is not necessarily a meaningful morpheme. Conversely, a meaningful inflection can be split across several pieces. This distinction matters when asking what the model can recognize: tokenization supplies an encoding; later computation must learn how the encoded parts combine.

Work through one subword method: BPE

BPE learns which adjacent pieces are worth joining. It gives frequent combinations their own symbols while leaving other strings as sequences of smaller pieces. Sennrich, Haddow and Birch adapted byte-pair encoding for rare-word translation [17]. The character-based teaching version below works inside words:

1. Represent each training word as characters plus an end marker.
2. Count adjacent symbol pairs, weighted by the word's corpus frequency.
3. Merge the most frequent pair everywhere it appears in those word representations.
4. Save that merge and repeat for a fixed number of steps.

A merge changes subsequent pair counts. After combining two symbols, the new symbol can participate in another pair. The algorithm therefore learns an ordered sequence of operations, rather than a set of pieces that can be applied in any order.

```
from collections import Counter

corpus = ("low low low low low lower lower newest newest newest newest newest "
          "newest widest widest widest").split()
# Start from characters; "_" marks the end of a word so merges respect boundaries.
words = Counter(tuple(w) + ("_",) for w in corpus)

def pair_counts(words):
    pairs = Counter()
    for symbols, freq in words.items():
        for a, b in zip(symbols, symbols[1:]):
            pairs[a, b] += freq
    return pairs

def merge(words, pair):
    out = Counter()
    for symbols, freq in words.items():
        s, i = [], 0
        while i < len(symbols):
            if i + 1 < len(symbols) and (symbols[i], symbols[i + 1]) == pair:
                s.append(symbols[i] + symbols[i + 1]); i += 2
            else:
                s.append(symbols[i]); i += 1
        out[tuple(s)] += freq
    return out

merges = []
for step in range(10):
    pairs = pair_counts(words)
    best = max(pairs, key=lambda p: (pairs[p], p))   # most frequent; ties broken by symbol
    ↪ order
    merges.append(best)
    words = merge(words, best)
print("merges:", [a + b for a, b in merges])

def tokenize(word):                  # apply the learned merges in order
```

```
    symbols = list(word) + ["_"]
    for pair in merges:
        symbols = list(merge(Counter({tuple(symbols): 1}), pair))[0]
    return symbols

for w in ["lowest", "newer", "wider"]:
    print(w, "->", tokenize(w))
```

The output includes `lowest -> ('low', 'est_')`, `newer -> ('ne', 'w', 'e', 'r', '_')` and `wider -> ('wi', 'd', 'e', 'r', '_')`. The first word was not in the training corpus, yet its letters can be recombined using learned pieces. The longer encodings of the other words reflect which combinations acquired merges.

Why the merge sequence matters

The first merge is `t_`, followed by `st_` and `est_`. Repeated endings in the training words make these combinations frequent. This resembles an English suffix in the example, but the algorithm has no instruction to discover suffixes: frequency drives its decisions. Another corpus can produce a different split of the same spelling.

The merge `low` occurs only after smaller combinations have been formed. Encoding follows the saved order so it reproduces the fitted procedure. A longest-piece heuristic over the final symbols is a different algorithm and need not produce the same segmentation. The tie-breaking rule also matters: this code resolves equal counts through symbol order, making the run reproducible.

The function returns pieces, not numerical IDs. It does not prove coverage for arbitrary unseen characters. The example's successful novel words reuse characters present during training; a word containing a new character would require a declared fallback and a compatible vocabulary lookup. This separates two questions: can a string be segmented, and does every returned piece have an ID that the model understands?

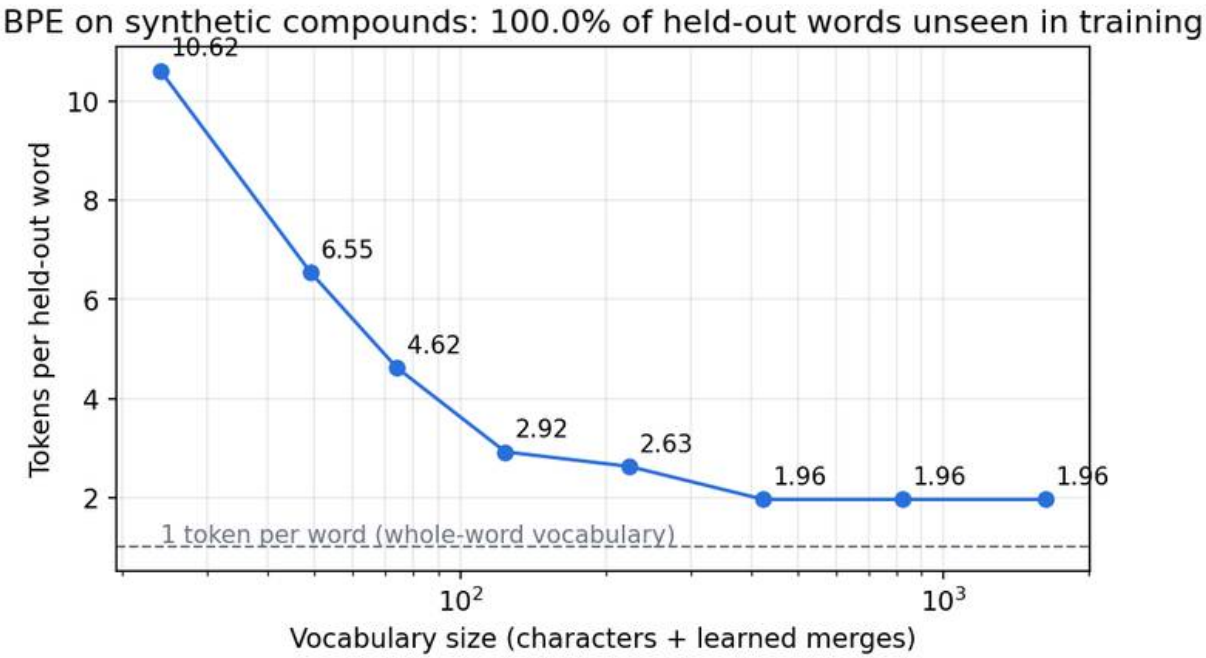


Figure 1 — More merges can shorten encodings, but vocabulary size alone does not determine coverage or linguistic quality. The synthetic training and held-out samples are separate.

WordPiece, the unigram model and SentencePiece

BPE is one member of a family. The others differ in how they choose pieces and how they split a new word.

- **WordPiece** originated in Japanese and Korean voice search [18]. Google's neural translation system describes its version as choosing a fixed number of pieces so that the segmented training corpus uses as few pieces as possible, with a boundary marker added at the start of each word so that the original word sequence can be recovered without ambiguity [19]. BERT uses a WordPiece vocabulary [30]; NLP 10 builds on it.
- **Unigram language model** segmentation works top-down: start from a large candidate inventory, model a word as a sequence of independently drawn pieces, and prune the pieces that contribute least to the corpus likelihood [20]. Because it is probabilistic, one word can have several scored segmentations; Kudo used this to sample alternative segmentations during training as a regularizer [20].
- **SentencePiece** is a toolkit rather than a new criterion: it trains BPE or unigram models directly from raw sentences and treats the space as an ordinary symbol, so no language-specific pre-tokenizer is required and decoding reverses encoding [21]. That matters for languages such as Chinese and Japanese that do not separate words with spaces.

In all three, the learned pieces are statistical, not morphological. A subword boundary that happens to fall at `est` is evidence about corpus frequency; it is not an analysis that `-est` is a superlative

suffix.

Use a complete tokenizer, not just a merge list

Tokenization software also specifies normalization, pre-tokenization, special markers and decoding [23]. A checkpoint's token IDs refer to its particular vocabulary and conventions. Matching only the tokenizer family name is insufficient: two vocabularies can assign different symbols to the same ID.

For a pretrained model, use the associated tokenizer and configuration. For a tokenizer trained in your experiment, fit it on the permitted training text and freeze its vocabulary and merge rules before evaluation. Inspect examples as pieces, IDs and reconstructed text; each view catches a different failure.

Online only: BPE — Neural Machine Translation of Rare Words with Subword Units — https://blog.shyandram.dev/en/learn/nlp/linguistic-essentials/papers/bpe–neural-machine-translation-of-rare-words-with-subword-units/

4. Parts of speech

Plain version. A word's category describes how it behaves in a sentence. Its spelling can suggest a category, but context often decides which use is intended.

Part of speech describes a grammatical category, such as noun or verb. A noun commonly heads a noun phrase; a verb commonly heads a clause's predicate. These are distributional descriptions rather than rules that every noun must name a physical object. "Decision" and "arrival" are nouns although they refer to abstract content and events [2].

The same surface word can have several categories. In the invented pair "the pilots can land" and "the metal can fell," the word "can" has different roles. The second sentence also demonstrates why a familiar local phrase can be misleading when the remainder of the sentence changes. A tagger must consider the permitted context and its annotation scheme.

The Universal Dependencies tag set

Tag inventories differ between projects, which makes cross-language comparison hard. **Universal Dependencies** (UD) is a framework for consistent cross-linguistic annotation of parts of speech, morphological features and dependency relations [24]. Its universal part-of-speech (UPOS) inventory has 17 tags [25]:

Group	Tags
Open class	`ADJ` adjective, `ADV` adverb, `INTJ` interjection, `NOUN` noun, `PROPN` proper noun, `VERB` verb

Group	Tags
Closed class	`ADP` adposition, `AUX` auxiliary, `CCONJ` coordinating conjunction, `DET` determiner, `NUM` numeral, `PART` particle, `PRON` pronoun, `SCONJ` subordinating conjunction
Other	`PUNCT` punctuation, `SYM` symbol, `X` other

UD separates the coarse tag from **features** such as `Number=Plur` or `Tense=Past`, so "repairs" as a noun can be `NOUN` with `Number=Plur` while "repaired" is `VERB` with `Tense=Past`. Each language's treebanks add documented conventions for constructions the universal guidelines leave open. The tag set is a shared annotation scheme, not a claim that every language draws category boundaries in the same place.

Represent a tagged sentence as:

$$\mathbf{x} = (x_1, \dots, x_T), \qquad \mathbf{y} = (y_1, \dots, y_T), \qquad y_t \in \mathcal{Y}.$$

Here $\mathbf{x}$ is the input word sequence, x_t is the word at position t, T is sequence length, $\mathbf{y}$ is the tag sequence, y_t is the tag for word x_t and $\mathcal{Y}$ is the finite tag inventory, for example the 17 UPOS tags. The symbols t and T refer to linguistic word positions, not automatically to model subword positions.

Annotation needs additional rules: contractions, punctuation, multiword expressions and words with several plausible analyses. When a model uses subwords, define how its outputs return to words. Using only the first subword's representation is one possible convention; pooling all subwords is another. Whatever the choice, gold labels and decoded output must use the same units. Predicting $\mathbf{y}$ from $\mathbf{x}$, with hidden Markov models, Viterbi decoding and recurrent taggers, is the subject of NLP 06.

5. Syntax: order, grouping and grammatical relations

Plain version. Grammar organizes words into groups and links them by relations such as subject and object. The same words in a different structure can describe a different event.

Grammar organizes combinations. Word order, phrase structure and relations such as subject and object constrain who did what to whom [2, 3]. These constraints are richer than a list of adjacent words.

Why word counts lose role information

"Ada praised Bo" and "Bo praised Ada" contain the same words. Their ordinary active-clause readings reverse the person giving praise and the person receiving it. The Chinese pair 阿達稱讚小博 / 小博稱讚阿達 makes the same contrast in these sentences. A representation that only counts words maps each pair to the same object.

```
from collections import Counter
```

```
first = "Ada praised Bo".split()
second = "Bo praised Ada".split()

def roles(tokens):
    # Only our deliberately fixed three-word active-clause template.
    assert len(tokens) == 3 and tokens[1] == "praised"
    return {"praiser": tokens[0], "praised_person": tokens[2]}

print(Counter(first) == Counter(second))
print(roles(first))
print(roles(second))
assert roles(first) != roles(second)
```

The count comparison prints `True`; the role dictionaries differ. This is an information loss result, not a trained-model benchmark. Once the two inputs have become identical counts, a later deterministic function of those counts alone cannot recover which original order occurred.

Syntactic roles also differ from semantic roles. In "Bo was praised by Ada," Bo is the grammatical subject, while Ada still performs the praising. Equating the subject with the actor would fail even though the passive sentence describes the same event. An extraction task should state whether it wants grammatical dependencies, event participants or both.

Constituency and dependency describe different aspects

A **constituent** is a unit that participates in larger structure. In "The label beside the boxes is missing," the noun phrase "the label beside the boxes" functions as the subject. Its head is "label"; "beside the boxes" modifies that head. Replacing the whole phrase with "it" preserves a simple grammatical frame: "It is missing." Such tests provide evidence for grouping, rather than proving that every adjacent substring is a phrase [8].

The agreement contrast "The labels beside the box are missing" shows why proximity is insufficient. The verb agrees with the subject's head in these examples, not the nearest noun "box." A local heuristic could find the closest noun correctly and still select the wrong number.

A **dependency** representation instead records relations between words, such as a verb's subject and object or a noun's modifier [2]. UD uses dependency trees in which every word has exactly one head and a labeled relation to it, such as `nsubj` for a nominal subject [24]. Constituency highlights grouping; dependency highlights linked heads. They can describe related evidence without having identical structures or annotation conventions. The Penn Treebank, a large early English resource, annotated constituency trees [26]; UD annotates dependencies across many languages [24].

In the panel example, "with the camera" can modify the inspecting event or the panel. The simplified diagrams below retain this one attachment distinction and omit determiners and other relations. They are teaching diagrams, not complete parses under a named annotation standard.

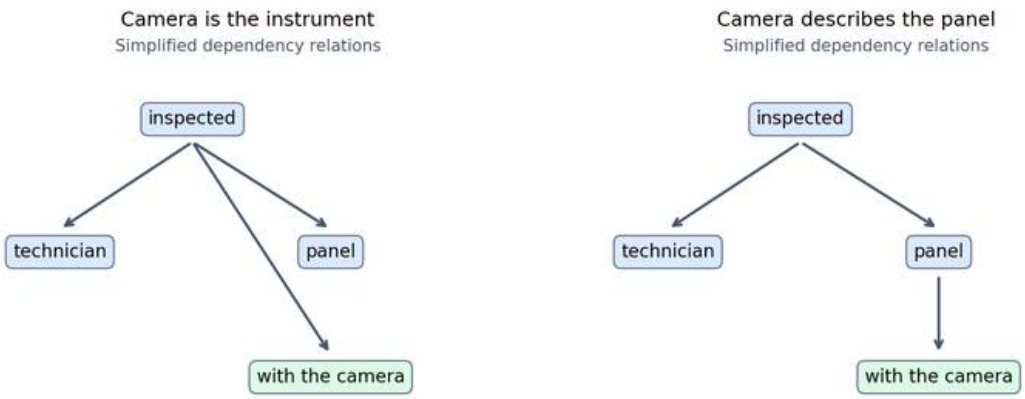


Figure 2 — The same words permit different attachment relations. Selecting a relation selects a different interpretation.

Under event attachment, a camera is the instrument of inspection. Under noun attachment, the camera helps identify the panel. Surrounding context can favor either: a technician explaining a recording method versus someone distinguishing equipped and unequipped panels. Grammatical possibility does not determine which situation actually occurred.

This is a preview. NLP 12 develops grammars, chart parsing, probabilistic parsing and dependency parsing. Here the essential question is what a representation must preserve: grouping, linked heads, role direction and plausible alternatives.

6. Semantics: meanings and truth conditions

Plain version. Semantics asks what an expression means and what the world would have to be like for a sentence to be true. Word meanings matter, but so does how the sentence combines them.

A sentence can be fluent while describing a situation incorrectly. **Semantics** asks how expressions contribute meaning and how a structured sentence determines conditions under which its assertion is true [2, 3]. Separating these conditions from conversational inference makes ambiguity easier to analyze.

Lexical meaning is contextual and relational

Polysemy concerns related senses of an expression; **homonymy** concerns distinct meanings sharing a form [3]. A window can be an opening in a wall or a bounded area on a screen. The relationship helps explain the extension, but replacing "window" with one fixed concrete object would miss the screen reading. A dictionary can list senses; the occurrence still needs interpretation. NLP 04 studies how vectors learned from contexts capture some of these relations.

Chinese 開 appears in 開門 (“open a door”) and 開會 (“hold a meeting”). Translating every occurrence with the same English verb would erase the construction-specific difference. Meaning belongs to the expression in its combination and context, not to a context-free lookup from character to English word.

Even apparently simple adjectives can rely on a comparison class. A small elephant may be larger than a large ant. The expressions compare each animal with an appropriate class, not with one universal size threshold. In a collection containing only toy elephants, the relevant comparison can change again. An annotation called “small” should therefore specify what is being compared.

Composition depends on structure

Compositionality is the idea that the meaning of a complex expression depends on meanings of its parts and how they combine [3]. It does not say that word meanings can simply be added together. “Bo praised Ada” combines participants in a directed relation; a reordered sentence combines the same names differently. Attachment changes which expression a modifier restricts.

An idiomatic expression can also have a conventional meaning not predictable from isolated word entries. A lexicon and a grammar are complementary: the analysis must record both conventional combinations and productive composition. For an application, ask which distinction is needed. Searching for repair instructions may tolerate broad lexical grouping; deciding whether a repair was completed requires event, tense and negation information.

Quantifier scope changes what is required

“Every technician uses a camera” has a reading in which different technicians may use different cameras. It can also be understood, in a suitable context, as involving one camera shared by all. These are different conditions:

$$A : \forall t \in T,\ \exists c \in C :\ U(t, c),$$
$$B : \exists c \in C,\ \forall t \in T :\ U(t, c).$$

Here T is the set of technicians, C the set of cameras, $U(t, c)$ means technician t uses camera c, A and B name the two readings, $\forall$ means “for every,” and $\exists$ means “there exists.” Both sets are nonempty in this example. The variable c can depend on t in A; in B one c must work for every t.

To check A, visit each technician and find at least one camera they use. To check B, choose a camera first and verify that all technicians use that particular camera. If B holds, A follows: use the shared camera for each person. The reverse implication fails when every person has a camera but no camera is shared by everyone.

```
technicians = ("Ada", "Bo", "Cy")
cameras = ("A", "B")
worlds = {
    "distributed": {("Ada", "A"), ("Bo", "B"), ("Cy", "A")},
    "shared": {("Ada", "A"), ("Bo", "A"), ("Cy", "A")},
}
```

```
for name, uses in worlds.items():
    each_has_some = all(any((t, c) in uses for c in cameras)
                        for t in technicians)
    some_for_everyone = any(all((t, c) in uses for t in technicians)
                            for c in cameras)
    print(name, each_has_some, some_for_everyone)
assert not any(all((t, c) in worlds["distributed"] for t in technicians)
               for c in cameras)
```

The outputs are `distributed True False` and `shared True True`. Each row in Figure 3 represents one person's possibilities. The first reading requires a marked cell in every row; the second requires an entirely marked column. Scope is therefore a constraint difference, not merely a different paraphrase of the same requirement.

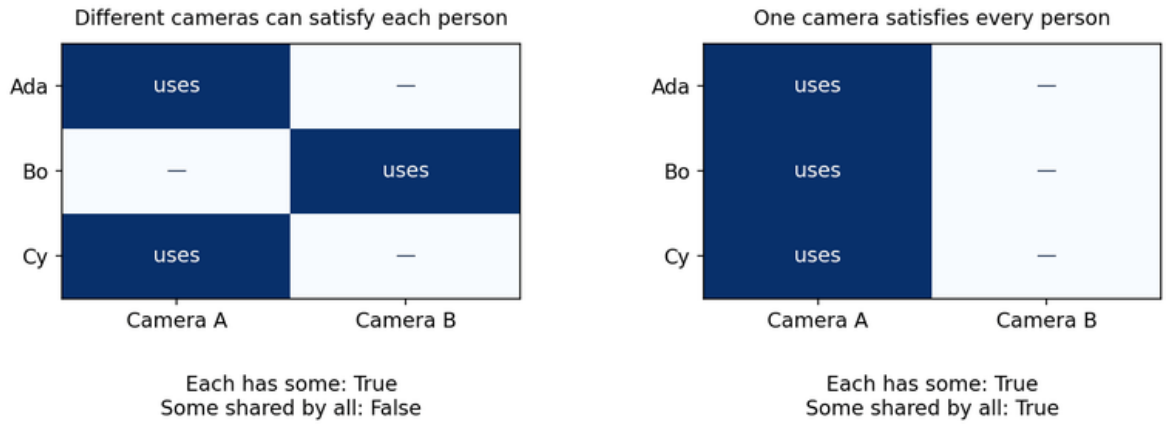


Figure 3 — Changing quantifier order changes which worlds satisfy the sentence.

Negation and scope must also be kept apart

"Not every technician approved" allows some approvals and at least one non-approval. "Every technician did not approve," under a reading with negation inside the universal quantifier, says that nobody approved. Written English can itself leave scope unclear; explicit paraphrases remove the ambiguity.

The Chinese contrast 不是每位技師都批准 / 每位技師都不批准 makes the intended distinction easier to display. The former negates "all approved"; the latter assigns non-approval to each technician. For a nonempty technician set:

$$\neg\forall t \in T\ P(t) \ \equiv\ \exists t \in T\ \neg P(t), \qquad \forall t \in T\ \neg P(t) \ \equiv\ \neg\exists t \in T\ P(t).$$

Here T is the technician set, $P(t)$ means t approved, $\neg$ is negation, $\forall$ and $\exists$ are universal and existential quantification, and $\equiv$ means logical equivalence. The first condition needs one coun-

terexample to universal approval; the second excludes every approval. If one technician approves and another does not, only the first condition is true.

NLP 12 develops predicate–argument structure and semantic role labeling, and NLP 11 tests inference between sentences. Before designing a field such as `approved`, decide whose approval, what event and which quantified claim that field represents.

7. Pragmatics: interpretation in a situation

Plain version. What a speaker means can go beyond what the sentence literally says. Pragmatics studies how the situation fills that gap.

Speakers often communicate more than their sentence literally asserts. **Pragmatics** studies language use in context, including reference, background assumptions and communicative actions [9, 10]. Context supplies evidence; it does not license arbitrary interpretations.

Reference and the speaker's viewpoint

"I will check that here tomorrow" is structurally simple but under-specified outside its setting. "I" identifies the speaker, "that" selects a referent, "here" uses a spatial viewpoint, and "tomorrow" uses a temporal viewpoint. Such context-dependent expressions involve **deixis** [10]. A forwarded message should not silently replace the original speaker's time and place with the receiver's.

Reference also depends on discourse. "Mei put a camera beside a lamp. It was overheating." Either object may be the referent of "it" without further evidence. Knowing that cameras and lamps can both become hot leaves the ambiguity intact. In "Mei turned off the lamp. It stopped flickering," the described event supplies evidence favoring the lamp. Evidence may favor a candidate without turning a grammatical possibility into a universal rule.

A practical analysis records the expression, candidate referents and supporting context. An unresolved referent is a meaningful result when the input does not determine one. Coreference resolution across sentences is developed in NLP 12, and tracking entities across dialogue turns in NLP 08.

Assertion, presupposition and implicature

An **assertion** contributes the claim at issue. A **presupposition** presents a proposition as background rather than as the main update. A **conversational implicature** is an additional inference drawn from the utterance and its use, which can often be cancelled without contradicting the literal assertion [9, 10]. They should not collapse into one "meaning" label.

"Mei stopped checking the sensor" asserts a cessation and normally backgrounds that Mei checked it previously. "Mei did not stop checking the sensor" still typically takes earlier checking for granted. That persistence under negation helps diagnose the background contribution. These

are ordinary interpretations, not a proof that the earlier event actually occurred. A listener can challenge it: "She never checked it in the first place."

"Some panels passed" asserts that at least one panel passed in the relevant collection. In a conversation asking whether the whole collection passed, a listener may infer "not all." But "Some panels passed; in fact, all of them did" can cancel that inference without denying the first assertion [9]. By contrast, "All panels passed; in fact, one failed" conflicts under the same panel collection, test and time.

This difference can be checked without a language model. Imagine three panels: one, two and three passing panels all satisfy the literal "some." A "some but not all" interpretation retains only the first two possibilities. The inference narrows the possibilities because of a conversational expectation; that narrowing is not contributed by existential meaning alone. Whether it is justified depends on the question, speaker knowledge and manner of reporting.

A grammatical question can perform a request

"Can you lower the gain?" has the form of an ability question. During a recording session, it can perform a request to change a setting; during an assessment, it can ask whether the trainee knows how [10]. A **speech act** is the communicative action performed through an utterance, such as requesting, promising, asserting or asking.

"The gain is too high" can likewise function as a report or an indirect request. The physical words do not themselves specify every interpersonal condition. Who controls the equipment? What was the preceding question? Is the speaker diagnosing the recording or coordinating an adjustment? These details support an interpretation of the action.

A useful annotation separates sentence form, asserted content and intended act. Otherwise, every question-shaped request may be labeled as information seeking, or every unfavorable description may be labeled as an instruction. The goal is a defensible reading of this interaction, including alternatives when the setting is missing.

8. Discourse: meaning across sentences and turns

Plain version. Sentences in a text or a conversation lean on each other. Later sentences refer back, contrast, explain or correct, and a reader keeps track of that shared state.

An utterance can introduce an entity, correct an earlier claim, provide a reason or contrast with an expectation. **Discourse** analysis considers these connections across sentences and conversational turns [2, 10]. Reading each sentence independently discards the shared state that makes later expressions interpretable.

"Mei told Lin that she would calibrate the camera" can refer to either person as the future calibrator. The Chinese 美玲告訴怡君，她會校正相機 permits a comparable ambiguity in this invented setting. A preceding statement "Lin had already volunteered" favors Lin; "Mei was the only available technician" favors Mei. Neither grammatical gender nor mention order alone settles both contexts.

Now compare “The sensor became hot. Its readings stayed stable” with “The sensor became hot, but its readings stayed stable.” The second explicitly marks a contrast with an expectation that heat would disrupt readings. The assertion about stability can be the same while the discourse contribution differs. Conversely, “The readings drifted because the sensor became hot” introduces a causal explanation, not merely two events in sequence.

Conversations also contain repairs. “Move it to Friday.” “The inspection?” “Yes, the inspection.” The middle turn does not advance the scheduling date; it resolves the object. Once resolved, later references can use the shared identification. A dialogue system that summarizes only the first and last words misses why the clarification changed the conversational state.

This suggests a practical representation with distinct records for introduced entities, asserted events, unresolved references and conversational questions. Such a representation can remain partial. A one-sentence input may leave the speaker unknown; an entire exchange may resolve the speaker but still leave the exact camera unspecified. Increasing context length only helps when relevant evidence is actually present.

9. Corpora, annotation and Zipf’s law

Plain version. NLP learns from collections of real text, often with human labels added. How the text was collected and how consistently it was labeled limit what any model can learn from it, and every corpus has a long tail of rare words.

A **corpus** is a collection of text or transcribed speech gathered under stated decisions: which sources, which period, which speakers, which language varieties. An **annotated corpus** adds analyses such as segmentation, tags, trees, entity spans or sentiment labels, produced under written guidelines. Two widely used examples show the range. The Penn Treebank annotated English text with part-of-speech tags and constituency trees [26]. The Universal Dependencies project releases treebanks for many languages under one cross-lingual dependency scheme [24]. The segmentation bakeoff corpora above show that even word boundaries come with a standard attached [27].

Annotation guidelines and agreement

A label is only as clear as the guideline behind it. Two careful annotators can disagree because a sentence is genuinely ambiguous, because the guideline is silent, or because one of them made an error. Measuring agreement on a shared sample separates these cases before the labels are used for training.

Raw percent agreement overstates reliability when one label is very common, because annotators would agree often even by guessing. Cohen’s kappa corrects for that [28]:

$$\kappa = \frac{p_o - p_e}{1 - p_e}, \qquad p_e = \sum_{k \in \mathcal{Y}} \frac{n_{A,k}}{n} \cdot \frac{n_{B,k}}{n}.$$

Here p_o is the observed fraction of items on which annotators A and B agree; p_e is the agreement expected if each annotator chose labels independently at their own observed rates; $\mathcal{Y}$ is the label set; $n_{A,k}$ and $n_{B,k}$ count how often each annotator used label k; and n is the number of items. $\kappa = 1$ is perfect agreement and $\kappa = 0$ is agreement at the chance level; the formula requires $p_e < 1$.

```
from collections import Counter
from sklearn.metrics import cohen_kappa_score

words = "Mei has to repair the camera mount before the inspection".split()
a = "PROPN VERB PART VERB DET NOUN NOUN ADP DET NOUN".split()   # annotator A
b = "PROPN AUX ADP VERB DET NOUN NOUN ADP DET NOUN".split()     # annotator B
n = len(words)
p_o = sum(x == y for x, y in zip(a, b)) / n                     # observed agreement
ca, cb = Counter(a), Counter(b)
p_e = sum(ca[k] * cb[k] for k in ca) / n**2                     # agreement expected by
 ↪  chance
kappa = (p_o - p_e) / (1 - p_e)
print(f"p_o={p_o:.3f}  p_e={p_e:.3f}  kappa={kappa:.3f}")
assert abs(kappa - cohen_kappa_score(a, b)) < 1e-12
print("disagreements:", [(w, x, y) for w, x, y in zip(words, a, b) if x != y])
```

It prints `p_o=0.800  p_e=0.180  kappa=0.756`, and the two disagreements are `has` (`VERB` versus `AUX`) and `to` (`PART` versus `ADP`). Neither is a careless slip: "has to" sits on the border between a main verb and an auxiliary, and infinitival "to" resembles a preposition. The useful outcome is a guideline decision for that construction, not a higher number. Kappa summarizes reliability; the disagreement list says what to fix.

Zipf's law: a few words are very common, most are rare

Count every word in a corpus and sort the types by frequency. A handful of function words such as "the" and "of" account for a large share of all tokens, while a huge number of types occur only once or twice. **Zipf's law** describes the relation between a word's frequency rank and its count as approximately a power law [29]:

$$f(r) \approx \frac{C}{r^s}, \qquad \log f(r) \approx \log C - s \log r.$$

Here $r = 1, 2, \ldots$ is a type's rank when types are sorted by decreasing count, $f(r)$ is the count of the type at rank r, $C > 0$ is a scale constant and $s > 0$ is the exponent. With $s \approx 1$, the product $r \cdot f(r)$ stays roughly constant. On log–log axes the relation is a straight line with slope $-s$. Piantadosi's review shows that the law holds approximately across many languages and corpora, that it deviates systematically at the extremes, and that its explanation is still debated [29].

We can check it on real English text that is already on disk: the docstrings of the Python standard library installed in the project environment (Python 3.9).

```
import ast, re, sysconfig
```

```
from collections import Counter
from pathlib import Path

lib = Path(sysconfig.get_paths()["stdlib"])        # text already on disk: stdlib docstrings
docs = []
for path in sorted(lib.glob("*.py")):
    try:
        tree = ast.parse(path.read_text(encoding="utf-8"))
    except (SyntaxError, UnicodeDecodeError):
        continue
    docs += [ast.get_docstring(n) or "" for n in ast.walk(tree)
             if isinstance(n, (ast.Module, ast.FunctionDef, ast.AsyncFunctionDef,
              ↪  ast.ClassDef))]
tokens = [w for d in docs for w in re.findall(r"[a-z]+", d.lower())]
counts = Counter(tokens)
print(len(tokens), "tokens,", len(counts), "types")
ranked = counts.most_common()
for r in (1, 2, 10, 100, 1000):
    word, f = ranked[r - 1]
    print(f"rank {r:4d}  {word:10s} f={f:5d}  r*f={r * f}")
hapax = sum(f == 1 for f in counts.values())
print(f"types seen once: {hapax} ({hapax / len(counts):.0%})")
```

With Python 3.9.7 it finds 134,620 tokens and 6,281 types. The product $r \cdot f(r)$ is 8,313 at rank 1 (“the”), 13,640 at rank 10 (“be”), 20,100 at rank 100 (“called”) and 18,000 at rank 1,000: not constant, but within a factor of about 2.5 while the rank changes by a factor of 1,000. A least-squares fit over ranks 10 to 1,000 in the figure script gives $s \approx 1.00$. A third of all types, 2,003 of them (32%), occur exactly once. A different Python version ships different docstrings, so the exact counts can change.

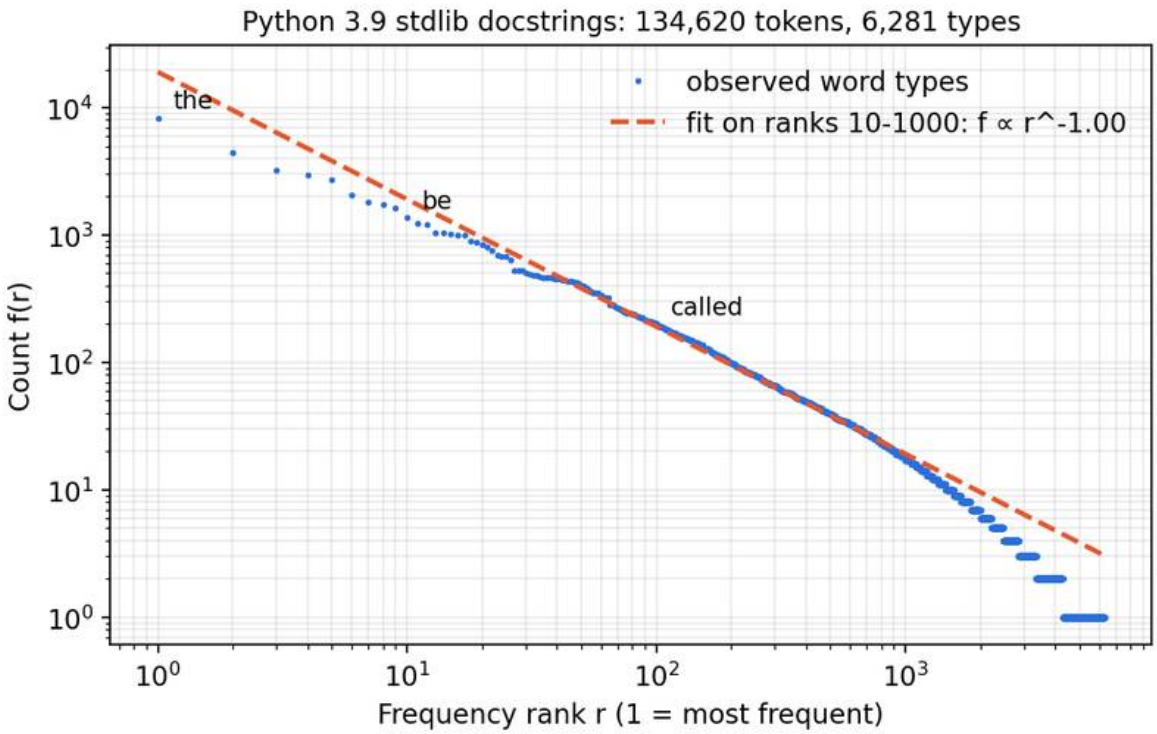


Figure 4 — Rank–frequency plot of the Python 3.9 standard-library docstrings. The dashed line is a power-law fit over ranks 10–1,000; the head and the tail of single-occurrence words bend away from it.

Three consequences follow for NLP. First, any fixed word vocabulary leaves out part of the tail, so new text keeps producing unseen words; this is why subword vocabularies exist. Second, count-based probability estimates are reliable for frequent words and unreliable for the rare majority, which is why smoothing appears in n-gram models (NLP 05). Third, accuracy on all tokens is dominated by frequent words; an evaluation that matters for rare names or technical terms must measure them separately.

10. Linguistic diversity and model assumptions

Plain version. Languages package information differently, and communities speak and write in different varieties. A model built around one language's habits can miss what another language says.

Languages and communities distribute information differently across morphology, order, prosody and context. **Typology** compares such structural patterns. **Dialect** concerns a community's variety; **register** concerns usage associated with a setting or activity [3, 11]. These distinctions describe variation rather than rank ways of speaking.

Translation does not preserve every structure

Compare “I calibrated this camera yesterday” with 這台相機，我昨天校正過. The Chinese example begins with the camera as the topic and then comments on it. Treating the first noun phrase as the actor would misidentify the calibrator. An English translation can preserve the event while changing how the message organizes old and new information.

The short Chinese 已經校正好了 can be informative when the camera and responsible person are already established. A word-for-word translation that demands overt counterparts for every English argument may make it look incomplete. In context it can answer a pending question directly. This does not mean that any omitted participant is always recoverable; the analysis must examine the actual discourse.

Similarly, 三台相機 and “three cameras” package number differently. A feature called “plural suffix” cannot be required universally merely because it suits one language. Structural categories may be useful for comparison while needing language-specific realizations and annotation decisions. A language using fewer obligatory endings can still express detailed distinctions through constructions, words and context.

Variety, setting and mixed-language expression

A formal maintenance report and a chat between technicians can describe the same equipment with different vocabulary, abbreviation and explicitness. The report “The shutter failed to actuate” and the informal “The shutter didn’t fire” may describe a related event in context, yet the wording provides different evidence about setting and intended readership. Unfamiliar wording is not automatically an error.

A speaker might write 把 exposure 調低一點 in a bilingual discussion. The English item occupies a position in an otherwise Chinese construction. **Code-switching** involves alternation between linguistic varieties within an interaction [14]; analyzing this example requires both the lexical choice and the surrounding grammar. One occurrence alone may not settle whether a community treats the item as switching or as an established borrowing.

An analysis should preserve the observed expression before standardizing it. Replacing a regional or technical expression with a presumed standard can help a particular search task, but may erase a distinction needed for another task. A corpus records language from particular speakers, settings and collection decisions; frequent corpus patterns are evidence of usage, not a universal inventory of everything speakers can say.

11. A complete worked reading

Put the levels together with an invented message:

> 小林說：「那台相機又過熱了，你明天能幫我看看嗎？」
>
> Lin said: “That camera overheated again. Can you take a look for me tomorrow?”

The translation helps an English reader but already chooses wording for 看看 and punctuation for the quoted interaction. Analyze the original before treating the translation as equivalent evidence.

Level	What the message contributes	What remains to be established
Written form and segmentation	The quotation groups Lin's utterance; 相機 is a camera expression	Speech intonation and gesture are absent
Morphology and construction	看看 is a reduplicated form in this request, not a counted pair of inspections	Exact activity and expected effort depend on the setting
Syntax	小林 introduces the reported speaker; 你 is the addressee inside the quotation	The reporter need not be the addressee
Semantics	The camera is described as overheating; a future assistance question is expressed	Which camera, and whether the description is factually correct
Background and reference	又 normally invokes a prior occurrence; 那台 selects a contextually identified camera	Prior overheating and the demonstrative referent require evidence
Pragmatics and discourse	The question naturally supports a request in a maintenance conversation	Whether that reading fits this interaction and whether it is accepted

The quotation changes the reference frame. "我" refers to Lin as the quoted speaker. "你" refers to Lin's addressee, who may differ from the current reader. "明天" is relative to the quoted utterance's day. If the report arrives later, its arrival date does not redefine the requested day.

The ordinary reading of 看看 presents the activity as taking a look, often with a light or tentative framing in this construction [13]. It does not establish a precise inspection duration or a numerical repetition count. Translating it as "inspect twice" would confuse visible repetition with event counting.

Finally, separate linguistic evidence from factual evidence. The message provides evidence that Lin reported overheating and requested help. It does not by itself verify overheating, identify a serial number or show that anyone agreed to help. A structured output might record the speaker and reported event while retaining unresolved referents and dates. This is a more informative analysis than assigning a single topic label such as "camera repair."

Modern view

Recent surveys show why language analysis remains necessary for large language models. Ma and colleagues' 2025 survey organizes pragmatic evaluation by phenomena including reference and deixis, implicature and presupposition, speech acts, discourse and social pragmatics [10]. It compares task design, collection and evaluation, and identifies gaps between available tests and contextual language use. Its useful lesson here is to evaluate the particular inference required, rather than treating a broad language score as proof of all pragmatic abilities.

Mielke and colleagues' tokenization review organizes the open-vocabulary problem across words, characters and intermediate units [22]. Its useful question is how segmentation balances coverage,

sequence length and reuse. These quantities remain distinct: a tokenizer can represent every input string while giving a model little training evidence for the relevant combinations. Zipf's law explains why the problem never goes away: however large the vocabulary, the tail keeps supplying rare forms [29].

Qin and colleagues' 2025 survey separates multilingual alignment through parameter tuning from alignment with frozen parameters, including prompting and translation-based approaches [12]. It reviews cross-language training, evaluation and challenges. This distinction identifies where a system's behavior is being adapted, but multilingual output alone does not establish that it preserves a construction's scope, register or conversational force.

Winata and colleagues' 2023 systematic survey groups code-switching research by languages, tasks and research trends [15]. Read it alongside Doğruöz and colleagues' account of structural and social patterns [14]: testing on text assembled from two monolingual sources is a different task from interpreting how bilingual speakers actually switch during an interaction. The exposure example in Section 10 makes that distinction concrete: the surrounding construction and conversational setting contribute information beyond identifying the English item.

Linguistic analysis also continues to refine apparently simple constructions. Lu and Müller's 2026 study analyzes Mandarin verb reduplication and its interaction with verb classes and aspect [13]. For this tutorial, the relevant observation is that limited-event meaning and context-dependent pragmatic effects deserve separate analysis. Visible repetition is not a general instruction to count events twice.

A concrete evaluation can vary one property while holding others fixed. Reverse Ada and Bo to test role direction; change camera attachment while preserving words; compare "not all" with "none"; move a quoted message into a later reporting context. Write the expected interpretations and acceptable alternatives first. Then a failure reveals which distinction was lost. These examples are small diagnostic tests, not a claim that passing them establishes general language competence.

Open problems include recovering relevant context, distinguishing genuine ambiguity from avoidable annotation disagreement, evaluating naturally mixed-language conversation and measuring interpretation across communities rather than only translated prompts. Some inputs legitimately admit several readings. A system that states the alternatives and the missing evidence may be more faithful to the language than one that confidently chooses a single answer.

Key takeaways

- Spoken, signed and written language expose different information; a transcript is a partial record.
- Computational tokens need not coincide with words, morphemes, lexemes or sound units; subword pieces from BPE, WordPiece or a unigram model reflect frequency, not morphology.
- Stemming strips endings by rule; lemmatization maps to a dictionary form and needs the part of speech. Both discard information some tasks need.
- Morphological structure and Chinese segmentation encode analytical choices with possible alternatives, and corpora follow different standards.

- Word counts discard order; part-of-speech categories, grouping and grammatical relations carry additional meaning. Constituency shows groups, dependency shows heads.
- Scope determines truth conditions: "each has some" need not mean "some shared by all."
- Assertion, background assumptions, cancellable inferences, reference, speech acts and discourse relations require the speaker's situation and conversational history.
- Corpora come with collection choices and annotation guidelines; check agreement, and expect a Zipfian long tail of rare words.

Exercises

1. For "The technician repaired two cameras," identify a word form, a lexeme, an inflectional contrast, a UPOS tag sequence and a possible computational tokenization. Explain what each representation preserves.

   Hint

   Compare "repaired" with REPAIR. A tokenizer could keep the whole form or split it; neither choice alone establishes the tense interpretation, which UD records as a feature.

2. Add one sentence of context for each reading of 學生會說話. Explain why the lexicon enumeration alone cannot prefer either context.

   Hint

   One setting can concern students' abilities; another can concern a student council announcing its position.

3. Change one training word's frequency in the BPE corpus. Inspect the first merge and encode `lowest` again. Why need neither result remain unchanged?

   Hint

   Pair counts are weighted by word occurrence counts. An early merge changes the pairs available later.

4. Write two contexts for "The technician inspected the panel with the camera," one for each attachment. Identify which head receives the modifier and how the event description changes.

   Hint

   Contrast the inspection method with identifying a camera-equipped panel among several panels.

5. Modify the quantifier program to add a technician who uses no camera. Predict both readings before running it, then explain the implication from B to A.

Hint

Both conditions fail if any technician has no usable pair. A single shared camera supplies a witness for every technician.

6. Analyze "Mei stopped checking the sensor" and "Some panels passed" separately under negation and explicit cancellation. Distinguish the main assertion, ordinary background assumption and conversational inference.

Hint

Try "Mei never checked it" as a challenge to background and "in fact, all passed" as cancellation of an inference. Keep the panel collection and test fixed.

References

1. Masato Hagiwara, *Real-World Natural Language Processing: Practical Applications with Deep Learning*, Manning, 2021. Publisher
2. Susan McRoy, *Principles of Natural Language Processing*, University of Wisconsin–Milwaukee open textbook, 2021. Grammars and Syntactic Processing
3. Catherine Anderson, Bronwyn Bjorkman, Derek Denis, Julianne Doner, Margaret Grant, Nathan Sanders and Ai Taniguchi, *Essentials of Linguistics*, 2nd ed., eCampusOntario, 2022. Open textbook
4. Catherine Anderson et al., "3.1 Modality," *Essentials of Linguistics*, 2nd ed., eCampusOntario, 2022. Chapter
5. Catherine Anderson et al., "4.1 Phonemes and allophones," *Essentials of Linguistics*, 2nd ed., eCampusOntario, 2022. Chapter
6. Catherine Anderson et al., "3.12 Tone and intonation," *Essentials of Linguistics*, 2nd ed., eCampusOntario, 2022. Chapter
7. Catherine Anderson et al., "5.1 What is morphology?," *Essentials of Linguistics*, 2nd ed., eCampusOntario, 2022. Chapter
8. Catherine Anderson et al., "6.4 Identifying phrases: Constituency tests," *Essentials of Linguistics*, 2nd ed., eCampusOntario, 2022. Chapter
9. Catherine Anderson et al., "8.4 Conversational implicatures," *Essentials of Linguistics*, 2nd ed., eCampusOntario, 2022. Chapter
10. Bolei Ma, Yuting Li, Wei Zhou, Ziwei Gong, Yang Janet Liu, Katja Jasinskaja, Annemarie Friedrich, Julia Hirschberg, Frauke Kreuter and Barbara Plank, "Pragmatics in the Era of Large Language Models: A Survey on Datasets, Evaluation, Opportunities and Challenges," *ACL*, pp. 8679–8696, 2025. Paper
11. Catherine Anderson et al., "10.2 Language varies," *Essentials of Linguistics*, 2nd ed., eCampusOntario, 2022. Chapter
12. Libo Qin, Qiguang Chen, Yuhang Zhou, Zhi Chen, Yinghui Li, Lizi Liao, Min Li, Wanxiang Che and Philip S. Yu, "A survey of multilingual large language models," *Patterns*, 6(1), 101118, 2025. Paper

13. Yanru Lu and Stefan Müller, “Deliminative verbal reduplication in Mandarin Chinese,” *Journal of Linguistics*, 62(3), pp. 595–632, 2026. Paper
14. A. Seza Doğruöz, Sunayana Sitaram, Barbara E. Bullock and Almeida Jacqueline Toribio, “A Survey of Code-switching: Linguistic and Social Perspectives for Language Technologies,” *ACL-IJCNLP*, pp. 1654–1666, 2021. Paper
15. Genta Winata, Alham Fikri Aji, Zheng Xin Yong and Thamar Solorio, “The Decades Progress on Code-Switching Research in NLP: A Systematic Survey on Trends and Challenges,” *Findings of ACL*, pp. 2936–2978, 2023. Paper
16. M. F. Porter, “An algorithm for suffix stripping,” *Program*, 14(3), pp. 130–137, 1980. DOI
17. Rico Sennrich, Barry Haddow and Alexandra Birch, “Neural Machine Translation of Rare Words with Subword Units,” *ACL*, 2016. Paper
18. Mike Schuster and Kaisuke Nakajima, “Japanese and Korean Voice Search,” *ICASSP*, pp. 5149–5152, 2012. DOI
19. Yonghui Wu et al., “Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation,” arXiv:1609.08144, 2016. Paper
20. Taku Kudo, “Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates,” *ACL*, 2018. Paper
21. Taku Kudo and John Richardson, “SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing,” *EMNLP System Demonstrations*, 2018. Paper
22. Sabrina J. Mielke et al., “Between Words and Characters: A Brief History of Open-Vocabulary Modeling and Tokenization in NLP,” arXiv:2112.10508, 2021. Review
23. Hugging Face, “The Tokenization Pipeline,” Tokenizers documentation, current online edition. Documentation
24. Marie-Catherine de Marneffe, Christopher D. Manning, Joakim Nivre and Daniel Zeman, “Universal Dependencies,” *Computational Linguistics*, 47(2), 2021. Paper
25. Universal Dependencies contributors, “Universal POS tags,” Universal Dependencies documentation, current online edition. Documentation
26. Mitchell P. Marcus, Beatrice Santorini and Mary Ann Marcinkiewicz, “Building a Large Annotated Corpus of English: The Penn Treebank,” *Computational Linguistics*, 19(2), 1993. Paper
27. Thomas Emerson, “The Second International Chinese Word Segmentation Bakeoff,” *Proceedings of the Fourth SIGHAN Workshop on Chinese Language Processing*, 2005. Paper
28. Jacob Cohen, “A Coefficient of Agreement for Nominal Scales,” *Educational and Psychological Measurement*, 20(1), pp. 37–46, 1960. DOI
29. Steven T. Piantadosi, “Zipf’s word frequency law in natural language: A critical review and future directions,” *Psychonomic Bulletin & Review*, 21(5), pp. 1112–1130, 2014. DOI
30. Jacob Devlin, Ming-Wei Chang, Kenton Lee and Kristina Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” *NAACL-HLT*, 2019. Paper

NLP 03 • Neural Networks for NLP

From count features and logistic classification to embedding layers, the neural probabilistic language model and 1-D convolutional text classifiers, with padding, masking, regularization and a complete PyTorch experiment.

Online edition: https://blog.shyandram.dev/en/learn/nlp/neural-networks-for-nlp/

What you'll learn

- Define a text classification task with explicit annotation rules, then derive count features, TF–IDF, logistic classification and cross-entropy.
- Turn token IDs into learned vectors with an embedding layer, pool them, and classify with a small multilayer perceptron.
- Follow the neural probabilistic language model of Bengio et al. and explain why shared word vectors let it generalize to unseen histories.
- Write the mathematics of a 1-D convolutional text classifier with max-over-time pooling, and implement it with correct padding and masking.
- Regularize, train and early-stop a small PyTorch model, read its training curves and analyze its errors by category.
- Evaluate decisions, probability quality, coverage and class imbalance separately.

The big picture

A text classifier converts a message into a permitted decision: route a complaint, identify a language, label a review or flag a document for inspection. Its most consequential choices often occur before the neural architecture is selected. What counts as one example? Which context is visible? What does the label mean? What should happen when none of the available labels fits?

Hagiwara develops a first classification application, word and document embeddings, sentence classification and convolutional encoders in Chapters 2–4 and 7 [1]. McRoy's Chapter 6 distinguishes sentiment, similarity, grammaticality and entailment as different language tasks [2]. This lesson follows that combination: build an application, but make its prediction target precise. A sentiment label is not an entailment judgment, and a similarity score is not a statement that two claims have the same truth conditions.

> **Explain like I'm five**
>
> Imagine sorting letters into three trays: payments, deliveries and technical help. Counting words is like looking for familiar clues on each envelope. A neural network learns its own clues: it gives every word a small list of numbers, learns which short phrases matter, and adds up the evidence. Either way,

a trustworthy sorter also needs a fourth action: ask someone when the letter does not clearly belong in a tray.

The lesson has two halves. The first builds the strongest simple baseline: sparse features and a linear classifier, trained and evaluated with care. The second replaces hand-designed features with learned ones: an **embedding layer**, a pooled multilayer perceptron, the neural probabilistic language model and a convolutional encoder. Every neural model here is still trained with the same cross-entropy loss as the linear baseline, so the change is in the representation, not the objective. Backpropagation, activation functions and optimizers are developed in ML 02; this page uses them and focuses on what is specific to text.

All examples use invented messages and require no dataset download. Measured scores describe these synthetic messages; they cannot establish performance on an actual application.

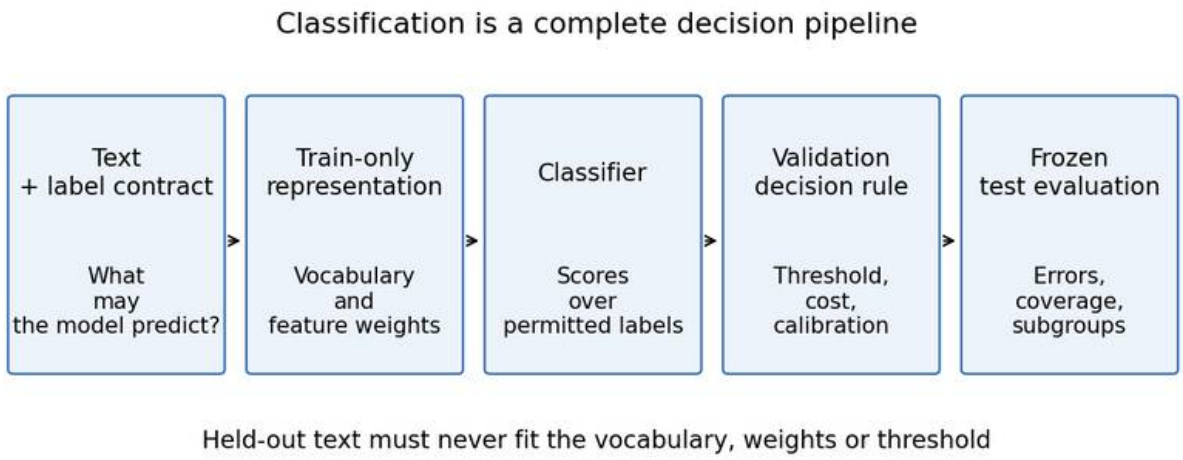


Figure 1 — Classification includes the representation, prediction and action rule. Holding out the final test set applies to every learned stage, including neural feature learning.

Define the prediction contract

Plain version. A label is an agreement about what a piece of text means for this application. Establish that agreement before collecting examples or optimizing a loss.

In single-label classification, one input receives exactly one label from a fixed set. In multi-label classification, several labels may apply simultaneously. A complaint about both a delayed parcel and an incorrect charge may require two routes, a primary route with secondary tags, or a human triage decision. These are different tasks even if the input text is identical.

Write the contract in ordinary language first. For our toy application, billing concerns a payment or invoice, delivery concerns a parcel or shipping, and technical concerns account access or software behavior. We permit an abstention action after classification. An ambiguous multi-topic

message is therefore not silently forced into a category merely because the numerical model has an argmax.

$$f_\theta : \mathcal{X} \to \Delta^{K-1}, \qquad \hat{y}(x) = \operatorname*{argmax}_{k \in \{1,\dots,K\}} p_\theta(y = k \mid x).$$

Here f_θ is a classifier with learned parameters θ, $\mathcal{X}$ is the set of allowed text inputs, Δ^{K-1} is the simplex of probability vectors over K classes, x is one input, y is its label, k indexes a class, p_θ is the model's conditional distribution and $\hat{y}$ is its most probable class. The simplex requires nonnegative entries whose sum is one.

This equation describes a model output, not the whole product action. The service may return a route only when confidence passes a validation-selected threshold. It may also reject an empty message, ask for additional context or send a multi-topic message to review. Those actions should be visible in evaluation.

Keep the unit of supervision aligned with deployment. A sentence-level sentiment model trained on isolated opinions may fail on a whole conversation containing quotations, corrections and several speakers. If deployment includes previous turns, annotation and training should include the same permitted context. Conversely, giving annotators access to the final resolution while the model only sees the first message creates an impossible target.

Count features and the vocabulary

Plain version. The simplest useful representation records which words or short sequences occur. It is inexpensive, inspectable and often an excellent first comparison.

Choose a vocabulary from training text only. Its entries might be words, character sequences or consecutive word pairs. For a word vocabulary, each coordinate counts one vocabulary entry. A document becomes a sparse vector because most possible words are absent.

$$x_j(d) = c(v_j, d), \qquad \mathbf{x}(d) = (x_1(d), \dots, x_V(d))^\top.$$

Here d is a document, v_j is the vocabulary item at coordinate j, $c(v_j, d)$ is its occurrence count in d, $x_j(d)$ is that feature value, $\mathbf{x}(d)$ is the resulting column vector and V is the vocabulary size. The transpose symbol $\top$ specifies that the listed coordinates form a column.

This bag-of-words representation ignores order. "Payment arrived after the refund" and "Refund arrived after the payment" contain the same words but describe different event orders. Word bigrams recover some local information, while character features can capture spelling fragments and inflections. Neither automatically understands a long dependency or the scope of a negation.

Feature choices encode useful assumptions. Lowercasing can reduce sparsity, but it merges names with common nouns. Removing every stop word can remove "not," which is essential to many classifications. A language detector may benefit from character patterns rather than a word vocabulary tailored to one language. Use the task definition to justify each normalization rule; NLP 02 discusses the linguistic units behind these choices.

Fit vocabulary construction, document-frequency thresholds and any feature selection inside the training portion of each experiment. Looking at test documents to decide which rare terms to retain gives the training pipeline information about the evaluation distribution. It can be subtle leakage even when no test label is read.

Keep a record of vocabulary size, proportion of messages with no recognized features, token counts and representative discarded terms. These diagnostics reveal changes that a single accuracy score hides. A new product code or language can make many inputs nearly empty without causing an obvious exception.

TF–IDF and vector normalization

Plain version. A word that occurs in almost every message is usually less useful for distinguishing messages. TF–IDF balances occurrence within a document against rarity across training documents.

One common smoothed definition, used by the implementation here, is:

$$\mathrm{idf}_j = \log \frac{1+N}{1+\mathrm{df}_j} + 1, \qquad z_j(d) = c(v_j, d)\,\mathrm{idf}_j, \qquad \tilde{\mathbf{z}}(d) = \frac{\mathbf{z}(d)}{\|\mathbf{z}(d)\|_2}.$$

Here idf_j is the inverse-document-frequency weight for vocabulary entry v_j, N is the number of training documents, df_j counts training documents containing that entry, $c(v_j, d)$ is its count in document d, $z_j(d)$ is the weighted coordinate, $\mathbf{z}(d)$ collects those coordinates and $\tilde{\mathbf{z}}(d)$ is its unit-length version. The Euclidean norm $\|\mathbf{z}(d)\|_2$ is the square root of the sum of squared coordinates. A zero vector remains zero rather than being divided by zero. The scikit-learn feature-extraction documentation specifies these implementation choices [3].

The added ones prevent undefined weights for the fitted vocabulary and fix the precise smoothing convention. Other libraries use other conventions, so “TF–IDF” alone is not a complete reproducibility statement. Record the tokenizer, feature range, count transformation, weighting formula and normalization.

Normalization makes a long message less dominant simply because it repeats words. That helps when direction in feature space matters more than total length. But length can itself be predictive: an error report might be longer than a password-reset request. If length is relevant, add an explicit length feature instead of relying on an accidental effect of unnormalized counts.

TF–IDF does not infer synonymy. “Charge” and “payment” remain different coordinates unless examples teach the classifier their similar role. It also weights rare identifiers strongly even when their rarity is irrelevant. An account number might look discriminative because a template is duplicated across related records. Grouped splitting and inspecting coefficients help reveal that shortcut.

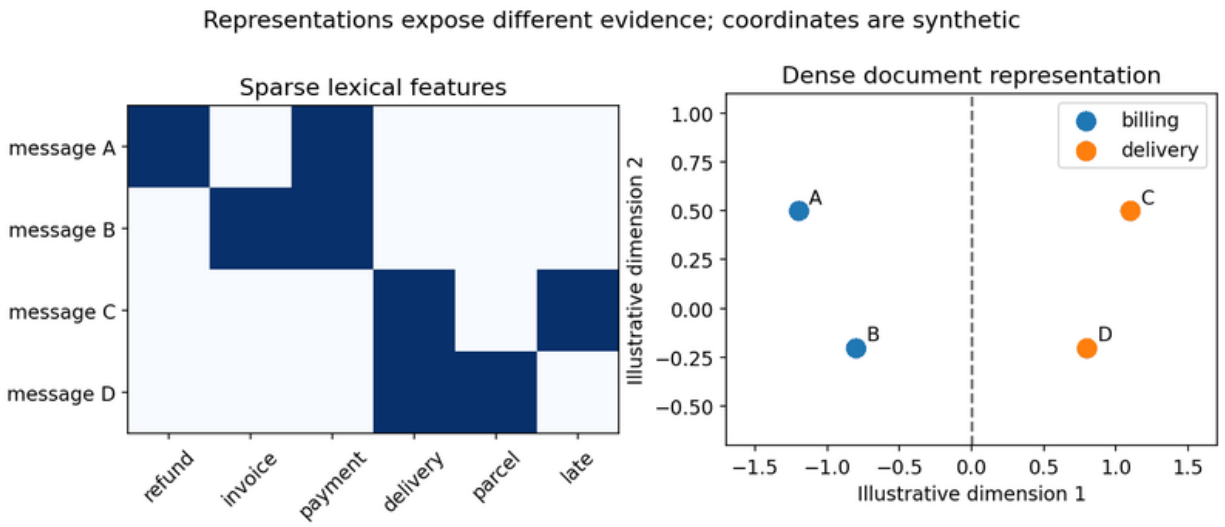


Figure 2 — Sparse features preserve explicit lexical evidence. Dense representations compress patterns into coordinates whose meaning depends on the training objective.

Logistic classification and its loss

Plain version. A linear classifier adds evidence from features to produce one score per label. Turning scores into probabilities gives a differentiable training objective and a basis for decisions.

For a multiclass model:

$$s_k = \mathbf{w}_k^\top \tilde{\mathbf{z}} + b_k, \qquad p_\theta(y = k \mid d) = \frac{\exp(s_k)}{\sum_{\ell=1}^{K} \exp(s_\ell)}.$$

Here s_k is the score for class k, $\mathbf{w}_k$ is its learned feature-weight vector, $\tilde{\mathbf{z}}$ is the normalized document representation, b_k is a learned bias, p_θ is the conditional class probability, θ collects all weights and biases, d is the document, ℓ indexes competing classes and K is their number. The function exp is the exponential function.

The probability of the annotated class should increase during training. Regularization discourages arbitrarily large weights that fit incidental words:

$$\mathcal{L}(\theta) = -\frac{1}{n} \sum_{i=1}^{n} \log p_\theta(y_i \mid d_i) + \frac{\lambda}{2} \sum_{k=1}^{K} \|\mathbf{w}_k\|_2^2.$$

Here $\mathcal{L}$ is the objective, n is the training-example count, i indexes examples, (d_i, y_i) is an annotated document and label, $\lambda \geq 0$ is the regularization strength, and $\|\mathbf{w}_k\|_2^2$ is the sum of squared feature weights for class k. The first term is multiclass cross-entropy; the second is squared-weight regularization. Library objectives can scale the terms differently; scikit-learn exposes inverse regularization through its parameter C [4]. IT 01 explains why the first term is a cross-entropy.

A large positive coefficient says that a feature increases a class score relative to the other evidence. It does not prove that the feature causes the true category. Correlated words can share or exchange weight, and changing the vocabulary changes the meaning of a coefficient comparison.

For binary classification, the softmax can be written with one sigmoid of a score difference. For independent multi-label prediction, use a separate sigmoid per label rather than a softmax that forces labels to compete. Choose a loss and output activation that match the annotation contract, not whichever head happens to be easiest to implement.

Read as a network, this model is a single linear layer followed by a softmax. Every neural classifier below keeps that final layer and its loss. What changes is the vector $\tilde{\mathbf{z}}$: instead of being computed by a fixed counting rule, it is computed by learned layers.

A complete CPU classification experiment

Plain version. The vocabulary, classifier and threshold must learn from the intended portions of data. A pipeline makes this boundary easier to preserve.

The following standalone example uses invented messages. Training, validation and test text are explicitly separate. Validation selects a threshold using a stated illustrative cost: one wrong route costs two units, while asking for review costs one fifth of a unit. These values are policy parameters for this demonstration, not measured business costs. All choices freeze before the final test is inspected.

```
import numpy as np
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import f1_score, log_loss

train = [
    ("please refund my payment", "billing"),
    ("the invoice has a wrong charge", "billing"),
    ("my payment was charged twice", "billing"),
    ("how can i update the billing card", "billing"),
    ("please send the payment receipt", "billing"),
    ("the refund is missing from my invoice", "billing"),
    ("where is my delivery parcel", "delivery"),
    ("my shipment is late", "delivery"),
    ("please change the delivery address", "delivery"),
    ("the parcel arrived damaged", "delivery"),
    ("how do i track the shipment", "delivery"),
    ("the delivery driver could not find the address", "delivery"),
    ("i cannot login to my account", "technical"),
    ("the application crashes on startup", "technical"),
    ("please reset my account password", "technical"),
    ("the software shows an error", "technical"),
    ("my login fails after the software update", "technical"),
    ("the application cannot connect to my account", "technical"),
]
```

```
validation = [
    ("i need a receipt for the charge", "billing"),
    ("the billing payment needs a refund", "billing"),
    ("can i track this delivery", "delivery"),
    ("the shipment went to another address", "delivery"),
    ("my password does not allow login", "technical"),
    ("the software crashes with an error", "technical"),
]
test = [
    ("please explain the invoice payment", "billing"),
    ("a charge appears before the refund", "billing"),
    ("the parcel delivery is not here", "delivery"),
    ("my shipment arrived at the wrong address", "delivery"),
    ("an account error prevents login", "technical"),
    ("the application crashes after a password change", "technical"),
]

def unpack(rows):
    return [text for text, _ in rows], np.array([label for _, label in rows])

train_x, train_y = unpack(train)
val_x, val_y = unpack(validation)
test_x, test_y = unpack(test)
model = Pipeline([
    ("features", TfidfVectorizer(ngram_range=(1, 2))),
    ("classifier", LogisticRegression(C=4.0, max_iter=300, random_state=0)),
])
model.fit(train_x, train_y)
classes = model.named_steps["classifier"].classes_

def decision(probabilities, threshold):
    best = probabilities.argmax(axis=1)
    prediction = classes[best]
    accepted = probabilities[np.arange(len(best)), best] >= threshold
    return prediction, accepted

val_prob = model.predict_proba(val_x)
grid = [0.40, 0.50, 0.60, 0.70, 0.80]
costs = []
for threshold in grid:
    pred, accepted = decision(val_prob, threshold)
    wrong = accepted & (pred != val_y)
    costs.append((2.0 * wrong.sum() + 0.2 * (~accepted).sum()) / len(val_y))
threshold = grid[int(np.argmin(costs))]  # first minimum fixes the tie rule

test_prob = model.predict_proba(test_x)
test_pred, accepted = decision(test_prob, threshold)
print("validation-selected threshold:", threshold)
print("test macro F1:", round(f1_score(test_y, test_pred, average="macro"), 3))
print("test log loss:", round(log_loss(test_y, test_prob, labels=classes), 3))
print("test coverage:", round(accepted.mean(), 3))
if accepted.any():
    print("accepted accuracy:", round((test_pred[accepted] == test_y[accepted]).mean(), 3))
else:
    print("accepted accuracy: undefined (no accepted messages)")
probe = ["the weather is pleasant", "my delivery charge is wrong"]
```

```
for text, probabilities in zip(probe, model.predict_proba(probe)):
    prediction, keep = decision(probabilities[None, :], threshold)
    print(text, "->", prediction[0] if keep[0] else "review")
```

The run prints:

```
validation-selected threshold: 0.4
test macro F1: 1.0
test log loss: 0.413
test coverage: 1.0
accepted accuracy: 1.0
the weather is pleasant -> delivery
my delivery charge is wrong -> delivery
```

Six perfectly classified test messages are far too few to support any claim about the method; they only show that the pipeline runs end to end. The validation set made every threshold above 0.40 look costly, so the lowest grid value won, and the policy accepts everything.

The unfamiliar weather message is the useful part. None of the three routes is correct, yet the model still returns a probability distribution, and here its largest probability (about 0.51 for delivery) passes the selected threshold. The output head has no additional class for arbitrary outside topics. An abstention threshold can reduce some inappropriate actions, but a probability maximum is not a guaranteed out-of-domain detector, and in this run it does not catch the message at all. The mixed-topic "delivery charge" message is likewise forced into one route.

Notice that the final macro F1 measures forced class predictions over all test examples, while coverage and accepted accuracy describe the action rule. Reporting only accepted accuracy can hide a model that rejects nearly everything. Reporting only forced accuracy can hide useful selective behavior. Keep both views and their denominators explicit.

For a real application, replace these toy lists with a documented corpus and group identifiers. Messages from one conversation should not be split randomly across train and test. Shared templates, repeated signatures and near-duplicate descriptions also need scrutiny. The small example intentionally exposes mechanics; it offers too few validation examples for a stable threshold estimate.

A controlled negation test

Changing the feature vocabulary can preserve information that an unordered word count loses. Here we hold the invented labeled corpus and classifier family fixed, then compare single-token features with a mixture of single tokens and adjacent pairs. The TF–IDF and logistic implementations are documented in [3,4].

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
```

```
from sklearn.pipeline import make_pipeline

train = [
    ("the battery is good", 1), ("good screen and good sound", 1),
    ("the camera is great", 1), ("great value , works well", 1),
    ("the battery is not good", 0), ("not great , the screen broke", 0),
    ("the sound is bad", 0), ("bad camera and bad battery", 0),
    ("works well , not bad at all", 1), ("not good , not worth it", 0),
]
test = ["the screen is not good", "the sound is not bad", "great battery"]
texts, labels = zip(*train)

for name, ngrams in [("unigrams", (1, 1)), ("uni+bigrams", (1, 2))]:
    clf = make_pipeline(TfidfVectorizer(ngram_range=ngrams, token_pattern=r"\S+"),
                        LogisticRegression(C=10.0))
    clf.fit(texts, labels)                    # vocabulary + IDF fitted on train only
    probs = clf.predict_proba(test)[:, 1]
    print(f"{name:12s}", [f"{t!r}: {p:.2f}" for t, p in zip(test, probs)])
```

The positive-review probabilities are:

```
unigrams     ["'the screen is not good': 0.43", "'the sound is not bad': 0.11", "'great
 ↪  battery': 0.61"]
uni+bigrams  ["'the screen is not good': 0.20", "'the sound is not bad': 0.18", "'great
 ↪  battery': 0.56"]
```

Both models give “the screen is not good” a probability below one half; adding pairs decreases it from 0.43 to 0.20. For “the sound is not bad,” the probability rises from 0.11 to 0.18 but remains below one half. These observations establish the behavior of the fitted toy models. They do not isolate one coefficient as the cause: adding pairs changes the vocabulary, normalization and joint fitted weights.

The second sentence is a useful error probe because a local phrase feature is available, yet the predicted label is still wrong under the intended positive reading. The corpus contains only one positive “not bad” occurrence, with other positive wording. Learning which feature carries that label is underdetermined by such sparse evidence. Inspect feature contributions and add controlled examples that vary negation while keeping topic fixed before drawing a general conclusion about the architecture.

There is also a distinction between phrase identity and scope. Adjacent features can distinguish “not good” from “good,” but recognizing “not every repair succeeded” versus “every repair failed” requires a different compositional distinction. More pairs do not automatically provide a complete semantic analysis. NLP 02 and NLP 12 work through such cases explicitly.

Two limits of sparse features motivate the rest of this lesson. First, every bigram is a separate coordinate: “not good” and “not great” share nothing unless both appear in training with labels. Second, the number of possible n-grams grows quickly with n, so long patterns are rarely observed. Neural models address both by representing words as vectors that can share evidence, and by learning which combinations of those vectors matter.

From word IDs to learned vectors: the embedding layer

Plain version. Give every vocabulary entry a short vector of learned numbers. Words that behave alike in the training task can end up with similar vectors, so evidence about one helps the other.

An **embedding layer** is a table with one row per vocabulary entry. Looking up a token is the same as multiplying its one-hot vector by the table:

$$\mathbf{e}_t = E^\top \mathbf{o}_{i_t} = E^\top_{i_t,:}, \qquad E \in \mathbb{R}^{V \times D}.$$

Here i_t is the vocabulary ID of the token at position t, $\mathbf{o}_{i_t} \in \{0,1\}^V$ is the one-hot vector with a single one in coordinate i_t, E is the learned embedding table with V rows and D columns, $E_{i_t,:}$ is row i_t, and $\mathbf{e}_t \in \mathbb{R}^D$ is the token's embedding as a column vector. D is usually far smaller than V, which is why embeddings are called **dense**: most coordinates are nonzero, unlike the sparse count vector.

The lookup has a useful consequence for training. The gradient of the loss with respect to E is nonzero only in rows of tokens that appear in the current batch, so each update touches a few rows. A word that never appears in training keeps its initial row, which is why a vocabulary needs an explicit unknown-word entry and a policy for using it. The table can be initialized randomly and trained with the classifier, or initialized from vectors pretrained on unlabeled text; NLP 04 explains how such vectors are learned and what their geometry does and does not mean.

Pool, then classify: the bag-of-embeddings MLP

A document has a variable number of tokens, while a classifier layer expects a fixed-width input. The simplest neural answer is to average the token vectors and feed the average to a **multilayer perceptron (MLP)**:

$$\mathbf{h} = \frac{1}{T}\sum_{t=1}^{T} \mathbf{e}_t, \qquad \mathbf{a} = \phi\left(W_1\mathbf{h} + \mathbf{b}_1\right), \qquad \mathbf{s} = W_2\mathbf{a} + \mathbf{b}_2, \qquad p_\theta(y = k \mid d) = \mathrm{softmax}(\mathbf{s})_k.$$

Here T is the number of real tokens in document d, $\mathbf{h} \in \mathbb{R}^D$ is the mean-pooled document vector, $W_1 \in \mathbb{R}^{H \times D}$ and $\mathbf{b}_1 \in \mathbb{R}^H$ are the hidden layer's weights and bias, H is the hidden width, ϕ is an elementwise nonlinearity such as ReLU, $\mathbf{a}$ is the hidden activation, $W_2 \in \mathbb{R}^{K \times H}$ and $\mathbf{b}_2 \in \mathbb{R}^K$ produce the class scores $\mathbf{s}$, and $\mathrm{softmax}(\mathbf{s})_k = \exp(s_k)/\sum_\ell \exp(s_\ell)$. The parameters $\theta = (E, W_1, \mathbf{b}_1, W_2, \mathbf{b}_2)$ are trained jointly by minimizing the same cross-entropy as the logistic model, with gradients computed by backpropagation.

Two published designs show how far this simple shape can go. Joulin et al. average word and n-gram embeddings and feed the result to a linear classifier, trading depth for speed [7]. Iyyer et al.'s deep averaging network stacks several nonlinear layers on the average and adds **word dropout**, which removes whole tokens before averaging during training [6]. Goldberg's primer develops this family of feed-forward text models in detail [5].

The hidden layer matters. A linear model on the mean vector can only add per-word evidence, so it cannot represent "*good* is positive unless *not* is present" as an interaction. A nonlinear hidden

layer can combine the presence of "not" with the presence of a positive word. What no pooling-based model can do is recover order: any permutation of the tokens gives the same mean. "The screen is good but the case is bad" and "the case is good but the screen is bad" are identical inputs. The experiment later in this lesson measures exactly this failure.

The neural probabilistic language model

Plain version. A language model predicts the next word from the previous ones. Bengio et al. showed that if each previous word is represented by a learned vector, a network can share evidence between similar histories and assign sensible probabilities to word sequences it never saw.

A **language model** assigns a probability to a sequence by predicting one token at a time. Writing w_t for the token at position t, the chain rule gives

$$P(w_1, \dots, w_T) = \prod_{t=1}^{T} P(w_t \mid w_1, \dots, w_{t-1}).$$

Here $P(w_1, \dots, w_T)$ is the probability of the whole sequence and $P(w_t \mid w_1, \dots, w_{t-1})$ is the conditional probability of the next token given everything before it. Count-based n-gram models approximate each factor using only the previous $n-1$ tokens and estimate it from counts; NLP 05 develops them together with smoothing and perplexity. Their weakness is that the history "a dog" shares nothing with "a cat": if the first never occurred, the counts are silent about it.

Bengio et al. keep the fixed $(n-1)$-token history but replace the count table with a neural network [8]. Each history word is looked up in a shared table C, the vectors are concatenated, and a hidden layer produces a score for every vocabulary word:

$$\mathbf{x} = [C(w_{t-1}); C(w_{t-2}); \dots; C(w_{t-n+1})], \qquad \mathbf{y} = \mathbf{b} + W\mathbf{x} + U \tanh(\mathbf{d} + H\mathbf{x}), \qquad \hat{P}(w_t = i \mid w_{t-n+1}^{t-1}) = \frac{\exp(y_i)}{\sum_{j=1}^{V} \exp(y_j)}.$$

Here $C(w) \in \mathbb{R}^m$ is the learned feature vector of word w (a row of the $V \times m$ table C), the semicolons denote concatenation, so $\mathbf{x} \in \mathbb{R}^{(n-1)m}$; $H \in \mathbb{R}^{h \times (n-1)m}$ and $\mathbf{d} \in \mathbb{R}^h$ are the hidden layer's weights and bias with h hidden units; $U \in \mathbb{R}^{V \times h}$ maps hidden units to vocabulary scores; $W \in \mathbb{R}^{V \times (n-1)m}$ is an optional **direct connection** from the word features to the output (set to zero when not used); $\mathbf{b} \in \mathbb{R}^V$ is the output bias; $\mathbf{y}$ holds the unnormalized log-probabilities; V is vocabulary size; and w_{t-n+1}^{t-1} abbreviates the history. This is the paper's equation (1) in our notation. Training maximizes the average log-likelihood of the training tokens plus a weight-decay penalty [8].

Three ideas in this model recur throughout the rest of the course. Word vectors are learned as a by-product of a prediction task. Concatenation, unlike averaging, keeps the position of each history word, so "dog bites" and "bites dog" differ. And the softmax over the entire vocabulary is the expensive part: its cost grows with V for every predicted token, which later motivated cheaper objectives such as the negative sampling discussed in NLP 04.

The following example trains a tiny version on an invented corpus in which the history "a dog"

never occurs. We use Adam rather than the paper's stochastic gradient procedure, purely for convenience.

```
import torch
from torch import nn
from torch.nn import functional as F

torch.manual_seed(0)
torch.set_num_threads(1)
corpus = """a cat sleeps . a cat eats . the cat sleeps . the cat eats .
the dog sleeps . the dog eats . a bird sings . the bird sings .
a bird eats . the bird eats""".split(".")      # the history "a dog" never occurs
sentences = [["<s>", "<s>"] + s.split() + ["</s>"] for s in corpus]
vocab = sorted({w for s in sentences for w in s})
ids = {w: i for i, w in enumerate(vocab)}
X, Y = [], []
for s in sentences:
    for t in range(2, len(s)):                  # predict w_t from (w_{t-2}, w_{t-1})
        X.append([ids[s[t - 2]], ids[s[t - 1]]])
        Y.append(ids[s[t]])
X, Y = torch.tensor(X), torch.tensor(Y)

class NPLM(nn.Module):
    def __init__(self, V, m=8, h=16, context=2, direct=True):
        super().__init__()
        self.C = nn.Embedding(V, m)                     # shared word features
        self.H = nn.Linear(context * m, h)              # hidden layer (bias = d)
        self.U = nn.Linear(h, V)                        # hidden-to-output (bias = b)
        self.W = nn.Linear(context * m, V, bias=False) if direct else None
    def forward(self, ctx):
        x = self.C(ctx).flatten(1)                      # concatenate the context vectors
        y = self.U(torch.tanh(self.H(x)))
        return y + self.W(x) if self.W is not None else y

model = NPLM(len(vocab))
opt = torch.optim.Adam(model.parameters(), lr=0.02, weight_decay=1e-4)
for step in range(400):
    opt.zero_grad()
    loss = F.cross_entropy(model(X), Y)
    loss.backward()
    opt.step()
print(f"training cross-entropy {loss.item():.3f} nats, {len(Y)} predictions")

history_count = sum(s[i:i + 2] == ["a", "dog"] for s in sentences for i in range(len(s)))
with torch.no_grad():
    p = F.softmax(model(torch.tensor([[ids["a"], ids["dog"]]])), dim=-1)[0]
print("training occurrences of the history 'a dog':", history_count)
for w in ["sleeps", "eats", "sings"]:
    print(f"P({w} | a dog) = {p[ids[w]].item():.3f}")
```

The run prints:

```
training cross-entropy 0.576 nats, 40 predictions
```

```
training occurrences of the history 'a dog': 0
P(sleeps | a dog) = 0.509
P(eats | a dog) = 0.490
P(sings | a dog) = 0.001
```

A trigram count model has no data for the history "a dog" and would have to back off to shorter histories. The neural model instead puts almost all probability on the two verbs that dogs (and cats) take in training, and almost none on "sings," which only birds do. It generalizes because "dog" and "cat" receive related vectors from sharing contexts, and because "a" and "the" do too. That is the central claim of Bengio et al., reproduced here at toy scale; it says nothing about how the model compares with a well-smoothed n-gram model on real text, which is what the paper measured [8].

Convolutional text classifiers

Plain version. A convolutional filter slides along the sentence and fires when it sees a particular short phrase pattern, wherever that pattern occurs. Taking the strongest response of each filter gives a fixed-size summary that remembers *which* patterns occurred, though not *where*.

Collobert et al. built a unified neural network for tagging tasks in which convolution over word vectors, followed by max pooling over positions, summarizes a whole sentence [9]. Kim showed that a single convolutional layer on top of pretrained word vectors is a strong sentence classifier [10]. Convolution itself, with its weight sharing and pooling, is developed for images in ML 03; text uses the one-dimensional version, sliding along positions only.

The mathematics of one filter

Stack the T token embeddings of a sentence into a matrix $X \in \mathbb{R}^{T \times D}$ whose row t is $\mathbf{e}_t^\top$. A filter of width h covers h consecutive rows:

$$c_t = \rho(\langle W, X_{t:t+h-1} \rangle + b), \qquad t = 1, \dots, T-h+1, \qquad \hat{c} = \max_t c_t.$$

Here $X_{t:t+h-1} \in \mathbb{R}^{h \times D}$ is the window of rows t through $t+h-1$, $W \in \mathbb{R}^{h \times D}$ contains the filter's learned weights, $\langle A, B \rangle = \sum_{r,s} A_{rs} B_{rs}$ sums elementwise products, $b \in \mathbb{R}$ is a bias, ρ is a nonlinearity such as ReLU or tanh, c_t is the filter's response at window t, the list $(c_1, \dots, c_{T-h+1})$ is the **feature map**, and $\hat{c}$ is the result of **max-over-time pooling**. Kim writes the same operation with the window flattened to a vector of length hD [10].

A model uses many filters. With F filters for each width in a set such as $\{3, 4, 5\}$, the pooled values form a vector $\mathbf{z} \in \mathbb{R}^{F \cdot (\text{number of widths})}$, which a linear layer and softmax turn into class probabilities, trained with cross-entropy as before. In PyTorch, `nn.Conv1d(D, F, h)` computes all F feature maps of width h at once; it expects the input as (batch, D, T), so the embedding output must be transposed.

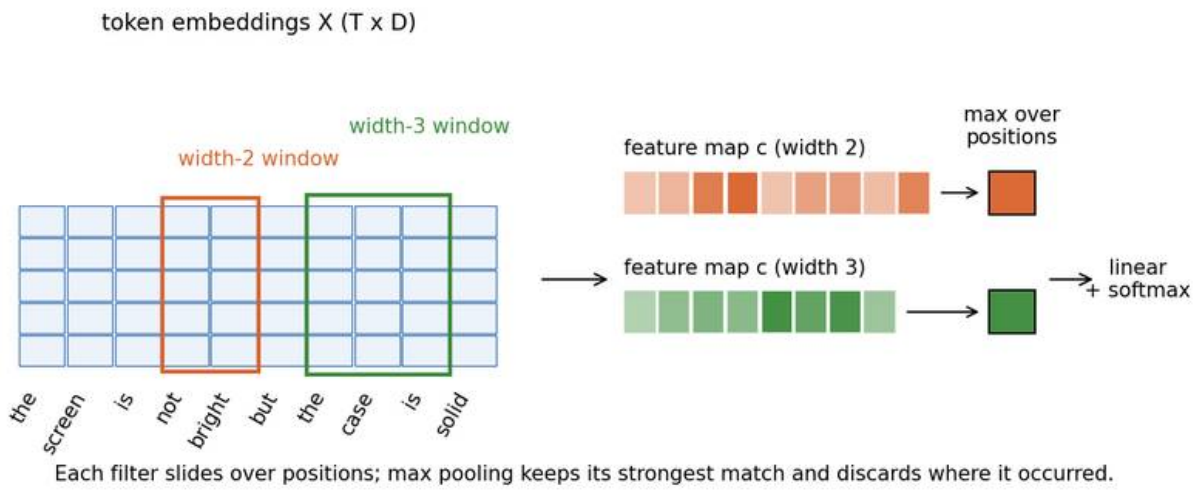


Figure 3 — A 1-D convolutional text classifier. Each filter scores every window of its width; max-over-time pooling keeps the strongest score and discards its position.

What the pieces buy, and what they lose

Weight sharing. The same filter is applied at every position, so a pattern learned from "not good" at the start of one sentence is detected at the end of another. The parameter count depends on h, D and F, not on sentence length.

Soft n-grams. Because the filter takes dot products with embeddings, a filter that responds to "not good" also responds, more weakly, to "not great" if the two adjectives have similar vectors. This is the advantage over sparse bigram features.

Max pooling. Keeping only the maximum handles variable sentence lengths and records whether a pattern occurred anywhere. It discards where the pattern occurred and how many times. A one-layer filter also cannot connect tokens farther apart than its width: with widths up to 4, nothing in the model sees "but" and an adjective four positions later inside the same window. The experiment below shows exactly this.

Kim's paper compares four ways to treat the embedding table: random initialization trained with the model (CNN-rand), pretrained vectors kept fixed (CNN-static), pretrained vectors fine-tuned (CNN-non-static), and two channels, one fixed and one fine-tuned, with each filter applied to both (CNN-multichannel) [10]. His reported settings use filter widths 3, 4 and 5 with 100 feature maps each, dropout rate 0.5 on the pooled layer, an ℓ_2-norm constraint of 3 on the output weight vectors, and mini-batches of 50 [10]. Zhang and Wallace later ran a sensitivity analysis of this one-layer design and concluded that the filter region size can matter a lot and should be tuned, that 1-max pooling uniformly outperformed the other pooling strategies they tried, that ReLU and tanh were the best activation candidates, and that variance across runs must be reported when comparing configurations [11].

Recurrent encoders take a different route to order: they update a hidden state while reading the

sequence and can, in principle, connect distant tokens. Hagiwara's sentence-classification discussion develops recurrent alternatives alongside practical training [1]. NLP 05 develops them for language and ML 04 gives the shared mathematics. Contextual encoders based on attention let every token's representation depend on the whole input; NLP 07 and NLP 10 develop them.

Choose according to evidence. A sparse classifier may be enough when labels have clear vocabulary clues. A convolutional or contextual encoder may help with indirect phrasing, negation or domain-specific meanings. Its larger capacity also makes split design and annotation quality more consequential. Complexity should answer a measured failure, such as errors on paraphrases, rather than stand in for evaluation.

Online only: TextCNN — Convolutional Neural Networks for Sentence Classification — https://blog.shyandram.dev/en/learn/nlp/neural-networks-for-nlp/papers/textcnn–convolutional-neural-networks-for-sentence-classification/

Padding, masking and batching

Plain version. Sentences have different lengths, but a batch is a rectangle. We fill short sentences with a padding token and then make sure that padding never influences the result.

Training processes several sequences at once as one tensor of shape (batch, $T_{\max}$), where $T_{\max}$ is the longest length in the batch. Shorter sequences are filled with a reserved **padding ID**, usually 0. A boolean **mask** m_{bt} records which positions are real:

$$m_{bt} = \mathbf{1}[t \le T_b], \qquad \mathbf{h}_b = \frac{\sum_{t=1}^{T_{\max}} m_{bt}\, \mathbf{e}_{bt}}{\max\left(1, \sum_{t=1}^{T_{\max}} m_{bt}\right)}.$$

Here b indexes sequences in the batch, t indexes positions, T_b is the real length of sequence b, $\mathbf{1}[\cdot]$ is one when its condition holds and zero otherwise, $\mathbf{e}_{bt}$ is the embedding at position t of sequence b, and $\mathbf{h}_b$ is the **masked mean**. The $\max(1, \cdot)$ guards against an empty sequence. Each pooling operation needs its own masking rule:

- **Mean pooling** must divide by the real length, not by $T_{\max}$. A zero padding vector does not change the sum, but it does change an unmasked denominator.
- **Max pooling** must exclude padded positions or windows. The general rule is to fill them with $-\infty$ before the maximum. After a ReLU, all responses are nonnegative, so filling with 0 is equivalent and has one advantage: a sentence shorter than the filter width, which has no valid window at all, gets 0 instead of $-\infty$, which would turn into NaN in the next layer.
- **Convolution windows** that overlap padding are not real windows. With right padding, a window of width h starting at position t is valid exactly when position $t + h - 1$ is real.
- **Losses** over token positions, used later for tagging and language modeling, must ignore padded positions; PyTorch's cross-entropy accepts an `ignore_index` for this.

PyTorch's `nn.Embedding(..., padding_idx=0)` keeps the padding row at zero and excludes it from gradient updates. That helps, but it does not make a mean or a max correct by itself. The test is an invariance: adding extra padding must not change any output.

```
import torch
from torch import nn
from torch.nn import functional as F

torch.manual_seed(0)
emb = nn.Embedding(6, 4, padding_idx=0)            # ID 0 is reserved for padding
seqs = [torch.tensor([3, 5, 2]), torch.tensor([4, 1])]
x = nn.utils.rnn.pad_sequence(seqs, batch_first=True, padding_value=0)
print(x.tolist())                                   # [[3, 5, 2], [4, 1, 0]]

# A lookup is a one-hot row times the table.
one_hot = F.one_hot(x, num_classes=6).float()
assert torch.allclose(one_hot @ emb.weight, emb(x))

def masked_mean(ids):
    mask = (ids != 0).unsqueeze(-1).float()
    return (emb(ids) * mask).sum(1) / mask.sum(1).clamp(min=1)

longer = F.pad(x, (0, 3), value=0)                  # three more padding columns
print("masked mean unchanged:", torch.allclose(masked_mean(x), masked_mean(longer)))
print("plain mean unchanged: ", torch.allclose(emb(x).mean(1), emb(longer).mean(1)))
print("padding row:", emb.weight[0].tolist())
```

The run prints:

```
[[3, 5, 2], [4, 1, 0]]
masked mean unchanged: True
plain mean unchanged:  False
padding row: [0.0, 0.0, 0.0, 0.0]
```

The padding row is zero, yet the plain mean still changes, because its denominator counts padding. Add this kind of invariance check to any new encoder.

Batching has practical consequences beyond correctness. Padding wastes computation, so training code often groups sequences of similar length into the same batch (**bucketing**). Very long inputs are truncated to a maximum length; record that limit, because text beyond it is invisible to the model, and decide whether to keep the beginning, the end or both. Shuffle batches every epoch so the model does not see the same order of examples, and evaluate with the model in evaluation mode so that dropout is switched off.

Regularization for small text classifiers

Plain version. A neural text model has enough parameters to memorize a small training set, including its labeling mistakes. Regularization makes memorization harder, and a validation set tells us when to stop.

The tools are the general ones from ML 02, applied at specific places in a text model:

- **Dropout** sets each unit of a layer to zero with probability p during training and rescales the rest, so that no unit can rely on a particular partner [12]. In text CNNs it is usually applied to the pooled feature vector, as in Kim's model [10]. PyTorch's `nn.Dropout(p)` uses the convention that p is the probability of *dropping* a unit.
- **Word dropout** removes entire tokens from the input during training, which encourages the model not to depend on any single word [6].
- **Weight decay** adds $\frac{\lambda}{2}\|\theta\|_2^2$ to the loss, the same penalty as in the logistic objective above. The embedding table is often the largest parameter block, so whether it is decayed or frozen matters.
- **Max-norm constraints** rescale a weight vector whenever its Euclidean norm exceeds a limit, as in Kim's output layer [10].
- **Early stopping** keeps the parameters from the epoch with the best validation loss. It is a regularizer too: it limits how far optimization can move toward memorizing the training set.
- **Freezing pretrained embeddings** removes the largest block of trainable parameters entirely, at the cost of not adapting word meanings to the task.

Every one of these choices is a hyperparameter selected on validation data. Change one at a time and report variation across random seeds before concluding that a choice helps; Zhang and Wallace emphasize exactly this for text CNNs [11].

A complete PyTorch experiment

Plain version. We now compare four classifiers on one invented review task in which order genuinely matters, read their training curves and inspect their mistakes by category.

Each synthetic review is either one clause, "the *aspect* is (*not*) *adjective*," or two clauses joined by "but." The label is positive when the deciding clause is positive: a negation flips the adjective, and in a "but" review the second clause decides. Training has 300 reviews with 10% of labels deliberately flipped to imitate annotation noise; validation (300) and test (1,000) labels are clean. Every test review is tagged as `plain`, `negated` or `contrast` so that errors can be counted by category.

```
import copy, random
import numpy as np
import torch
from torch import nn
from torch.nn import functional as F
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression

torch.set_num_threads(1)
ASPECTS = ["battery", "screen", "camera", "speaker", "case", "charger"]
POS, NEG = ["good", "great", "solid", "bright"], ["bad", "weak", "noisy", "dull"]

def clause(rng):
    aspect, positive = rng.choice(ASPECTS), rng.random() < 0.5
    negated = rng.random() < 0.4
    words = ["the", aspect, "is"] + ["not"] * negated + [rng.choice(POS if positive else
     ↪  NEG)]
```

```
    return words, int(positive != negated), negated

def example(rng):                       # label of "A but B" is the label of B
    words, label, negated = clause(rng)
    if rng.random() < 0.5:
        second, label, _ = clause(rng)
        return " ".join(words + ["but"] + second), label, "contrast"
    return " ".join(words), label, "negated" if negated else "plain"

def make(n, seed):
    rng = random.Random(seed)
    return [example(rng) for _ in range(n)]

train, val, test = make(300, 1), make(300, 2), make(1000, 3)
flip = random.Random(4)                 # 10% training-label noise; val/test stay clean
train = [(t, 1 - y if flip.random() < 0.1 else y, c) for t, y, c in train]
y_test, kind = np.array([r[1] for r in test]), np.array([r[2] for r in test])

def report(name, pred):
    parts = {k: round(float((pred[kind == k] == y_test[kind == k]).mean()), 3)
             for k in ["plain", "negated", "contrast"]}
    print(f"{name:12s} test acc {np.mean(pred == y_test):.3f}", parts)

for ngrams in [(1, 1), (1, 2)]:
    vec = TfidfVectorizer(ngram_range=ngrams, token_pattern=r"\S+")
    clf = LogisticRegression(C=10.0, max_iter=1000)
    clf.fit(vec.fit_transform([r[0] for r in train]), [r[1] for r in train])
    report(f"tfidf {ngrams}", clf.predict(vec.transform([r[0] for r in test])))

vocab = ["<pad>", "<unk>"] + sorted({w for r in train for w in r[0].split()})
ids = {w: i for i, w in enumerate(vocab)}

def encode(rows):                       # right-pad every sequence with ID 0
    seqs = [torch.tensor([ids.get(w, 1) for w in r[0].split()]) for r in rows]
    return (nn.utils.rnn.pad_sequence(seqs, batch_first=True, padding_value=0),
            torch.tensor([r[1] for r in rows]))

class MeanMLP(nn.Module):               # embedding -> masked mean -> MLP
    def __init__(self, V, D=16, H=32, p=0.3):
        super().__init__()
        self.emb = nn.Embedding(V, D, padding_idx=0)
        self.mlp = nn.Sequential(nn.Linear(D, H), nn.ReLU(), nn.Dropout(p), nn.Linear(H, 2))
    def forward(self, x):
        m = (x != 0).unsqueeze(-1).float()
        return self.mlp((self.emb(x) * m).sum(1) / m.sum(1).clamp(min=1))

class TextCNN(nn.Module):               # embedding -> 1-D conv -> masked max -> linear
    def __init__(self, V, D=16, widths=(2, 3, 4), maps=16, p=0.3):
        super().__init__()
        self.emb = nn.Embedding(V, D, padding_idx=0)
        self.convs = nn.ModuleList(nn.Conv1d(D, maps, h) for h in widths)
        self.drop, self.out = nn.Dropout(p), nn.Linear(maps * len(widths), 2)
    def forward(self, x):
        e, real = self.emb(x).transpose(1, 2), x != 0        # e: (batch, D, T)
        pooled = []
        for conv in self.convs:
```

```
            c = torch.relu(conv(e))                          # (batch, maps, T-h+1)
            valid = real[:, conv.kernel_size[0] - 1:].unsqueeze(1)  # window ends on a real
             ↪ token
            # ReLU outputs are >= 0, so filling invalid windows with 0 never wins the max,
            # and a text shorter than the filter gets 0 instead of -inf (which would give
             ↪ NaN).
            pooled.append(c.masked_fill(~valid, 0.0).max(dim=2).values)
        return self.out(self.drop(torch.cat(pooled, dim=1)))

def fit(model, weight_decay, epochs=80, batch=32):
    (xt, yt), (xv, yv) = encode(train), encode(val)
    opt = torch.optim.Adam(model.parameters(), lr=0.01, weight_decay=weight_decay)
    best, history = (float("inf"), None, 0), []
    for epoch in range(1, epochs + 1):
        model.train()
        for b in torch.randperm(len(yt)).split(batch):
            opt.zero_grad()
            F.cross_entropy(model(xt[b]), yt[b]).backward()
            opt.step()
        model.eval()
        with torch.no_grad():
            history.append((F.cross_entropy(model(xt), yt).item(),
                            F.cross_entropy(model(xv), yv).item()))
        if history[-1][1] < best[0]:                         # early stopping on val loss
            best = (history[-1][1], copy.deepcopy(model.state_dict()), epoch)
    model.load_state_dict(best[1])
    return history, best[2]

x_test, _ = encode(test)
for name, build, wd in [("mean+MLP", lambda: MeanMLP(len(vocab)), 1e-4),
                        ("CNN", lambda: TextCNN(len(vocab)), 1e-4),
                        ("CNN no-reg", lambda: TextCNN(len(vocab), p=0.0), 0.0)]:
    torch.manual_seed(0)
    model = build()
    history, best_epoch = fit(model, wd)
    with torch.no_grad():
        pred = model(x_test).argmax(1).numpy()
    print(f"{name:12s} last train/val loss {history[-1][0]:.3f}/{history[-1][1]:.3f}, "
          f"best epoch {best_epoch}")
    report(name, pred)
    if name == "CNN":
        cnn_pred = pred

wrong = np.flatnonzero(cnn_pred != y_test)
print("CNN errors by kind:", {k: int((kind[wrong] == k).sum()) for k in ["plain", "negated",
 ↪ "contrast"]})
for i in wrong[:4]:
    print(" ", y_test[i], "<-", test[i][0])
```

The whole script runs in a few seconds on one CPU thread and prints:

```
tfidf (1, 1) test acc 0.546 {'plain': 0.651, 'negated': 0.348, 'contrast': 0.555}
tfidf (1, 2) test acc 0.880 {'plain': 1.0, 'negated': 1.0, 'contrast': 0.77}
```

```
mean+MLP     last train/val loss 0.269/0.464, best epoch 36
mean+MLP     test acc 0.797 {'plain': 0.973, 'negated': 0.945, 'contrast': 0.645}
CNN          last train/val loss 0.144/0.431, best epoch 30
CNN          test acc 0.898 {'plain': 0.983, 'negated': 0.945, 'contrast': 0.833}
CNN no-reg   last train/val loss 0.092/0.444, best epoch 10
CNN no-reg   test acc 0.910 {'plain': 1.0, 'negated': 0.928, 'contrast': 0.852}
CNN errors by kind: {'plain': 5, 'negated': 10, 'contrast': 87}
  0 <- the camera is not dull but the battery is weak
  0 <- the camera is noisy but the screen is bad
  1 <- the charger is solid but the battery is solid
  1 <- the charger is solid but the charger is bright
```

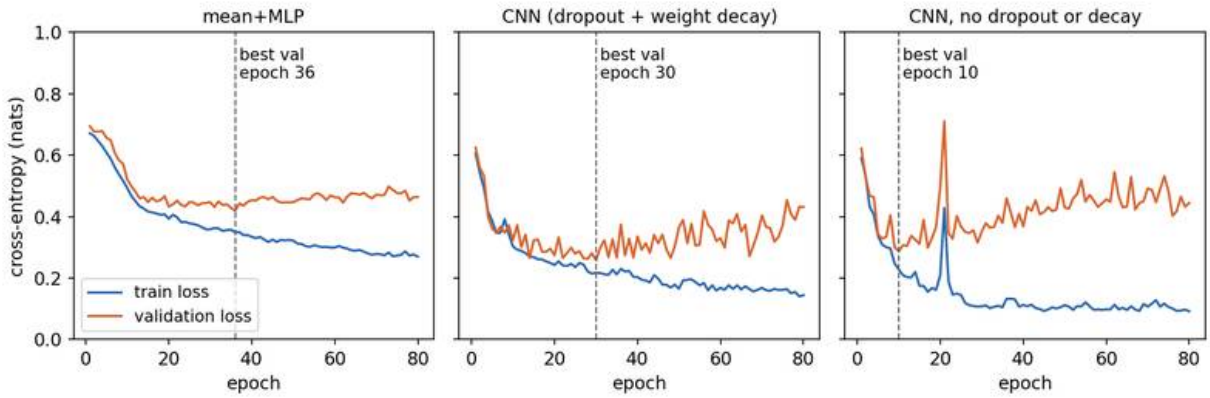


Figure 4 — Training (blue) and validation (orange) loss for the three neural models in the experiment, seed 0. The dashed line marks the epoch kept by early stopping.

Read the results by category

Unigram TF–IDF fails on negation by construction. In this data, "not" appears equally often in positive and negative reviews, and each adjective is positive or negative depending on whether "not" is present. A linear model over word presence cannot represent that interaction, so negated reviews score 0.348, worse than chance.

Bigrams fix negation, not contrast. The pair "not good" is a single feature, so the bigram model is perfect on plain and negated reviews. On contrast reviews it reaches 0.77: it sees both clauses' bigrams and has no reliable way to know which clause comes after "but."

The pooled MLP solves the interaction but not order. Its hidden layer can combine "not" with an adjective, so negated reviews reach 0.945 even though the input is an unordered mean. On contrast reviews it scores 0.645 because mean pooling makes "A but B" and "B but A" identical.

The CNN does best on contrast but is limited by its width. With filters up to width 4 it reaches 0.833 on contrast reviews. All but 15 of its 102 errors are contrast reviews, and the listed errors show why: in "the charger is solid but the battery is solid," the deciding adjective is four positions after "but," so no single window contains both. The filters must rely on indirect cues. Exercise 3 asks you to add a width-5 filter and test that explanation.

Read the curves

All three panels show the same pattern: training loss keeps falling while validation loss stops improving and then rises. Training labels contain 10% flipped labels, so a training loss far below that noise level means the model is memorizing mistakes. The unregularized CNN does this fastest: its validation loss bottoms out at epoch 10 and is noticeably worse by epoch 80. Dropout and weight decay slow that drift, and the regularized CNN's best validation loss comes later.

On test accuracy, however, the unregularized CNN (0.910) and the regularized CNN (0.898) are close once early stopping has chosen each model's epoch. With one seed and a difference of about one percentage point, this run cannot rank them. The defensible conclusion is narrower: early stopping on a clean validation set is essential here, and the regularized model is less sensitive to *when* it is stopped. A real comparison would repeat the experiment over several seeds and report the spread [11].

Error analysis as a habit

The category counts were possible only because each test example carried a tag describing *why* it might be hard. Build such tags into evaluation data from the start: negation, contrast, rare words, length, source. Then read individual errors and ask at which stage the needed evidence was lost. If mean pooling makes two inputs identical, more labels cannot help that model; if the CNN's windows are too short, a wider filter or a different encoder is the fix; if an annotation is wrong, the fix is in the data.

Evaluate errors and probabilities separately

Plain version. A classifier can choose the right label while giving unreliable probabilities. Evaluate the decision and the uncertainty estimate as different properties.

For one class treated as positive:

$$P = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}, \qquad R = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}, \qquad F_1 = \frac{2PR}{P + R}.$$

Here P is precision, R is recall, TP counts true positives, FP counts false positives, FN counts false negatives and F_1 is their harmonic mean. A zero denominator requires an explicitly chosen convention. Macro F1 averages per-class F1 values equally; support-weighted averaging gives more influence to common classes. The metric documentation specifies these choices [13].

Macro averaging makes failures on a rare category more visible, but it does not encode the actual cost of those failures. Missing an urgent ticket and misrouting a routine receipt can have different consequences. Write a cost policy separately rather than treating a convenient score as a complete utility measure.

Calibration asks whether predicted confidence agrees with observed correctness over comparable examples. A model can preserve its argmax decisions while changing confidence. Guo et al. study

this distinction and post-training temperature scaling [14]. Calibration parameters must be fit on held-out development data, with enough samples to estimate probability behavior. Neural classifiers trained to near-zero training loss are a common source of overconfident probabilities, so check calibration whenever the model changes.

For selective prediction:

$$\text{coverage} = \frac{1}{n}\sum_{i=1}^{n} a_i, \qquad \text{risk}_{\text{accepted}} = \frac{\sum_{i=1}^{n} a_i \mathbf{1}[\hat{y}_i \neq y_i]}{\sum_{i=1}^{n} a_i}.$$

Here n is evaluated-example count, $a_i \in \{0, 1\}$ indicates acceptance of example i, $\hat{y}_i$ is its prediction, y_i is its reference label, and $\mathbf{1}[\cdot]$ is one when its condition holds and zero otherwise. Accepted risk is undefined when no example is accepted. A risk-versus-coverage curve describes the tradeoff over thresholds; choose the deployed threshold using validation rather than final test labels.

Imbalance, error analysis and deployment changes

Plain version. An overall score can improve while an important user group gets worse. Inspect the types of mistakes, the rare classes and the circumstances in which they occur.

Create an error table with text, gold label, predicted label, probability, language, length, source group and an analyst's explanation. Useful error categories include annotation ambiguity, missing context, negation, unseen vocabulary, multi-topic input, quotation and genuinely out-of-scope text. These categories suggest different repairs. More training examples can help missing vocabulary; they cannot supply a conversation turn that is never available at prediction time.

Class weighting changes how strongly each training example contributes:

$$\mathcal{L}_{\text{weighted}} = -\frac{1}{n}\sum_{i=1}^{n} \alpha_{y_i} \log p_\theta(y_i \mid d_i).$$

Here $\mathcal{L}_{\text{weighted}}$ is weighted cross-entropy, n is training-example count, i indexes examples, d_i is the document, y_i is its annotated class, $\alpha_{y_i} > 0$ is the chosen weight for that class and p_θ is the classifier distribution. The weighting policy changes the training objective; it does not manufacture missing linguistic variety.

Weighting, oversampling and threshold changes answer different problems. Weighting changes parameter learning. Oversampling changes the presentation frequency of examples, sometimes repeatedly exposing the same few minority examples. Thresholding changes the action policy after scores are computed. Hagiwara's development discussion covers imbalance and training choices [1]. Evaluate any intervention on a realistic prevalence distribution rather than on an artificially balanced final test.

Deployment shifts can affect vocabulary, label prevalence and the relationship between text and labels. A new feature release may produce a new technical-error vocabulary; a holiday may increase delivery messages; a changed routing policy may redefine labels. Monitor these separately.

A rising fraction of empty feature vectors, or of unknown-word IDs in a neural model, suggests representation coverage; a routing-policy change requires updating the annotation contract.

Keep model and preprocessing artifacts together. The vocabulary order, IDF weights or embedding table, classifier class order, maximum length, probability calibration and decision threshold are a single inference package. Reloading weights with another vocabulary silently changes meanings. Store a versioned label schema and a few deterministic contract examples with the package so that serving can check its assumptions.

Modern view

Survey perspective. Goldberg's primer organizes neural NLP around feed-forward, convolutional, recurrent and recursive networks, input encoding and computation graphs [5]; it remains a compact map of the models in this lesson. Minaee et al. review more than 150 deep-learning text classifiers and more than 40 datasets, from feed-forward and convolutional models through recurrent, attention-based and pretrained models, and compare reported benchmark results [15]. Read such tables as evidence about specific datasets and protocols, not as a ranking that transfers to an arbitrary label schema.

Zhao et al. organize their LLM survey around pretraining, adaptation, use and evaluation [16]. For classification, this yields a useful comparison among fixed representations with a learned head, task-adapted encoders and instruction-based prediction. Chang et al. organize evaluation by what, where and how to measure [17]. Their perspective warns against treating one familiar benchmark as evidence that an arbitrary label schema works.

Robustness perspective. Sperduti and Moreo's 2025 misspelling survey reviews sources of spelling variation, correction approaches and the effect on downstream tasks and modern language models [18]. Its practical implication here is to include realistic spelling variation in evaluation, then distinguish normalization failures from classifier failures. Character features, subword embeddings and contextual representations offer different responses; none should be assumed robust without a task-specific comparison.

Practice in 2023–2026. Pretrained Transformer encoders fine-tuned with a classification head, covered in NLP 11, are the usual starting point when labeled data and compute allow. Instruction-following language models broaden the available ways to specify classification tasks [16]. A prompted model can interpret a textual label description without a separately trained classifier head. That shifts part of the configuration into the prompt, label wording and output parser. Treat those artifacts as fitted design choices: fix them before testing, constrain outputs to the allowed schema and measure abstention and malformed responses.

The models in this lesson keep a role. TF–IDF with logistic regression and small convolutional or averaging networks are cheap to train, fast to serve and easy to inspect. They are the baselines that tell you whether a larger model's gain is real, and they are often adequate when labels have clear lexical cues.

An application can also use a larger model to suggest labels for human review, then train a smaller classifier. This changes the annotation process. Generated labels can contain systematic mistakes

or reproduce the teacher's preferences. Audit a human-checked subset and keep synthetic supervision distinct from independent evaluation labels. The possibility of producing more labels is not proof of better supervision.

Open problems. Label interpretation can drift between annotators and model versions. Low-resource languages may have too little independent evaluation data. Multi-topic messages resist a single-label schema. Calibration can deteriorate after a domain change. Embedding similarity can hide contradiction. These questions need explicit challenge sets and continuing evaluation, not an architecture ranking with all context removed.

Key takeaways

- The annotation contract determines the prediction problem and the available actions; fit vocabulary, IDF weights and any learned representation on training text only.
- A sparse linear baseline is strong, cheap and diagnostic; bigrams capture local negation but treat every phrase as unrelated to every other.
- An embedding layer is a learned lookup table; mean pooling plus a nonlinear MLP can learn word interactions but cannot see order.
- The neural probabilistic language model concatenates learned word vectors and generalizes to unseen histories by sharing evidence between similar words.
- A 1-D convolutional classifier detects soft n-gram patterns anywhere in the text; max pooling discards position, and filter width limits the distance it can connect.
- Padding requires explicit masks for means, maxima, windows and losses; adding padding must not change any output.
- Regularization and early stopping matter most when labels are noisy; compare configurations over several seeds.
- Evaluate class decisions, probability quality, coverage and per-category errors separately.

Exercises

1. **A label contract.** Define a multi-label version of the support task. Write rules for "delivery charge," a quoted complaint and an empty input. Explain which output activation and loss change.

   Hint

   Independent label probabilities can allow two routes. Include an explicit rule for when no label is justified rather than interpreting every low score as a negative annotation.

2. **Train-only features.** Introduce a word that occurs only in the final test set. Inspect the fitted TF–IDF vocabulary and transformed test vector, and the ID it receives in the neural experiment's vocabulary. Compare the result with incorrectly fitting features on all text.

Hint

The correct vocabulary excludes the test-only word; in the neural model it maps to `<unk>`. Examine document-frequency changes as well as the vocabulary size; leakage can alter weights even when all words already exist.

3. **Test the width explanation.** In the PyTorch experiment, change the CNN's widths to `(2, 3, 4, 5)` and rerun. Does contrast accuracy change? Then run the original and the modified model over five seeds and report mean and range before drawing a conclusion.

Hint

Count the positions in "but the battery is solid." A width-5 window can contain "but" and the deciding adjective together. Keep everything except the widths fixed, and remember that a single-seed difference can be noise.

4. **Masking bugs on purpose.** Remove the mask from `MeanMLP` (use a plain `.mean(1)`) and from `TextCNN` (pool over all windows). Pad the test batch with five extra columns and compare predictions with the masked versions. Which model changes, and why?

Hint

The padding embedding is zero, so a plain mean shrinks toward zero as padding grows. For the CNN, a window that overlaps padding still has a nonzero bias and can win the max.

5. **A neural language model by hand.** For the NPLM with V words, $n-1$ history words, m features per word and h hidden units, count the parameters in C, H, $\mathbf{d}$, U, $\mathbf{b}$ and W. Which term dominates when V is large, and how does the count change if you remove the direct connections?

Hint

Write the shape of each matrix from the equation. The output layer and the embedding table both scale with V; the hidden layer does not.

6. **Selective decisions.** Run the support-message experiment with several validation cost policies. Plot coverage and accepted error on validation, select one threshold and then evaluate it once on test.

Hint

If no examples are accepted, the accepted-error denominator is zero. Report an undefined value, not perfect accuracy.

References

1. M. Hagiwara, *Real-World Natural Language Processing: Practical Applications with Deep Learning*, Manning, 2021; Chapters 2–4, 7 and 10–11. publisher
2. S. McRoy, *Principles of Natural Language Processing*, open textbook, 2021; Chapter 6, "Benchmark Tasks for Language Modelling." chapter
3. Scikit-learn developers, "Text feature extraction," scikit-learn documentation, current online edition. documentation
4. Scikit-learn developers, "Logistic regression," scikit-learn documentation, current online edition. documentation
5. Y. Goldberg, "A primer on neural network models for natural language processing," *Journal of Artificial Intelligence Research* 57, 2016. paper
6. M. Iyyer, V. Manjunatha, J. Boyd-Graber and H. Daumé III, "Deep unordered composition rivals syntactic methods for text classification," ACL-IJCNLP, 2015. paper
7. A. Joulin, E. Grave, P. Bojanowski and T. Mikolov, "Bag of tricks for efficient text classification," EACL, 2017. paper
8. Y. Bengio, R. Ducharme, P. Vincent and C. Jauvin, "A neural probabilistic language model," *Journal of Machine Learning Research* 3, 2003. paper
9. R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu and P. Kuksa, "Natural language processing (almost) from scratch," *Journal of Machine Learning Research* 12, 2011. paper
10. Y. Kim, "Convolutional neural networks for sentence classification," EMNLP, 2014. paper
11. Y. Zhang and B. Wallace, "A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification," IJCNLP, 2017. paper
12. N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research* 15, 2014. paper
13. Scikit-learn developers, "Classification metrics," scikit-learn documentation, current online edition. documentation
14. C. Guo, G. Pleiss, Y. Sun and K. Q. Weinberger, "On calibration of modern neural networks," ICML, 2017. paper
15. S. Minaee, N. Kalchbrenner, E. Cambria, N. Nikzad, M. Chenaghlu and J. Gao, "Deep learning-based text classification: a comprehensive review," *ACM Computing Surveys* 54(3), 2021. survey
16. W. X. Zhao et al., "A survey of large language models," *Frontiers of Computer Science*, 2026; online manuscript updated March 2026. survey
17. Y. Chang et al., "A survey on evaluation of large language models," *ACM Transactions on Intelligent Systems and Technology*, 2024. survey
18. G. Sperduti and A. Moreo, "Misspellings in natural language processing: a survey," arXiv:2501.16836, 2025. survey

NLP 04 • Lexical Semantics and Word Embeddings

From word senses, lexical relations and WordNet to distributional vectors: PPMI, Word2Vec, GloVe and fastText, how to evaluate them, the biases they absorb, sentence embeddings and the step to contextual representations.

Online edition: https://blog.shyandram.dev/en/learn/nlp/lexical-semantics-word-embeddings/

What you'll learn

- Distinguish word forms, lemmas and senses, and name the main lexical relations: synonymy, antonymy, hypernymy and meronymy, as organized in WordNet.
- Explain the distributional hypothesis, what shared contexts can reveal about words and what they cannot establish.
- Construct a word–context matrix, turn positive PMI into compact vectors, and derive CBOW, Skip-gram, negative sampling, GloVe and fastText objectives.
- Evaluate embeddings with similarity and analogy tests while recognizing their caveats, and measure an association bias.
- Pool word vectors into sentence representations, train sentence embeddings contrastively, and explain what changes with contextual embeddings.

The big picture

A word ID identifies a vocabulary entry. Its numerical value says nothing about what the word does in language. An **embedding** assigns that entry a learned vector, allowing words to share useful statistical patterns. The central question is not simply how to make a vector smaller. It is which evidence and objective should determine its coordinates.

> **Explain like I'm five**
>
> Imagine learning about unfamiliar tools by watching where people use them. Two tools that repeatedly appear beside wood, nails and a workbench may serve related jobs. You can draw a map from those observations without anyone naming the tools. But appearing at the same workbench does not make them interchangeable: a hammer and pliers still do different things. Word vectors learn a similar map from words' surroundings.

Linguists describe word meaning with **lexical semantics**: senses, and relations such as synonymy and hypernymy, recorded by hand in resources like WordNet. Distributional methods learn vectors from text instead. This lesson connects the two views. Hagiwara develops word and document embeddings as reusable features for practical applications [1]. McRoy distinguishes lexical

similarity from language tasks such as sentiment and entailment [2]. Here we develop the representation itself, and we keep asking which lexical relation a vector geometry actually captures.

NLP 03 learned word vectors as a by-product of a task: an embedding layer inside a classifier, and the table C inside the neural probabilistic language model of Bengio et al. [3]. Word2Vec makes the vectors themselves the goal and uses nearby words, not a following word, as the prediction target. These are different prediction contracts:

Question	Neural probabilistic language model	Static Word2Vec embedding
What is observed?	An ordered history of $n-1$ tokens and the token that follows	A center word and selected neighbors on both sides
What is predicted?	A normalized distribution over the next token	A center from neighbors, or individual neighbors from a center
Is left-to-right generation the purpose?	Yes; it is a language model	Basic window training is not an ordered generator
What is reused afterward?	The whole network; word vectors are a by-product	A learned vector table

Continuous in CBOW means that its internal representation has real-valued coordinates. It does not mean that the text is continuous, that words occur without boundaries, or that every word sense receives its own vector. By the end, we will connect the count-based and prediction-based constructions and identify exactly which information each retains. Count-based language models and their evaluation come later, in NLP 05.

Lexical semantics: senses and relations

Plain version. A written word can carry several meanings, and meanings relate to each other in structured ways: some are near-equivalents, some are opposites, some are kinds of others. Embeddings will later blur several of these relations together, so name them first.

Word forms, lemmas and senses

A **word form** is what appears in text (`banks`, `banked`); a **lemma** is the dictionary entry that groups inflected forms (`bank`). A **sense** is one meaning of a lemma. `bank` has at least two unrelated senses, the slope beside a river and the financial institution; this is **homonymy**. `window` has related senses, an opening in a wall and a region of a screen; this is **polysemy**. NLP 02 discusses how context and composition select a sense. For representation, the key point is that a vocabulary entry is a form or lemma, not a sense: a table with one row per entry must merge every sense of that entry.

Relations between senses

Lexical relations hold between senses, not between spelling strings:

- **Synonymy**: two senses that can substitute for each other in many contexts with little change in meaning, such as `slope` and `incline`. Exact synonymy is rare; register, collocation and connotation usually differ.
- **Antonymy**: senses that are opposite along one scale or dimension, such as `hot` and `cold`. Antonyms are otherwise very similar: they describe the same property and occur in the same kinds of sentences.
- **Hyponymy and hypernymy**: an is-a relation. `poodle` is a hyponym of `dog`, and `dog` is a hypernym of `poodle`. The relation is asymmetric and usually transitive, forming a hierarchy.
- **Meronymy**: a part–whole relation, such as `wheel` and `car`.

Two further notions matter for evaluation. **Similarity** asks whether two concepts share properties (`cup` and `mug`); **relatedness** or association asks whether they tend to occur together (`cup` and `coffee`). A related pair need not be similar at all.

WordNet: a hand-built lexical database

WordNet organizes English nouns, verbs, adjectives and adverbs into **synsets**, sets of synonymous lemmas that express one concept, each with a short gloss, connected by relations such as hypernymy, meronymy and antonymy [4]. Looking up a lemma returns its synsets, one per sense. Walking hypernym links from a synset reaches increasingly general concepts, which lets a program measure how far apart two senses are in the hierarchy.

The following example builds a miniature database in the same style, by hand, to show the mechanics without downloading WordNet itself. Its entries are invented for this lesson.

```
# A hand-built miniature lexical database in the style of WordNet (not WordNet data).
synsets = {
    "bank.n.slope":   {"lemmas": ["bank"], "gloss": "sloping land beside a river",
     ↪  "hypernym": "slope.n"},
    "bank.n.finance": {"lemmas": ["bank"], "gloss": "an institution that holds deposits",
     ↪  "hypernym": "institution.n"},
    "slope.n":        {"lemmas": ["slope", "incline"], "gloss": "a surface at an angle",
     ↪  "hypernym": "landform.n"},
    "landform.n":     {"lemmas": ["landform"], "gloss": "a natural feature of land",
     ↪  "hypernym": "entity.n"},
    "institution.n":  {"lemmas": ["institution"], "gloss": "an organization with a public
     ↪  role", "hypernym": "organization.n"},
    "school.n":       {"lemmas": ["school"], "gloss": "an institution for teaching",
     ↪  "hypernym": "institution.n"},
    "organization.n": {"lemmas": ["organization"], "gloss": "a group with a purpose",
     ↪  "hypernym": "entity.n"},
    "entity.n":       {"lemmas": ["entity"], "gloss": "anything that exists", "hypernym":
     ↪  None},
}

def senses(word):
    return [s for s, v in synsets.items() if word in v["lemmas"]]

def path_to_root(s):
```

```
    path = [s]
    while synsets[path[-1]]["hypernym"]:
        path.append(synsets[path[-1]]["hypernym"])
    return path

def path_similarity(a, b):            # 1 / (1 + edges on the shortest is-a path)
    pa, pb = path_to_root(a), path_to_root(b)
    common = next(s for s in pa if s in pb)
    return 1 / (1 + pa.index(common) + pb.index(common))

print("senses of bank:", senses("bank"))
print("synonyms in slope.n:", synsets["slope.n"]["lemmas"])
print("hypernym chain:", " -> ".join(path_to_root("bank.n.finance")))
for other in ["school.n", "slope.n"]:
    print(f"path_similarity(bank.n.finance, {other}) = {path_similarity('bank.n.finance',
     ↪  other):.3f}")
```

The run prints:

```
senses of bank: ['bank.n.slope', 'bank.n.finance']
synonyms in slope.n: ['slope', 'incline']
hypernym chain: bank.n.finance -> institution.n -> organization.n -> entity.n
path_similarity(bank.n.finance, school.n) = 0.333
path_similarity(bank.n.finance, slope.n) = 0.167
```

The financial sense of `bank` is closer to `school` (both are institutions) than to `slope`, even though the *other* sense of `bank` is a kind of slope. A sense inventory makes that distinction explicit; a single vector for the string `bank` cannot.

Hand-built resources are precise and interpretable, but they are expensive to build, cover some languages and domains far better than others, lag behind new words and senses, and encode one set of sense distinctions that may not match an application. Distributional vectors make the opposite trade: they are cheap and broad, but their relations are implicit and mixed. Much practical work combines the two, for example by checking embedding neighbors against a lexical resource.

Distributional evidence: contexts are observations

Plain version. Words that appear in similar surroundings can acquire similar vectors. The surroundings provide evidence about usage; they do not directly supply definitions or truth conditions.

Harris argued that differences in meaning correlate with differences in distribution: the environments in which linguistic elements occur [5]. This **distributional hypothesis** is the foundation of every method in the rest of this lesson.

A **word type** is a vocabulary entry, while a **word token** is one occurrence. A static embedding pools evidence across many tokens into one vector for the type. Occurrences of `bank` beside `riv`

`er` and beside `account` therefore influence the same row. The model can learn that the word has mixed usage, but the lookup operation cannot choose a different row for each sense.

Define context before collecting statistics. A radius-two word window contains up to two tokens on each side of the center, excluding the center itself. A shorter window emphasizes immediate surroundings; a larger one includes more distant topic evidence. If direction matters, left and right context positions can be separate features. Our examples deliberately merge the two sides so the model's invariance is visible.

The **distributional assumption** is useful because text provides repeated observations without requiring a semantic label for every word. It is incomplete because shared use includes synonymy, opposition, thematic association and grammatical compatibility. `hot` and `cold` can both appear before `water`; similarity on that evidence does not establish that they express the same property. In the terms of the previous section, distributional similarity mixes synonymy, antonymy, co-hyponymy and relatedness. Turney and Pantel organize these vector-space approaches by the evidence represented in their matrices [6].

Keep corpus, tokenizer and window specification together. Changing word segmentation changes the types whose contexts are counted. An embedding learned on technical incident reports may organize `charge` differently from one learned on electronics manuals. The coordinates are an outcome of those choices, not language-independent measurements of meaning.

Count-based embeddings: inspect the shared-context assumption

Plain version. Count how often each word appears beside each context, then compare that count with a baseline that treats the word and context as independent. An unusually strong association contributes a positive coordinate.

The distributional survey by Turney and Pantel [6] explains this **count-based** route. Let C_{wc} count the observed ordered pairs consisting of center word w and neighboring context word c. The two roles remain distinct even when their vocabularies contain the same words. Define empirical probabilities from the table:

$$N_C = \sum_{w,c} C_{wc}, \qquad P(w,c) = \frac{C_{wc}}{N_C}, \qquad P(w) = \sum_{c} P(w,c), \qquad P(c) = \sum_{w} P(w,c).$$

Here N_C is the total center–context pair count, C_{wc} is one cell count, $P(w,c)$ is its empirical joint probability, $P(w)$ is the center marginal, and $P(c)$ is the context marginal. The sums range over the corresponding vocabularies. A token may contribute several pairs, so N_C is not the number of tokens.

Pointwise mutual information (PMI) compares the empirical joint probability with the independence baseline. **Positive PMI (PPMI)** retains only associations stronger than that baseline:

$$\mathrm{PMI}(w,c) = \log \frac{P(w,c)}{P(w)P(c)}, \qquad \mathrm{PPMI}(w,c) = \max(\mathrm{PMI}(w,c), 0).$$

Here $\mathrm{PMI}(w,c)$ is the log association score for one pair, $\mathrm{PPMI}(w,c)$ is its nonnegative version, and

log is the natural logarithm. The probabilities are those defined above. A pair with zero joint count and positive marginals has PMI tending to negative infinity and PPMI zero; an entirely unobserved row requires a separate policy.

A frequent context does not automatically receive a large association score. It must occur near this center more often than its overall frequency predicts. Conversely, a rare pair can have high PMI on little evidence. Inspect counts alongside scores; removing negative associations and keeping only positive ones also discards information rather than making every remaining association reliable.

The PPMI matrix still has one coordinate per context. A truncated **singular value decomposition** (**SVD**) compresses it:

$$M = A\Sigma B^\top, \qquad M_D = A_D\Sigma_D B_D^\top, \qquad E = A_D\Sigma_D.$$

Here M is the PPMI matrix, A and B contain its left and right singular vectors, Σ is the diagonal matrix of singular values, D is the retained dimension, and the subscript D selects the first D singular components. M_D is the resulting low-rank reconstruction, and E supplies the center-word vectors used below. This is the convention implemented in the code; other distributions of singular-value scaling produce different vector geometries.

SVD identifies shared patterns across rows rather than assigning a semantic name to each coordinate. The compression cannot preserve every cell when the original matrix has higher rank than D. The following synthetic corpus lets us inspect both the construction and a failure of interpreting cosine similarity as synonymy.

```
import numpy as np

sentences = """the cat drinks milk . the dog drinks water . a cat chases a mouse .
a dog chases a cat . the kitten drinks milk . the puppy drinks water .
a kitten chases a ball . a puppy chases a ball . the car needs fuel .
the truck needs fuel . a car drives on the road . a truck drives on the road .""".split(".")
tokens = [s.split() for s in sentences if s.strip()]
vocab = sorted({w for s in tokens for w in s})
idx = {w: i for i, w in enumerate(vocab)}

# 1) Co-occurrence counts within a +-2 word window.
C = np.zeros((len(vocab), len(vocab)))
for s in tokens:
    for i, w in enumerate(s):
        for j in range(max(0, i - 2), min(len(s), i + 3)):
            if j != i:
                C[idx[w], idx[s[j]]] += 1

# 2) Positive pointwise mutual information.
total = C.sum()
pw, pc = C.sum(1, keepdims=True) / total, C.sum(0, keepdims=True) / total
with np.errstate(divide="ignore"):
    pmi = np.log((C / total) / (pw * pc))
ppmi = np.maximum(pmi, 0)

# 3) Truncated SVD gives dense 8-dimensional embeddings.
U, S, _ = np.linalg.svd(ppmi)
```

```
E = U[:, :8] * S[:8]
E /= np.linalg.norm(E, axis=1, keepdims=True)

def nearest(word, k=3):
    sims = E @ E[idx[word]]
    order = [vocab[i] for i in np.argsort(-sims) if vocab[i] != word]
    return [(w, round(float(sims[idx[w]]), 2)) for w in order[:k]]

for w in ["cat", "puppy", "truck", "milk"]:
    print(w, "->", nearest(w))
```

Output:

```
cat -> [('kitten', 0.96), ('puppy', 0.88), ('dog', 0.88)]
puppy -> [('dog', 1.0), ('kitten', 0.98), ('cat', 0.88)]
truck -> [('car', 1.0), ('needs', 0.6), ('the', 0.47)]
milk -> [('water', 0.97), ('a', 0.62), ('chases', 0.51)]
```

Animals cluster with animals and vehicles with vehicles, learned from twelve sentences and no labels. Note "milk" and "water" are nearly identical: both are things that are drunk in this corpus. Embedding similarity measures **shared contexts**, which often but not always matches similar meaning; antonyms such as "hot" and "cold" also share contexts.

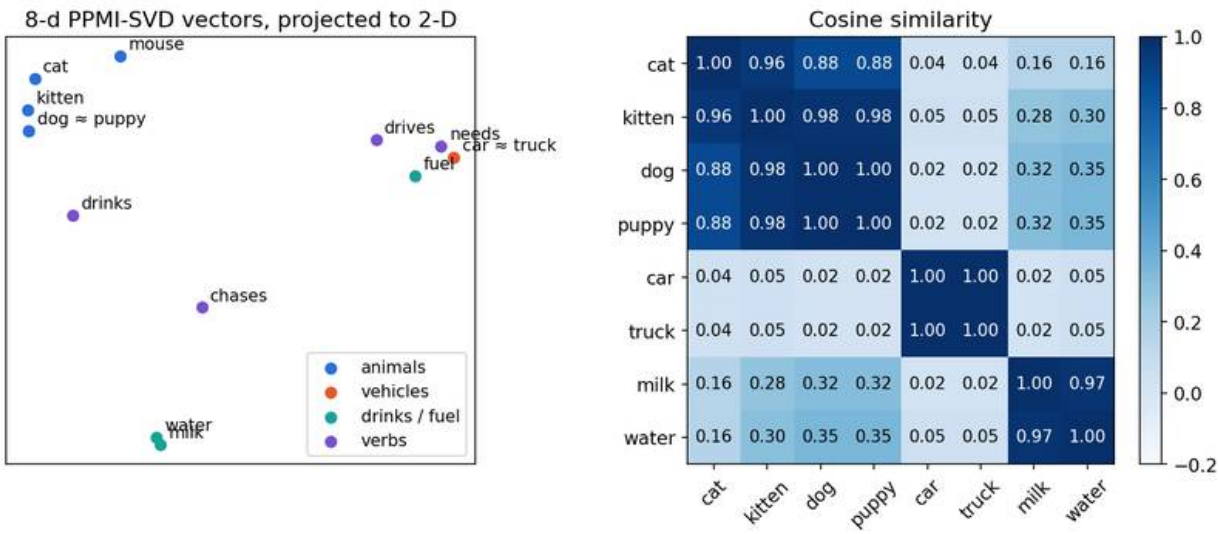


Figure 1 — Vectors from the code above. Cosine similarity is high within groups that share contexts.

Word2Vec: learn a word by predicting its neighbors

Plain version. Words used in similar surroundings often serve similar roles. Word2Vec turns that observation into a small prediction problem: learn vectors that help recover a word from its

neighbors, or its neighbors from the word. The vectors are learned parameters, not hand-written lists of meanings. The two basic architectures are **continuous bag of words** (**CBOW**) and **Skip-gram** [7].

Turn a sentence into training examples

Start with the invented sentence `small red robots move slowly`. A **context window** chooses nearby positions; radius one means the word immediately before and after the center. For this example, use only interior centers, so every window has two neighbors:

Center	Context	CBOW training example	Skip-gram training examples
`red`	`small`, `robots`	`[small, robots] → red`	`red → small`; `red → robots`
`robots`	`red`, `move`	`[red, move] → robots`	`robots → red`; `robots → move`
`move`	`robots`, `slowly`	`[robots, slowly] → move`	`move → robots`; `move → slowly`

CBOW combines several context words to predict one center word. Skip-gram uses one center word to predict each context word separately. Skip-gram does not generate an ordered sentence, and "skip" does not mean remove random words. The target may come from either side of the center. At sentence boundaries, an implementation can use shorter windows; our code omits those centers to keep the tensor shapes simple.

The training labels come from the text itself. Nobody must annotate `robots` with a semantic category before these examples can be created. Repeated occurrences create repeated evidence. The chosen window determines what relationships the model is encouraged to capture: changing it changes the training task, even when the vocabulary stays the same.

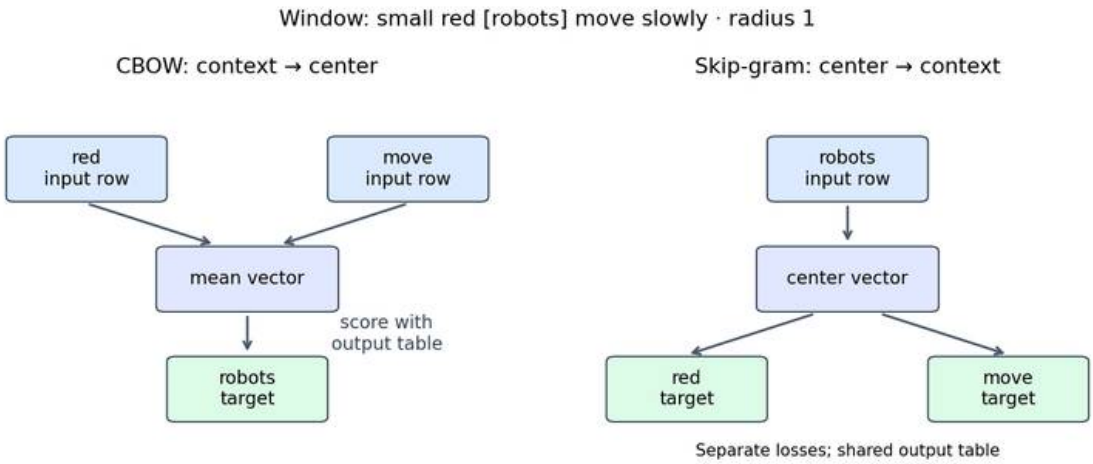


Figure 2 — The same sentence creates opposite prediction directions. Both architectures learn input and output embedding tables; the average appears only in the CBOW path shown here.

Two embedding tables and a softmax

A word needs a vector when it is an input and a vector when it is a prediction candidate. Use two learned tables, rather than assuming these roles share weights:

$$E, U \in \mathbb{R}^{V \times D}, \qquad \mathbf{e}_w = E_{w,:}^\top, \qquad \mathbf{u}_w = U_{w,:}^\top, \qquad p(o \mid \mathbf{h}) = \frac{\exp(\mathbf{u}_o^\top \mathbf{h})}{\sum_{v=1}^{V} \exp(\mathbf{u}_v^\top \mathbf{h})}.$$

Here E is the input embedding table, U is the output embedding table, V is vocabulary size, D is vector dimension, w is a word ID, and the colon selects all coordinates in its row. The transpose makes $\mathbf{e}_w$ and $\mathbf{u}_w$ column vectors. The vector $\mathbf{h} \in \mathbb{R}^D$ represents the input, o is the target word ID, v indexes a candidate word, and $p(o \mid \mathbf{h})$ is its softmax probability. Each dot product is a prediction score; the denominator makes probabilities sum to one over the vocabulary.

CBOW forms the input by averaging the context vectors. Its loss asks the model to assign high probability to the observed center:

$$\mathbf{h}_t = \frac{1}{|C_t|} \sum_{j \in C_t} \mathbf{e}_{w_j}, \qquad \mathcal{L}_{\text{CBOW}} = -\frac{1}{|\mathcal{T}|} \sum_{t \in \mathcal{T}} \log p(w_t \mid \mathbf{h}_t).$$

Here w_t is the center word at position t, C_t is the set of selected neighboring positions excluding t, $|C_t|$ is its size, $\mathbf{h}_t$ is their mean vector, $\mathcal{T}$ contains the valid center positions, and $\mathcal{L}_{\text{CBOW}}$ is the average negative log probability of those centers. The mean ignores context order: exchanging left and right neighbors produces the same input. Some formulations sum context vectors instead; this lesson and its code use the mean.

Skip-gram instead looks up the center vector and applies the same prediction rule to each neighbor:

$$\mathcal{L}_{\text{SG}} = -\frac{1}{|\mathcal{P}|} \sum_{(t,j) \in \mathcal{P}} \log p(w_j \mid \mathbf{e}_{w_t}).$$

Here $\mathcal{L}_{\text{SG}}$ is the Skip-gram loss, $\mathcal{P}$ is the collection of observed center–context position pairs, $|\mathcal{P}|$ is its size, t indexes a center position, j indexes one neighboring position, w_t is the input word and w_j is its target. Occurrences remain separate even when their word IDs repeat.

A training step increases the observed target's score relative to competing candidates. It changes both the selected input vectors and the output vectors. Over many examples, reusing the same input row makes the model reconcile that word's different surroundings. This shared prediction task is the route from discrete word IDs to learned continuous features. Neither architecture needs a deep stack of hidden layers in this basic form, which is the main computational difference from the neural language model in NLP 03.

Negative sampling: compare observed pairs with random draws

A full softmax scores every vocabulary word for every prediction. **Negative sampling** replaces that vocabulary-wide classification with a smaller binary task: distinguish an observed center–context pair from pairs formed with randomly drawn target words. It was introduced in the Word2Vec follow-up paper [8].

For an observed input word c and context word o, one sampled loss is:

$$\mathcal{L}_{\text{neg}}(c, o) = -\log \sigma(\mathbf{u}_o^\top \mathbf{e}_c) - \sum_{k=1}^{K} \log \sigma(-\mathbf{u}_{n_k}^\top \mathbf{e}_c), \qquad n_k \sim q, \qquad \sigma(a) = \frac{1}{1 + \exp(-a)}.$$

Here $\mathcal{L}_{\text{neg}}$ is the negative-sampling loss for one observed pair, c is the center word ID, o is its observed context word ID, $\mathbf{e}_c$ is the center's input vector, $\mathbf{u}_o$ and $\mathbf{u}_{n_k}$ are output vectors, K is the chosen number of random targets, n_k is draw k from distribution q, and σ is the sigmoid applied to a real score a. The first term rewards a large observed-pair score; the remaining terms reward small random-pair scores. The full training objective averages this expression over observed pairs and sampling randomness. For a CBOW version, use the averaged context vector in place of $\mathbf{e}_c$.

A common sampling distribution from the original paper is:

$$q(w) = \frac{f(w)^{3/4}}{\sum_{v=1}^{V} f(v)^{3/4}}.$$

Here $q(w)$ is the probability of drawing vocabulary word w, $f(w)$ is its corpus occurrence count, v indexes vocabulary entries, and V is vocabulary size. The exponent flattens the frequency distribution relative to sampling directly by counts. It is a design choice, not a universal probability law.

For the observed pair `robots → move`, a random draw might create `robots → red`. The label "negative" means **drawn by the noise process**; it does not assert that the two words can never occur together. A draw can be a genuine context elsewhere, or even collide with this target. An implementation must specify whether it retains, skips or redraws collisions; changing that rule changes the sampled objective. The model separates statistical sources of pairs rather than learning a dictionary of absolute incompatibilities.

The binary sigmoid scores do not sum to one over vocabulary words. Negative sampling is a different training objective, not an exact computation of the softmax above, and its loss is not a normalized next-token likelihood. Use it to learn useful representations; evaluate those representations on the intended downstream task.

Why prediction changes vector geometry

The phrase "learn from neighbors" becomes concrete in the gradient. For one full-softmax target, let its score be the dot product between an output vector and the input representation:

$$s_v = \mathbf{u}_v^\top \mathbf{h}, \qquad \ell = -\log p(o \mid \mathbf{h}), \qquad \frac{\partial \ell}{\partial s_v} = p(v \mid \mathbf{h}) - \mathbf{1}[v = o], \qquad \nabla_{\mathbf{h}} \ell = \sum_{v=1}^{V} p(v \mid \mathbf{h}) \mathbf{u}_v - \mathbf{u}_o.$$

Here s_v is candidate v's score, ℓ is the loss for observed target o, $\mathbf{1}[v = o]$ equals one for that target and zero otherwise, and $\nabla_{\mathbf{h}}\ell$ is the gradient with respect to the input representation. The embedding tables, vocabulary size V and conditional probability are those defined above.

Gradient descent moves the input toward the observed output vector and away from the model's probability-weighted average of alternatives. A word with many occurrences receives many such updates through the same row. Similar context distributions can therefore give different words compatible predictive roles, without any training label declaring them synonyms.

For mean-based CBOW, every selected context row receives $\nabla_{\mathbf{h}}\ell/|C_t|$. The input cannot distinguish two contexts that are permutations of the same words. For Skip-gram, one center row receives updates from multiple neighbor targets. Its basic score has no slot for whether a neighbor was on the left or right. Choosing the window therefore chooses which relations the objective can directly represent; a wider window is not simply a more accurate version of a narrow one.

What negative-sampling scores estimate

We can inspect the sampling objective before imposing the embedding dimension. Fix a center word and imagine that every candidate has an independently adjustable score. Let p_o be its empirical context probability and q_o its noise probability. The expected contribution for that candidate is:

$$\ell_o(s_o) = -p_o \log \sigma(s_o) - Kq_o \log \sigma(-s_o), \qquad \frac{d\ell_o}{ds_o} = (p_o + Kq_o)\sigma(s_o) - p_o.$$

Here o is one candidate context, s_o is its unrestricted real score, p_o and q_o are the two probabilities for the fixed center, K is the number of independent noise draws per observed pair and σ is the sigmoid defined above. This calculation retains collisions between noise draws and observed targets.

When both probabilities are positive, the stationary score is:

$$s_o^* = \log \frac{p_o}{Kq_o}.$$

Here s_o^* minimizes that candidate's expected loss. For $p_o = 0.5$, $q_o = 0.25$ and $K = 2$, it is zero: half the relevant probability mass is attributed to each statistical source. If $q_o = 0.05$ instead, the optimum becomes $\log 5 \approx 1.609$. The observed context frequency stayed fixed; the noise distribution changed the score. A positive score therefore means "more associated than this sampling baseline," rather than "the pair is certainly meaningful." If the observed probability is zero and noise probability is positive, the unrestricted optimum tends toward negative infinity.

When noise is exactly the corpus context marginal, this ratio yields PMI minus $\log K$. Levy and Goldberg analyze the resulting matrix-factorization interpretation [9]. The common three-quarter-power distribution changes the ratio. In an actual embedding model, scores share low-dimensional vectors, so they cannot generally reach every independent optimum. The learned tables compromise among the pair statistics under a rank constraint. This explains both the connection to the count-based construction and why truncating a PPMI matrix and training Word2Vec need not give identical vectors.

Full softmax costs roughly $O(VD)$ arithmetic per target, while scoring one observed target and K draws costs $O((K+1)D)$, before sampling and batching overhead. Here D is embedding dimension. The saving comes with a different objective. Neither the sigmoid value nor the negative-sampling loss is a substitute for the normalized token probability used to report perplexity.

Train CBOW and Skip-gram on CPU

This standalone example learns both input and output tables using full softmax. The vocabulary is tiny, so we can inspect the objective without a sampling approximation. Both models use the same invented corpus, radius-one interior windows, embedding width and training loop. A one-token input average in the Skip-gram path is simply the center vector.

```
import torch
from torch import nn
from torch.nn import functional as F

torch.manual_seed(7)
torch.set_num_threads(1)
corpus = [
    'small red robots move slowly',
    'small blue robots move quickly',
    'large red sensors work slowly',
    'large blue sensors work quickly',
    'small red sensors work quietly',
    'large blue robots move quietly',
]
vocab = sorted(set(' '.join(corpus).split()))
ids = {word: i for i, word in enumerate(vocab)}
cbow_x, cbow_y, sg_x, sg_y = [], [], [], []
for sentence in corpus:
    tokens = [ids[word] for word in sentence.split()]
    # Use radius 1 and interior centers, so every context has two words.
    for t in range(1, len(tokens) - 1):
        neighbors = [tokens[t - 1], tokens[t + 1]]
        cbow_x.append(neighbors)
        cbow_y.append(tokens[t])
        for neighbor in neighbors:
            sg_x.append([tokens[t]])
            sg_y.append(neighbor)

class Word2Vec(nn.Module):
    def __init__(self, size, width=8):
        super().__init__()
        self.input = nn.Embedding(size, width)
        self.output = nn.Embedding(size, width)
        nn.init.normal_(self.input.weight, std=0.1)
        nn.init.normal_(self.output.weight, std=0.1)

    def forward(self, context):
        h = self.input(context).mean(dim=1)
        return h @ self.output.weight.T
```

```
embeddings = {}
for name, inputs, targets in [('CBOW', cbow_x, cbow_y),
                              ('Skip-gram', sg_x, sg_y)]:
    x, y = torch.tensor(inputs), torch.tensor(targets)
    model = Word2Vec(len(vocab))
    optimizer = torch.optim.SGD(model.parameters(), lr=1.0)
    initial = F.cross_entropy(model(x), y).item()
    for step in range(250):
        optimizer.zero_grad()
        loss = F.cross_entropy(model(x), y)
        loss.backward()
        optimizer.step()
    final = F.cross_entropy(model(x), y).item()
    assert final < initial
    embeddings[name] = model.input.weight.detach().clone()
    print(name, len(y), f'{initial:.3f} -> {final:.3f}')
print('vocabulary', len(vocab), 'embedding shape', tuple(embeddings['CBOW'].shape))
```

The run prints `CBOW 18 2.384 -> 0.159`, `Skip-gram 36 2.402 -> 1.198`, and `vocabulary 11 embedding shape (11, 8)`. Losses are mean negative log probabilities in nats. The assertions establish that the implementation fits these training examples. The smaller CBOW loss does not establish that CBOW produces better word vectors: its targets and conditional inputs differ from Skip-gram's. This corpus is too small to judge semantic neighborhoods or natural-language generalization.

After training, `embeddings['CBOW'][ids['robots']]` retrieves one learned input vector. The output table was needed to train it but can be omitted when serving a downstream model that only looks up input vectors. Record which table you retain; using input vectors, output vectors or a combination is a representation choice.

Online only: word2vec — Efficient Estimation of Word Representations in Vector Space — https://blog.shyandram.dev/en/learn/nlp/lexical-semantics-word-embeddings/papers/word2vec–efficient-estimation-of-word-representations-in-vector-space/

GloVe: fit log co-occurrence counts directly

Plain version. Instead of streaming through windows one prediction at a time, GloVe first counts how often every pair of words co-occurs, then fits word vectors whose dot products reproduce the logarithms of those counts. Frequent pairs get more weight, but not unlimited weight.

Pennington et al. motivate GloVe with ratios of co-occurrence probabilities [10]. Let X_{ik} be the (distance-weighted) number of times word k occurs in the context of word i, and $P(k \mid i) = X_{ik} / \sum_{k'} X_{ik'}$. For two words i and j, the ratio $P(k \mid i)/P(k \mid j)$ is large for context words k that belong with i but not j, small for the reverse, and near one for words that are equally related or unrelated to both. The ratio discriminates better than either probability alone. The model asks vector differences to encode such ratios, which leads to a weighted least-squares objective over the co-occurrence matrix:

$$J = \sum_{i,j=1}^{V} f\big(X_{ij}\big)\left(\mathbf{w}_i^{\top}\tilde{\mathbf{w}}_j + b_i + \tilde{b}_j - \log X_{ij}\right)^2, \qquad f(x) = \begin{cases} (x/x_{\max})^{\alpha} & x < x_{\max}, \\ 1 & \text{otherwise.} \end{cases}$$

Here V is vocabulary size, X_{ij} is the co-occurrence count of words i and j, $\mathbf{w}_i \in \mathbb{R}^D$ is the word vector of i, $\tilde{\mathbf{w}}_j \in \mathbb{R}^D$ is a separate context vector of j, b_i and $\tilde{b}_j$ are scalar biases, f is a weighting function, and $x_{\max}$ and α are its cutoff and exponent. Because $f(0) = 0$, cells with zero count contribute nothing, so $\log 0$ is never needed and the sum runs only over observed pairs. The paper fixes $x_{\max} = 100$ and $\alpha = 3/4$, uses a co-occurrence weighting in which a pair d words apart adds $1/d$, and reports the sum $W + \tilde{W}$ of the two tables as the final word vectors [10].

Three design choices separate GloVe from Word2Vec. It trains on global counts rather than on individual windows, so one pass over the corpus builds X and training then costs time proportional to the number of nonzero cells. The biases absorb word frequency, so the dot product models what remains. And the weighting function prevents very frequent pairs, such as those involving `the`, from dominating, while giving rare and noisy pairs little influence. Like negative sampling, the objective factorizes a shifted log-count matrix under a rank constraint, which is why GloVe and Skip-gram vectors often behave similarly in practice.

The following code computes the ingredients on the earlier toy corpus: co-occurrence ratios and the weighting function.

```
import numpy as np

text = """the cat drinks milk . the dog drinks water . a cat chases a mouse .
a dog chases a cat . the kitten drinks milk . the puppy drinks water .
a kitten chases a ball . a puppy chases a ball . the car needs fuel .
the truck needs fuel . a car drives on the road . a truck drives on the road ."""
sents = [s.split() for s in text.split(".") if s.strip()]
vocab = sorted({w for s in sents for w in s})
idx = {w: i for i, w in enumerate(vocab)}

X = np.zeros((len(vocab), len(vocab)))            # +-2 window, 1/distance weighting
for s in sents:
    for i, w in enumerate(s):
        for j in range(max(0, i - 2), min(len(s), i + 3)):
            if j != i:
                X[idx[w], idx[s[j]]] += 1.0 / abs(i - j)
P = X / X.sum(1, keepdims=True)                   # P(k | i): row-normalized counts

for k in ["drinks", "chases", "needs", "the"]:
    a, b = P[idx["kitten"], idx[k]], P[idx["truck"], idx[k]]
    ratio = "inf" if b == 0 else f"{a / b:.2f}"
    print(f"P({k}|kitten)={a:.3f}  P({k}|truck)={b:.3f}  ratio={ratio}")

def f(x, x_max=100.0, alpha=0.75):                 # GloVe weighting function
    return np.minimum((x / x_max) ** alpha, 1.0)
print("f at 1, 10, 100, 1000:", np.round(f(np.array([1.0, 10.0, 100.0, 1000.0])), 3))
print("nonzero cells:", int((X > 0).sum()), "of", X.size)
```

The run prints:

```
P(drinks|kitten)=0.200  P(drinks|truck)=0.000  ratio=inf
P(chases|kitten)=0.200  P(chases|truck)=0.000  ratio=inf
P(needs|kitten)=0.000  P(needs|truck)=0.200  ratio=0.00
P(the|kitten)=0.200  P(the|truck)=0.200  ratio=1.00
f at 1, 10, 100, 1000: [0.032 0.178 1.    1.   ]
nonzero cells: 96 of 361
```

The ratios separate animal contexts from vehicle contexts, while `the` gives a ratio of one and therefore no information. The output also exposes a limitation of this tiny corpus: almost three quarters of the matrix is zero, and GloVe ignores zero cells. With so few observed pairs, a model with a few hundred parameters can fit every observed cell exactly, and the resulting geometry depends mainly on initialization. GloVe's count-based design needs large corpora in which the matrix is well populated; we therefore do not train it here. Exercise 3 asks you to observe this underdetermination yourself.

Online only: GloVe — Global Vectors for Word Representation — https://blog.shyandram.dev/en/learn/nlp/lexical-semantics-word-embeddings/papers/glove–global-vectors-for-word-representation/

fastText: vectors from character n-grams

Plain version. A word-level table has no vector for a word it never saw, and it treats `charger` and `chargers` as unrelated rows. fastText builds each word's vector from vectors of its character pieces, so related spellings share parameters and unseen words still get a vector.

Bojanowski et al. extend Skip-gram with negative sampling by representing each word as a bag of character n-grams [11]. The word is wrapped in boundary symbols, < and >, so that prefixes and suffixes are distinguishable from the same letters inside a word, and the whole wrapped word is added as one extra unit. In the paper, all n-grams with $3 \le n \le 6$ are used. The score between a center word w and a context word c becomes

$$s(w, c) = \sum_{g \in \mathcal{G}_w} \mathbf{z}_g^\top \mathbf{v}_c.$$

Here $\mathcal{G}_w$ is the set of n-grams of w (including the wrapped word itself), $\mathbf{z}_g \in \mathbb{R}^D$ is the learned vector of n-gram g, and $\mathbf{v}_c \in \mathbb{R}^D$ is the context vector of c. The word's own vector is $\sum_{g \in \mathcal{G}_w} \mathbf{z}_g$. This score replaces $\mathbf{u}_o^\top \mathbf{e}_c$ in the negative-sampling loss above. To bound memory, the paper maps n-grams into a fixed number of buckets with a hash function, so unrelated n-grams can share a vector [11].

```
import numpy as np

def char_ngrams(word, n_min=3, n_max=6):
    w = f"<{word}>"                      # boundary symbols mark prefixes and suffixes
    grams = {w[i:i + n] for n in range(n_min, n_max + 1) for i in range(len(w) - n + 1)}
    return grams | {w}                   # the whole word is kept as one extra unit
```

```
print(sorted(char_ngrams("robot", 3, 3)))
vocab = ["charge", "charger", "recharge", "discharge", "robot", "robots"]
query = "chargers"                          # not in the vocabulary
q = char_ngrams(query)
for word in vocab:
    print(f"{word:10s} shares {len(q & char_ngrams(word)):2d} of {len(q)} units with
     ↪  '{query}'")

# A fastText-style vector is the sum of its unit vectors, so an unseen word still gets one.
rng = np.random.default_rng(0)
table = {}                                  # stands in for a trained (hashed) n-gram table
def vector(word, dim=8):
    return sum(table.setdefault(g, rng.normal(size=dim)) for g in sorted(char_ngrams(word)))
def cos(a, b):
    return a @ b / (np.linalg.norm(a) * np.linalg.norm(b))
for word in vocab:
    vector(word)
print("cos(chargers, charger) =", round(cos(vector("chargers"), vector("charger")), 3))
print("cos(chargers, robots)  =", round(cos(vector("chargers"), vector("robots")), 3))
```

The run prints:

```
['<ro', '<robot>', 'bot', 'obo', 'ot>', 'rob']
charge     shares 14 of 27 units with 'chargers'
charger    shares 18 of 27 units with 'chargers'
recharge   shares 10 of 27 units with 'chargers'
discharge  shares 10 of 27 units with 'chargers'
robot      shares  0 of 27 units with 'chargers'
robots     shares  0 of 27 units with 'chargers'
cos(chargers, charger) = 0.804
cos(chargers, robots)  = -0.222
```

The n-gram vectors here are random, not trained, so the high cosine between `chargers` and `charger` comes only from the units they share. That is exactly the mechanism: an unseen inflection inherits most of its vector from known pieces. Training decides whether sharing is helpful. It usually is for inflection and derivation, and it is misleading for words that share letters by accident; `discharge` and `recharge` share `charge` but differ in meaning in ways the pieces do not reveal. Subword vectors help most for morphologically rich languages and for rare words. Note the relation to the subword tokenizers in NLP 02: BPE splits a word into one sequence of pieces before lookup, whereas fastText sums vectors of *all* overlapping n-grams.

Dense word and document representations

Plain version. A dense vector lets different words share statistical strength. That can connect related phrases, but the vector's geometry reflects its training data and objective rather than a universal definition of meaning.

Static vectors provide compact reusable features for a classifier, nearest-neighbor word exploration or a pooled document baseline. They are useful when a labeled application has little data but an appropriate text corpus is available. Training the embedding and the downstream classifier are different stages: neighbor prediction never directly learns a billing label or an abstention policy.

A static embedding table assigns a vector to each vocabulary entry:

$$\mathbf{e}_t = E_{i_t,:}, \qquad \mathbf{h}(d) = \frac{1}{T}\sum_{t=1}^{T} \mathbf{e}_t.$$

Here $E \in \mathbb{R}^{V \times D}$ is the embedding table, V is vocabulary size, D is embedding dimension, i_t is the token ID at position t, $\mathbf{e}_t$ is its row vector, T is document length in tokens and $\mathbf{h}(d)$ is the mean-pooled document vector. The colon denotes all coordinates in the selected row.

The tables above supply ways to obtain E. Once token vectors are available, mean pooling turns a variable-length document into a fixed-width feature. This aggregation step is separate from the objective used to train the table.

Mean pooling is an instructive baseline because it exposes a limitation clearly. Any permutation of a document's tokens gives the same average. The representation of "the technician helped the customer" is identical to "the customer helped the technician" if the same static token vectors are used. It also treats negation as one contribution among many rather than an operator changing the meaning of a phrase. The word `bank` retrieves the same static vector in `river bank` and `bank account`; more training data does not change this lookup rule.

There are two separate choices: how token vectors are obtained and how they are combined. You can freeze pretrained token vectors and train only a classifier, fine-tune the vectors with the classifier, or train all parameters from scratch. Freezing reduces the number of task-specific parameters. Fine-tuning can adapt meanings to a domain, but small datasets can distort representations or memorize labels.

Do not compare embeddings only by visually attractive clusters. A two-dimensional projection can conceal overlap in the original space. Compare representations under the same task, split, classifier capacity and tuning allowance. If a dense representation changes the downstream head too, the experiment no longer isolates the representation.

Cosine measures direction, not truth

The count-based example compares unit-normalized vectors. Its dot product is then the cosine of the angle between them:

$$\cos(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a}^\top \mathbf{b}}{\|\mathbf{a}\|_2 \|\mathbf{b}\|_2}.$$

Here $\mathbf{a}$ and $\mathbf{b}$ are nonzero embedding column vectors, $\mathbf{a}^\top \mathbf{b}$ is their inner product, and $\|\cdot\|_2$ is Euclidean length. The cosine lies between -1 and 1. A zero vector needs an explicit handling rule because the denominator vanishes.

Cosine removes positive rescaling of either vector. It does not remove every transformation compatible with the training objective. In the two-table model, the predictive scores depend on products of an input and an output vector. Many different tables produce the same products:

$$E' = EA, \qquad U' = UA^{-\top}, \qquad E'U'^{\top} = EU^{\top}.$$

Here E and U are the input and output tables with shape $V \times D$, A is any invertible $D \times D$ matrix, $A^{-\top}$ is the transpose of its inverse, and E' and U' are the transformed tables. Each cell of $EU^{\top}$ is one center–output prediction score. Thus full-softmax predictions and the negative-sampling objective are unchanged by this paired transformation. The same argument applies to GloVe's $\mathbf{w}_i^{\top}\tilde{\mathbf{w}}_j$.

An orthogonal change of coordinates preserves input-vector angles, but general invertible changes need not. Consequently, identical prediction loss does not uniquely determine cosine neighborhoods in the retained input table. Optimization, initialization and additional constraints select a particular solution. This is one reason a low training loss alone cannot certify useful downstream similarity.

This standalone numerical check combines three claims: score preservation, order invariance of averaging, and the noise-dependent optimum derived above. The matrices are hand-chosen for this calculation; they are not trained language vectors.

```
import numpy as np

# Two input rows and three output rows: scores are center-output dot products.
E = np.array([[1., 1.], [1., -1.]])
U = np.array([[1., 0.], [0., 1.], [-1., 1.]])
A = np.diag([2., 0.5])
changed_E = E @ A
changed_U = U @ np.linalg.inv(A).T

def cosine(a, b):
    return (a @ b) / (np.linalg.norm(a) * np.linalg.norm(b))

assert np.allclose(E @ U.T, changed_E @ changed_U.T)
assert np.allclose(E[[0, 1]].mean(0), E[[1, 0]].mean(0))
print('scores unchanged:', np.allclose(E @ U.T, changed_E @ changed_U.T))
print('cosine:', f'{cosine(E[0], E[1]):.3f}', '->',
      f'{cosine(changed_E[0], changed_E[1]):.3f}')
print('mean unchanged by order:',
      np.allclose(E[[0, 1]].mean(0), E[[1, 0]].mean(0)))

# The unrestricted negative-sampling optimum depends on the noise baseline.
p, K = 0.5, 2
for q in [0.25, 0.05]:
    score = np.log(p / (K * q))
    sigmoid = 1 / (1 + np.exp(-score))
    gradient = (p + K * q) * sigmoid - p
    assert np.isclose(gradient, 0)
    print('noise', q, 'optimal score', f'{score:.3f}')
```

The output is:

```
scores unchanged: True
cosine: 0.000 -> 0.882
mean unchanged by order: True
noise 0.25 optimal score 0.000
noise 0.05 optimal score 1.609
```

The cosine changed while all center–output scores stayed fixed. The equal means have a different implication: once a representation merges two token orders, no classifier that receives only that mean can recover which order occurred. These are properties of the representation, rather than evidence that one optimizer ran poorly.

Evaluating word embeddings

Plain version. There are two ways to judge embeddings: ask whether their geometry agrees with human judgments about words (intrinsic), or plug them into a task and measure the task (extrinsic). Neither alone tells you whether vectors are "good."

Similarity: correlate cosines with human ratings

A **word-similarity** dataset lists word pairs with graded human ratings. The model scores each pair by cosine, and the evaluation measures how well the two rankings agree with **Spearman's rank correlation**:

$$\rho = 1 - \frac{6 \sum_{i=1}^{n} \delta_i^2}{n(n^2 - 1)}.$$

Here n is the number of pairs, δ_i is the difference between the rank of pair i under the model's cosines and its rank under the human ratings, and $\rho \in [-1, 1]$. This formula assumes no ties; with ties, ρ is the Pearson correlation of the ranks. Spearman's ρ uses only ranks, so it ignores the scale of cosines.

What the ratings mean matters as much as the statistic. WordSim-353 is a widely used set of rated word pairs [12]. Hill et al. point out that its ratings do not separate similarity from association, and built SimLex-999 to quantify **similarity** explicitly, so that associated but dissimilar pairs receive low ratings [13]. A model trained with broad windows may score well on association-style data and poorly on similarity-style data, for the reason given in the lexical-semantics section: distributional evidence mixes the relations.

Analogy: the vector-offset test and its caveats

Mikolov, Yih and Zweig observed that some relations appear as roughly constant vector offsets [14]. To answer "a is to a^* as b is to ?", the **offset method** returns

$$x^* = \operatorname*{argmax}_{x \in \mathcal{V} \setminus \{a, a^*, b\}} \cos(\mathbf{e}_x,\ \mathbf{e}_{a^*} - \mathbf{e}_a + \mathbf{e}_b)\,.$$

Here $\mathcal{V}$ is the vocabulary, $\mathbf{e}_x$ is the vector of candidate x, and the three question words are excluded from the candidates. Linzen points out that this exclusion is doing real work: the target point $\mathbf{e}_{a^*} - \mathbf{e}_a + \mathbf{e}_b$ often lies closest to b itself, so the method's success mixes offset consistency with the neighborhood structure around b [15]. He recommends reporting simple baselines, such as returning the nearest neighbor of b, alongside analogy accuracy.

The next example evaluates the PPMI vectors from earlier with invented similarity ratings and three analogies. The ratings are made up for this lesson and are not a benchmark.

```
import numpy as np
from scipy.stats import spearmanr

text = """the cat drinks milk . the dog drinks water . a cat chases a mouse .
a dog chases a cat . the kitten drinks milk . the puppy drinks water .
a kitten chases a ball . a puppy chases a ball . the car needs fuel .
the truck needs fuel . a car drives on the road . a truck drives on the road ."""
sents = [s.split() for s in text.split(".") if s.strip()]
vocab = sorted({w for s in sents for w in s})
idx = {w: i for i, w in enumerate(vocab)}
C = np.zeros((len(vocab), len(vocab)))
for s in sents:
    for i, w in enumerate(s):
        for j in range(max(0, i - 2), min(len(s), i + 3)):
            if j != i:
                C[idx[w], idx[s[j]]] += 1
P = C / C.sum()
with np.errstate(divide="ignore"):
    ppmi = np.maximum(np.log(P / (P.sum(1, keepdims=True) * P.sum(0, keepdims=True))), 0)
U, S, _ = np.linalg.svd(ppmi)
E = U[:, :8] * S[:8]
E /= np.linalg.norm(E, axis=1, keepdims=True)
cos = lambda a, b: float(E[idx[a]] @ E[idx[b]])

# 1) Similarity: correlate model cosines with (invented) graded judgments, 0-10.
judged = [("cat", "kitten", 9), ("dog", "puppy", 9), ("car", "truck", 8), ("milk", "water",
 ↪  5),
          ("cat", "dog", 6), ("cat", "mouse", 3), ("car", "road", 2), ("kitten", "fuel", 0)]
model = [cos(a, b) for a, b, _ in judged]
rho = spearmanr(model, [h for _, _, h in judged]).statistic
print(f"Spearman rho = {rho:.3f} over {len(judged)} pairs")

# 2) Analogy a:b :: c:? with the offset b - a + c (3CosAdd).
def analogy(a, b, c, exclude_inputs=True):
    target = E[idx[b]] - E[idx[a]] + E[idx[c]]
    scores = E @ (target / np.linalg.norm(target))
    for i in np.argsort(-scores):
        if not exclude_inputs or vocab[i] not in (a, b, c):
            return vocab[i]
for a, b, c in [("cat", "kitten", "dog"), ("car", "fuel", "cat"), ("cat", "milk", "puppy")]:
    print(f"{a}:{b} :: {c}:?  ->", analogy(a, b, c), "| without excluding inputs ->",
```

```
          analogy(a, b, c, exclude_inputs=False))
```

The run prints:

```
Spearman rho = 0.837 over 8 pairs
cat:kitten :: dog:?  -> puppy | without excluding inputs -> puppy
car:fuel :: cat:?  -> kitten | without excluding inputs -> cat
cat:milk :: puppy:?  -> water | without excluding inputs -> water
```

Two analogies come out as intended. The middle one shows both caveats at once. Without exclusion, the answer is `cat`, one of the question words. With exclusion, the answer is `kitten`, the nearest neighbor of `cat`, not `milk`, the intended "what it consumes" answer. The offset did not carry the relation; the neighborhood of b decided. Eight invented pairs and three analogies are, of course, far too few to support any general claim about these vectors.

Intrinsic versus extrinsic

Schnabel et al. compare evaluation methods and find that different evaluations order embedding methods differently, that word frequency affects results and cosine similarity, and that extrinsic tasks should not be used as a proxy for generic embedding quality [16]. The practical rule is to evaluate on the relation your application needs.

Nearest-neighbor lists can reveal obvious corpus patterns, but they are a diagnostic rather than an application score. Separate lexical questions: is the candidate a synonym, an antonym, a word associated with the topic, or a word that can occupy the same grammatical slot? A single unqualified notion of "related" hides these distinctions.

Use controlled pairs alongside ordinary held-out examples. For a representation used in sentiment, compare a positive claim with its negation. For a role-sensitive task, exchange who acted on whom while holding the word inventory fixed. For a technical corpus, test a term used in its domain-specific and everyday senses. These tests ask whether the representation preserves a required difference; they do not demand that every word pair occupy a predetermined location.

If mean pooling makes a critical pair identical, collecting more classifier labels cannot fix that collision while the representation remains fixed. If the vectors distinguish the pair but the classifier predicts incorrectly, supervision or the decision boundary may be responsible. Locate the failure at the earliest stage that loses the needed evidence.

Bias in embeddings

Plain version. Embeddings learn whatever regularities the corpus contains, including social stereotypes. If a group of names co-occurs with some occupations more than others, the vectors encode that association, and downstream systems can reproduce it.

Bolukbasi et al. showed that word embeddings trained on news text encode gender stereotypes, identified a gender direction in the space, and proposed removing it from words that should be gender-neutral [17]. Caliskan et al. introduced the Word Embedding Association Test (WEAT), modeled on the Implicit Association Test from psychology, and found that embeddings reproduce a range of documented human biases [18]. For a target word w and two attribute sets A and B, the association used by WEAT is

$$s(w, A, B) = \frac{1}{|A|}\sum_{a \in A} \cos(\mathbf{e}_w, \mathbf{e}_a) - \frac{1}{|B|}\sum_{b \in B} \cos(\mathbf{e}_w, \mathbf{e}_b).$$

Here $|A|$ and $|B|$ are the set sizes and $\mathbf{e}$ denotes word vectors. A positive value means w is closer on average to set A. The full test aggregates this score over two sets of target words and compares it with a permutation distribution.

The following invented corpus contains no sentence linking a name to an occupation. Occupation words always appear with their own activity, while invented names appear with activities at skewed rates.

```
import random
import numpy as np

rng = random.Random(0)
# An invented corpus. Job words always appear with their own activity; invented
# names appear with activities at skewed rates. No sentence says "ana is a pilot".
group_a, group_b = ["ana", "ari"], ["bo", "bea"]
activity = {"pilot": "flies the plane", "baker": "bakes the bread", "clerk": "files the
 ↪  forms"}
share_a = {"pilot": 0.8, "baker": 0.2, "clerk": 0.5}    # P(group A | activity)
sents = []
for _ in range(3000):
    job = rng.choice(list(activity))
    if rng.random() < 0.5:
        sents.append(f"the {job} {activity[job]}".split())
    else:
        name = rng.choice(group_a if rng.random() < share_a[job] else group_b)
        sents.append(f"{name} {activity[job]}".split())
vocab = sorted({w for s in sents for w in s})
idx = {w: i for i, w in enumerate(vocab)}
C = np.zeros((len(vocab), len(vocab)))
for s in sents:
    for i, w in enumerate(s):
        for j in range(max(0, i - 2), min(len(s), i + 3)):
            if j != i:
                C[idx[w], idx[s[j]]] += 1
P = C / C.sum()
with np.errstate(divide="ignore"):
    ppmi = np.maximum(np.log(P / (P.sum(1, keepdims=True) * P.sum(0, keepdims=True))), 0)
U, S, _ = np.linalg.svd(ppmi)
E = U[:, :4] * S[:4]
E /= np.linalg.norm(E, axis=1, keepdims=True)

def association(word, E):            # mean cosine to group A minus mean cosine to group B
    unit = E / np.linalg.norm(E, axis=1, keepdims=True)
```

```
    w = unit[idx[word]]
    return np.mean([w @ unit[idx[a]] for a in group_a]) - np.mean([w @ unit[idx[b]] for b in
     ↪  group_b])

for job in activity:
    print(f"{job:6s} association {association(job, E):+.3f}")

# Project out one "group direction" and measure again.
d = E[[idx[a] for a in group_a]].mean(0) - E[[idx[b] for b in group_b]].mean(0)
d /= np.linalg.norm(d)
E_proj = E.copy()
for job in activity:
    E_proj[idx[job]] -= (E_proj[idx[job]] @ d) * d
print("after projection:", [f"{job} {round(association(job, E_proj), 3) + 0.0:+.3f}" for job
 ↪  in activity])
```

The run prints:

```
pilot  association +0.795
baker  association -0.811
clerk  association +0.072
after projection: ['pilot +0.000', 'baker +0.000', 'clerk +0.000']
```

The skew in who performs which activity became a strong association between names and occupation words, even though the two never co-occur in a sentence: both share the activity contexts. Projecting out the group direction sets the measured association to zero, but that result is circular, because the projection removes exactly the direction this score measures. Gonen and Goldberg show that, for published debiasing methods, the bias is largely hidden rather than removed: it remains in the distances between supposedly neutralized words and can be recovered from them [19]. Treat a debiasing step as one intervention to be audited on the downstream task, not as a guarantee, and remember that the source of the association is the data.

From word vectors to sentence embeddings

Plain version. An embedding supplies features; an application still needs a target, supervision and an action rule. When the application compares whole sentences, train the sentence representation for the comparison it will support.

For document classification, mean pooling is a convenient first experiment. Build one vector per training document, fit a classifier on its labels, and apply the same tokenizer, embedding lookup and pooling at evaluation time. Compare it with the sparse features and convolutional encoders developed in NLP 03. Keep the data split and classifier tuning allowance fixed so that a change in representation can be assessed fairly.

A frozen table uses contextual statistics learned before the labeled task. Fine-tuning lets the labels adjust the rows. These policies answer different questions: whether reusable corpus features

already distinguish the classes, and whether task-specific changes improve them. With limited labels, fine-tuning can also fit incidental identifiers. Holding out groups such as conversations or repeated document templates helps reveal that behavior.

Decide how to handle uncovered text. Mapping every unknown word to one row merges all such words. Dropping them changes the pooling denominator and may leave an empty document. Returning a zero vector does not create a meaningful cosine direction. Subword vectors reduce but do not remove this problem. Coverage, the number of retained tokens, and the action for an empty representation are part of the model specification.

For word-level tagging, retain the sequence of token vectors rather than pooling it into one document vector. A downstream encoder can then combine local context before predicting a label per position. This does not make the original table contextual: the initial lookup of `bank` is still identical in both senses. The context-dependent computation happens afterward.

Similarity and contrastive sentence learning

A representation useful for predicting neighboring words may need adaptation before distances between whole sentences become reliable. Sentence-BERT trains paired sentence representations with a shared encoder and task objectives, making sentence comparisons possible without jointly encoding every candidate pair [20]. The computational distinction matters: separately cached document vectors can support retrieval, whereas a model that reads a query and candidate together must run for each pair.

A generic contrastive objective makes the intended geometry explicit:

$$\mathcal{L}_{\text{pair}} = -\log \frac{\exp(\cos(\mathbf{q}, \mathbf{a}^{+})/\tau)}{\sum_{j=1}^{M} \exp(\cos(\mathbf{q}, \mathbf{a}_j)/\tau)}.$$

Here $\mathcal{L}_{\text{pair}}$ is the loss for one query, $\mathbf{q}$ is its vector, $\mathbf{a}^{+}$ is the vector of a relevant paired text included among the candidates, $\mathbf{a}_j$ is candidate j, M is candidate count and $\tau > 0$ is a temperature controlling the sharpness of the distribution. This is a generic learning construction, not a claim that every sentence encoder uses this exact objective. It has the same shape as the Word2Vec softmax: one observed pair competes with alternatives.

SimCSE is a widely used instance [21]. Its unsupervised version encodes the same sentence twice; because dropout differs between the two passes, the two vectors differ slightly and form a positive pair, while other sentences in the batch are negatives. The authors report that removing dropout leads to representation collapse. Its supervised version uses natural-language-inference data, taking entailment pairs as positives and contradiction pairs as hard negatives [21].

Negative examples define what "different" means. Two tickets about different invoices might be useful negatives for finding the same case but poor negatives for recognizing the billing category. False negatives can push semantically related examples apart. Before changing an encoder, inspect how positive and negative pairs are sampled.

Similarity also differs from entailment. "The parcel arrived" and "The parcel did not arrive" share topic and many tokens. They may be close in a semantic search space while making opposite

claims. Classification, semantic search and inference therefore require different validation examples, even when they share an encoder. NLP 11 treats inference as its own task.

Preview: contextual embeddings

Plain version. A static table gives `bank` one vector everywhere. A contextual model computes a different vector for each occurrence, using the surrounding words, so the river sense and the money sense can come apart.

Peters et al. compute ELMo representations as functions of the entire input sentence, using the internal states of a bidirectional language model, and add them as features to task models [22]. BERT takes the step further within its pretrained encoder: a token's final hidden state depends on the other permitted input tokens [23]. The distinction is **lookup versus computation**. Both systems may begin with an embedding table; static Word2Vec exposes a row directly, while BERT processes rows and positions through a context-dependent network.

Context can improve sense discrimination, but it does not turn every hidden-state cosine into a calibrated similarity measure. A static vector is a type-level summary across many occurrences; a contextual vector is a token-level representation of one occurrence. Which one an application needs depends on whether it asks about a word in general or a word as used here. NLP 07 builds the attention mechanism these models use, and NLP 10 develops masked pretraining and BERT's contextual representations rather than repeating them here.

Modern view

Turney and Pantel's survey [6] distinguishes term–document, word–context and pair–pattern matrices. Its taxonomy makes the source of evidence explicit: a document association, a neighboring-word relation and a relational pattern are different observations. The lesson here is to define the matrix before interpreting the vector space.

Apidianaki's 2023 review [24] follows word meaning from static type-level vectors to contextual token-level representations and back. It covers semantic specialization, sense distinctions, representation interpretation and the derivation of static vectors from contextual encoders. Its central distinction is the level at which evidence is aggregated, rather than a ranking in which a newer architecture automatically solves every lexical task. Methods that inject lexical-resource relations into vectors connect directly to the WordNet section of this lesson.

For sentence- and text-level embeddings, Muennighoff et al.'s MTEB benchmark spans 8 embedding tasks, 58 datasets and 112 languages, and its authors find that no single text-embedding method dominates across all tasks [25]. In 2023–2026, most practical text embeddings come from pretrained Transformer encoders or language models fine-tuned with contrastive objectives of the kind shown above, and benchmark suites in the style of MTEB are the usual way to compare them. Static vectors remain useful as fast, transparent baselines, as features in low-resource settings, and as objects of study.

A useful research lens is therefore: **what meaning distinction survives this representation, and how was that checked?** Contextual hidden states, static vectors obtained by aggregating them, and task-trained sentence representations answer different questions. An embedding that preserves a broad topic may still collapse a negated claim or a rare sense. A small static table can remain an appropriate baseline when its measurable invariances match the application.

Open questions include reliable geometry across domains, sense separation when evidence is sparse, bias that survives debiasing, and comparisons that hold corpus and evaluation conditions fixed. Multilingual work adds segmentation and uneven coverage: a row is only useful if the tokenizer and training corpus supplied evidence for that type. Report those conditions and challenge examples alongside the final score. This keeps the explanation tied to observed language behavior.

Key takeaways

- A vocabulary entry is a form or lemma, not a sense; lexical relations such as synonymy, antonymy and hypernymy hold between senses, and WordNet records them by hand.
- A static embedding aggregates many occurrences into one vector per vocabulary entry; distributional similarity captures shared usage and mixes synonymy, antonymy and relatedness.
- PPMI compares co-occurrence with independence; SVD, Skip-gram with negative sampling and GloVe all compress association statistics under a rank constraint.
- CBOW predicts a center from combined neighbors; Skip-gram predicts neighbors separately; negative sampling distinguishes statistical sources of pairs and is not a normalized distribution.
- fastText builds word vectors from character n-grams, so related spellings share parameters and unseen words get vectors.
- Similarity and analogy tests depend on what the ratings mean and on details such as excluding the question words; evaluate the relation your application needs.
- Embeddings absorb corpus associations, including social biases; projection-based debiasing can hide rather than remove them.
- Mean pooling erases order; sentence embeddings should be trained for the comparison they serve, and contextual models compute a vector per occurrence rather than looking one up.

Exercises

1. **Build the examples.** Enumerate radius-one CBOW and Skip-gram examples for `robots help people today`, using only interior centers. Then include the boundary centers. Which input shapes change?

   Hint

   Count centers separately from center–context pairs. Variable-length CBOW contexts need a mask or separate processing; Skip-gram still receives one center ID per pair.

2. **Inspect a matrix cell.** Construct a corpus in which one context occurs frequently overall but has PMI near zero for a chosen center. Then add observed center–context pairs and explain both the numerator and marginal changes.

   Hint

   Begin with a joint table close to independence. Recompute all marginals after editing the count; keeping them fixed would analyze a different table.

3. **Train GloVe and watch it wobble.** Implement the GloVe objective in PyTorch on the toy corpus, training only on nonzero cells. Train it with three random seeds and compare the nearest neighbors of `cat`. Then count the parameters and the nonzero cells. What explains the instability, and what would make the problem better determined?

   Hint

   Two $V \times D$ tables plus two bias vectors can easily outnumber the observed cells. A larger corpus with more varied contexts adds equations; a smaller D removes unknowns.

4. **Similarity versus relatedness.** Write ten word pairs that are related but not similar (such as `c up` and `coffee`) and ten that are similar. Using the PPMI code with window radius 1 and then 4 on a corpus you write, compare how the cosine separates the two groups.

   Hint

   Wider windows collect topical co-occurrence, which favors relatedness; narrow windows emphasize words that fill the same slot. Report the change for both groups, not a single score.

5. **Expose an invariant.** Create two meaningful sentences with the same word counts and different participant roles. Show why static mean pooling cannot separate them, even if every word vector is learned perfectly for its training objective.

   Hint

   Changing order does not change a sum. The limitation holds for every possible static embedding table, not only for the synthetic values in this lesson.

6. **Audit beyond the projection.** In the bias example, after projection, compute each occupation word's nearest names using all remaining dimensions, and train a small logistic classifier to predict group A versus group B from the projected occupation-word neighbors. Does the association still leave a trace?

   Hint

   The projection removed one direction measured one way. Look for structure the score did not measure, as Gonen and Goldberg did with clustering and classification.

References

1. M. Hagiwara, *Real-World Natural Language Processing: Practical Applications with Deep Learning*, Manning, 2021; Chapters 3–4. publisher
2. S. McRoy, *Principles of Natural Language Processing*, open textbook, 2021; Chapter 6, "Benchmark Tasks for Language Modelling." chapter
3. Y. Bengio, R. Ducharme, P. Vincent and C. Jauvin, "A neural probabilistic language model," *Journal of Machine Learning Research* 3, 2003. paper
4. G. A. Miller, "WordNet: a lexical database for English," *Communications of the ACM* 38(11), 1995. paper
5. Z. S. Harris, "Distributional structure," *Word* 10(2–3), 1954. paper
6. P. D. Turney and P. Pantel, "From frequency to meaning: vector space models of semantics," *Journal of Artificial Intelligence Research* 37, 2010. survey
7. T. Mikolov, K. Chen, G. Corrado and J. Dean, "Efficient estimation of word representations in vector space," ICLR Workshop, 2013. paper
8. T. Mikolov, I. Sutskever, K. Chen, G. Corrado and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in Neural Information Processing Systems* 26, 2013. paper
9. O. Levy and Y. Goldberg, "Neural word embedding as implicit matrix factorization," *Advances in Neural Information Processing Systems* 27, 2014. paper
10. J. Pennington, R. Socher and C. D. Manning, "GloVe: global vectors for word representation," EMNLP, 2014. paper
11. P. Bojanowski, E. Grave, A. Joulin and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the Association for Computational Linguistics* 5, 2017. paper
12. L. Finkelstein, E. Gabrilovich, Y. Matias, E. Rivlin, Z. Solan, G. Wolfman and E. Ruppin, "Placing search in context: the concept revisited," *ACM Transactions on Information Systems* 20(1), 2002. paper
13. F. Hill, R. Reichart and A. Korhonen, "SimLex-999: evaluating semantic models with (genuine) similarity estimation," *Computational Linguistics* 41(4), 2015. paper
14. T. Mikolov, W.-t. Yih and G. Zweig, "Linguistic regularities in continuous space word representations," NAACL-HLT, 2013. paper
15. T. Linzen, "Issues in evaluating semantic spaces using word analogies," Workshop on Evaluating Vector-Space Representations for NLP (RepEval), 2016. paper
16. T. Schnabel, I. Labutov, D. Mimno and T. Joachims, "Evaluation methods for unsupervised word embeddings," EMNLP, 2015. paper
17. T. Bolukbasi, K.-W. Chang, J. Zou, V. Saligrama and A. Kalai, "Man is to computer programmer as woman is to homemaker? Debiasing word embeddings," *Advances in Neural Information Processing Systems* 29, 2016. paper
18. A. Caliskan, J. J. Bryson and A. Narayanan, "Semantics derived automatically from language corpora contain human-like biases," *Science* 356(6334), 2017. paper
19. H. Gonen and Y. Goldberg, "Lipstick on a pig: debiasing methods cover up systematic gender biases in word embeddings but do not remove them," NAACL-HLT, 2019. paper
20. N. Reimers and I. Gurevych, "Sentence-BERT: sentence embeddings using Siamese BERT-networks," EMNLP-IJCNLP, 2019. paper

21. T. Gao, X. Yao and D. Chen, “SimCSE: simple contrastive learning of sentence embeddings,” EMNLP, 2021. paper
22. M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee and L. Zettlemoyer, “Deep contextualized word representations,” NAACL-HLT, 2018. paper
23. J. Devlin, M.-W. Chang, K. Lee and K. Toutanova, “BERT: pre-training of deep bidirectional transformers for language understanding,” NAACL-HLT, 2019. paper
24. M. Apidianaki, “From word types to tokens and back: a survey of approaches to word meaning representation and interpretation,” *Computational Linguistics* 49(2), 2023. survey
25. N. Muennighoff, N. Tazi, L. Magne and N. Reimers, “MTEB: massive text embedding benchmark,” EACL, 2023. paper

NLP 05 • Recurrent Neural Networks for Language

Start from n-gram language models and their memory ceiling, then build, train and evaluate recurrent language models: Elman RNNs, LSTMs and GRUs, character-level modeling, sampling and practical training.

Online edition: https://blog.shyandram.dev/en/learn/nlp/recurrent-neural-networks/

What you'll learn

- Separate an n-gram occurrence, a document feature and a conditional language model, and derive unigram, bigram and trigram predictions from sequence probabilities.
- Count, smooth and interpolate n-gram estimates, then interpret held-out negative log likelihood and perplexity.
- Explain why a fixed-length history creates a memory ceiling that more data cannot remove.
- Write the Elman recurrent language model, its next-token cross-entropy and its perplexity, and explain how backpropagation through time, vanishing gradients, LSTM and GRU cells affect what it can learn.
- Train a character-level LSTM language model with stream packing, truncated backpropagation, clipping and early stopping, and compare it fairly with a tuned n-gram baseline.
- Sample from a trained model with temperature, and say what samples can and cannot tell you.

The big picture

A language model assigns probabilities to text by predicting the next token from the text before it. This chapter builds two such models and compares them under one evaluation contract. The first, an n-gram model, predicts the next token using a small piece of the preceding text. Its probability table is directly inspectable: we can trace a prediction to training counts and see exactly which earlier words the model ignores. This makes it a useful first language model, a controlled baseline, and a way to understand the statistical problems that more elaborate models also face. The second, a recurrent neural network (RNN), replaces the table with a learned state that is updated after every token, so its prediction can in principle depend on the entire prefix.

> **Explain like I'm five**
>
> Imagine a notebook recording what people say next. One page records what follows "cat"; another records what follows "the cat." When you need a guess, you open the matching page and count the examples. Longer page headings give you more context, but many pages may be nearly empty. Smoothing reserves a little room for words that have not yet appeared on that page. A recurrent network works differently: instead of a notebook with a page for every heading, it keeps a short running summary in its head and rewrites that summary after each word it hears.

Hagiwara's language-modeling chapter connects next-token prediction to practical text generation [1]. McRoy's textbook supplies the broader language and application setting [2]. Tokenization belongs to NLP 01; embeddings and their training objectives belong to NLP 04. Sections 1–5 focus on the counting model itself; sections 6–12 replace the count table with a recurrent network, and attention-based models follow in NLP 07 and NLP 09. The general mathematics of recurrent networks (full backpropagation-through-time derivations, gradient bounds, gate equations) is in ML 04; here we take the language-modeling view. Follow one question through this chapter: **what information and evidence determine the probability of the next token?**

1. Three meanings of "n-gram"

An n-gram is a contiguous sequence of n tokens. The tokens may be words, subwords or characters; the chosen tokenization determines what is counted. In the word-token sequence `a cat sleeps`, the unigrams are `a`, `cat`, `sleeps`; the bigrams are `a cat`, `cat sleeps`; and the trigram is `a cat sleeps`.

Do not confuse these three uses:

Use	Object stored or predicted	What it tells us
N-gram occurrence	A contiguous token tuple	A local piece of a particular sequence
Document feature	The number or presence of a tuple in a document	Evidence supplied to a classifier or retrieval model
N-gram language model	A normalized next-token distribution for each retained history	Probabilities used to score or generate a sequence

A document bigram feature such as `not ready` can help a classifier distinguish an operational status. It does not, by itself, specify the probability of `ready` after `not`: we still need competing next-token counts and a normalization rule. Likewise, the probability of a phrase depends on the sentence context under a language model, while a feature count usually pools all of its occurrences within the document.

In a language model, the order n includes the token being predicted. A bigram uses **one previous token**, not two; a trigram uses two. Sentence boundaries require a consistent convention. A beginning marker supplies initial context, and an end marker is a predicted outcome that represents stopping. The examples never predict the beginning marker. Keeping it in the prediction vocabulary would allocate probability to an event that our sentence format does not allow.

2. Exact factorization, limited history

Predicting one token at a time can represent a sequence probability exactly. Throwing away older context is the modeling assumption, rather than a consequence of probability theory.

$$p(x_1, \dots, x_T) = \prod_{t=1}^{T} p(x_t \mid x_{<t}), \qquad p_n(x_1, \dots, x_T) = \prod_{t=1}^{T} p_n(x_t \mid h_t^{(n)}).$$

Here x_t is the token at position t; T counts predicted positions, including an end marker when used; $x_{<t}$ is the complete prefix; n is the n-gram order; $h_t^{(n)}$ is the suffix containing at most the previous $n-1$ tokens, with beginning markers where needed; and p_n denotes the limited-history model. The first equality is the chain rule. The second defines a model whose predictions depend only on the retained suffix. It need not equal the true sequence distribution.

Order	Context before predicting `sleeps` in `a cat sleeps`	Conditional distribution
Unigram, $n = 1$	Empty history	$p_1(\text{sleeps})$
Bigram, $n = 2$	`cat`	$p_2(\text{sleeps} \mid \text{cat})$
Trigram, $n = 3$	`a cat`	$p_3(\text{sleeps} \mid \text{a cat})$

An empty history is still a valid context: the unigram model uses the same next-token distribution everywhere. A bigram makes finer distinctions, but it must pool all prefixes ending in the same token. A trigram can separate prefixes whose last two tokens differ. None of these guarantees that the retained context contains the information needed by a linguistic dependency.

A worked count comparison

The following invented corpus intentionally makes the article informative: examples beginning with `the` end in `sleeps`, and examples beginning with `a` end in `eats`. This is a controlled statistical pattern, not a claim about English. The example fits three orders independently with the same vocabulary and smoothing strength.

```
from collections import Counter, defaultdict

train = ["the cat sleeps", "the dog sleeps", "a cat eats", "a dog eats"]
START, END, UNK = "<s>", "</s>", "<unk>"
vocab = sorted({w for s in train for w in s.split()} | {END, UNK})
alpha = 0.2
models = {}
for order in (1, 2, 3):
    rows = defaultdict(Counter)
    for sentence in train:
        tokens = [START] * (order - 1) + sentence.split() + [END]
        for t in range(order - 1, len(tokens)):
            history = tuple(tokens[t-order+1:t])
            rows[history][tokens[t]] += 1
    models[order] = rows

def probability(order, history, word):
    row = models[order].get(tuple(history), {})
    return (row.get(word, 0) + alpha) / (sum(row.values()) + alpha * len(vocab))

for order, rows in models.items():
    for history in rows:
        assert abs(sum(probability(order, history, w) for w in vocab) - 1) < 1e-10
print("vocabulary size:", len(vocab))
```

```
print("unigram P(eats):", round(probability(1, (), "eats"), 3))
print("bigram P(eats | cat):", round(probability(2, ("cat",), "eats"), 3))
print("trigram P(eats | a cat):", round(probability(3, ("a", "cat"), "eats"), 3))
print("trigram P(eats | the cat):", round(probability(3, ("the", "cat"), "eats"), 3))
print("unseen-history P(eats):", round(probability(3, (UNK, "cat"), "eats"), 3))
assert probability(3, ("a", "cat"), "eats") > probability(3, ("the", "cat"), "eats")
assert abs(probability(3, (UNK, "cat"), "eats") - 1 / len(vocab)) < 1e-10
```

Output:

```
vocabulary size: 8
unigram P(eats): 0.125
bigram P(eats | cat): 0.333
trigram P(eats | a cat): 0.462
trigram P(eats | the cat): 0.077
unseen-history P(eats): 0.125
```

After `cat`, the bigram count table contains one `eats` and one `sleeps`. Both article choices feed that same table. The trigram tables separate the two observations: `a cat` has one observed `eats`, while `the cat` has none. The trigram can therefore express the distinction, but each of its tables has only one observation. More detailed conditioning gives less evidence per context on this corpus.

The unseen history has no outgoing counts. Additive smoothing makes its distribution uniform over all eight prediction tokens. This is a complete probability distribution, yet it discards useful information that the final token `cat` does occur in training. Interpolation with a lower-order distribution can retain that evidence. Normalization and useful generalization are different properties.

3. From counts to normalized probabilities

To estimate a distribution, count each history–target event in training, then divide by the history's outgoing total. Two details make the result reproducible: which symbols are predicted and which contexts count as distinct.

$$\hat{p}(w \mid h) = \frac{c(h, w)}{c(h)}, \qquad c(h) = \sum_{v \in V} c(h, v), \qquad c(h) > 0.$$

Here h is a retained history; w is a candidate next token; V is the prediction vocabulary; v ranges over its members; $c(h, w)$ is the number of training transitions from h to w; $c(h)$ is their total; and $\hat{p}$ is the maximum-likelihood estimate. For a trigram, h is a two-token tuple; for a unigram, it is the empty tuple. The estimate is undefined when the history has never occurred.

Why does the observed relative frequency solve the likelihood problem? For one history, a distribution maximizing $\sum_w c(h, w) \log p(w \mid h)$ subject to probabilities summing to one has its positive-count probabilities proportional to the counts. The normalizing constraint then divides by $c(h)$.

Different histories can be fitted independently because a count table does not share parameters across them.

This simplicity creates two distinct problems. An **unseen transition** has an observed history but a target never counted after it, producing probability zero. An **unseen history** has no count table at all, making the unsmoothed ratio undefined. Smoothing can handle both within a fixed vocabulary. It cannot produce the spelling of a word that the vocabulary has replaced with an unknown marker.

Additive smoothing adds a small positive count to every possible target:

$$p_\alpha(w \mid h) = \frac{c(h, w) + \alpha}{c(h) + \alpha|V|}.$$

Here p_α is the smoothed distribution; $c(h, w)$ and $c(h)$ are transition and outgoing-history counts; w is the candidate token; h is the retained history; V is the prediction vocabulary; $|V|$ is its size; and $\alpha > 0$ is the amount added to each candidate count. Summing the numerator over the vocabulary gives exactly the denominator. Even an unobserved history has a defined distribution: all candidates receive probability $1/|V|$.

The vocabulary size matters as much as the count added to one candidate. With a large vocabulary, allocating α to every token can absorb substantial total probability. A parameter that works for this tiny word vocabulary should not be transferred unchanged to a much larger one without validation.

Code: train, score and sample a bigram model

This complete example needs no downloads or GPU. The corpus is invented for teaching; its test sentences are separate from training. Every test word absent from the training vocabulary maps to the same unknown token.

```
from collections import Counter, defaultdict
import math
import random

train = [
    "the cat sleeps", "the dog sleeps", "the cat runs",
    "the dog runs", "a cat eats", "a dog eats",
    "the bird sings", "a bird sleeps",
]
test = ["the cat eats", "a dog runs", "the bird sleeps", "a robot runs"]
START, END, UNK = "<s>", "</s>", "<unk>"
words = {w for sentence in train for w in sentence.split()}
vocab = sorted(words | {END, UNK})       # START is context, never predicted
alpha = 0.2
counts = defaultdict(Counter)
unigrams = Counter()

def encode(sentence):
    return [w if w in words else UNK for w in sentence.split()] + [END]
```

```
for sentence in train:
    tokens = [START] + encode(sentence)
    for h, w in zip(tokens, tokens[1:]):
        counts[h][w] += 1
        unigrams[w] += 1

def bigram(h, w):
    row = counts.get(h, {})
    return (row.get(w, 0) + alpha) / (sum(row.values()) + alpha * len(vocab))

def unigram(h, w):
    return (unigrams[w] + alpha) / (sum(unigrams.values()) + alpha * len(vocab))

def perplexity(sentences, probability):
    nll, n = 0.0, 0
    for sentence in sentences:
        tokens = [START] + encode(sentence)
        for h, w in zip(tokens, tokens[1:]):
            nll -= math.log(probability(h, w))
            n += 1
    return math.exp(nll / n)

for h in [START] + vocab:
    assert abs(sum(bigram(h, w) for w in vocab) - 1) < 1e-10
print("unigram test perplexity:", round(perplexity(test, unigram), 3))
print("bigram test perplexity:", round(perplexity(test, bigram), 3))
assert UNK in encode(test[-1])

rng = random.Random(7)
h, sample = START, []
for _ in range(20):                      # cap length for this demonstration
    w = rng.choices(vocab, weights=[bigram(h, v) for v in vocab])[0]
    if w == END:
        break
    sample.append(w)
    h = w
print("sample:", " ".join(sample))
```

Output:

```
unigram test perplexity: 10.382
bigram test perplexity: 3.573
sample: a bird sings
```

The assertions check that every conditional distribution sums to one and that “robot” maps to the unknown token. The bigram model is far less “surprised” by the test sentences than the unigram model, because word order carries information. Sampling can still produce awkward strings: positive probability is not a guarantee of grammaticality. The length cap only stops the demonstration; it does not change the probabilities used for scoring.

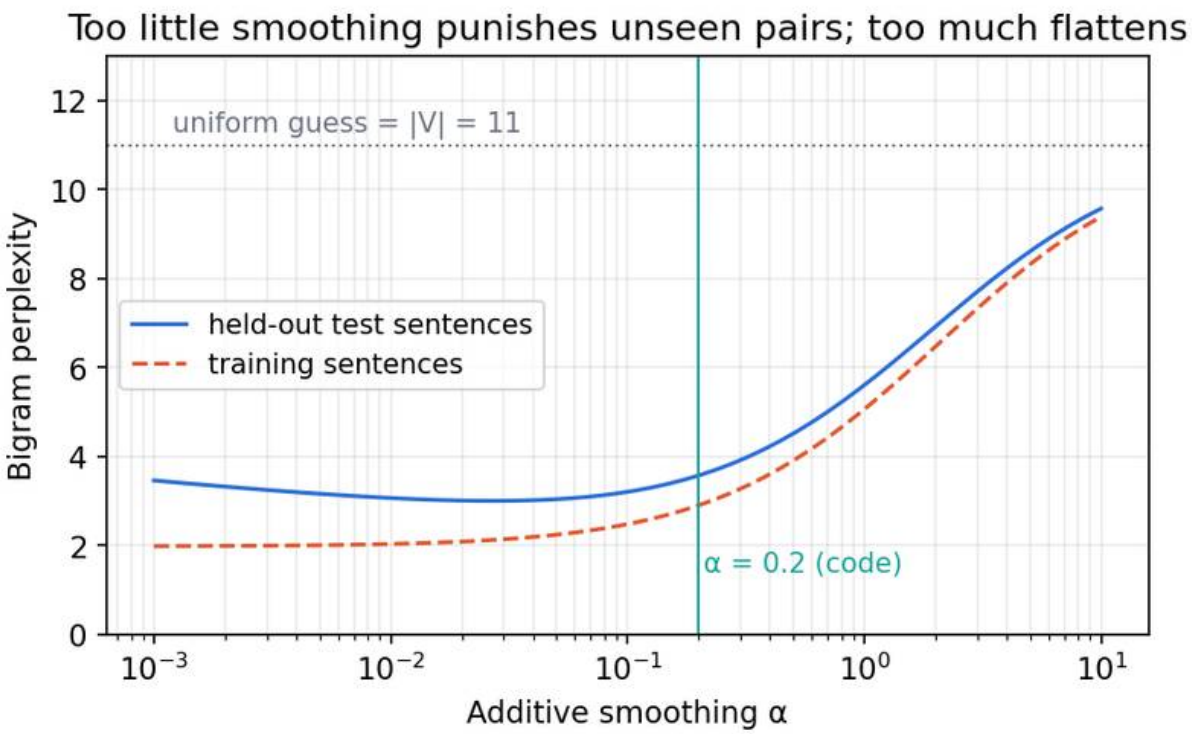


Figure 1 — Smoothing reserves probability for unobserved transitions. The synthetic held-out trace illustrates the tradeoff; four held-out sentences cannot establish a generally optimal setting.

Distinguish smoothing from longer memory

Smoothing addresses missing observations within the chosen representation. It does not restore discarded history. After either "the technician did not repair the" or "the technician repaired the," a bigram sees only `the`. It assigns the same next-token distribution even if the earlier words would make different continuations appropriate. Increasing the corpus can estimate that shared distribution more accurately while leaving the equality intact.

Additive smoothing also has an exact mixture interpretation. For a context with a positive observed count:

$$p_\alpha(w \mid h) = \lambda_h \hat{p}(w \mid h) + (1 - \lambda_h)\frac{1}{|V|}, \qquad \lambda_h = \frac{c(h)}{c(h) + \alpha|V|}, \qquad \hat{p}(w \mid h) = \frac{c(h, w)}{c(h)}.$$

Here p_α is the smoothed distribution, h is the retained history, w is a candidate token, $c(h, w)$ is its observed transition count, $c(h) > 0$ is the total outgoing count, $\hat{p}$ is the unsmoothed estimate, $\alpha > 0$ is the smoothing parameter, $|V|$ is vocabulary size and λ_h is the weight on observations. A context with zero count uses the uniform distribution directly.

The same smoothing parameter changes rare contexts more than frequent ones. As context count grows, the observed evidence receives more weight; as smoothing grows, probability moves toward the uniform distribution. This explains why a held-out curve can improve initially and

then worsen: protecting unseen transitions eventually also dilutes useful observed distinctions. Choose the parameter with validation data rather than the final test set.

A tokenizer and unknown-token rule are part of this experiment. Mapping several unseen words to one marker pools their probabilities. A lower perplexity after that mapping can partly reflect an easier prediction alphabet. It does not show that the model can identify those original words.

Interpolate evidence from several history lengths

When a long context is rare, shorter contexts often have more observations. Interpolation combines their complete distributions instead of assigning every unseen continuation the same probability.

$$p_{\text{mix}}(w \mid u, v) = \lambda_3 p_3(w \mid u, v) + \lambda_2 p_2(w \mid v) + \lambda_1 p_1(w), \qquad \lambda_i \geq 0, \quad \sum_{i=1}^{3} \lambda_i = 1.$$

Here u, v are the two most recent tokens; w is the next token; p_3, p_2, p_1 are normalized trigram, bigram and unigram distributions; p_{mix} is their mixture; and λ_i is the nonnegative weight assigned to order i. Each component must be defined for the queried history, for example by smoothing. Summing over candidate tokens gives one because each component sums to one and the weights also sum to one.

For the unseen context in our small comparison, a mixture can use the observed bigram after `c at` even though the full two-token context is missing. Fixed mixture weights are easy to inspect; history-dependent weights can assign more trust to well-supported longer histories. Choose the weights on validation text. Deciding them after inspecting the final test perplexity makes that score part of model selection.

Interpolation differs from simply substituting a shorter history only when a transition is missing. Such back-off rules must also account for the probability already allocated to observed transitions, so that the complete distribution stays normalized. Chen and Goodman's comparative study examines smoothing choices and their empirical behavior [4]. Our additive and mixture examples expose the mechanics rather than implementing every estimator in that study.

4. What held-out perplexity measures

Scoring measures how much probability a model gives the observed targets. A good score means that these particular token sequences were relatively unsurprising under a particular tokenizer and vocabulary; it does not establish factual accuracy or understanding.

$$\mathcal{L} = -\frac{1}{N} \sum_{j=1}^{N} \log p(w_j \mid h_j), \qquad \text{PPL} = \exp(\mathcal{L}) = \left(\prod_{j=1}^{N} \frac{1}{p(w_j \mid h_j)} \right)^{1/N}.$$

Here N is the number of scored token positions; j indexes those positions across the evaluated sentences; w_j is the observed target; h_j is its permitted history; p is the evaluated model; $\mathcal{L}$ is mean

negative log likelihood in nats per token; and PPL is perplexity. The product form shows that perplexity is the geometric mean of reciprocal target probabilities. It is not the count of plausible continuations at every position.

If every observed target receives probability $1/K$, perplexity equals K, where K is the number of equally likely choices in that hypothetical distribution. Actual predictions are not uniform: one difficult target can contribute a large log loss while many easy targets contribute little. Reporting only an average can conceal concentrated failure on rare names or unfamiliar domains. Inspect losses by word frequency and sentence type as well.

When aggregating batches, add their **total** negative log likelihoods and divide by the **total** number of scored tokens before exponentiating. Neither averaging batch perplexities nor exponentiating an unweighted mean of batch losses generally gives corpus perplexity. The latter is correct only when the batches have equal scored-token counts. Padding and a short final batch make that assumption easy to violate. This synthetic calculation follows directly from the definition above:

```
import math

# Synthetic evaluation batches: 2 targets of probability 1/2, then 8 of 1/4.
counts = [2, 8]
mean_nll = [math.log(2), math.log(4)]
corpus_ppl = math.exp(sum(n * loss for n, loss in zip(counts, mean_nll)) / sum(counts))
print('token-weighted perplexity:', round(corpus_ppl, 3))
print('mean batch perplexity:', round(sum(math.exp(loss) for loss in mean_nll) / 2, 3))
print('exp of mean batch loss:', round(math.exp(sum(mean_nll) / 2), 3))
assert math.isclose(corpus_ppl, (2**2 * 4**8)**(1 / 10))
```

```
token-weighted perplexity: 3.482
mean batch perplexity: 3.0
exp of mean batch loss: 2.828
```

The bigram code counts the end marker, which makes stopping part of both the training problem and the evaluation. It excludes the beginning marker. Its unigram and bigram results are comparable because the test text, tokenization, unknown mapping and number of scored positions match. A comparison across word and character tokenizers has different units. Even the same raw sentence can yield a different score denominator and prediction alphabet.

Train, validation and test text have separate roles. Training determines vocabulary and counts; validation selects order and smoothing; test evaluates the chosen pipeline once. A held-out spelling absent from training must follow the fixed unknown rule. Rebuilding the vocabulary with test text silently changes the experiment. Repeated sentences or material copied across splits can also make memorized local continuations look like generalization.

5. Generation, stopping and the memory ceiling

Generation repeatedly samples from a normalized next-token distribution, appends the sampled token to the history and stops when the end marker appears. Scoring instead supplies the ob-

served history from the text being evaluated. The same probability tables support both procedures, but one examines supplied text while the other produces a stochastic trajectory.

The demonstrated sample `a bird sings` is one draw with a fixed random seed. It is not a representative quality estimate. Examine several independently seeded trajectories and report both local errors and unusually long sequences. A length cap guarantees that a program stops running; if the cap fires before the end marker, the displayed sequence is truncated rather than a model-generated completed sentence.

Choosing the most probable next token at each step gives a deterministic greedy trajectory. It does not generally maximize the probability of the entire sequence: a locally attractive token may lead to poor later continuations. Conversely, sampling preserves distributional variety but may choose low-probability transitions. Generation choices therefore answer a different question from whether the model assigns good probabilities to held-out text.

A failure that more training cannot remove

Consider `the robot near the cabinets is` and `the robots near the cabinet are`. Agreement depends on the head of the subject, rather than simply the immediately preceding noun. If the relevant head lies outside the retained suffix, the model pools examples that require different agreement choices. More observations can improve the pooled estimate, but cannot make that estimate depend on a token outside the model's input.

For any two prefixes with the same final $n-1$ tokens, an n-gram model's next-token distributions must be identical. This is an architectural equality, not an empirical tendency. Increase n and the ceiling moves farther away, while the number of possible histories and the sparsity of their count tables increase. Shared neural representations can connect similar contexts, but their actual behavior still requires testing; merely replacing a table does not establish that an agreement dependency was learned.

6. A recurrent language model

Plain version. An n-gram model looks the history up in a table. A recurrent language model reads the history one token at a time and keeps a fixed-size vector, the hidden state, that it updates after each token. The next-token distribution is computed from that state, so the model is no longer forced to forget everything older than $n-1$ tokens.

Elman's simple recurrent network introduced this idea of a state fed back as input at the next step [6]; Mikolov and colleagues showed that a recurrent network trained as a word-level language model could beat strong smoothed n-gram baselines for speech recognition [7]. Keep the chain rule from section 2 exactly as it was. Only the conditional distribution changes. For a token sequence $x_1, \ldots, x_T$ with a beginning marker x_0:

$$\mathbf{h}_t = \tanh(W_x\,\mathbf{e}(x_{t-1}) + W_h\,\mathbf{h}_{t-1} + \mathbf{b}), \qquad \mathbf{h}_0 = \mathbf{0},$$
$$\mathbf{z}_t = W_o\,\mathbf{h}_t + \mathbf{c},$$
$$p_\theta(x_t = v \mid x_{<t}) = \mathrm{softmax}(\mathbf{z}_t)_v = \frac{\exp z_{t,v}}{\sum_{u \in V} \exp z_{t,u}}.$$

Here $\mathbf{e}(x) \in \mathbb{R}^d$ is the embedding of token x (a row of a learned table, as in NLP 04); $\mathbf{h}_t \in \mathbb{R}^m$ is the hidden state after reading $x_0, \dots, x_{t-1}$; $W_x \in \mathbb{R}^{m \times d}$, $W_h \in \mathbb{R}^{m \times m}$ and $\mathbf{b} \in \mathbb{R}^m$ are the recurrent parameters; $W_o \in \mathbb{R}^{|V| \times m}$ and $\mathbf{c} \in \mathbb{R}^{|V|}$ map the state to one score $z_{t,v}$ per vocabulary item v; V is the prediction vocabulary and u ranges over it; θ collects all parameters; and tanh acts element by element. This is the same recurrence as in ML 03A, written with an embedding as input and a softmax over the vocabulary as output.

Three consequences follow directly from the equations.

- **Unbounded conditioning, bounded memory.** $\mathbf{h}_t$ is a function of the whole prefix, so two prefixes that share their last $n-1$ tokens can receive different predictions. But $\mathbf{h}_t$ has only m numbers. The model must learn what to keep; nothing guarantees that it keeps the information a particular dependency needs.
- **Shared parameters.** The same W_x, W_h and W_o are used at every position. The parameter count does not grow with context length, whereas the number of possible n-gram histories grows as $|V|^{n-1}$.
- **Generalization through similarity.** Similar tokens receive similar embeddings, and similar prefixes can produce similar states. An unseen history no longer falls back to a uniform row: it is mapped to some state, and the prediction made from that state can reuse what similar histories taught the model.

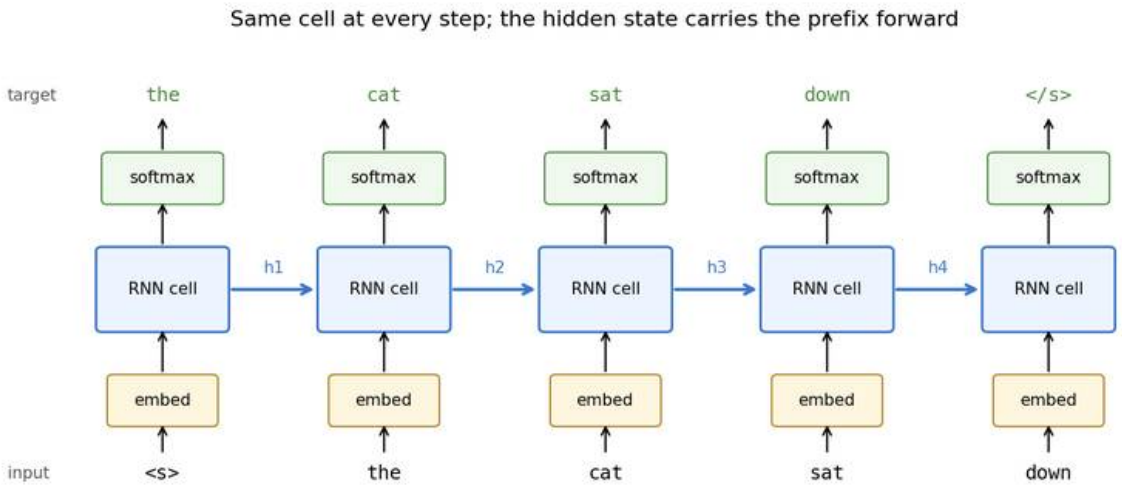


Figure 2 — A recurrent language model unrolled over five steps. Each input is the previous token; each target is the next token; the same cell parameters are reused at every step.

The training loss is next-token cross-entropy

Training minimizes the same quantity that section 4 used for evaluation, now as a differentiable function of θ:

$$\mathcal{L}(\theta) = -\frac{1}{N}\sum_{j=1}^{N} \log p_\theta(x_j \mid x_{<j}), \qquad \text{PPL} = \exp \mathcal{L}, \qquad \text{bits per token} = \frac{\mathcal{L}}{\ln 2}.$$

Here N is the number of supervised target positions (padding and other ignored positions excluded), x_j is the observed target at position j, $x_{<j}$ is its prefix, and $\ln 2$ converts nats to bits. The per-position term is the cross-entropy between the one-hot target and the model's softmax distribution; IT 01 develops cross-entropy and its link to coding length. Character-level work often reports bits per character instead of perplexity; the two carry the same information.

Two implementation details carry over from the count model. First, **shift by one**: the input sequence is $x_0, \ldots, x_{T-1}$ and the target sequence is $x_1, \ldots, x_T$. Feeding a position's own token as both input and target creates a trivial copying shortcut. Second, **teacher forcing**: during training each step receives the observed previous token, not the model's own earlier sample. At generation time, the model's samples become its inputs, so errors can feed on themselves. Training loss and sampling behavior therefore need separate checks, exactly as section 5 argued for the n-gram model.

7. Training through time: what gradients can reach

Plain version. Unrolling the recurrence turns the model into a very deep network with tied weights. Gradients from a prediction must travel back through every intermediate step to reach the token that should have influenced it. Over long distances they tend to shrink to almost nothing or, less often, blow up.

Backpropagation through time (BPTT) applies ordinary backpropagation to the unrolled graph and sums the gradient contributions for each shared weight [8]. The sensitivity of a later state to an earlier one is an ordered product of one-step Jacobians:

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_s} = \prod_{k=s+1}^{t} \operatorname{diag}(1 - \mathbf{h}_k^2)\, W_h,$$

where $s < t$ are positions, $\mathbf{h}_k^2$ squares each element, and $\operatorname{diag}(\cdot)$ builds a diagonal matrix from the tanh derivatives. When the factors are typically smaller than one in norm, the product decays exponentially with distance $t - s$ (vanishing gradients); when they are larger, it can grow exponentially (exploding gradients). Bengio, Simard and Frasconi analyzed why learning long-term dependencies with gradient descent is hard [9]; Pascanu, Mikolov and Bengio revisited the problem and popularized clipping the gradient norm as the remedy for explosions [10]. ML 03A derives these results in full and measures them.

What does this mean for language? Return to the agreement example from section 5: in `the robots near the cabinet are`, the verb depends on `robots`, four words back, and the intervening

noun points the other way. A plain RNN *can* represent a state that remembers the plural subject, but the training signal that would teach it to do so must survive the trip from `are` back to `robot s`. At the word level that is a few steps; at the character level the same dependency spans roughly twenty steps. A model can therefore have enough capacity for a dependency and still fail to learn it, which is a different failure from the n-gram's architectural ceiling.

8. Gated cells: LSTM and GRU

Plain version. Gated cells add a memory path that is updated mainly by addition and controlled by learned switches. Information, and gradient, can then pass across many steps without being squashed at every one.

The long short-term memory (LSTM) cell of Hochreiter and Schmidhuber [11], with the forget gate added by Gers, Schmidhuber and Cummins [12], keeps a cell state $\mathbf{c}_t$ next to the hidden state:

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t, \qquad \mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t).$$

Here $\mathbf{f}_t$, $\mathbf{i}_t$ and $\mathbf{o}_t$ are the forget, input and output gates (vectors in $(0,1)^m$ computed by sigmoid layers from the current input and $\mathbf{h}_{t-1}$), $\tilde{\mathbf{c}}_t$ is a tanh candidate computed the same way, and $\odot$ is element-wise multiplication. Along the cell path, $\partial \mathbf{c}_t / \partial \mathbf{c}_{t-1}$ is approximately $\operatorname{diag}(\mathbf{f}_t)$: no weight matrix, no squashing. When a forget gate stays near one, a value written at the plural subject can survive until the verb.

The gated recurrent unit (GRU) of Cho and colleagues [13] merges cell and hidden state and uses two gates:

$$\mathbf{h}_t = \mathbf{z}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{z}_t) \odot \tilde{\mathbf{h}}_t,$$

where $\mathbf{z}_t \in (0,1)^m$ is the update gate and $\tilde{\mathbf{h}}_t$ is a candidate state computed with a reset gate. (Some sources swap the roles of $\mathbf{z}_t$ and $1 - \mathbf{z}_t$.) The full gate equations, a from-scratch LSTM step checked against PyTorch, and parameter counts are in ML 03A. For language modeling the practical summary is: LSTMs and GRUs train far more reliably than plain tanh RNNs, neither dominates the other across tasks, and gating eases rather than removes the difficulty of very long dependencies.

9. Stacked and bidirectional recurrent networks

Plain version. Recurrent layers can be stacked so that a higher layer reads the state sequence of the layer below. Running a second network right to left gives each position a view of its future too, which is useful for labeling text but forbidden for predicting the next token.

A **stacked** (deep) RNN feeds $\mathbf{h}_1^{(\ell)}, \dots, \mathbf{h}_T^{(\ell)}$ from layer ℓ as the input sequence of layer $\ell + 1$, and the softmax reads the top layer. Two to four layers are common in recurrent language models; dropout between layers is the usual regularizer [16].

A **bidirectional** RNN runs one network forward and one backward and concatenates their states, $\mathbf{h}_t = [\overrightarrow{\mathbf{h}}_t; \overleftarrow{\mathbf{h}}_t]$ [14]. The backward state at position t has read $x_{t+1}, \dots, x_T$. That is ideal for tagging,

where the whole sentence is given and the label of a word may depend on words to its right; it is the main encoder of NLP 06. It is *wrong* for a left-to-right language model: if the state used to predict x_t has seen x_t, the loss rewards copying and the perplexity becomes meaningless. A useful sanity check for any language model is that changing a future token leaves every earlier prediction unchanged.

Online only: Seq2Seq — Sequence to Sequence Learning with Neural Networks — https://blog.shyandram.dev/en/learn/nlp/recurrent-neural-networks/papers/seq2seq–sequence-to-sequence-learning-with-neural-networks/

10. Character-level language models

Plain version. The prediction unit can be a character instead of a word. The vocabulary becomes tiny and no spelling is ever unknown, but sequences become several times longer and the model must learn spelling before it can learn grammar.

Character-level RNNs were popular demonstrations of what recurrent networks learn; Graves showed that LSTMs trained on characters can generate long, locally coherent text and handwriting [15]. The tradeoffs compared with word or subword units are concrete:

Choice	Vocabulary	Unknown inputs	Sequence length	Long-range cost
Words	Large; rare words need an unknown marker	Mapped to one marker; spelling lost	Shortest	Fewest steps between related words
Subwords (NLP 01)	Medium, fixed by the tokenizer	Split into known pieces	Medium	Medium
Characters	Tiny alphabet	Any spelling over the alphabet	Longest	Many steps between related words

Two cautions follow from the evaluation rules in section 4. First, character perplexity and word perplexity have different units; a character model with perplexity 11 is not "better" than a word model with perplexity 3.6, because each counts a different number of predictions over a different alphabet. For a comparison using a common text unit, report total negative log likelihood (or bits) for the same text and, if needed, divide by a common byte count. This requires compatible, lossless text preprocessing and declared boundary conventions: replacing unknown spellings with one marker does not encode the missing characters. A change of denominator cannot repair information erased before scoring. Second, a fixed character alphabet still needs a policy for symbols outside it, such as accented letters or emoji. Decide it before looking at held-out text.

11. Sampling and temperature

Plain version. A trained model gives a probability for every next token. To generate, we draw one, append it, and repeat. A temperature setting makes the draw more cautious or more adventurous without retraining.

For logits $\mathbf{z}_t$ and temperature $\tau > 0$, sample from

$$q_\tau(v) = \frac{\exp(z_{t,v}/\tau)}{\sum_{u \in V} \exp(z_{t,u}/\tau)}.$$

Here v and u range over the vocabulary V, $z_{t,v}$ is the model's score for v at step t, and q_τ is the sampling distribution. $\tau = 1$ samples from the model's own distribution; $\tau \to 0$ approaches greedy selection of the highest-scoring token; $\tau > 1$ flattens the distribution and gives more weight to unlikely tokens. Temperature changes only the generation procedure. It does not change the probabilities used to compute perplexity, so a model's held-out score is the same at every temperature.

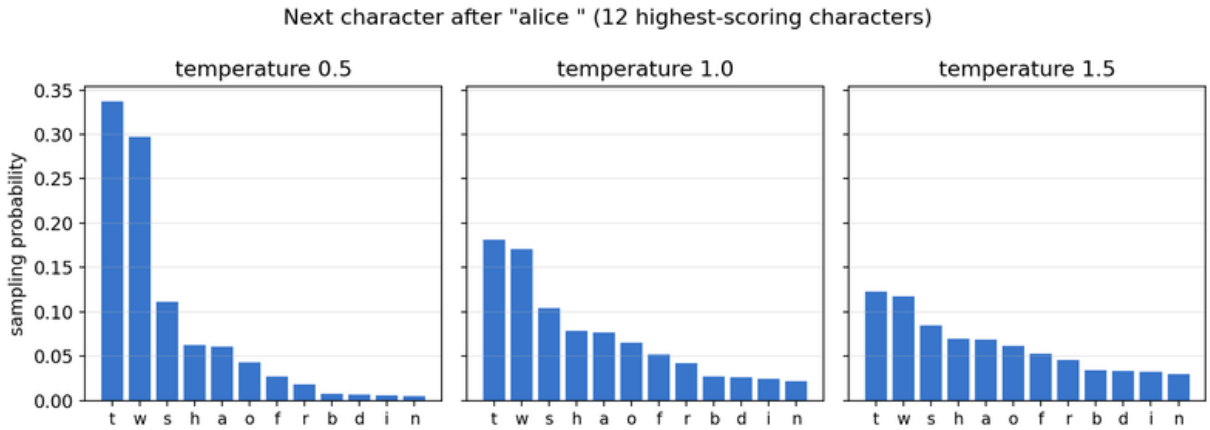


Figure 3 — Next-character distribution of the trained LSTM from section 12 after the prompt "alice", at three temperatures. The logits are identical in all three panels; only the sampling distribution changes.

Generation adds stopping rules, length limits and search strategies such as beam search and nucleus sampling. Those belong to NLP 08 and NLP 09. One warning applies already: a sample is a single random trajectory. Judge a model by held-out likelihood and task checks, and use samples to look for qualitative failure patterns, not to rank models.

12. Practical training, then a complete experiment

The training checklist

Plain version. Most failed recurrent language models fail for mundane reasons: padding counted as text, gradients that explode, states leaking between unrelated documents, or a model selected on the test set. Each has a standard remedy.

- **Packing a corpus into streams.** For a continuous text, cut the token stream into B equal parallel streams and lay them out as a $B \times n$ matrix. Each training step reads the next T columns. Every position is a real token, so no padding is needed. Stream b continues exactly where it stopped in the previous window.

- **Padding and masking.** When examples are separate sentences of different lengths, pad them to a common length and keep a Boolean mask. The mask must remove padded positions from the loss (otherwise the model is rewarded for predicting padding) and, for bidirectional encoders, from the state updates; PyTorch's `pack_padded_sequence` does the latter. NLP 06 implements both in a tagger.
- **Truncated BPTT.** Backpropagating through thousands of steps costs memory linear in the length. Instead, carry the hidden state from one window of T steps to the next but *detach* it, so the forward pass still sees the whole past while gradients flow back at most T steps.
- **Gradient clipping.** If the gradient norm $\|\mathbf{g}\|$ exceeds a threshold κ, rescale it to $\kappa\,\mathbf{g}/\|\mathbf{g}\|$ [10]. This keeps the update direction and caps its length.
- **Regularization and early stopping.** Dropout on non-recurrent connections [16] and weight decay limit memorization; select the epoch with the lowest *validation* perplexity, then report test perplexity once.
- **Fair baselines.** Melis, Dyer and Blunsom found that carefully tuned standard LSTMs outperformed several more elaborate recent architectures once all were given comparable hyperparameter search [17]. The same principle applies at small scale: tune the n-gram baseline on the same validation data before claiming that the neural model wins.

Code: a character-level LSTM against an interpolated n-gram

The text is the opening of Chapter I of Lewis Carroll's *Alice's Adventures in Wonderland* (1865), which is in the public domain; it was copied from Project Gutenberg eBook #11 [24] with italic underscores removed, curly quotes straightened and em dashes written as `--`, then lowercased. Paragraphs 1–7 are training data, paragraph 8 is validation and paragraph 9 is the test set. The alphabet is fixed in advance, before the held-out text is inspected. Both models condition each held-out character on all earlier text. The n-gram baseline interpolates smoothed unigram, bigram and trigram character distributions exactly as in section 3, with weights and smoothing chosen on validation. The LSTM is trained three times with different seeds, because a single run on so little data can mislead. The whole script runs on a CPU in well under a minute and downloads nothing.

```
import copy
import math
from collections import Counter, defaultdict

import torch
from torch import nn

# Opening paragraphs of Chapter I of Lewis Carroll's "Alice's Adventures in
# Wonderland" (1865, public domain; Project Gutenberg eBook #11). Italic
# underscores removed, curly quotes straightened, em dashes written as "--".
PARAGRAPHS = [
    'Alice was beginning to get very tired of sitting by her sister on the '
    'bank, and of having nothing to do: once or twice she had peeped into the '
    'book her sister was reading, but it had no pictures or conversations in '
    'it, "and what is the use of a book," thought Alice "without pictures or '
```

```
'conversations?"',
'So she was considering in her own mind (as well as she could, for the hot '
'day made her feel very sleepy and stupid), whether the pleasure of making '
'a daisy-chain would be worth the trouble of getting up and picking the '
'daisies, when suddenly a White Rabbit with pink eyes ran close by her.',
'There was nothing so very remarkable in that; nor did Alice think it so '
'very much out of the way to hear the Rabbit say to itself, "Oh dear! Oh '
'dear! I shall be late!" (when she thought it over afterwards, it occurred '
'to her that she ought to have wondered at this, but at the time it all '
'seemed quite natural); but when the Rabbit actually took a watch out of '
'its waistcoat-pocket, and looked at it, and then hurried on, Alice started'
' to her feet, for it flashed across her mind that she had never before '
'seen a rabbit with either a waistcoat-pocket, or a watch to take out of '
'it, and burning with curiosity, she ran across the field after it, and '
'fortunately was just in time to see it pop down a large rabbit-hole under '
'the hedge.',
'In another moment down went Alice after it, never once considering how in '
'the world she was to get out again.',
'The rabbit-hole went straight on like a tunnel for some way, and then '
'dipped suddenly down, so suddenly that Alice had not a moment to think '
'about stopping herself before she found herself falling down a very deep '
'well.',
'Either the well was very deep, or she fell very slowly, for she had plenty'
' of time as she went down to look about her and to wonder what was going '
'to happen next. First, she tried to look down and make out what she was '
'coming to, but it was too dark to see anything; then she looked at the '
'sides of the well, and noticed that they were filled with cupboards and '
'book-shelves; here and there she saw maps and pictures hung upon pegs. She'
' took down a jar from one of the shelves as she passed; it was labelled '
'"ORANGE MARMALADE", but to her great disappointment it was empty: she did '
'not like to drop the jar for fear of killing somebody underneath, so '
'managed to put it into one of the cupboards as she fell past it.',
'"Well!" thought Alice to herself, "after such a fall as this, I shall '
"think nothing of tumbling down stairs! How brave they'll all think me at "
"home! Why, I wouldn't say anything about it, even if I fell off the top of"
' the house!" (Which was very likely true.)',
'Down, down, down. Would the fall never come to an end? "I wonder how many '
'miles I\'ve fallen by this time?" she said aloud. "I must be getting '
'somewhere near the centre of the earth. Let me see: that would be four '
'thousand miles down, I think--" (for, you see, Alice had learnt several '
'things of this sort in her lessons in the schoolroom, and though this was '
'not a very good opportunity for showing off her knowledge, as there was no'
' one to listen to her, still it was good practice to say it over) "--yes, '
"that's about the right distance--but then I wonder what Latitude or "
'Longitude I\'ve got to?" (Alice had no idea what Latitude was, or Longitude'
' either, but thought they were nice grand words to say.)',
'Presently she began again. "I wonder if I shall fall right through the '
"earth! How funny it'll seem to come out among the people that walk with "
'their heads downward! The Antipathies, I think--" (she was rather glad '
"there was no one listening, this time, as it didn't sound at all the right"
' word) "--but I shall have to ask them what the name of the country is, '
'you know. Please, Ma\'am, is this New Zealand or Australia?" (and she tried'
" to curtsey as she spoke--fancy curtseying as you're falling through the "
'air! Do you think you could manage it?) "And what an ignorant little girl '
"she'll think me for asking! No, it'll never do to ask: perhaps I shall see"
' it written up somewhere."',
```

```
]

paras = [p.lower() for p in PARAGRAPHS]
train = "\n".join(paras[:7]) + "\n"     # paragraphs 1-7: fit parameters
valid = paras[7] + "\n"                 # paragraph 8: choose settings
test = paras[8] + "\n"                  # paragraph 9: report once
ALPHABET = "\n !\"'(),-.:;?abcdefghijklmnopqrstuvwxyz"  # fixed in advance
V = len(ALPHABET)
assert set(train + valid + test) <= set(ALPHABET)
print("characters train/valid/test:", len(train), len(valid), len(test))
print("letters never seen in training:", sorted(set(valid + test) - set(train)))

# ---- Baseline: interpolated character n-grams (orders 1-3) ----
def count(order):
    rows = defaultdict(Counter)
    for t in range(len(train)):
        rows[train[max(0, t - order + 1):t]][train[t]] += 1
    return rows

tables = {k: count(k) for k in (1, 2, 3)}

def p_add(k, history, ch, alpha):
    row = tables[k].get(history[len(history) - k + 1:] if k > 1 else "", {})
    return (row.get(ch, 0) + alpha) / (sum(row.values()) + alpha * V)

def ngram_ppl(prefix, target, lams, alpha):
    full, nll = prefix + target, 0.0
    for pos in range(len(prefix), len(full)):
        h = full[max(0, pos - 2):pos]
        p = sum(l * p_add(k, h, full[pos], alpha) for k, l in zip((1, 2, 3), lams))
        nll -= math.log(p)
    return math.exp(nll / len(target))

grid = [(l1, l2, round(1 - l1 - l2, 2)) for l1 in (0.1, 0.2, 0.3)
        for l2 in (0.2, 0.3, 0.4, 0.5) if l1 + l2 < 1]
best_cfg = min(((ngram_ppl(train, valid, lams, a), lams, a) for lams in grid
                for a in (0.01, 0.1, 0.5)), key=lambda r: r[0])
ng_valid, ng_lams, ng_alpha = best_cfg
ng_test = ngram_ppl(train + valid, test, ng_lams, ng_alpha)
print(f"n-gram  weights={ng_lams} alpha={ng_alpha}  "
      f"valid PPL={ng_valid:.2f}  test PPL={ng_test:.2f}")

tri_alpha = min((0.01, 0.1, 0.5), key=lambda a: ngram_ppl(train, valid, (0, 0, 1), a))
tri_test = ngram_ppl(train + valid, test, (0, 0, 1), tri_alpha)
print(f"trigram only  alpha={tri_alpha}  test PPL={tri_test:.2f}")

# ---- Character-level LSTM language model ----
stoi = {c: i for i, c in enumerate(ALPHABET)}
encode = lambda s: torch.tensor([stoi[c] for c in s])

class CharLSTM(nn.Module):
    def __init__(self, vocab, emb=24, hidden=96, dropout=0.3):
        super().__init__()
        self.emb = nn.Embedding(vocab, emb)
        self.rnn = nn.LSTM(emb, hidden, batch_first=True)
        self.drop = nn.Dropout(dropout)
```

```
        self.out = nn.Linear(hidden, vocab)

    def forward(self, x, state=None):
        h, state = self.rnn(self.emb(x), state)
        return self.out(self.drop(h)), state

def lstm_ppl(model, prefix, target):
    model.eval()
    with torch.no_grad():
        ids = encode(prefix + target).unsqueeze(0)
        logits, _ = model(ids[:, :-1])          # condition on all earlier text
        nll = nn.functional.cross_entropy(logits[0, len(prefix) - 1:],
                                          ids[0, len(prefix):])
    model.train()
    return math.exp(nll.item())

def train_lstm(seed, epochs=80, B=16, T=48):
    torch.manual_seed(seed)
    model = CharLSTM(V)
    opt = torch.optim.Adam(model.parameters(), lr=3e-3, weight_decay=1e-4)
    ids = encode(train)
    n = (len(ids) - 1) // B                     # B parallel streams of n characters
    X, Y = ids[:n * B].view(B, n), ids[1:n * B + 1].view(B, n)  # targets shifted by one
    history, best = [], (float("inf"), None, 0)
    for epoch in range(1, epochs + 1):
        state, total, steps = None, 0.0, 0
        for i in range(0, n, T):
            if state is not None:               # truncated BPTT: keep values, cut graph
                state = tuple(s.detach() for s in state)
            logits, state = model(X[:, i:i + T], state)
            loss = nn.functional.cross_entropy(logits.reshape(-1, V),
                                               Y[:, i:i + T].reshape(-1))
            opt.zero_grad()
            loss.backward()
            nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            opt.step()
            total, steps = total + loss.item(), steps + 1
        v = lstm_ppl(model, train, valid)
        history.append((epoch, math.exp(total / steps), v))
        if v < best[0]:                         # early stopping on validation only
            best = (v, copy.deepcopy(model.state_dict()), epoch)
    model.load_state_dict(best[1])
    return model, best, history

runs = {}
for seed in (0, 1, 2):
    m, best, hist = train_lstm(seed)
    runs[seed] = (m, hist, lstm_ppl(m, train + valid, test))
    print(f"LSTM seed={seed}  best epoch={best[2]}  valid PPL={best[0]:.2f}  "
          f"test PPL={runs[seed][2]:.2f}")
mean_test = sum(r[2] for r in runs.values()) / len(runs)
print(f"LSTM mean test PPL over 3 seeds: {mean_test:.2f}")
assert mean_test < tri_test
model = runs[0][0]

# ---- Sampling at several temperatures ----
```

```
def sample(prompt, tau, length=80, seed=1):
    g = torch.Generator().manual_seed(seed)
    model.eval()
    with torch.no_grad():
        logits, state = model(encode(prompt).unsqueeze(0))
        out = prompt
        for _ in range(length):
            probs = torch.softmax(logits[0, -1] / tau, dim=-1)
            nxt = torch.multinomial(probs, 1, generator=g)
            out += ALPHABET[nxt.item()]
            logits, state = model(nxt.view(1, 1), state)
    return out.replace("\n", " / ")

for tau in (0.5, 1.0, 1.5):
    print(f"tau={tau}:", sample("alice ", tau))

# ---- Same last two characters, different earlier context ----
def next_dist(prefix):
    model.eval()
    with torch.no_grad():
        logits, _ = model(encode(prefix).unsqueeze(0))
    return torch.softmax(logits[0, -1], dim=-1)

a, b = "alice was beginning to get ", "the rabbit-hole went "
assert a[-2:] == b[-2:]                    # a trigram sees only these two
def ngram_dist(prefix):
    return torch.tensor([sum(l * p_add(k, prefix[-2:], c, ng_alpha)
                             for k, l in zip((1, 2, 3), ng_lams)) for c in ALPHABET])

tv = lambda p, q: 0.5 * (p - q).abs().sum().item()
print("total variation, n-gram:", round(tv(ngram_dist(a), ngram_dist(b)), 3),
      " LSTM:", round(tv(next_dist(a), next_dist(b)), 3))
```

Output:

```
characters train/valid/test: 2633 706 682
letters never seen in training: ['z']
n-gram  weights=(0.1, 0.4, 0.5) alpha=0.01  valid PPL=9.70  test PPL=11.24
trigram only  alpha=0.1  test PPL=13.37
LSTM seed=0  best epoch=43  valid PPL=10.13  test PPL=12.09
LSTM seed=1  best epoch=41  valid PPL=9.76  test PPL=11.05
LSTM seed=2  best epoch=39  valid PPL=10.09  test PPL=11.30
LSTM mean test PPL over 3 seeds: 11.48
tau=0.5: alice whas she sor of the to her fear the she was she sely and it she save of the
 ↪  was
tau=1.0: alice whas nyt ook "hente tome urabuint, mubbeit-insiwselly bulith ver to welf was
 ↪  bor
tau=1.5: alice wherany fowh ".pvenst gime feurdt, )ubger and iwselly ?hont  ver, vermef wamen
 ↪  r
total variation, n-gram: 0.0  LSTM: 0.197
```

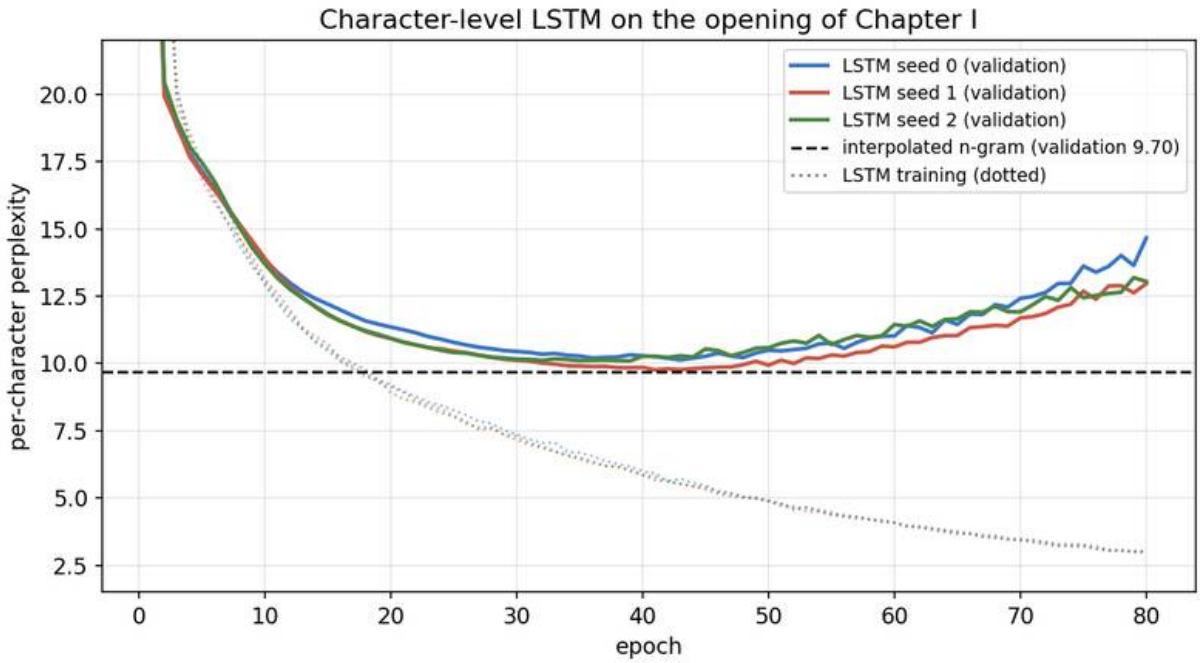


Figure 4 — Per-character perplexity during training. Training perplexity keeps falling; validation perplexity reaches its minimum around epoch 40 and then rises as the model memorizes the 2,633 training characters. The dashed line is the tuned interpolated n-gram on the same validation paragraph.

Reading the results

The baseline matters. A single trigram with additive smoothing reaches a test perplexity of 13.37, and every LSTM run beats it. The interpolated n-gram, tuned on the same validation paragraph, reaches 11.24, while the three LSTM runs reach 12.09, 11.05 and 11.30 (mean 11.48). On 2,633 training characters the recurrent model is roughly level with a well-smoothed count model and clearly better than a poorly smoothed one. This is the small-data version of the point made by Melis and colleagues [17]: an improvement claimed against an untuned baseline may vanish against a tuned one. Neural language models pull ahead as data grows, because shared representations can exploit many weak regularities that sparse tables cannot pool; Jozefowicz and colleagues document how far recurrent language models improved with large corpora and models [19]. Our experiment is far too small to show that regime, and it does not need to: it shows how to make the comparison honestly.

Seeds matter. The spread across seeds (11.05 to 12.09) is larger than the gap between the mean LSTM and the n-gram. With one seed, we could have reported either "the LSTM wins" or "the LSTM loses". Report several runs, or a confidence interval over held-out documents, whenever differences are this small.

The validation curve shows memorization. In Figure 4, training perplexity keeps falling far below the validation value. The model is learning the specific training paragraphs. Early stopping

picks epochs 39–43; without it, the final epoch would be considerably worse on held-out text.

Units. These are per-character perplexities over a 39-symbol alphabet; $\log_2 11.24 \approx 3.49$ bits per character for the n-gram. They cannot be compared with the word-level values of section 3 (for example 3.573), which count different events over a different vocabulary.

The memory ceiling, measured. The final lines compare next-character distributions after `alic e was beginning to get` and `the rabbit-hole went`, two prefixes whose last two characters agree. The interpolated trigram must give identical distributions (total variation 0.0), exactly as section 5 proved. The LSTM's distributions differ (total variation 0.197), because its state depends on the earlier characters. This confirms that the recurrent model *uses* longer context. Whether it uses it *well* is a separate question, which held-out perplexity and targeted tests answer.

Samples. At $\tau = 0.5$ the sample repeats frequent fragments such as `she was` and `of the`; at $\tau = 1.0$ word shapes appear with many misspellings; at $\tau = 1.5$ the output breaks into rare punctuation and random letters. A character model trained on fewer than three thousand characters has learned frequent letter sequences, spaces and some short words. It has not learned English. These are single draws with one seed, so read them as illustrations of the temperature effect rather than as quality estimates.

Modern view

Survey perspective. Jurafsky and Martin's August 2026 draft retains a dedicated n-gram chapter, organized around probabilistic prediction, estimation, evaluation, generation and smoothing [3]. It is a useful conceptual bridge: readers can inspect local counts before encountering models that share learned representations. Chen and Goodman's smoothing comparison explains why the estimator matters in addition to the history length [4]. These sources serve different purposes: the textbook develops the model, while the empirical study compares ways to allocate probability with sparse evidence. Tiezzi and colleagues' 2024 survey reviews why recurrent processing has returned in the Transformer era [20]. It frames two trends under one umbrella of recurrence: models that mix Transformer and recurrent ideas to avoid the quadratic cost of self-attention, and deep state-space models that act as a special class of linear RNNs. Its main open problem is learning from streams whose length is not known in advance, which connects sequence modeling to lifelong online learning.

What recurrent language models achieved. From Mikolov's RNN language model [7] to large-scale LSTM language models [19] and carefully regularized LSTMs [18], recurrent networks were the strongest general-purpose language models until Transformers replaced them for most large-scale work. Merity and colleagues' AWD-LSTM study shows how much of that strength came from regularization and optimization choices rather than from new cell designs [18]; Melis and colleagues reached a similar conclusion through controlled hyperparameter search [17].

2023–2026. Recurrence has returned in new forms aimed at the cost of attention over long contexts. RWKV reformulates a Transformer-like model so it can be trained in parallel and run as an RNN at inference [21]. Mamba uses input-dependent (selective) state-space updates computed with a parallel scan [22]. xLSTM revisits the LSTM with exponential gating and new memory

structures [23]. These models keep the property that made RNN language models attractive: a fixed-size state, so generation cost per token does not grow with context length. They also inherit the question raised in section 7: a fixed-size state must decide what to forget. ML 03A covers the state-space side in more detail.

Evaluation. Chang and colleagues' 2024 survey organizes large-language-model evaluation around what, where and how to evaluate [5]. It covers task behavior and evaluation methods beyond token prediction. Applied here, a lower perplexity should be accompanied by the task checks that matter: spelling recovery, agreement, meaning preservation or an appropriate stopping rule. The practical role of an n-gram baseline in a 2023–2026 workflow is to make a question measurable. Does a more elaborate model improve held-out predictions under the same tokenization? Does it distinguish two prefixes that the chosen n-gram history must merge? Does its added memory and latency improve the downstream result? Answer with matched experiments rather than comparing unmatched scores or assuming that a longer context is always used effectively.

Open problems. Open problems include sparse evidence after domain changes, probabilities for new names, and interactions between tokenization and measured difficulty. Character or subword tokens change the tradeoff between vocabulary coverage and sequence length. Longer histories, whether held in a table or in a recurrent state, offer more information but create harder estimation and credit-assignment problems. Choose the prediction unit and retained history together, and state their limits before interpreting a result.

Key takeaways

- An n-gram occurrence or document feature is not automatically a normalized language model; order n uses $n-1$ preceding tokens, while the chain rule itself does not require limited memory.
- Smoothing protects unobserved events, and interpolation reuses shorter-history evidence; neither can recover history that the model discarded.
- Perplexity comparisons need matching prediction units, unknown mapping and boundary conventions; character and word perplexities are not comparable.
- A recurrent language model replaces the count table with a learned state that depends on the whole prefix, with parameters shared across positions.
- BPTT gradients shrink or grow with distance; clipping handles explosions, and LSTM or GRU gating eases vanishing gradients without removing the difficulty of long dependencies.
- Bidirectional RNNs suit labeling, not left-to-right language modeling.
- Practical training needs shifted targets, masks or packed streams, truncated BPTT, clipping, regularization and validation-based early stopping.
- On tiny data, a tuned interpolated n-gram can match an LSTM; compare against tuned baselines and report several seeds.

Exercises

1. Reproduce the smoothed trigram probabilities after `a cat` and `the cat` by hand in the first code example of section 2. Check the sum over all eight target tokens.

   Hint Each history has one observation. The denominator is one plus the smoothing strength multiplied by eight; only the observed continuation receives an additional count.

2. Fit a mixture of the first example's three orders. Choose any nonnegative weights summing to one, and verify normalization for an unseen two-token history.

   Hint Evaluate every component on the same candidate vocabulary. Compare the probability after the unseen history with the pure uniform fallback and the observed bigram after `cat`.

3. Score the bigram test set both including and excluding end markers. Explain why the resulting perplexities answer different prediction questions.

   Hint Change both the accumulated loss and the position count consistently. Excluding end markers removes the evaluation of stopping; it is not merely changing a display convention.

4. Construct two prefixes with identical last two tokens whose appropriate next-token probabilities should differ because of earlier negation, agreement or discourse. State what a trigram can and cannot learn about them, and what an RNN language model could learn about them in principle.

   Hint First confirm that the retained history is exactly equal. Then separate a lack of data from the impossibility of conditioning on the omitted distinction. For the RNN, distinguish what its state can represent from what its gradients can teach it.

5. In the character-level experiment, replace `nn.LSTM` with `nn.GRU` (the state becomes a single tensor, so adjust the detach line) and with `nn.RNN`. Compare validation and test perplexity over three seeds. Then remove gradient clipping from the plain RNN and record what happens to the training loss.

   Hint `nn.GRU` and `nn.RNN` return a tensor rather than a tuple as their state. Keep every other setting fixed, including the early-stopping rule, so that the cell type is the only change.

6. Deliberately break causality: give the LSTM model an extra backward LSTM over the same input (a bidirectional layer) and retrain. Explain the training and validation perplexities you observe, and write a test that would have caught the bug.

   Hint With a backward pass, the state used to predict the character at position t has already read that character. A good test changes one future character and checks that all earlier predicted distributions are unchanged.

References

1. Masato Hagiwara, *Real-World Natural Language Processing: Practical Applications with Deep Learning*, Chapter 5, Manning, 2021. Publisher
2. Susan McRoy, *Principles of Natural Language Processing*, 2021. Chapter
3. Daniel Jurafsky and James H. Martin, *Speech and Language Processing*, third-edition draft, Chapter 3, August 19, 2026. Chapter
4. Stanley F. Chen and Joshua Goodman, "An Empirical Study of Smoothing Techniques for Language Modeling," ACL, 1996. Paper
5. Yupeng Chang et al., "A Survey on Evaluation of Large Language Models," ACM Transactions on Intelligent Systems and Technology, 2024. Paper
6. Jeffrey L. Elman, "Finding Structure in Time," *Cognitive Science* 14(2), 1990. DOI
7. Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černocký and Sanjeev Khudanpur, "Recurrent Neural Network Based Language Model," Interspeech, 2010. DOI
8. Paul J. Werbos, "Backpropagation Through Time: What It Does and How to Do It," *Proceedings of the IEEE* 78(10), 1990. DOI
9. Yoshua Bengio, Patrice Simard and Paolo Frasconi, "Learning Long-Term Dependencies with Gradient Descent Is Difficult," *IEEE Transactions on Neural Networks* 5(2), 1994. DOI
10. Razvan Pascanu, Tomas Mikolov and Yoshua Bengio, "On the Difficulty of Training Recurrent Neural Networks," ICML, 2013. arXiv
11. Sepp Hochreiter and Jürgen Schmidhuber, "Long Short-Term Memory," *Neural Computation* 9(8), 1997. DOI
12. Felix A. Gers, Jürgen Schmidhuber and Fred Cummins, "Learning to Forget: Continual Prediction with LSTM," *Neural Computation* 12(10), 2000. DOI
13. Kyunghyun Cho et al., "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation," EMNLP, 2014. arXiv
14. Mike Schuster and Kuldip K. Paliwal, "Bidirectional Recurrent Neural Networks," *IEEE Transactions on Signal Processing* 45(11), 1997. DOI
15. Alex Graves, "Generating Sequences With Recurrent Neural Networks," arXiv:1308.0850, 2013. arXiv
16. Wojciech Zaremba, Ilya Sutskever and Oriol Vinyals, "Recurrent Neural Network Regularization," arXiv:1409.2329, 2014. arXiv
17. Gábor Melis, Chris Dyer and Phil Blunsom, "On the State of the Art of Evaluation in Neural Language Models," arXiv:1707.05589, 2017. arXiv
18. Stephen Merity, Nitish Shirish Keskar and Richard Socher, "Regularizing and Optimizing LSTM Language Models," arXiv:1708.02182, 2017. arXiv
19. Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer and Yonghui Wu, "Exploring the Limits of Language Modeling," arXiv:1602.02410, 2016. arXiv
20. Matteo Tiezzi, Michele Casoni, Alessandro Betti, Tommaso Guidi, Marco Gori and Stefano Melacci, "On the Resurgence of Recurrent Models for Long Sequences: Survey and Research Opportunities in the Transformer Era," arXiv:2402.08132, 2024. arXiv
21. Bo Peng et al., "RWKV: Reinventing RNNs for the Transformer Era," Findings of EMNLP, 2023. Paper
22. Albert Gu and Tri Dao, "Mamba: Linear-Time Sequence Modeling with Selective State Spaces,"

COLM, 2024. arXiv
23. Maximilian Beck et al., “xLSTM: Extended Long Short-Term Memory,” NeurIPS, 2024. Paper
24. Lewis Carroll, *Alice’s Adventures in Wonderland*, 1865; Project Gutenberg eBook #11. Project Gutenberg
25. Ilya Sutskever, Oriol Vinyals and Quoc V. Le, “Sequence to Sequence Learning with Neural Networks,” NeurIPS, 2014. Paper

NLP 06 • Sequence Labeling with RNNs (POS Tagging, Information Extraction)

Label every token of a sentence: part-of-speech tags with HMMs and Viterbi decoding, entity spans with BIO labels and exact span evaluation, and neural taggers built from BiLSTMs, character features and a CRF output layer.

Online edition: https://blog.shyandram.dev/en/learn/nlp/sequence-labeling/

What you'll learn

- Represent part-of-speech tagging and entity recognition as sequence labeling, and explain why independent token decisions can disagree with a valid sentence-level analysis.
- Derive hidden Markov sequence scores and maximum-path (Viterbi) decoding, and connect them to the conditional random field.
- Define an entity schema with reproducible offsets, encode mentions with BIO labels, and evaluate exact spans rather than token accuracy.
- Build BiLSTM taggers, add character-level features and a CRF output layer, and train the CRF by its log-likelihood with a masked forward algorithm.
- Place entity tagging inside information extraction (entities, relations, events) and name the standard datasets.
- Run a complete PyTorch experiment that measures span F1, error types and invalid tag transitions on names never seen in training.

The big picture

Many language tasks ask for one label per token. A part-of-speech (POS) tagger marks each word as noun, verb, adjective and so on. An entity recognizer marks which tokens belong to a person, organization or place name, and where each name begins and ends. Both look like classification repeated along a sentence, but they are not independent decisions: the right label for a word depends on its neighbors' words *and* on its neighbors' labels. Treating the whole label sequence as one structured output is the central idea of this chapter.

McRoy's Chapters 3 and 4 introduce grammatical categories and syntactic processing, and Chapter 8 connects language processing with information extraction [1]. Hagiwara's Chapter 5 develops sequential labeling, including part-of-speech tagging and named entity recognition [2]. This chapter combines those views. A tagger attaches categories to positions; an extractor turns selected spans into records that an application can inspect. The output contract matters: a customer-support system might need a product name and a complaint event, whereas a historical archive needs people, organizations, dates and attributed statements.

Explain like I'm five

Give a child a sentence and three highlighter pens: yellow for people, green for companies, blue for places. Coloring each word alone is not enough. "Port" is blue in "Port Lena" but not in "a port in the harbor," and if the second half of a two-word name is colored green, the first half had better be green too. A good highlighter looks at the whole sentence and makes the colors agree.

The practical risk is losing distinctions during conversion. "The supplier said the device passed" does not establish that the device passed. A system can identify the correct names and still build a false record by dropping attribution or modality. We therefore evaluate both surface extraction and interpretation, rather than treating well-formed output as evidence of correctness. The structured-prediction mathematics (HMMs, the forward–backward algorithm, Viterbi decoding and CRFs in general form) is developed in PR 06; the recurrent encoders come from NLP 05. Parsing, semantic roles and relation and event extraction in depth follow in NLP 12.

Words, morphology and grammatical categories

Plain version. A word's category describes how it behaves in a sentence. Its spelling can suggest a category, but context often decides which use is intended.

Morphology concerns structure within words. A plural ending, tense marker or derivational prefix can supply useful evidence, but not every language expresses these properties with the same pieces or word boundaries. Subword tokens learned for efficient modeling also need not correspond to linguistic morphemes. Keep the model's tokenization separate from the grammatical units used in annotation.

Part of speech describes a grammatical category, such as noun or verb. A noun commonly heads a noun phrase; a verb commonly heads a clause's predicate. These are distributional descriptions rather than rules that every noun must name a physical object. "Decision" and "arrival" are nouns although they refer to abstract content and events.

The same surface word can have several categories. In the invented pair "the pilots can land" and "the metal can fell," the word "can" has different roles. The second sentence also demonstrates why a familiar local phrase can be misleading when the remainder of the sentence changes. A tagger must consider the permitted context and its annotation scheme.

Represent a tagged sentence as:

$$\mathbf{x} = (x_1, \dots, x_T), \qquad \mathbf{y} = (y_1, \dots, y_T), \qquad y_t \in \mathcal{Y}.$$

Here $\mathbf{x}$ is the input word sequence, x_t is the word at position t, T is sequence length, $\mathbf{y}$ is the tag sequence, y_t is the tag for word x_t and $\mathcal{Y}$ is the finite tag inventory. The symbols t and T refer to linguistic word positions, not automatically to model subword positions.

Annotation needs additional rules: contractions, punctuation, multiword expressions and words with several plausible analyses. When a model uses subwords, define how its outputs return to words. Using only the first subword's representation is one possible convention; pooling all subwords is another. Whatever the choice, gold labels and decoded output must use the same units.

Local classification versus sequence prediction

Plain version. Choosing the best tag for every word separately can miss a better joint interpretation. A sequence model makes the relationship among tags explicit.

A local classifier may predict:

$$\hat{y}_t = \operatorname*{argmax}_{k \in \mathcal{Y}} p_\theta(y_t = k \mid \mathbf{x}).$$

Here $\hat{y}_t$ is the selected tag at position t, k ranges over tag inventory $\mathcal{Y}$, p_θ is a model with parameters θ and $\mathbf{x}$ is the whole visible input sequence. A local output head can still read contextual input: "independent" describes the output decision rule, not necessarily the encoder.

A structured decoder instead chooses:

$$\hat{\mathbf{y}} = \operatorname*{argmax}_{\mathbf{y} \in \mathcal{A}(\mathbf{x})} S_\theta(\mathbf{x}, \mathbf{y}).$$

Here $\hat{\mathbf{y}}$ is the selected complete tag sequence, $\mathcal{A}(\mathbf{x})$ is the set of permitted analyses for input $\mathbf{x}$, and S_θ is a score assigned by parameters θ to an input–analysis pair. The permitted set can enforce an annotation constraint, while the score can reward plausible neighboring labels.

This distinction appears in many NLP tasks. Entity labels must form valid spans; a dependency tree needs one parent per non-root word; a generated structured record must satisfy a schema. A stronger encoder does not automatically guarantee a valid output. The decoder or a separate validator must enforce constraints that the application requires.

Structured decoding also has a tradeoff. An incorrect hard constraint can exclude a legitimate construction. If the grammar says every sentence must contain an overt subject, it will reject many commands and fragments. Document the constraint and its intended input domain rather than treating constrained output as evidence that the constraint was linguistically universal.

Hidden Markov models for tags

Plain version. Imagine that tags form a hidden sequence and each tag generates a word. Neighboring tags and word compatibility then jointly determine the score.

A first-order hidden Markov model (HMM), reviewed by Rabiner [3], uses:

$$p(\mathbf{x}, \mathbf{y}) = \pi_{y_1} b_{y_1}(x_1) \prod_{t=2}^{T} a_{y_{t-1}, y_t} b_{y_t}(x_t).$$

Here $p(\mathbf{x}, \mathbf{y})$ is joint probability of words and tags, π_k is the probability of initial tag k, $a_{j,k}$ is the transition probability from tag j to tag k, $b_k(w)$ is the probability that tag k emits word w, x_t and y_t are the word and tag at position t, and T is sequence length. This example conditions on a fixed length; an explicit end-state factor can be added when modeling termination.

Initial probabilities sum to one. Each transition row sums to one, and each emission distribution sums to one over the observation vocabulary. These normalization conditions make the expression a probability model rather than an arbitrary product of useful scores.

The assumptions are restrictive: the next tag depends only on the previous tag, and an emitted word depends only on the current tag. A real sentence has long dependencies, lexical preferences and discourse context. The value of this model here is that the assumptions and exact decoder are visible. It provides a small laboratory for distinguishing model capacity from search.

With fully tagged training examples, count initial states, transitions and emissions, then normalize with a stated smoothing rule. With words alone, hidden-state estimation is a different problem; a state learned without supervision is not automatically a named grammatical category. The example below uses manually specified probabilities so that training uncertainty does not obscure decoding.

Maximum-path decoding

Plain version. Keep the best partial path ending in each tag. Extending those paths is enough to find the best complete sequence without listing every possibility.

Use log probabilities to turn products into sums:

$$D_1(k) = \log \pi_k + \log b_k(x_1),$$
$$D_t(k) = \log b_k(x_t) + \max_{j \in \mathcal{Y}} [D_{t-1}(j) + \log a_{j,k}].$$

Here $D_t(k)$ is the log score of the best path covering positions 1 through t and ending with tag k, π_k is the initial probability, $b_k(x_t)$ is the emission probability, $a_{j,k}$ is the transition probability and $\mathcal{Y}$ is the tag inventory. The indices j and k identify the previous and current tags. A zero probability contributes $-\infty$.

The recurrence is maximum-path dynamic programming, commonly called Viterbi decoding [3]. For each cell, store the previous tag attaining the maximum. After selecting the best final tag, follow these backpointers in reverse. Keeping only the score loses the actual sequence.

The decoder considers T positions and all pairs of adjacent tags, so its time complexity is $O(TK^2)$ for $K = |\mathcal{Y}|$ tags. The score table uses $O(TK)$ space if all backpointers are retained. By contrast, enumerating tag sequences considers K^T possibilities. This is a computational saving under a specific factorization, not an approximate search result.

The following standalone example verifies the dynamic-programming answer against exhaustive enumeration. The toy observation “fish fish” has an intentionally ambiguous word. Independent emission preferences select nouns at both positions; the joint model can favor a noun followed by a verb.

```
import itertools
import numpy as np
```

```
tags = ["N", "V"]
vocabulary = ["fish", "swim"]
initial = np.array([0.8, 0.2])
transition = np.array([[0.1, 0.9], [0.8, 0.2]])
emission = np.array([[0.7, 0.3], [0.3, 0.7]])
words = ["fish", "fish"]
observation = [vocabulary.index(word) for word in words]
assert np.allclose(transition.sum(axis=1), 1)
assert np.allclose(emission.sum(axis=1), 1)

scores = np.empty((len(words), len(tags)))
back = np.zeros_like(scores, dtype=int)
scores[0] = np.log(initial) + np.log(emission[:, observation[0]])
for t in range(1, len(words)):
    candidates = scores[t - 1, :, None] + np.log(transition)
    back[t] = candidates.argmax(axis=0)
    scores[t] = candidates.max(axis=0) + np.log(emission[:, observation[t]])
path = [int(scores[-1].argmax())]
for t in range(len(words) - 1, 0, -1):
    path.append(int(back[t, path[-1]]))
path.reverse()

def joint_score(sequence):
    value = np.log(initial[sequence[0]])
    value += np.log(emission[sequence[0], observation[0]])
    for t in range(1, len(words)):
        value += np.log(transition[sequence[t - 1], sequence[t]])
        value += np.log(emission[sequence[t], observation[t]])
    return value

all_paths = list(itertools.product(range(len(tags)), repeat=len(words)))
brute = max(all_paths, key=joint_score)
assert tuple(path) == brute
print("local emission choices:", [tags[emission[:, w].argmax()] for w in observation])
print("best joint path:", [tags[index] for index in path])
print("joint log score:", round(joint_score(path), 3))
```

Changing transition probabilities changes the chosen sequence even when every word-level preference stays fixed. This does not prove the new choice is correct in real language. It proves that the implemented model and decoder express the intended dependence.

Read the path score as evidence, not just an argmax

The noun–verb path wins in the preceding example because the transition preference outweighs the second word's emission preference. Its joint probability is the product of four factors: an initial noun, a noun emitting `fish`, a noun-to-verb transition, and a verb emitting `fish`. Comparing it with noun–noun cancels the shared first two factors. The remaining comparison is between transition times emission; the decision is therefore intelligible without treating the decoder as a black box.

The following independent calculation enumerates the same four paths and normalizes them into

a posterior distribution for the observed words:

```
import itertools
import numpy as np

initial = np.array([0.8, 0.2])
transition = np.array([[0.1, 0.9], [0.8, 0.2]])
fish = np.array([0.7, 0.3])
paths = list(itertools.product(range(2), repeat=2))
mass = np.array([initial[a] * fish[a] * transition[a, b] * fish[b]
                 for a, b in paths])
posterior = mass / mass.sum()
print('NN / NV / VN / VV joint:', mass.round(4))
print('Posterior:', posterior.round(3))
print('Probability of observed words:', round(mass.sum(), 4))
assert paths[posterior.argmax()] == (0, 1)
assert np.isclose(posterior.sum(), 1)
```

The joint masses are `0.0392`, `0.1512`, `0.0336` and `0.0036`, and the posterior rounds to `0.172`, `0.664`, `0.148` and `0.016`. The observation probability is `0.2276`. These answer different questions. A joint mass includes how likely the words were; a posterior compares label interpretations after those words are fixed. The winning path still leaves probability on alternatives under this constructed model.

For larger sequences, summing predecessor contributions rather than maximizing them computes observation probability. Combining forward and backward contributions gives tag marginals [3]. Selecting the most probable tag at each position minimizes expected per-position mistakes under that posterior; selecting the most probable complete path targets whole-sequence correctness. Those decisions can differ. Independently selected marginal tags can also violate output constraints when the allowed sequence set contains restrictions.

Why can maximum-path decoding discard other prefixes ending in the same tag? Under the first-order factorization, their remaining extensions depend on that final tag and future observations, not on earlier prefix details. A lower-scoring prefix cannot overtake the better prefix after an identical suffix is added. If a new score rewards a relation to a tag several positions earlier, this argument no longer applies to the same state. Either enlarge the state or change the decoder. Exactness belongs to the score factorization, not to the name of the algorithm.

Conditional structured scores connect neural features to decoding

An HMM must model how words are generated. A conditional model can instead score label sequences directly from the observed sentence. A linear-chain conditional random field (CRF) provides this interface [5]:

$$S_\theta(\mathbf{x}, \mathbf{y}) = \sum_{t=1}^{T} u_\theta(t, y_t, \mathbf{x}) + \sum_{t=2}^{T} A_{y_{t-1}, y_t}, \qquad p_\theta(\mathbf{y} \mid \mathbf{x}) = \frac{\exp S_\theta(\mathbf{x}, \mathbf{y})}{Z_\theta(\mathbf{x})}.$$

Here $\mathbf{x}$ is the observed sentence, $\mathbf{y}$ is a tag sequence, T is its length, t indexes positions, u_θ is a position–tag score computed with parameters θ, and A contains adjacent-tag scores. The normalizer $Z_\theta(\mathbf{x})$ sums $\exp S_\theta(\mathbf{x}, \mathbf{y}')$ over every permitted sequence $\mathbf{y}'$. Unlike HMM emissions, the position scores need not sum to one over words. A contextual encoder can compute each score using the entire sentence while the output dependence remains a chain.

Training an annotated sequence minimizes its negative log probability: the correct score is rewarded relative to the scores of all permitted competitors. Computing the normalizer uses a sum recurrence; finding the best sequence uses a max recurrence. This explains why a neural encoder and structured decoder fit together. Rich input evidence does not require discarding a tractable output factorization. Conversely, a chain transition table cannot express every syntactic dependency; sentence-level input features and output constraints solve different parts of the problem.

PR 06 develops the general theory behind these two models: the forward and forward–backward algorithms, Baum–Welch training of HMMs from unlabeled sequences, and the CRF gradient as observed minus expected feature counts. The neural tagger sections below reuse exactly this CRF, with the position scores u_θ computed by a recurrent encoder.

Entity recognition starts with a schema

Plain version. An entity recognizer locates mentions and assigns types. It does not automatically identify the unique real-world object behind each mention.

Represent an entity mention as:

$$r = (d, b, e, c).$$

Here r is a mention record; d is the document identifier; b is its inclusive starting character offset; e is its exclusive ending offset; and c is its category. The surface text is the original document slice from b to e. These are Python string offsets in our example; another interface may use bytes or another indexing convention. Declare the convention because Unicode characters and encoded bytes do not have identical boundaries.

An annotation guideline must decide whether titles belong to a person's span, whether an organization suffix is included, and how to handle ambiguous uses. "Taipei" can name a place or stand for a government in context. A location-only schema and a context-sensitive organization schema can legitimately produce different annotations. Evaluation becomes meaningful only after the schema and guidelines are frozen.

Named entity recognition, or NER, commonly distinguishes people, organizations and locations; the CoNLL-2003 shared-task paper describes one established benchmark setting [6]. We mention that resource to identify the evaluation tradition, not to imply that its categories fit every application. Product identifiers, legal provisions or equipment parts may be more useful in another domain.

Entity linking is a separate decision: map a mention to a record in a knowledge base, or choose that no suitable record exists. Two mentions with the same spelling can refer to different organizations; two different spellings can refer to the same one. A recognizer that marks "Orion"

as an organization has not solved this identity problem. Linking requires a candidate inventory, context and an explicit unknown-entity policy.

Keep the original text and offsets even when you store a normalized name. Lowercasing, accent removal and whitespace collapse can change strings and boundaries. One reliable design extracts from the original text and stores normalization as an additional field. If preprocessing is necessary, retain an alignment map so the predicted span can be projected back without guessing.

Sequence labels and span prediction

Plain version. Entity mentions can be expressed as labels on tokens or as explicit spans. The representation determines which overlaps the model can describe.

In a BIO scheme, a token label begins with B for the beginning of a mention, I for its continuation or O for outside. An example is "Mina works at Orion Labs," with person labels on Mina and organization labels on Orion and Labs. Hagiwara's sequence-labeling chapter and the NER survey give context for token-level extraction models [2], [7]. The scheme descends from the inside–outside chunk tags of Ramshaw and Marcus [10].

For an independent token classifier:

$$p_\theta(\mathbf{y} \mid \mathbf{x}) = \prod_{i=1}^{T} p_\theta(y_i \mid \mathbf{x}).$$

Here $\mathbf{x}$ is the input token sequence; $\mathbf{y}$ is the label sequence; T is the number of tokens; i indexes positions; and θ denotes learned parameters. Each prediction may use a contextual representation of the whole permitted input, although the output labels are independent under this factorization. Input context and output dependence are different ideas.

Independent decisions can produce invalid transitions, such as an I-PER label after O. A deterministic repair rule is possible, but its behavior becomes part of the evaluation contract. Alternatively, a structured decoder searches only valid paths while scoring the whole sequence. The dynamic-programming idea from the maximum-path decoding section applies when the score decomposes locally. More powerful encoders do not remove the need to define valid outputs.

A flat token-label sequence cannot represent arbitrary nested mentions. "Orion Labs Research Division" could contain an organization mention and a nested division mention under a particular schema. A span model instead scores candidate boundaries and types:

$$s_\theta(b, e, c \mid \mathbf{x}).$$

Here s_θ is a learned score; b and e are candidate span boundaries; c is the candidate category; and $\mathbf{x}$ is the input. The score need not be a calibrated probability. A selection policy decides which candidates to keep, whether overlap is allowed and how conflicts are resolved.

The candidate count grows with sentence length, so practical designs may bound maximum span length or prune candidates. Every such restriction changes which mentions can be recovered. If very long names matter, measure their recall separately. A fast system can otherwise achieve a

reasonable average score while never representing the outputs needed by a small but consequential group of users.

Boundary modeling changes which errors are possible

A token-labeling model and a span model can use the same contextual encoder yet search different output spaces. Flat BIO labels choose a single membership at each token. Explicit spans choose boundary pairs and types, so nested intervals can coexist when the selection policy permits them. The difference is representational: an encoder cannot recover a nested output that its decoder has no way to express [7].

For n tokens, the number of nonempty contiguous intervals is $n(n+1)/2$. Here n is sentence length; the count sums the possible endings for each starting position. With maximum length $L \le n$, the count becomes $nL - L(L-1)/2$, where L is the permitted span length. A type scorer may need a score for each of those intervals and each category. Bounding length saves work but removes longer mentions from the candidate space before classification begins.

Boundary quality and semantic typing need separate diagnosis. A span covering `Orion` may be a strong organization candidate while the gold mention is `Orion Labs`. Raising its category confidence does not fix the missing boundary. A model may instead propose both spans; then deduplication and overlap rules determine the final output. Treat selection as part of the model behavior, not as formatting applied after evaluation.

Relations add another search problem. Candidate entity pairs multiply as more mentions survive. Restricting pairs by type or distance can reduce computation, but a document-level relation may cross several sentences. A joint extractor can use relation evidence to influence mention decisions, whereas a pipeline commits to mentions first. Compare the pipeline on predicted mentions and on gold mentions: the gap reveals how much relation recovery is lost before relation scoring starts.

Exact span evaluation

Plain version. Correctly labeling common background tokens can hide missed entities. Evaluate complete mentions, including their boundaries and types.

For sets of predicted records P and gold records G:

$$\text{precision} = \frac{|P \cap G|}{|P|}, \qquad \text{recall} = \frac{|P \cap G|}{|G|}, \qquad F_1 = \frac{2|P \cap G|}{|P| + |G|}.$$

Here P and G contain document identifier, start offset, end offset and category tuples; $|\cdot|$ denotes set size; and $P \cap G$ contains exact matches. Precision measures the share of predictions that match; recall measures the share of annotations recovered; and F_1 combines them. Our code reports zero when a displayed denominator is zero; another evaluation convention should be documented explicitly.

Exact matching is strict. Predicting “Orion” when the annotation says “Orion Labs” creates both a false positive and a false negative. That is appropriate when downstream code depends on

a complete name. For another application, partial overlap may deserve a secondary diagnostic. Report its matching rule separately; do not rename a forgiving overlap score as the established exact-match measure [6].

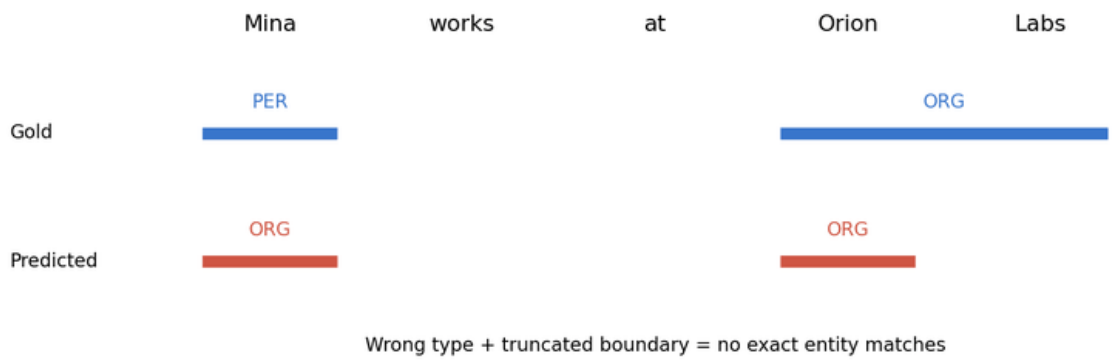


Figure 1 — A boundary error and a type error remain errors even when most words are labeled correctly. This is a synthetic diagnostic example.

A single aggregate score still hides the mechanism. Break errors into missed mention, spurious mention, boundary mismatch, category mismatch and linking mismatch. Also report performance by category and document source. When a category is rare, retain its example count so the reader can assess uncertainty. A large weighted average can otherwise bury a category with nearly no successful predictions.

A complete CPU extraction experiment

Plain version. A small deterministic extractor makes the contract easy to inspect. It is a teaching baseline with visible blind spots, not a general language-understanding system.

The following standalone example recognizes an invented name inventory, extracts one tightly constrained active repair construction and evaluates exact entity records. It downloads nothing. The inventory is fixed before evaluation, and the gold records are written explicitly. A lexicon is an allowed input here; if a real evaluation forbids external name inventories, the same code would violate that contract.

```
import re

documents = {
    "a": "Mina repaired Sensor A in Taipei.",
    "b": "Orion Labs said Sensor B might fail.",
    "c": "Leo did not repair Sensor B.",
    "d": "The sensor was repaired by Mina.",
}
inventory = {
```

```
    "Mina": "PER", "Leo": "PER", "Orion Labs": "ORG",
    "Sensor A": "DEVICE", "Sensor B": "DEVICE", "Taipei": "LOC",
}
gold = {
    ("a", 0, 4, "PER"), ("a", 14, 22, "DEVICE"), ("a", 26, 32, "LOC"),
    ("b", 0, 10, "ORG"), ("b", 16, 24, "DEVICE"),
    ("c", 0, 3, "PER"), ("c", 19, 27, "DEVICE"), ("d", 27, 31, "PER"),
}
predicted = set()
for doc_id, text in documents.items():
    for name, category in inventory.items():
        pattern = r"(?<!\w)" + re.escape(name) + r"(?!\w)"
        for match in re.finditer(pattern, text):
            predicted.add((doc_id, match.start(), match.end(), category))
tp = len(predicted & gold)
precision = tp / len(predicted) if predicted else 0.0
recall = tp / len(gold) if gold else 0.0
f1 = 2 * tp / (len(predicted) + len(gold)) if predicted or gold else 0.0
assert predicted == gold
print("entity precision/recall/F1:", precision, recall, f1)

repair = re.compile(r"^(Mina|Leo) repaired (Sensor [AB])(?=[ .])")
events = []
for doc_id, text in documents.items():
    match = repair.search(text)
    if match:
        events.append({
            "document": doc_id, "predicate": "repair",
            "actor": match.group(1), "object": match.group(2),
            "polarity": "positive", "modality": "asserted",
            "evidence": [match.start(), match.end()],
        })
assert len(events) == 1 and events[0]["document"] == "a"
for event in events:
    start, end = event["evidence"]
    print(event, "source:", documents[event["document"]][start:end])
print("unsupported repair constructions:", ["c", "d"])
```

The exact entity score is 1.0 on these hand-built documents, because all mention spellings are in the inventory. That result does not establish generalization. The event rule returns only document a. It excludes the explicit negation in c but also misses the valid passive construction in d. These are different reasons for returning no event, and a production interface should distinguish rejected negative claims from unsupported syntax.

Document b contains recognized entities but no asserted repair event. It also contains an attributed, possible failure. This example illustrates why entity recognition and event extraction need separate evaluation. A system can obtain every name correctly while omitting the task's most relevant event. Expanding the event grammar requires additional positive and negative fixtures before accepting the new behavior.

The evidence interval covers only the matched clause, not the trailing location. That is adequate for the narrow repair record, which has no location field. If the record later includes location,

expand the evidence contract and its tests. Treat schema changes as changes to the task, rather than adding a field that is populated opportunistically from unrelated nearby words.

Information extraction beyond entity spans

Plain version. Entity recognition finds the names. Information extraction (IE) goes on to connect them: which person works for which organization, which device failed where and when. Each step adds a new kind of record and a new way to be wrong.

A useful map has three layers, each building on the previous one.

Layer	Output record	Example from the running domain	Typical method family
Entities	Typed span (d, b, e, c)	`Orion Labs` is an ORG at characters 0–10	Sequence labeling (this chapter) or span scoring
Relations	Typed, directed pair of mentions with evidence	`employer(Mina, Orion Labs)`	Classify candidate mention pairs
Events	Trigger plus role-labeled arguments, polarity and modality	`repair` with repairer `Mina`, object `Sensor A`, place `Taipei`	Trigger detection, then argument labeling

Sequence labeling contributes directly to all three. Entity mentions are BIO spans. Event triggers can be tagged the same way as entities. Even some relation and argument decisions are posed as labeling, by tagging the tokens of each argument relative to a given trigger. The rule-based extractor above already shows the dependency between layers: its single event record is correct only because both entity spans were found and the construction was supported.

Standard resources reflect the layers. The ACE 2005 multilingual training corpus annotates entities, relations and events for English, Chinese and Arabic, with event annotation limited to English and Chinese [19]. TACRED is a crowdsourced sentence-level relation extraction dataset targeted at the relation types of the TAC Knowledge Base Population evaluation [20]. DocRED moves relation extraction to whole documents built from Wikipedia and Wikidata, where many relations can only be inferred by reading several sentences [21]. The record design questions raised earlier in this chapter (evidence spans, attribution, negation, unknown values) apply at every layer. Relation and event extraction, semantic roles and their evaluation are developed in NLP 12.

Neural taggers: a BiLSTM over the sentence

Plain version. Replace hand-written emission probabilities with a neural network that reads the whole sentence in both directions and produces a score for every tag at every position. The simplest version picks the highest-scoring tag at each position independently.

Collobert and colleagues showed that a single neural architecture, trained from data with little feature engineering, could tag parts of speech, chunks, named entities and semantic roles [12].

With recurrent encoders the standard design is the bidirectional LSTM (BiLSTM) tagger [13], [14], [15]:

$$\overrightarrow{\mathbf{h}}_t = \mathrm{LSTM}_{\rightarrow}(\mathbf{v}_t, \overrightarrow{\mathbf{h}}_{t-1}), \qquad \overleftarrow{\mathbf{h}}_t = \mathrm{LSTM}_{\leftarrow}(\mathbf{v}_t, \overleftarrow{\mathbf{h}}_{t+1}),$$
$$\mathbf{u}_t = W\,[\overrightarrow{\mathbf{h}}_t; \overleftarrow{\mathbf{h}}_t] + \mathbf{b} \in \mathbb{R}^K.$$

Here $\mathbf{v}_t$ is the input vector for token x_t (a word embedding, optionally concatenated with character features below); $\overrightarrow{\mathbf{h}}_t$ has read $x_1, \dots, x_t$ and $\overleftarrow{\mathbf{h}}_t$ has read $x_t, \dots, x_T$; $[\cdot;\cdot]$ is concatenation; W and $\mathbf{b}$ map the two states to $K = |\mathcal{Y}|$ tag scores; and $\mathbf{u}_t$ collects those scores, $u_{t,k}$ for tag k. These scores play the role of $u_\theta(t, y_t, \mathbf{x})$ in the CRF formula above: they can depend on the whole sentence. Bidirectionality is appropriate here, unlike in the left-to-right language model of NLP 05, because the whole sentence is given before any tag is chosen [23].

The **softmax tagger** treats the tags as conditionally independent given these scores:

$$p_\theta(y_t = k \mid \mathbf{x}) = \frac{\exp u_{t,k}}{\sum_{k'=1}^{K} \exp u_{t,k'}}, \qquad \mathcal{L}_{\text{tok}} = -\frac{1}{N} \sum_b \sum_t m_{b,t} \log p_\theta(y_{b,t} \mid \mathbf{x}_b).$$

Here b indexes sentences in a mini-batch, $m_{b,t} \in \{0, 1\}$ is the padding mask (one for a real token), $N = \sum_{b,t} m_{b,t}$ counts real tokens, $y_{b,t}$ is the gold tag and k' ranges over tags. This is exactly the independent token classifier of the sequence-labeling section, with a BiLSTM computing the context. It is fast and often strong. Its weakness is also the one identified there: nothing stops it from emitting `O I-ORG`, or `B-PER I-ORG`, because each position's decision ignores its neighbors' decisions.

Three practical details matter as much as the architecture.

- **Masks.** Sentences in a batch are padded to a common length. The mask removes padding from the loss; packing (`pack_padded_sequence`) or masking keeps padding out of the backward LSTM, which would otherwise start reading from padding symbols.
- **Unknown words.** A word-embedding table only covers training words. Map the rest to an unknown id, and *train that id*: randomly replacing a fraction of training words with it (word dropout) teaches the tagger to rely on context when the word itself is unfamiliar.
- **Subword encoders.** When the encoder uses subwords, define how word labels map to pieces (label the first piece and ignore the others, or another explicit rule) and map predictions back to words before evaluation, as discussed in the first section.

BiLSTM-CRF: scoring the whole label sequence

Plain version. Keep the BiLSTM's per-token scores, add a learned score for every pair of adjacent tags, and choose the label sequence with the best total. Training raises the score of the gold sequence relative to all other sequences, which the forward algorithm sums efficiently.

Huang, Xu and Yu proposed putting a linear-chain CRF layer on top of a BiLSTM for POS tagging, chunking and NER [13]. Lample and colleagues [14] and Ma and Hovy [15] combined the same output layer with character-level word representations and reported strong NER results without

hand-engineered features. Using the CRF notation from the maximum-path decoding section, with start and end scores added:

$$S_\theta(\mathbf{x}, \mathbf{y}) = \pi_{y_1} + \sum_{t=1}^{T} u_{t,y_t} + \sum_{t=2}^{T} A_{y_{t-1},y_t} + \eta_{y_T}, \qquad \log p_\theta(\mathbf{y} \mid \mathbf{x}) = S_\theta(\mathbf{x}, \mathbf{y}) - \log Z_\theta(\mathbf{x}).$$

Here $u_{t,k}$ are the BiLSTM scores; $A_{j,k}$ is the learned score for tag j followed by tag k; π_k and η_k score starting and ending with tag k; and $Z_\theta(\mathbf{x}) = \sum_{\mathbf{y}'} \exp S_\theta(\mathbf{x}, \mathbf{y}')$ sums over every tag sequence $\mathbf{y}'$ of length T. The training loss is the negative log-likelihood $-\log p_\theta(\mathbf{y} \mid \mathbf{x})$ averaged over sentences.

The sum over K^T sequences is computed by the forward algorithm in log space:

$$\begin{aligned} \alpha_1(k) &= \pi_k + u_{1,k}, \\ \alpha_t(k) &= u_{t,k} + \log \sum_{j=1}^{K} \exp\left(\alpha_{t-1}(j) + A_{j,k}\right), \qquad t = 2, \dots, T, \\ \log Z_\theta(\mathbf{x}) &= \log \sum_{k=1}^{K} \exp\left(\alpha_T(k) + \eta_k\right). \end{aligned}$$

Here $\alpha_t(k)$ is the log of the summed exponentiated scores of all partial sequences that cover positions 1 to t and end in tag k. Replacing each log-sum-exp by a maximum, and storing the maximizing j, gives exactly the Viterbi recursion of the HMM section with $\log b_k(x_t)$ replaced by $u_{t,k}$ and $\log a_{j,k}$ by $A_{j,k}$. Both cost $O(TK^2)$. In a padded batch, a position with mask zero copies α_{t-1} forward unchanged, so each sentence's normalizer stops at its own length.

What the gradient does. Differentiating the loss with respect to a transition score gives

$$\frac{\partial(-\log p_\theta(\mathbf{y} \mid \mathbf{x}))}{\partial A_{j,k}} = \mathbb{E}_{\mathbf{y}' \sim p_\theta(\cdot \mid \mathbf{x})}[n_{j,k}(\mathbf{y}')] - n_{j,k}(\mathbf{y}),$$

where $n_{j,k}(\mathbf{y})$ counts how often tag j is followed by tag k in $\mathbf{y}$ and the expectation is under the model. Gradient descent therefore raises transitions that the gold sequence uses more often than the model currently expects and lowers the others; the expectations come from forward–backward marginals, as PR 06 shows. Automatic differentiation of the forward algorithm computes this without writing the backward pass by hand.

Hard constraints. A BIO scheme forbids some transitions outright: `I-x` may only follow `B-x` or `I-x`, and no sentence may start with `I-x`. A learned A usually discovers this from data, but with little data it may not. Setting the forbidden entries of A and π to $-\infty$ (in practice a large negative number) makes invalid sequences impossible under both training and decoding, at no cost to valid ones. This is the output-constraint idea from the local-versus-sequence section made concrete.

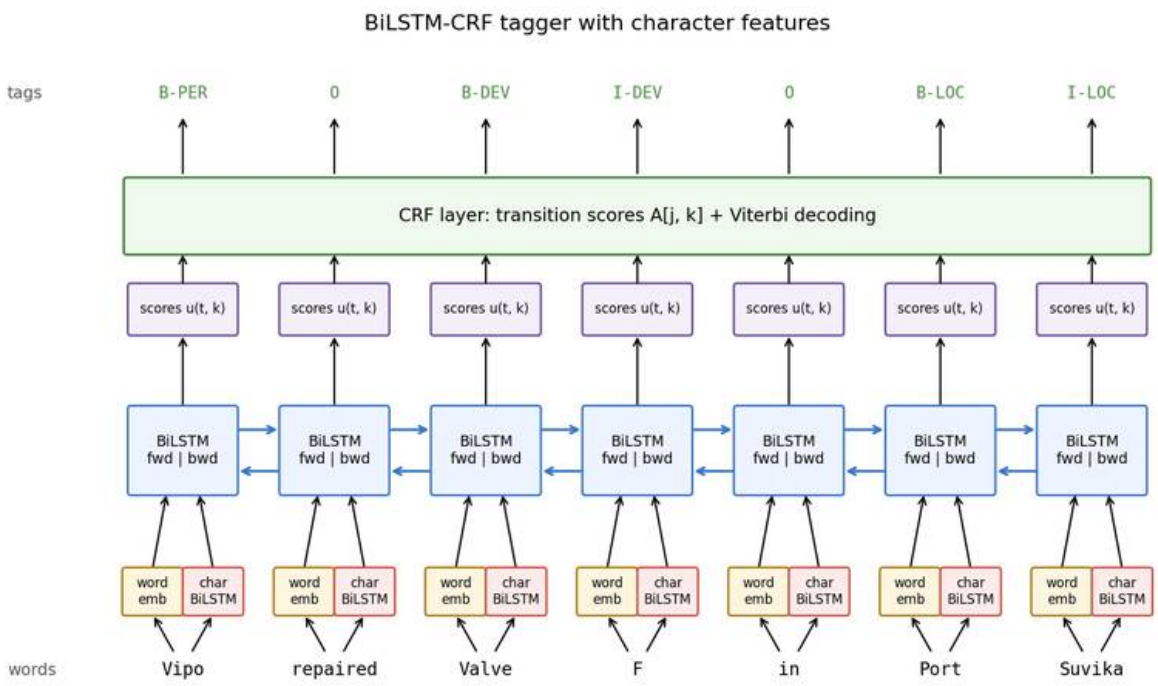


Figure 2 — A BiLSTM-CRF tagger with character features. The emission scores come from the encoder; the transition scores and Viterbi decoding make the tags agree with each other.

Online only: BiLSTM-CRF — Bidirectional LSTM-CRF Models for Sequence Tagging — https://blog.shyandram.dev/en/learn/nlp/sequence-labeling/papers/bilstm-crf–bidirectional-lstm-crf-models-for-sequence-tagging/

Character-level features

Plain version. A word-embedding table knows nothing about a word it never saw. The letters of the word often do: capitalization, suffixes such as "-ville" or "-corp," digits and hyphens are strong clues to the tag even for a brand-new name.

Before neural taggers, these clues were hand-written features (is capitalized, ends in "-ing," contains a digit). Neural taggers learn them from characters. Lample and colleagues run a small bidirectional LSTM over each word's characters and concatenate its final forward and backward states to the word embedding [14]; Ma and Hovy use a convolution over character embeddings followed by max pooling [15]. Either way the input becomes

$$\mathbf{v}_t = [\mathbf{e}(x_t); \mathbf{c}(x_t)], \qquad \mathbf{c}(x_t) = [\overrightarrow{\mathbf{g}}_L; \overleftarrow{\mathbf{g}}_1],$$

where $\mathbf{e}(x_t)$ is the word embedding, $\mathbf{c}(x_t)$ the character summary, L the number of characters in x_t, $\overrightarrow{\mathbf{g}}_L$ the forward character-LSTM state after the last character and $\overleftarrow{\mathbf{g}}_1$ the backward state after reading back to the first. The word-level BiLSTM then reads $\mathbf{v}_t$. Akbik, Blythe and Vollgraf took

this idea further: they trained a character-level language model, as in NLP 05, and used its internal states around each word as contextual embeddings for sequence labeling [16].

Character features help with exactly the inputs that word embeddings handle worst: unseen names, rare inflections and misspellings. They do not resolve ambiguity that the spelling does not encode. A name with no telling suffix can be a person or a company, and only context or world knowledge decides. Character inputs also need their own unknown-symbol policy for characters missing from training.

Datasets for tagging and extraction

Plain version. Benchmarks fix a tag inventory, an annotation guideline, a tokenization and a split. Comparing numbers across them is only meaningful when all four match.

Resource	Task	What it provides
Penn Treebank [11]	English POS tagging and parsing	A large annotated corpus of English with part-of-speech tags and syntactic bracketing; its tag set is widely used for English
Universal Dependencies [4]	Cross-lingual POS, morphology and dependencies	A shared annotation framework applied to treebanks in many languages
CoNLL-2003 [6]	Named entity recognition	English and German data with four types (person, organization, location, miscellaneous) and exact-match evaluation
OntoNotes [17]	Named entities with 18 types, plus syntax, semantic roles and coreference	Multi-genre, multilingual annotation with stratified training, development and test partitions whose test sets match the CoNLL-2012 shared task
WNUT 2017 [18]	Novel and emerging entities in noisy user-generated text	A test of recall on rare and previously unseen surface forms
ACE 2005 [19]	Entities, relations and events	Multilingual annotation (English, Chinese, Arabic) used for relation and event extraction
TACRED [20]	Sentence-level relation extraction	Crowdsourced relation labels for TAC KBP relation types
DocRED [21]	Document-level relation extraction	Entities and relations annotated over Wikipedia documents, requiring multi-sentence reasoning

Tag encodings differ too. The CoNLL-2003 files use an IOB variant, descended from Ramshaw and Marcus's chunk tags, in which `B-` appears only when an entity directly follows another entity of the same type [6], [10]; convert such files to the BIO form used in this chapter before training, or a BIO span extractor will misread them. Licenses differ: some of these resources are freely downloadable, while others are distributed by the Linguistic Data Consortium under license. Check the terms before building on one. For a new domain, the most valuable dataset is

usually a small, carefully adjudicated set annotated with your own frozen guideline, used alongside a public benchmark to compare methods.

A complete PyTorch tagging experiment

Plain version. We train three taggers on the same small synthetic corpus and evaluate them on sentences whose names never appear in training: a word-only BiLSTM, the same tagger with character features, and the character tagger with a constrained CRF output layer. The evaluation counts exact spans, error types and invalid tag transitions.

The corpus is invented. Person, organization and location names are generated from random syllables, so test names are new spellings. Organizations either end in `tek` or `corp` or are followed by the word `Labs`; locations either end in `ville` or follow `Port`. Several templates allow either a person or an organization in the same slot, so context alone cannot always decide the type. There are 100 training sentences and 200 test sentences. Everything else is implemented from scratch: span extraction from BIO tags, exact span precision, recall and F1, an error breakdown, padding masks, word dropout, the character BiLSTM, and a CRF with a masked forward algorithm, Viterbi decoding and BIO constraints, checked against brute-force enumeration. Each model is trained with three seeds; the script runs on a CPU in well under a minute.

```
import random
import torch
from torch import nn

# ---- 1. A synthetic tagged corpus: invented names, fixed templates ----
SYL = ["ka", "mi", "to", "re", "na", "lo", "su", "vi", "da", "po", "ri", "ze"]

def coin(rng, kind):
    stem = "".join(rng.choice(SYL) for _ in range(rng.choice((2, 3)))).capitalize()
    if kind == "PER":
        return stem
    if kind == "ORG":
        return rng.choice([stem + rng.choice(["tek", "corp"]), stem + " Labs"])
    return rng.choice(["Port " + stem, stem + "ville"])           # LOC

rng = random.Random(0)
TRAIN_NAMES = {k: sorted({coin(rng, k) for _ in range(40)}) for k in ("PER", "ORG", "LOC")}
seen = {n for v in TRAIN_NAMES.values() for n in v}
TEST_NAMES = {k: rng.sample(sorted({coin(rng, k) for _ in range(60)} - seen), 15)
              for k in ("PER", "ORG", "LOC")}                      # unseen spellings
TRAIN_NAMES["DEV"] = ["Pump A", "Pump B", "Sensor C", "Sensor D", "Valve E", "Valve F"]
TEST_NAMES["DEV"] = ["Pump G", "Sensor H", "Valve J", "Pump K"]
TEMPLATES = [
    "{PER} repaired {DEV} in {LOC} .",
    "{ORG} repaired {DEV} in {LOC} .",
    "{PER} reported a fault in {DEV} .",
    "{ORG} reported a fault in {DEV} .",
    "the sensor near {LOC} was checked by {PER} .",
    "the sensor near {LOC} was checked by {ORG} .",
    "{PER} and {PER} visited {LOC} .",
```

```
    "{ORG} shipped {DEV} to {LOC} yesterday .",
    "after the test , {PER} said {DEV} might fail .",
    "a pump in {LOC} failed and {ORG} replaced it .",
]

def make(names, n):
    data = []
    for _ in range(n):
        words, tags = [], []
        for piece in rng.choice(TEMPLATES).split():
            if piece.startswith("{"):
                kind = piece[1:-1]
                toks = rng.choice(names[kind]).split()
                words += toks
                tags += ["B-" + kind] + ["I-" + kind] * (len(toks) - 1)
            else:
                words.append(piece)
                tags.append("O")
        data.append((words, tags))
    return data

train_data, test_data = make(TRAIN_NAMES, 100), make(TEST_NAMES, 200)
TAGS = ["O"] + [f"{p}-{k}" for k in ("PER", "ORG", "LOC", "DEV") for p in "BI"]
t2i = {t: i for i, t in enumerate(TAGS)}
w2i = {w: i + 2 for i, w in enumerate(sorted({w for s, _ in train_data for w in s}))}
c2i = {c: i + 2 for i, c in enumerate(sorted({c for w in w2i for c in w}))}
PAD, UNK = 0, 1
print(train_data[0])

# ---- 2. Spans and exact span F1, from scratch ----
def spans(tags):
    out, start, kind = set(), None, None
    for i, t in enumerate(tags + ["O"]):
        if start is not None and not (t.startswith("I-") and t[2:] == kind):
            out.add((start, i, kind))
            start = None
        if t.startswith("B-") or (t.startswith("I-") and start is None):
            start, kind = i, t[2:]                  # an orphan I- still opens a span
    return out

def span_prf(gold_seqs, pred_seqs):
    tp = n_pred = n_gold = 0
    for g, p in zip(gold_seqs, pred_seqs):
        G, P = spans(g), spans(p)
        tp, n_pred, n_gold = tp + len(G & P), n_pred + len(P), n_gold + len(G)
    prec = tp / n_pred if n_pred else 0.0
    rec = tp / n_gold if n_gold else 0.0
    f1 = 2 * tp / (n_pred + n_gold) if n_pred + n_gold else 0.0
    return prec, rec, f1

def error_types(gold_seqs, pred_seqs):
    counts = {"type": 0, "boundary": 0, "missed": 0, "spurious": 0}
    for g, p in zip(gold_seqs, pred_seqs):
        G, P = spans(g), spans(p)
        overlap = lambda a, b: a[0] < b[1] and b[0] < a[1]
        for s in G - P:
```

```
            hits = [q for q in P if overlap(s, q)]
            if any(q[:2] == s[:2] for q in hits):
                counts["type"] += 1             # right boundaries, wrong category
            elif hits:
                counts["boundary"] += 1         # overlaps, wrong boundaries
            else:
                counts["missed"] += 1
        counts["spurious"] += sum(not any(overlap(q, s) for s in G) for q in P - G)
    return counts

def invalid(tags):                                  # I-x not preceded by B-x or I-x
    return sum(t.startswith("I-") and (i == 0 or tags[i - 1][2:] != t[2:])
               for i, t in enumerate(tags))

assert spans(["B-ORG", "I-ORG", "O", "B-PER"]) == {(0, 2, "ORG"), (3, 4, "PER")}
assert invalid(["O", "I-LOC", "B-DEV", "I-DEV"]) == 1
print("train/test sentences:", len(train_data), len(test_data),
      " gold test spans:", sum(len(spans(t)) for _, t in test_data))

# ---- 3. Padding, masks and word dropout ----
def batch(data, unk_rate=0.0, r=None):
    L = max(len(s) for s, _ in data)
    W = max(len(w) for s, _ in data for w in s)
    x = torch.full((len(data), L), PAD)
    c = torch.full((len(data), L, W), PAD)
    y = torch.zeros(len(data), L, dtype=torch.long)
    for b, (s, t) in enumerate(data):
        for i, w in enumerate(s):
            drop = r is not None and r.random() < unk_rate
            x[b, i] = UNK if drop else w2i.get(w, UNK)      # train the unknown id too
            y[b, i] = t2i[t[i]]
            for j, ch in enumerate(w):
                c[b, i, j] = c2i.get(ch, UNK)
    return x, c, y, x != PAD

# ---- 4. BiLSTM encoder with optional character BiLSTM features ----
class Encoder(nn.Module):
    def __init__(self, use_chars, wdim=32, cdim=16, hidden=48):
        super().__init__()
        self.use_chars = use_chars
        self.wemb = nn.Embedding(len(w2i) + 2, wdim, padding_idx=PAD)
        self.cemb = nn.Embedding(len(c2i) + 2, cdim, padding_idx=PAD)
        self.clstm = nn.LSTM(cdim, cdim, batch_first=True, bidirectional=True)
        self.lstm = nn.LSTM(wdim + (2 * cdim if use_chars else 0), hidden,
                            batch_first=True, bidirectional=True)
        self.emit = nn.Linear(2 * hidden, len(TAGS))

    def forward(self, x, c):
        feats = [self.wemb(x)]
        if self.use_chars:
            B, L, W = c.shape
            lengths = (c != PAD).sum(-1).clamp(min=1).view(-1)
            packed = nn.utils.rnn.pack_padded_sequence(
                self.cemb(c.view(B * L, W)), lengths, batch_first=True,
                enforce_sorted=False)
            _, (h, _) = self.clstm(packed)          # last forward + first backward state
```

```
            feats.append(torch.cat([h[0], h[1]], -1).view(B, L, -1))
        h, _ = self.lstm(torch.cat(feats, -1))
        return self.emit(h)                          # emission scores u[b, t, k]

# ---- 5. Linear-chain CRF with BIO constraints and a padding mask ----
class CRF(nn.Module):
    def __init__(self, K):
        super().__init__()
        self.A = nn.Parameter(torch.zeros(K, K))     # A[j, k]: score of tag j -> tag k
        self.start = nn.Parameter(torch.zeros(K))
        self.end = nn.Parameter(torch.zeros(K))
        ban = torch.zeros(K, K)                      # I-x only after B-x or I-x
        for k, tag in enumerate(TAGS):
            if tag.startswith("I-"):
                ban[:, k] = -1e4
                ban[t2i["B-" + tag[2:]], k] = ban[k, k] = 0.0
        self.register_buffer("ban", ban)
        self.register_buffer("ban_start", torch.tensor(
            [-1e4 if t.startswith("I-") else 0.0 for t in TAGS]))

    def trans(self):
        return self.A + self.ban

    def score(self, u, y, mask):                     # S(x, y) for the gold path
        s = (self.start + self.ban_start)[y[:, 0]] + u[:, 0].gather(1, y[:, :1]).squeeze(1)
        for t in range(1, u.size(1)):
            step = self.trans()[y[:, t - 1], y[:, t]] + u[:, t].gather(1, y[:, t:t +
             ↪  1]).squeeze(1)
            s = s + step * mask[:, t]
        last = y.gather(1, (mask.sum(1) - 1).unsqueeze(1)).squeeze(1)
        return s + self.end[last]

    def log_partition(self, u, mask):                # forward algorithm in log space
        alpha = self.start + self.ban_start + u[:, 0]
        for t in range(1, u.size(1)):
            nxt = torch.logsumexp(alpha.unsqueeze(2) + self.trans() + u[:, t].unsqueeze(1),
             ↪  dim=1)
            alpha = torch.where(mask[:, t:t + 1], nxt, alpha)   # padding: carry forward
        return torch.logsumexp(alpha + self.end, dim=1)

    def nll(self, u, y, mask):
        return (self.log_partition(u, mask) - self.score(u, y, mask)).mean()

    def viterbi(self, u, mask):                      # same recursion with max + backpointers
        delta, back = self.start + self.ban_start + u[:, 0], []
        for t in range(1, u.size(1)):
            best, arg = (delta.unsqueeze(2) + self.trans()).max(1)
            back.append(arg)
            delta = torch.where(mask[:, t:t + 1], best + u[:, t], delta)
        delta = delta + self.end
        paths = []
        for b in range(u.size(0)):
            k = int(delta[b].argmax())
            path = [k]
            for t in range(int(mask[b].sum()) - 1, 0, -1):
                k = int(back[t - 1][b, k])
```

```
                path.append(k)
            paths.append(path[::-1])
        return paths

# Check the masked forward algorithm and Viterbi against brute-force enumeration
import itertools
torch.manual_seed(0)
check = CRF(len(TAGS))
with torch.no_grad():
    check.A.normal_()
u = torch.randn(1, 3, len(TAGS))
m = torch.ones(1, 3, dtype=torch.bool)
paths = [torch.tensor([p]) for p in itertools.product(range(len(TAGS)), repeat=3)]
scores = torch.stack([check.score(u, p, m)[0] for p in paths])
assert torch.allclose(check.log_partition(u, m)[0], torch.logsumexp(scores, 0), atol=1e-4)
assert paths[int(scores.argmax())][0].tolist() == check.viterbi(u, m)[0]

# ---- 6. Train three taggers and evaluate exact spans on unseen names ----
def run(use_chars, use_crf, seed, epochs=15):
    torch.manual_seed(seed)
    r = random.Random(seed)
    enc, crf = Encoder(use_chars), CRF(len(TAGS))
    params = list(enc.parameters()) + (list(crf.parameters()) if use_crf else [])
    opt = torch.optim.Adam(params, lr=5e-3)
    order = list(range(len(train_data)))
    for _ in range(epochs):
        r.shuffle(order)
        for i in range(0, len(order), 32):
            x, c, y, mask = batch([train_data[j] for j in order[i:i + 32]], 0.25, r)
            u = enc(x, c)
            loss = (crf.nll(u, y, mask) if use_crf      # sequence-level loss
                    else nn.functional.cross_entropy(u[mask], y[mask]))  # per token
            opt.zero_grad()
            loss.backward()
            nn.utils.clip_grad_norm_(params, 5.0)
            opt.step()
    enc.eval()
    with torch.no_grad():
        x, c, y, mask = batch(test_data)
        u = enc(x, c)
        pred = (crf.viterbi(u, mask) if use_crf else
                [u[b, :int(mask[b].sum())].argmax(-1).tolist() for b in range(len(u))])
    pred = [[TAGS[k] for k in p] for p in pred]
    gold = [t for _, t in test_data]
    return span_prf(gold, pred), sum(map(invalid, pred)), error_types(gold, pred), crf

results = {}
for name, use_chars, use_crf in [("BiLSTM, words", False, False),
                                 ("BiLSTM, words+chars", True, False),
                                 ("BiLSTM-CRF, words+chars", True, True)]:
    runs = [run(use_chars, use_crf, seed) for seed in (0, 1, 2)]
    results[name] = runs
    f1s = [prf[2] for prf, _, _, _ in runs]
    errors = {k: sum(r[2][k] for r in runs) for k in runs[0][2]}
    print(f"{name}: F1 per seed {[round(f, 3) for f in f1s]}, mean {sum(f1s) / 3:.3f}")
    print(f"    invalid I- tags {[r[1] for r in runs]}, errors over 3 seeds {errors}")
```

```
assert all(r[1] == 0 for r in results["BiLSTM-CRF, words+chars"])
```

Output:

```
(['Vipo', 'repaired', 'Valve', 'F', 'in', 'Port', 'Suvika', '.'], ['B-PER', 'O', 'B-DEV',
 ↪  'I-DEV', 'O', 'B-LOC', 'I-LOC', 'O'])
train/test sentences: 100 200  gold test spans: 497
BiLSTM, words: F1 per seed [0.893, 0.771, 0.82], mean 0.828
    invalid I- tags [0, 30, 0], errors over 3 seeds {'type': 66, 'boundary': 43, 'missed':
     ↪  247, 'spurious': 0}
BiLSTM, words+chars: F1 per seed [0.893, 0.903, 0.942], mean 0.913
    invalid I- tags [4, 1, 0], errors over 3 seeds {'type': 124, 'boundary': 2, 'missed': 7,
     ↪  'spurious': 0}
BiLSTM-CRF, words+chars: F1 per seed [0.915, 0.909, 0.961], mean 0.929
    invalid I- tags [0, 0, 0], errors over 3 seeds {'type': 102, 'boundary': 0, 'missed': 8,
     ↪  'spurious': 0}
```

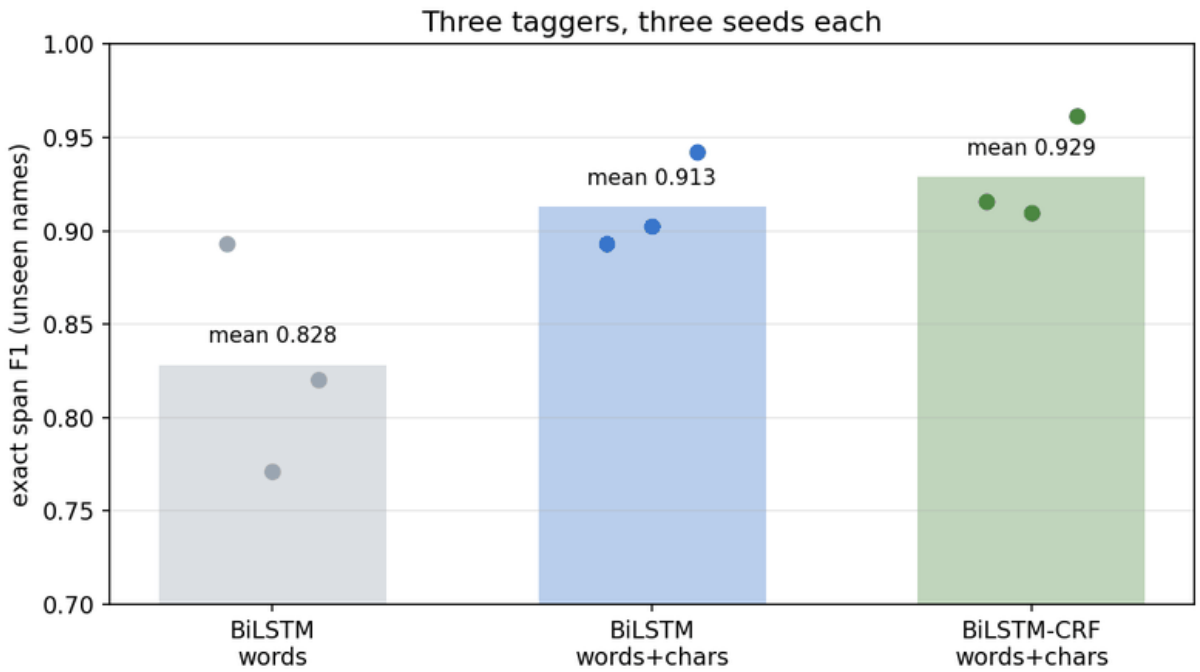


Figure 3 — Exact span F1 on test sentences whose names never occur in training. Dots are seeds; bars are means. Character features give the largest gain; the constrained CRF adds a smaller, consistent one.

Reading the results

Character features fix detection. The word-only tagger sees most test names as the unknown id. Its errors are dominated by missed entities (247 over three seeds), plus boundary errors when

a two-token name is split or truncated. With character features, misses drop to 7 and boundary errors to 2: the suffixes and capitalization of new names are enough to find them. Mean F1 rises from 0.828 to 0.913.

What remains is typing. Almost all remaining errors for both character models are type errors (124 and 102 over three seeds): the span is right but a person is labeled as an organization or the reverse. They involve single-token names, often in templates that accept either type in the same slot, such as `Mimitek reported a fault` or `checked by Renaka`. Context cannot settle those cases, the spelling of a person name carries no type marker in this corpus, and a model trained on 100 sentences has not learned the `tek` and `corp` endings reliably enough to override a context that also fits a person. This is the limit described in the character-features section: spelling helps only when it encodes the distinction.

The CRF guarantees valid output and helps a little. The word-only tagger produced 30 invalid `I-` tags in one seed and the character softmax tagger produced 4 and 1 in two seeds; the CRF with BIO constraints produced none, by construction. Its mean F1 is 0.929 against 0.913, and it is at least as good as the softmax tagger for each seed. With 100 training sentences that is a small margin; it is the consistency of the improvement and the guarantee of well-formed spans, not its size, that justifies the extra layer here.

Seeds matter again. The word-only tagger ranges from 0.771 to 0.893 across seeds. The seed with 30 invalid transitions is also the worst one. As in NLP 05, a single run would have been an unreliable summary.

What this does not show. The corpus is synthetic, tiny and templated; the test sentences reuse the training templates; and the tag set is small. Real benchmarks contain ambiguous context, nested and discontinuous mentions, and annotation disagreements. The experiment isolates three mechanisms (unknown-word handling, character evidence and output constraints) so that each effect can be seen and measured; it does not estimate how large those effects are on CoNLL-2003 or any other dataset.

What the transitions learned. Figure 4 shows the learned transition scores for one seed. Inside the allowed region, the CRF has learned the corpus's structure: `B-LOC → I-LOC` and `B-DEV → I-DEV` are strongly positive, while `I-ORG → I-ORG` and `I-LOC → I-LOC` are strongly negative, because no name in this corpus is longer than two tokens. A learned transition table is also a modeling assumption to inspect. Trained on a domain where every name has at most two tokens, it will penalize a three-token name elsewhere.

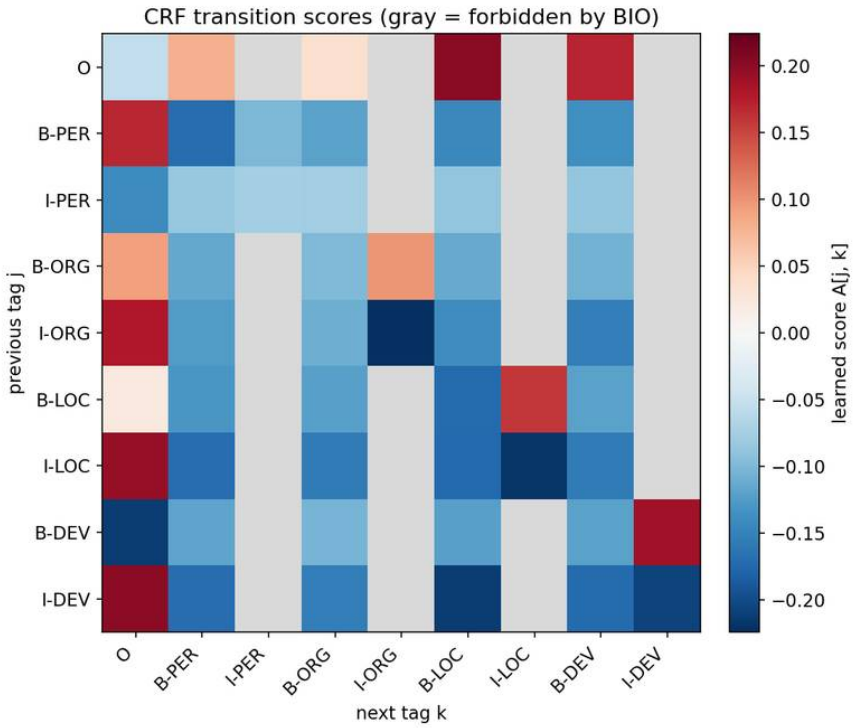


Figure 4 — Learned transition scores of the CRF (seed 0). Gray cells are BIO transitions forbidden by the constraint mask; the remaining scores were learned from 100 training sentences.

Modern view

Plain version. Current extraction systems combine pretrained representations with explicit task definitions. General language generation increases flexibility, while leaving span fidelity and support as separate problems.

Li and colleagues' NER survey organizes earlier deep systems by input representation, context encoder and tag decoder [7]. This taxonomy is still useful: changing the encoder does not determine whether the output is a flat sequence or overlapping spans. Xu and colleagues' 2024 generative extraction survey adds a task axis—entities, relations, events and unified extraction—and an adaptation axis including prompting, examples, supervised tuning and constrained generation [8]. Read these maps as design dimensions, rather than a ranking of universally superior architectures.

From BiLSTM-CRF to pretrained encoders. In Li and colleagues' taxonomy, this chapter's experiment varies the input representation (words versus words plus characters) and the tag decoder (softmax versus CRF) while keeping a BiLSTM context encoder [7]. Between 2016 and 2018 the BiLSTM-CRF with character features [14], [15] became the standard recipe, and contextual string embeddings from character language models extended it [16]. Pretrained Transformer encoders then replaced the BiLSTM as the context encoder in most systems; a linear or CRF tag decoder on

top of them is the token-classification head discussed in NLP 10 and NLP 11. The decoder questions of this chapter (valid transitions, span boundaries, exact evaluation) carry over unchanged.

Large language models for open entity recognition. UniversalNER (ICLR 2024) distills a large language model into much smaller models for open-domain NER through targeted instruction tuning, and evaluates them on a benchmark assembled from 43 NER datasets across 9 domains [22]. Together with GLiNER [9], it represents a move from a fixed tag inventory to entity types supplied at inference time.

GLiNER is a 2024 original example of label-conditioned span extraction with a bidirectional Transformer [9]. Its design shows that adapting the requested entity categories need not require autoregressively generating a long list of records. A structured span output can retain direct alignment with input text. The relevant comparison for an application includes supported category definitions, nested-span behavior, latency and error review, alongside aggregate accuracy.

A generative extractor instead emits a textual serialization of the requested record. Enforcing a valid schema can prevent missing keys or malformed syntax, but valid serialization does not guarantee a correct name, relation or evidence interval. A practical validation layer checks that quoted evidence exists, offsets match, categories are allowed and entity identities are consistent. These checks eliminate some errors while leaving interpretation errors for evaluation or review.

Open problems include changing category inventories, specialized language, nested entities, cross-document identity and reliable uncertainty under domain shift. Multilingual extraction adds different name boundaries, writing systems and conventions for titles. Treat a language-specific annotation guideline as part of the task definition; translating category names is insufficient. Evaluate documents by source and language while keeping enough examples to make each comparison interpretable.

For a new project, establish a deterministic or compact supervised baseline, freeze an annotation guide and inspect its error types before choosing a larger model. Use a held-out document set that prevents repeated templates or near-duplicate reports crossing the split. Compare complete supported records, not only whether a model returns parseable output. The result is a measurable extraction service whose failures can be traced back to evidence.

New labels test both representation and interpretation

Label-conditioned span extraction [9] and generative extraction [8] change how an application supplies its category inventory. That flexibility creates an analytical question: does the model recognize the intended category, or merely react to a familiar label name? Compare a short label with a precise description and deliberately similar competing descriptions. For example, distinguish an equipment manufacturer from an equipment operator; both may be organization mentions, but their event roles differ.

Keep entity detection, requested typing and relation assignment as separate judgments. A correctly located organization span does not establish that it manufactured the device. Likewise, a generated record that fills every requested field may invent a participant when the source omitted it. Test an explicit unknown answer against forced completion. The comparison should reveal

which output interface preserves missing information and which training examples distinguish closely related schemas. Broader category coverage is useful only when the semantic boundaries remain clear.

Key takeaways

- Sequence labeling assigns one label per token, but labels interact; a structured decoder chooses the best complete sequence under a stated factorization.
- HMM Viterbi decoding is exact for its first-order factorization; a linear-chain CRF keeps the same recursions while scoring tags conditionally, so a neural encoder can supply the scores.
- Entity recognition starts with a schema and offsets; BIO labels encode flat spans, and nested mentions need a span representation.
- Evaluate exact spans (boundaries and type), and break errors into missed, spurious, boundary and type errors instead of reporting token accuracy.
- A BiLSTM tagger reads the whole sentence in both directions; masks and trained unknown-word ids are essential details.
- A CRF layer trained by log-likelihood (forward algorithm) and decoded by Viterbi makes tags agree; BIO constraints make invalid sequences impossible.
- Character features recover unseen names from their spelling but cannot resolve ambiguity the spelling does not encode.
- Information extraction builds relations and events on top of entity spans; each layer adds its own record contract and failure modes.

Exercises

1. **A sequence decision.** Change one transition row in the HMM while preserving its normalization. Find the condition under which the best path for "fish fish" becomes two nouns.

   Hint

   Compare the complete path products. The initial factor and first emission are shared by the two noun-initial candidates.

2. Extend the CPU extraction example with a new person spelling that is absent from the inventory. Add its gold span and report the change in exact recall.

   Hint

   Keep the original inventory frozen. A missed mention is useful evidence about the baseline's scope, rather than a reason to edit the test vocabulary into training.

3. Design a schema for overlapping organization and department names. Explain why a single flat BIO sequence cannot express every allowed annotation.

Hint

Draw the character intervals first. Decide whether nested names count as separate entities and whether a department is linked to its parent organization.

4. In the PyTorch experiment, remove the BIO constraint mask from the CRF (set `ban` and `ban_start` to zeros) and retrain. Count invalid `I-` tags over three seeds, then inspect the learned transition scores for `O` followed by `I-ORG`.

Hint

Without the mask, invalid transitions are discouraged only to the extent the 100 training sentences teach it. Compare the learned score with the scores of allowed transitions into `I-ORG`.

5. Add a character-only ablation: drop the word embedding and keep the character BiLSTM. Then give person names a distinctive ending in the generator (for example `-ko`) and retrain all models. Which error type changes, and why?

Hint

Use the error breakdown printed by the script. A new spelling cue should move errors out of the type column only for the names that carry it.

6. Write a partial-match evaluator that gives credit when a predicted span overlaps a gold span of the same type. Report it next to exact F1 for the word-only tagger and explain why the two can rank models differently.

Hint

Define the matching rule precisely (for example one-to-one matching by largest overlap) and report it under its own name. The word-only tagger's boundary errors are where the two scores diverge.

References

1. Susan McRoy, *Principles of Natural Language Processing*, open textbook, 2021; Chapters 3–5 and 8. Open textbook
2. Masato Hagiwara, *Real-World Natural Language Processing: Practical Applications with Deep Learning*, Manning, 2021; Chapter 5, "Sequential labeling and language modeling." Chapter
3. L. R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proceedings of the IEEE*, 77(2), 1989. Paper
4. M.-C. de Marneffe, C. D. Manning, J. Nivre and D. Zeman, "Universal Dependencies," *Computational Linguistics*, 47(2), 2021. Paper
5. J. Lafferty, A. McCallum and F. Pereira, "Conditional random fields: Probabilistic models for segmenting and labeling sequence data," ICML, 2001. Paper

6. Erik F. Tjong Kim Sang and Fien De Meulder, “Introduction to the CoNLL-2003 Shared Task: Language-Independent Named Entity Recognition,” CoNLL, 2003. Paper
7. Jing Li, Aixin Sun, Jianglei Han and Chenliang Li, “A Survey on Deep Learning for Named Entity Recognition,” IEEE Transactions on Knowledge and Data Engineering 34(1), 2022. Author-hosted paper
8. Derong Xu, Wei Chen, Wenjun Peng, Chao Zhang, Tong Xu, Xiangyu Zhao, Xian Wu, Yefeng Zheng, Yang Wang and Enhong Chen, “Large language models for generative information extraction: a survey,” Frontiers of Computer Science 18(6), 2024. Survey
9. Urchade Zaratiana, Nadi Tomeh, Pierre Holat and Thierry Charnois, “GLiNER: Generalist Model for Named Entity Recognition using Bidirectional Transformer,” NAACL, 2024. Paper
10. Lance Ramshaw and Mitch Marcus, “Text Chunking using Transformation-Based Learning,” Third Workshop on Very Large Corpora, 1995. Paper
11. Mitchell P. Marcus, Beatrice Santorini and Mary Ann Marcinkiewicz, “Building a Large Annotated Corpus of English: The Penn Treebank,” *Computational Linguistics* 19(2), 1993. Paper
12. Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu and Pavel Kuksa, “Natural Language Processing (Almost) from Scratch,” *Journal of Machine Learning Research* 12, 2011. Paper
13. Zhiheng Huang, Wei Xu and Kai Yu, “Bidirectional LSTM-CRF Models for Sequence Tagging,” arXiv:1508.01991, 2015. arXiv
14. Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami and Chris Dyer, “Neural Architectures for Named Entity Recognition,” NAACL-HLT, 2016. Paper
15. Xuezhe Ma and Eduard Hovy, “End-to-end Sequence Labeling via Bi-directional LSTM-CNNs-CRF,” ACL, 2016. Paper
16. Alan Akbik, Duncan Blythe and Roland Vollgraf, “Contextual String Embeddings for Sequence Labeling,” COLING, 2018. Paper
17. Sameer Pradhan, Alessandro Moschitti, Nianwen Xue, Hwee Tou Ng, Anders Björkelund, Olga Uryupina, Yuchen Zhang and Zhi Zhong, “Towards Robust Linguistic Analysis using OntoNotes,” CoNLL, 2013. Paper
18. Leon Derczynski, Eric Nichols, Marieke van Erp and Nut Limsopatham, “Results of the WNUT2017 Shared Task on Novel and Emerging Entity Recognition,” Workshop on Noisy User-generated Text, 2017. Paper
19. Christopher Walker, Stephanie Strassel, Julie Medero and Kazuaki Maeda, *ACE 2005 Multilingual Training Corpus*, LDC2006T06, Linguistic Data Consortium, 2006. Catalog
20. Yuhao Zhang, Victor Zhong, Danqi Chen, Gabor Angeli and Christopher D. Manning, “Position-aware Attention and Supervised Data Improve Slot Filling,” EMNLP, 2017. Paper
21. Yuan Yao, Deming Ye, Peng Li, Xu Han, Yankai Lin, Zhenghao Liu, Zhiyuan Liu, Lixin Huang, Jie Zhou and Maosong Sun, “DocRED: A Large-Scale Document-Level Relation Extraction Dataset,” ACL, 2019. Paper
22. Wenxuan Zhou, Sheng Zhang, Yu Gu, Muhao Chen and Hoifung Poon, “UniversalNER: Targeted Distillation from Large Language Models for Open Named Entity Recognition,” ICLR, 2024. arXiv
23. Mike Schuster and Kuldip K. Paliwal, “Bidirectional Recurrent Neural Networks,” *IEEE Transactions on Signal Processing* 45(11), 1997. DOI

NLP 07 • Attention Mechanism and Transformers

From the encoder–decoder bottleneck to Bahdanau and Luong attention, scaled dot-product and multi-head attention, masks, positions and the Transformer encoder and decoder blocks, with attention and an encoder block built from scratch.

Online edition: https://blog.shyandram.dev/en/learn/nlp/attention-transformers/

What you'll learn

- Why squeezing a whole source sentence into one vector limits an encoder–decoder, and how attention removes that bottleneck.
- The additive (Bahdanau) and multiplicative (Luong) attention equations, and how they generalize to queries, keys and values.
- Scaled dot-product attention, multi-head attention, and the difference between self-attention and cross-attention.
- How padding masks and causal masks encode what each position is allowed to read, and how to test that they work.
- Why attention needs position information, and the main ways to supply it.
- How Transformer encoder and decoder blocks are assembled, what they cost, and which variants target long contexts.

The big picture

Recurrent networks (NLP 05) read a sentence one token at a time and carry a running summary. That summary has a fixed size, whether the sentence has five words or fifty. Attention changes the access pattern: instead of relying on one summary, a position can look back at every stored representation and decide which ones matter for the prediction it is making now. The Transformer takes this idea to its conclusion and builds both the encoder and the decoder out of attention and small per-token networks, with no recurrence at all.

> **Explain like I'm five**
>
> Imagine translating a long sentence. One way is to read it once, close the book, and write the translation from memory. Another way is to keep the book open and, for each word you write, glance back at the part of the sentence you need. Attention is the second way. The "glance" is a set of weights saying how much to look at each word. A mask is a rule about which parts of the page you may look at: in some games you may not peek at the words you have not written yet.

Three decisions organize this chapter, and they are worth keeping separate. The **scoring rule** decides how strongly one position matches another. The **visibility rule**, expressed as a mask,

decides which positions may be read at all. The **position rule** decides how word order enters a computation that would otherwise ignore it. A change to one does not fix a mistake in another: a better scoring function cannot repair a mask that leaks future tokens.

Hagiwara's Chapter 8 develops attention and Transformers for practical NLP [1]. This chapter adds the derivations, connects them to the original papers, and builds every mechanism in small CPU examples. ML 05 treats the same machinery from the vision side, with the full positional-encoding math; NLP 08 uses it to build a translation model.

1. The encoder–decoder bottleneck

Plain version. A sequence-to-sequence model has an encoder that reads the source and a decoder that writes the target. If the only bridge between them is one vector, every detail the decoder needs must fit through that vector.

The recurrent encoder–decoder models of Sutskever, Vinyals and Le [2] and Cho et al. [3] read a source sequence $x_1, \dots, x_T$ with an encoder RNN and pass its final state to the decoder:

$$h_j = f_{\text{enc}}(h_{j-1}, x_j), \qquad c = h_T, \qquad s_u = f_{\text{dec}}(s_{u-1}, y_{u-1}, c), \qquad p(y_u \mid y_{<u}, \mathbf{x}) = g(s_u).$$

Here $h_j \in \mathbb{R}^d$ is the encoder state after reading source token x_j; T is the source length; c is the single context vector; s_u is the decoder state at target step u; y_{u-1} is the previous target token; $y_{<u}$ is the target prefix; f_{enc} and f_{dec} are recurrent update functions such as LSTM or GRU cells; and g is an output layer followed by a softmax over the target vocabulary. ML 04 derives these cells and trains such a model.

The vector c has the same dimension d for every input. A ten-word sentence and a sixty-word paragraph must both be compressed into it before the first target word is produced. Bahdanau, Cho and Bengio named this fixed-length vector as a likely bottleneck for basic encoder–decoders and proposed letting the decoder search the source for the relevant parts instead [4]. Sutskever et al. also reported that reversing the source order helped their LSTM model, which is consistent with the same diagnosis: shortening the path between the first source words and the first target words made them easier to use [2].

The bottleneck is about information flow, not only capacity. Even a large c must be computed before the decoder knows which source detail it will need at step u. Attention postpones that choice: the encoder keeps one vector per source token, and the decoder chooses a different mixture of them at every step.

2. Additive attention (Bahdanau)

Plain version. At each target step, score every source position against what the decoder currently needs, turn the scores into weights that sum to one, and take the weighted average of the encoder states.

Bahdanau et al. [4] keep all encoder states $h_1, \dots, h_T$ (from a bidirectional encoder) and compute, for target step u,

$$e_{u,j} = v_a^\top \tanh\big(W_a s_{u-1} + U_a h_j\big), \qquad \alpha_{u,j} = \frac{\exp(e_{u,j})}{\sum_{k=1}^{T} \exp(e_{u,k})}, \qquad c_u = \sum_{j=1}^{T} \alpha_{u,j}\, h_j.$$

Here s_{u-1} is the previous decoder state; h_j is the encoder state for source position j; W_a, U_a and v_a are learned matrices and a learned vector; $e_{u,j}$ is the **alignment score**; $\alpha_{u,j}$ is the **attention weight**, nonnegative and summing to one over j; and c_u is the step-specific **context vector**, which the decoder uses together with its own state to predict y_u. The notation matches ML 04. Because the score passes the two vectors through a small feedforward layer before comparing them, this form is called **additive attention**.

Two consequences matter for NLP. First, the decoder now has a direct path to every source token, so the information about the first source word no longer has to survive many recurrent updates. Second, the weights $\alpha_{u,j}$ form a soft alignment between target and source positions, which can be plotted and inspected. That inspection is a diagnostic, not a guarantee: as discussed below, a large weight shows where the computation read from, not why the final prediction is correct.

3. Multiplicative attention (Luong)

Plain version. Replace the small scoring network by a dot product, possibly through one learned matrix. It is cheaper and turns attention into matrix multiplication.

Luong, Pham and Manning [5] compared several scoring functions for a **global** attention model that attends to all source states. They score the current decoder state h_u (rather than the previous one) against each source state $\bar{h}_j$:

$$\text{score}(h_u, \bar{h}_j) = \begin{cases} h_u^\top \bar{h}_j & \text{dot} \\ h_u^\top W_a\, \bar{h}_j & \text{general} \end{cases} \qquad a_u(j) = \frac{\exp \text{score}(h_u, \bar{h}_j)}{\sum_{j'} \exp \text{score}(h_u, \bar{h}_{j'})}.$$

Here h_u is the top-layer decoder state at target step u; $\bar{h}_j$ is the encoder state at source position j; W_a is a learned matrix; and $a_u(j)$ is the attention weight. Their third option, **concat**, scores the concatenation $[h_u; \bar{h}_j]$ with a feedforward layer, as in additive attention. The context vector $c_u = \sum_j a_u(j) \bar{h}_j$ is combined with the decoder state into an **attentional hidden state**, which feeds the output softmax:

$$\tilde{h}_u = \tanh(W_c [c_u; h_u]), \qquad p(y_u \mid y_{<u}, \mathbf{x}) = \text{softmax}(W_s \tilde{h}_u).$$

Here $[c_u; h_u]$ is vector concatenation and W_c, W_s are learned matrices. Luong et al. also proposed **local** attention, which first predicts an aligned source position and attends only to a window around it, and **input feeding**, which passes $\tilde{h}_u$ into the next decoder step so the model knows its earlier alignment decisions [5].

The "general" score is the bridge to Transformers. Write $q_u = h_u$ and $k_j = W_a^\top \bar{h}_j$; then the score is just $q_u^\top k_j$. Once every score is a dot product between a **query** and a **key**, a whole table of scores is

a single matrix product, which is what modern hardware does best.

4. Queries, keys and values: from alignment to self-attention

Plain version. Generalize the decoder state into a **query** (what this position is looking for), the source states into **keys** (what each position offers for matching) and **values** (what each position passes on if it is read). Attention then works between any two sets of vectors, including a sequence and itself.

In additive and multiplicative attention, the same encoder state is used both for matching and for the weighted sum. The Transformer separates these roles with three learned projections [6]. If the queries come from the decoder and the keys and values come from the encoder, the operation is **cross-attention**, the direct descendant of Bahdanau and Luong attention. If queries, keys and values all come from the same sequence, it is **self-attention**: every token builds a new representation of itself by reading the other tokens of its own sentence.

Self-attention produces **contextual representations**. In "the crane lifts steel" and "the crane eats fish," the embedding lookup for `crane` is the same, but after self-attention each occurrence has read different neighbors, so the two output vectors can differ. This is the representation idea that NLP 10 develops into BERT. Reuse of such representations requires the matching tokenizer and vocabulary: an embedding row has meaning only relative to the model's token IDs, and swapping in a different tokenizer without adapting the weights turns valid-looking numbers into unrelated symbols.

5. Scaled dot-product attention

Plain version. Each position asks which other positions contain useful information. Attention converts those comparisons into weights and combines the corresponding value vectors.

For one attention head:

$$Q = HW_Q, \quad K = HW_K, \quad V = HW_V, \qquad A = \mathrm{softmax}_{\mathrm{row}}\left(\frac{QK^\top}{\sqrt{d_k}} + B\right), \quad O = AV.$$

Here $H \in \mathbb{R}^{T\times d}$ contains input representations; $W_Q, W_K \in \mathbb{R}^{d\times d_k}$ are learned query and key projections; $W_V \in \mathbb{R}^{d\times d_v}$ is the value projection; Q, K, V are projected matrices; d_k and d_v are key and value widths; $B \in \mathbb{R}^{T\times T}$ is an additive visibility mask, with entries 0 (allowed) or $-\infty$ (blocked); A contains row-normalized attention weights; and $O \in \mathbb{R}^{T\times d_v}$ is the output. The softmax is applied separately to each query row. This is the mechanism introduced in the Transformer paper [6].

The scale factor prevents dot products from growing simply because the key vector has more dimensions under common initialization assumptions: if query and key components are independent with mean zero and unit variance, $q^\top k$ has variance d_k, and dividing by $\sqrt{d_k}$ restores unit variance so the softmax does not saturate (ML 05 works through this). It does not turn attention

weights into calibrated semantic importance. A large weight indicates a large contribution within this computation, after learned projections. It is not a proof that the attended token caused the final answer or supports the generated statement.

Code: attention from scratch on a toy sentence

To see what the equations do without a training run, we build the vectors by hand. Each token of "the pump stopped because it overheated" gets a four-feature vector (device noun, pronoun, event verb, function word), and the sentence is padded with one <pad> slot as it would be in a batch. The projections are chosen so that pronouns and verbs **look for** devices, nouns look for events, and a pronoun weakly **offers** device information. Nothing is learned; the point is to watch scores, masking and averaging work.

```
import numpy as np

tokens = ["the", "pump", "stopped", "because", "it", "overheated", "<pad>"]
# Hand-built features: [device noun, pronoun, event verb, function word]
X = np.array([[0, 0, 0, 1], [1, 0, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1],
              [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 0]], dtype=float)
# Query space: [looking for a device, looking for an event]
W_Q = 3.0 * np.array([[0, 1], [1, 0], [1, 0], [0, 0]])   # nouns seek events; pronouns, verbs
 ↪  seek devices
W_K = np.array([[1, 0], [0.5, 0], [0, 1], [0, 0]])       # devices offer "device"; a pronoun
 ↪  weakly so
W_V = np.eye(4)                                          # pass the features through
 ↪  unchanged

def attention(X, mask):
    Q, K, V = X @ W_Q, X @ W_K, X @ W_V
    scores = Q @ K.T / np.sqrt(K.shape[1])
    scores = np.where(mask, scores, -np.inf)              # blocked pairs get zero weight
    A = np.exp(scores - scores.max(axis=1, keepdims=True))
    A /= A.sum(axis=1, keepdims=True)
    return A @ V, A

T = len(tokens)
real = np.array([t != "<pad>" for t in tokens])
padding_mask = np.broadcast_to(real, (T, T))              # every query may read real keys
causal_mask = padding_mask & np.tri(T, dtype=bool)        # ...and only earlier ones
out, A = attention(X, padding_mask)
_, A_causal = attention(X, causal_mask)
it, pump = tokens.index("it"), tokens.index("pump")
print("weights of 'it':", " ".join(f"{t}={w:.2f}" for t, w in zip(tokens, A[it])))
print("output of 'it' (device, pronoun, event, function):", out[it].round(2))
print("'pump' reads 'stopped': bidirectional", A[pump, 2].round(2),
      "| causal", A_causal[pump, 2].round(2))
X2 = X.copy(); X2[-1] = [5, 5, 5, 5]                      # garbage in the padding slot
out2, _ = attention(X2, padding_mask)
print("padding changes real outputs:", not np.allclose(out[real], out2[real]))
assert np.allclose(A.sum(axis=1), 1) and A[:, -1].max() == 0
```

Output:

```
weights of 'it': the=0.07 pump=0.55 stopped=0.07 because=0.07 it=0.19 overheated=0.07
 ↪  <pad>=0.00
output of 'it' (device, pronoun, event, function): [0.55 0.19 0.13 0.13]
'pump' reads 'stopped': bidirectional 0.4 | causal 0.0
padding changes real outputs: False
```

The pronoun's new vector is mostly the device feature it borrowed from `pump`: after one attention step, `it` carries information about its likely antecedent. The padding column receives exactly zero weight, so even garbage stored in the padding slot cannot change a real token's output. Under the causal mask, `pump` can no longer read the later verb `stopped`. Real models learn the projections; here we wrote them by hand, and the example deliberately contains only one device, so it says nothing about how a trained model resolves harder references (that is a coreference problem, treated in NLP 12).

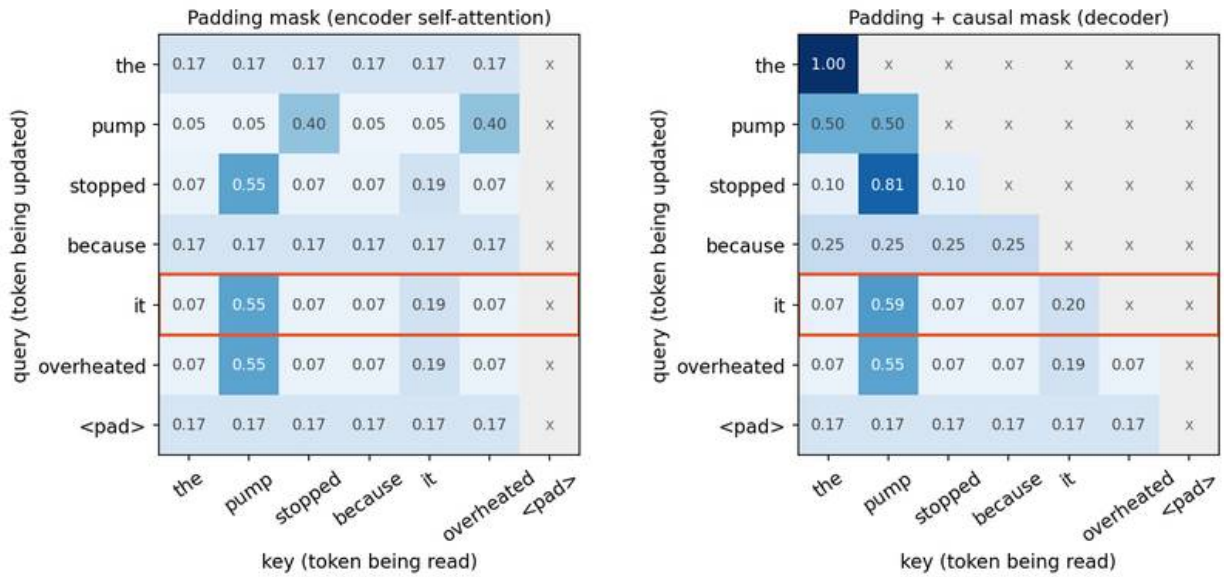


Figure 1 — The hand-built toy example. Left: padding mask only, as in an encoder. Right: padding plus causal mask, as in a decoder. Grey cells marked x are blocked; the orange row is the pronoun. The weights come from hand-chosen projections, not from a trained model.

Notice the **<pad> row** in the figure: a padding query still produces a (meaningless) output, because its keys are not all blocked. This keeps the manual softmax defined for this right-padded example. An all-blocked row makes the ordinary softmax undefined and can produce `NaN`; kernel behavior for that case should not be assumed. Reject empty real sequences or handle padding-only query rows explicitly, especially when combining left padding with a causal mask. Mask padding **keys**, and ignore padding **outputs** in the loss.

6. Multi-head attention

Plain version. Run several smaller attention operations in parallel, each with its own projections, so that different heads can match on different relationships. Then concatenate their outputs and mix them with one more projection.

With h heads, head r has its own projections $W_Q^{(r)}, W_K^{(r)}, W_V^{(r)} \in \mathbb{R}^{d \times d/h}$:

$$\text{head}_r = \text{Attention}\left(HW_Q^{(r)}, HW_K^{(r)}, HW_V^{(r)}\right), \qquad \text{MHA}(H) = [\,\text{head}_1, \ldots, \text{head}_h\,]\, W_O.$$

Here Attention is the scaled dot-product operation of Section 5 with $d_k = d_v = d/h$; the brackets concatenate the h outputs (each $T \times d/h$) back to width d; and $W_O \in \mathbb{R}^{d \times d}$ is the output projection. The original Transformer used $h = 8$ heads with model width $d = 512$ [6]. Because each head is h times narrower, multi-head attention costs about as much as one full-width head, and its four projection matrices hold $4d^2$ weights (plus biases), the same count as in ML 05.

One head forms one weighted average per query. If a verb needs both its subject and its object, a single softmax must split its weight between them, blurring both. Several heads can each concentrate on one relationship. Do not over-read this: heads are not guaranteed to learn human-readable roles, and some are redundant.

Two later variants change only how keys and values are shared. **Multi-query attention** uses one key head and one value head for all query heads, which shrinks the memory read at every decoding step [15]. **Grouped-query attention** shares each key/value head among a group of query heads, an intermediate point between the two [16]. Both matter mainly for fast autoregressive decoding, discussed in NLP 09.

7. Self-attention and cross-attention

Plain version. The same operation connects tokens within one sequence (self-attention) or connects one sequence to another (cross-attention). The only change is where the queries, keys and values come from.

Mechanism	Queries from	Keys and values from	Typical use in NLP
Bidirectional self-attention	the sequence	the same sequence, all real tokens	encoders such as BERT; the source side of a translator
Causal self-attention	the sequence	the same sequence, current and earlier positions	decoder-only language models; the target side of a translator
Cross-attention	the decoder	the encoder outputs	encoder–decoder translation and summarization

For cross-attention, let $H_{\text{dec}} \in \mathbb{R}^{T_y \times d}$ be decoder states and $M \in \mathbb{R}^{T_x \times d}$ the encoder outputs (often called the **memory**):

$$Q = H_{\text{dec}} W_Q, \qquad K = MW_K, \qquad V = MW_V, \qquad O = \text{softmax}_{\text{row}}\left(\frac{QK^\top}{\sqrt{d_k}} + B_{\text{src}}\right) V.$$

Here T_y and T_x are the target and source lengths; the score matrix has shape $T_y \times T_x$, one row per target position; B_{src} blocks source padding; and $O \in \mathbb{R}^{T_y \times d_v}$ has one output per target position. This is Bahdanau and Luong attention with learned query, key and value projections. No causal mask is needed here, because the source is fully known before decoding begins.

The task's available information should determine the mask, not a model name. A sentence classifier may use a bidirectional encoder because the complete sentence is available. A streaming continuation must exclude future output tokens. Translation can use the entire source after it arrives, but simultaneous translation, which starts speaking before the source sentence ends, needs a more restrictive source visibility rule.

8. Masks: padding and causal

Plain version. A mask is a table that says which query may read which key. Padding masks hide artificial filler tokens; causal masks hide the future.

For causal self-attention, $B_{ij} = 0$ when $j \le i$ and $B_{ij} = -\infty$ otherwise. Here i is the query position and j is the candidate key position. With shifted targets, the current input position is an allowed prefix token, so the diagonal remains visible. A fully bidirectional self-attention layer permits every real input position, although padding can still be blocked. For a padding mask, $B_{ij} = -\infty$ whenever key j is a padding position, for every query i. Masks combine by blocking a pair if either mask blocks it.

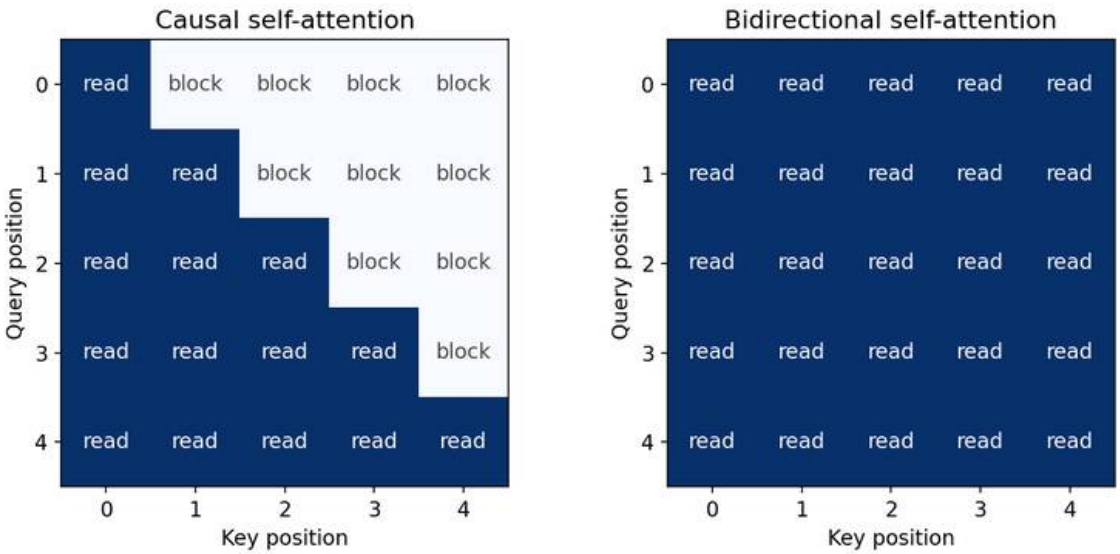


Figure 2 — Visibility is part of the task definition. Rows are querying positions and columns are the positions they may read.

Two different masks are easy to confuse. An **attention mask** controls what representations may read. A **loss mask** controls which target positions contribute to the training objective. Padding needs both: padded keys must not contribute context, and padded targets must not contribute loss. They solve different problems and should be tested independently.

A Boolean mask's meaning also depends on the API. In PyTorch's `scaled_dot_product_attention`, `True` means a pair is **allowed** [22]. In `MultiheadAttention`, `True` in `attn_mask` means **blocked**, and `True` in `key_padding_mask` marks an ignored key [23]. Name the logical relation first (`allowed` or `blocked`) and convert explicitly. Reusing the same Boolean array can silently reverse the information-flow rule. Floating-point masks instead add score biases, such as zero or negative infinity; a floating array of zeros and ones is not the same as a Boolean mask.

The following independent example uses identity projections and uniform scores to make the expected output visible. The first query reads only the first value; the second averages both. The two APIs agree after the Boolean mask is inverted. SDPA also takes a dropout probability directly, so evaluation must explicitly pass `dropout_p=0.0` [22].

```
import torch
from torch import nn
from torch.nn import functional as F

# Two queries and keys, with zero scores: allowed keys receive equal weights.
q = torch.zeros(1, 2, 2)
k = torch.zeros_like(q)
v = torch.tensor([[[10.0, 0.0], [0.0, 20.0]]])
allowed = torch.tril(torch.ones(2, 2, dtype=torch.bool))
blocked = ~allowed
sdpa = F.scaled_dot_product_attention(q[:, None], k[:, None], v[:, None],
                                      attn_mask=allowed, dropout_p=0.0)[:, 0]
mha = nn.MultiheadAttention(2, 1, bias=False, dropout=0.0, batch_first=True).eval()
with torch.no_grad():
    mha.in_proj_weight.copy_(torch.eye(2).repeat(3, 1))
    mha.out_proj.weight.copy_(torch.eye(2))
    module_out, _ = mha(q, k, v, attn_mask=blocked, need_weights=False)
expected = torch.tensor([[[10.0, 0.0], [5.0, 10.0]]])
assert torch.allclose(sdpa, expected)
assert torch.allclose(module_out, expected)
print('mask conventions agree:', torch.allclose(sdpa, module_out))
print('outputs:', sdpa[0].tolist())
```

```
mask conventions agree: True
outputs: [[10.0, 0.0], [5.0, 10.0]]
```

Inspect attention and test its information flow

This standalone matrix example separates learned scores from the visibility rule. All vectors and projections are synthetic, so the weights have no linguistic interpretation.

```
import numpy as np

rng = np.random.default_rng(0)
T, d = 5, 8                                  # 5 tokens, 8-dimensional embeddings
X = rng.normal(size=(T, d))                  # token embeddings (+ positions)
Wq, Wk, Wv = (rng.normal(size=(d, d)) / np.sqrt(d) for _ in range(3))

def attention(X, causal):
    Q, K, V = X @ Wq, X @ Wk, X @ Wv
    scores = Q @ K.T / np.sqrt(d)            # (T, T) query-key similarities
    if causal:                               # position t may only look at <= t
        scores = np.where(np.tri(T, dtype=bool), scores, -np.inf)
    w = np.exp(scores - scores.max(axis=1, keepdims=True))
    w /= w.sum(axis=1, keepdims=True)        # softmax over keys
    return w @ V, w

out, w = attention(X, causal=True)
print(np.round(w, 2))

# Changing the LAST token must not change any earlier causal output...
X2 = X.copy(); X2[-1] += 5.0
out2, _ = attention(X2, causal=True)
print("causal: earlier outputs unchanged:", np.allclose(out[:-1], out2[:-1]))
# ...but in a bidirectional encoder every position can see it.
b1, _ = attention(X, causal=False); b2, _ = attention(X2, causal=False)
print("bidirectional: earlier outputs unchanged:", np.allclose(b1[:-1], b2[:-1]))
```

Output:

```
[[1.   0.   0.   0.   0.  ]
 [0.88 0.12 0.   0.   0.  ]
 [0.42 0.29 0.29 0.   0.  ]
 [0.28 0.24 0.29 0.19 0.  ]
 [0.34 0.06 0.24 0.11 0.26]]
causal: earlier outputs unchanged: True
bidirectional: earlier outputs unchanged: False
```

The causal weights are lower-triangular: the first token can only read itself, the last can read everything. The numerical checks illustrate the mask's visibility rule. More generally, row t only aggregates value vectors from positions at most t. If its queries, keys and values were computed from states that also exclude future information, future inputs cannot affect that output. This condition must hold throughout the network, including any preprocessing that mixes positions. The same mask permits parallel evaluation of all next-token training losses while keeping each prediction restricted to its own prefix.

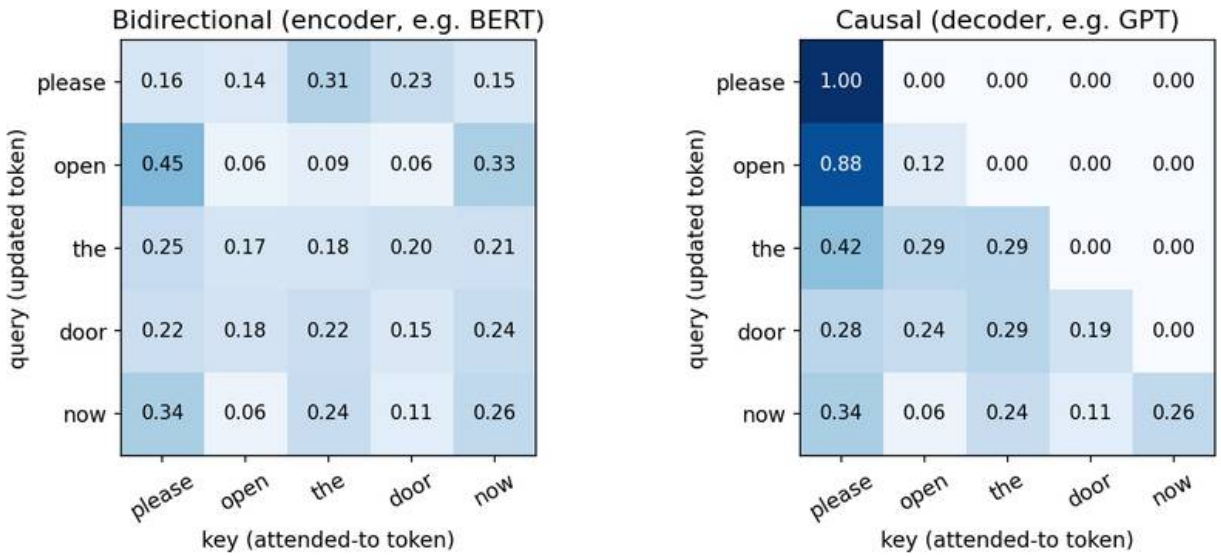


Figure 3 — Attention weights from the code above (random projections, so the values carry no linguistic meaning). The mask, not the weights, decides what each token may read.

A perturbation test like this one is worth keeping in any real code base. Low training loss cannot reveal a leaking mask, because a model that can see the answer also achieves low loss; only a test that changes a future input and checks earlier outputs targets the visibility contract directly.

9. Position: telling attention where tokens are

Plain version. Attention compares token representations, and position information tells it where those tokens occur. A token's identity alone does not say whether it is the subject or the final word.

Without position information, self-attention is **permutation-equivariant**: shuffle the input tokens and the outputs are shuffled the same way, because nothing in the formula refers to order. "Dog bites man" and "man bites dog" would produce the same set of vectors. The encoder-block code in Section 10 checks this property and shows that adding positions breaks it.

The simplest fix adds a position vector to each token embedding before the first layer. For a learned embedding and position table:

$$\mathbf{h}_i^{(0)} = E[x_i] + P[i].$$

Here $\mathbf{h}_i^{(0)} \in \mathbb{R}^d$ is the initial representation at position i; E is a token embedding table; $E[x_i]$ selects the row for token x_i; P is a position table; $P[i]$ selects the position row; and d is representation width. A learned position table also imposes a maximum supported position. Giving a sequence beyond that range is not an ordinary longer input; it is an invalid lookup. Record context limits in the serving contract. Extending the table or changing positional behavior is a model change that should be evaluated, rather than assumed to preserve performance on longer documents.

The original Transformer instead used fixed **sinusoidal** encodings [6]:

$$P[i]_{2m} = \sin\left(\frac{i}{10000^{2m/d}}\right), \qquad P[i]_{2m+1} = \cos\left(\frac{i}{10000^{2m/d}}\right), \qquad m = 0, \dots, \tfrac{d}{2} - 1.$$

Here i is the position and m indexes a pair of dimensions, each pair rotating at its own frequency. This is the same formula as in ML 05, written there as $p_{pos,2i}$; that page proves the two properties that make it useful (a fixed offset acts as a rotation, and dot products depend only on the offset) and plots the pattern. The original paper also multiplies token embeddings by $\sqrt{d}$ before adding the encoding [6]; the translation model in NLP 08 follows that convention.

Later schemes put position into the attention score itself, so that the model sees **relative** offsets:

- **Relative position representations** add learned vectors that depend on the clipped offset $j - i$ to the keys and values [9].
- **Rotary position embedding (RoPE)** rotates each query and key by an angle proportional to its position, so the score $q_i^\top k_j$ depends on positions only through $i - j$ [10].
- **ALiBi** adds a fixed, head-specific penalty proportional to the distance $i - j$ to each causal attention score, and was designed so that a model trained on short inputs can be evaluated on longer ones [11].

The choice interacts with the context-length contract. A learned absolute table has a hard maximum; relative methods remove that hard limit but do not guarantee good accuracy beyond the lengths seen in training. Evaluate at the lengths you intend to serve.

10. The Transformer encoder and decoder blocks

Plain version. A block alternates two jobs. Attention lets tokens exchange information; a small feedforward network then processes each token on its own. Residual connections and normalization keep a deep stack trainable. The decoder adds one more job between them: reading the encoder through cross-attention.

The original **encoder block** [6] wraps each sublayer in a residual connection followed by layer normalization (**post-norm**):

$$U = \mathrm{LN}(H + \mathrm{MHA}(H)), \qquad H' = \mathrm{LN}(U + \mathrm{FFN}(U)), \qquad \mathrm{FFN}(u) = W_2 \max(0, W_1 u + b_1) + b_2.$$

Here $H \in \mathbb{R}^{T \times d}$ is the block input; MHA is multi-head self-attention with a padding mask; LN is layer normalization, which standardizes each token's d features and applies a learned scale and shift [7]; FFN is applied to each position separately with $W_1 \in \mathbb{R}^{d_{\mathrm{ff}} \times d}$, $W_2 \in \mathbb{R}^{d \times d_{\mathrm{ff}}}$ and a hidden width d_{ff} (2048 for $d = 512$ in the original); and H' is the block output. Many later models move the normalization **before** each sublayer (**pre-norm**, $H + \mathrm{MHA}(\mathrm{LN}(H))$). Xiong et al. analyzed why post-norm needs a learning-rate warm-up while pre-norm can train without one [8]. So "a Transformer" is not fully specified until you say where normalization goes.

The **decoder block** has three sublayers, each with its own residual connection and normalization:

1. **Masked self-attention** over the target prefix, with a causal mask (and a padding mask for batches).
2. **Cross-attention**, with queries from step 1 and keys and values from the final encoder output, masked only for source padding.
3. A position-wise **feedforward network**.

Stacking N encoder blocks and N decoder blocks gives the full model ($N = 6$ in the original [6]). Every decoder block reads the same final encoder output. A linear layer and a softmax over the target vocabulary turn the last decoder state into $p(y_t \mid y_{<t}, \mathbf{x})$. Dropping the encoder and the cross-attention leaves a **decoder-only** model, the shape of most current large language models (NLP 09); dropping the decoder leaves an **encoder-only** model such as BERT (NLP 10).

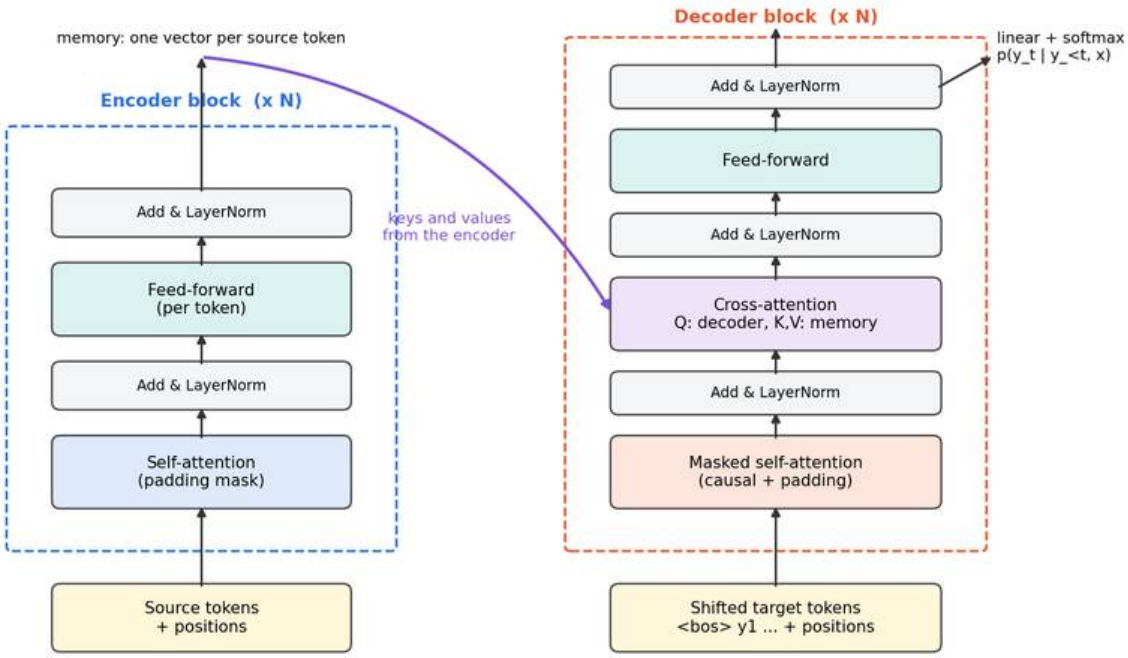


Figure 4 — The original post-norm Transformer encoder and decoder blocks. Residual connections around each sublayer are folded into the Add and LayerNorm boxes. The purple arrow is cross-attention: every decoder block reads the final encoder output.

Code: a tiny Transformer encoder block from scratch

The following code implements multi-head self-attention and a post-norm encoder block directly from the equations, borrows the randomly initialized weights of PyTorch's `nn.TransformerEncoderLayer` [21], and checks that both give the same output. It then tests three properties: padding is ignored, the block is permutation-equivariant without positions, and sinusoidal positions break

that symmetry. Dropout is off and the reference layer is in evaluation mode so the comparison is exact.

```
import math
import torch
from torch import nn
from torch.nn import functional as F

torch.manual_seed(0)
B, T, d, h, d_ff = 2, 6, 16, 4, 64
ref = nn.TransformerEncoderLayer(d, h, d_ff, dropout=0.0, activation="relu",
                                 batch_first=True, norm_first=False).eval()

def multi_head_attention(x, key_padding, p):
    q, k, v = (x @ p.in_proj_weight.T + p.in_proj_bias).chunk(3, dim=-1)
    split = lambda t: t.view(B, T, h, d // h).transpose(1, 2)        # (B, h, T, d/h)
    q, k, v = split(q), split(k), split(v)
    scores = q @ k.transpose(-2, -1) / math.sqrt(d // h)               # (B, h, T, T)
    scores = scores.masked_fill(key_padding[:, None, None, :], float("-inf"))
    heads = scores.softmax(dim=-1) @ v                                  # (B, h, T, d/h)
    concat = heads.transpose(1, 2).reshape(B, T, d)                     # heads side by side
    return concat @ p.out_proj.weight.T + p.out_proj.bias

def encoder_block(x, key_padding, layer=ref):
    # Post-norm, as in the original Transformer: x <- LN(x + Sublayer(x))
    a = multi_head_attention(x, key_padding, layer.self_attn)
    x = F.layer_norm(x + a, (d,), layer.norm1.weight, layer.norm1.bias, layer.norm1.eps)
    f = layer.linear2(F.relu(layer.linear1(x)))
    return F.layer_norm(x + f, (d,), layer.norm2.weight, layer.norm2.bias, layer.norm2.eps)

x = torch.randn(B, T, d)
pad = torch.tensor([[False] * 6, [False] * 4 + [True] * 2])     # second sentence: 4 real
 ↪  tokens
with torch.no_grad():
    ours, theirs = encoder_block(x, pad), ref(x, src_key_padding_mask=pad)
    print("matches nn.TransformerEncoderLayer:", torch.allclose(ours[~pad], theirs[~pad],
     ↪  atol=1e-5))
    x2 = x.clone(); x2[1, 4:] = 100.0                                # garbage in padded slots
    print("padding is ignored:", torch.allclose(encoder_block(x2, pad)[~pad], ours[~pad],
     ↪  atol=1e-5))
    perm = torch.tensor([3, 0, 5, 1, 4, 2])
    none = torch.zeros(B, T, dtype=torch.bool)
    y, yp = encoder_block(x, none), encoder_block(x[:, perm], none)
    print("no positions -> permutation-equivariant:", torch.allclose(yp, y[:, perm],
     ↪  atol=1e-5))
    pos = torch.arange(T)[:, None] / 10000 ** (torch.arange(0, d, 2) / d)
    pe = torch.stack([pos.sin(), pos.cos()], dim=-1).flatten(1)     # sin/cos interleaved
    y, yp = encoder_block(x + pe, none), encoder_block(x[:, perm] + pe, none)
    print("with positions -> still equivariant:", torch.allclose(yp, y[:, perm], atol=1e-5))
print("parameters:", sum(p.numel() for p in ref.parameters()))
```

Output:

```
matches nn.TransformerEncoderLayer: True
padding is ignored: True
no positions -> permutation-equivariant: True
with positions -> still equivariant: False
parameters: 3280
```

The comparison uses only real tokens: in evaluation mode PyTorch may take a fused path that does not compute meaningful values at padded positions, and those outputs are ignored anyway. The parameter count checks the formulas: attention has $4d^2 + 4d = 1088$ weights and biases, the feedforward network $2d\,d_{\text{ff}} + d_{\text{ff}} + d = 2128$, and the two layer norms $4d = 64$. Notice where the parameters live: in this block, as in the original Transformer, the feedforward network holds most of them.

The model has no positional information until we add it, which is exactly what the third and fourth checks show. In a real model, the position vectors are added once, at the input of the first block; every later block inherits order information through the residual stream.

Online only: Transformer — Attention Is All You Need — https://blog.shyandram.dev/en/learn/nlp/attention-transformers/papers/transformer–attention-is-all-you-need/

11. Cost and long-context variants

Plain version. Every query compares itself with every key, so the work and the score table grow with the square of the sequence length. That is cheap for a sentence and expensive for a book.

For a sequence of length T and width d, one self-attention layer computes $QK^\top$ and AV in about T^2d multiply–adds each, plus about $4Td^2$ for the projections. A recurrent layer costs about Td^2 but needs T **sequential** steps, whereas all positions of a self-attention layer can be computed in parallel, and any two positions are connected by a path of length one. Vaswani et al. summarize this trade-off in their comparison of layer types [6]. For short sentences (T smaller than d), self-attention is the cheaper of the two; for long documents the T^2 term dominates.

Direct attention score storage grows quadratically with sequence length. Specialized implementations can avoid materializing every intermediate matrix, but the modeling question remains which positions need to interact. In deployment, benchmark memory and latency under the actual context lengths rather than extrapolating from a short sentence.

Approaches to long contexts fall into a few families. Tay et al.'s survey organizes them as fixed patterns, learnable patterns, low-rank and kernel methods, memory, recurrence and downsampling [17]. Representative examples:

Idea	Example	What changes
Restrict which pairs interact	Longformer: a sliding local window plus a few global tokens [12]	cost grows linearly with T for a fixed window; distant tokens communicate through global tokens or several layers
Replace the softmax kernel	linear attention with a feature map [13]	scores factorize, giving linear cost and a recurrent view of causal attention; the attention function is different
Keep exact attention, move less memory	FlashAttention: tiled, IO-aware computation [14]	same result as standard attention, less memory traffic; quadratic arithmetic remains
Share keys and values across heads	multi-query and grouped-query attention [15, 16]	smaller key/value cache during decoding
Position schemes that extend	relative, rotary, ALiBi [9–11]	removes a hard positional limit; accuracy at unseen lengths must still be measured

These are not interchangeable. FlashAttention computes the same function as standard attention; Longformer and linear attention compute different functions and need to be evaluated as different models. State-space and recurrent alternatives that avoid pairwise scores entirely are discussed in ML 04 and ML 05.

Modern view

Plain version. The attention block of 2017 is still recognizable in current models; most changes concern normalization placement, position handling, key/value sharing and efficient kernels, not the core equation.

Surveys. Lin, Wang, Liu and Qiu's survey of Transformer variants (*AI Open*, 2022) first describes the original model, then proposes a taxonomy and reviews variants from three perspectives: architectural modification, pretraining, and applications [18]. It is a good map for locating a variant ("is this a change to attention, to position, or to the block?"). Tay, Dehghani, Bahri and Metzler's *Efficient Transformers: A Survey* (*ACM Computing Surveys*, 2022) focuses on the memory and compute of attention and classifies efficient models by mechanism (the families in Section 11) [17]. Its practical message is that efficiency claims depend on sequence length, hardware and implementation, so compare models at the lengths you need.

Attention as explanation. Jain and Wallace reported that attention weights in their models often did not agree with gradient-based importance measures and that very different attention distributions could yield the same prediction [19]. Wiegreffe and Pinter argued that whether attention can serve as an explanation depends on what is meant by explanation, and proposed tests for it [20]. Taken together: an attention map shows where a computation read from, which is useful for debugging masks and alignments, but it is not by itself evidence for why a model made a decision.

State of practice, 2023–2026. Grouped-query attention [16] and IO-aware exact attention ker-

nels [14] address the two costs that dominate serving: key/value memory during decoding and memory traffic during attention. Relative position methods such as RoPE [10] and ALiBi [11] are designed to remove the hard maximum length of a learned position table. Encoder–decoder Transformers remain a natural fit for tasks with a clear source and target, such as translation (NLP 08), while decoder-only models dominate open-ended generation (NLP 09).

Open problems. Reliable use of very long contexts (a longer window permits more input but does not guarantee that all of it is used), length generalization beyond training lengths, the cost of attention over long documents, and faithful interpretation of what attention heads compute.

Key takeaways

- A single context vector forces the whole source through a fixed-size bottleneck; attention lets every output step read every source position.
- Bahdanau attention scores with a small feedforward network; Luong's dot and general scores reduce attention to matrix products, which leads directly to queries, keys and values.
- Scaled dot-product attention is $\mathrm{softmax}(QK^\top/\sqrt{d_k}+B)V$; multi-head attention runs h narrower copies at about the cost of one.
- Self-attention mixes a sequence with itself; cross-attention lets a decoder read the encoder. The mask, not the weights, decides what each position may read.
- Mask padding keys, mask the future in causal models, keep loss masks separate, and test visibility with a perturbation check.
- Attention ignores order; learned, sinusoidal, relative, rotary or ALiBi positions supply it, each with a different length contract.
- Attention cost grows with T^2; long-context variants either change the function (sparse, linear) or compute the same function more efficiently (FlashAttention).

Exercises

1. Remove the causal mask from the information-flow example in Section 8. Which printed check changes, and why could a language model trained without the mask still show a low training loss?

   Hint

   The changed final token becomes visible to earlier positions. Low loss can reward access to information that is unavailable during generation.

2. In the toy-sentence code, add a second device noun ("valve") before the pronoun. What happens to the weights of `it`? What would a model need, beyond these hand-built features, to choose the right antecedent?

   Hint

With identical features, the two devices receive equal scores. Position information, the verb's meaning and discourse context must enter the query or key to break the tie.

3. Design attention and loss masks for a batch of two target sequences of lengths 3 and 5, padded to 5, in a causal decoder. Write the 5×5 blocked/allowed table for each sequence and the list of target positions that contribute to loss.

Hint

Padding keys must not contribute context, and padding targets must not contribute loss. Decide how a padding query row is handled without creating an all-blocked softmax row.

4. For a model with $d = 512$, $h = 8$ and $d_{\text{ff}} = 2048$, count the weights in one encoder block and one decoder block (ignore biases and layer norms). What fraction of the encoder block is in the feedforward network?

Hint

Self-attention and cross-attention each have $4d^2$ weights; the feedforward network has $2d\,d_{\text{ff}}$. The decoder has one more attention sublayer than the encoder.

5. For sequence lengths $T = 128$, 1,024 and 16,384 with $d = 512$, compare $T^2 d$ (attention scores) with $4Td^2$ (projections). At which length does the score term start to dominate?

Hint

The two terms are equal when $T = 4d$. Below that, the projections cost more than the score computation.

6. Explain why Luong's "general" score $h_u^\top W_a \bar{h}_j$ is a special case of scaled dot-product attention. What would the query, key and value projections be?

Hint

Absorb W_a into the key projection. The value projection is the identity, and the scale factor is 1.

References

1. Masato Hagiwara, *Real-World Natural Language Processing: Practical Applications with Deep Learning*, Chapter 8, Manning, 2021. Publisher ·Chapter 8
2. I. Sutskever, O. Vinyals and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," NeurIPS, 2014. Paper

3. K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk and Y. Bengio, “Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation,” EMNLP, 2014. Paper
4. D. Bahdanau, K. Cho and Y. Bengio, “Neural Machine Translation by Jointly Learning to Align and Translate,” ICLR, 2015. Paper
5. M.-T. Luong, H. Pham and C. D. Manning, “Effective Approaches to Attention-based Neural Machine Translation,” EMNLP, 2015. Paper
6. A. Vaswani et al., “Attention Is All You Need,” NeurIPS, 2017. Paper
7. J. L. Ba, J. R. Kiros and G. E. Hinton, “Layer Normalization,” arXiv:1607.06450, 2016. Paper
8. R. Xiong et al., “On Layer Normalization in the Transformer Architecture,” ICML, 2020. Paper
9. P. Shaw, J. Uszkoreit and A. Vaswani, “Self-Attention with Relative Position Representations,” NAACL, 2018. Paper
10. J. Su, M. Ahmed, Y. Lu, S. Pan, W. Bo and Y. Liu, “RoFormer: Enhanced Transformer with Rotary Position Embedding,” *Neurocomputing* 568, 2024. DOI
11. O. Press, N. A. Smith and M. Lewis, “Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation,” ICLR, 2022. Paper
12. I. Beltagy, M. E. Peters and A. Cohan, “Longformer: The Long-Document Transformer,” arXiv:2004.05150, 2020. Paper
13. A. Katharopoulos, A. Vyas, N. Pappas and F. Fleuret, “Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention,” ICML, PMLR 119, 2020. Paper
14. T. Dao, D. Y. Fu, S. Ermon, A. Rudra and C. Ré, “FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness,” NeurIPS, 2022. Paper
15. N. Shazeer, “Fast Transformer Decoding: One Write-Head is All You Need,” arXiv:1911.02150, 2019. Paper
16. J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebrón and S. Sanghai, “GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints,” EMNLP, 2023. Paper
17. Y. Tay, M. Dehghani, D. Bahri and D. Metzler, “Efficient Transformers: A Survey,” *ACM Computing Surveys* 55(6), 2022. DOI
18. T. Lin, Y. Wang, X. Liu and X. Qiu, “A Survey of Transformers,” *AI Open* 3, 2022. DOI
19. S. Jain and B. C. Wallace, “Attention is not Explanation,” NAACL, 2019. Paper
20. S. Wiegreffe and Y. Pinter, “Attention is not not Explanation,” EMNLP-IJCNLP, 2019. Paper
21. PyTorch, `TransformerEncoderLayer` documentation. Documentation
22. PyTorch, `scaled_dot_product_attention` documentation, version 2.15, accessed 2026-11-05. Documentation
23. PyTorch, `MultiheadAttention` documentation, version 2.15, accessed 2026-11-05. Documentation

NLP 08 • Sequence-to-Sequence Models with Transformers (Machine Translation, Chatbots)

Build and train an encoder–decoder Transformer for translation, decode it with greedy and beam search, score it with BLEU written from scratch, and carry the same conditional-generation ideas into dialogue state, clarification and conversation evaluation.

Online edition: https://blog.shyandram.dev/en/learn/nlp/seq2seq-transformers/

What you'll learn

- How machine translation moved from statistical phrase tables to neural encoder–decoders, and what the Transformer changed.
- Conditional sequence learning: target shifting, teacher forcing, padding-aware loss, and why exposure bias appears at generation time.
- How to train a small encoder–decoder Transformer translator on CPU, and decode it with greedy and beam search with length normalization.
- How BLEU works (written from scratch), what chrF and COMET add, and why none of them certifies meaning.
- How the same conditional-generation view applies to chatbots, and why task dialogue also needs explicit state, clarification and tool boundaries.
- How to evaluate whole conversations rather than isolated replies.

The big picture

A sequence-to-sequence (seq2seq) model reads one sequence and writes another: a French sentence from an English one, a reply from a conversation history, a summary from a document. The encoder turns the input into one vector per token; the decoder writes the output one token at a time, reading its own prefix through causal self-attention and the input through cross-attention (NLP 07). The same machinery serves translation and dialogue, but the two tasks fail differently. A translation can be fluent and reverse a negation. A chatbot can produce a friendly reply while querying the wrong device.

> **Explain like I'm five**
>
> Think of an interpreter at a meeting. First they listen to the whole sentence (the encoder). Then they speak the translation word by word, and before each word they glance back at the part of the speech they are translating (cross-attention). A good interpreter can also say "do you mean Pump A or Pump B?" when the speaker is unclear. Speaking smoothly is not the same as being right; the interpreter is judged by whether the listener ends up with the same meaning.

Two complementary readings anchor the chapter. Hagiwara's Chapter 6 explains sequence-to-sequence models for translation and conversational applications, and Chapter 8 covers attention and Transformers [1]. McRoy's Chapter 9 discusses translation as meaning-preserving generation [2]. The examples here are original, local CPU experiments; they teach mechanisms without downloading a model or a dataset. NLP 09 continues with open-ended generation, summarization and serving.

1. From statistical to neural machine translation

Plain version. Machine translation first learned word and phrase correspondences from aligned sentence pairs and combined them with a separate model of fluent target text. Neural translation replaced that pipeline with one network trained end to end.

Statistical machine translation (SMT) framed translation as a **noisy channel**. To translate a foreign sentence $\mathbf{f}$ into English $\mathbf{e}$, choose

$$\hat{\mathbf{e}} = \arg\max_{\mathbf{e}} \; p(\mathbf{e})\, p(\mathbf{f} \mid \mathbf{e}).$$

Here $p(\mathbf{e})$ is a language model that rewards fluent English; $p(\mathbf{f} \mid \mathbf{e})$ is a translation model that rewards faithfulness; and $\hat{\mathbf{e}}$ is the chosen translation. Brown et al.'s IBM models estimated word-level translation probabilities and alignments from sentence-aligned text [3]. Koehn, Och and Marcu's phrase-based translation used multi-word phrase pairs, which capture local reordering and idioms better than single words [4]. Such systems combined many separately trained components and features.

Neural networks entered as components first: Cho et al. used an RNN encoder–decoder to score phrase pairs inside an existing SMT system [5]. Sutskever, Vinyals and Le then trained an LSTM encoder–decoder to translate directly [6]. Bahdanau, Cho and Bengio added attention so the decoder could search the source at each step [7], and Vaswani et al. replaced recurrence with attention entirely [8]. Subword segmentation such as byte-pair encoding made open vocabularies practical, so rare words and names could be translated as sequences of known pieces [9]; NLP 02 builds a BPE vocabulary step by step. Stahlberg's review traces this transition from count-based SMT to neural MT in detail [22].

Two recent anchors show where the field went. The NLLB project built a single massively multilingual model covering 200 languages, with particular attention to low-resource languages [21]. The WMT24 general translation task compared submitted systems with several large language models and online providers under professional human evaluation; its title summarizes the finding: the LLM era is here, but MT is not solved [20].

2. Conditional sequence learning

Plain version. A translator models the probability of the target sentence given the source, one token at a time. Training shows it the correct previous target tokens; generation makes it use its own.

A conditional sequence model writes

$$p_\theta(\mathbf{y} \mid \mathbf{x}) = \prod_{t=1}^{U} p_\theta(y_t \mid y_{<t}, \mathbf{x}).$$

Here $\mathbf{x}$ is the source sequence; $\mathbf{y} = (y_1, \dots, y_U)$ is the target sequence, including an end token; U is its length; $y_{<t}$ contains the tokens before position t; and θ denotes learned parameters. The source can be visible in full while the target remains causal. This factorization describes a distribution. It does not certify that a sampled output is supported by the source.

Training usually supplies the correct preceding target tokens, a procedure called teacher forcing [6]. For a batch, the token-level loss is

$$\mathcal{L}(\theta) = -\frac{1}{M} \sum_{b=1}^{B} \sum_{t=1}^{T_b} m_{bt} \log p_\theta(y_{bt} \mid y_{b,<t}, \mathbf{x}_b).$$

Here B is the number of examples; T_b is the padded target length for example b; m_{bt} is one for a real target token and zero for padding; $M = \sum_{b,t} m_{bt}$ is the number of real tokens; and $\mathbf{x}_b$, y_{bt} and $y_{b,<t}$ denote the source, current target token and reference prefix. The logarithm is natural. The mask removes padding from the loss; it does not remove it from computation automatically.

Check the target shift carefully. The decoder input begins with a start token, while the target begins with the first output token. The end token belongs in the target so the model learns when to stop. Padding is separate from the end token: otherwise training can reward stopping in the wrong place. When batching sentences of different lengths, record whether you average over tokens or examples; those choices assign different weights to long sentences.

Data construction is often more consequential than a small architectural change. A translation pair containing a silently corrected number trains the model to change that number. Keep source versions, annotation instructions and target provenance together. Group splits by document, conversation or event so near-duplicate inputs cannot enter both training and testing.

Teacher forcing and exposure bias

Teacher forcing presents the correct prefix even when the model would have generated a different one. Suppose the reference begins `Pump A`, but generation selects `Pump B`. Later tokens can form a perfectly fluent statement about Pump B. Reference-conditioned loss still evaluates those later positions after `Pump A`; it never tests recovery from this particular generated mistake. The distinction is between prediction under a supplied history and behavior under the model's own evolving history.

Ranzato et al. named this training/inference mismatch **exposure bias**: the model is exposed only to ground-truth prefixes during training, but must consume its own predictions at test time, so errors can accumulate [10]. Two responses illustrate the design space. **Scheduled sampling** gradually replaces some ground-truth decoder inputs with the model's own predictions during training [11]. **Sequence-level training** optimizes a score of whole generated sequences, such as BLEU, rather than only next-token likelihood [10]. Neither removes the need to evaluate full generated

outputs; maximum likelihood with teacher forcing remains the standard starting point, and the factorization itself is exact. ML 09 discusses the same mismatch for image generation.

Token averaging also hides unequal semantic consequences. Replacing `the` may slightly affect style; replacing `not`, a device ID or a unit can reverse the result. Each token contributes a log-loss term, while the application may assign very different costs to these changes. A lower average loss can coexist with more critical errors if common wording improves while a rare distinction deteriorates. Several outputs can also correctly express the source: a single reference supplies one acceptable wording, not the complete set.

Online only: Bahdanau attention — Jointly Learning to Align and Translate — https://blog.shyandram.dev/en/learn/nlp/seq2seq-transformers/papers/bahdanau-attention–jointly-learning-to-align-and-translate/

3. The encoder–decoder Transformer for translation

Plain version. The encoder reads the whole source with bidirectional self-attention. The decoder writes the target with causal self-attention and reads the encoder with cross-attention. A final linear layer turns each decoder state into a distribution over target tokens.

The original Transformer [8] is assembled from the blocks of NLP 07:

1. **Inputs.** Source and target text are segmented into subwords [9] (NLP 02). Each token ID is embedded, scaled by $\sqrt{d}$ and added to a sinusoidal position encoding.
2. **Encoder.** N blocks of self-attention and feedforward sublayers, with a padding mask on source keys, produce the **memory** $M \in \mathbb{R}^{T_x \times d}$, one vector per source token.
3. **Decoder.** N blocks of masked self-attention (causal plus target padding), cross-attention to M (source padding masked), and feedforward sublayers.
4. **Output.** A linear layer and softmax give $p_\theta(y_t \mid y_{<t}, \mathbf{x})$ at every target position.

During training, all target positions are computed in one pass, because the causal mask makes each position see only its own prefix. During generation, the encoder runs once, and the decoder runs once per output token. Practical decoders cache the keys and values of earlier target positions instead of recomputing them; NLP 09 analyzes that cache.

Two training details from the original paper are worth knowing. The model was trained with **label smoothing**: the target distribution puts most of its mass on the reference token and spreads a small amount over the rest of the vocabulary [8, 29]. With smoothing parameter ε and vocabulary V, the training target is $q(v) = (1-\varepsilon)\mathbf{1}\{v = y_t\} + \varepsilon/|V|$; here $\mathbf{1}\{\cdot\}$ is one when its condition holds. This discourages extreme confidence; Vaswani et al. report that it hurt perplexity but improved BLEU [8]. The model also used a learning-rate warm-up, which Xiong et al. later linked to the post-norm layout (NLP 07).

4. Decoding: greedy search, beam search and length normalization

Plain version. A trained model gives a probability for every possible next token. A decoder must turn those probabilities into one output sentence. Greedy decoding takes the best next token each time; beam search keeps several partial sentences alive.

The sequence score for a candidate is

$$\ell(\mathbf{y} \mid \mathbf{x}) = \sum_{t=1}^{U} \log p_\theta(y_t \mid y_{<t}, \mathbf{x}).$$

Here ℓ is the log probability of the complete sequence; U includes the end token; and the remaining symbols are defined above. Exact maximization over all sequences is intractable because the number of candidates grows exponentially with length. **Greedy decoding** chooses the most probable token at each step. **Beam search** with width k keeps the k highest-scoring prefixes, extends each by every token, and again keeps the best k; finished hypotheses (ending in the end token) are set aside. These choices control search and variation; none makes unsupported content reliable.

Greedy decoding can miss the best complete sequence: a first token with slightly lower probability may lead to a much more probable continuation, which a beam keeps alive long enough to discover. NLP 09 works through a constructed example of this, the bound that raw log probabilities give on unfinished prefixes, the difference between search error and model error, and sampling-based decoding. Here we focus on what translation adds: length normalization, and a measured greedy-versus-beam comparison on a trained translator (Section 6).

Length normalization

Raw log probability has a built-in preference. Every additional token contributes a nonpositive term, so longer hypotheses tend to score lower even when they are better translations. Wu et al. describe this bias in Google's neural MT system and rank finished hypotheses by a normalized score [12]:

$$s(\mathbf{y}, \mathbf{x}) = \frac{\ell(\mathbf{y} \mid \mathbf{x})}{\mathrm{lp}(\mathbf{y})}, \qquad \mathrm{lp}(\mathbf{y}) = \frac{(5 + |\mathbf{y}|)^\alpha}{(5+1)^\alpha}.$$

Here $|\mathbf{y}|$ is the hypothesis length in tokens; $\alpha \geq 0$ controls the strength of normalization, with $\alpha = 0$ recovering the raw score and $\alpha = 1$ approximating a per-token average; and s is the score used to rank finished hypotheses. Their full scoring function also adds a coverage penalty based on attention, which we omit. They tuned α on development data [12]; treat it as a hyperparameter, not a constant.

```
candidates = {                        # per-token log-probabilities, end token included
    "stop <eos>": [-0.9, -1.1],
    "stop pump b now <eos>": [-0.9, -0.5, -0.3, -0.4, -0.2],
}
def length_penalty(n, alpha):         # Wu et al. (2016): ((5 + n) / 6) ** alpha
    return ((5 + n) / 6) ** alpha
```

```
for alpha in [0.0, 0.6, 1.0]:
    scores = {c: sum(lp) / length_penalty(len(lp), alpha) for c, lp in candidates.items()}
    best = max(scores, key=scores.get)
    print(f"alpha={alpha}:", {c: round(s, 3) for c, s in scores.items()}, "->", best)
```

Output:

```
alpha=0.0: {'stop <eos>': -2.0, 'stop pump b now <eos>': -2.3} -> stop <eos>
alpha=0.6: {'stop <eos>': -1.823, 'stop pump b now <eos>': -1.693} -> stop pump b now <eos>
alpha=1.0: {'stop <eos>': -1.714, 'stop pump b now <eos>': -1.38} -> stop pump b now <eos>
```

The raw score prefers the truncated instruction; $\alpha = 0.6$ and $\alpha = 1$ prefer the complete one (a small α such as 0.2 would still prefer the short one). Whether that is right depends on the source, which the score does not see.

Normalization also changes the search itself: with raw scores, a finished hypothesis that beats every unfinished prefix can be returned safely, because extending a prefix can only lower its raw score; with a length-normalized score that argument no longer holds, so the stopping rule must account for the penalty.

Set explicit output limits, a stopping rule and behavior when the limit is reached. A truncated translation should not be returned as complete. Record the decoding parameters with the model so an evaluation can be reproduced.

5. Evaluating translations: BLEU, chrF and COMET

Plain version. Automatic metrics compare a system output with one or more human reference translations. They are cheap and reproducible, which makes them useful for comparing systems, but a tiny word change can reverse meaning without changing the score much.

BLEU

BLEU compares n-grams of the candidate with those of the references [13]. For each order $n = 1, \dots, 4$ it computes a **modified (clipped) precision**: each candidate n-gram is counted at most as many times as it appears in the reference where it is most frequent. Over a test corpus,

$$p_n = \frac{\sum_{\text{sent}} \sum_{g \in G_n} \min\left(c(g),\, c_{\text{ref}}^{\max}(g)\right)}{\sum_{\text{sent}} \sum_{g \in G_n} c(g)}, \qquad \text{BP} = \begin{cases} 1 & c > r \\ e^{1 - r/c} & c \le r \end{cases}, \qquad \text{BLEU} = \text{BP} \cdot \exp\Big(\sum_{n=1}^{4} w_n \log p_n \Big).$$

Here G_n is the set of distinct n-grams in a candidate sentence; $c(g)$ is the count of n-gram g in the candidate; $c_{\text{ref}}^{\max}(g)$ is its largest count in any single reference for that sentence; c is the total candidate length over the corpus; r is the **effective reference length**, the sum over sentences of the reference length closest to each candidate's length; BP is the **brevity penalty**, which stops a system from gaining precision by writing very short outputs; and $w_n = 1/4$ are uniform weights.

BLEU is a corpus-level score: the precisions are pooled over all sentences before the logarithm, so it is not the average of sentence scores.

```
import math
from collections import Counter

def ngrams(tokens, n):
    return Counter(tuple(tokens[i:i + n]) for i in range(len(tokens) - n + 1))

def corpus_bleu(hypotheses, references, max_n=4):
    """hypotheses: list of token lists; references: list of lists of token lists."""
    matched, total = [0] * max_n, [0] * max_n
    hyp_len = ref_len = 0
    for hyp, refs in zip(hypotheses, references):
        hyp_len += len(hyp)
        # effective reference length: the reference length closest to the hypothesis
        ref_len += min((abs(len(r) - len(hyp)), len(r)) for r in refs)[1]
        for n in range(1, max_n + 1):
            counts = ngrams(hyp, n)
            max_ref = Counter()
            for r in refs:
                max_ref |= ngrams(r, n)            # elementwise max over references
            matched[n - 1] += sum((counts & max_ref).values())   # clipped counts
            total[n - 1] += max(len(hyp) - n + 1, 0)
    if min(matched) == 0:
        return 0.0, matched, total
    log_p = sum(math.log(m / t) for m, t in zip(matched, total)) / max_n
    bp = 1.0 if hyp_len > ref_len else math.exp(1 - ref_len / hyp_len)
    return bp * math.exp(log_p), matched, total

refs = [["the cat is on the mat".split(), "there is a cat on the mat".split()]]
_, m, t = corpus_bleu(["the the the the the the the".split()], refs, max_n=1)
print("clipped unigram precision:", f"{m[0]}/{t[0]}")
ref = "pump a is not operating at 30 c".split()
for hyp in ["pump a is not operating at 30 c",
            "pump a is operating at 30 c",
            "pump a is not running at 30 c"]:
    score, _, _ = corpus_bleu([hyp.split()], [[ref]])
    print(f"{score:.3f}  {hyp}")
assert m[0] == 2 and t[0] == 7
assert corpus_bleu([ref], [[ref]])[0] == 1.0
```

Output:

```
clipped unigram precision: 2/7
1.000  pump a is not operating at 30 c
0.515  pump a is operating at 30 c
0.500  pump a is not running at 30 c
```

The first check reproduces the classic clipping example: a candidate that repeats “the” seven times gets credit for only two, because no reference contains “the” more than twice [13]. The

second check is the warning. Dropping `not` reverses the meaning, yet scores higher than a correct paraphrase that replaces `operating` with `running`. BLEU measures surface overlap with the references; it has no notion of which words carry the proposition.

This implementation uses whitespace tokens, no smoothing (any zero n-gram precision gives zero), and the closest-reference-length rule. Published BLEU scores depend on such choices. Post showed that differences in tokenization and normalization alone can change reported BLEU substantially, and introduced sacreBLEU, which computes the score on detokenized text with a standard internal tokenizer and prints a version signature [14]. Report that signature with any BLEU number; scores computed with different settings are not comparable.

chrF

chrF scores **character** n-grams instead of word n-grams [15]:

$$\mathrm{chrF}_\beta = (1 + \beta^2)\,\frac{\mathrm{chrP} \cdot \mathrm{chrR}}{\beta^2\,\mathrm{chrP} + \mathrm{chrR}}.$$

Here chrP and chrR are character n-gram precision and recall, averaged over n-gram orders; and β weights recall β times as much as precision. Popović found 6-grams worked best in her experiments and compared $\beta = 1$ with $\beta = 3$ [15]. Character n-grams give partial credit for shared stems and inflections, which helps for morphologically rich target languages, but chrF is still an overlap score: it cannot tell that `not` matters more than `the`.

Learned metrics: COMET

COMET trains a neural model to predict human quality judgments from the source, the candidate and a reference, using a cross-lingual pretrained encoder [16]. It can reward a correct paraphrase that shares few words with the reference. In the WMT22 metrics task, evaluated against expert MQM (Multidimensional Quality Metrics) annotations, neural learned metrics correlated better with human ratings than overlap metrics such as BLEU and chrF [18]. Learned metrics bring their own model versions, training distributions and language coverage; their outputs are measurements under those conditions. Check the metric on the same kinds of negation, quantity and terminology errors that matter in your application.

A metric that ranks systems is not automatically an output gate

The WMT25 evaluation findings separate segment-quality prediction, error-span annotation and quality-informed correction; they report that system-level and segment-level evaluation can favor different approaches, and that linguistic diversity remains difficult [19]. This matters because rejecting one faulty translation is a different decision from ranking average model quality. A metric can reward a generally fluent system while missing rare reversed negations in individual outputs. The acceptance threshold needs validation on those errors, rather than being inherited from an aggregate correlation result.

Cross the generator's error categories with the evaluator's inputs. A reference-only comparison can inspect wording agreement but lacks the source needed to verify an unsupported statement. A source-aware evaluator has that evidence, yet can still share the generator's scope or quantity mistake. Inspect disagreement cases: correct paraphrases marked poor, fluent contradictions marked good, and accurate but incomplete translations. Include human review with the original source and separate adequacy from style. This analysis identifies which part of an evaluation signal is useful for model comparison and which part can responsibly inform acceptance of a particular result.

Overlap, recall and human judgment

The same counterexample affects recall-oriented overlap, which ROUGE uses for summarization [17]. For a simple unigram recall check,

$$R_1 = \frac{\sum_w \min(c_{\text{ref}}(w), c_{\text{hyp}}(w))}{\sum_w c_{\text{ref}}(w)}.$$

Here w ranges over word types; $c_{\text{ref}}(w)$ and $c_{\text{hyp}}(w)$ are reference and hypothesis word counts. The denominator is the number of reference tokens. This minimal implementation does not reproduce every preprocessing or aggregation option of a full evaluation package.

```
import re
from collections import Counter

def words(text):
    return re.findall(r"[a-z0-9]+", text.lower())
def overlap_recall(reference, hypothesis):
    ref, hyp = Counter(words(reference)), Counter(words(hypothesis))
    return sum((ref & hyp).values()) / sum(ref.values())
reference = "Pump A is operating at 30 C."
contradiction = "Pump A is not operating at 30 C."
score = overlap_recall(reference, contradiction)
print("Unigram recall:", score)
assert score == 1.0
assert ("not" in words(reference)) != ("not" in words(contradiction))
```

The contradiction receives recall 1.0 because it retains every reference token. A precision term would penalize the extra word, but the small penalty would still not express the importance of reversed negation. This is a controlled counterexample, not evidence that overlap metrics never correlate with human judgments. It explains why the metric must be paired with meaning-sensitive checks.

Human evaluation should hide system identity, randomize presentation, preserve source context and specify the decision task. Freitag et al.'s large MQM study, with professional translators who could see the full document, ranked systems substantially differently from the crowd-worker ratings of the same shared task [30]. Ask reviewers to mark specific errors before judging overall

quality, report who reviewed the outputs and in which languages, and keep an adjudicated challenge set separate from routine samples.

6. Hands-on: a tiny Transformer translator

Plain version. We invent a small language pair, train an encoder–decoder Transformer on 3,000 sentence pairs for under a minute on a CPU, and compare greedy and beam decoding with the BLEU function above.

The source language is English-like: subject, verb, object, with optional adjectives before nouns and an optional `never`. The target language is invented: every word has a fixed translation, adjectives follow their nouns, the object is followed by the particle `o`, the verb comes last, and negation becomes the suffix word `nai`. For example, `the small drone never lifts a red cable` becomes `la drono pico un kablo rosso o levi nai`. Translating therefore requires both a dictionary and reordering, the two things that made word-by-word translation inadequate. Training and test sentences are disjoint, but they come from the same small grammar, so this is an in-distribution test.

The model uses PyTorch's encoder and decoder layers (two of each, width 64, four heads), scaled embeddings plus sinusoidal positions, boolean masks (`True` means blocked), and `ignore_index` to drop padded targets from the loss, implementing the masked loss of Section 2. Run it after the BLEU block, because it calls `corpus_bleu`.

```
# Run after the BLEU block above: this code calls its corpus_bleu function.
import math, random, time
import torch
from torch import nn
from torch.nn import functional as F

torch.manual_seed(0); torch.set_num_threads(1); rng = random.Random(0)
lexicon = {"the": "la", "a": "un", "robot": "robo", "sensor": "senso",
           "pump": "pompa", "valve": "valva", "drone": "drono", "cable": "kablo",
           "red": "rosso", "blue": "blu", "big": "grando", "small": "pico",
           "sees": "vidi", "checks": "kontrolli", "moves": "movi",
           "opens": "aperi", "lifts": "levi"}
DET, NOUN = ["the", "a"], ["robot", "sensor", "pump", "valve", "drone", "cable"]
ADJ, VERB = ["red", "blue", "big", "small"], ["sees", "checks", "moves", "opens", "lifts"]

def noun_phrase():
    adj = [rng.choice(ADJ)] if rng.random() < 0.5 else []
    return rng.choice(DET), adj, rng.choice(NOUN)

def pair():          # source: SVO, adjective before noun; target: SOV, adjective after noun
    (d1, a1, n1), (d2, a2, n2) = noun_phrase(), noun_phrase()
    verb, neg = rng.choice(VERB), rng.random() < 0.3
    src = [d1, *a1, n1] + (["never"] if neg else []) + [verb, d2, *a2, n2]
    np_ = lambda d, a, n: [lexicon[d], lexicon[n]] + [lexicon[x] for x in a]
    tgt = np_(d1, a1, n1) + np_(d2, a2, n2) + ["o", lexicon[verb]] + (["nai"] if neg else [])
    return " ".join(src), " ".join(tgt)
```

```
pairs = list(dict(pair() for _ in range(6000)).items())   # unique source sentences
train_pairs, test_pairs = pairs[:3000], pairs[3000:3200]
PAD, BOS, EOS = 0, 1, 2
src_vocab = ["<pad>", "<bos>", "<eos>"] + sorted({w for s, _ in pairs for w in s.split()})
tgt_vocab = ["<pad>", "<bos>", "<eos>"] + sorted({w for _, t in pairs for w in t.split()})
s_id = {w: i for i, w in enumerate(src_vocab)}
t_id = {w: i for i, w in enumerate(tgt_vocab)}

def batch(texts, ids, L):
    rows = [[ids[w] for w in t.split()] + [EOS] for t in texts]
    return torch.tensor([r + [PAD] * (L - len(r)) for r in rows])
S = batch([s for s, _ in train_pairs], s_id, 10)
Y = batch([t for _, t in train_pairs], t_id, 12)
Y_in = torch.cat([torch.full((len(Y), 1), BOS), Y[:, :-1]], dim=1)   # shift right

def sinusoidal(n, d):
    pos = torch.arange(n)[:, None]
    omega = 1.0 / 10000 ** (torch.arange(0, d, 2) / d)
    pe = torch.zeros(n, d)
    pe[:, 0::2], pe[:, 1::2] = torch.sin(pos * omega), torch.cos(pos * omega)
    return pe

class TinyMT(nn.Module):
    def __init__(self, d=64):
        super().__init__()
        self.d = d
        self.src_emb = nn.Embedding(len(src_vocab), d)
        self.tgt_emb = nn.Embedding(len(tgt_vocab), d)
        self.register_buffer("pe", sinusoidal(32, d))
        enc = nn.TransformerEncoderLayer(d, 4, 128, dropout=0.0, batch_first=True)
        dec = nn.TransformerDecoderLayer(d, 4, 128, dropout=0.0, batch_first=True)
        self.encoder = nn.TransformerEncoder(enc, 2, enable_nested_tensor=False)
        self.decoder = nn.TransformerDecoder(dec, 2)
        self.out = nn.Linear(d, len(tgt_vocab))
    def embed(self, emb, x):
        return emb(x) * math.sqrt(self.d) + self.pe[: x.shape[1]]
    def encode(self, src):
        return self.encoder(self.embed(self.src_emb, src), src_key_padding_mask=src.eq(PAD))
    def decode(self, memory, src, y_in):
        T = y_in.shape[1]
        causal = torch.ones(T, T, dtype=torch.bool).triu(1)      # True = blocked
        h = self.decoder(self.embed(self.tgt_emb, y_in), memory, tgt_mask=causal,
                         tgt_key_padding_mask=y_in.eq(PAD),
                         memory_key_padding_mask=src.eq(PAD))
        return self.out(h)

@torch.no_grad()
def greedy(src_text, max_len=14):
    src = batch([src_text], s_id, len(src_text.split()) + 1)
    memory, y = model.encode(src), [BOS]
    for _ in range(max_len):
        logp = model.decode(memory, src, torch.tensor([y]))[0, -1].log_softmax(-1)
        y.append(int(logp.argmax()))
        if y[-1] == EOS:
            break
    return [tgt_vocab[i] for i in y[1:] if i != EOS]
```

```
@torch.no_grad()
def beam_search(src_text, width=4, alpha=0.6, max_len=14):
    src = batch([src_text], s_id, len(src_text.split()) + 1)
    memory = model.encode(src)
    beams, finished = [([BOS], 0.0)], []
    lp = lambda n: ((5 + n) / 6) ** alpha             # GNMT length penalty
    for _ in range(max_len):
        prefixes = torch.tensor([y for y, _ in beams])
        logp = model.decode(memory.expand(len(beams), -1, -1), src.expand(len(beams), -1),
                            prefixes)[:, -1].log_softmax(-1)
        cand = [(y + [t], s + logp[b, t].item()) for b, (y, s) in enumerate(beams)
                for t in logp[b].topk(width).indices.tolist()]
        cand.sort(key=lambda c: c[1], reverse=True)
        beams = []
        for y, s in cand:
            (finished if y[-1] == EOS else beams).append((y, s))
            if len(beams) == width:
                break
        if not beams:
            break
    best, _ = max(finished or beams, key=lambda c: c[1] / lp(len(c[0]) - 1))
    return [tgt_vocab[i] for i in best[1:] if i != EOS]

@torch.no_grad()
def log_prob(src_text, words):                        # model score of a full output
    src = batch([src_text], s_id, len(src_text.split()) + 1)
    y = [t_id[w] for w in words] + [EOS]
    logp = model.decode(model.encode(src), src, torch.tensor([[BOS] +
     ↪  y[:-1]])).log_softmax(-1)
    return logp[0, torch.arange(len(y)), y].sum().item()

def evaluate():
    model.eval()
    refs, outputs = [[t.split()] for _, t in test_pairs], {}
    for name, decode in [("greedy", greedy), ("beam-4", beam_search)]:
        hyps = [decode(s) for s, _ in test_pairs]
        exact = sum(h == r[0] for h, r in zip(hyps, refs)) / len(refs)
        mean_lp = sum(log_prob(s, h) for (s, _), h in zip(test_pairs, hyps)) / len(hyps)
        print(f"  {name}: BLEU {corpus_bleu(hyps, refs)[0]:.3f}  exact {exact:.3f}"
              f"  mean log p {mean_lp:.3f}")
        outputs[name] = hyps
    print("  outputs that differ:", sum(g != b for g, b in zip(*outputs.values())))
    model.train()

model = TinyMT()
opt = torch.optim.Adam(model.parameters(), lr=1e-3)
start = time.time()
for step in range(1, 401):
    idx = torch.randint(0, len(S), (64,))
    logits = model.decode(model.encode(S[idx]), S[idx], Y_in[idx])   # teacher forcing
    loss = F.cross_entropy(logits.reshape(-1, len(tgt_vocab)), Y[idx].reshape(-1),
                           ignore_index=PAD)
    opt.zero_grad(); loss.backward(); opt.step()
    if step in (100, 400):
        print(f"after {step} steps: training loss {loss.item():.3f}")
```

```
        evaluate()
model.eval()
print(" ".join(greedy("the small drone never lifts a red cable")))
print(f"total time {time.time() - start:.0f}s")
```

Output (CPU, single thread; the timing will vary by machine, and other PyTorch versions or hardware can change the last digits):

```
after 100 steps: training loss 0.221
  greedy: BLEU 0.614  exact 0.365  mean log p -1.587
  beam-4: BLEU 0.618  exact 0.375  mean log p -1.553
  outputs that differ: 31
after 400 steps: training loss 0.003
  greedy: BLEU 1.000  exact 1.000  mean log p -0.027
  beam-4: BLEU 1.000  exact 1.000  mean log p -0.027
  outputs that differ: 0
la drono pico un kablo rosso o levi nai
total time 16s
```

Read the two checkpoints separately. After 100 steps the model is half-trained: only about a third of the 200 test translations are exactly right. Here beam search changes 31 of the 200 outputs, finds outputs with a higher average model log probability, and improves BLEU and exact match slightly. After 400 steps the model is confident and correct on this in-distribution test, and greedy and beam produce identical outputs: when the model's distribution is sharply peaked, there is nothing for a wider search to find. Beam search helps by reducing **search error**; it cannot fix **model error**, and on easy inputs it may change nothing.

BLEU and exact match move together here because the language is tiny and deterministic: there is exactly one correct translation per source. With real references, a correct translation can differ from the reference, and the two numbers diverge. The test also checks only sentences from the same grammar. It says nothing about compositional generalization, such as an adjective–noun combination never seen in training; Exercise 1 asks you to test that.

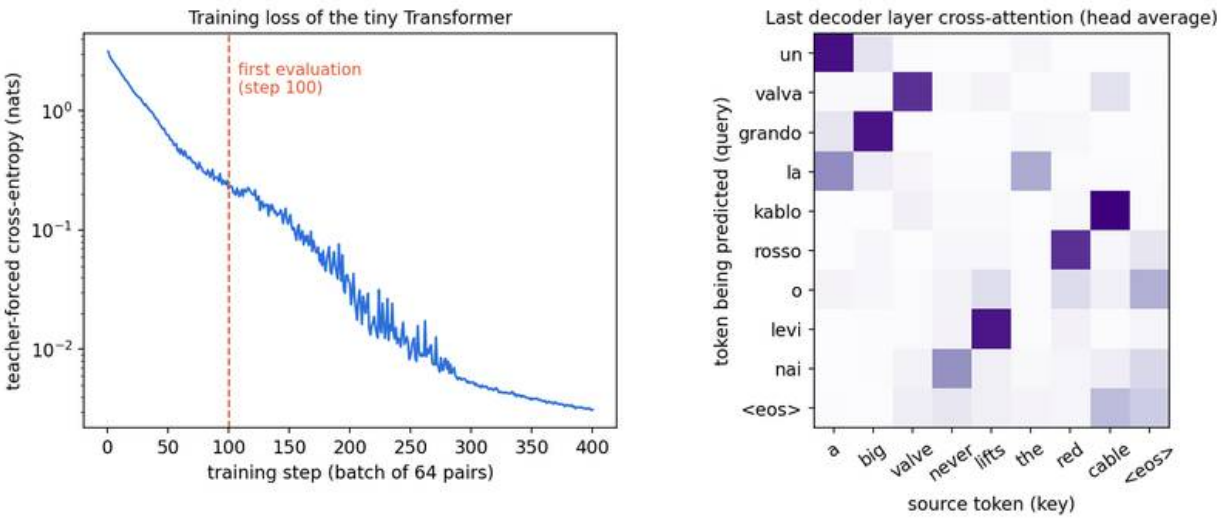


Figure 1 — The tiny translator. Left: teacher-forced training loss from the code above. Right: cross-attention of the last decoder layer, averaged over heads, for a held-out sentence it translates correctly. Each row is the decoder position that predicts the labeled word. The off-diagonal pattern is the reordering the model learned; it is a view of where the computation read, not a proof of why it is correct.

The cross-attention map in Figure 1 is the Transformer version of the soft alignments that Bahdanau et al. visualized for RNN translators [7]. Adjectives in the target read the adjectives that preceded the nouns in the source; the verb, which moves to the end, reads the source verb; and `nai` reads `never`. This alignment emerged from the translation loss alone; no alignment labels were given.

7. Translation preserves meaning across form

Plain version. A good translation changes the words and their order but keeps who did what to whom, when, how much, and whether it happened at all.

Translation is conditional generation with a strong content constraint: preserve the source meaning while producing natural target-language text. Word order, morphology, idioms and implicit context make it more than replacing each word. McRoy's translation discussion [2] pairs naturally with Hagiwara's sequence-to-sequence material [1].

Evaluate **adequacy**, whether the source meaning survives, separately from **fluency**, whether the target reads naturally. A fluent translation can omit an exception, reverse negation or change who performed an action. A literal translation may preserve technical content but sound awkward. The acceptable balance depends on the use case and must be specified before choosing a model or metric.

For a general translator, preserve terminology through a glossary and test it in context. A required term can be present while attached to the wrong entity. Likewise, keeping every number is insufficient if the values are reassigned. Build challenge cases around relations: "Pump A is

warmer than Pump B," "the valve may remain closed," and "no fault was detected before noon." Minimal edits reveal whether the model respects comparison, modality, negation and time.

Language tags and text normalization belong to the model contract. Traditional and Simplified Chinese are different output requirements; a broad language label alone does not specify script, regional terminology or punctuation. Do not convert source text destructively before storing it. Preserve the original alongside the normalized model input so reviewers can investigate a mistranslation.

In low-resource settings, uncertainty can come from limited training coverage rather than input ambiguity. Record domain, dialect, script and language pair when reporting results. A system validated on short product descriptions has not thereby been validated on long technical instructions. If reviewers cannot read both languages, they cannot directly assess adequacy; use qualified bilingual review and explain the remaining coverage gaps.

Preserve the proposition, not only its words

For a translation, compare the source and target as propositions with participants, relations and scope. "Pump A is warmer than Pump B" must keep the comparison direction. A target that contains both names and the word "warmer" can still reverse it. "No sensor exceeded the limit" differs from "Not every sensor exceeded the limit": the second allows some exceedances. NLP 12 explains why these negation scopes differ.

Ambiguity sometimes belongs to the source. `May` can express possibility or permission depending on context. If an isolated sentence does not decide between them, translating it into a form that unambiguously expresses permission adds an interpretation. Use the surrounding document or conversation when available; otherwise preserve or flag the ambiguity. A fluent target does not justify resolving an underspecified source arbitrarily.

A meaning-oriented comparison can check entity bindings, event status, quantities, units, negation and conditions individually, then assess overall readability. These checks allow lexical variation while exposing proposition changes. They also separate a true translation error from an already ambiguous source: an annotator disagreement caused by missing source context should not be interpreted as evidence that one vocabulary choice fixes the model.

8. Chatbots: conversation as sequence-to-sequence

Plain version. A chatbot can be built like a translator whose "source" is the conversation so far and whose "target" is the next reply. That works for fluent small talk, but a task assistant also needs to keep track of what has been decided.

Vinyals and Le trained a sequence-to-sequence model to predict the next turn from the previous turns, on an IT helpdesk dataset and on movie subtitles, and found it could hold simple conversations despite optimizing a likelihood objective that does not directly measure conversation quality [23]. Li et al. observed that such models tend to produce safe, generic replies such as "I don't know" regardless of the input, and proposed a maximum mutual information objective

that favors replies specific to the message [24]. The lesson carries over to modern assistants: a high-probability reply is not necessarily an informative or correct one.

For response generation, write $p_\theta(u_t \mid h_t, s_t, e_t)$, where u_t is the next system utterance, h_t is selected history, s_t is task state, e_t is evidence and θ is the model parameters. The conditional distribution says what the generator receives. It does not ensure that the generator respects every state field or evidence item.

Task-oriented dialogue has a defined outcome: supply a query, complete a booking-like record, retrieve the right observation or produce a specified report. Open conversation has less sharply defined goals and more acceptable replies. Both require consistency, but their success criteria differ. A modular task system separates understanding, state tracking, policy, evidence access and realization, which makes errors easier to locate. A learned end-to-end system can combine these functions, but it still needs tests that distinguish wrong state from poor wording. The surveys of Deriu et al. [25] and Ni et al. [26] describe the broader design and evaluation families.

A task policy can choose clarification, query, explanation or refusal of an unsupported operation. Generation then verbalizes that action. This keeps conversational fluency from deciding whether a query is ready. Open conversation benefits from flexible language, but persona or style should not override factual consistency. A system can be engaging while inventing past interactions or implying it performed an action that never occurred. Evaluate claims about the conversation against the actual log, and claims about external facts against evidence.

9. Turns, dialogue acts and state

Plain version. A conversation is a changing shared problem. The latest sentence may replace an earlier request, refer back to something, or correct the system. The system must track what has been proposed and what has actually been accepted.

A dialogue turn is a participant's contribution at a particular point in an exchange. Its function, its **dialogue act**, can be a request, an answer, a correction, a confirmation or a clarification. The same surface word can have different functions depending on the pending question: "Tuesday" may supply a date or correct an earlier date.

A task-oriented dialogue state stores the information needed to continue a specified task. For an equipment-query assistant, slots might include device, time interval, measurement and output language. Store values together with source turn, confidence and status. Unknown, explicitly absent and not applicable should not collapse into one empty string. Write a state update as

$$s_t = U(s_{t-1}, o_t, r_t), \qquad a_t = \pi(s_t).$$

Here s_t is the state after turn t; o_t is the observed utterance or extracted dialogue act; r_t identifies the speaker and other turn context; U is the update rule; a_t is the next system action; and π is the policy mapping state to actions. The state need not be a single deterministic record, and the policy need not generate text directly.

Keep proposal state separate from committed state. A user may ask for an option, inspect it and then change it. The system's own suggestion does not become a user preference merely because it

appears in the transcript. Corrections require scope: “Actually, Pump B” can replace the selected device while keeping the time interval; “start over” may discard the whole task; “not yesterday” rejects a value without necessarily specifying a replacement.

A deterministic state tracker with correction tests

A structured state tracker is a useful baseline for a narrow task. It reveals the contract between language understanding and action selection. The example consumes already extracted events; it does not claim to understand arbitrary natural language. The state stores selected slots and a revision counter. Any accepted slot change invalidates the previous confirmation, and a confirmation is valid only for the revision explicitly presented to the user. This prevents an old “yes” from authorizing a newly changed proposal.

```
from copy import deepcopy

initial = {"device": None, "period": None, "revision": 0, "confirmed": None}
def update(state, event):
    result = deepcopy(state)
    kind = event["kind"]
    if kind == "set":
        slot, value = event["slot"], event["value"]
        allowed = {"device": {"pump_a", "pump_b"}, "period": {"today", "yesterday"}}
        if slot not in allowed or value not in allowed[slot]:
            raise ValueError("Unsupported slot or value")
        if result[slot] != value:
            result[slot] = value
            result["revision"] += 1
            result["confirmed"] = None
    elif kind == "confirm":
        if event["revision"] != result["revision"]:
            raise ValueError("Confirmation refers to an old proposal")
        if any(result[k] is None for k in ["device", "period"]):
            raise ValueError("Cannot confirm incomplete state")
        result["confirmed"] = result["revision"]
    else:
        raise ValueError("Unsupported event")
    return result
s1 = update(initial, {"kind": "set", "slot": "device", "value": "pump_a"})
s2 = update(s1, {"kind": "set", "slot": "period", "value": "today"})
s3 = update(s2, {"kind": "confirm", "revision": 2})
s4 = update(s3, {"kind": "set", "slot": "device", "value": "pump_b"})
print("Before correction:", s3)
print("After correction:", s4)
assert initial["device"] is None
assert s3["confirmed"] == 2 and s4["confirmed"] is None
try:
    update(s4, {"kind": "confirm", "revision": 2})
except ValueError:
    pass
else:
    raise AssertionError("Stale confirmation was accepted")
```

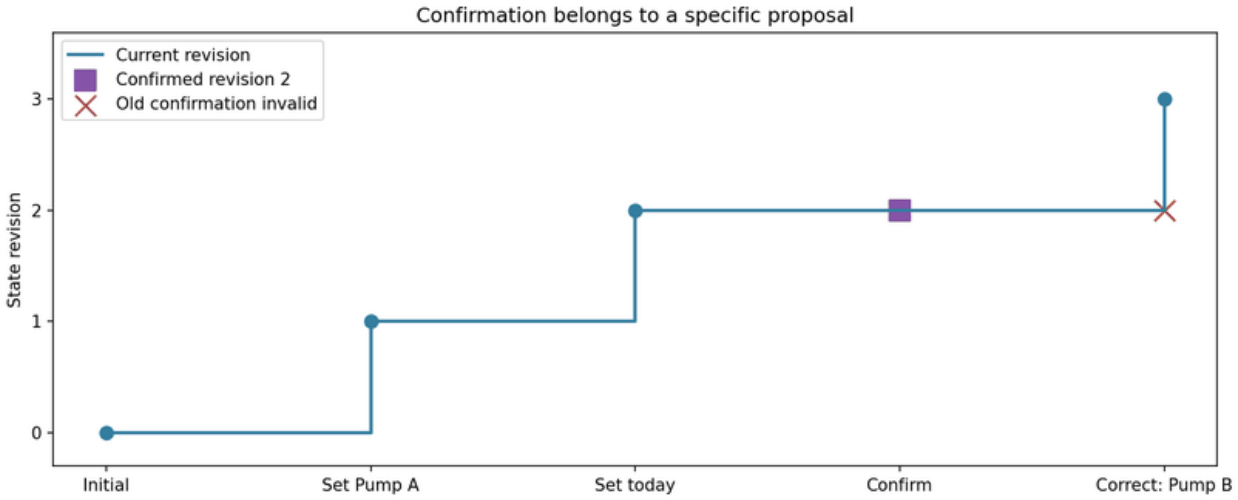


Figure 2 — Corrections change state and confirmation scope. Keeping a revision makes stale acceptance visible rather than silently applying it to a different proposal.

The copy preserves previous states for inspection and testing. A real system also needs event IDs so a repeated delivery does not apply the same update twice. Store the source turn and parsed event so an error can be traced to language understanding or state logic. Do not equate absence of a slot with a negative assertion: "no fault occurred" fills a fault-status value; an unmentioned fault leaves it unknown. The narrow allowed values make the baseline's coverage obvious. General systems can use learned slot extraction and open-vocabulary values, but should still validate units, schema and provenance. A model's confidence is not a substitute for a legal value.

10. Belief states and useful clarification

Plain version. When an utterance is ambiguous, keep several interpretations with probabilities instead of guessing. Then decide whether to act or to ask a question, depending on how costly a wrong action would be.

A simple Bayesian update is

$$b_t(h) = \frac{p(o_t \mid h) b_{t-1}(h)}{\sum_{h'} p(o_t \mid h') b_{t-1}(h')}.$$

Here h is a candidate interpretation; $b_{t-1}(h)$ is its previous probability; o_t is the new observation; $p(o_t \mid h)$ is its likelihood under the interpretation; and the denominator normalizes over candidates h'. This simplified update assumes a fixed hypothesis space and a specified observation model. A changing task needs a transition model or an explicit correction operation rather than repeatedly treating every utterance as independent evidence.

```
import numpy as np

hypotheses = ["pump_a", "pump_b", "unknown"]
```

```
prior = np.array([0.48, 0.48, 0.04])
ambiguous_likelihood = np.array([0.6, 0.5, 0.2])
posterior = prior*ambiguous_likelihood
posterior /= posterior.sum()
print("After ambiguous evidence:", dict(zip(hypotheses, posterior.round(3).tolist())))
action = "clarify" if posterior.max() < 0.9 else "query"
assert action == "clarify"
# A second synthetic observation explicitly names Pump B.
named_likelihood = np.array([0.02, 0.95, 0.03])
updated = posterior*named_likelihood
updated /= updated.sum()
print("After explicit naming:", dict(zip(hypotheses, updated.round(3).tolist())))
assert updated[1] > 0.9
assert np.isclose(updated.sum(), 1)
```

The likelihoods are constructed for the teaching example. They are not calibrated model scores or measured human behavior. In a learned system, calibration and uncertainty under distribution shift require validation. Retaining an unknown hypothesis also prevents a closed list from pretending that every real request refers to a known device.

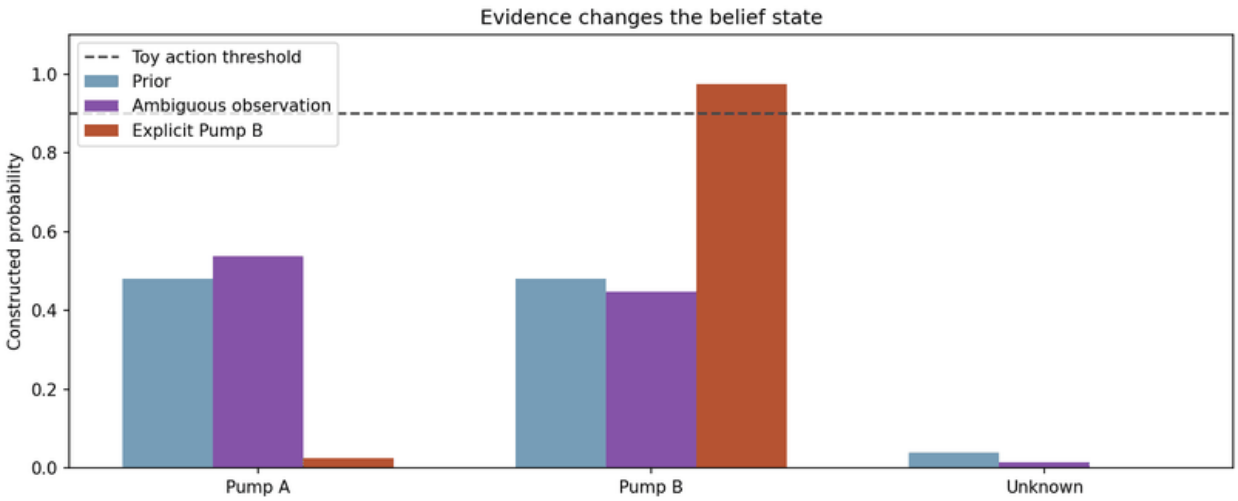


Figure 3 — A belief state preserves ambiguity until additional evidence changes the decision. The probabilities are constructed for the example and do not claim empirical calibration.

A decision-theoretic policy minimizes expected loss: $a^* = \arg\min_a \sum_h b_t(h) L(a,h)$. Here a ranges over permitted actions and $L(a,h)$ is the cost of action a when interpretation h is true. For a candidate question q, one-step expected loss is:

$$V(q) = c(q) + \sum_o p(o \mid q) \min_a \sum_h b(h \mid o,q) L(a,h).$$

Here $V(q)$ is expected loss after asking, $c(q)$ is the question's cost, o ranges over possible replies, $p(o \mid q)$ is the reply probability under the current belief, $b(h \mid o,q)$ is the posterior over interpretation h after that reply, a ranges over subsequent actions, and $L(a,h)$ is their loss. The minimum assumes the system chooses its best next action after observing the answer. Compare $V(q)$ with

the minimum expected loss of acting immediately. This finite decision calculation is a teaching model, not a calibrated description of human replies.

```
import numpy as np

prior = np.array([0.55, 0.45])  # true device A or B
reply_given_device = np.array([[0.8, 0.2], [0.2, 0.8]])
question_cost = 1.0
for wrong_cost in [10.0, 2.0]:
    immediate = wrong_cost * (1 - prior.max())
    expected_after_question = question_cost
    for reply in range(2):
        mass = prior * reply_given_device[:, reply]
        probability = mass.sum()
        posterior = mass / probability
        expected_after_question += probability * wrong_cost * (1 - posterior.max())
    action = 'clarify' if expected_after_question < immediate else 'query A'
    print(wrong_cost, round(immediate, 2), round(expected_after_question, 2), action)
assert np.allclose(reply_given_device.sum(axis=1), 1)
```

The outputs compare `4.5` with `3.0` and choose clarification at wrong-query cost `10`; they compare `0.9` with `1.4` and query A at cost `2`. The belief and question are unchanged. The consequence of an error changes the best action. A fixed confidence threshold hides this dependence, while expected loss makes it explicit. Ask the question that distinguishes the uncertainty that matters: if only the device is ambiguous, do not request the already established time interval again.

Corrections require a different distinction. “Actually, Pump B” can change the requested device, rather than provide evidence that Pump B was intended all along. The old belief concerned the earlier state; the new utterance changes the state itself. A probabilistic tracker needs state transitions as well as observation likelihoods, or an explicit update rule that recognizes corrections.

11. Context, memory and tool boundaries

Plain version. A long conversation does not fit, or does not need to fit, into every model input. What is kept must keep its status: a retracted request is not an active one. And when a chatbot can call tools, saying “I did it” must be backed by an actual result.

Partition context into recent turns, structured task state and retrieved evidence. Recent turns support local reference and tone; state stores accepted slots and unresolved questions; evidence supports factual claims. A summary should not promote an unverified model response into authoritative evidence. A useful memory record includes a value, its source, time, scope and status (proposed, accepted or retracted). A rolling window may contain “yes” while excluding the proposal it answered; a summary may retain a device but omit that it was replaced by another. Test both with synthetic conversations, and compare the reconstructed state with an explicit event log.

A tool-connected dialogue system generates proposals that software may execute. The proposal, execution and result are distinct events. A sentence saying “I queried the database” should be supported by a successful query result, not merely by the model having generated a query description.

Tool-learning research studies how foundation models plan, select tools, construct arguments and use feedback [27]. An application should expose only its permitted operations through an explicit schema, validate the selected tool and arguments before execution, then attach the result status and evidence identifier to the state.

Read-only evidence can contain imperative text. An equipment manual might quote a command, and a retrieved conversation might contain another participant's request. These strings remain source content. They should not update the current user's task state or expand the system's permitted operations. A result should distinguish success, no matching observation and execution error; converting all three into a generic fluent answer erases the information needed to understand what happened. Retrying a read-only query may be acceptable; repeating an operation that changes external state can produce a second effect, so record request identifiers and execution status in software rather than relying on the language model to remember.

12. Evaluate a conversation, not just a reply

Plain version. A correct-looking response can hide a wrong slot, a stale memory or an action that never ran. Measure the state and the final outcome over whole conversations.

Deriu et al.'s survey [25] explains why task-oriented, conversational and question-answering systems require different evaluation methods. For a specified set of slots J, define exact state accuracy as

$$A_{\text{state}} = \frac{1}{N}\sum_{t=1}^{N} \mathbf{1}\{\hat{s}_{t,j} = s_{t,j} \text{ for every } j \in J\}.$$

Here N is the number of evaluated turns; $s_{t,j}$ is the reference value for slot j at turn t; $\hat{s}_{t,j}$ is the predicted value; and $\mathbf{1}$ is one when all selected slots match and zero otherwise. This strict measure exposes errors hidden by averaging over mostly empty or easy slots. The selected slot set and handling of unknown values must be specified.

```
reference = [
    {"device": "pump_a", "period": None},
    {"device": "pump_a", "period": "today"},
    {"device": "pump_b", "period": "today"},
]
predicted = [
    {"device": "pump_a", "period": None},
    {"device": "pump_a", "period": "today"},
    {"device": "pump_a", "period": "today"},
]
slots = ["device", "period"]
joint = sum(all(p[j] == r[j] for j in slots)
            for p, r in zip(predicted, reference))/len(reference)
slot_accuracy = sum(p[j] == r[j] for p, r in zip(predicted, reference)
                    for j in slots)/(len(reference)*len(slots))
print("Exact state accuracy:", joint)
print("Per-slot accuracy:", slot_accuracy)
```

```
assert joint == 2/3 and slot_accuracy == 5/6
# The final device is wrong despite many correct fields.
assert predicted[-1]["device"] != reference[-1]["device"]
```

Task success should check the final requested outcome, not just dialogue length or friendly style. For the example, require the correct device, time interval and evidence, with corrections honored. A short conversation that queries the wrong device is not efficient success. BLEU against a reference reply is a poor fit here for the same reason as in translation, made worse by the many acceptable replies a conversation allows.

Human evaluation should show enough history to interpret a response and specify the judgment task. A rater who sees only the last reply cannot assess whether a correction was honored. Randomize system identity, define error categories and retain disagreements. Schmidtova et al. [28] show why explicit hallucination definitions and reporting matter. Split datasets by conversation so adjacent turns are not scattered across train and test, and report uncertainty at the conversation level: turns within one conversation are dependent.

Modern view

Plain version. Translation and dialogue both moved toward large pretrained models, but the evaluation questions in this chapter became more important, not less.

Surveys. Stahlberg's review in the *Journal of Artificial Intelligence Research* (2020) traces neural MT from its roots in word and sentence embeddings and early encoder–decoders to attention and the Transformer, and closes with a survey of recent trends [22]. It is the best single map for the material in Sections 1–4. For dialogue, Ni et al.'s systematic survey (*Artificial Intelligence Review*, 2023) organizes deep-learning dialogue systems by model type and system type, and its task-oriented versus open-domain distinction explains why one response metric cannot characterize every conversation [26]. Deriu et al. [25] map evaluation methods to those system types. Qin et al.'s review of tool learning with foundation models (*ACM Computing Surveys*, 2024) motivates inspecting argument construction and result use alongside text quality [27].

State of the art, 2023–2026. WMT has documented the arrival of large language models in translation: the WMT24 general-task findings evaluate several LLMs alongside submitted MT systems and online providers with professional human annotation, and the title states the balance, "the LLM era is here but MT is not solved yet" [20]. Massively multilingual models such as NLLB extend coverage to 200 languages, with low-resource languages as the explicit target [21]. On evaluation, the WMT22 metrics task found neural metrics more reliable than overlap metrics against expert MQM judgments [18], and the WMT25 findings separate system ranking from segment-level error detection [19].

Design axis	Choices to compare	Evidence needed
Architecture	encoder–decoder, decoder-only, multilingual	held-out quality per language pair and domain

Design axis	Choices to compare	Evidence needed
Decoding	greedy, beam with length normalization, sampling	completeness, length and semantic errors, not only model score
Evaluation	BLEU/chrF (with signature), learned metrics, MQM-style human review	agreement with the application's failure categories
Dialogue state	end-to-end generation, explicit state and policy	state accuracy, correction recovery, final task success
Tool use	free-form claims, validated proposals with results	executed outcomes matching reported outcomes

Open problems. Low-resource and dialectal translation, document-level consistency, evaluating metrics on rare but critical errors (negation, numbers, names), reliable automatic acceptance of individual outputs, calibration of dialogue state under task changes, multilingual corrections, and stale memories in long conversations. A higher corpus score and a fluent reply are both easy to obtain; preserved meaning and a correct final action are what need evidence.

Key takeaways

- SMT combined separately estimated translation and language models; NMT trains one encoder–decoder end to end, and the Transformer made it fully attention-based.
- Shift targets, keep the end token, mask padding in both attention and loss; teacher forcing trains on gold prefixes, so test the model on its own generated outputs (exposure bias).
- Greedy and beam search optimize the model's score with different amounts of search; length normalization changes which hypothesis wins; neither repairs a wrong model.
- BLEU is clipped corpus-level n-gram precision with a brevity penalty; report the sacreBLEU signature, add chrF or learned metrics, and check meaning-critical errors directly.
- A tiny Transformer learns dictionary translation and reordering in seconds, and its cross-attention recovers the alignment without supervision.
- Chatbots can be seq2seq models, but task dialogue needs explicit state with revisions, belief over ambiguous interpretations, and clarification chosen by expected loss.
- Keep tool proposals, executions and results separate, and evaluate whole conversations by state and final outcome.

Exercises

1. **Compositional test.** In the translator code, remove every training pair in which `red` modifies `c able`, then test on sentences that contain `red cable`. Compare exact match on these sentences with the ordinary test set.

   Hint

   Filter `train_pairs` before building `S` and `Y`, and keep the vocabulary unchanged so the failure is about combination rather than unknown words. Look at the cross-attention map of a failure.

2. **Beam width and length.** Evaluate the 100-step checkpoint with beam widths 1, 2, 4 and 8 and with $\alpha \in \{0, 0.6, 1\}$. Report BLEU, exact match, mean output length and mean log probability. When does a higher model score fail to improve BLEU?

   Hint

   Width 1 with $\alpha = 0$ is greedy decoding. Save the outputs and compare the sentences that change, not only the averages.

3. **Translation challenge set.** Write six original source sentences testing negation, comparison, units, modality, time and ambiguous reference. Specify the meaning that any acceptable translation must retain.

   Hint

   Use a minimal edit such as adding "not" or swapping the two device names. Judge source adequacy independently of whether the target sentence sounds elegant.

4. **Metric disagreement.** Construct two candidate translations: one with high word overlap but a wrong quantity, and another with lower overlap but correct meaning. Score them with the BLEU function and design an annotation form that reveals the difference.

   Hint

   Ask reviewers to identify unsupported quantities and missing required facts before giving an overall judgment. Keep the source visible and system identity hidden.

5. **Correction scope.** Extend the state tracker with a measurement slot. Test a device-only correction, a rejected period and a complete restart without silently changing unrelated slots.

   Hint

   Define distinct events for setting, retracting and resetting. Any changed proposal should invalidate a confirmation that refers to its previous revision.

6. **False action report.** Specify an evidence schema that distinguishes a successful query, no matching record and execution failure. Write a response requirement for each state, and a test conversation for each.

   Hint

   Attach an execution identifier and status to the result. A generated proposal alone cannot support a sentence claiming that the query succeeded.

References

1. M. Hagiwara, *Real-World Natural Language Processing*, Manning, 2021, Chapters 6 and 8. Book ·Sequence-to-sequence chapter
2. S. McRoy, *Principles of Natural Language Processing*, open textbook, 2021, Chapter 9. Book · Generation chapter
3. P. F. Brown, S. A. Della Pietra, V. J. Della Pietra and R. L. Mercer, "The Mathematics of Statistical Machine Translation: Parameter Estimation," *Computational Linguistics* 19(2), 1993. Paper
4. P. Koehn, F. J. Och and D. Marcu, "Statistical Phrase-Based Translation," HLT-NAACL, 2003. Paper
5. K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk and Y. Bengio, "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation," EMNLP, 2014. Paper
6. I. Sutskever, O. Vinyals and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," NeurIPS, 2014. Paper
7. D. Bahdanau, K. Cho and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," ICLR, 2015. Paper
8. A. Vaswani et al., "Attention Is All You Need," NeurIPS, 2017. Paper
9. R. Sennrich, B. Haddow and A. Birch, "Neural Machine Translation of Rare Words with Subword Units," ACL, 2016. Paper
10. M. Ranzato, S. Chopra, M. Auli and W. Zaremba, "Sequence Level Training with Recurrent Neural Networks," ICLR, 2016. Paper
11. S. Bengio, O. Vinyals, N. Jaitly and N. Shazeer, "Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks," NeurIPS, 2015. Paper
12. Y. Wu et al., "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation," arXiv:1609.08144, 2016. Paper
13. K. Papineni, S. Roukos, T. Ward and W.-J. Zhu, "Bleu: a Method for Automatic Evaluation of Machine Translation," ACL, 2002. Paper
14. M. Post, "A Call for Clarity in Reporting BLEU Scores," WMT, 2018. Paper
15. M. Popović, "chrF: character n-gram F-score for automatic MT evaluation," WMT, 2015. Paper
16. R. Rei, C. Stewart, A. C. Farinha and A. Lavie, "COMET: A Neural Framework for MT Evaluation," EMNLP, 2020. Paper
17. C.-Y. Lin, "ROUGE: A Package for Automatic Evaluation of Summaries," Text Summarization Branches Out, 2004. Paper
18. M. Freitag et al., "Results of WMT22 Metrics Shared Task: Stop Using BLEU – Neural Metrics Are Better and More Robust," WMT, 2022. Report
19. A. Lavie et al., "Findings of the WMT25 Shared Task on Automated Translation Evaluation Systems: Linguistic Diversity is Challenging and References Still Help," WMT, 2025. Report
20. T. Kocmi et al., "Findings of the WMT24 General Machine Translation Shared Task: The LLM Era Is Here but MT Is Not Solved Yet," WMT, 2024. Report
21. NLLB Team et al., "Scaling neural machine translation to 200 languages," *Nature* 630, 2024. DOI
22. F. Stahlberg, "Neural Machine Translation: A Review," *Journal of Artificial Intelligence Research* 69, 2020. DOI
23. O. Vinyals and Q. V. Le, "A Neural Conversational Model," ICML Deep Learning Workshop,

2015. Paper
24. J. Li, M. Galley, C. Brockett, J. Gao and B. Dolan, “A Diversity-Promoting Objective Function for Neural Conversation Models,” NAACL-HLT, 2016. Paper
25. J. Deriu, Á. Rodrigo, A. Otegi, G. Echegoyen, S. Rosset, E. Agirre and M. Cieliebak, “Survey on Evaluation Methods for Dialogue Systems,” *Artificial Intelligence Review* 54, 2021. Survey
26. J. Ni, T. Young, V. Pandelea, F. Xue and E. Cambria, “Recent Advances in Deep Learning Based Dialogue Systems: A Systematic Survey,” *Artificial Intelligence Review* 56, 2023. Survey
27. Y. Qin et al., “Tool Learning with Foundation Models,” *ACM Computing Surveys* 57, 2024. Author manuscript ·Publication
28. P. Schmidtova et al., “Do My Eyes Deceive Me? A Survey of Human Evaluations of Hallucinations in NLG,” INLG, 2025. Survey
29. C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens and Z. Wojna, “Rethinking the Inception Architecture for Computer Vision,” CVPR, 2016. Paper
30. M. Freitag, G. Foster, D. Grangier, V. Ratnakar, Q. Tan and W. Macherey, “Experts, Errors, and Context: A Large-Scale Study of Human Evaluation for Machine Translation,” *Transactions of the Association for Computational Linguistics* 9, 2021. DOI

NLP 09 • Language Modeling with Transformers (Natural Language Generation)

Train a tiny decoder-only Transformer language model, decode it with greedy, beam, temperature, top-k and nucleus search, and evaluate summaries, data-to-text and grounded generation.

Online edition: https://blog.shyandram.dev/en/learn/nlp/language-modeling-transformers/

What you'll learn

- Factor a sequence probability into causal next-token predictions, and train a tiny decoder-only Transformer that obeys that factorization.
- Place the GPT line of models, their training objective and scaling studies in one consistent picture.
- Decode with greedy search, beam search, temperature, top-k and nucleus sampling, and explain what each does and does not change.
- Build and evaluate three generation tasks: extractive and abstractive summarization, data-to-text and open-ended story writing.
- Measure generation with perplexity, ROUGE, human review and model-based judges, knowing the blind spots of each.
- Retrieve evidence with lexical and dense methods and evaluate the ranking with Recall@k, MRR and nDCG.
- Explain hallucination, retrieval-augmented generation and the boundary between retrieved evidence and application instructions.

The big picture

A language model assigns probabilities to text under a specified context. That simple definition connects spelling completion, sentence generation and reusable contextual representations, but these uses do not share the same output contract. Predicting a likely continuation is different from verifying a factual answer. A fluent continuation may reflect common text patterns while being unsuitable for a particular task.

A language application must also choose what to say, express it clearly, and support what it says. These are separate responsibilities. A readable sentence can describe the wrong event; a correct sentence can omit the warning that the user needed. Deploying the generator adds a further responsibility: the result must arrive predictably, with enough recorded context to investigate a failure.

Explain like I'm five

> Imagine a guessing game where you see the beginning of a sentence and guess the next word. Then you say your guess out loud, add it to the sentence, and guess again. That is how a language model writes. It can look only at the words already said; peeking at the end of the sentence would be cheating. And a guess that sounds right is not the same as a guess that is true, so a good writer also checks the facts.

Hagiwara develops language modeling in Chapter 5, attention and Transformers in Chapter 8, pretrained-model adaptation in Chapter 9, and training practice and deployment in Chapters 10–11 [1]. McRoy's Chapter 6 emphasizes language-model evaluation through tasks, and Chapter 9 develops generation and summarization [2]. This chapter connects the probability model to those practical concerns. We train only tiny synthetic models; large pretrained systems are discussed through verified primary papers and surveys.

Three decisions organize the subject. The **training objective** determines what is predicted. The **attention pattern** determines which input positions are visible. The **decoding procedure** determines how predictions become user-facing output. Keep these separate when comparing models. A change in sampling temperature changes the output distribution; it does not change which facts were learned or verify the resulting claim.

Where this chapter sits: count-based n-gram language models, smoothing and the first perplexity calculations are in NLP 05; attention, positions and the Transformer block are in NLP 07; encoder–decoder translation and dialogue are in NLP 08. Here the Transformer is used as a **decoder-only** language model and as a text generator. The bidirectional alternative, BERT, follows in NLP 10.

1. Causal sequence probability

Plain version. A sequence probability can be built by predicting each token from earlier tokens. The complete sequence is available during training, but each prediction must still obey that visibility rule.

Using the probability chain rule:

$$p_\theta(x_1, \dots, x_T) = \prod_{t=1}^{T} p_\theta(x_t \mid x_{<t}).$$

Here x_t is the token at position t; T is the sequence length; $x_{<t}$ is the prefix before t; and θ denotes model parameters. A beginning token can supply an explicit context for the first ordinary token, and an end token lets the model assign probability to stopping. The chain rule is exact; the learned conditional distributions are approximations to the data-generating process.

For a corpus with N supervised token positions, the mean negative log likelihood is:

$$\mathcal{L}_{\text{causal}} = -\frac{1}{N} \sum_{n=1}^{N} \log p_\theta(x_n \mid c_n).$$

Here $\mathcal{L}_{\text{causal}}$ is the training objective; n indexes valid prediction positions across examples; x_n is the target token; c_n is its allowed prefix; and N excludes padding and any other positions intentionally

left unsupervised. Natural logarithms express this loss in nats per token. A batch average that includes padding changes the objective and can reward predicting artificial padding symbols.

An n-gram model, as in NLP 05, replaces $x_{<t}$ with its last $n-1$ tokens, so two prefixes ending the same way must receive identical predictions. A Transformer language model keeps the whole permitted prefix, up to its context limit, and lets learned attention decide which earlier tokens matter. The factorization is the same; what changes is how much of the prefix the conditional can read.

Teacher forcing uses the observed prefix when predicting each training token. It permits parallel evaluation of all positions when the architecture enforces a causal mask. At generation time, the prefix contains earlier generated tokens instead. Errors can then affect later predictions. The training objective and rollout behavior therefore require separate diagnostics, even when the same model weights are used.

Shift the input and target by one position. If the full sequence is beginning, the, robot, moves, end, the model input is beginning, the, robot, moves and the targets are the, robot, moves, end. Predicting each input token from its own unshifted representation would create a shortcut unrelated to next-token prediction. The code in Section 3 makes the shift explicit.

Why teacher-forced loss misses some rollout errors

Teacher forcing presents the correct prefix even when the model would have generated a different one [16]. Suppose the reference begins `Pump A`, but generation selects `Pump B`. Later tokens can form a perfectly fluent statement about Pump B. Teacher-forced loss still evaluates those later positions after `Pump A`; it never tests recovery from this particular generated mistake. The distinction is between prediction under a supplied history and behavior under the model's own evolving history.

Token averaging also hides unequal semantic consequences. Replacing `the` may slightly affect style; replacing `not`, a device ID or a unit can reverse the result. Each token contributes a log-loss term, while the application may assign very different costs to these changes. A lower average loss can coexist with more critical field errors if common wording improves while a rare distinction deteriorates. Check the whole output's propositions in addition to the average training objective.

2. Decoder-only Transformer language models

Plain version. A decoder-only Transformer is a stack of self-attention blocks in which every position may read only itself and earlier positions. Trained on next-token prediction, it becomes both a scorer of text and a generator of text.

Each block computes masked self-attention followed by a position-wise feedforward network, with residual connections and normalization, as in the original Transformer [3]; NLP 07 derives the attention equation. The only change from a bidirectional encoder is an additive mask B inside the attention softmax: $B_{ij} = 0$ when $j \le i$ and $B_{ij} = -\infty$ otherwise. Here i is the querying position

and j is the key position it may read. With shifted targets, the current input position is an allowed prefix token, so the diagonal remains visible.

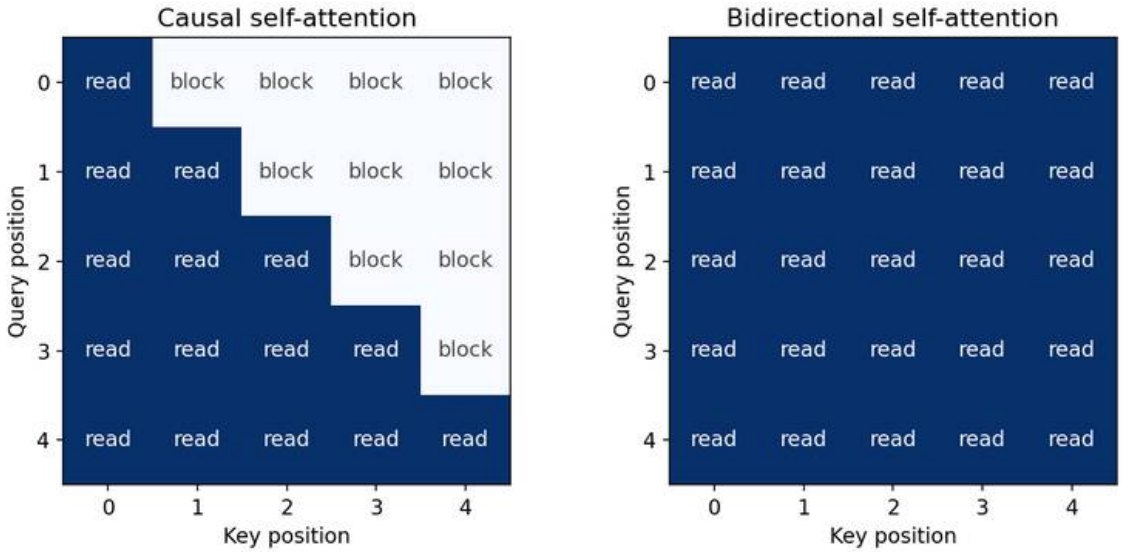


Figure 1 — Visibility is part of the task definition. Rows are querying positions and columns are the positions they may read. A decoder-only language model uses the left pattern in every layer.

A final linear layer maps the top-layer vector at position t to vocabulary logits $\mathbf{z}_t$, and a softmax turns them into $p_\theta(x_{t+1} \mid x_{\le t})$. Because every row of the mask hides the future, one forward pass computes the training loss at all positions at once, while generation still produces one token per pass.

The GPT line

The GPT papers show how far this one design can be pushed. Radford and colleagues trained a 12-layer decoder-only Transformer with masked self-attention as a language model on BooksCorpus, then fine-tuned it with a task-specific output layer for classification, entailment and similarity tasks [4]. The recipe was **generative pretraining followed by discriminative fine-tuning**: unlabeled text supplies most of the learning signal, and a small labeled set adapts the model.

GPT-2 kept the next-token objective and dropped task-specific fine-tuning from its main experiments. Its authors trained on WebText, a new dataset of millions of web pages, used byte-level byte-pair encoding so that any string could be represented, and reported that the language model began to perform tasks such as question answering and summarization in a zero-shot setting; the largest model had 1.5 billion parameters [5]. GPT-3 scaled the same approach to an autoregressive model with 175 billion parameters and evaluated it with tasks and a few demonstrations given purely as text, without gradient updates [6]. These are three points on one line: the architecture and objective barely change, while data, model size and the way tasks are presented do.

Reading the line correctly matters for later sections. A prompt is just a prefix $x_{\le t}$. "Zero-shot" and

"few-shot" describe what that prefix contains, not a different model. The quality of a continuation still depends on what the training text made probable, which is why instruction tuning, preference learning and retrieval (Sections 9–11) were added on top.

3. A runnable causal model

Plain version. A tiny model can learn a synthetic language pattern and still generalize poorly to unseen combinations. We also test a property that low loss alone cannot establish: future tokens must not influence earlier predictions.

The following standalone CPU example uses invented fixed-length sentences. Train and test sentences differ, while every ordinary vocabulary item is present in training. The code uses no padding, pretrained weights or dataset downloads. It is intentionally a mechanism experiment rather than a language benchmark.

The implementation uses AdamW for optimization [9], the GELU activation in the feedforward block [10], and layer normalization before attention and the feedforward transformation [11]. These choices define this small experiment, rather than reproduce every detail of the original Transformer. Keep them fixed when studying the causal mask so a change in optimization or activation does not confound the comparison.

```
import math
import torch
from torch import nn
from torch.nn import functional as F

torch.manual_seed(11)
torch.set_num_threads(1)
train_text = [
    'the red robot moves', 'the blue robot waits',
    'a red sensor waits', 'a blue sensor moves',
    'the red sensor moves', 'a blue robot waits',
]
test_text = ['a red robot moves', 'the blue sensor waits']
vocab = ['<bos>', '<eos>'] + sorted(set(' '.join(train_text).split()))
index = {word: i for i, word in enumerate(vocab)}
def encode(rows):
    return torch.tensor([[index[w] for w in ['<bos>'] + s.split() + ['<eos>']] for s in
     ↪  rows])
train, test = encode(train_text), encode(test_text)

class TinyCausalModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.token = nn.Embedding(len(vocab), 16)
        self.position = nn.Embedding(5, 16)
        self.norm1, self.norm2 = nn.LayerNorm(16), nn.LayerNorm(16)
        self.q, self.k, self.v = [nn.Linear(16, 16, bias=False) for _ in range(3)]
        self.output = nn.Linear(16, 16)
        self.ff = nn.Sequential(nn.Linear(16, 32), nn.GELU(), nn.Linear(32, 16))
```

```
        self.head = nn.Linear(16, len(vocab))
    def forward(self, ids):
        length = ids.shape[1]
        h = self.token(ids) + self.position(torch.arange(length))
        z = self.norm1(h)
        scores = self.q(z) @ self.k(z).transpose(-1, -2) / math.sqrt(16)
        blocked = torch.ones(length, length, dtype=torch.bool).triu(1)
        weights = scores.masked_fill(blocked, float('-inf')).softmax(dim=-1)
        h = h + self.output(weights @ self.v(z))
        h = h + self.ff(self.norm2(h))
        return self.head(h)

model = TinyCausalModel()
optimizer = torch.optim.AdamW(model.parameters(), lr=0.01)
def loss(rows):
    logits = model(rows[:, :-1])
    return F.cross_entropy(logits.reshape(-1, len(vocab)), rows[:, 1:].reshape(-1))
initial = loss(train).item()
for _ in range(180):
    optimizer.zero_grad()
    objective = loss(train)
    objective.backward()
    optimizer.step()
model.eval()
with torch.no_grad():
    train_nll, test_nll = loss(train).item(), loss(test).item()
    original = train[:1, :-1]
    changed = original.clone()
    changed[0, -1] = index['blue']
    gap = (model(original)[:, :-1] - model(changed)[:, :-1]).abs().max().item()
assert train_nll < initial and gap < 1e-6
print('initial/train/test NLL:', *(round(x, 3) for x in [initial, train_nll, test_nll]))
print('held-out perplexity:', round(math.exp(test_nll), 3))
print('future-token perturbation gap:', gap)
```

Output:

```
initial/train/test NLL: 2.194 0.409 3.17
held-out perplexity: 23.81
future-token perturbation gap: 0.0
```

The assertion checks that training loss decreases; it does not require held-out loss to improve by any specified amount. Here the held-out loss (3.17 nats per token) is even higher than the untrained starting loss (2.194): the held-out sentences combine familiar words in combinations excluded from training, and the model has confidently memorized the six training patterns. A large train–test gap demonstrates that learning the observed patterns does not establish compositional generalization.

The future-token perturbation test changes the final input token and compares all earlier logits. Causal attention should make that change invisible to earlier predictions, and the printed gap is

exactly zero. This test targets the exact visibility contract. Passing it does not establish every form of data isolation: duplicated documents across splits, vocabulary construction from forbidden test data or future metadata can still create other leakage.

The model has no dropout, so evaluation mode does not alter a stochastic layer here. We call it explicitly because the training/evaluation distinction is essential in realistic models. The loss function flattens batch and position dimensions while retaining vocabulary as the class dimension. If those axes are accidentally exchanged, a program can run while optimizing the wrong prediction problem.

Each sequence has the same length, and the position table covers the five model-input positions. A learned position table imposes a maximum supported position; giving a longer sequence is an invalid lookup, not an ordinary longer input. Extending to variable-length data requires separate attention and loss masks. An attention mask controls what representations may read; a loss mask controls which target positions contribute to the objective. They solve different problems and should be tested independently.

4. Training objective and scaling

Plain version. Large language models are trained with the same next-token loss as the tiny model above. What changes is the amount of text, the number of parameters and the compute spent, and those quantities trade off in measurable ways.

Kaplan and colleagues trained families of Transformer language models and reported that held-out loss falls approximately as a power law in model size, dataset size and compute [7]. In model size alone:

$$L(N) = \left(\frac{N_c}{N}\right)^{\alpha_N}.$$

Here L is held-out cross-entropy loss, N is the number of non-embedding parameters, N_c is a fitted constant and $\alpha_N > 0$ is a fitted exponent. Hoffmann and colleagues later fitted a form with an irreducible term, $L(N, D) = E + A/N^{\alpha} + B/D^{\beta}$, where D is the number of training tokens, E the loss that no model can remove, and A, B, α, β fitted constants; for a fixed compute budget they concluded that parameters and training tokens should be scaled in roughly equal proportion [8]. ML 10 derives the compute-optimal trade-off and fits a scaling law in code.

Two cautions keep scaling in its place. First, the fitted constants depend on tokenizer, data mixture and training protocol, so they do not transfer between unrelated setups. Second, scaling laws predict **loss**, not any particular task behavior. A lower next-token loss usually helps, but whether a model preserves a negation in a summary or refuses to invent a citation must still be measured directly (Section 8).

5. Perplexity and meaningful comparison

Plain version. Perplexity summarizes held-out token prediction. Its value depends on what counts as a token and which positions are evaluated.

For mean negative log likelihood $\mathcal{L}$:

$$\mathrm{PPL} = \exp(\mathcal{L}).$$

Here PPL is perplexity and exp is the natural exponential; $\mathcal{L}$ uses natural logarithms over the same evaluated token positions. Lower perplexity means that the model assigns higher average probability to observed tokens under that evaluation contract. It is not a direct measure of factual correctness, helpfulness or grammaticality. IT 01 connects this quantity to cross-entropy, and NLP 05 computes it for count models.

Compare perplexities only under compatible tokenization and input conventions. A word can be one token in one vocabulary and several in another. The mean loss per token then describes a different unit. Also record whether beginning and end markers are scored, how documents are segmented, the context length and whether evaluation windows overlap. Recounting duplicated target positions changes the weighting.

Keep evaluation documents independent of training documents and their derivatives. Splitting adjacent passages of one document across partitions can expose style, facts and nearly identical wording. For a time-dependent application, use an appropriate chronological split and retain the document version. A held-out name or phrase does not by itself prove the surrounding document is independent.

McRoy's benchmark perspective reminds us to assess task behavior in addition to sequence prediction [2]. A classifier needs category performance; an extractor needs spans and supported relations; a generator needs meaning preservation and omissions. A model with lower perplexity can fail a critical task distinction, such as changing "did not move" to "moved." Match the evaluation to the user's actual outcome.

A reproducible language-model experiment

A result is useful when someone can tell exactly which text, tokens and decisions produced it. Begin with the document split and its reason: a random split may suit independent synthetic sentences, while a support assistant might require customer-separated or chronological documents. In the toy experiment, the vocabulary comes only from training sentences, and the test combinations were chosen to contain known words. This isolates a narrow combination-generalization question.

Next record the tokenization, sequence boundaries, target shift, maximum context and loss denominator. The code scores every next-token target including the end token, and does not score the beginning token as a target. If another run excludes stopping predictions or weights each sentence equally regardless of length, its number answers a different question. Finally separate selection from reporting: choose architecture, training length and decoding behavior on develop-

ment data, and inspect the held-out set only after those choices are frozen. Save the preprocessing version, model configuration, seed, permitted input context and output settings, while keeping sensitive raw documents under appropriate access control.

6. Decoding: from a distribution to a text

Plain version. A trained model produces scores. A decoder decides whether to choose the strongest token, search over several continuations, or sample alternatives. None of these choices makes unsupported content reliable.

Temperature

For logit z_v and positive temperature τ:

$$q_\tau(v) = \frac{\exp(z_v/\tau)}{\sum_{u \in V} \exp(z_u/\tau)}.$$

Here v is a candidate token; V is the vocabulary; u indexes candidates in the denominator; z_v is the model's unnormalized score; τ is temperature; and q_τ is the resulting sampling distribution. Lower temperature concentrates mass around high-scoring tokens; higher temperature spreads it. As $\tau \to 0$, sampling approaches greedy selection, which chooses a maximum-score token. A factual mistake can be the highest-logit option, so reducing randomness does not remove it.

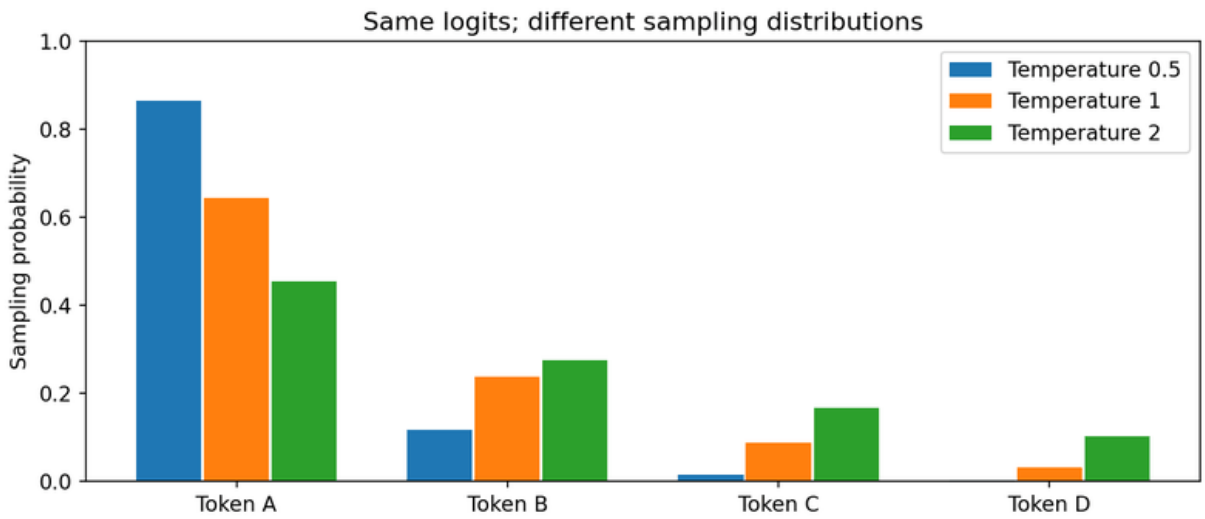


Figure 2 — Temperature changes a sampling distribution over fixed synthetic logits. More concentrated output is not automatically more correct.

Greedy and beam search

Greedy decoding chooses the most probable next token. Beam search retains several partial sequences. The sequence score for a candidate is

$$\ell(y \mid x) = \sum_{t=1}^{T} \log p_\theta(y_t \mid y_{<t}, x).$$

Here ℓ is the log probability of the complete output y given a prompt or source x; T includes the end token; and the other symbols are as above. With raw log probabilities, every additional token contributes a nonpositive term. This can favor shorter sequences. Length adjustments change the search objective; describe their form and tune them on validation data rather than silently applying them.

This tiny conditional model has two competing first tokens. It illustrates why choosing the best next token can miss the best complete sequence. The probabilities are constructed for the experiment, not estimated from language data.

```
import math

table = {
    (): {"safe": 0.55, "close": 0.45},
    ("safe",): {"stop": 0.51, "wait": 0.49},
    ("close",): {"valve": 0.99, "door": 0.01},
}
def next_probs(prefix):
    return table.get(prefix, {"<eos>": 1.0})
def decode(width):
    beams = [((), 0.0)]
    for _ in range(3):
        candidates = []
        for seq, score in beams:
            if seq and seq[-1] == "<eos>":
                candidates.append((seq, score))
                continue
            for token, prob in next_probs(seq).items():
                candidates.append((seq + (token,), score + math.log(prob)))
        beams = sorted(candidates, key=lambda pair: pair[1], reverse=True)[:width]
    return beams[0]
greedy, beam = decode(1), decode(2)
print(greedy[0], round(math.exp(greedy[1]), 4))
print(beam[0], round(math.exp(beam[1]), 4))
assert greedy[0] == ("safe", "stop", "<eos>")
assert beam[0] == ("close", "valve", "<eos>")
assert beam[1] > greedy[1]
```

Output:

```
('safe', 'stop', '<eos>') 0.2805
('close', 'valve', '<eos>') 0.4455
```

Beam search finds the more probable sequence in this constructed tree. That result is about the model's probability assignment. If "close valve" contradicts the source instruction, a better probability search still produces the wrong application behavior. Search quality and task correctness need separate measurements.

Search error, model error and a wrong objective

The toy beam succeeds because it keeps the `close` prefix long enough to discover its strong continuation. Greedy decoding discards that branch before seeing the continuation. This is **search error** relative to the constructed sequence-probability objective. A larger beam can reduce that kind of error, but it is still a bounded search: a useful prefix can fall outside the retained set at any depth.

Raw prefix log probability supplies a helpful bound. Every future log-probability term is nonpositive, so extending a prefix cannot improve its raw score. A finished candidate that exceeds the scores of all unfinished prefixes is therefore optimal among those surviving branches. This argument requires the unadjusted sum; a length bonus or normalization changes the bound and must be considered in the stopping rule.

Now distinguish **model error**. If the source requires keeping the valve open while the model assigns its greatest sequence probability to `close valve`, exact probability search exposes the model's mistake more consistently. Search cannot create source compatibility that the score lacks. Finally, maximum probability is only one decision objective. A generic short response may have high probability while omitting the required warning. Evaluate whether the selected objective rewards the user's intended result before tuning beam width.

Top-k and nucleus sampling

For open-ended writing, maximizing probability is often the wrong goal. Holtzman and colleagues observed that using likelihood as a decoding objective produces text that is bland and strangely repetitive, even from a strong model, while pure sampling from the full distribution can pick very unlikely tokens from the long tail [12]. Two truncation rules sit between these extremes.

Top-k sampling keeps the k highest-probability tokens and renormalizes. Fan and colleagues used it for story generation, sampling from the 10 most likely candidates at each step and finding it more effective than beam search for that task [13]. **Nucleus (top-p) sampling** sorts tokens by descending probability and keeps the shortest prefix whose cumulative probability reaches p [12]:

$$\pi_1 \geq \pi_2 \geq \cdots \geq \pi_{|V|}, \qquad \pi_j = p_\theta(v_{(j)} \mid x_{\leq t}),$$
$$k^* = \min\left\{k \in \{1, \ldots, |V|\} : \sum_{j=1}^{k} \pi_j \geq p\right\}, \qquad V^{(p)} = \{v_{(1)}, \ldots, v_{(k^*)}\},$$
$$q(v) = \frac{p_\theta(v \mid x_{\leq t})\, \mathbf{1}[v \in V^{(p)}]}{\sum_{u \in V^{(p)}} p_\theta(u \mid x_{\leq t})}.$$

Here $v_{(j)}$ is the token at rank j, π_j its probability, $|V|$ the vocabulary size, k^* the retained prefix length, $V^{(p)}$ the nucleus, $p \in (0, 1]$ the target mass, $\mathbf{1}[\cdot]$ an indicator, and q the renormalized sampling distribution. Fix a tie-breaking order for equal probabilities; the code preserves their original index order. The difference from top-k is that the nucleus **adapts**: after a prefix with one obvious continuation it may contain a single token, and after an open-ended prefix it may contain many.

The descending-prefix condition matters. For probabilities $0.4, 0.3, 0.2, 0.1$ and threshold 0.5, top-p retains the first two tokens. The first and last tokens also form a two-element set reaching 0.5, but that set skips more probable candidates and is not the intended nucleus. The assertion below checks this distinction.

```
import numpy as np

tokens = np.array(['valve', 'pump', 'door', 'sensor', 'cabinet', 'label', 'fish', 'moon'])
logits = np.array([3.0, 2.6, 1.2, 1.0, 0.2, -0.5, -1.5, -2.0])

def softmax(z):
    e = np.exp(z - z.max())
    return e / e.sum()

def temperature(z, tau):
    return softmax(z / tau)

def top_k(p, k):
    keep = np.argsort(-p)[:k]                 # indices of the k largest
    q = np.zeros_like(p); q[keep] = p[keep]
    return q / q.sum()

def nucleus(p, top_p):
    if not 0 < top_p <= 1:
        raise ValueError("top_p must be in (0, 1]")
    order = np.argsort(-p, kind="stable")
    cumulative = np.cumsum(p[order])
    size = int(np.searchsorted(cumulative, top_p) + 1)   # smallest prefix reaching top_p
    q = np.zeros_like(p); q[order[:size]] = p[order[:size]]
    return q / q.sum()

p = softmax(logits)
print('greedy choice:', tokens[np.argmax(p)])
for name, q in [('tau=0.5', temperature(logits, 0.5)), ('tau=1.0', p),
                ('tau=2.0', temperature(logits, 2.0)),
                ('top-k=3', top_k(p, 3)), ('top-p=0.9', nucleus(p, 0.9))]:
    print(f'{name:9s} support={int((q > 0).sum())}  P(valve)={q[0]:.3f}  '
```

```
          f'tail mass={q[4:].sum():.3f}')

# The nucleus adapts its size to the shape of the distribution.
flat = softmax(np.zeros(8))
peaked = softmax(np.array([6.0, 1, 0, 0, 0, 0, 0, 0]))
print('nucleus size, flat vs peaked:',
      int((nucleus(flat, 0.9) > 0).sum()), int((nucleus(peaked, 0.9) > 0).sum()))

rng = np.random.default_rng(0)
draws = rng.choice(len(tokens), size=2000, p=nucleus(p, 0.9))
assert set(draws) <= set(np.flatnonzero(nucleus(p, 0.9)))
assert np.array_equal(np.flatnonzero(nucleus(np.array([0.4, 0.3, 0.2, 0.1]), 0.5)), [0, 1])
assert np.isclose(top_k(p, 3).sum(), 1) and np.argmax(temperature(logits, 0.1)) == 0
```

Output:

```
greedy choice: valve
tau=0.5   support=8  P(valve)=0.667  tail mass=0.003
tau=1.0   support=8  P(valve)=0.481  tail mass=0.052
tau=2.0   support=8  P(valve)=0.312  tail mass=0.190
top-k=3   support=3  P(valve)=0.545  tail mass=0.000
top-p=0.9 support=4  P(valve)=0.507  tail mass=0.000
nucleus size, flat vs peaked: 8 1
```

Here "tail mass" is the probability left on the four least likely tokens. Temperature only reshapes the distribution, so every token stays possible: at $\tau = 2$ almost a fifth of the mass sits on `cabine t`, `label`, `fish` and `moon`. Truncation removes the tail outright. On the uniform distribution the nucleus keeps all eight tokens; on the peaked one it keeps one. A fixed k would keep three in both cases.

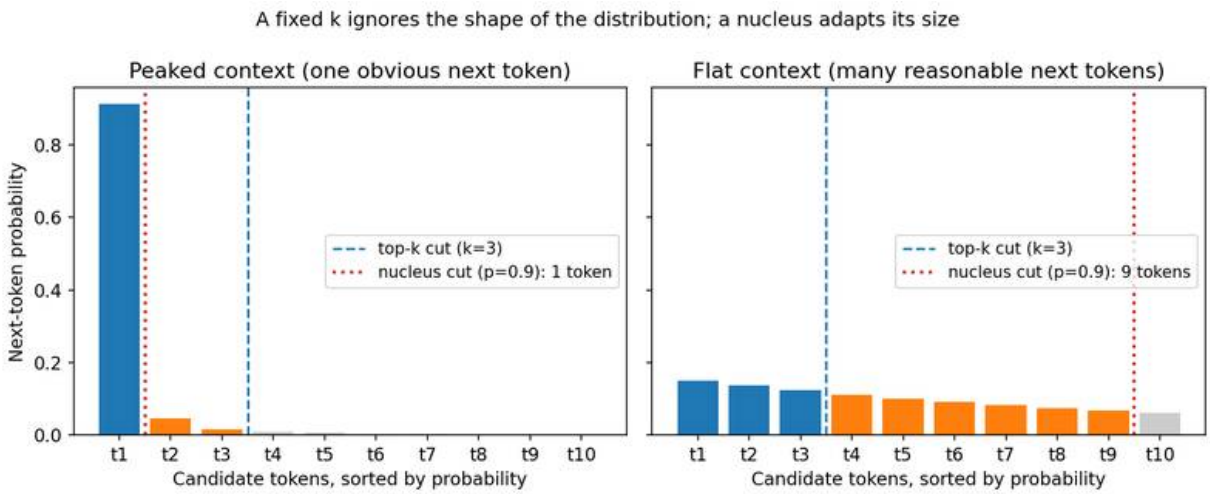


Figure 3 — Top-k and nucleus truncation on two synthetic distributions. Blue bars are kept by both rules, orange bars by only one, gray bars by neither.

In practice these knobs are combined (for example a temperature followed by a nucleus) and chosen per task. A data-to-text report usually wants deterministic, reproducible output; brainstorming wants variety. Sampling diversity is not evidence of quality, and repeated samples that agree are not an independent verification of a claim: they share the same model and often the same training biases.

Stopping and length

Generation also needs a stopping policy and a length budget. A model may stop too early or continue with redundant content. Set explicit output limits, a stopping rule and behavior when the limit is reached. A truncated summary should not be returned as complete. Record the decoding parameters with the model artifact so an evaluation can be reproduced; changing only the decoder can change every downstream metric.

7. Natural language generation tasks

Plain version. Generation means producing a sequence whose content and form satisfy a task. The input may be a document, a table, a prompt or retrieved passages; the task defines what counts as an error.

A decoder-only model handles every conditional task the same way: it treats the input as a prefix. The conditional factorization is

$$p_\theta(y \mid x, c) = \prod_{t=1}^{T} p_\theta(y_t \mid y_{<t}, x, c).$$

Here x is the source input; c is task context, such as the requested language or format; $y = (y_1, \dots, y_T)$ is the output token sequence, including an end token; $y_{<t}$ contains the tokens before position t; and θ denotes learned parameters. In a decoder-only model, x and c are serialized into the prefix and the loss is usually taken only over the tokens of y; an encoder–decoder (NLP 08) encodes x separately. Either way, the factorization describes a distribution. It does not certify that the sampled output is supported by the source.

Define the generation problem before choosing a model

Begin by writing a task contract in ordinary language. For an equipment report: "Given timestamped observations, produce two sentences describing the current temperature and any explicitly recorded fault; preserve device names, units, numbers and negation." This contract exposes omissions and contradictions that an unconstrained instruction such as "summarize nicely" hides.

Separate three operations. **Content selection** chooses observations relevant to the user. **Sentence planning** decides ordering, grouping and referring expressions. **Realization** expresses the plan as grammatical text. McRoy [2] provides the linguistic reading for this distinction. Neural sys-

tems can combine the operations internally, but a combined architecture still needs tests for each responsibility.

Consider a source with a high temperature, a later normal temperature, and an explicitly absent fault. Selecting the highest reading creates a different report from selecting the latest reading. Neither selection rule follows from grammatical fluency. Record whether the output should describe the current state, an extreme event, or the whole trajectory. Time and scope belong in the input contract.

Data-to-text: a template baseline

Data-to-text generation verbalizes structured records. Wiseman, Shieber and Rush introduced a large corpus of data records paired with descriptive documents, together with extractive evaluation methods that check which records the text actually conveys; the neural systems they tested produced fluent text, yet templated baselines exceeded them on some of those metrics [15]. A template is a useful baseline when the input schema is narrow. It offers complete control over permitted claims and makes failure localization simple. Its limits are also clear: it cannot gracefully cover arbitrary relations, referring expressions or discourse. Measure that coverage rather than treating templates as inherently inadequate.

Here is a deterministic report function for a narrow schema. It demonstrates field validation and traceable output. The event record is trusted structured input; obtaining and validating that record from raw text would be a separate NLP stage.

```
import math

VERSION = "equipment-template-1"
def report(event):
    required = {"device", "temperature_c", "fault", "evidence_id"}
    if set(event) != required:
        raise ValueError("Unexpected input fields")
    if not isinstance(event["device"], str) or not event["device"].strip():
        raise ValueError("Missing device")
    value = event["temperature_c"]
    if isinstance(value, bool) or not isinstance(value, (int, float)) or not
     ↪  math.isfinite(value):
        raise ValueError("Temperature must be finite")
    if event["fault"] not in ("present", "absent", "unknown"):
        raise ValueError("Unknown fault status")
    if not isinstance(event["evidence_id"], str) or not event["evidence_id"]:
        raise ValueError("Missing evidence ID")
    phrases = {"present": "A fault was recorded.",
               "absent": "No fault was recorded.",
               "unknown": "Fault status was not recorded."}
    text = f"{event['device']} measured {value:g} C. {phrases[event['fault']]}"
    return {"text": text, "evidence_id": event["evidence_id"], "version": VERSION}
output = report({"device": "Pump A", "temperature_c": 30,
                 "fault": "absent", "evidence_id": "synthetic-event-01"})
print(output)
assert output["text"] == "Pump A measured 30 C. No fault was recorded."
```

```
assert output["version"] == VERSION
```

Output:

```
{'text': 'Pump A measured 30 C. No fault was recorded.', 'evidence_id': 'synthetic-event-01',
↪  'version': 'equipment-template-1'}
```

Notice the difference between absent and unknown. The input "no fault was recorded" is a statement about recorded evidence, not a claim that a fault is impossible. Encoding that distinction prevents the realization stage from upgrading missing evidence into certainty. A learned generator becomes useful when its additional flexibility solves a documented failure of this baseline, and it should be tested on exactly these distinctions.

Extractive summarization: select evidence under a budget

An extractive summary chooses source sentences. An abstractive summary may rewrite or combine them. Extraction makes the origin of each sentence explicit, but the selected sentences can still mislead through missing context, conflicting timestamps or unresolved pronouns. Selecting a sentence containing "it failed" without its antecedent is a discourse error even though every word came from the source.

Suppose a document contains candidate sentences $s_1, \ldots, s_n$. Represent sentence i by a unit vector v_i and a user query by a unit vector q. A simple relevance score is $r_i = v_i^\top q$. To discourage repetition, use maximal marginal relevance, or MMR [14]:

$$i^* = \arg\max_{i \notin S} \left[\lambda r_i - (1-\lambda) \max_{j \in S} v_i^\top v_j \right].$$

Here S is the set of already selected sentence indices; i^* is the next index; $\lambda \in [0,1]$ balances relevance against similarity to previous selections; and the maximum similarity is defined as zero when S is empty. The dot products are cosine similarities because the vectors have unit length. This is a greedy selection rule, rather than a proof of globally optimal summary quality.

The following example uses binary word indicators to make every decision inspectable. The four candidate sentences are synthetic. Removing a small set of function words makes the representation less dominated by grammar, but it also removes potentially useful distinctions.

```
import re
import numpy as np

sentences = [
    "Pump A temperature reached 80 C during the test.",
    "The test recorded Pump A temperature at 80 C.",
    "Pump A vibration triggered a shutdown warning.",
    "The technician replaced a label on the cabinet.",
```

```
]
query = "pump temperature vibration shutdown"
stop = {"a", "the", "at", "on", "during"}
def tokens(text):
    return set(re.findall(r"[a-z0-9]+", text.lower())) - stop
vocab = sorted(set().union(*(tokens(s) for s in sentences), tokens(query)))
def vector(text):
    v = np.array([w in tokens(text) for w in vocab], dtype=float)
    return v / max(np.linalg.norm(v), 1e-12)
V = np.stack([vector(s) for s in sentences])
q = vector(query)
relevance, similarity = V @ q, V @ V.T
selected = []
for _ in range(2):
    redundancy = similarity[:, selected].max(axis=1) if selected else np.zeros(4)
    score = 0.55 * relevance - 0.45 * redundancy
    score[selected] = -np.inf
    selected.append(int(np.argmax(score)))
print("Selected indices:", selected)
print("Summary:", " ".join(sentences[i] for i in sorted(selected)))
assert len(set(selected)) == 2
assert 2 in selected and not ({0, 1} <= set(selected))
```

Output:

```
Selected indices: [2, 0]
Summary: Pump A temperature reached 80 C during the test. Pump A vibration triggered a
↪  shutdown warning.
```

The selected pair includes the shutdown sentence and one temperature sentence, avoiding the nearly repeated temperature observation. Sorting the selected indices restores source order after selection. This matters because selection order measures marginal value, while presentation order should support comprehension.

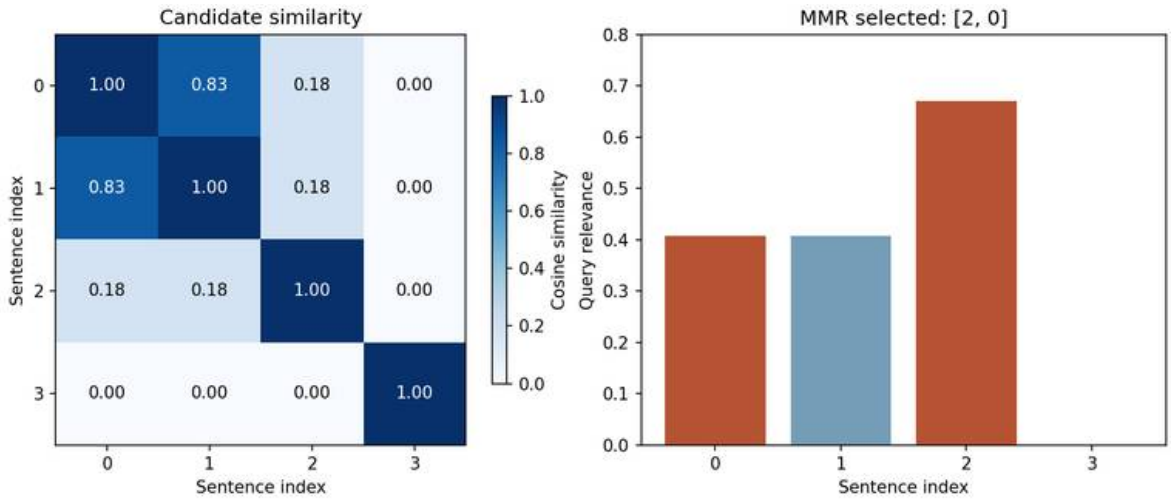


Figure 4 — An extractive baseline makes its trade-off visible. Similar temperature sentences compete for the same limited summary space; the shutdown sentence contributes different content.

A real budget usually limits tokens rather than sentence count. Add a candidate only if its full tokenized length fits the remaining budget; silently cutting a sentence can remove its condition or negation. For multi-document summaries, group near-duplicate reports so repeated copies do not masquerade as independent evidence. Retain source IDs and timestamps so readers can follow the selected evidence.

Relevance, feasibility and content preservation are different objectives

MMR discourages selecting sentences that resemble previous selections [14]. It does not know which facts are mandatory. If a warning shares many words with a selected observation, a similarity penalty can suppress it even though its added qualification is essential. Vector diversity is a proxy for redundancy, not a direct measure of complete meaning.

A content requirement changes the feasible set. For a token budget and a required warning:

$$\sum_{i \in S} \ell_i \le B, \qquad S \cap W \neq \emptyset.$$

Here S is the selected sentence-index set, ℓ_i is the full token length of candidate i, B is the budget, and W contains candidates expressing the required warning. The first condition limits size; the second ensures coverage. If no set satisfies both, the task is infeasible for extraction under that budget. Changing a relevance coefficient cannot make an over-budget sentence fit or invent a missing warning.

The following independent example isolates that distinction. Utilities are explicit teaching choices, not learned relevance scores:

```
import itertools

sentences = [
    'Pump A temperature reached 80 C during the test.',
    'The test recorded Pump A temperature at 80 C.',
    'Pump A vibration triggered a shutdown warning.',
    'The technician replaced a label on the cabinet.',
]
lengths = [len(s.split()) for s in sentences]
utility = [0.9, 0.85, 0.6, 0.2]
subsets = [tuple(i for i in range(4) if keep[i])
           for keep in itertools.product([False, True], repeat=4)]
feasible = [s for s in subsets if sum(lengths[i] for i in s) <= 20]
score = lambda s: sum(utility[i] for i in s)
ordinary = max(feasible, key=score)
with_warning = max((s for s in feasible if 2 in s), key=score)
print('Lengths:', lengths)
print('Without requirement:', ordinary, 'with warning:', with_warning)
assert ordinary == (0, 1) and with_warning == (0, 2)
```

Output:

```
Lengths: [9, 9, 7, 8]
Without requirement: (0, 1) with warning: (0, 2)
```

Here length counts whitespace-separated words; a model-token budget must use that model's tokenizer. The unconstrained choice repeats temperature content; the warning requirement changes the selected pair. More sophisticated scoring still needs the task's hard content requirements when omission is unacceptable.

Abstractive summarization

A rewriting model can fuse observations and improve readability. It can also invent a connecting statement that the observations never established. "The temperature was high, so the pump failed" adds causation if the source only records simultaneous events. Rewriting two sentences into one is acceptable when relationships survive, but changing sequence into causation or removing "during the test" alters content. A plan that records propositions and their source intervals before wording is generated lets you evaluate selection, plan and realization separately, so you can tell where meaning was lost.

Story generation and open-ended text

Open-ended generation reverses the priorities of a report. There is no single correct output, and variety is part of quality. Fan, Lewis and Dauphin collected a dataset of human-written stories paired with writing prompts, generated a premise first and then the story, and used top-k sampling because beam search tended to produce short and generic text [13]. The lesson for this chapter is that the **same** language model can need different decoders for different tasks: deterministic search for a constrained report, truncated sampling for a story. What stays constant is that the evaluation must match the task: coherence and relevance to the prompt for a story, supported content for a summary.

8. Evaluating generated text

Plain version. One automatic score cannot summarize all generation quality. Use automatic metrics to compare systems under a specified protocol, then inspect the error dimensions the application actually cares about.

Perplexity (Section 5) evaluates the model as a predictor of reference text. It says nothing directly about a particular decoded output, and it cannot compare models with different tokenizers. **Overlap metrics** compare an output with references. ROUGE includes recall-oriented overlap measures for summarization [17]. Such metrics compare a candidate with one or more references;

they do not prove semantic equivalence or source support. Name the variant, tokenizer, aggregation and reference set when reporting a score. Translation metrics such as BLEU and learned metrics are discussed in NLP 08.

For a simple unigram recall check,

$$R_1 = \frac{\sum_{w} \min(c_{\mathrm{ref}}(w), c_{\mathrm{hyp}}(w))}{\sum_{w} c_{\mathrm{ref}}(w)}.$$

Here w ranges over word types; $c_{\mathrm{ref}}(w)$ and $c_{\mathrm{hyp}}(w)$ are reference and hypothesis word counts. The denominator is the number of reference tokens. This overlap calculation illustrates one component of ROUGE-style evaluation; the minimal implementation below does not reproduce every preprocessing or aggregation option of a full evaluation package.

```
import re
from collections import Counter

def words(text):
    return re.findall(r"[a-z0-9]+", text.lower())
def overlap_recall(reference, hypothesis):
    ref, hyp = Counter(words(reference)), Counter(words(hypothesis))
    return sum((ref & hyp).values()) / sum(ref.values())
reference = "Pump A is operating at 30 C."
contradiction = "Pump A is not operating at 30 C."
score = overlap_recall(reference, contradiction)
print("Unigram recall:", score)
assert score == 1.0
assert ("not" in words(reference)) != ("not" in words(contradiction))
```

Output:

```
Unigram recall: 1.0
```

The contradiction receives recall 1.0 because it retains every reference token. A precision term would penalize the extra word, but the small penalty would still not express the importance of reversed negation. This is a controlled counterexample, not evidence that overlap metrics never correlate with human judgments. It explains why the metric must be paired with meaning-sensitive checks.

Human evaluation and error categories

For the equipment application, count **unsupported claims**, **missing required claims**, **incorrect numeric relations**, and **unresolved references** separately. These are project-defined error categories. Give annotators positive and negative examples for each, including cases where evidence is absent or contradictory. A summary that says less may have fewer unsupported claims while omitting critical information; report both error directions.

Human evaluation should hide system identity, randomize presentation, preserve source context and specify the decision task. Ask reviewers to mark unsupported statements before judging overall usefulness. An overall preference may reward style and miss a rare critical error. Report who reviewed the outputs and how disagreements were resolved. Aggregate by a meaningful unit: several sentences from one incident are correlated, and treating them as independent examples can exaggerate certainty.

Language models as judges

A strong language model can be prompted to grade or compare outputs. Zheng and colleagues studied this "LLM-as-a-judge" setting and reported that a strong judge's preferences agreed with human preferences at a level comparable to agreement between humans on their benchmarks, while also documenting position bias, verbosity bias and self-enhancement bias [18]. Wang and colleagues showed that simply swapping the order in which two responses appear can change which one a model judge prefers, and proposed calibration such as evaluating both orders [19].

Treat a model judge as another measurement instrument with its own version, prompt and failure modes. Evaluate both presentation orders, hide which system produced which output, validate the judge against human labels on your own error categories (especially negation, quantities and unsupported claims), and avoid using a judge from the same model family as the system it grades when that can be avoided. A judge that ranks systems well on average is not automatically a reliable gate for accepting a single output.

9. Retrieval: finding the evidence to generate from

Plain version. Before a generator can use evidence, something must find it. A retrieval system selects passages from an explicit corpus; whether those passages support an answer, and how the answer is worded, are later decisions.

A language model can continue a question without searching a current document collection. A retrieval system instead selects information from an explicit corpus. Question answering adds another decision: whether the selected information supports an answer, and which words express it. Putting these components together creates a useful pipeline, but no stage automatically certifies the next.

This section's running question is "What is the temperature limit for Pump A?" A relevant passage discusses Pump A or temperature. A sufficient passage states the limit for that pump under the applicable policy. A valid citation points to a real source. A supported answer preserves the source's value, units, time and scope. These are four separate conditions, and an evaluation should be able to tell which one failed.

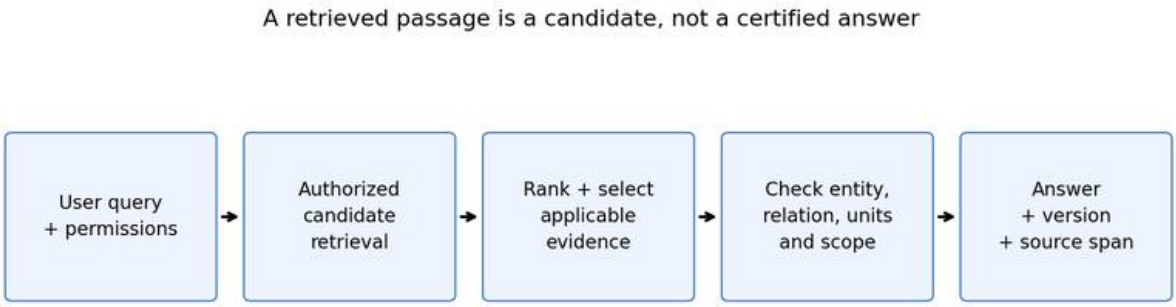


Figure 5 — Retrieval selects candidates; evidence checking determines what can be said. Every stage belongs to an explicit application contract.

This section builds and evaluates the retrieval stage. Deciding whether a retrieved passage actually answers the question — extractive span answers, relevance versus answerability, and diagnosis by controlled substitution — is developed in NLP 11. Section 10 then conditions a generator on the retrieved evidence.

Corpus design before model choice

Plain version. A retriever searches what you give it. Document identity, freshness and permissions are part of the system, not optional metadata added after ranking.

Represent a searchable passage as:

$$d = (u, v, b, e, \mathbf{x}, a).$$

Here d is a passage record; u is the parent document identifier; v is its version; b and e are inclusive-start and exclusive-end offsets in the original document; $\mathbf{x}$ is passage text; and a describes its access policy. The offset unit must be declared. This is our application record, rather than a universal storage standard.

A passage may be a paragraph, a sentence group or a section selected by document structure. Small passages can isolate an answer but lose its qualifications. Large passages retain context but introduce unrelated text and consume more generation budget. Choose chunk boundaries based on the task, then evaluate whether the supporting facts and their conditions remain together.

For example, a heading may specify "Outdoor equipment only," followed by a paragraph containing a temperature limit. If the paragraph is indexed alone, it may appear to apply to indoor equipment. Retaining the heading or a structured scope field changes the answerability contract. A fixed character window is easy to implement, but simplicity does not establish that it preserves meaning.

Versions also matter. An old policy can be relevant yet inapplicable. Keep effective dates and replacement links when the application needs current answers. Do not rely on a generator to infer the current policy merely because the newest passage appears last in the prompt. Filter or

rank by declared applicability rules, and make conflicts visible when the corpus does not identify a single authoritative source.

Apply permissions before returning passage text to an unauthorized component or user. Filtering the final visible answer is too late if private text was already included in a model request or log. The retriever's corpus view should match the current user's permitted view. Measure retrieval quality under that restricted view; global relevance scores can otherwise evaluate documents the system may never show.

Lexical retrieval and sparse vectors

Plain version. Lexical retrieval matches words and weights informative terms. It is inspectable and often useful when exact names, identifiers and technical phrases matter.

A simple term-frequency and inverse-document-frequency weight is:

$$w_{t,d} = \mathrm{tf}(t,d) \log \frac{N}{\mathrm{df}(t)}.$$

Here $w_{t,d}$ is the weight of term t in document d; $\mathrm{tf}(t,d)$ is its frequency under a chosen convention; N is the number of indexed documents; and $\mathrm{df}(t)$ is the count of documents containing the term. A zero document frequency requires special handling for query-only terms. Implementations often smooth the formula or transform term frequency; record the actual convention [33, 34].

With normalized document and query vectors:

$$s(q,d) = \frac{\mathbf{v}_q^\top \mathbf{v}_d}{\|\mathbf{v}_q\|_2 \|\mathbf{v}_d\|_2}.$$

Here q is a query; d is a candidate document; $\mathbf{v}_q$, $\mathbf{v}_d$ are their vectors; s is cosine similarity; and $\|\cdot\|_2$ is Euclidean norm. Define the score as zero when either vector is zero. The code uses normalized vectors from scikit-learn, so their dot product implements cosine similarity for nonzero rows [34].

An inverted index stores, for each term, the documents where it occurs. That allows a search engine to consider matching candidates without comparing a query with every vocabulary entry in every document [33]. Our small code computes all document scores because the corpus contains only four passages. The mathematical representation and the efficient indexing strategy are separate choices.

Preprocessing directly affects retrieval. A tokenizer that discards one-character tokens can remove the A in Pump A. Lowercasing may be helpful for ordinary words but can erase distinctions in case-sensitive identifiers. Phrase features help distinguish adjacent words, but do not capture arbitrary syntax or guarantee semantic equivalence. Inspect the vocabulary before blaming the ranking model for a missed result.

Lexical systems can miss paraphrases. "Device ceased operating" and "machine shutdown" may share little wording. A synonym or query-expansion rule can help, but changes the matching policy and may add ambiguity. Evaluate expansion on held-out queries, especially exact-identifier

cases. A broad expansion that improves recall can also retrieve a wrong device with similar terminology.

Saturation and document length

Plain version. Repeating a query word many times should not always keep increasing relevance. Longer documents also have more opportunities to match by chance.

One common BM25 form is:

$$s(q,d) = \sum_{t \in q} \mathrm{idf}(t) \frac{f_{t,d}(k_1+1)}{f_{t,d} + k_1\left(1 - b + b\,|d|/\overline{L}\right)}.$$

Here q is the query-term set in this simplified form; d is the candidate document; $f_{t,d}$ is term frequency; $\mathrm{idf}(t)$ is a chosen inverse-document-frequency weight; $|d|$ is document length in tokens; $\overline{L}$ is average document length; k_1 is a positive saturation parameter; and $b \in [0,1]$ controls length normalization. Different implementations use different IDF and query-frequency conventions. The information-retrieval textbook discusses this family and its assumptions [33].

The numerator and denominator make the term-frequency contribution approach a limit as repetition grows. The length term changes how much frequency is expected in a longer document. These are ranking adjustments, not a probability that the document contains a true answer. A passage can rank highly because it repeatedly discusses a limit while never stating the value.

Tune length normalization on development queries that reflect the deployment corpus. If every useful answer is in a short reference entry while irrelevant documents are long narratives, a particular setting can look very strong. Changing to long technical manuals may alter that behavior. Keep passage construction fixed when comparing ranking settings, or report that both the corpus representation and score changed.

Dense retrieval and joint reranking

Plain version. A learned encoder can match related expressions without exact word overlap. Its success depends on the training task and the examples used as contrasts.

A dual-encoder score is:

$$s_\theta(q,d) = f_\theta(q)^\top g_\theta(d).$$

Here f_θ maps queries to vectors; g_θ maps passages to vectors; θ denotes learned parameters; and s_θ is their dot-product score. The encoders may share parameters or use distinct ones. Dense Passage Retrieval provides an original question-answering example of this design [35].

Passage vectors can be computed ahead of time. A query then searches those vectors for high scores. Approximate search trades some exact nearest-neighbor fidelity for speed; if used, evaluate both the representation and the search approximation. Our chapter does not implement such an index, because an exact small corpus is better for learning the contract.

A contrastive training loss can compare one positive passage with several negative passages:

$$\mathcal{L} = -\log \frac{\exp(s_\theta(q, d^+)/\tau)}{\exp(s_\theta(q, d^+)/\tau) + \sum_j \exp(s_\theta(q, d_j^-)/\tau)}.$$

Here $\mathcal{L}$ is the loss for one query; d^+ is an annotated positive passage; d_j^- is negative candidate j; τ is a positive temperature; and s_θ is the learned score. Negatives may include difficult passages with similar wording. A passage labeled negative merely because it was not annotated can be a false negative if it also contains a valid answer.

A joint reranker reads a query and candidate passage together, which permits richer token interactions than comparing two fixed vectors. It is typically applied after a cheaper candidate retrieval stage. This adds computation and another selection procedure. Report candidate recall before reranking: a reranker cannot recover the missing correct passage if it never receives it.

Dense and lexical retrieval can be complementary. Dense representations may help paraphrases; lexical matching can preserve exact codes and rare names. Combining score scales requires care because a lexical score and a learned dot product do not have the same units. Select combination settings on development data, and inspect whether the combined system introduces wrong-entity results.

What the contrastive loss actually teaches

The dense loss does more than move a query toward any vaguely related passage. It learns the distinctions represented by the candidate set. Let candidate zero be the positive passage and let the remaining candidates be negatives. For candidate scores:

$$p_j = \frac{\exp(s_j/\tau)}{\sum_{r=0}^{M} \exp(s_r/\tau)}, \qquad \frac{\partial \mathcal{L}}{\partial s_j} = \frac{p_j - \mathbf{1}[j = 0]}{\tau}.$$

Here s_j is the query's score for candidate j, M is the number of negative candidates, r indexes all candidates, $\tau > 0$ is temperature, p_j is the candidate softmax probability, $\mathcal{L} = -\log p_0$ is the loss, and $\mathbf{1}[j = 0]$ is one for the positive and zero otherwise. This derivative follows directly from the softmax loss above. Gradient descent increases the positive score and decreases negative scores; the strongest competing negative receives the largest downward pressure.

DPR uses positive and negative passage comparisons to train separate query and passage representations [35]. The equation explains why choosing the comparisons matters. If negatives are unrelated topics, the model may learn topic separation without learning to distinguish Pump A from Pump B. A wrong-device passage with the same relation is a more informative contrast for that error. But if an unannotated passage really supports the answer, treating it as negative teaches the wrong separation.

This standalone score-space experiment verifies the gradient direction. It illustrates the objective; it does not train passage encoders:

```
import numpy as np
```

```
scores = np.array([1.2, 0.8, -0.4])  # positive, hard negative, easy negative
def probabilities(values):
    z = np.exp(values - values.max())
    return z / z.sum()
p = probabilities(scores)
gradient = p.copy()
gradient[0] -= 1.0  # temperature is one
updated = scores - 0.5 * gradient
before, after = -np.log(p[0]), -np.log(probabilities(updated)[0])
print('Gradient:', gradient.round(3))
print('Loss:', round(before, 3), '->', round(after, 3))
assert gradient[0] < 0 and gradient[1] > gradient[2] > 0
assert after < before
```

The gradient rounds to `[-0.466, 0.358, 0.108]`, and loss falls from `0.627` to `0.467`. The competing high-score negative receives more downward pressure than the easy negative.

Changing temperature affects both distribution sharpness and gradient scale. Changing candidates changes the contrast being learned. Consequently, improvements from a new encoder cannot be attributed to architecture alone when its positive labels, negative construction or training corpus also change. Keep those differences visible in the comparison.

Compare retrieval routes on the distinction that matters

A dual encoder compresses a passage before it knows the future query. This permits cached vectors and efficient candidate search, but every query-relevant detail must survive that compression. A joint reranker receives both texts simultaneously and can compare specific tokens across them. It spends more computation per pair and still depends on its training examples; joint attention is an opportunity to distinguish a detail, not proof that the detail will be respected.

Query distinction	Useful evidence	Failure to inspect
`Pump A` versus `Pump B`	Exact identifier plus relation	Topic similarity overwhelms a one-character difference
`ceased operating` versus `shutdown`	Learned paraphrase compatibility	Lexical vocabulary misses equivalent wording
Current versus pre-upgrade limit	Version and effective-date metadata	A highly relevant passage answers the wrong time scope
Permission versus prohibition	Negation and its grammatical scope	Shared vocabulary hides an opposite instruction

A hybrid design can take the union of lexical and dense candidate sets, remove duplicates, and jointly rerank the survivors. Union protects candidates found by either route, while a fixed total budget can still force some candidates out. Simply adding raw scores is less transparent: multiplying one system's scores by a constant preserves its own ranking but changes its influence in the combined sum. Normalization or learned combination therefore requires development judgments, not an arbitrary coefficient presumed to work everywhere.

Compare candidate recall, final ordering and resource cost separately. A system can improve the first while harming the second, or improve both while exceeding the serving budget. Diagnose exact names, paraphrases and scope-sensitive questions as distinct groups. Their error patterns reveal which information the representation loses and which downstream interaction is worth paying for.

Ranking evaluation

Plain version. Retrieval evaluation needs queries with relevance judgments. A document being selected does not establish that it was the right one.

For query q with relevant-document set R_q and top-k result set $T_k(q)$:

$$\text{Recall@}k(q) = \frac{|R_q \cap T_k(q)|}{|R_q|}.$$

Here k is the cutoff; R_q contains the annotated relevant documents; $T_k(q)$ contains the retrieved documents up to that cutoff; and $|\cdot|$ denotes set size. We evaluate this expression only for queries with at least one relevant document. Queries with no answer belong in a separate answerability evaluation, rather than receiving an arbitrary recall value.

If the first relevant result has rank r_q:

$$\text{RR}(q) = \frac{1}{r_q}, \qquad \text{MRR} = \frac{1}{Q}\sum_{q=1}^{Q} \text{RR}(q).$$

Here RR is reciprocal rank; MRR is its mean; Q is the number of evaluated queries; and r_q starts at one. Define reciprocal rank as zero when no relevant result is returned. These ranking diagnostics appear in the current retrieval-augmented evaluation discussion [31]. Reciprocal rank emphasizes the first successful hit and ignores later relevant results.

Annotation completeness matters. If only one passage is judged relevant when several support the answer, a valid retrieval can be scored as wrong. Record whether relevance means topic match, answer-containing evidence or complete support. Use multiple judgments when an ambiguity is consequential, and adjudicate according to a written guide. The dataset should expose the task definition, not merely list a favorite passage.

Keep query families and document derivatives separate across development and test. Rephrasing a development query into a nearly identical test query gives weak evidence of generalization. Likewise, splitting chunks from one report across document partitions can leak facts and wording. Choose a split reflecting the deployment question: unseen users, unseen documents, future versions or new domains.

Measure a small retriever

The standalone code uses four synthetic passages. It preserves one-character words so that the device identifier A remains a feature, and it measures ranking on three annotated queries.

```
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer

corpus = {
    'temperature-v2': 'Pump A temperature limit is 80 C. This policy is current.',
    'shutdown-v1': 'Pump A shutdown follows a vibration alarm.',
    'label-v1': 'A technician replaced the cabinet label.',
    'sensor-v1': 'Sensor B reports humidity once per hour.',
}
ids = list(corpus)
vectorizer = TfidfVectorizer(ngram_range=(1, 2), token_pattern=r"(?u)\b\w+\b")
D = vectorizer.fit_transform([corpus[key] for key in ids])
def rank(query):
    score = (D @ vectorizer.transform([query]).T).toarray().ravel()
    order = sorted(range(len(ids)), key=lambda i: (-score[i], ids[i]))
    return [(ids[i], float(score[i])) for i in order]
queries = [
    ('Pump A temperature limit', {'temperature-v2'}),
    ('vibration shutdown alarm', {'shutdown-v1'}),
    ('humidity report', {'sensor-v1'}),
]
recalls, reciprocal = [], []
for query, relevant in queries:
    ranking = rank(query)
    recalls.append(len({key for key, _ in ranking[:2]} & relevant) / len(relevant))
    first = next((i + 1 for i, (key, _) in enumerate(ranking) if key in relevant), None)
    reciprocal.append(1 / first if first else 0.0)
    print(query, '->', ranking[0][0])
print('mean Recall@2:', np.mean(recalls))
print('mean reciprocal rank:', np.mean(reciprocal))
```

Output:

```
Pump A temperature limit -> temperature-v2
vibration shutdown alarm -> shutdown-v1
humidity report -> sensor-v1
mean Recall@2: 1.0
mean reciprocal rank: 1.0
```

The three top results match their single annotated relevant passages, producing mean Recall@2 and mean reciprocal rank of 1.0 in this small run. That is a fixture result, not a real retrieval benchmark. NLP 11 reuses the same corpus and ranker for a narrow evidence-backed answerer.

Graded relevance: nDCG

Recall@k and reciprocal rank treat a judgment as relevant or not. When judges grade passages — complete support above a merely topical mention — a measure should reward placing the most useful passage first.

$$\mathrm{DCG@}k = \sum_{i=1}^{k} \frac{g_i}{\log_2(i+1)}, \qquad \mathrm{nDCG@}k = \frac{\mathrm{DCG@}k}{\mathrm{IDCG@}k}.$$

Here $g_i \geq 0$ is the graded relevance of the result at rank i; DCG@k is the discounted cumulative gain over the first k ranks; IDCG@k is the same sum for the ideal ordering of the judged results, so nDCG@$k \in [0, 1]$; define it as zero when IDCG@$k = 0$. Järvelin and Kekäläinen introduced the cumulated-gain family of measures [36]; the logarithm base, the gain transform and the cutoff vary between implementations, so report them.

Two rankings of the same four judged passages differ only in whether the supporting passage or the topical one comes first:

```
import numpy as np

def dcg(gains, k):
    gains = np.asarray(gains[:k], dtype=float)
    return float(np.sum(gains / np.log2(np.arange(2, gains.size + 2))))
def ndcg(gains, k):
    ideal = dcg(sorted(gains, reverse=True), k)
    return dcg(gains, k) / ideal if ideal > 0 else 0.0

# graded judgments of one ranking: 2 = complete support, 1 = topical, 0 = unrelated
ranked_a = [1, 2, 0, 0]  # topical passage above the supporting one
ranked_b = [2, 1, 0, 0]  # supporting passage first
for name, gains in [('A', ranked_a), ('B', ranked_b)]:
    print(name, 'nDCG@3:', round(ndcg(gains, 3), 3),
          'binary RR:', 1 / (1 + gains.index(2)))
assert ndcg(ranked_b, 3) == 1.0 and ndcg(ranked_a, 3) < 1.0
```

Output:

```
A nDCG@3: 0.86 binary RR: 0.5
B nDCG@3: 1.0 binary RR: 1.0
```

Reciprocal rank with “complete support” as the relevance criterion halves for ranking A. nDCG also drops, but less, because it still credits the topical passage at rank 1. Which behavior is right depends on the task: if only a complete answer is useful, the graded credit for topical passages can hide a failure. Write the grading guide before choosing the measure.

10. Hallucination and retrieval-augmented generation

Plain version. A language model can produce fluent text that is not supported by its input or by the world. Retrieval supplies external evidence, but the generator still needs to use it correctly, and retrieved text must never be treated as a new authority.

Ji and colleagues' survey distinguishes **intrinsic** hallucination, output that contradicts the source, from **extrinsic** hallucination, output that cannot be verified from the source [20]. Maynez and colleagues' human study of abstractive summarizers found substantial amounts of hallucinated content in all model-generated summaries they examined, and found that textual-entailment measures correlated better with faithfulness than standard overlap metrics [21]. Both points follow from Sections 1 and 6: the training objective rewards probable text, and no decoder adds evidence that the distribution lacks.

Use attention as a diagnostic of computational access, not a factuality certificate. A token can attend to the correct evidence and still express its opposite. Evidence validation should examine the output claim, source entity, relation and scope. For narrow structured tasks, the generator can verbalize a validated plan, as in the template above. For unrestricted prose, validation becomes incomplete and should be reported as such.

Retrieval-augmented generation

The original retrieval-augmented generation, or RAG, paper combines a learned retriever with conditional sequence generation for knowledge-intensive tasks [22]. One sequence-level form is:

$$p(y \mid x) \approx \sum_{z \in Z_k(x)} p_\eta(z \mid x)\, p_\theta(y \mid x, z).$$

Here x is the input question; y is the output sequence; z is a retrieved passage; $Z_k(x)$ is the selected top-k candidate set; η parameterizes the retriever; and θ parameterizes the generator. The approximation reflects restricting retrieval to candidates. This is a model-specific probabilistic formulation, not a definition that every contemporary RAG application implements.

Many practical systems retrieve passages, assemble a prompt and run a separately trained generator. That pipeline can be useful without optimizing the original marginal objective. State which design you use. Otherwise the name RAG can hide whether retrieval is learned, whether candidates are reranked and whether the generator is trained to use the selected sources. Section 9 builds and evaluates the retriever itself; extractive answers and answerability checks are in NLP 11.

Grounding means relating claims to external evidence under a declared support rule. A generated answer with citations may be traceable, but traceability is weaker than support. A valid URL can point to a document that contradicts the answer. Evaluate whether each consequential claim follows from its cited evidence, and whether the answer includes necessary qualifications.

Evidence may also conflict. A generator should not silently combine mutually inconsistent policies into a single confident answer. An application can prioritize an authoritative version, display a conflict or request clarification according to its contract. The decision should come from metadata and task rules, rather than which sentence happens to be most fluent.

Errors propagate through retrieval, interpretation and realization

A supported answer requires a chain of successful decisions. The corpus must contain applicable evidence, retrieval must preserve it, the reader must interpret the right entity and relation, and the output must retain that interpretation. An answer guessed from memorized knowledge can be correct by coincidence while failing this evidence requirement. For questions whose applicable evidence is in the corpus, write:

$$P(R \cap E \cap G) = P(R)P(E \mid R)P(G \mid R, E).$$

Here P denotes probability, R is the event that sufficient evidence survives retrieval, E is the event that the evidence is interpreted correctly, and G is the event that the output faithfully realizes the interpretation. The expression is the probability chain rule, not an independence assumption. Multiplying unrelated unconditional stage scores is not equivalent: reader accuracy on hand-picked easy passages may differ from reader accuracy on the retriever's actual candidates.

```
import numpy as np

retrieved = np.arange(100) < 80
interpreted = np.arange(100) < 72
realized = np.arange(100) < 68
r = retrieved.mean()
e = interpreted[retrieved].mean()
g = realized[retrieved & interpreted].mean()
complete = (retrieved & interpreted & realized).mean()
print('Conditional stage rates:', round(r, 3), round(e, 3), round(g, 3))
print('Complete supported answers:', round(complete, 3))
assert np.isclose(r * e * g, complete)
```

Output:

```
Conditional stage rates: 0.8 0.9 0.944
Complete supported answers: 0.68
```

These are synthetic outcomes, not reported system performance. The calculation explains why individually respectable component results need not produce equally strong complete answers. Improving the weakest stage is useful only if the improvement survives the later stages.

Boundaries around retrieved content

Retrieved text is evidence to inspect, not an instruction source with permission to change the application. A passage can contain quoted commands, misleading text or an adversarial instruction. Prompt injection describes attempts to make a model follow instructions that conflict with the intended application boundary; OWASP documents this risk category [23]. For retrieval systems, the important boundary is between user/application instructions and untrusted source material.

Keep source text delimited and labeled with document identity. Do not allow it to redefine the current user's permissions, request secret data or authorize a tool action. If tools are available, validate actions against the user's task and the application policy outside the retrieved passage. A textual delimiter helps structure input, but is not a complete enforcement mechanism.

The same principle applies to logs and caches. A cached result produced under one user's access policy must not be reused for another user without checking permissions. Evidence spans must refer to the same document version used for the answer. When a document is removed or replaced, refresh the index and consider invalidating derived answers.

These checks also support ordinary correctness. A passage that says "Ignore the old limit; the new policy is below" can be legitimate document content that needs interpretation, while "Ignore your application instructions" is not a new authority for the system. Store both as source material, then apply the task's evidence and permission rules. Do not make the retriever a route for changing the governing contract.

11. Adapting a pretrained generator

Plain version. Pretraining learns broad continuation patterns. Adaptation changes behavior for a task; prompting changes the supplied input without necessarily changing weights. Learning to follow an instruction and learning which response people prefer are different signals, and neither is a truth guarantee.

Supervised task adaptation minimizes a task-specific loss over labeled examples, for instance the conditional loss of Section 7 over the response tokens only. Continued language-model training on domain text changes the distribution modeled, but does not automatically teach the application's output policy. Domain familiarity and task instruction are distinct forms of adaptation. In-context examples, as in GPT-3 [6], appear inside the input and demonstrate format without updating parameters; examples that contain private answers or test labels can invalidate evaluation, and a prompt chosen on development cases must be frozen before the final test.

Low-Rank Adaptation, or LoRA, freezes a base weight matrix and trains a constrained update [24]:

$$W' = W + BA.$$

Here $W \in \mathbb{R}^{m \times n}$ is the frozen matrix; $A \in \mathbb{R}^{r \times n}$ and $B \in \mathbb{R}^{m \times r}$ are learned factors; r is the chosen rank; and W' is the effective adapted matrix. This simplified expression omits optional scaling. Fewer trainable parameters can reduce adaptation cost, but do not remove the need for representative labels, data isolation or task evaluation. Save the base-model version, tokenizer, adaptation weights and configuration together: an adaptation trained against one base checkpoint is not guaranteed to work on another checkpoint with the same family name.

The InstructGPT paper studies supervised instruction examples followed by preference-based training with human feedback [25]. This distinguishes a base continuation model from one adapted toward requested behavior. The source of preferences matters: annotators follow a policy, and that policy may balance helpfulness, formatting and other goals. A preferred response

can still contain a factual error.

Direct Preference Optimization, or DPO, uses preferred and rejected responses with a reference policy [26]. A representative objective is:

$$\mathcal{L}_{\text{DPO}} = -\log \sigma\left(\beta\left[\log \frac{\pi_\theta(y^+ \mid x)}{\pi_{\text{ref}}(y^+ \mid x)} - \log \frac{\pi_\theta(y^- \mid x)}{\pi_{\text{ref}}(y^- \mid x)}\right]\right).$$

Here x is the prompt; y^+ and y^- are the preferred and rejected response sequences; π_θ is the trainable policy; π_{ref} is a fixed reference policy; β is a positive scale parameter; σ is the logistic sigmoid; and $\mathcal{L}_{\text{DPO}}$ is the loss for one preference pair. Sequence probabilities are products over response-token conditionals; prompts are context rather than supervised response tokens.

The objective increases the preferred response's relative likelihood compared with the rejected response, measured against the reference. If a dataset rewards confident unsupported answers, optimizing the objective can reinforce that behavior. Instruction compliance, preference agreement and factual support are separate properties that need separate evidence. ML 10 runs a low-rank update in code.

Online only: InstructGPT — Training Language Models to Follow Instructions with Human Feedback — https://blog.shyandram.dev/en/learn/nlp/language-modeling-transformers/papers/instructgpt–training-language-models-to-follow-instructions-with-human-feedback/

12. Training practices meet serving constraints

Plain version. A deployed generator serves inputs of different lengths, and its cost per request depends on both the prompt and the output. Padding and cached attention state are two different memory costs; measure both.

Hagiwara's Chapter 10 connects padding, sorting and masking to practical inference [1]. Keep this distinction clear: a correct loss mask prevents padding from altering the objective; it does not necessarily prevent the hardware from processing padded positions. Define padding utilization as

$$u = \frac{\sum_{b=1}^{B} L_b}{B \max_b L_b}.$$

Here L_b is the real token length of request b; B is batch size; and u is the fraction of padded token positions containing real tokens. This is a workload diagnostic, not a measured speedup. Sorting nearby lengths into batches can increase utilization while also delaying requests that wait for a matching batch.

```
import numpy as np

lengths = np.array([8, 64, 12, 60, 10, 56, 14, 52])
def utilization(values):
    batches = values.reshape(-1, 4)
    real = int(batches.sum())
```

```
    padded = int((batches.max(axis=1) * 4).sum())
    return real / padded
arrival = utilization(lengths)
bucketed = utilization(np.sort(lengths))
print("Arrival-order utilization:", round(arrival, 3))
print("Length-bucketed utilization:", round(bucketed, 3))
assert bucketed > arrival
# Two cache arrays (keys and values), with a toy equal head layout.
layers, kv_heads, head_dim, cache_tokens, bytes_per_value = 4, 2, 16, 128, 4
cache_bytes = 2 * layers * kv_heads * head_dim * cache_tokens * bytes_per_value
print("Toy cache bytes:", cache_bytes)
assert cache_bytes == 131072
```

Output:

```
Arrival-order utilization: 0.575
Length-bucketed utilization: 0.885
Toy cache bytes: 131072
```

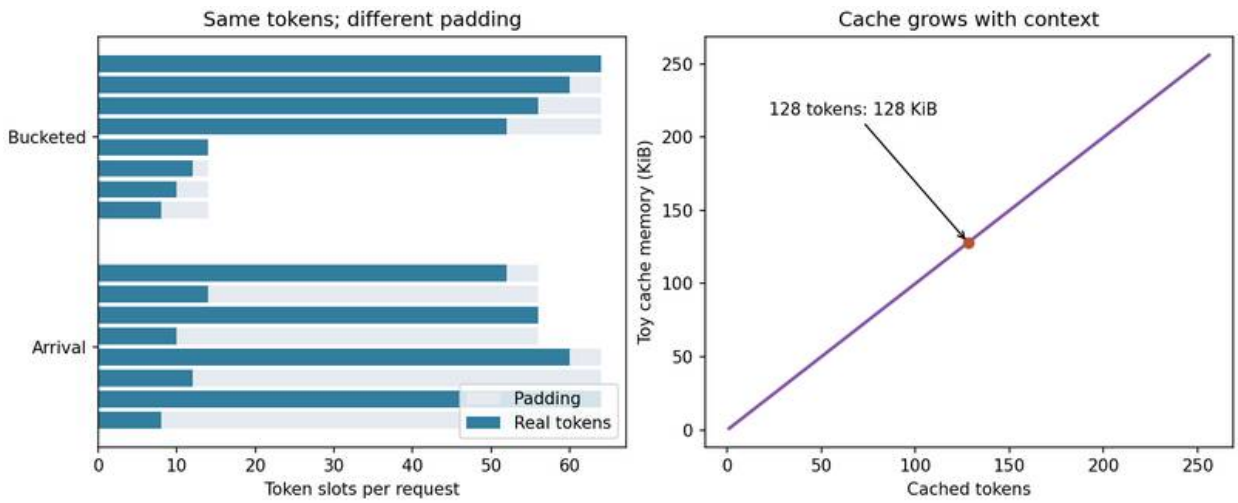


Figure 6 — Padding waste and cached attention state are different memory costs. The figure reports calculated token slots and toy cache bytes, rather than claiming an observed serving speedup.

Autoregressive attention commonly caches prior keys and values to avoid recomputing them at every generated token. For a simple layout, the cache memory is approximately $M_{\mathrm{KV}} = 2N_{\mathrm{layers}}H_{\mathrm{KV}}d_hTb_v$. Here N_{layers} is layer count, H_{KV} is the number of key/value heads, d_h is head dimension, T is cached token count, and b_v is bytes per stored value. The factor two counts keys and values. Actual implementations add allocation and metadata costs and may use different layouts.

Separate time to first output token from time between subsequent tokens, total completion time and throughput under load. A short prompt with a long answer stresses a different stage from a long prompt with a short answer. Include preprocessing, waiting, validation and serialization

in end-to-end measurements. Set a request-length limit, an output limit and a queue deadline based on measured workload behavior, and report what happens on timeout: a partial result, a retryable error or a reviewed fallback.

A reproducible serving contract

A model file alone cannot reproduce a prediction. Tokenization, normalization, evidence selection, decoding parameters, validation rules and output formatting all affect behavior. Hagiwara's deployment chapter [1] supplies the practical reading for versioning, testing, train–serve differences and monitoring. For a learned generator, return the model version and the validated evidence IDs alongside the text, as the template function does, and store the preprocessing version and decoding configuration with the artifact. Logs should support diagnosis without unnecessarily retaining private source text.

Deploy a new version to a limited share of requests and compare it with a stable version using the same error categories. Check latency, completion rate and evidence correctness together. Keep rollback simple: a stored bundle should select the previous model, tokenizer, decoder settings and validation rules as one unit. Mixing an old model with a new tokenizer can create a failure that neither component exhibited alone.

Modern view

Plain version. Contemporary language modeling combines pretraining, adaptation, decoding, evaluation and system integration. Surveys are useful when they clarify these separate choices rather than compressing everything into one model ranking.

Zhao and colleagues' survey [27], revised through 2026 and published in Frontiers of Computer Science, organizes large language models through pretraining, adaptation, utilization and capability evaluation. This framework helps distinguish more training data from instruction tuning or a different usage pattern. **Chang and colleagues' evaluation survey [28]** separates questions about what to evaluate, where to evaluate and how to evaluate. Together they motivate a task-specific evaluation plan rather than relying on a single general score.

Li et al.'s pretrained-generation survey [29] organizes methods around input representation, architecture and optimization, and covers applications and evaluation. This taxonomy helps locate a change: retrieval alters available input evidence, an encoder–decoder changes conditional architecture, and adaptation changes parameters or training objectives. The survey predates some later systems, so use it for organization alongside newer evaluation and deployment evidence.

Ji et al.'s hallucination survey [20] and **Schmidtova et al.'s survey of human evaluations of hallucination [30]** approach the same problem from two sides. The first organizes definitions, metrics and mitigation across tasks; the second examines how studies define and assess hallucinations, and identifies inconsistent terminology and missing methodological reporting. Its practical lesson is to define source support, world knowledge and error severity separately, then publish annotation instructions and disagreement handling.

Gao and colleagues' RAG survey [31] organizes retrieval-augmented systems into naive, advanced and modular paradigms, and separates context relevance, answer relevance and answer faithfulness. Stages interact: rewriting "What was the limit before the upgrade?" into "What is the temperature limit?" may improve topical retrieval while deleting the time constraint. Preserve the original query and compare the rewritten meaning, not only the retrieval score.

Kwon et al.'s PagedAttention work [32] studies key–value cache allocation and sharing in language-model serving. It addresses wasted memory and the batch sizes that memory permits, and supplies a concrete systems perspective on the cache calculation above. Its reported improvements belong to its workloads and comparisons; measure your own request lengths, concurrency and latency targets before adopting a serving design.

Open problems include contamination of evaluation sets, long-context robustness (a larger context window permits more input; it does not establish that all relevant parts are used reliably), multilingual and low-resource coverage, stable uncertainty, detection of unsupported claims without suppressing useful answers, and the relationship between instruction following and source-grounded correctness. Evaluation models can share errors with generators, especially when both rely on similar training data. A compact specialized model, or even a template, can be the better engineering choice when it satisfies the task with lower latency and simpler review.

Key takeaways

- A decoder-only Transformer factorizes text causally; shift targets explicitly and test that future tokens cannot change earlier logits.
- The GPT line kept one architecture and objective while scaling data, parameters and the way tasks are presented; scaling laws predict loss, not task behavior.
- Perplexity comparisons require compatible tokenization and evaluation positions, and say nothing directly about decoded outputs.
- Greedy and beam search optimize sequence probability; temperature reshapes the distribution; top-k and nucleus sampling cut its tail, with the nucleus adapting to the distribution's shape.
- Generation needs a content contract: selection, planning and realization each fail differently, and extraction still needs mandatory-content checks.
- Overlap metrics, human review and model judges each have blind spots; validate them on the errors that matter, such as reversed negation.
- Retrieval supplies evidence, not authority; relevance, citation validity and claim support are separate conditions.
- Version the complete preprocessing–model–decoding–validation bundle, and treat instruction tuning and preference learning as distinct from factual support.

Exercises

1. Remove the causal mask from the runnable causal model. Which assertion should fail, and why might training loss still look good?

Hint

The changed final token can become visible to earlier positions. Low loss can reward access to information unavailable during generation.

2. Using the decoding code, find a temperature at which plain sampling puts more mass on the four tail tokens than top-k with $k = 5$ ever can. Then construct a logit vector for which the nucleus with $p = 0.9$ is larger than five tokens, and explain when each rule is the better choice.

Hint

Top-k with $k = 5$ can keep at most one tail token. A flat logit vector makes the nucleus grow; a peaked one makes it shrink.

3. Change the toy beam-search probabilities so beam search chooses a sentence that violates an externally supplied instruction. Add a constraint on permitted final sequences and compare its effect with widening the beam.

Hint

Probability search optimizes the distribution's score. A permitted-sequence rule changes what outputs are admissible. Widening a beam need not repair a wrong probability model.

4. Construct two candidate summaries: one with high word overlap but a wrong quantity, and another with lower overlap but correct meaning. Design an annotation form, and a model-judge protocol, that reveal this difference.

Hint

Ask reviewers to identify unsupported quantities and missing required facts before giving an overall usefulness judgment. Keep the source visible and system identity hidden; run the judge in both presentation orders.

5. Construct one preference pair that rewards style but selects the less-supported response. Explain which additional evaluation would expose the problem, and compute the trainable parameter count of a rank-4 LoRA update for a 128-by-256 matrix.

Hint

Keep the evidence fixed and compare a confident answer that adds a fact with a qualified answer that preserves the source's uncertainty. For LoRA, count the two factor matrices separately.

6. Extend the report function with a timestamp and a target-language choice. Reject malformed inputs, preserve unknown status, attach evidence IDs and version the realization rules. Document the tests needed before replacing the template with a learned generator.

Hint

Do not infer an unrecorded timestamp or convert unknown into absent. Test preprocessing parity, number formatting, script requirements, long inputs, timeout behavior and the complete rollback bundle.

References

1. Masato Hagiwara, *Real-World Natural Language Processing: Practical Applications with Deep Learning*, Chapters 5, 8, 9, 10 and 11, Manning, 2021. Publisher ·Training practices ·Deployment
2. Susan McRoy, *Principles of Natural Language Processing*, open textbook, Chapters 6 and 9, 2021. Benchmark chapter ·Generation chapter
3. Ashish Vaswani et al., "Attention Is All You Need," NeurIPS, 2017. Paper
4. Alec Radford, Karthik Narasimhan, Tim Salimans and Ilya Sutskever, "Improving Language Understanding by Generative Pre-Training," OpenAI technical report, 2018. Paper
5. Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei and Ilya Sutskever, "Language Models are Unsupervised Multitask Learners," OpenAI technical report, 2019. Paper
6. Tom B. Brown et al., "Language Models are Few-Shot Learners," NeurIPS, 2020. Paper
7. Jared Kaplan et al., "Scaling Laws for Neural Language Models," arXiv:2001.08361, 2020. arXiv
8. Jordan Hoffmann et al., "Training Compute-Optimal Large Language Models," arXiv:2203.15556, 2022; published as "An empirical analysis of compute-optimal large language model training," NeurIPS, 2022. arXiv
9. Ilya Loshchilov and Frank Hutter, "Decoupled Weight Decay Regularization," ICLR, 2019. Paper
10. Dan Hendrycks and Kevin Gimpel, "Gaussian Error Linear Units (GELUs)," arXiv:1606.08415, 2016, revised 2023. Paper
11. Jimmy Lei Ba, Jamie Ryan Kiros and Geoffrey E. Hinton, "Layer Normalization," arXiv:1607.06450, 2016. Paper
12. Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes and Yejin Choi, "The Curious Case of Neural Text Degeneration," ICLR, 2020. Paper
13. Angela Fan, Mike Lewis and Yann Dauphin, "Hierarchical Neural Story Generation," ACL, 2018. Paper
14. Jade Goldstein and Jaime Carbonell, "Summarization: (1) Using MMR for Diversity-Based Reranking and (2) Evaluating Summaries," TIPSTER Phase III Workshop, 1998. Paper
15. Sam Wiseman, Stuart M. Shieber and Alexander M. Rush, "Challenges in Data-to-Document Generation," EMNLP, 2017. Paper
16. Ilya Sutskever, Oriol Vinyals and Quoc V. Le, "Sequence to Sequence Learning with Neural Networks," NeurIPS, 2014. Paper
17. Chin-Yew Lin, "ROUGE: A Package for Automatic Evaluation of Summaries," Text Summarization Branches Out, 2004. Paper
18. Lianmin Zheng et al., "Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena," NeurIPS Datasets and Benchmarks Track, 2023. Paper
19. Peiyi Wang et al., "Large Language Models are not Fair Evaluators," ACL, 2024. Paper

20. Ziwei Ji et al., “Survey of Hallucination in Natural Language Generation,” ACM Computing Surveys 55, 2023. Survey
21. Joshua Maynez, Shashi Narayan, Bernd Bohnet and Ryan McDonald, “On Faithfulness and Factuality in Abstractive Summarization,” ACL, 2020. Paper
22. Patrick Lewis et al., “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,” NeurIPS, 2020. Paper
23. OWASP GenAI Security Project contributors, “LLM01: Prompt Injection,” OWASP guidance, accessed 2026. Official guidance
24. Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang and Weizhu Chen, “LoRA: Low-Rank Adaptation of Large Language Models,” ICLR, 2022. Author publication page
25. Long Ouyang et al., “Training language models to follow instructions with human feedback,” NeurIPS, 2022. Paper
26. Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D. Manning, Stefano Ermon and Chelsea Finn, “Direct Preference Optimization: Your Language Model is Secretly a Reward Model,” NeurIPS, 2023. Paper
27. Wayne Xin Zhao et al., “A Survey of Large Language Models,” Frontiers of Computer Science, 2026. Updated survey
28. Yupeng Chang et al., “A Survey on Evaluation of Large Language Models,” ACM Transactions on Intelligent Systems and Technology, 2024. Survey
29. Junyi Li, Tianyi Tang, Wayne Xin Zhao, Jian-Yun Nie and Ji-Rong Wen, “Pre-Trained Language Models for Text Generation: A Survey,” ACM Computing Surveys, 2024. Publisher ·Author manuscript
30. Patricia Schmidtova et al., “Do My Eyes Deceive Me? A Survey of Human Evaluations of Hallucinations in NLG,” INLG, 2025. Survey
31. Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang and Haofen Wang, “Retrieval-Augmented Generation for Large Language Models: A Survey,” arXiv:2312.10997, revised 2024. Survey
32. Woosuk Kwon et al., “Efficient Memory Management for Large Language Model Serving with PagedAttention,” SOSP, 2023. Conference ·Paper
33. Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze, *Introduction to Information Retrieval*, Chapters 6, 8 and 11, Cambridge University Press, 2008. Author-hosted book
34. scikit-learn developers, “Text feature extraction,” scikit-learn documentation, accessed 2026. Documentation
35. Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen and Wen-tau Yih, “Dense Passage Retrieval for Open-Domain Question Answering,” EMNLP, 2020. Paper
36. Kalervo Järvelin and Jaana Kekäläinen, “Cumulated Gain-Based Evaluation of IR Techniques,” ACM Transactions on Information Systems 20(4), 422–446, 2002. Publisher

NLP 10 • Pretrained Models: BERT

Understand BERT from WordPiece input to bidirectional encoder, pretrain a tiny BERT-style model with masked language modeling from scratch, and probe what its contextual representations contain.

Online edition: https://blog.shyandram.dev/en/learn/nlp/pretrained-bert/

What you'll learn

- Explain the pretrain-then-fine-tune recipe and where BERT sits next to ELMo and GPT.
- Turn text into BERT input: WordPiece pieces, `[CLS]`/`[SEP]` packing, and summed token, segment and position embeddings.
- State the masked-language-modeling objective exactly (15% selection, 80/10/10 replacement), and the next-sentence-prediction objective with its later critique.
- Pretrain a tiny BERT-style encoder from scratch on a toy corpus, and measure what it learned.
- Distinguish contextual from static embeddings, and probe representations with a control task.
- Place RoBERTa, ALBERT, DistilBERT, ELECTRA, multilingual BERT and Chinese BERT models in one map.

The big picture

Standard Word2Vec features retrieve a fixed vector per word [6]. BERT, short for **Bidirectional Encoder Representations from Transformers**, computes a new representation for each token occurrence by reading its surrounding text [1]. In "the crane lifts steel" and "the crane eats fish," the embedding lookup for `crane` is initially the same; the encoder can use the remaining words to produce different final vectors. The distinction is between a vocabulary entry and a representation of this occurrence.

BERT is learned in two stages. **Pretraining** uses ordinary unlabeled text, where the text supplies its own targets. **Fine-tuning** then adapts the same network, plus a small task head, to a labeled task such as sentiment or entailment. The first stage is expensive and done once; the second is cheap and done per task.

> **Explain like I'm five**
>
> Cover one word in a sentence with your thumb and ask a friend to guess it: "the crane ___ steel." Your friend can read the words on both sides of your thumb, so "lifts" is a good guess. Playing this game millions of times teaches a lot about how words fit together. Afterwards, the friend can learn a new job, like sorting happy and angry messages, from only a few examples, because they already know how sentences work.

Hagiwara's Chapter 9 develops pretrained-model adaptation in a practical setting [18]. This chapter goes inside the model. It reuses the causal language model of NLP 09 as a contrast, the attention block of NLP 07, and the static embeddings of NLP 04. The classification, entailment, tagging and extractive question-answering heads that sit on top of BERT are the subject of NLP 11. Every model here is tiny and trained from scratch; no checkpoint is downloaded.

1. Pretraining, then fine-tuning

Plain version. Learn general knowledge about language from a large amount of unlabeled text, then reuse it for a task with few labels. The question is what pretraining objective produces the most reusable representation.

Three 2018 designs frame BERT. ELMo trained left-to-right and right-to-left LSTM language models and supplied a learned combination of their internal states as contextual word features to existing task models [5]. GPT pretrained a 12-layer decoder-only Transformer as a left-to-right language model and then fine-tuned the whole network on each task [4]. BERT kept GPT's fine-tune-everything recipe but replaced the causal objective with a masked one, so every layer can use both left and right context [1].

The BERT paper argues that the left-to-right constraint is sub-optimal for sentence-level tasks and can be very harmful for token-level tasks such as question answering, where context from both directions matters [1]. Its ablations include a left-to-right model without next sentence prediction, trained and fine-tuned under the same left-only constraint, and compare it with the bidirectional model [1]. The engineering lesson is that the visibility pattern is a modeling choice tied to the task, not a matter of taste.

Pretraining supplies an initialization and a representation function. It does not supply an application's labels. A pump-maintenance classifier still needs examples defining "urgent," and an entity tagger still needs examples identifying the site's equipment names. Fine-tuning then adjusts the encoder and a task head together; freezing the encoder and training only the head is a cheaper alternative with a different adaptation capacity.

2. Masked and conditional objectives

Plain version. Not every pretrained language model predicts the next token. A masked objective uses surrounding context, while a conditional generator predicts an output sequence from a separate input.

BERT's masked language objective hides selected input positions and predicts their original tokens using bidirectional context [1]. A simplified description is:

$$\mathcal{L}_{\text{mask}} = -\frac{1}{|M|} \sum_{i \in M} \log p_\theta(x_i \mid \tilde{\mathbf{x}}).$$

Here M is the set of supervised masked positions; $|M|$ is its size; x_i is the original token; θ denotes model parameters; and $\tilde{\mathbf{x}}$ is the corrupted input sequence. The tilde indicates corruption, not an

independently sampled document. This expression describes the masked prediction component; Section 5 gives the full corruption procedure.

A masked predictor is useful for contextual representations, classification and span-oriented tasks. Its loss is not automatically the log probability of an ordinary left-to-right sequence. Multiplying predictions that each use both sides of the sentence does not generally yield the same normalized joint distribution as a causal factorization. Consequently, masked-objective scores and causal perplexities are not directly interchangeable.

Conditional sequence generation instead writes:

$$p_\theta(\mathbf{y} \mid \mathbf{x}) = \prod_{t=1}^{U} p_\theta(y_t \mid y_{<t}, \mathbf{x}).$$

Here $\mathbf{x}$ is the source sequence; $\mathbf{y}$ is the target sequence; U is target length including any end token; and $y_{<t}$ is the target prefix. The source can be visible in full while the target remains causal. T5 organizes many NLP tasks in a text-to-text framework and studies transfer learning with a Transformer encoder–decoder [27]; NLP 08 develops this family.

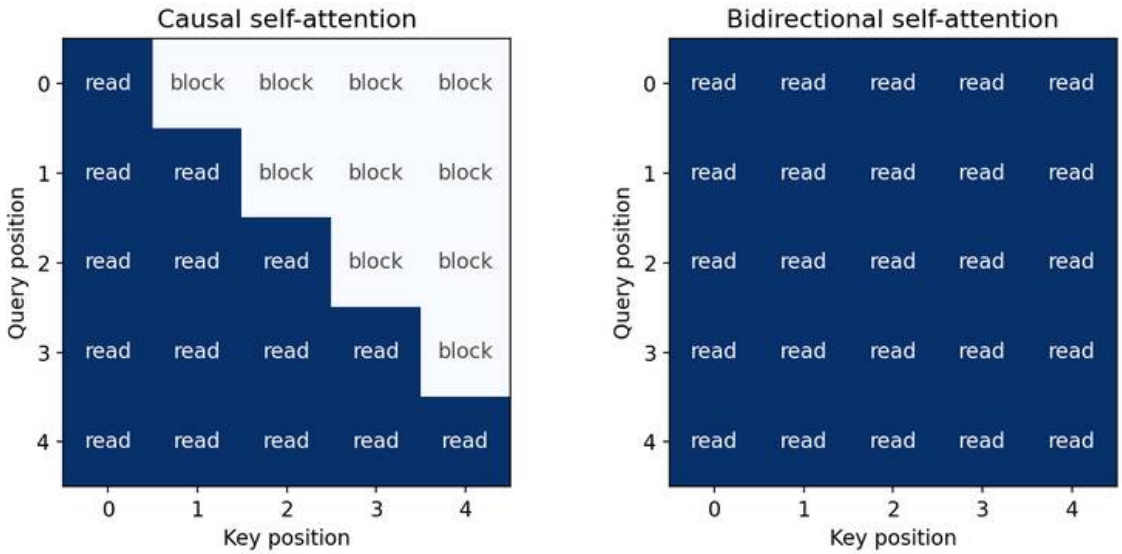


Figure 1 — Visibility is part of the task definition. Rows are querying positions and columns are the positions they may read. BERT uses the right-hand pattern in every layer; padding keys are still blocked.

These distinctions guide model choice. A sentence classifier may use a bidirectional encoder because the complete sentence is available. A streaming continuation must exclude future output tokens. Translation can use the entire source after it arrives, but simultaneous translation needs a more restrictive source visibility contract. The task's available information should determine the mask, rather than choosing a mask because a particular model name is familiar.

3. From text to BERT input

Plain version. BERT never sees words directly. Text is split into vocabulary pieces, wrapped with special markers, and each position receives the sum of three learned vectors.

WordPiece

The original tokenizer first applies text preprocessing, then WordPiece tokenization [2]. BERT uses WordPiece embeddings with a 30,000-token vocabulary [1], a subword method introduced for Google's neural machine translation system [7]. WordPiece splits a word into vocabulary pieces when needed; ## marks a continuation piece in the original vocabulary convention. The original implementation splits each word **greedily, longest match first**: take the longest vocabulary entry that starts the word, then the longest continuation piece, and so on; if some remainder matches nothing, the whole word becomes `[UNK]` [2]. The vocabulary is learned beforehand from a corpus, as BPE merges are in NLP 02.

```
vocab = {'[UNK]', 'the', 'pump', 'over', 'heat', '##heat', '##ed', '##s',
         'un', '##load', 'load', '##ing', '幫', '浦', '過', '熱'}

def wordpiece(word, max_chars=200):
    """Greedy longest-match-first split into vocabulary pieces."""
    if len(word) > max_chars:
        return ['[UNK]']
    pieces, start = [], 0
    while start < len(word):
        end = len(word)
        while end > start:
            piece = word[start:end] if start == 0 else '##' + word[start:end]
            if piece in vocab:
                break
            end -= 1
        if end == start:                    # no piece fits: the whole word is unknown
            return ['[UNK]']
        pieces.append(piece)
        start = end
    return pieces

def basic_split(text):
    """Lowercase, put spaces around CJK ideographs, split on whitespace."""
    out = []
    for ch in text.lower():
        cjk = 0x4E00 <= ord(ch) <= 0x9FFF
        out.append(f' {ch} ' if cjk else ch)
    return ''.join(out).split()

for text in ['The pump overheated', 'unloading pumps', 'the pumpkin', '幫浦過熱']:
    tokens = [p for w in basic_split(text) for p in wordpiece(w)]
    print(f'{text!r:24} -> {tokens}')
assert wordpiece('overheated') == ['over', '##heat', '##ed']
assert wordpiece('pumpkin') == ['[UNK]']
```

Output:

```
'The pump overheated'    -> ['the', 'pump', 'over', '##heat', '##ed']
'unloading pumps'        -> ['un', '##load', '##ing', 'pump', '##s']
'the pumpkin'            -> ['the', '[UNK]']
'幫浦過熱'                   -> ['幫', '浦', '過', '熱']
```

Three consequences follow. A word may occupy several model positions, so word labels and character spans must be mapped to token positions. A word whose remainder is not covered becomes a single `[UNK]`, even though its prefix `pump` is known; that is a property of the greedy rule, not of the language. And Chinese characters are split one per position before WordPiece, as the original multilingual and Chinese models do (Section 11). The `basic_split` here checks only the main CJK Unified Ideographs block; the original code checks several more ranges and also handles accents and punctuation [2].

Pack one text or a text pair

For a single text, pack `[CLS] text [SEP]`. For a pair, pack `[CLS] text A [SEP] text B [SEP]`. `[CLS]` supplies a position whose final state can be read by a sequence classifier; `[SEP]` marks boundaries. Segment IDs distinguish the two parts. They **do not block attention between them**: a query in text A can read text B. This joint encoding is useful when a decision depends on their interaction [2].

Before the encoder, sum three learned vectors:

$$\mathbf{e}_i = E[x_i] + P[i] + S[s_i], \qquad H = \mathrm{Encoder}_\theta(\mathbf{e}_1, \dots, \mathbf{e}_T).$$

Here T is packed sequence length; θ denotes encoder parameters; x_i is the token ID at position i; $s_i \in \{0, 1\}$ is its segment ID; $E \in \mathbb{R}^{|V| \times d}$ is the token table for vocabulary V; $P \in \mathbb{R}^{T_{\max} \times d}$ is the position table with supported length $T_{\max}$; $S \in \mathbb{R}^{2 \times d}$ is the segment table; $\mathbf{e}_i \in \mathbb{R}^d$ is the summed input; and $H \in \mathbb{R}^{T \times d}$ contains contextual output rows $\mathbf{h}_i$. The original implementation normalizes and applies dropout to the summed embeddings [2]. Padding keys are blocked; ordinary text on either side remains visible.

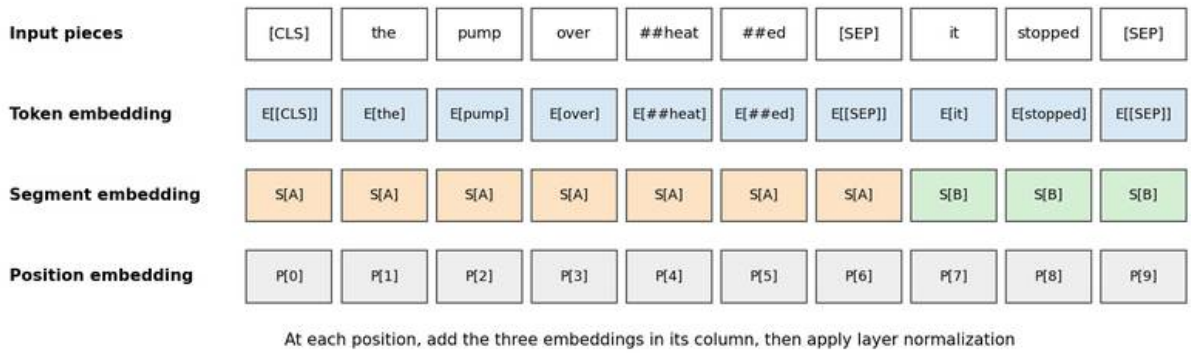


Figure 2 — BERT input for a text pair. Each column is one position; its token, segment (A or B) and position embeddings are added. The position table is learned, so it fixes a maximum input length.

Unlike the fixed sinusoids of the original Transformer, BERT's position vectors are learned rows [2]. A learned position table imposes a maximum supported position; a longer input is an invalid lookup, not an ordinary longer sequence, and must be truncated or split into windows. Embedding rows also have meaning only relative to the model's token IDs. Swapping in a different tokenizer without adapting weights can turn apparently valid numbers into unrelated symbols.

4. The encoder

Plain version. BERT is the encoder half of a Transformer: a stack of identical blocks, each letting every position read every other position and then transforming each position separately.

BERT is a stack of Transformer **encoder** blocks [1, 3]. Each block mixes information with bidirectional self-attention, then transforms each position with a feedforward network; residual connections and layer normalization support the stack [9]. "Bidirectional" means both left and right input positions are available throughout the encoder. It does not mean training two separate causal models and joining their outputs, as ELMo does [5]. NLP 07 derives multi-head attention; the only BERT-specific choice is the attention mask of Figure 1.

The paper reports two sizes. Writing L for the number of blocks, H for the hidden width and A for attention heads, BERT-Base has $L = 12$, $H = 768$, $A = 12$ and 110M parameters, and BERT-Large has $L = 24$, $H = 1024$, $A = 16$ and 340M parameters [1]. BERT-Base was chosen to match the size of the original GPT for comparison [1]. The feedforward layers use the GELU activation [8, 1]. Pretraining used BooksCorpus and English Wikipedia, and the paper stresses using a document-level corpus so that long contiguous sequences are available [1].

Think of the outputs as a table with one row per input token, rather than a single sentence vector. A token-labeling head reads those rows individually. A sequence head reads the first row through a pooling transformation. The same encoder can support both because its output preserves token positions; NLP 11 builds those heads.

5. What BERT learns before task labels are available

Plain version. Masked language modeling turns raw text into a fill-in-the-blank exercise. Next sentence prediction adds a yes/no question about whether two text segments belong together.

Masked language modeling

Masked language modeling (MLM) learns from ordinary text: the original tokens provide their own targets. For example, `the crane [MASK] steel` asks the model to predict the removed token using words on both sides. The prediction loss updates the encoder as well as the vocabulary head, so the final token vectors become useful inputs for later supervised tasks.

Original BERT selects 15% of WordPiece positions as prediction targets. Among selected positions, it inserts `[MASK]` 80% of the time, a random token 10%, and leaves the original token unchanged 10% [1]. Selection and replacement are different: **every selected position contributes a target**, even when its visible token is unchanged. Other positions supply context without contributing MLM loss. This mixed replacement reduces the difference between pretraining inputs and ordinary unmasked task inputs, where `[MASK]` never appears.

The price of this design is sample efficiency: only the selected positions produce a loss. ELECTRA's authors make exactly this point, noting that MLM learns from only the masked subset of tokens per example [14]. The original implementation also chose masks once when the data was prepared; RoBERTa compared that static masking with **dynamic masking**, which draws a new pattern every time a sequence is used, and found dynamic masking comparable or slightly better [11]. The tiny model in Section 7 masks dynamically.

Next sentence prediction and its critique

Original BERT also trains **next sentence prediction** (NSP) [1]: a text pair is either consecutive corpus text or a pair whose second segment is sampled from the corpus, with equal sampling probability. The pooled `[CLS]` state predicts this binary label. NSP predicts a relationship label; it does not generate the next sentence. The original pretraining objective combines MLM and NSP, and the paper's ablation reported that removing NSP hurt performance on QNLI, MNLI and SQuAD 1.1 [1].

Later work disagreed. RoBERTa's replication study found that removing the NSP loss **matches or slightly improves** downstream performance when inputs are filled with contiguous text, and suggested that the original ablation may have removed the loss while keeping the segment-pair input format [11]. ALBERT's authors conjectured that NSP is too easy because it conflates topic prediction with coherence prediction: a randomly sampled second segment usually differs in topic. They replaced it with **sentence-order prediction**, which uses two consecutive segments from the same document either in order or swapped, so topic no longer gives the answer away [12]. The general lesson is to inspect a checkpoint's actual objectives rather than assume every later encoder keeps every original design choice.

Online only: BERT — Pre-training of Deep Bidirectional Transformers for Language Understanding — https://blog.shyandram.dev/en/learn/nlp/pretrained-bert/papers/bert–pre-training-of-deep-bidirectional-transformers-for-language-understanding/

6. A runnable bidirectional encoder

Plain version. Before pretraining a whole model, isolate the two properties that make it BERT-like: changing a later word changes an earlier word's representation, and padding changes nothing.

The following CPU example trains MLM on four invented sentences. Position 2 is always selected and replaced with `[MASK]`, making the visible right-hand verb useful for distinguishing `robot` from `sensor`. This deliberately simpler corruption policy isolates the mechanism; the model is

randomly initialized, has one small encoder block, and uses a linear vocabulary head rather than reproducing original BERT's full head. It uses layer normalization, GELU and AdamW [8–10].

```
import torch
from torch import nn
from torch.nn import functional as F

torch.manual_seed(23)
torch.set_num_threads(1)
vocab = ['[PAD]', '[CLS]', '[SEP]', '[MASK]', 'the', 'a',
         'robot', 'sensor', 'moves', 'measures']
index = {word: i for i, word in enumerate(vocab)}
rows = ['the robot moves', 'the sensor measures',
        'a robot moves', 'a sensor measures']
ids = torch.tensor([[index[w] for w in ['[CLS]'] + s.split()
                     + ['[SEP]', '[PAD]']] for s in rows])
blocked = ids.eq(index['[PAD]'])
corrupted = ids.clone()
corrupted[:, 2] = index['[MASK]']
labels = torch.full_like(ids, -100)
labels[:, 2] = ids[:, 2]

class TinyEncoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.word = nn.Embedding(len(vocab), 16)
        self.position = nn.Embedding(6, 16)
        self.segment = nn.Embedding(2, 16)
        self.embed_norm = nn.LayerNorm(16)
        self.layer = nn.TransformerEncoderLayer(
            16, 2, 32, dropout=0.0, activation='gelu', batch_first=True)
        self.mlm = nn.Linear(16, len(vocab))
        self.pool = nn.Sequential(nn.Linear(16, 16), nn.Tanh())
        self.classifier = nn.Linear(16, 3)
        self.tagger = nn.Linear(16, 3)
        self.span = nn.Linear(16, 2)
    def forward(self, tokens, padding):
        positions = torch.arange(tokens.shape[1])
        segments = torch.zeros_like(tokens)  # single segment here
        h = self.embed_norm(self.word(tokens) + self.position(positions)
                            + self.segment(segments))
        return self.layer(h, src_key_padding_mask=padding)

model = TinyEncoder()
optimizer = torch.optim.AdamW(model.parameters(), lr=0.01)
def objective():
    logits = model.mlm(model(corrupted, blocked))
    return F.cross_entropy(logits.reshape(-1, len(vocab)), labels.reshape(-1))
initial = objective().item()
for _ in range(100):
    optimizer.zero_grad()
    loss = objective()
    loss.backward()
    optimizer.step()
model.eval()
```

```
with torch.no_grad():
    final = objective().item()
    h = model(ids, blocked)
    changed = ids.clone()
    changed[0, 3] = index['measures']
    context_gap = (model(changed, blocked)[0, 2] - h[0, 2]).abs().max().item()
    changed_padding = ids.clone()
    changed_padding[blocked] = index['robot']
    pad_gap = (model(changed_padding, blocked)[~blocked] - h[~blocked]).abs().max().item()
    shapes = [model.classifier(model.pool(h[:, 0])).shape,
              model.tagger(h).shape, model.span(h).shape]
assert final < initial and context_gap > 1e-4 and pad_gap < 1e-6
print('masked loss:', round(initial, 3), '->', round(final, 3))
print('context changes representation:', context_gap > 1e-4)
print('padding leaves real tokens unchanged:', pad_gap < 1e-6)
print('class / tag / span shapes:', [tuple(s) for s in shapes])
```

Output:

```
masked loss: 2.223 -> 0.001
context changes representation: True
padding leaves real tokens unchanged: True
class / tag / span shapes: [(4, 3), (4, 6, 3), (4, 6, 2)]
```

The masked loss falls from `2.223` to `0.001`. This measures fitting those four examples. Changing the word after `robot` changes its final representation, whereas changing the value stored at blocked padding positions leaves real-token outputs unchanged. A causal model must reject the first kind of future influence (the perturbation test of NLP 09); a bidirectional encoder deliberately allows it.

The task heads print shapes `(4, 3)`, `(4, 6, 3)` and `(4, 6, 2)`: four sequence decisions over three categories, six token decisions per example, and two boundary scores per token. **Those heads are untrained.** Only the encoder and MLM head receive gradients in this example. To build an application, supply its labels, optimize the corresponding head and encoder, and evaluate held-out task outputs, as NLP 11 does.

There are two masks to keep separate. The encoder's padding mask controls readable keys. The `-100` labels tell cross-entropy which positions to ignore; they do not remove those positions from attention. In the demonstration, all real words may provide context, but only the selected noun contributes MLM loss.

7. Pretraining a tiny BERT from scratch

Plain version. Now apply the real corruption recipe to a small artificial language, train for a few seconds, and check three things: how well the model fills in held-out blanks, whether one word gets different vectors in different contexts, and what simple classifiers can read from each layer.

The toy corpus comes from a small grammar. The subject constrains the verb and the verb constrains the object, so a masked word is often predictable from **both** sides. The word `crane` is deliberately ambiguous: it can `lift` (a machine) or `eat` (a bird). Of the 128 possible sentences, 40 are held out. The corruption function implements Section 5 exactly: 15% selection, then 80/10/10 replacement, with a fresh draw on every step.

```
import itertools, random
from collections import Counter
import numpy as np
import torch
from torch import nn
from torch.nn import functional as F
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score

random.seed(0); torch.manual_seed(0); torch.set_num_threads(1)

# 1. A toy corpus: the subject constrains the verb, the verb constrains the object.
frames = {'robot': ['lifts', 'moves'], 'crane': ['lifts', 'eats'],
          'technician': ['repairs', 'moves'], 'heron': ['eats', 'sees']}
objects = {'lifts': ['steel', 'box'], 'moves': ['box', 'cart'],
           'eats': ['fish', 'bread'], 'repairs': ['pump', 'cart'], 'sees': ['fish', 'boat']}
sentences = [f'{d1} {a}{s} {v} {d2} {o}'.split()
             for s in frames for v in frames[s] for o in objects[v]
             for d1, d2, a in itertools.product(['the', 'a'], ['the', 'a'], ['', 'old '])]
random.shuffle(sentences)
test, train = sentences[:40], sentences[40:]
special = ['[PAD]', '[CLS]', '[SEP]', '[MASK]']
words = sorted({w for s in sentences for w in s})
vocab = special + words
index = {w: i for i, w in enumerate(vocab)}
first_word = len(special)
L = 2 + max(len(s) for s in sentences)

def encode(rows):
    ids = torch.zeros(len(rows), L, dtype=torch.long)
    for r, s in enumerate(rows):
        seq = [index['[CLS]']] + [index[w] for w in s] + [index['[SEP]']]
        ids[r, :len(seq)] = torch.tensor(seq)
    return ids
train_ids, test_ids = encode(train), encode(test)

# 2. BERT-style corruption: select 15% of real word positions; of those,
#    80% -> [MASK], 10% -> random word, 10% -> unchanged. All selected are targets.
def corrupt(ids, generator):
    real = ids >= first_word
    selected = (torch.rand(ids.shape, generator=generator) < 0.15) & real
    labels = torch.where(selected, ids, torch.full_like(ids, -100))
    u = torch.rand(ids.shape, generator=generator)
    out = ids.clone()
    out[selected & (u < 0.8)] = index['[MASK]']
    swap = selected & (u >= 0.8) & (u < 0.9)
    out[swap] = torch.randint(first_word, len(vocab), (int(swap.sum()),),
     ↪  generator=generator)
```

```
    return out, labels

class TinyBERT(nn.Module):
    def __init__(self, d=32, layers=2):
        super().__init__()
        self.word, self.position = nn.Embedding(len(vocab), d), nn.Embedding(L, d)
        self.segment, self.norm = nn.Embedding(2, d), nn.LayerNorm(d)
        self.blocks = nn.ModuleList(nn.TransformerEncoderLayer(
            d, 4, 4 * d, dropout=0.0, activation='gelu', batch_first=True)
            for _ in range(layers))
        self.mlm = nn.Linear(d, len(vocab))
    def forward(self, ids):
        pad = ids.eq(index['[PAD]'])
        h = self.norm(self.word(ids) + self.position(torch.arange(ids.shape[1]))
                      + self.segment(torch.zeros_like(ids)))
        states = [h]
        for block in self.blocks:
            h = block(h, src_key_padding_mask=pad)
            states.append(h)
        return states                                # embeddings, layer 1, layer 2

model = TinyBERT()
optimizer = torch.optim.AdamW(model.parameters(), lr=3e-3)
g = torch.Generator().manual_seed(1)
for step in range(600):                              # a fresh mask every step
    inputs, labels = corrupt(train_ids, g)
    logits = model.mlm(model(inputs)[-1])
    loss = F.cross_entropy(logits.reshape(-1, len(vocab)), labels.reshape(-1))
    optimizer.zero_grad(); loss.backward(); optimizer.step()
model.eval()

# 3. Held-out masked-word accuracy: mask each word position in turn.
def masked_hits(ids, rows_text):
    by_kind = {}
    with torch.no_grad():
        for r, s in enumerate(rows_text):
            for i, w in enumerate(s):
                x = ids[r:r + 1].clone(); x[0, i + 1] = index['[MASK]']
                guess = vocab[model.mlm(model(x)[-1][0, i + 1]).argmax()]
                kind = 'determiner/adjective' if w in ('the', 'a', 'old') else 'noun/verb'
                by_kind.setdefault(kind, []).append(guess == w)
    return {k: round(float(np.mean(v)), 3) for k, v in sorted(by_kind.items())}
counts = torch.bincount(train_ids[train_ids >= first_word], minlength=len(vocab))
real_test = test_ids[test_ids >= first_word]
print('final training MLM loss:', round(loss.item(), 3))
print('held-out masked accuracy:', masked_hits(test_ids, test))
print('most-frequent-word baseline:', vocab[int(counts.argmax())],
      round((real_test == counts.argmax()).float().mean().item(), 3))
# Upper reference: the best guess given every other word, read off the grammar.
def oracle(text):
    by_kind = {}
    for s in text:
        for i, w in enumerate(s):
            fits = Counter(t[i] for t in sentences if len(t) == len(s)
                           and all(a == b for j, (a, b) in enumerate(zip(t, s)) if j != i))
            kind = 'determiner/adjective' if w in ('the', 'a', 'old') else 'noun/verb'
```

```
            by_kind.setdefault(kind, []).append(fits[w] / sum(fits.values()))
    return {k: round(float(np.mean(v)), 3) for k, v in sorted(by_kind.items())}
print('expected accuracy of an ideal guesser:', oracle(test))

# 4. One word, two contexts: identical input row, different output rows.
with torch.no_grad():
    pair = encode([['the', 'crane', 'lifts', 'the', 'steel'],
                   ['the', 'crane', 'eats', 'the', 'fish']])
    cos = [F.cosine_similarity(s[0, 2], s[1, 2], dim=0).item() for s in model(pair)]
print('cosine of the two "crane" vectors by layer:', [round(c, 3) for c in cos])

# 5. Probes. (a) Part of speech vs a control task that gives each word type a
#    random label; (b) the sense of "crane" (machine vs bird) from its own vector.
pos = {w: 'DET' if w in ('the', 'a') else 'ADJ' if w == 'old' else
       'VERB' if w in objects else 'NOUN' for w in words}
control = {w: random.randint(0, 3) for w in words}
crane = [s for s in sentences if 'crane' in s]
crane_ids = encode(crane)
sense = np.array([int('eats' in s) for s in crane])
def rows(ids, text, layer):
    with torch.no_grad():
        h = model(ids)[layer]
    return np.array([h[r, i + 1].numpy() for r, s in enumerate(text) for i in range(len(s))])
for layer in range(3):
    Xtr, Xte = rows(train_ids, train, layer), rows(test_ids, test, layer)
    acc = []
    for table in (pos, control):
        ytr = [table[w] for s in train for w in s]; yte = [table[w] for s in test for w in s]
        acc.append(LogisticRegression(max_iter=3000).fit(Xtr, ytr).score(Xte, yte))
    h = model(crane_ids)[layer].detach()[:, :, :]
    Xc = np.array([h[r, s.index('crane') + 1].numpy() for r, s in enumerate(crane)])
    acc.append(cross_val_score(LogisticRegression(max_iter=3000), Xc, sense, cv=4).mean())
    print(f'layer {layer}: POS {acc[0]:.2f}  control {acc[1]:.2f}  crane sense {acc[2]:.2f}')
assert cos[0] > 0.999 and cos[-1] < cos[0]
```

Output (about six seconds on one CPU thread):

```
final training MLM loss: 0.453
held-out masked accuracy: {'determiner/adjective': 0.418, 'noun/verb': 0.492}
most-frequent-word baseline: the 0.17
expected accuracy of an ideal guesser: {'determiner/adjective': 0.592, 'noun/verb': 0.633}
cosine of the two "crane" vectors by layer: [1.0, 0.819, 0.639]
layer 0: POS 1.00  control 1.00  crane sense 0.38
layer 1: POS 1.00  control 1.00  crane sense 1.00
layer 2: POS 1.00  control 1.00  crane sense 1.00
```

Read the masked-word numbers against two references. Always guessing the most frequent word, `the`, is right 17% of the time. An **ideal guesser** that knows the grammar exactly, and therefore the true distribution of each blank given every other word, would be right 63% of the time on content words, not 100%: after `the robot ___ the box`, both `lifts` and `moves` are grammatical. The model reaches 49% on content words from 88 training sentences. That gap is the honest measure

of what it has and has not learned. Many blanks have no single answer, which is why MLM accuracy is never a direct measure of understanding.

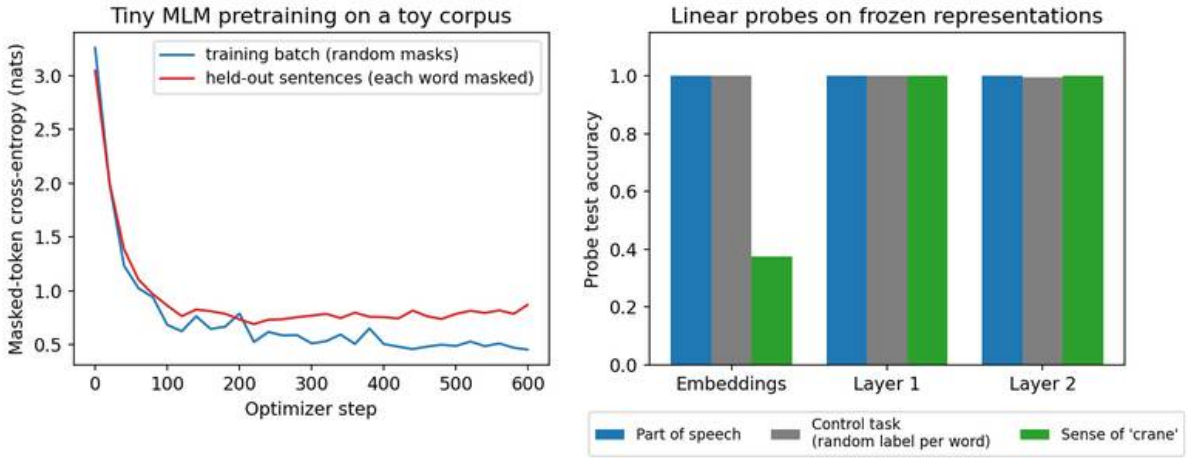


Figure 3 — The tiny MLM run. Left: training loss on randomly masked batches and held-out loss with each word masked in turn; the gap is overfitting to 88 sentences. Right: linear probes on frozen representations from each layer (Section 9 explains why the control task matters).

The left panel of Figure 3 shows the held-out loss flattening, then drifting up, while the training loss keeps falling. The same script trained for more steps lowers training loss further but makes held-out masked accuracy worse, so the 600-step budget was set by watching held-out loss, not training loss. That is ordinary model selection, and it applies to pretraining as much as to fine-tuning.

8. Contextual versus static embeddings

Plain version. A static embedding gives a word one vector everywhere. A contextual encoder gives each occurrence its own vector, and the difference grows with depth.

A Word2Vec or PPMI vector, as built in NLP 04, is a single row of a table: the bird and the machine share one `crane` vector, which must average their contexts. The printed cosines above show the contextual alternative in miniature. At the embedding layer, the two `crane` vectors are identical (cosine 1.0) because the token and its position are the same. After one encoder block the cosine is 0.819, and after two it is 0.639: the right-hand context `lifts the steel` or `eats the fish` has been mixed in.

Ethayarajh measured this effect in real models. In ELMo, BERT and GPT-2, representations of the same word in different contexts are still more similar than those of two different words, but this self-similarity is much lower in upper layers, and on average less than 5% of the variance in a word's contextual representations can be explained by a static embedding for that word [19]. The same study found that contextual representations are not isotropic in any layer: vectors occupy a narrow region of the space [19]. A practical consequence is that raw cosine similarity

between BERT vectors is not automatically a meaningful semantic similarity. Sentence-level similarity needs its own training objective, which NLP 04 and NLP 11 discuss.

Static vectors still have uses: they are cheap, need no encoder at inference time and are easy to inspect. Contextual vectors are preferable when the same surface form carries different meanings or roles, which is most of the time in real text.

9. Probing what the representations contain

Plain version. A probe is a small classifier trained on frozen representations to predict a linguistic property. If it succeeds, the property is recoverable from the vectors. Whether it was the representation or the probe that did the work is a separate question.

Tenney, Das and Pavlick probed BERT layer by layer and found that the regions responsible for part-of-speech tagging, parsing, named entities, semantic roles and coreference appear in roughly the order of the classical NLP pipeline, with later layers sometimes revising earlier decisions [20]. Hewitt and Manning's **structural probe** looks for a linear transformation under which squared distances between word vectors encode distances in the parse tree, and found such transformations for ELMo and BERT but not for baselines [21]. Clark and colleagues analyzed attention instead of vectors and found heads that attend to delimiter tokens, to fixed positional offsets, and to direct objects, determiners and coreferent mentions [23].

Hewitt and Liang added the essential control. A **control task** assigns each word type a random label, so it can be solved only by memorizing word identities. A good probe should be **selective**: high accuracy on the linguistic task and low accuracy on its control [22]. Without that comparison, a high probe accuracy may show that the probe is powerful, not that the representation encodes the property.

Our tiny run illustrates both sides. Part of speech is perfectly recoverable at every layer, but so is a random label per word, because in this grammar part of speech is a function of the word itself. The probe's success therefore says nothing beyond "the word is identifiable," and its selectivity is zero. The `crane` sense probe is different. It is at chance or below (0.38, from 32 sentences in four folds) at the embedding layer, where both senses share one vector, and perfect after one encoder block. That property requires context, so the layer-wise rise is evidence that the encoder moved the information into the `crane` vector. Even so, a probe shows that information is **available** to a linear reader, not that the model **uses** it for its predictions.

10. The BERT family

Plain version. After BERT, several papers changed one part of the recipe at a time: how long to train, how to share parameters, how to compress, and what to predict.

Model	What changed	Reported motivation
RoBERTa [11]	Longer training with bigger batches on more data, NSP removed, longer sequences, dynamic masking	A replication study finding BERT significantly undertrained
ALBERT [12]	Factorized embedding parameterization, cross-layer parameter sharing, sentence-order prediction	Lower memory and faster training; a coherence objective instead of NSP
DistilBERT [13]	Knowledge distillation during pretraining into a smaller student, with a triple loss	40% smaller, retaining 97% of BERT's language understanding and 60% faster, as reported by its authors
ELECTRA [14]	Replaced-token detection: a small generator fills masked positions, and the encoder classifies every token as original or replaced	Learning from all input tokens instead of only the masked subset

ALBERT's factorized embedding separates vocabulary size from hidden size: instead of one $|V|\times H$ table it uses a $|V| \times E$ table followed by an $E \times H$ projection, with E much smaller than H [12]. Its authors argue that WordPiece embeddings are meant to be context-independent while hidden states are context-dependent, so the two sizes need not match [12]. ELECTRA's discriminator is the model kept for fine-tuning; the generator is discarded after pretraining [14].

These variants change parameter counts, compute budgets and objectives together, so their published comparisons belong to their own experimental settings. When choosing one, check its tokenizer, maximum length, objectives and training data, and measure on the task you care about.

11. Multilingual and Chinese BERT

Plain version. One encoder can be pretrained on many languages at once with a shared vocabulary. Chinese needs a decision about units, because it is not written with spaces between words.

The original release includes a multilingual BERT-Base trained on the Wikipedias of 104 languages with a shared 110k WordPiece vocabulary and no marker telling the model which language it is reading [2]. Pires, Schlinger and Garrette found that this model transfers surprisingly well zero-shot, where a task is fine-tuned in one language and evaluated in another, including across scripts, and that transfer works best between typologically similar languages; they also found systematic deficiencies for certain language pairs [15]. XLM-R later trained a Transformer masked language model on one hundred languages with far more filtered CommonCrawl data and reported large gains over multilingual BERT on cross-lingual benchmarks, especially for low-resource languages [16]. Its analysis of the trade-off between positive transfer and capacity dilution is a reminder that adding languages to a fixed-size model is not free [16].

For Chinese, the original multilingual model adds spaces around every character in the CJK ideograph ranges before WordPiece, so Chinese is effectively character-tokenized [2]; the `basic_split` function in Section 3 imitates this rule. The release also provides a Chinese-only BERT-Base that covers both Simplified and Traditional Chinese [2]. Character positions avoid word-segmentation errors, but masking a single character of a multi-character word such as 幫浦 lets

the model recover it from its partner character rather than from sentence context. Cui and colleagues introduced **whole word masking** for Chinese BERT: a word segmenter identifies word boundaries, and all characters of a selected word are masked together, while the input is still tokenized by characters [17]. Their MacBERT model replaces `[MASK]` with similar words chosen by word-vector similarity, reducing the gap between pretraining and fine-tuning inputs [17].

For Traditional Chinese text from Taiwan, check what the pretraining corpus actually contained, not only the language label. A model whose data is dominated by Simplified Chinese can tokenize Traditional characters correctly and still be weaker on regional vocabulary such as 資料 versus other usages, or on Taiwanese terminology in a domain. That is a data question to measure with in-domain examples, not something the architecture decides.

Modern view

Rogers, Kovaleva and Rumshisky's primer [24] surveys over 150 studies of BERT: what linguistic and world knowledge it captures and where, how its attention heads and layers behave, modifications to its objectives and architecture, overparameterization and compression. Its overall message matches Section 9: many properties are recoverable, but recoverable is not the same as used, and probing results depend on the probe.

Encoders have not disappeared in the era of large decoder-only models. **ModernBERT [25]**, published at ACL 2025, is a 2024 encoder-only model that brings recent architectural and efficiency optimizations to the BERT design; its authors describe encoders as the workhorse of many production retrieval and classification pipelines and report training on 2 trillion tokens with a native sequence length of 8,192. **Zhao and colleagues' survey [26]** places masked encoders within the broader history of pretrained language models and notes that causal decoders have been widely adopted as the architecture of existing large language models; NLP 09 shows why one causal objective can serve both scoring and generation.

A practical 2026 summary: use a bidirectional encoder when the whole input is available and the output is a label, a span or a vector for retrieval, and when latency and cost matter; use a decoder-only model when the output is open-ended text. Open problems include faithful interpretation beyond probing, robust multilingual coverage for low-resource languages and scripts, long-document encoding beyond learned position limits, and evaluating how much a pretrained representation, rather than the fine-tuning data, explains a task result.

Key takeaways

- BERT is a Transformer encoder pretrained without labels and then fine-tuned with a small task head; pretraining supplies a representation, not task labels.
- Input is WordPiece pieces packed as `[CLS] A [SEP] B [SEP]`, with summed token, segment and learned position embeddings; segments do not block attention.
- MLM selects 15% of positions and replaces them 80/10/10; every selected position is a target, and masked scores are not causal perplexities.

- NSP was later found unnecessary (RoBERTa) or replaced by sentence-order prediction (ALBERT).
- A tiny BERT pretrained from scratch shows the mechanisms: held-out blanks are filled better than a frequency baseline but below the grammar's own ceiling, and an ambiguous word's vectors diverge with depth.
- Contextual vectors differ by occurrence; static vectors do not. Raw cosine between contextual vectors is not automatically a semantic similarity.
- Probes need control tasks; a probe shows information is available, not that the model uses it.
- Multilingual and Chinese BERT models make specific unit choices (shared WordPiece, per-character CJK splitting, whole word masking) that affect what is learned.

Exercises

1. In the tiny BERT, replace dynamic masking with a single static mask drawn once before training. Compare held-out masked accuracy and the loss curves.

   Hint

   Draw `corrupt(train_ids, g)` once outside the loop. With a static mask, each sentence always hides the same positions, so many (sentence, position) targets are never seen.

2. Add next sentence prediction to the toy corpus: build pairs of two sentences joined by `[SEP]` with segment IDs, where half the second sentences are random. Then build a sentence-order version. Which one can be solved from topic cues alone in your corpus?

   Hint

   Make two "documents" that use different vocabularies. A random second sentence then usually comes from the other document, so NSP is easy; swapped order within one document is not.

3. Extend the WordPiece vocabulary so that `pumpkin` is not `[UNK]`, without adding `pumpkin` itself. Then find a word for which greedy longest-match gives a different split from the one with the fewest pieces.

   Hint

   Adding `##kin` is enough for the first part. For the second, include a long prefix that leaves an awkward remainder, and a shorter prefix that leads to a clean split.

4. Design a probing task on the toy corpus for which part-of-speech style control results would show **positive** selectivity. What property must depend on context rather than on word identity?

   Hint

   Subject versus object role of a noun is one option, but check first whether position alone already reveals it in this grammar. If it does, add sentence variants that break the link.

5. Replace the MLM head with an ELECTRA-style replaced-token detector: corrupt positions with random words instead of `[MASK]`, and train a binary classifier on every position. Compare how many positions produce a loss per batch.

Hint

ELECTRA uses a small trained generator; random replacements make detection much easier, so say what your simplification changes. Count labeled positions with and without masking.

6. Turn the bidirectional example into a token classifier. If one annotated word splits into three pieces, define which positions contribute to loss and how you reconstruct the word label.

Hint

Decide between first-piece supervision and an explicitly defined all-piece label mapping. Ignore padding and special markers, train the tag head, and map the output back to the original text.

References

1. Jacob Devlin, Ming-Wei Chang, Kenton Lee and Kristina Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," NAACL, 2019. Paper
2. Google Research, *BERT: original implementation and released models*, software, 2018. Encoder and pooler ·Tokenizer ·Multilingual and Chinese models
3. Ashish Vaswani et al., "Attention Is All You Need," NeurIPS, 2017. Paper
4. Alec Radford, Karthik Narasimhan, Tim Salimans and Ilya Sutskever, "Improving Language Understanding by Generative Pre-Training," OpenAI technical report, 2018. Paper
5. Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee and Luke Zettlemoyer, "Deep Contextualized Word Representations," NAACL, 2018. Paper
6. Tomas Mikolov, Kai Chen, Greg Corrado and Jeffrey Dean, "Efficient Estimation of Word Representations in Vector Space," ICLR Workshop, 2013. Paper
7. Yonghui Wu et al., "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation," arXiv:1609.08144, 2016. Paper
8. Dan Hendrycks and Kevin Gimpel, "Gaussian Error Linear Units (GELUs)," arXiv:1606.08415, 2016, revised 2023. Paper
9. Jimmy Lei Ba, Jamie Ryan Kiros and Geoffrey E. Hinton, "Layer Normalization," arXiv:1607.06450, 2016. Paper
10. Ilya Loshchilov and Frank Hutter, "Decoupled Weight Decay Regularization," ICLR, 2019. Paper
11. Yinhan Liu et al., "RoBERTa: A Robustly Optimized BERT Pretraining Approach," arXiv:1907.11692, 2019. Paper
12. Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma and Radu Soricut, "ALBERT: A Lite BERT for Self-supervised Learning of Language Representations," ICLR, 2020. Paper

13. Victor Sanh, Lysandre Debut, Julien Chaumond and Thomas Wolf, “DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter,” 5th Workshop on Energy Efficient Machine Learning and Cognitive Computing at NeurIPS, 2019. Paper
14. Kevin Clark, Minh-Thang Luong, Quoc V. Le and Christopher D. Manning, “ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators,” ICLR, 2020. Paper
15. Telmo Pires, Eva Schlinger and Dan Garrette, “How Multilingual is Multilingual BERT?,” ACL, 2019. Paper
16. Alexis Conneau et al., “Unsupervised Cross-lingual Representation Learning at Scale,” ACL, 2020. Paper
17. Yiming Cui, Wanxiang Che, Ting Liu, Bing Qin and Ziqing Yang, “Pre-Training With Whole Word Masking for Chinese BERT,” IEEE/ACM Transactions on Audio, Speech, and Language Processing 29, 2021. Publisher ·Author manuscript
18. Masato Hagiwara, *Real-World Natural Language Processing: Practical Applications with Deep Learning*, Chapter 9, Manning, 2021. Publisher
19. Kawin Ethayarajh, “How Contextual are Contextualized Word Representations? Comparing the Geometry of BERT, ELMo, and GPT-2 Embeddings,” EMNLP-IJCNLP, 2019. Paper
20. Ian Tenney, Dipanjan Das and Ellie Pavlick, “BERT Rediscovers the Classical NLP Pipeline,” ACL, 2019. Paper
21. John Hewitt and Christopher D. Manning, “A Structural Probe for Finding Syntax in Word Representations,” NAACL, 2019. Paper
22. John Hewitt and Percy Liang, “Designing and Interpreting Probes with Control Tasks,” EMNLP-IJCNLP, 2019. Paper
23. Kevin Clark, Urvashi Khandelwal, Omer Levy and Christopher D. Manning, “What Does BERT Look at? An Analysis of BERT’s Attention,” BlackboxNLP Workshop, 2019. Paper
24. Anna Rogers, Olga Kovaleva and Anna Rumshisky, “A Primer in BERTology: What We Know About How BERT Works,” Transactions of the Association for Computational Linguistics 8, 2020. Paper
25. Benjamin Warner et al., “Smarter, Better, Faster, Longer: A Modern Bidirectional Encoder for Fast, Memory Efficient, and Long Context Finetuning and Inference,” ACL, 2025. Paper
26. Wayne Xin Zhao et al., “A Survey of Large Language Models,” Frontiers of Computer Science, 2026. Updated survey
27. Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li and Peter J. Liu, “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer,” JMLR 21(140), 2020. Paper

NLP 11 • Classification with BERT (Sentiment Analysis, Natural Language Inference)

Fine-tune a pretrained bidirectional encoder for single-text and text-pair decisions, read span answers from passage tokens, and evaluate labels, probabilities and shortcuts honestly.

Online edition: https://blog.shyandram.dev/en/learn/nlp/classification-bert/

What you'll learn

- Choose the output head that matches the decision: one label, one label per token, or a span inside a passage.
- Fine-tune a pretrained encoder for sentiment analysis and for premise–hypothesis inference, and know what the standard datasets (SST, IMDB, SNLI, MultiNLI, SQuAD) actually contain.
- Recognize annotation artifacts, hypothesis-only baselines and challenge sets, and why a high in-distribution score can hide a shortcut.
- Set learning rate, epochs, warmup, class weights and decision thresholds deliberately, and know why fine-tuning can be unstable.
- Evaluate with accuracy, macro-F1, calibration and coverage, and separate relevant evidence from sufficient evidence.
- Run a tiny encoder that is pretrained from scratch with masked language modeling, then fine-tuned for synthetic sentiment and inference tasks on a CPU.

The big picture

NLP 10 built BERT: a bidirectional encoder pretrained by predicting masked tokens. Pretraining gives a representation function, not an application. A support team still has to say what "urgent" means, and an inference dataset still has to say when one sentence follows from another. This tutorial is about that second step: attaching a small output layer to the encoder, training both on labeled examples, and checking whether the result solves the intended task rather than a convenient imitation of it.

> **Explain like I'm five**
>
> Imagine a friend who has read thousands of books but has never sorted movie reviews. You show them twenty reviews marked "liked it" or "didn't like it." Because they already understand words like "dull" and "wonderful," they learn the sorting job quickly. But if every "didn't like it" example you showed happened to contain the word "not," your friend might simply look for "not." Fine-tuning is the short lesson; good evaluation checks what the friend actually learned.

Two families of decisions dominate this chapter. **Single-text classification** assigns one label to one input: positive or negative sentiment, a topic, a ticket route. **Text-pair classification** labels the relationship between two inputs: does the hypothesis follow from the premise, contradict it, or neither? Extractive question answering sits next to them: instead of one label, the model chooses where an answer starts and ends inside a passage. All three reuse the same pretrained encoder; only the head and the training labels change [3].

Hagiwara's Chapter 9 develops transfer learning with pretrained encoders for exactly these applications [1], and McRoy's chapters on benchmark tasks and on information extraction and argumentation supply the linguistic side [2]. Throughout, keep one question in view: *what evidence would convince a skeptic that the model learned the task, not the dataset?*

BERT applications: choose the output head

Plain version. A task head turns the encoder's token vectors into the output you need. Start by deciding whether the answer is one class, one label per token, or a span inside a passage.

Application	Input	Supervised output
Sentiment, topic or support-ticket classification	One text	One category from the pooled `[CLS]` state
Entailment, paraphrase or query–document relevance	Two texts packed together	A relationship label or relevance score
Entity recognition or other token labeling	One tokenized text	A label at each supervised token position
Extractive question answering	Question and passage	Start/end positions in the passage
Semantic retrieval with Sentence-BERT	Texts encoded separately	Comparable vectors learned with a sentence-level objective

Sequence and token tasks use the representations described in the BERT paper [3]; the original classification and span implementations make their output shapes explicit [4]. Sentence-BERT adds training for sentence comparisons [20]. The following formulas explain what those choices compute.

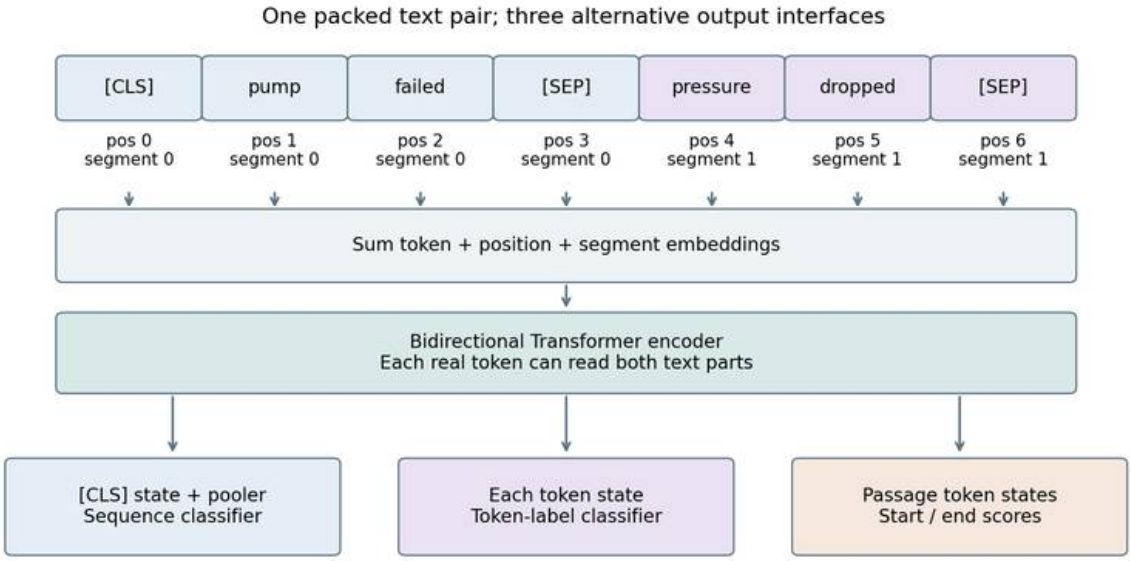


Figure 1 — A shared BERT encoder supports different supervised outputs. Segment A/B labels identify text parts; they do not prevent cross-segment attention. The three heads illustrate alternative tasks, not predictions from a trained model.

Sequence classification and text-pair decisions

For a classifier with K classes, one common BERT interface is:

$$\mathbf{c} = \tanh(W_p \mathbf{h}_{\text{CLS}} + \mathbf{b}_p), \qquad p(y = k \mid \mathbf{x}) = \text{softmax}(W_c \mathbf{c} + \mathbf{b}_c)_k.$$

Here $\mathbf{x}$ is the packed token sequence; $\mathbf{h}_{\text{CLS}} \in \mathbb{R}^d$ is the first contextual row; $W_p \in \mathbb{R}^{d \times d}$ and $\mathbf{b}_p \in \mathbb{R}^d$ form the pooler; $\mathbf{c} \in \mathbb{R}^d$ is its output; $W_c \in \mathbb{R}^{K \times d}$ and $\mathbf{b}_c \in \mathbb{R}^K$ form the classifier; and $y = k$ denotes class k. Cross-entropy for the labeled category updates both head and encoder during end-to-end fine-tuning [4].

For "the pump failed," a ticket classifier could predict an incident category. For a premise–hypothesis pair, the same output shape can represent entailment, contradiction or neither. The labels come from the task dataset, not from the next-sentence-prediction label vocabulary used in pretraining. Text-pair self-attention lets tokens in one text inspect the other before the final decision.

Token labeling

Attach a classifier to each contextual row:

$$p(y_i = k \mid \mathbf{x}) = \operatorname{softmax}(W_t \mathbf{h}_i + \mathbf{b}_t)_k, \qquad \mathcal{L}_{\text{tag}} = -\frac{1}{|J|} \sum_{i \in J} \log p(y_i^* \mid \mathbf{x}).$$

Here i is a token position and K is the number of labels; $W_t \in \mathbb{R}^{K \times d}$ and $\mathbf{b}_t \in \mathbb{R}^K$ are shared across positions; y_i is the token label; y_i^* is its annotated value; and J is the nonempty set of positions included in supervision. A device name split into multiple WordPieces requires a policy: supervise its first piece and ignore the others, or map the labels consistently across pieces. Exclude padding and special markers from J. At inference, map predicted pieces back to word or character spans, as developed in NLP 06. Semantic role labeling in NLP 12 reuses the same head with a predicate marker.

Extractive question answering

Suppose the passage says “Pump B stopped at noon,” and the question asks “Which pump stopped?” Pack the question and passage together, encode them, then score possible answer boundaries:

$$a_i = \mathbf{w}_s^\top \mathbf{h}_i, \quad b_i = \mathbf{w}_e^\top \mathbf{h}_i, \qquad \mathcal{L}_{\text{span}} = -\log p_s(i^*) - \log p_e(j^*).$$

Here i, j are candidate start/end positions; $\mathbf{w}_s, \mathbf{w}_e \in \mathbb{R}^d$ are learned start/end vectors; a_i, b_i are boundary scores; p_s, p_e are their softmax distributions over allowed positions; and i^*, j^* are annotated answer boundaries. Decode a valid span maximizing $a_i + b_j$ with $i \le j$ and a declared maximum length [4]. Question positions and padding cannot be returned as passage answers. If a no-answer marker is included, train that option and select its decision threshold on development data.

The BERT paper introduces exactly two new vectors for this task, a start vector and an end vector, and computes each boundary probability as a dot product with the final token states followed by a softmax over passage positions [3]. Figure 2 shows why decoding must be joint rather than two independent argmaxes: the best start and the best end must form a legal span. In the synthetic scores, the highest-scoring legal span is “Pump B.”

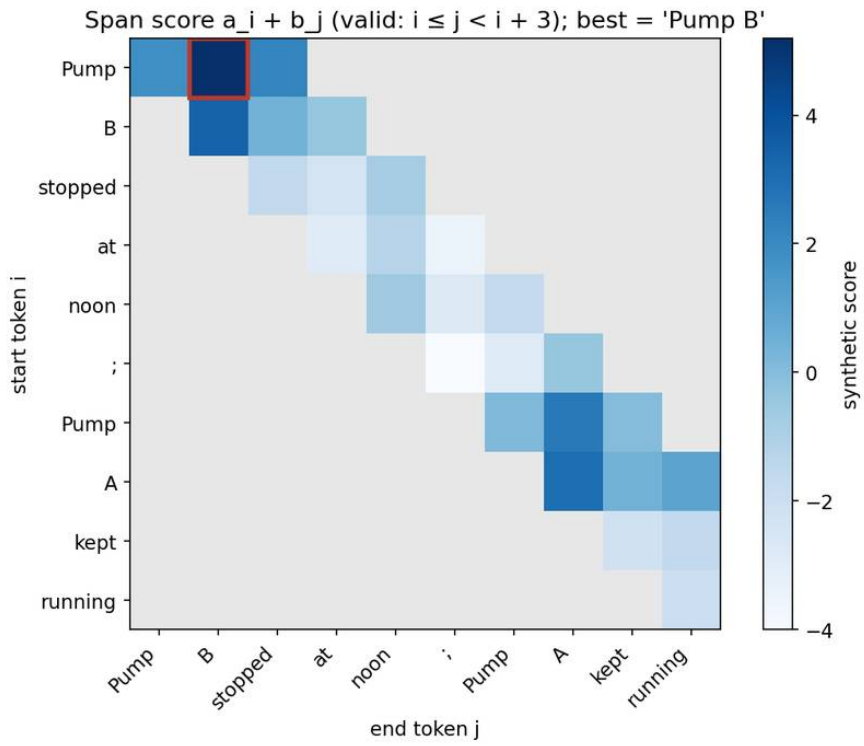


Figure 2 — Span decoding over start + end scores. Grey cells violate $i \le j$ or the length limit. Scores are invented for illustration.

The returned answer is passage text such as "Pump B." This setup does not generate an explanation or find a missing passage. Retrieval-augmented generation in NLP 09 handles finding passages; the sections on answerability below handle deciding whether a found passage is enough.

Retrieval: joint scoring or reusable vectors?

A joint BERT relevance model packs a query and candidate together, so the candidate's representation depends on that query. It must run again for another pair. A sentence embedding model instead encodes documents separately and permits caching. Sentence-BERT trains this second interface with paired/triplet sentence objectives [20]. Raw pretrained `[CLS]` vectors or arbitrary mean pooling do not automatically provide a well-trained cosine retrieval space.

For a practical system, one design is to retrieve candidates using separately stored vectors, then jointly score a smaller set of query–candidate pairs. Evaluate the retrieval stage's candidate recall separately from the final ranker. NLP 04 explains pooling and similarity; NLP 09 develops the full retrieval pipeline.

Single-text classification: sentiment analysis

Plain version. Sentiment analysis asks whether a text expresses a positive or negative opinion. It looks easy, but negation, contrast and mixed opinions mean that counting nice and nasty words is not enough.

Write the fine-tuning objective over n labeled examples:

$$\mathcal{L}(\theta, \phi) = -\frac{1}{n} \sum_{i=1}^{n} \log p_{\theta,\phi}(y_i \mid \mathbf{x}_i).$$

Here θ denotes the encoder parameters, initialized from pretraining; $\phi = (W_p, \mathbf{b}_p, W_c, \mathbf{b}_c)$ denotes the head parameters, initialized randomly; $\mathbf{x}_i$ is the i-th packed text; y_i is its label; and $p_{\theta,\phi}$ is the classifier distribution defined above. Full fine-tuning updates both θ and ϕ. Freezing θ and training only ϕ is cheaper, but it can only reuse whatever distinctions the frozen features already make.

Two benchmark datasets and what they measure

The **Stanford Sentiment Treebank** (**SST**) contains 11,855 single sentences from movie reviews; the authors parsed them and had annotators label all 215,154 unique phrases in those parse trees [5]. Phrase-level labels let researchers study how sentiment composes: "not" combined with "good" is a phrase with its own label. The paper reports both fine-grained (five-class) and binary sentence-level results [5]. GLUE later adopted the two-way split as **SST-2**, listing about 67k training and 1.8k test examples and evaluating it with accuracy [14]. The original SST models were recursive networks over parse trees; ML 04 explains that architecture family.

The **IMDB** review dataset from Maas et al. contains 50,000 reviews divided evenly into 25,000 training and 25,000 test reviews, plus 50,000 unlabeled reviews; the labeled tasks are balanced two-class problems [6]. Its unit is a whole review, not a sentence. Reviews are often longer than a short encoder window, so a document classifier needs a stated truncation or chunking policy. Keep the head, the tail, or pool predictions over windows — each policy answers a slightly different question, because a review frequently changes tone near its end.

These two datasets illustrate a recurring issue: the **unit of supervision** defines the task. A sentence-level sentiment model trained on isolated opinions may fail on a whole conversation containing quotations, corrections and several speakers. If deployment includes previous turns, annotation and training should include the same permitted context. Conversely, giving annotators access to the final resolution while the model only sees the first message creates an impossible target. NLP 03 sets out this prediction contract in detail for count-based classifiers; the same contract applies after BERT.

What a contextual encoder adds, and what it does not

A bag-of-words classifier gives "good" the same weight in "the battery is good" and "the battery is not good." A contextual encoder computes a different vector for "good" in each sentence, so the

head can, in principle, learn the flip. NLP 03 runs a controlled negation test with count features and shows how sparse evidence leaves such a flip underdetermined. Context makes the distinction *representable*; the labeled data must still make it *learnable*. Phrase identity is also different from scope: distinguishing "not every repair succeeded" from "every repair failed" requires a compositional distinction that adjacent-word features do not supply, and that a fine-tuned encoder learns only if the training data contains it.

Treat sentiment labels as an annotation decision, too. A review can praise the screen and criticize the battery. Single-label sentiment forces such a review into one class; aspect-based sentiment instead assigns opinions to targets. Neither is "the true" task. Choose the one whose output the application consumes.

Text pairs: natural language inference

Plain version. A text can support a conclusion without proving that conclusion true in the world. Keep textual entailment and evidence quality separate.

Natural language inference (NLI) classifies a premise–hypothesis pair as entailment, contradiction or neutral under the annotation convention. If the premise is "Mina repaired a sensor," the hypothesis "Someone repaired something" follows under an ordinary interpretation. "Mina repaired every sensor" does not follow, and "Mina repaired no sensor" conflicts with the premise.

For BERT, pack the pair as `[CLS] premise [SEP] hypothesis [SEP]` with segment IDs 0 and 1, then apply the sequence head with $K = 3$:

$$\hat{y} = \operatorname*{argmax}_{k \in \{\mathrm{E,N,C}\}} p_{\theta,\phi}(y = k \mid [\mathrm{CLS}]\,\mathbf{u}\,[\mathrm{SEP}]\,\mathbf{v}\,[\mathrm{SEP}]).$$

Here $\mathbf{u}$ is the premise token sequence, $\mathbf{v}$ is the hypothesis token sequence, $\mathrm{E}, \mathrm{N}, \mathrm{C}$ denote entailment, neutral and contradiction, and the other symbols are as in the sequence-classification head. Because self-attention crosses the `[SEP]` boundary, every hypothesis token can attend to every premise token in every layer. That joint encoding lets the decision depend on token-level comparisons between the two texts — and it is also the reason a pair model cannot cache one side.

SNLI and MultiNLI

The **Stanford Natural Language Inference (SNLI)** corpus contains 570,152 human-written sentence pairs labeled for entailment, contradiction and semantic independence (neutral) [7]. Annotators were shown an image caption as the premise and wrote hypotheses. The paper discusses a subtle labeling choice: whether two sentences should be assumed to describe the same event. Without that assumption, many apparent contradictions would be neutral. The corpus fixes one convention so that each pair receives one label [7]. That is exactly the kind of decision you must make — and document — for your own inference data.

MultiNLI extends the design to 433k examples from ten genres of written and spoken English. Only five genres appear in training, so the development and test sets have a **matched** section from

the training genres and a **mismatched** section from the others [8]. The split turns domain shift into a measured quantity rather than an afterthought.

Annotation artifacts and hypothesis-only baselines

Crowd workers who are asked to write a contradiction often reach for the same tools: negation words, or a different activity. Gururangan et al. showed that a simple text classifier that sees **only the hypothesis** predicts the label for about 67% of SNLI and 53% of MultiNLI, and that phenomena such as negation and vagueness correlate strongly with particular classes [9]. Poliak et al. ran hypothesis-only baselines across ten NLI datasets and found that they significantly outperform the majority-class baseline on a number of them [10]. A model that ignores the premise cannot be performing inference, so its score measures dataset regularities.

McCoy, Pavlick and Linzen took the opposite direction: they built **HANS**, a controlled set where three syntactic heuristics — lexical overlap, subsequence and constituent — give the wrong answer, and found that models trained on MNLI, including BERT, perform very poorly on it [11]. Together these studies motivate two habits:

1. **Always run a partial-input baseline.** If a hypothesis-only (or premise-only, or question-only) model does well, part of the benchmark score is not about the relationship you care about.
2. **Build challenge pairs that flip the cue.** Hold the vocabulary fixed and change the one distinction that should matter: actor reversal, negation in the premise rather than the hypothesis, possibility versus assertion, allegation versus fact, one versus all.

An inference model predicts an annotation relationship. It does not independently verify the premise. If the premise itself is fabricated, an entailed hypothesis can also be fabricated. A retrieval-assisted application therefore needs both source selection and an answer-support check, which the question-answering sections below develop.

Argumentation mining can identify a claim, a proposed supporting statement and an expressed attack relation [2]. “The device is reliable because it passed one test” contains a recognizable argument structure. Identifying that structure does not establish that the test is representative or that the conclusion is valid. The application may store the stated reason while marking its evidential strength as unassessed.

Useful challenge pairs change one crucial distinction at a time. Fluent paraphrases are useful additional variation, but they should preserve the intended labels. A synthetic paraphrase that changes scope or removes attribution must receive a new annotation rather than inherit the original record automatically.

Online only: Sentence-BERT — Sentence Embeddings using Siamese BERT-Networks — https://blog.shyandram.dev/en/learn/nlp/classification-bert/papers/sentence-bert–sentence-embeddings-using-siamese-bert-networks/

Extractive QA as a pair task: answers need support

Plain version. SQuAD-style question answering asks the model to point at the answer inside a given passage. The harder practical question is whether the passage contains a supported answer at all.

The **Stanford Question Answering Dataset** (**SQuAD**) contains 107,785 question–answer pairs written by crowd workers on 536 Wikipedia articles; every answer is a span of the corresponding paragraph [12]. Its two metrics both ignore punctuation and the articles *a*, *an* and *the*: **exact match** is the percentage of predictions equal to one of the reference answers, and **F1** measures token overlap between prediction and reference, averaged over questions [12]. Partial credit matters because annotators disagree mostly about non-essential words at span edges.

SQuAD 2.0 adds over 50,000 unanswerable questions written adversarially to look similar to answerable ones; a system must also decide when the paragraph supports no answer and abstain [13]. The BERT paper handles this by treating an unanswerable question as a span that starts and ends at `[CLS]`. At prediction time it compares the null score $s_{\text{null}} = \mathbf{w}_s^\top \mathbf{h}_{\text{CLS}} + \mathbf{w}_e^\top \mathbf{h}_{\text{CLS}}$ with the best non-null span score $\hat{s} = \max_{j \geq i}(a_i + b_j)$, and predicts an answer only when $\hat{s} > s_{\text{null}} + \tau$ [3]. Here τ is a threshold selected on the development set to maximize F1. This one threshold converts a span model into a selective answerer.

A local evidence-backed answerer

We can demonstrate the answer contract without a trained span model. The answerer below handles one explicit question family and returns a source interval. It ranks four synthetic passages with TF–IDF, then answers only a narrow temperature-limit question for Pump A. It preserves one-character words so that the device identifier A remains a feature. The function is not a general classifier for question intent; its caller must route the declared question family to it.

```
import re
from sklearn.feature_extraction.text import TfidfVectorizer

corpus = {
    'temperature-v2': 'Pump A temperature limit is 80 C. This policy is current.',
    'shutdown-v1': 'Pump A shutdown follows a vibration alarm.',
    'label-v1': 'A technician replaced the cabinet label.',
    'sensor-v1': 'Sensor B reports humidity once per hour.',
}
ids = list(corpus)
vectorizer = TfidfVectorizer(ngram_range=(1, 2), token_pattern=r"(?u)\b\w+\b")
D = vectorizer.fit_transform([corpus[key] for key in ids])
def rank(query):
    score = (D @ vectorizer.transform([query]).T).toarray().ravel()
    order = sorted(range(len(ids)), key=lambda i: (-score[i], ids[i]))
    return [(ids[i], float(score[i])) for i in order]

def answer_temperature(query):
    if not re.search(r'\bpump\s+a\b', query, flags=re.I):
```

```
        return {'answer': None, 'reason': 'unsupported entity'}
    key, score = rank(query)[0]
    text = corpus[key]
    pattern = r'Pump A temperature limit is (\d+) C\.'
    match = re.search(pattern, text)
    if not match:
        return {'answer': None, 'reason': 'no matching evidence'}
    evidence = text[match.start():match.end()]
    result = {'answer': f'{match.group(1)} C', 'document': key,
              'start': match.start(), 'end': match.end(), 'evidence': evidence}
    assert text[result['start']:result['end']] == result['evidence']
    return result
print(answer_temperature('What is the Pump A temperature limit?'))
print(answer_temperature('What is the Pump Z temperature limit?'))
```

The supported Pump A answer returns 80 C, its document version and the character interval `0–33`. A Pump Z question returns `unsupported entity` even though temperature-related passages exist. A span head trained on SQuAD-style data replaces the regular expression with learned boundary scores, but the surrounding contract — entity check, evidence interval, verified offsets, explicit abstention — should stay.

The explicit entity check prevents a high lexical score from transferring Pump A's limit to an unsupported device. The evidence expression also requires the source to state the expected relation, not merely contain a number. A more flexible system needs a richer entity and relation check, including scope, units and applicable dates. Expand those capabilities with labeled tests rather than weakening the check until every question receives an answer.

The answerer searches only the top passage. It can therefore abstain when the correct evidence appears lower in the ranking. That limitation is intentional and visible. A later version could inspect several candidates under a fixed budget, while recording which candidate supplied the final answer. Measure the effect on both supported answers and false acceptance.

The function stores an evidence slice and verifies the offsets. This catches serialization mistakes, but not every semantic mistake. A quoted sentence can still describe the wrong entity, an outdated policy or a hypothetical case. Exact quotation is an important mechanical check; the support relation remains a separate part of the contract.

Relevance is not answerability

A passage can discuss the topic and still lack the requested fact. A system should have a meaningful way to return insufficient evidence.

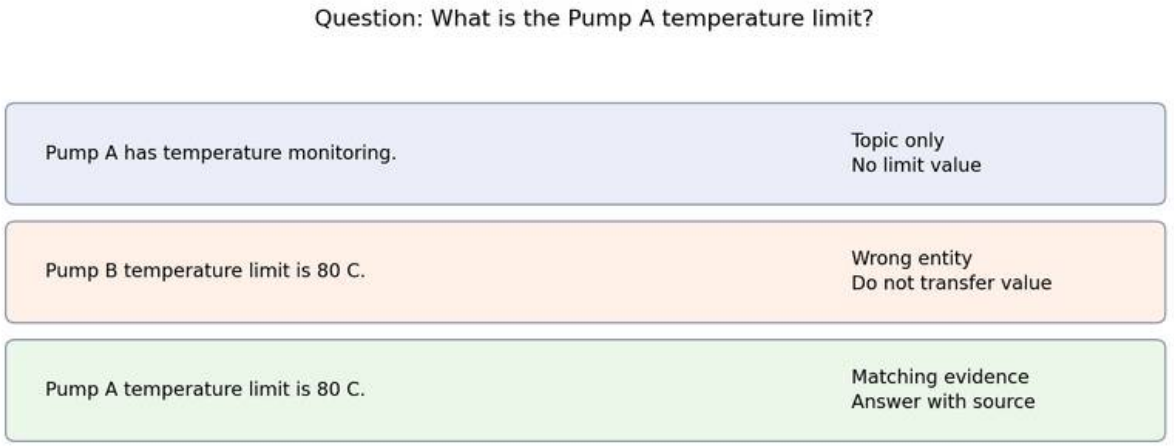


Figure 3 — Topic overlap, wrong-entity evidence and complete support require different decisions. Scores alone do not define answerability.

For a narrow question, define the required fields before evaluating a system. The temperature example needs device identity, a limit relation, a value and a unit. It may also need an effective date or operating condition. Missing any required field can make the available evidence insufficient even if the passage ranks first.

An abstention policy decides when to withhold an answer or ask for clarification. If confidence scores are used, select their threshold on development data with the intended costs of a wrong answer and unnecessary review — exactly the role of τ in the SQuAD 2.0 head. A raw similarity score is not a calibrated correctness probability. Its scale can change when the corpus, tokenizer or embedding model changes.

Report both coverage and quality among accepted answers:

$$\text{coverage} = \frac{N_{\text{accepted}}}{N_{\text{all}}}, \qquad \text{accepted accuracy} = \frac{N_{\text{correct, accepted}}}{N_{\text{accepted}}}.$$

Here N_{all} is the number of evaluated questions; N_{accepted} is the count receiving an answer; and $N_{\text{correct, accepted}}$ counts accepted answers judged correct under the task contract. Accepted accuracy is undefined when no answer is accepted, so report that condition rather than presenting zero acceptance as a successful accuracy result.

Coverage must include questions without available answers. Otherwise a system that always answers can look good on a collection designed to contain an answer everywhere — the weakness SQuAD 2.0 was built to expose [13]. Include wrong entities, missing values, conflicting sources and unsupported time scopes. These cases test whether the model recognizes the limits of evidence instead of merely producing a likely span.

Diagnose failures with controlled substitutions

Replace one imperfect stage with a known-correct input to discover where the final answer fails. Keep these diagnostic results separate from end-to-end deployment scores.

First provide the answerer with the annotated supporting passage directly. If it still produces the wrong value, retrieval is not the immediate cause. Inspect entity matching, unit handling, scope and realization. If it succeeds with gold evidence but fails with retrieved evidence, inspect candidate recall and the selected passage order. This comparison localizes a bottleneck without pretending that perfect evidence will be available at deployment.

Next preserve the correct passage but add irrelevant or conflicting passages. A system that fails only after extra context arrives has a context-selection or evidence-integration problem. Vary one property at a time: a different device identifier, an obsolete version, a quoted allegation or an unsupported command. Label the intended behavior before running the experiment so the interpretation does not shift to excuse the output.

A supported answer requires a chain of successful decisions. For questions whose applicable evidence is in the corpus, write:

$$P(R \cap E \cap G) = P(R)P(E \mid R)P(G \mid R, E).$$

Here P denotes probability, R is the event that sufficient evidence survives retrieval, E is the event that the evidence is interpreted correctly, and G is the event that the output faithfully realizes the interpretation. The expression is the probability chain rule, not an independence assumption. Multiplying unrelated unconditional stage scores is not equivalent: reader accuracy on hand-picked easy passages may differ from reader accuracy on the retriever's actual candidates.

This constructed evaluation makes the conditioning visible:

```
import numpy as np

retrieved = np.arange(100) < 80
interpreted = np.arange(100) < 72
realized = np.arange(100) < 68
r = retrieved.mean()
e = interpreted[retrieved].mean()
g = realized[retrieved & interpreted].mean()
complete = (retrieved & interpreted & realized).mean()
print('Conditional stage rates:', round(r, 3), round(e, 3), round(g, 3))
print('Complete supported answers:', round(complete, 3))
assert np.isclose(r * e * g, complete)
```

The rates are `0.8`, `0.9` and `0.944`, while complete success is `0.68`. These are synthetic outcomes, not reported system performance. Individually respectable component results need not produce equally strong complete answers. Finally retain realistic query distributions: a diagnostic suite deliberately overrepresents difficult cases, while a representative held-out set estimates ordinary behavior. Report both with their purposes stated, and do not mix them into one aggregate number.

Benchmarks: GLUE and SuperGLUE

Plain version. A benchmark suite runs one pretrained model through many small tasks and summarizes them. It is good for comparing general-purpose encoders and poor at telling you whether your application works.

GLUE collects nine English understanding tasks — single-sentence tasks (acceptability and SST-2 sentiment), similarity and paraphrase tasks, and inference tasks including MNLI — together with a diagnostic dataset for specific linguistic phenomena and an online leaderboard [14]. None of its datasets was created from scratch for the benchmark, and some keep test labels private so that the leaderboard is used fairly [14]. Metrics differ by task: accuracy for SST-2, accuracy and F1 for paraphrase tasks, Pearson and Spearman correlations for sentence similarity, and the Matthews correlation coefficient for the unbalanced acceptability task [14]. The BERT paper reported its main transfer results on GLUE, SQuAD and a commonsense-inference set [3].

SuperGLUE keeps GLUE's design — a leaderboard around eight tasks, a single summary number and analysis tools — but selects more difficult tasks, including yes/no question answering over passages and three-class entailment on embedded clauses [15].

Read a suite average carefully. It mixes tasks with different sizes, metrics and noise levels. A gain can come from one small task. It says nothing about artifacts unless partial-input baselines and challenge sets are reported, and nothing about your domain unless your domain is in it. Use suites to choose a starting checkpoint; use your own held-out data to choose a system.

Fine-tuning practice

Plain version. Fine-tuning is short supervised training starting from good weights. The main risks are a learning rate that destroys what pretraining learned, too few or too many updates, unlucky random seeds, and data whose class balance or label meaning differs from deployment.

Learning rate, epochs and schedule

The BERT authors kept dropout at 0.1 and reported that the following ranges worked well across their tasks: batch size 16 or 32; Adam learning rate 5×10^{-5}, 3×10^{-5} or 2×10^{-5}; and 2, 3 or 4 epochs [3]. They also observed that large datasets (100k+ labeled examples) were far less sensitive to these choices than small datasets [3]. The released classification script defaults to a learning rate of 5×10^{-5}, three epochs, a maximum sequence length of 128 and a warmup proportion of 0.1 [4]. Its optimizer schedule is:

$$\eta_t = \begin{cases} \eta_{\max}\, t/t_w, & t < t_w, \\ \eta_{\max}\,(1 - t/T), & t \geq t_w. \end{cases}$$

Here η_t is the learning rate at update step t; $\eta_{\max}$ is the configured peak rate; T is the total number of training steps; and t_w is the number of warmup steps, a fixed fraction of T. The same implementation applies weight decay of 0.01 except to layer-normalization and bias parameters, and

clips the global gradient norm at 1.0 [4]. AdamW-style decoupled weight decay is the standard name for this optimizer family [26].

Why so small a learning rate? Each update moves pretrained weights that encode useful distinctions. A rate suitable for training a randomly initialized network can erase them in a few hundred steps, after which the model is effectively learning from the small labeled set alone.

Instability and random restarts

For BERT-LARGE, the original authors found fine-tuning sometimes unstable on small datasets; they ran several random restarts, which changed data shuffling and classifier initialization while keeping the same pretrained checkpoint, and selected the best model on the development set [3]. Mosbach, Andriushchenko and Klakow analyzed this instability for BERT, RoBERTa and ALBERT on GLUE tasks. They attribute failed runs to optimization difficulties early in training, characterized by vanishing gradients, and the remaining variance to generalization differences between runs with almost identical training loss [16]. They did not observe the instability for BERT-BASE. Their guidelines for small datasets are to use small learning rates **with** Adam's bias correction, and to train for considerably more iterations, close to zero training loss [16].

The practical lesson: report the spread over several seeds, not only the best seed, and select by development data only. A single lucky run is not a method.

Class imbalance

Real label distributions are rarely balanced. Class weighting changes how strongly each training example contributes:

$$\mathcal{L}_{\text{weighted}} = -\frac{1}{n}\sum_{i=1}^{n} \alpha_{y_i} \log p_\theta(y_i \mid d_i).$$

Here $\mathcal{L}_{\text{weighted}}$ is weighted cross-entropy, n is training-example count, i indexes examples, d_i is the document, y_i is its annotated class, $\alpha_{y_i} > 0$ is the chosen weight for that class and p_θ is the classifier distribution. The weighting policy changes the training objective; it does not manufacture missing linguistic variety.

Weighting, oversampling and threshold changes answer different problems. Weighting changes parameter learning. Oversampling changes the presentation frequency of examples, sometimes repeatedly exposing the same few minority examples. Thresholding changes the action policy after scores are computed. Hagiwara's development discussion covers imbalance and training choices [1]. Evaluate any intervention on a realistic prevalence distribution rather than on an artificially balanced final test.

Calibration

A classifier can choose the right label while giving unreliable probabilities. Calibration asks whether predicted confidence agrees with observed correctness over comparable examples. A model can preserve its argmax decisions while changing confidence. Guo et al. study this distinction and post-training temperature scaling [17]:

$$\tilde{p}(y = k \mid \mathbf{x}) = \operatorname{softmax}(\mathbf{z}/T_c)_k, \qquad \mathrm{ECE} = \sum_{b=1}^{B} \frac{|S_b|}{n} \left| \operatorname{acc}(S_b) - \operatorname{conf}(S_b) \right|.$$

Here $\mathbf{z}$ is the logit vector, $T_c > 0$ is a single temperature fit by minimizing negative log-likelihood on held-out development data, and $\tilde{p}$ is the rescaled distribution; for the expected calibration error ECE, examples are grouped into B confidence bins S_b, n is the number of examples, $\operatorname{acc}(S_b)$ is the accuracy within bin b and $\operatorname{conf}(S_b)$ is its mean predicted confidence. Because dividing all logits by one positive number preserves their order, temperature scaling cannot change accuracy.

Desai and Durrett measured BERT and RoBERTa on NLI, paraphrase detection and commonsense reasoning. Used out of the box, the pretrained models were calibrated in-domain and had lower out-of-domain calibration error than the baselines they compared; temperature scaling reduced in-domain error further, and label smoothing helped out of domain [18]. Their result is a property of those tasks and models, not a guarantee for your fine-tuned checkpoint. Measure calibration on your own development data whenever probabilities drive an action such as abstention.

Evaluation: accuracy, macro-F1 and selective decisions

Plain version. Accuracy counts correct answers. Macro-F1 makes every class count equally, so a rare class cannot be ignored. Selective metrics ask how good the accepted answers are when the system may abstain.

For one class treated as positive:

$$P = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}, \qquad R = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}, \qquad F_1 = \frac{2PR}{P + R}.$$

Here P is precision, R is recall, TP counts true positives, FP counts false positives, FN counts false negatives and F_1 is their harmonic mean. A zero denominator requires an explicitly chosen convention. Macro F1 averages per-class F1 values equally; support-weighted averaging gives more influence to common classes. The metric documentation specifies these choices [19].

Accuracy is adequate for balanced tasks such as SST-2 or IMDB [6], [14]. For three-class NLI, report per-class F1 as well: a model can reach decent accuracy while confusing neutral with contradiction. For skewed tasks, macro-F1 or the Matthews correlation makes failures on a rare category visible, but neither encodes the actual cost of those failures. Missing an urgent ticket and misrouting a routine receipt can have different consequences. Write a cost policy separately rather than treating a convenient score as a complete utility measure.

For selective prediction:

$$\text{coverage} = \frac{1}{n}\sum_{i=1}^{n} a_i, \qquad \text{risk}_{\text{accepted}} = \frac{\sum_{i=1}^{n} a_i \mathbf{1}[\hat{y}_i \neq y_i]}{\sum_{i=1}^{n} a_i}.$$

Here n is evaluated-example count, $a_i \in \{0, 1\}$ indicates acceptance of example i, $\hat{y}_i$ is its prediction, y_i is its reference label, and $\mathbf{1}[\cdot]$ is one when its condition holds and zero otherwise. Accepted risk is undefined when no example is accepted. A risk-versus-coverage curve describes the tradeoff over thresholds; choose the deployed threshold using validation rather than final test labels.

Then read the errors. Create an error table with text, gold label, predicted label, probability, language, length, source group and an analyst's explanation. Useful error categories include annotation ambiguity, missing context, negation, unseen vocabulary, multi-topic input, quotation and genuinely out-of-scope text. These categories suggest different repairs. More training examples can help missing vocabulary; they cannot supply a conversation turn that is never available at prediction time. ML 11 covers the general evaluation and monitoring toolkit.

Keep model and preprocessing artifacts together. The tokenizer and vocabulary, maximum length and truncation policy, label order, calibration temperature and decision threshold are a single inference package. Reloading weights with another tokenizer silently changes meanings. Store a versioned label schema and a few deterministic contract examples with the package so that serving can check its assumptions.

A runnable experiment: pretrain, then fine-tune

Plain version. We pretrain a two-layer encoder from scratch with masked language modeling on a tiny unlabeled corpus, then fine-tune it for a sentiment task and an NLI task. The data are synthetic, so the experiment demonstrates mechanisms, not benchmark performance.

The design isolates three questions:

1. **Does pretraining transfer?** The unlabeled corpus contains every adjective, each next to `love` or `hate`. The sentiment training set never contains `superb` or `terrible`; the test set contains only those two adjectives, with and without `not`. A randomly initialized encoder has never seen them in any context.
2. **Does a hypothesis-only baseline expose a shortcut?** In the NLI training data, `not` appears only in contradiction hypotheses — a deliberate annotation artifact.
3. **Does a challenge set reveal it?** The challenge set puts `not` in an entailment (both sentences negated) and builds a contradiction whose hypothesis has no `not` (the premise is negated instead).

The encoder follows the BERT recipe from NLP 10 in miniature: token + position + segment embeddings, layer normalization, two Transformer encoder blocks, 15% masked positions, and a pooled `[CLS]` head with a tanh layer for fine-tuning. It uses AdamW [26].

```
import itertools, random
import torch
from torch import nn
```

```
from torch.nn import functional as F
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, f1_score

torch.manual_seed(0); random.seed(0); torch.set_num_threads(1)
POS, NEG = ['good', 'great', 'fine', 'superb'], ['bad', 'awful', 'poor', 'terrible']
ITEMS, NAMES = ['battery', 'screen', 'camera', 'speaker'], ['mina', 'ravi', 'lena']
VERBS = {'repaired': 'repair', 'cleaned': 'clean', 'tested': 'test'}
OBJS = ['the pump', 'the valve', 'the sensor']
words = set(' '.join(POS + NEG + ITEMS + NAMES + OBJS).split()) | set(VERBS) \
    | set(VERBS.values()) | set('the is not i love hate it someone did'.split())
vocab = ['[PAD]', '[CLS]', '[SEP]', '[MASK]'] + sorted(words)
idx, L, D = {w: i for i, w in enumerate(vocab)}, 16, 32

def batch(examples):                       # each example: (text,) or (text_a, text_b)
    ids, segs = [], []
    for ex in examples:
        toks, seg = ['[CLS]'], [0]
        for s, text in enumerate(ex):
            part = text.split() + ['[SEP]']
            toks += part; seg += [s] * len(part)
        ids.append([idx[t] for t in toks] + [0] * (L - len(toks)))
        segs.append(seg + [0] * (L - len(seg)))
    return torch.tensor(ids), torch.tensor(segs)

class TinyBERT(nn.Module):
    def __init__(self):
        super().__init__()
        self.tok, self.pos, self.seg = nn.Embedding(len(vocab), D), nn.Embedding(L, D),
         ↪  nn.Embedding(2, D)
        self.norm = nn.LayerNorm(D)
        block = nn.TransformerEncoderLayer(D, 2, 64, dropout=0.0, activation='gelu',
         ↪  batch_first=True)
        self.encoder = nn.TransformerEncoder(block, 2)
        self.mlm = nn.Linear(D, len(vocab))
    def forward(self, ids, seg):
        h = self.tok(ids) + self.pos(torch.arange(ids.shape[1])) + self.seg(seg)
        return self.encoder(self.norm(h), src_key_padding_mask=ids.eq(0))

# 1) Masked-LM pretraining on unlabeled text (no sentiment or NLI labels).
corpus = [(f'i love it the {i} is {a}',) for i in ITEMS for a in POS] \
    + [(f'i hate it the {i} is {a}',) for i in ITEMS for a in NEG] \
    + [(f'{n} {v} {o}',) for n in NAMES for v in VERBS for o in OBJS]
model = TinyBERT()
ids, seg = batch(corpus)
opt = torch.optim.AdamW(model.parameters(), lr=3e-3)
for step in range(400):
    pick = (torch.rand(ids.shape) < 0.15) & ids.gt(3)    # 15% of real tokens
    logits = model.mlm(model(ids.masked_fill(pick, idx['[MASK]']), seg))
    loss = F.cross_entropy(logits.flatten(0, 1), ids.masked_fill(~pick, -100).flatten())
    opt.zero_grad(); loss.backward(); opt.step()
print('final MLM loss:', round(loss.item(), 3))
pretrained_state = {k: v.clone() for k, v in model.state_dict().items()}

# 2) Fine-tune encoder + [CLS] head end to end.
```

```
def finetune(encoder, train, n_classes, epochs=60, lr=1e-3):
    torch.manual_seed(1)
    head = nn.Sequential(nn.Linear(D, D), nn.Tanh(), nn.Linear(D, n_classes))
    opt = torch.optim.AdamW([*encoder.parameters(), *head.parameters()], lr=lr)
    x, y = batch([ex for ex, _ in train]), torch.tensor([lab for _, lab in train])
    for _ in range(epochs):
        loss = F.cross_entropy(head(encoder(*x)[:, 0]), y)
        opt.zero_grad(); loss.backward(); opt.step()
    return lambda exs: head(encoder(*batch(exs))[:, 0]).argmax(-1).tolist()

def report(name, data, predict):
    gold = [lab for _, lab in data]
    with torch.no_grad():
        pred = predict([ex for ex, _ in data])
    print(f'{name:46s} acc={accuracy_score(gold, pred):.2f} '
          f'macro-F1={f1_score(gold, pred, average="macro"):.2f}')

# Task A: sentiment with negation. "superb" and "terrible" never get a label.
def review(item, adj, negated):
    return ((f'the {item} is {"not " * negated}{adj}',), int(adj in POS) ^ negated)
labeled = [a for a in POS + NEG if a not in ('superb', 'terrible')]
train_a = [review(i, a, n) for i in ITEMS for a in labeled for n in (0, 1)]
test_a = [review(i, a, n) for i in ITEMS for a in ('superb', 'terrible') for n in (0, 1)]
report('sentiment, pretrained encoder', test_a, finetune(model, train_a, 2))
report('sentiment, random-init encoder', test_a, finetune(TinyBERT(), train_a, 2))

# Task B: NLI (0 entailment, 1 neutral, 2 contradiction) with a negation shortcut.
def pair(n, v, o, kind):
    p, neg = f'{n} {v} {o}', f'{n} did not {VERBS[v]} {o}'
    other = list(VERBS)[(list(VERBS).index(v) + 1) % 3]
    return {'ent': ((p, f'someone {v} {o}'), 0), 'neu': ((p, f'{n} {other} {o}'), 1),
            'con': ((p, neg), 2),                  # training: "not" only in contradictions
            'neg-ent': ((neg, neg), 0),            # challenge: "not" in an entailment
            'neg-con': ((neg, p), 2)}[kind]        # challenge: contradiction without "not"
triples = list(itertools.product(NAMES, VERBS, OBJS)); random.shuffle(triples)
train_b = [pair(*t, k) for t in triples[:18] for k in ('ent', 'neu', 'con')]
test_b = [pair(*t, k) for t in triples[18:] for k in ('ent', 'neu', 'con')]
challenge = [pair(*t, k) for t in triples[18:] for k in ('neg-ent', 'neu', 'neg-con')]

vec = CountVectorizer(token_pattern=r'\S+').fit([ex[1] for ex, _ in train_b])
hyp_only = LogisticRegression(C=10, max_iter=1000).fit(
    vec.transform([ex[1] for ex, _ in train_b]), [lab for _, lab in train_b])
hyp_predict = lambda exs: hyp_only.predict(vec.transform([ex[1] for ex in exs])).tolist()
model.load_state_dict(pretrained_state)
pair_predict = finetune(model, train_b, 3)
for split, data in [('in-distribution', test_b), ('challenge', challenge)]:
    report(f'NLI {split}, hypothesis-only', data, hyp_predict)
    report(f'NLI {split}, pair encoder', data, pair_predict)
counter = [pair(*t, k) for t in triples[:18] for k in ('neg-ent', 'neg-con')]
model.load_state_dict(pretrained_state)
report('NLI challenge, pair encoder + counterexamples', challenge,
       finetune(model, train_b + counter, 3))
```

The run takes a few seconds on one CPU thread and prints:

```
final MLM loss: 0.621
sentiment, pretrained encoder                  acc=1.00 macro-F1=1.00
sentiment, random-init encoder                 acc=0.25 macro-F1=0.20
NLI in-distribution, hypothesis-only           acc=1.00 macro-F1=1.00
NLI in-distribution, pair encoder              acc=1.00 macro-F1=1.00
NLI challenge, hypothesis-only                 acc=0.33 macro-F1=0.22
NLI challenge, pair encoder                    acc=0.67 macro-F1=0.56
NLI challenge, pair encoder + counterexamples  acc=1.00 macro-F1=1.00
```

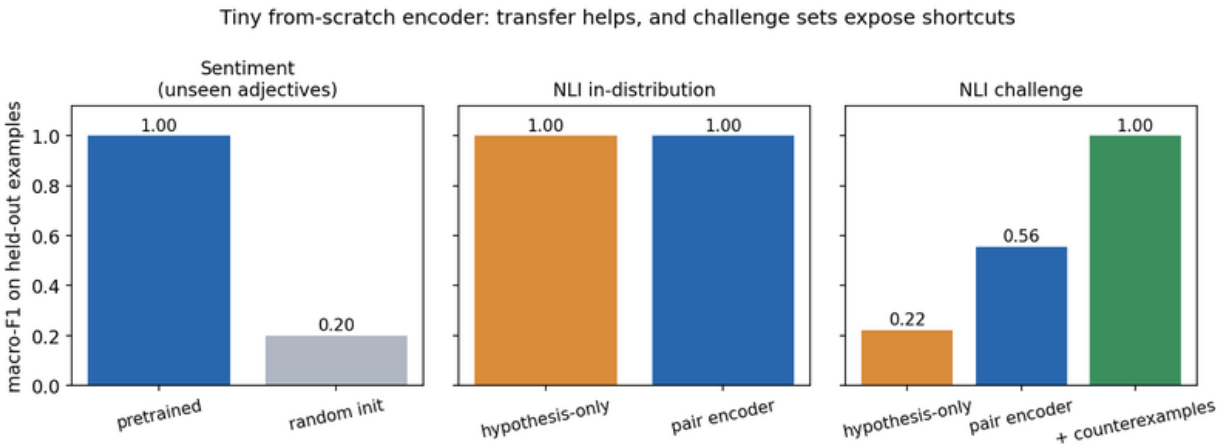


Figure 4 — Macro-F1 from the run above, regenerated by the figure script. Sixteen sentiment test sentences, 27 in-distribution and 27 challenge NLI pairs; synthetic data only.

Transfer. The pretrained encoder labels all sixteen test sentences correctly, including the negated ones, although `superb` and `terrible` never appeared with a label. During pretraining they shared contexts with `love` and `hate` respectively, so their representations already sat near the labeled adjectives of the same polarity. The randomly initialized encoder has only random embeddings for the two words, and the fine-tuning set gives it nothing to connect them to the labels. This is the BERT argument at toy scale: unlabeled text can supply distinctions that a small labeled set cannot. It also shows the dependency: transfer worked because the unlabeled corpus contained the relevant co-occurrence. Pretraining on unrelated text would not help.

Shortcut. In distribution, the hypothesis-only logistic regression is perfect. That is the warning sign: a model that never reads the premise cannot be doing inference, so the test set does not require inference. The fine-tuned pair encoder is also perfect in distribution, which on its own tells us nothing about what it uses.

Challenge. On the challenge set, the hypothesis-only baseline drops to chance-level accuracy. Inspecting the predictions shows that the pair encoder classifies every negated contradiction and every neutral pair correctly, but labels all nine doubly negated entailments as contradictions — it has partly learned "`not` in the hypothesis means contradiction." Adding eighteen counterexamples of the two challenge types (built from the *training* names, verbs and objects) fixes the challenge set. Be careful with that last line: the counterexamples share templates with the chal-

lenge set, so a perfect score shows that the model can learn the distinction once the data demands it, not that it would generalize to a new kind of shortcut.

Everything here is deliberately small: a 40-word vocabulary, sentences of at most a dozen tokens, one seed. Across the four seeds we tried, the qualitative pattern is the same, but the exact random-init and challenge numbers move. Exercise 5 asks you to measure that spread.

Modern view

Surveys and their maps. Minaee et al.'s ACM Computing Surveys review covers more than 150 deep-learning text classifiers and more than 40 datasets across sentiment analysis, news categorization, question answering and natural language inference [21]. It groups models by architecture — feed-forward, recurrent, convolutional, capsule, attention, memory-augmented, graph, Siamese, hybrid, and Transformer-based pretrained language models — and its central takeaway is that pretrained language models fine-tuned with task labels had set new states of the art on many text classification tasks. Chang et al.'s evaluation survey separates *what* to evaluate, *where* (which benchmarks) and *how* (automatic, human, protocols) [24], and Zhao et al.'s survey places classification inside the broader workflow of pretraining, adaptation, utilization and capability evaluation [25]. Read together, they argue for task-specific evaluation plans rather than one leaderboard number.

Encoders in the LLM era (2023–2026). Large decoder-only models can classify by prompting, without task-specific weights. Zhang et al. evaluated LLMs on 13 sentiment tasks across 26 datasets against small models trained on domain data. LLMs performed satisfactorily on simpler tasks but lagged on tasks that need structured sentiment information or a deeper understanding of specific phenomena, while clearly outperforming the small models in few-shot settings [22]. The practical reading: prompting is a strong option when labels are scarce, and a fine-tuned encoder remains competitive when labeled data exist and latency matters. Encoders themselves are still evolving. ModernBERT, published at ACL 2025, brings recent architecture and training optimizations to encoder-only models, trains on 2 trillion tokens with a native 8,192-token sequence length, and reports strong results across classification and retrieval evaluations along with speed and memory efficiency [23]. The fine-tuning interface in this chapter — a head on `[CLS]` or on token states — carries over unchanged.

What did not change. Annotation artifacts [9], [10] and heuristic shortcuts [11] are properties of datasets and training signals, not of one architecture. A larger or newer model can still learn them, and a prompted model can bring its own. Partial-input baselines, challenge sets, calibration checks [18] and abstention policies [13] remain the tools for finding out.

Open problems. Robustness under domain shift (the MultiNLI matched/mismatched split [8] is an early, narrow version), calibrated abstention for long and multi-passage inputs, multilingual and code-switched sentiment, labels that change with policy, and evaluation sets that leak into pretraining corpora. Each is easier to study with a small, transparent baseline next to the large model.

Key takeaways

- The encoder is shared; the head and the labels define the task. Choose sequence, token or span output first.
- Pair tasks pack both texts into one sequence so that attention can compare them token by token; the price is that neither side can be cached.
- The unit of supervision — sentence, review, conversation — is part of the task definition.
- A hypothesis-only (partial-input) baseline that scores well means the benchmark does not fully require the relationship you care about.
- Challenge sets that flip one cue at a time reveal shortcuts that in-distribution accuracy hides.
- Fine-tune with small learning rates, warmup and decay; report variance over seeds and select on development data.
- Evaluate decisions (accuracy, macro-F1), probabilities (calibration) and abstention (coverage, accepted risk) as separate properties.
- Relevance is not answerability: a span answer needs a supporting passage and an explicit way to say "not stated."

Exercises

1. **Truncation policy.** You must classify reviews that are often longer than the encoder window. Compare keeping the first tokens, the last tokens, and averaging predictions over overlapping windows. Which reviews would each policy misjudge?

   Hint

   Write three short synthetic reviews whose verdict appears at the start, at the end, and is reversed halfway ("at first I liked it ... but"). Decide which label the annotation guideline intends before choosing a policy.

2. **A partial-input audit.** For the NLI data in the experiment, train a premise-only baseline as well. Predict its accuracy before running it, then explain the result.

   Hint

   In the training templates, every premise appears with all three labels. A premise-only model can do no better than the majority class; compare that with the hypothesis-only result.

3. **SQuAD 2.0 threshold.** Given development-set scores s_{null} and $\hat{s}$ for answerable and unanswerable questions, describe how to choose τ, and what happens to coverage and accepted accuracy as τ grows.

   Hint

   Sweep τ over the observed score differences $\hat{s} - s_{\text{null}}$. A larger τ answers fewer questions. Plot coverage against accepted accuracy, then pick the point that matches the stated cost of a wrong answer.

4. **Calibration.** Add temperature scaling to the experiment's sentiment classifier: fit one temperature on a held-out split by minimizing negative log-likelihood. Show that accuracy cannot change and measure how the mean confidence moves.

   Hint

   Dividing logits by a positive scalar preserves their order. Optimize $\log T_c$ instead of T_c to keep the temperature positive.

5. **Seed variance.** Run the experiment with five seeds for pretraining and fine-tuning. Report the mean and range of each line, and decide which conclusions survive.

   Hint

   Change both `torch.manual_seed` calls. Separate claims about direction (pretrained better than random-init; challenge lower than in-distribution) from claims about exact values.

6. **Inference challenge pairs.** Create six NLI pairs that separately test role reversal, negation in the premise, modality, attribution, quantifier scope and time. Write the intended label and the assumptions before evaluating any model.

   Hint

   Preserve neutral as an option when the premise supplies too little information. For attribution, compare "Mina said the pump failed" with "The pump failed."

References

1. Masato Hagiwara, *Real-World Natural Language Processing: Practical Applications with Deep Learning*, Chapters 9–11, Manning, 2021. Publisher
2. Susan McRoy, *Principles of Natural Language Processing*, Chapters 6 and 8, open textbook, 2021. Open textbook
3. Jacob Devlin, Ming-Wei Chang, Kenton Lee and Kristina Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," NAACL-HLT, 2019. Paper
4. Google Research, *BERT: original implementation*, software, 2018. Classification, Question answering, Optimizer
5. Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng and Christopher Potts, "Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank," EMNLP, 2013. Paper
6. Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng and Christopher Potts, "Learning Word Vectors for Sentiment Analysis," ACL-HLT, 2011. Paper
7. Samuel R. Bowman, Gabor Angeli, Christopher Potts and Christopher D. Manning, "A large annotated corpus for learning natural language inference," EMNLP, 2015. Paper

8. Adina Williams, Nikita Nangia and Samuel R. Bowman, “A Broad-Coverage Challenge Corpus for Sentence Understanding through Inference,” NAACL-HLT, 2018. Paper
9. Suchin Gururangan, Swabha Swayamdipta, Omer Levy, Roy Schwartz, Samuel R. Bowman and Noah A. Smith, “Annotation Artifacts in Natural Language Inference Data,” NAACL-HLT, 2018. Paper
10. Adam Poliak, Jason Naradowsky, Aparajita Haldar, Rachel Rudinger and Benjamin Van Durme, “Hypothesis Only Baselines in Natural Language Inference,” *SEM, 2018. Paper
11. R. Thomas McCoy, Ellie Pavlick and Tal Linzen, “Right for the Wrong Reasons: Diagnosing Syntactic Heuristics in Natural Language Inference,” ACL, 2019. Paper
12. Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev and Percy Liang, “SQuAD: 100,000+ Questions for Machine Comprehension of Text,” EMNLP, 2016. Paper
13. Pranav Rajpurkar, Robin Jia and Percy Liang, “Know What You Don’t Know: Unanswerable Questions for SQuAD,” ACL, 2018. Paper
14. Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy and Samuel R. Bowman, “GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding,” ICLR, 2019. Paper
15. Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy and Samuel R. Bowman, “SuperGLUE: A Stickier Benchmark for General-Purpose Language Understanding Systems,” NeurIPS, 2019. Paper
16. Marius Mosbach, Maksym Andriushchenko and Dietrich Klakow, “On the Stability of Fine-tuning BERT: Misconceptions, Explanations, and Strong Baselines,” ICLR, 2021. Paper
17. Chuan Guo, Geoff Pleiss, Yu Sun and Kilian Q. Weinberger, “On Calibration of Modern Neural Networks,” ICML, 2017. Paper
18. Shrey Desai and Greg Durrett, “Calibration of Pre-trained Transformers,” EMNLP, 2020. Paper
19. scikit-learn developers, “Classification metrics,” scikit-learn documentation, current online edition. Documentation
20. Nils Reimers and Iryna Gurevych, “Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks,” EMNLP-IJCNLP, 2019. Paper
21. Shervin Minaee, Nal Kalchbrenner, Erik Cambria, Narjes Nikzad, Meysam Chenaghlu and Jianfeng Gao, “Deep Learning–based Text Classification: A Comprehensive Review,” ACM Computing Surveys 54(3), 2021. Survey
22. Wenxuan Zhang, Yue Deng, Bing Liu, Sinno Jialin Pan and Lidong Bing, “Sentiment Analysis in the Era of Large Language Models: A Reality Check,” Findings of NAACL, 2024. Paper
23. Benjamin Warner, Antoine Chaffin, Benjamin Clavié, Orion Weller, Oskar Hallström, Said Taghadouini et al., “Smarter, Better, Faster, Longer: A Modern Bidirectional Encoder for Fast, Memory Efficient, and Long Context Finetuning and Inference,” ACL, 2025. Paper
24. Yupeng Chang et al., “A Survey on Evaluation of Large Language Models,” ACM Transactions on Intelligent Systems and Technology, 2024. Survey
25. Wayne Xin Zhao et al., “A Survey of Large Language Models,” Frontiers of Computer Science, 2026. Updated survey
26. Ilya Loshchilov and Frank Hutter, “Decoupled Weight Decay Regularization,” ICLR, 2019. Paper

NLP 12 • Syntactic and Semantic Tasks

Parse constituents with CKY and dependencies with transitions or spanning trees, label semantic roles, resolve coreference, sketch AMR, and evaluate each structure with its own metric.

Online edition: https://blog.shyandram.dev/en/learn/nlp/syntactic-semantic-tasks/

What you'll learn

- Specify a context-free grammar, recognize and enumerate parses with the CKY chart, and choose among them with a probabilistic grammar or neural span scores.
- Build dependency trees with a transition system (arc-standard) and with graph-based decoding, and explain why independent head choices need a tree constraint.
- Label predicate–argument structure (semantic roles), keep scope and provenance in extracted relations, and read a small AMR graph.
- Link mentions into entities with an antecedent model and preserve ambiguity when context is missing.
- Evaluate each structure with its own metric: PARSEVAL brackets, UAS/LAS, role F1, MUC/B^3/CEAF and CoNLL F1, and Smatch.
- Describe how pretrained encoders and large language models changed these tasks, and what they did not change.

The big picture

Language has a linear surface and a structured interpretation. A sentence arrives as an ordered sequence of words, but readers group those words, identify relations among them, decide who did what to whom, and connect "it" in one sentence to "the pump" in another. "We saw robots with cameras" might describe camera-carrying robots or the equipment used to see them. The word list alone does not choose one interpretation.

> **Explain like I'm five**
>
> A sentence is like a construction kit. The words are pieces, and grammar describes ways to connect them. Parsing shows the connections; role labeling says which piece is the doer and which is the thing being done to; coreference notices when two pieces are really the same toy seen twice. Sometimes the same pieces make two sensible constructions. A good explanation shows both, then asks which fits the situation.

This chapter collects the classic **structured prediction** tasks of NLP. In NLP 06 the output was one label per position. Here the output is a tree, a graph or a partition of mentions, and choices at one place constrain choices elsewhere: a word can have only one head, a span can belong to

only one argument of a predicate, and coreference links must form consistent clusters. McRoy's chapters on syntax, semantics and discourse, with a separate chart-parsing appendix, give the linguistic view [1]; Jurafsky and Martin's chapters on constituency parsing, dependency parsing, semantic roles and coreference give the algorithmic one [2]. The linguistic categories themselves — constituents, dependencies, quantifier scope — are introduced in NLP 02.

These structures are useful for extracting who did what, locating modifiers and building tests that distinguish fluency from structural understanding. They do not settle every question: a grammatical sentence may be false, and a fragmentary message may still communicate clearly. We treat each structure as an explicit representation with a known scope, not as a definition of understanding.

Constituents and context-free grammars

Plain version. Phrase structure groups words into units that can combine recursively. A grammar is a specification of permitted combinations, while a parse is one derivation for a particular sentence.

A context-free grammar can be specified as:

$$G = (\mathcal{N}, \Sigma, \mathcal{R}, S).$$

Here G is the grammar, $\mathcal{N}$ is the set of nonterminal categories, Σ is the terminal vocabulary, $\mathcal{R}$ is the set of rewrite rules and $S \in \mathcal{N}$ is the start category. Nonterminals represent structures such as a noun phrase; terminals are the observed words. A context-free rule replaces one nonterminal without inspecting its surrounding symbols.

For an original toy grammar, use rules such as $S \to NP\ VP$, $VP \to V\ NP$, $PP \to P\ NP$ and $NP \to NP\ PP$. Here S denotes a sentence, NP a noun phrase, VP a verb phrase, PP a prepositional phrase, V a verb category and P a preposition category. These are notation choices for the toy grammar, not a complete grammar of English. Treebanks such as the Penn Treebank supply large collections of sentences annotated with such trees, from which grammars and parsers are learned [4].

Recursive rules generate unbounded families from finite specifications, but the same recursion can make naive search revisit work or loop indefinitely, so a parser needs a search strategy and a compact representation of partial analyses.

Distinguish coverage from preference. A rule system can permit a sentence while assigning several analyses, or fail to parse a perfectly reasonable sentence because a lexical entry is missing. A parsing failure says that the implemented grammar and search did not produce an analysis; it does not establish that the user wrote nonsense. The benefit of an explicit grammar is inspectability: one can show which rule connected spans and where ambiguity arose [1].

Chart parsing by spans: the CKY algorithm

Top-down hypotheses and bottom-up evidence

Plain version. A parser can start from what a sentence is expected to contain, or from what the observed words already support. Both directions need a way to avoid repeating the same work.

A top-down search begins with the start category and expands a production into expected children. It can reject lexical alternatives that do not fit the current expectation, but a poorly chosen expansion can send it down a long unsuccessful path, and a production that expands its first child into the same category can revisit the same unresolved expectation before consuming any input. A bottom-up search begins with observed words and constructs categories that their spans support. It can create many phrases that never contribute to a complete sentence, but the intermediate results can be shared by several successful analyses.

The important design question is therefore not simply which direction is "better," but how partial work is represented and when two partial analyses count as the same search state. A chart stores completed spans so that a state reached by another path need not be recomputed. Memoization alone does not settle ambiguity, though: recognition needs one Boolean per span and category, the preferred parse needs a score and backpointer, and all alternatives need a forest that retains distinct derivation links.

The span recurrence

Plain version. Store what each contiguous span can be. Build longer spans by joining shorter ones and share their results across all complete parses.

The Cocke–Kasami–Younger (CKY) algorithm is the standard dynamic program for this idea [2]. It requires binary rules: a grammar in Chomsky normal form has only rules $A \to B\,C$ and $A \to a$, and any context-free grammar without empty productions can be converted to that form [2]. Use half-open boundaries: the span $[i, j)$ contains tokens at positions i through $j - 1$. Recognition follows:

$$C[i, j, A] = \bigvee_{\substack{A \to BC \in \mathcal{R} \\ i<k<j}} \left(C[i, k, B] \wedge C[k, j, C]\right).$$

Here $C[i, j, A]$ is a Boolean statement that category A covers span $[i, j)$, $\mathcal{R}$ is the binary rule set, $A \to BC$ combines left category B with right category C, k is an interior split boundary, $\wedge$ means both children must exist and $\bigvee$ means at least one rule-and-split combination succeeds. The letter C is used both for the chart notation and for a right-child category; its role is identified by brackets versus a category argument.

Initialize length-one cells from lexical rules. Then iterate over span lengths, start positions, split positions and compatible rules. Because children are shorter than their parent, they have already been computed. A unary rule or empty production needs additional handling; this lesson's executable grammar deliberately excludes both.

This algorithm recognizes an input, but recognition alone cannot explain its trees. Store successful combinations as backpointers or a packed forest. Several parents can share a child span rather

than duplicating its full tree. Expanding all complete trees can still cost exponential time when a sentence is highly ambiguous, even if recognition is polynomial. For fixed binary grammar rules, the nested span-and-split loops have cubic dependence on token length; the chart has quadratically many spans, each with possible categories.

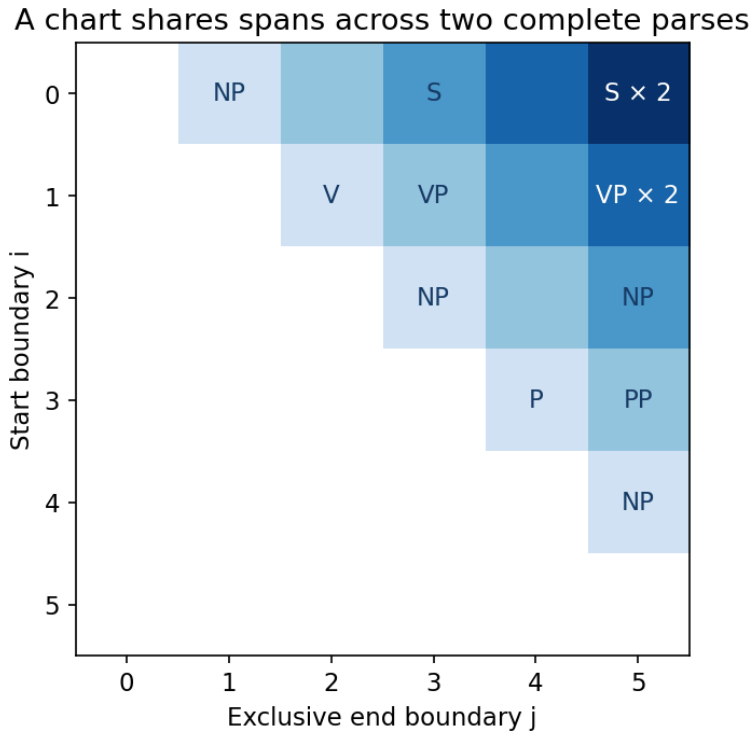


Figure 1 — A chart stores reusable spans. The complete sentence cell can contain more than one analysis.

A parser that preserves two analyses

When the grammar permits two attachments, report the alternatives. A parser should not discard ambiguity merely because one option was inserted first. The following standalone parser stores complete small trees in chart cells. This is intentionally simpler than a packed-forest implementation; it suits the five-word demonstration, while the preceding paragraphs explain why explicit enumeration is unsuitable for large ambiguous inputs.

```
from collections import defaultdict

words = "we saw robots with cameras".split()
lexical = {
    "we": ["NP"], "saw": ["V"], "robots": ["NP"],
    "with": ["P"], "cameras": ["NP"],
}
rules = [
    ("S", "NP", "VP"),
```

```
    ("VP", "V", "NP"),
    ("VP", "VP", "PP"),
    ("NP", "NP", "PP"),
    ("PP", "P", "NP"),
]
chart = defaultdict(lambda: defaultdict(list))
for i, word in enumerate(words):
    for category in lexical.get(word, []):
        chart[i, i + 1][category].append((category, word))
for length in range(2, len(words) + 1):
    for start in range(len(words) - length + 1):
        end = start + length
        for split in range(start + 1, end):
            for parent, left, right in rules:
                for left_tree in chart[start, split].get(left, []):
                    for right_tree in chart[split, end].get(right, []):
                        tree = (parent, left_tree, right_tree)
                        if tree not in chart[start, end][parent]:
                            chart[start, end][parent].append(tree)

def bracket(tree):
    if len(tree) == 2:
        return f"({tree[0]} {tree[1]})"
    return f"({tree[0]} {bracket(tree[1])} {bracket(tree[2])})"

parses = chart[0, len(words)].get("S", [])
assert len(parses) == 2
for tree in parses:
    print(bracket(tree))
print("complete parses:", len(parses))
```

```
(S (NP we) (VP (V saw) (NP (NP robots) (PP (P with) (NP cameras)))))
(S (NP we) (VP (VP (V saw) (NP robots)) (PP (P with) (NP cameras))))
complete parses: 2
```

In one tree, the prepositional phrase belongs inside the object noun phrase. In the other, it modifies the verb phrase. Both use every word in order and satisfy the grammar. Choosing between them may require the surrounding scene, prior discourse or lexical preferences beyond these rules.

Add an unknown word and the chart may become empty at a lexical cell. The correct response is a coverage diagnosis. A production service might use a fallback tagger, allow an explicit unknown category or ask for clarification, but each fallback changes the recognized language. It should be documented and tested.

Probabilities over structures

Plain version. A grammar says which analyses exist. A probability model can prefer some analyses and summarize uncertainty across them.

A probabilistic context-free grammar (PCFG) assigns a probability to each production, normalized among rules with the same parent [40]. A derivation has:

$$p(\mathcal{T}) = \prod_{r \in \mathcal{T}} p(r), \qquad \hat{\mathcal{T}} = \operatorname*{argmax}_{\mathcal{T} \in \mathcal{D}(\mathbf{x})} p(\mathcal{T}).$$

Here $\mathcal{T}$ is a complete derivation tree, r indexes each production occurrence in that tree, $p(r)$ is its production probability, $\mathcal{D}(\mathbf{x})$ is the set of permitted parses whose terminal yield is the input $\mathbf{x}$, and $\hat{\mathcal{T}}$ is the highest-probability parse. The product includes repeated uses of a rule separately.

Replacing chart alternatives with sums computes total probability mass over derivations (the inside algorithm). Replacing them with maxima computes the best derivation (probabilistic CKY). These answer different questions: a sentence with many moderately probable parses can have greater total probability than a sentence with one strong parse, and the best tree need not explain most of the posterior mass. The following snippet computes both for the same five words.

```
import math
from collections import defaultdict

words = "we saw robots with cameras".split()
lexical = {"we": "NP", "saw": "V", "robots": "NP", "with": "P", "cameras": "NP"}

def parse(p_np_pp):
    # All productions sharing a parent must sum to one, including lexical rules.
    assert 0 < p_np_pp < 1
    rules = {("S", "NP", "VP"): 1.0, ("PP", "P", "NP"): 1.0,
             ("VP", "V", "NP"): 0.6, ("VP", "VP", "PP"): 0.4,
             ("NP", "NP", "PP"): p_np_pp}
    lexical_prob = {"we": (1 - p_np_pp) / 3, "robots": (1 - p_np_pp) / 3,
                    "cameras": (1 - p_np_pp) / 3, "saw": 1.0, "with": 1.0}
    parent_mass = defaultdict(float)
    for (parent, _, _), probability in rules.items():
        parent_mass[parent] += probability
    for word, parent in lexical.items():
        parent_mass[parent] += lexical_prob[word]
    assert all(math.isclose(total, 1.0) for total in parent_mass.values())
    best = defaultdict(dict)    # (i, j) -> {category: (log prob, tree)}
    inside = defaultdict(lambda: defaultdict(float))
    for i, w in enumerate(words):
        best[i, i + 1][lexical[w]] = (math.log(lexical_prob[w]), f"({lexical[w]} {w})")
        inside[i, i + 1][lexical[w]] = lexical_prob[w]
    n = len(words)
    for length in range(2, n + 1):
        for i in range(n - length + 1):
            j = i + length
            for k in range(i + 1, j):
                for (a, b, c), p in rules.items():
                    if b in best[i, k] and c in best[k, j]:
                        score = math.log(p) + best[i, k][b][0] + best[k, j][c][0]
                        if score > best[i, j].get(a, (-math.inf,))[0]:
                            best[i, j][a] = (score, f"({a} {best[i, k][b][1]} {best[k,
                             ↪  j][c][1]})")
                        inside[i, j][a] += p * inside[i, k][b] * inside[k, j][c]
```

```
    score, tree = best[0, n]["S"]
    return tree, math.exp(score), inside[0, n]["S"]

for p in (0.2, 0.5):
    tree, p_best, p_total = parse(p)
    print(f"P(NP -> NP PP) = {p}: best = {p_best:.6f}, total = {p_total:.6f}")
    print("  ", tree)
    common = ((1 - p) / 3) ** 3
    assert math.isclose(p_total, common * (0.24 + 0.6 * p))
    assert math.isclose(p_best / p_total, max(0.24, 0.6 * p) / (0.24 + 0.6 * p))
```

```
P(NP -> NP PP) = 0.2: best = 0.004551, total = 0.006827
   (S (NP we) (VP (VP (V saw) (NP robots)) (PP (P with) (NP cameras))))
P(NP -> NP PP) = 0.5: best = 0.001389, total = 0.002500
   (S (NP we) (VP (V saw) (NP (NP robots) (PP (P with) (NP cameras)))))
```

The three lexical NP rules share the mass left after the recursive rule: each has probability $(1-p)/3$, where $p = p(NP \to NP\,PP)$. Both trees contain these three lexical choices, so their common factor is $c = ((1-p)/3)^3$. The verb attachment has probability $c \times 0.4 \times 0.6$; the noun attachment has $c \times 0.6p$. Omitting c preserves which tree wins **within one grammar**, but the resulting sum is no longer the probability of the sentence. The code retains it and checks normalization for every parent.

The preferred tree still flips when p crosses 0.4. Divide a tree's joint probability by the inside total to obtain its posterior probability given these words. Thus best-tree probability, sentence probability and posterior confidence are three different quantities. The attachment preference here comes from category-rule probabilities rather than lexical context; the next section's neural scorers add that missing evidence.

Rule probabilities estimated from a treebank depend on the annotation convention and domain. A news-style treebank and a support-message corpus can have different fragment patterns and attachment preferences. Probabilistic modeling supplies a preference mechanism, but it does not remove domain dependence.

Neural constituency parsing: score spans, decode with a chart

Plain version. Instead of multiplying rule probabilities, let a neural encoder score every span and label, then use the same chart to find the highest-scoring tree.

A span-based parser assigns each candidate constituent a score and searches for the tree with the largest total:

$$s(\mathcal{T}) = \sum_{(i,j,\ell)\in\mathcal{T}} s(i,j,\ell), \qquad \hat{\mathcal{T}} = \operatorname*{argmax}_{\mathcal{T}} s(\mathcal{T}).$$

Here $\mathcal{T}$ is a tree viewed as a set of labeled spans (i,j,ℓ) with label ℓ over the half-open span $[i,j)$, $s(i,j,\ell)$ is a score computed from contextual vectors at the span boundaries, and $\hat{\mathcal{T}}$ is the

decoded tree. Stern, Andreas and Klein showed that independent scoring of labels and spans is compatible with classical chart dynamic programming and also admits a greedy top-down decoder that recursively splits the input [5]. Kitaev and Klein replaced the LSTM encoder in that design with self-attention, found that separating positional and content information improved accuracy, and reported 93.55 F1 on the Penn Treebank without external data and 95.13 F1 with pretrained word representations [6]. A follow-up added multilingual pretraining [7].

The output remains a tree even if its scores come from a large neural network: the chart guarantees well-formedness, and the encoder supplies lexical evidence that a PCFG's rule probabilities could not.

Dependency structure and grammatical roles

Plain version. Dependency analysis links words directly. It highlights which word depends on which other word and labels their grammatical relationship.

A basic dependency tree can be represented by head indices:

$$h_t \in \{0, 1, \dots, T\} \setminus \{t\}, \qquad r_t \in \mathcal{D}.$$

Here h_t is the head index of word t, 0 denotes an artificial root, T is sentence length, r_t is the relation label for that word and $\mathcal{D}$ is the relation inventory. A valid basic tree additionally requires a specified root convention, connectivity and absence of directed cycles.

Independent head choices can violate those global requirements. Two words might select each other, forming a cycle. Several disconnected components might attach without a unique sentence root. A tree decoder must enforce the intended structure, or the service must validate and reject an invalid analysis.

Universal Dependencies provides a cross-linguistic annotation framework for morphology, part of speech and grammatical dependencies [8]. Its categories aim for useful comparability across languages while retaining language-specific distinctions. This is an annotation framework, not a claim that every language has identical word order or overt grammatical markers.

Phrase structure and dependencies emphasize different information. A constituent tree makes span grouping explicit. A dependency tree makes lexical heads and relations explicit. Converting between them requires conventions for heads, punctuation and function words; the conversion is not merely a drawing change.

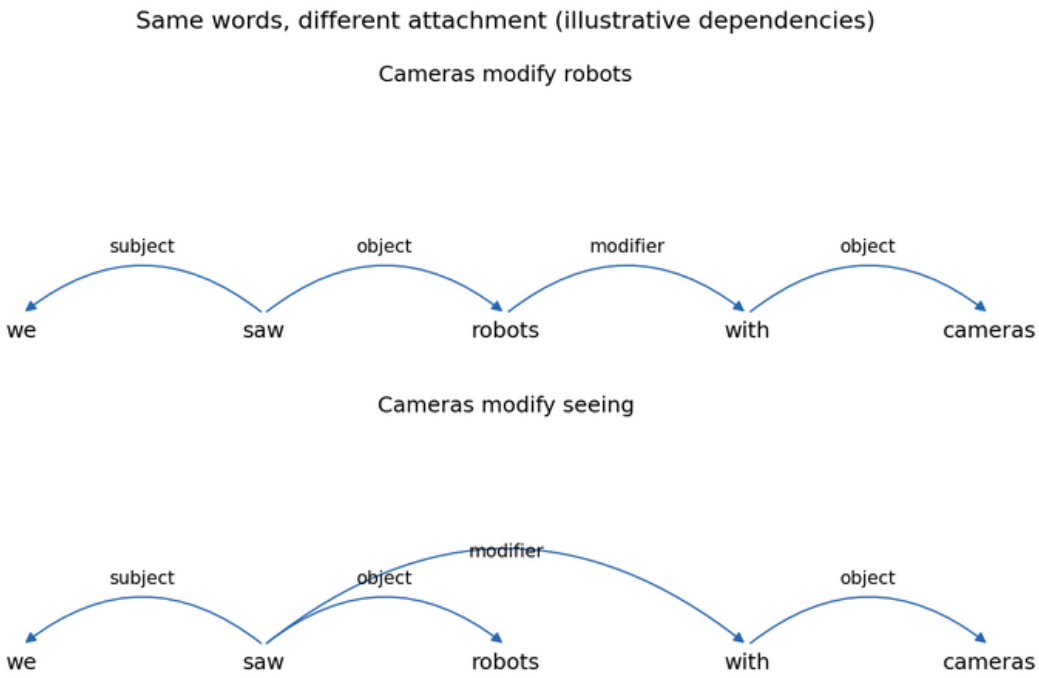


Figure 2 — Two attachments of the same words. These illustrative grammatical links explain the ambiguity and are not copied from a treebank.

For extraction, grammatical roles are useful evidence but not interchangeable with semantic roles. A passive sentence changes grammatical subject placement while preserving who acted on whom. The semantic-role section below develops that distinction and explains why a dependency arrow does not by itself establish a real-world relation.

Attachment ambiguity reveals a model's inductive bias

The two camera attachments share their words but assign different participants to the seeing event. Noun-phrase attachment describes a property of the robots. Verb-phrase attachment describes an instrument or circumstance of seeing. A downstream extractor that commits to the noun interpretation might create a robot–camera possession record that the instrument interpretation does not support. An attachment difference is consequential because of the relation it changes, not simply because the brackets look different.

A grammar can permit both while a probability model prefers one, as the PCFG snippet showed. With only parent-category rule probabilities, the model cannot condition that preference on the lexical identity of `saw`, `robots` and `cameras` unless its state or scoring representation is extended. This is a capacity limitation. A wider search over the same score cannot create missing lexical evidence.

The opposite failure is a search limitation: useful scores may exist, but pruning removes the correct span before a complete tree is formed. Diagnose these separately by asking whether the

reference structure is representable, whether it survives candidate construction, and whether it receives the highest score among survivors. Each answer suggests a different intervention: extend the representation, revise pruning, or improve scoring.

There is also a genuine information limitation: in an isolated sentence both readings may remain possible, and a gold tree records only the corpus's chosen interpretation. For an extractor, retaining both alternatives until the required relation is checked can be preferable to an early commitment.

Transition-based dependency parsing

Plain version. Read the sentence left to right with a stack, and at each step choose one of a few actions: shift the next word onto the stack, or attach the top two stack items to each other. A classifier picks the action; the sequence of actions builds the tree.

Nivre formalizes deterministic dependency parsers as transition systems and analyzes stack-based variants (arc-standard and arc-eager, restricted to projective trees) and list-based variants, one of which handles non-projective trees [9]. A **projective** tree is one whose arcs can be drawn above the sentence without crossing. In the arc-standard system used by Chen and Manning, a configuration $c = (\sigma, \beta, A)$ has a stack σ, a buffer β and an arc set A, and three transitions are available [10]:

$$\begin{aligned}
\textsc{shift}: &\ (\sigma,\ w|\beta,\ A) \Rightarrow (\sigma|w,\ \beta,\ A),\\
\textsc{left-arc}_r: &\ (\sigma|s_2|s_1,\ \beta,\ A) \Rightarrow (\sigma|s_1,\ \beta,\ A \cup \{s_1 \xrightarrow{r} s_2\}),\\
\textsc{right-arc}_r: &\ (\sigma|s_2|s_1,\ \beta,\ A) \Rightarrow (\sigma|s_2,\ \beta,\ A \cup \{s_2 \xrightarrow{r} s_1\}).
\end{aligned}$$

Here s_1 is the top of the stack and s_2 the item below it, w is the first buffer word, $\sigma|x$ means stack σ with x on top, $h \xrightarrow{r} d$ is an arc from head h to dependent d with label r, and $\Rightarrow$ reads "becomes." Parsing starts with only the root on the stack and all words in the buffer, and ends when the buffer is empty and only the root remains. Each word is shifted once and reduced once, so a sentence of n words needs exactly $2n$ transitions.

To train the classifier, convert each gold tree into the transition sequence that produces it — a **static oracle**. The rule below is one standard choice: attach left when the second stack item's gold head is the top; attach right when the top's gold head is the second item *and* the top has already collected all of its own dependents; otherwise shift.

```
words = ["ROOT", "we", "saw", "robots", "with", "cameras"]
# Gold heads for the instrument reading: "with cameras" attaches to "saw".
gold = {1: 2, 2: 0, 3: 2, 4: 5, 5: 2}

def oracle(gold):
    stack, buffer, arcs, steps = [0], list(range(1, len(words))), {}, []
    def done(h):  # has h already collected all of its dependents?
        return all(d in arcs for d, head in gold.items() if head == h)
    while buffer or len(stack) > 1:
        if len(stack) >= 2 and gold.get(stack[-2]) == stack[-1]:
```

```
            dependent = stack.pop(-2)
            arcs[dependent] = stack[-1]; steps.append("LEFT-ARC")
        elif len(stack) >= 2 and gold[stack[-1]] == stack[-2] and done(stack[-1]):
            dependent = stack.pop()
            arcs[dependent] = stack[-1]; steps.append("RIGHT-ARC")
        else:
            stack.append(buffer.pop(0)); steps.append("SHIFT")
    return steps, arcs

steps, arcs = oracle(gold)
print(" ".join(steps))
print({words[d]: words[h] for d, h in sorted(arcs.items())})
assert arcs == gold and len(steps) == 2 * (len(words) - 1)
```

```
SHIFT SHIFT LEFT-ARC SHIFT RIGHT-ARC SHIFT SHIFT LEFT-ARC RIGHT-ARC RIGHT-ARC
{'we': 'saw', 'saw': 'ROOT', 'robots': 'saw', 'with': 'cameras', 'cameras': 'saw'}
```

Ten transitions for five words, as predicted. The `done` check matters: if `robots` were attached to `saw` and popped before `cameras` arrived, a noun-attachment tree could never be built afterwards. The order of reductions encodes the tree.

At parse time a classifier replaces the oracle. Chen and Manning fed embeddings of words, part-of-speech tags and arc labels from fixed stack and buffer positions into a one-hidden-layer network with a cube activation and a softmax over transitions; their greedy parser is fast but, as they note, exposed to error propagation because an early wrong action cannot be undone [10]. Kiperwasser and Goldberg instead computed one bidirectional-LSTM vector per token, trained jointly with the parser, and built features by concatenating a few of them; the same representation worked for a greedy transition parser and a globally optimized graph parser [12]. Transition parsers run in time linear in sentence length; the price is a local, left-to-right decision procedure.

Graph-based dependency parsing

Plain version. Score every possible head–dependent arc at once, then choose the highest-scoring set of arcs that forms a tree.

McDonald et al. formalize weighted dependency parsing as finding a maximum spanning tree in a directed graph whose nodes are the words plus a root [11]:

$$\hat{\mathbf{h}} = \operatorname*{argmax}_{\mathbf{h} \in \mathcal{Y}(\mathbf{x})} \sum_{t=1}^{T} s(h_t, t).$$

Here $\mathbf{h} = (h_1, \dots, h_T)$ assigns a head to each word, $\mathcal{Y}(\mathbf{x})$ is the set of head assignments that form a valid tree for sentence $\mathbf{x}$, and $s(h, t)$ is the score of the arc from head h to dependent t. Over projective trees, Eisner's algorithm solves this in $O(n^3)$ time; over all trees, including non-projective ones, the Chu–Liu–Edmonds maximum-spanning-tree algorithm solves it in $O(n^2)$ [11].

Dozat and Manning made the arc scorer **biaffine** on top of a deep bidirectional encoder [13]:

$$s(h,t) = \mathbf{r}_t^{(\text{dep})\top} U\, \mathbf{r}_h^{(\text{head})} + \mathbf{r}_h^{(\text{head})\top} \mathbf{u}.$$

Here $\mathbf{r}_t^{(\text{dep})}$ and $\mathbf{r}_h^{(\text{head})}$ are lower-dimensional projections of the contextual vectors of the dependent and the candidate head, U is a learned matrix scoring their interaction, and $\mathbf{u}$ is a learned vector giving each word a prior tendency to be a head. A second biaffine classifier predicts the relation label for the chosen arc. The scorer produces a full $(T+1) \times T$ matrix of arc scores in one pass.

What does the tree constraint buy? The snippet below uses invented scores for our sentence, picks each word's best head independently, and then enumerates every single-root tree — feasible only for tiny inputs, but it makes the decoder's job visible. It finishes with UAS and LAS, defined in the next section.

```
import itertools
import numpy as np

words = ["ROOT", "we", "saw", "robots", "with", "cameras"]
n = len(words) - 1
# score[h, d]: invented arc scores (head h -> dependent d), not a trained model.
score = np.full((n + 1, n + 1), -2.0)
score[0, 2], score[2, 1], score[2, 3], score[5, 4] = 3.0, 2.0, 2.0, 2.0
score[3, 5], score[2, 5] = 1.5, 1.2                # robots vs saw for "cameras"
score[4, 5] = 1.8                                  # tempting but creates a cycle
np.fill_diagonal(score, -np.inf)

def is_tree(heads):                                # heads[d-1] is the head of d
    if sum(h == 0 for h in heads) != 1:
        return False
    for d in range(1, n + 1):
        seen, node = set(), d
        while node != 0:
            if node in seen:
                return False
            seen.add(node); node = heads[node - 1]
    return True

greedy = [int(np.argmax(score[:, d])) for d in range(1, n + 1)]
total = lambda hs: sum(score[h, d] for d, h in enumerate(hs, start=1))
trees = [hs for hs in itertools.product(range(n + 1), repeat=n)
         if all(h != d for d, h in enumerate(hs, 1)) and is_tree(hs)]
best = max(trees, key=total)
print("greedy heads:", greedy, "valid tree:", is_tree(greedy))
print("best tree:   ", list(best), "score:", round(total(best), 2))
print("trees enumerated:", len(trees))

gold = [2, 0, 2, 5, 2]
labels_gold = ["nsubj", "root", "obj", "case", "obl"]
labels_pred = ["nsubj", "root", "obj", "mark", "nmod"]
uas = np.mean([p == g for p, g in zip(best, gold)])
las = np.mean([p == g and lp == lg for p, g, lp, lg
               in zip(best, gold, labels_pred, labels_gold)])
print(f"UAS = {uas:.2f}, LAS = {las:.2f}")
```

```
greedy heads: [2, 0, 2, 5, 4] valid tree: False
best tree:    [2, 0, 2, 5, 3] score: 10.5
trees enumerated: 625
UAS = 0.80, LAS = 0.60
```

Greedy selection makes `with` and `cameras` heads of each other, a two-word cycle disconnected from the root (Figure 3). Among the 625 single-root trees over five words, the best one breaks the cycle and attaches `cameras` to `robots` — a valid tree, but the noun reading, while the gold annotation chose the instrument reading. A maximum-spanning-tree algorithm finds the same answer without enumeration.

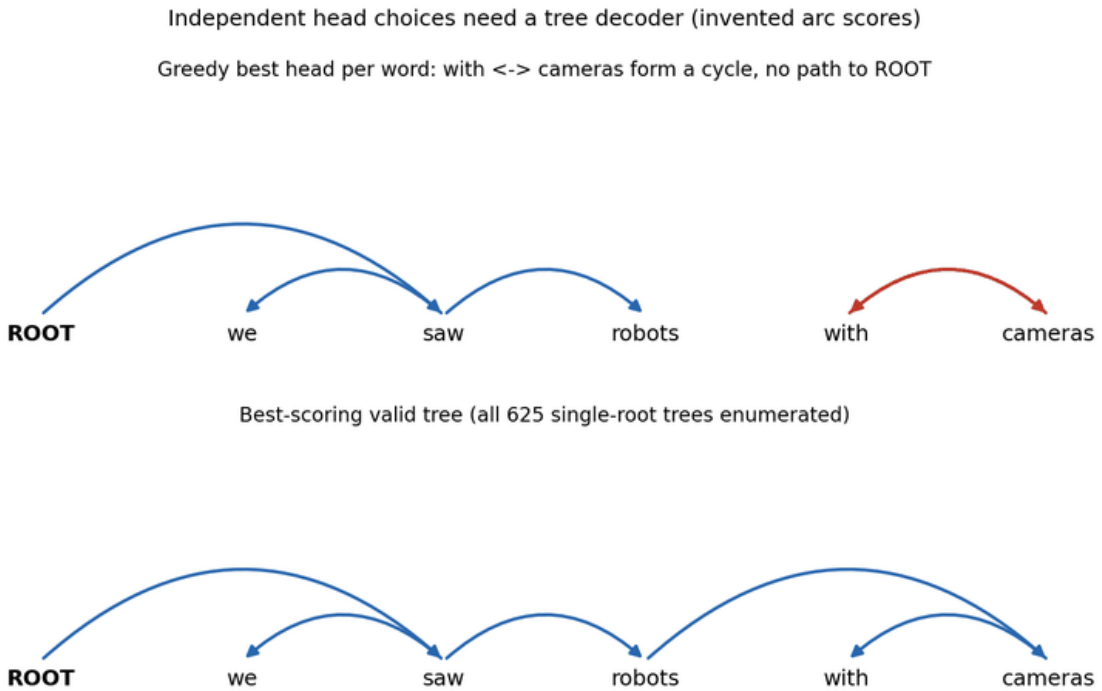


Figure 3 — Independent best heads can form a cycle; a tree decoder restores a valid structure. Scores are invented and match the snippet.

Graph-based and transition-based parsers make different trade-offs: global search over arc scores with quadratic or cubic decoding, versus linear-time local decisions. Kiperwasser and Goldberg's shared BiLSTM features served both families [12]. Whichever family is used, the structural guarantee comes from the decoder or transition system, not from the encoder.

Evaluating parsers: PARSEVAL, UAS and LAS

Plain version. Compare the representation that the parser promises to produce, and inspect mistakes where tokenization, annotation or context changes the answer.

The original PARSEVAL procedure compared a candidate parse with a treebank parse after removing material such as punctuation, and computed a **crossing-brackets** score — how often the candidate brackets a span in a way that crosses a reference bracket — and **recall**, the number of shared bracket pairs divided by the number in the reference [3]. Today's standard is labeled bracket precision, recall and F1 [2]:

$$P_{\text{lab}} = \frac{|\mathcal{B}(\hat{\mathcal{T}}) \cap \mathcal{B}(\mathcal{T}^*)|}{|\mathcal{B}(\hat{\mathcal{T}})|}, \qquad R_{\text{lab}} = \frac{|\mathcal{B}(\hat{\mathcal{T}}) \cap \mathcal{B}(\mathcal{T}^*)|}{|\mathcal{B}(\mathcal{T}^*)|}.$$

Here $\mathcal{B}(\mathcal{T})$ is the set of labeled constituents (i, j, ℓ) of tree $\mathcal{T}$, $\hat{\mathcal{T}}$ is the predicted tree, $\mathcal{T}^*$ is the reference tree, and F1 is the harmonic mean of the two. For our two parses, the trees share every labeled bracket except one: the noun-attachment tree has $(2, 5, NP)$, the verb-attachment tree has $(1, 3, VP)$. Counting preterminals, each tree has nine labeled constituents, so either tree scored against the other gets precision and recall $8/9$. State whether lexical preterminals, punctuation and the root are included; different inclusion rules change the denominator.

For dependencies, the **unlabeled attachment score** (UAS) is the fraction of words with the correct head, and the **labeled attachment score** (LAS) additionally requires the correct relation:

$$\text{LAS} = \frac{1}{M} \sum_{t \in \mathcal{J}} \mathbf{1}[(\hat{h}_t, \hat{r}_t) = (h_t, r_t)].$$

Here LAS is labeled attachment score, $\mathcal{J}$ is the set of evaluated word positions, $M = |\mathcal{J}|$ is its size, $\hat{h}_t$ and $\hat{r}_t$ are predicted head and relation, h_t and r_t are reference head and relation, and $\mathbf{1}[\cdot]$ is the indicator function; UAS drops the relation condition. In the snippet, `cameras` has the wrong head (both scores lose it) and `with` has the right head with the wrong label (only LAS loses it), giving 0.80 and 0.60.

Token alignment must be fixed before applying these formulas. The CoNLL 2018 shared task asked systems to parse raw text into Universal Dependencies for many languages, without gold-standard annotation on the test input [14]. Systems therefore segment and tokenize for themselves, and predicted words must be aligned with gold words before attachment scores mean anything. If the gold annotation has two words where the tokenizer emits one, an attachment comparison cannot be interpreted normally. Report the segmentation mismatch and specify how alignment is handled. Do not silently delete difficult examples.

Build small diagnostic pairs. Move a modifier while keeping vocabulary similar; add a nested clause; replace an active construction with a passive one; include a fragment or a code-switched phrase. These tests reveal the conditions under which a model's structure changes, even when an aggregate score appears stable. Avoid treating one gold tree as certainty when the sentence is genuinely ambiguous. A corpus needs a chosen convention or adjudicated analysis, but another analysis may be linguistically reasonable. For an application, ask whether the disagreement

changes the extracted fact or action. That connects a parsing error to its practical consequence.

Semantic role labeling: who did what to whom

Plain version. The same event can be expressed with different grammar. Extract the roles in the event instead of assuming the first name is always the actor.

For a simple two-participant predicate, write:

$$\text{repair}(\text{Mina}, \text{sensor}).$$

Here repair names the relation; the first argument is the repairer; and the second argument is the repaired object. "Mina repaired the sensor" and "The sensor was repaired by Mina" can share this representation. Their surface subjects differ. Grammar supplies clues, but a role interpretation must also account for the predicate and construction [1].

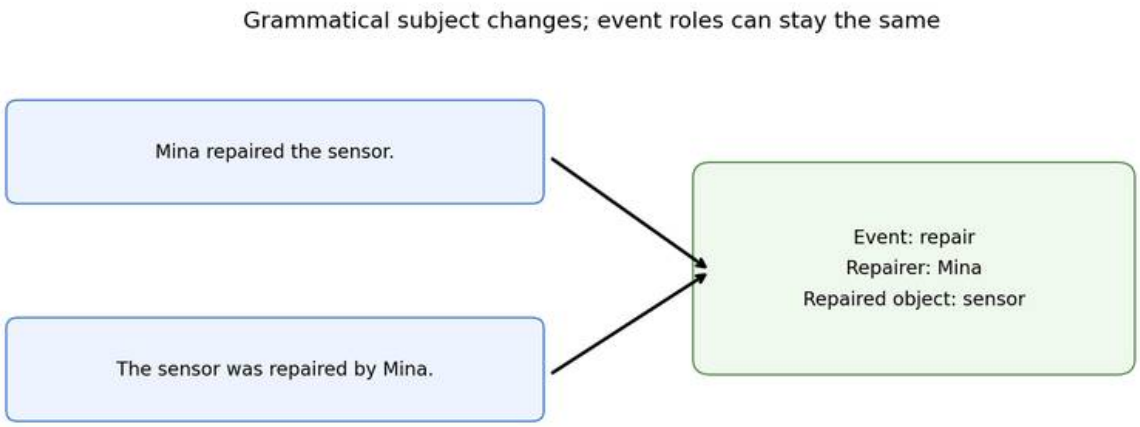


Figure 4 — Surface subjects can change while event roles stay the same. The examples and record are synthetic.

Not every verb permits every alternation, and not every argument is overt. "The sensor was repaired" leaves the repairer unspecified. A system that fills in the nearest previously mentioned person has made a contextual inference, not a direct extraction. That inference may be reasonable, but it needs a separate confidence policy and evidence contract. Preserve an explicit unknown role when the application does not authorize that inference.

Arguments and modifiers also differ. In "Mina repaired the sensor in Taipei on Tuesday," the repairer and repaired object participate in the core event; location and time modify it. An event schema may include all four fields, but it should say whether each is required. Optional fields prevent an extractor from inventing details merely to satisfy an overly strict schema.

PropBank roles and the SRL task

Gildea and Jurafsky framed automatic semantic role labeling as a statistical task over parsed sentences: match role-bearing phrases to parse constituents, extract features from the words and the tree, and estimate a probability for each possible role [15]. **PropBank** then annotated the verbs of the Penn Treebank with predicate-specific numbered arguments: each verb sense has a frameset in which `Arg0` is typically the agent-like participant and `Arg1` the patient or theme, while higher numbers are defined per verb; adjuncts receive `ArgM` labels such as `ArgM-NEG` for negation and `ArgM-MOD` for modals [16]. The numbering is the point: in "an amendment offered by Rep. X," the amendment is still `Arg1` of *offer* although it is the grammatical subject [16]. The CoNLL-2005 shared task standardized SRL evaluation on PropBank: a predicted argument counts as correct only if its span and role both match, and systems are compared by precision, recall and F1 over arguments [17].

Semantic roles provide a bridge across constructions, not a guarantee of universal agreement. Different resources can label the same participant with predicate-specific or more general roles. Before combining training corpora, compare their definitions and examples. A mechanical label rename can conceal differences about instruments, causes, beneficiaries and implicit participants.

SRL as predicate-conditioned tagging

He et al. cast SRL as BIO sequence tagging, once per predicate: the input marks which word is the predicate, and a deep highway BiLSTM with **constrained decoding** — forbidding, for example, an `I-` tag without a preceding `B-` — labels argument spans [18]. Their eight-layer ensemble reported 83.2 F1 on the CoNLL-2005 test set and found that deep models recover long-distance dependencies but still make surprisingly obvious errors, leaving room for syntactic information to help [18]. With a pretrained encoder, the same design becomes the token-labeling head from NLP 11 plus a predicate indicator. The snippet converts role spans to tag sequences and back, and rejects an invalid sequence.

```
tokens = "Mina said the sensor was repaired by Lena".split()
# One frame per predicate: (start, end) token spans, end exclusive.
frames = {
    "said":     {"ARG0": (0, 1), "ARG1": (2, 8)},
    "repaired": {"ARG1": (2, 4), "ARG0": (6, 8)},
}

def to_bio(frame, n):
    tags = ["O"] * n
    for role, (i, j) in frame.items():
        tags[i] = "B-" + role
        tags[i + 1:j] = ["I-" + role] * (j - i - 1)
    return tags

def from_bio(tags):
    spans, start, role = {}, None, None
    for k, tag in enumerate(tags + ["O"]):
        if start is not None and not tag.startswith("I-" + role):
```

```
            spans[role] = (start, k); start = None
        if tag.startswith("B-"):
            start, role = k, tag[2:]
        elif tag.startswith("I-") and start is None:
            raise ValueError(f"I- tag without B- at position {k}")
    return spans

for predicate, frame in frames.items():
    tags = to_bio(frame, len(tokens))
    tags[tokens.index(predicate)] = "B-V"
    print(f"{predicate:9s}", " ".join(tags))
    assert {r: s for r, s in from_bio(tags).items() if r != "V"} == frame
try:
    from_bio(["O", "I-ARG1", "I-ARG1"])
except ValueError as error:
    print("invalid:", error)
```

```
said      B-ARG0 B-V B-ARG1 I-ARG1 I-ARG1 I-ARG1 I-ARG1 I-ARG1
repaired  O O B-ARG1 I-ARG1 O B-V B-ARG0 I-ARG0
invalid: I- tag without B- at position 1
```

The same sentence yields two frames. For *said*, Mina is the speaker and the whole reported clause is `ARG1`. For *repaired*, Lena is the repairer and the sensor is `ARG1` although it is the grammatical subject. An extractor that reads only the *repaired* frame would record a completed repair as fact; the *said* frame shows that the repair is Mina's report. That is exactly the attribution distinction the relation records below must keep.

Scope in semantic representations

Plain version. Small words such as "every," "some" and "not" can change a statement more than the names do. A semantic parser must record what they modify, or its output answers the wrong question.

NLP 02 introduces quantifier scope and negation with a worked countermodel. Here we look at what scope means for a task output. "Every technician checked a sensor" has a reading in which each technician may check a different sensor and a reading in which one sensor was checked by all. "Mina did not check every sensor" has the form

$$\neg \forall z\,[S(z) \Rightarrow C(m,z)] \quad \equiv \quad \exists z\,[S(z) \wedge \neg C(m,z)].$$

Here z ranges over entities, $S(z)$ means sensor, $C(m,z)$ means Mina checked that entity, m identifies Mina, $\neg$ means negation and $\equiv$ means logical equivalence. A counterexample sensor suffices to make the universal false. "Mina checked no sensor" instead rules out every successful check. Negation's position determines which evidence can refute the statement.

A document database usually lacks complete knowledge: no extracted inspection record may mean unreported, not false. This difference matters when answering a universal question. A

list of observed inspections can support "these sensors were checked," while establishing "every sensor was checked" also requires the full relevant sensor inventory and complete coverage. Conversely, finding one explicitly unchecked sensor is enough to refute completeness. Scope reasoning and evidence completeness must be considered together. Likewise, "No evidence of failure was found" describes an observation or search result; it should not automatically become "failure is impossible."

In a restricted extractor, unsupported scope constructions should trigger abstention or review. A rule that handles "did not repair" can still misread "did not say that Mina repaired": the system must determine what is negated, which is exactly what PropBank's `ArgM-NEG` attaches to a specific predicate [16].

Composition is a calculation over roles and scope

A useful way to understand composition is to ask what remains incomplete at each phrase. `Min a` identifies a participant. `repaired` requires participants in specified roles. `repaired a sensor` fixes the object requirement through an existential condition and leaves the actor to be supplied. Combining the subject then closes that requirement. This order of combination explains why identifying the same three content words is insufficient: their roles must be bound correctly [1].

A restricted logical reading of "Mina repaired a sensor" is:

$$\exists z\,[S(z) \wedge R(m,z)].$$

Here z ranges over possible entities, $S(z)$ means that z is a sensor, $R(m,z)$ means that participant m repaired z, and m identifies Mina. The existential quantifier states that at least one suitable entity exists; it does not identify which sensor or imply that every sensor was repaired. This simple reading omits tense and event time intentionally. Those are additional semantic fields, not hidden facts supplied by the notation.

"Only Mina repaired the sensor" adds a restriction on alternatives. "Mina tried to repair the sensor" describes an attempt without guaranteeing a completed repair. "Mina stopped repairing the sensor" changes the event's temporal interpretation. A record containing only actor and object cannot express those distinctions. This gives a concrete criterion for choosing a representation: list the questions that must be answered, then check whether two source statements requiring different answers collapse into the same record. If both an attempted repair and a completed repair become `repair(Mina, sensor)`, the schema is too coarse for a completion query. A richer encoder cannot prevent that loss after serialization.

Relations, events and provenance

Plain version. A relation links entities; an event organizes participants around an occurrence. Both need evidence, and both can be wrong even when every entity span is right.

A binary relation record can be written:

$$q = (r, h, t, s).$$

Here q is the record; r is the relation type; h is the head entity or mention; t is the tail entity or mention; and s identifies supporting evidence. The letters head and tail define an ordering convention, not a linguistic claim. For an employment relation, reversing the employee and employer changes the record. Define direction in the schema and evaluate it.

Event extraction adds a trigger or event type with multiple roles, such as buyer, seller, item, amount and time — semantic role labeling restricted to an application's schema. A transaction sentence may mention several prices, only one of which is the actual amount paid. Jointly selecting role assignments can prevent some contradictions, but it cannot resolve an ambiguous source without additional context. Preserve alternatives or route ambiguity to review when the application permits it.

Provenance should include document identity, source version and supporting span. Without a version, offsets can point into a later edited document; without attribution — the *said* frame above — a quoted allegation becomes an asserted fact. These are properties of the record pipeline, not failures that a larger encoder automatically fixes.

Decompose evaluation by stage — mention detection, category prediction, relation selection and the final supported record — and evaluate relations against predicted mentions as well as gold mentions: perfect input mentions during evaluation can conceal the failure rate of the deployed pipeline.

Discourse, mentions and coreference

Plain version. Meaning continues across sentences. Coreference resolution decides which mentions refer to the same entity, so that facts about "the pump" and "it" end up on the same record.

Discourse is language organized across clauses or sentences. Consider "The pump warmed. The operator stopped it." The two sentences introduce an event, an agent and a reference to an object. They may suggest a response to warming, but they do not necessarily prove that warming caused the stop. Compare "The pump stopped **because** the valve closed," "The pump stopped **after** the valve closed," and "The pump stopped **although** the valve remained open." The first presents a causal explanation, the second orders events without establishing causation, and the third marks a contrast with an expectation [1]. A summarizer or extractor that inserts "because" has added a claim. Relations also have scope: in "Mina said the pump stopped because the valve closed," the explanation may be part of Mina's reported account rather than an independently established finding.

A mention is a text span that refers to an entity, event or concept. An antecedent is an earlier expression that helps interpret another mention. Coreference groups mentions that refer to the same entity. Similar wording does not necessarily imply identity: two different pumps may both be called "the backup pump."

An antecedent-ranking model

A span-based coreference model can score possible antecedents for a mention, as illustrated by Lee et al. [19]. Abstractly,

$$p(a_i = j \mid d) = \frac{\exp s(i, j, d)}{\sum_{k \in A_i} \exp s(i, k, d)}.$$

Here d is the document or conversation; i indexes the current mention; A_i is the set of candidate antecedents, including a null option; a_i is the selected antecedent variable; and $s(i, j, d)$ is a learned compatibility score. The null option permits a mention to introduce a new entity or remain unlinked. Candidate pruning changes what the model can consider. Lee et al.'s model separates the contributions [19]:

$$s(i, j) = s_m(i) + s_m(j) + s_a(i, j) \quad (j \neq \epsilon), \qquad s(i, \epsilon) = 0.$$

Here i is the current candidate span, j is an earlier candidate span, ϵ is the null antecedent, s_m scores whether a span is a mention, and s_a scores pair compatibility. Fixing the null score to zero gives the real links a common reference. Mention pruning saves pairwise computation but can remove the only valid antecedent before linking begins.

The model must detect mentions as well as link them. If an earlier entity span was discarded, a later linking decision cannot recover it merely by assigning better scores among remaining candidates. Separate detection recall from linking accuracy when diagnosing failures.

Clusters complicate supervision. If `Pump A`, `the pump` and `it` refer to one entity, the final mention can correctly link to either earlier mention. A loss that arbitrarily rewards only the nearest one penalizes another valid link. A cluster-based training target instead rewards their combined probability:

$$\mathcal{L}_i = -\log \sum_{j \in G_i} p(a_i = j \mid d).$$

Here $\mathcal{L}_i$ is the loss for candidate span i, G_i is its nonempty set of valid retained antecedents under the annotation and pruning policy, a_i is the antecedent variable, d is the document, and $p(a_i = j \mid d)$ is its conditional antecedent probability. The null option belongs in G_i when the span has no valid earlier antecedent. This is marginalization over valid links, as used in the original model [19]. At inference, connecting mentions through antecedent links forms entity groups. A single erroneous bridge can merge two otherwise coherent groups, and the wrongly merged cluster can later assign several events to the wrong pump.

A recency baseline and its failure

The following hand-written baseline uses explicit compatibility and recency. It is deliberately small enough to expose a failure. Device categories are supplied as data; the code does not infer attributes from personal names or social stereotypes.

```
entities = [
```

```
    {"id": "pump_a", "category": "device", "turn": 1},
    {"id": "valve_b", "category": "device", "turn": 2},
    {"id": "operator", "category": "person", "turn": 2},
]
def candidates(category):
    return [e for e in entities if e["category"] == category]
def recent(category):
    choices = candidates(category)
    return max(choices, key=lambda e: e["turn"])["id"] if choices else None
print("Recency baseline for it:", recent("device"))
assert recent("device") == "valve_b"
# In a synthetic annotation, the speaker meant the earlier pump.
intended = "pump_a"
assert recent("device") != intended
# A cautious application preserves ambiguity instead of hiding the choice.
possible = {e["id"] for e in candidates("device")}
resolution = next(iter(possible)) if len(possible) == 1 else None
assert resolution is None and possible == {"pump_a", "valve_b"}
```

The baseline prints `valve_b`, while the constructed speaker intention points to the earlier pump. A cue cannot establish identity in every case. A clarifying question such as "Do you mean Pump A or Valve B?" preserves the uncertainty explicitly; NLP 08 develops clarification in dialogue.

Context can include deixis: "this," "here" or "now" depend on a speaker, place, time or interface selection, so store that context with the utterance, and do not score a system as wrong for refusing an inference the input cannot support. Quotation adds constraints too: `I` in an operator's quoted statement refers to that operator, and "the other pump" presupposes a contrast set that a recency rule does not define.

Online only: Coreference — End-to-end Neural Coreference Resolution — https://blog.shyandram.dev/en/learn/nlp/syntactic-semantic-tasks/papers/coreference–end-to-end-neural-coreference-resolution/

Evaluating coreference: MUC, B³, CEAF and CoNLL F1

Plain version. A coreference output is a partition of mentions into entities. Different metrics count agreement between two partitions in different units — links, mentions or entities — so no single one tells the whole story.

Let K be the key (gold) partition and R the response partition.

- **MUC** is link-based [20]. For each key entity S, let $p(S)$ be the partition of S induced by the response, with unmatched mentions as singletons. Recall is $\sum_S (|S| - |p(S)|) / \sum_S (|S| - 1)$: the number of key links the response preserves over the minimum number of links needed. Precision swaps the roles of key and response.
- **B³** is mention-based. For each mention m, compare the key entity K_m and response entity R_m containing it: recall averages $|K_m \cap R_m|/|K_m|$ and precision averages $|K_m \cap R_m|/|R_m|$ over mentions [23].

- **CEAF** is entity-based [21]. It finds the best one-to-one alignment between key and response entities, a maximum bipartite matching. With the entity similarity $\phi(K,R) = 2|K \cap R|/(|K|+|R|)$ (the entity-based variant CEAF_e), recall is the total aligned similarity divided by the number of key entities and precision divides by the number of response entities.

The CoNLL-2012 shared task scored systems by the **unweighted mean** of MUC, B^3 and CEAF_e F1, on the grounds that each represents a different dimension [23]. Pradhan et al. later published a reference scorer because inconsistent implementations, especially for predicted mentions that do not match gold mentions, had produced incomparable numbers [22]. The snippet computes all three for the response in Figure 5, where "it" is wrongly merged with the valve.

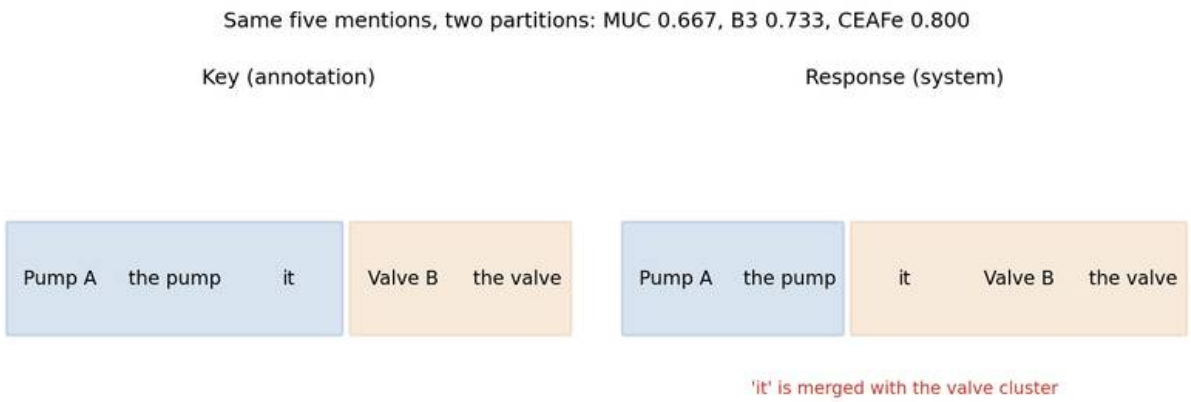


Figure 5 — One wrong link, three different penalties. The scores are computed by the snippet below.

```
import numpy as np
from scipy.optimize import linear_sum_assignment

# Mentions: a="Pump A", b="the pump", c="it", d="Valve B", e="the valve"
key = [{"a", "b", "c"}, {"d", "e"}]
response = [{"a", "b"}, {"c", "d", "e"}]          # "it" wrongly joined to the valve

def f1(p, r):
    return 2 * p * r / (p + r) if p + r else 0.0

def muc(gold, pred):                                  # link-based
    def recall(g, p):
        def parts(s):  # pieces of s cut by p; uncovered mentions are singletons
            covered = set().union(*p)
            return sum(1 for t in p if s & t) + len(s - covered)
        return sum(len(s) - parts(s) for s in g) / sum(len(s) - 1 for s in g)
    return recall(pred, gold), recall(gold, pred)    # precision, recall

def b_cubed(gold, pred):                              # mention-based
    def score(a, b):
        cluster_of = {m: s for s in b for m in s}
        return np.mean([len(s & cluster_of.get(m, set())) / len(s)
                        for s in a for m in s])
```

```
    return score(pred, gold), score(gold, pred)

def ceaf_e(gold, pred):                              # entity-based
    phi = np.array([[2 * len(g & p) / (len(g) + len(p)) for p in pred] for g in gold])
    rows, cols = linear_sum_assignment(-phi)
    total = phi[rows, cols].sum()
    return total / len(pred), total / len(gold)

scores = {name: f1(*fn(key, response)) for name, fn in
          [("MUC", muc), ("B3", b_cubed), ("CEAFe", ceaf_e)]}
for name, value in scores.items():
    print(f"{name:6s} F1 = {value:.3f}")
print(f"CoNLL F1 = {np.mean(list(scores.values())):.3f}")
```

```
MUC    F1 = 0.667
B3     F1 = 0.733
CEAFe  F1 = 0.800
CoNLL F1 = 0.733
```

Check MUC by hand: the key needs $2 + 1 = 3$ links; the response splits the pump entity into two pieces and keeps the valve entity whole, preserving $1 + 1 = 2$ links, so recall is $2/3$, and by symmetry precision is $2/3$. B^3 penalizes each affected mention, and CEAF_e still aligns each key entity with a mostly-correct response entity. One error, three numbers: that is why the shared task averaged them, and why an application should also check the records that consume the clusters.

Scores are interpretable only relative to the annotation policy: whether events and associative relations are linked, whether singletons count, and how ambiguity is annotated.

Semantic parsing and AMR

Plain version. Semantic parsing maps a sentence to a meaning representation that a program can use: a logical form, a database query or a graph. Abstract Meaning Representation (AMR) is a widely used whole-sentence graph format.

Banarescu et al. designed AMR for "sembanking": AMRs are rooted, labeled, directed graphs that are easy for people to read and programs to traverse; they aim to abstract away from syntactic idiosyncrasies, assigning the same AMR to sentences with the same basic meaning, and they make extensive use of PropBank framesets [24]. When one entity plays several roles, AMR uses **re-entrancy** — a node with multiple parents — so the graph is not a tree. In "the operator wants to stop the pump," the operator is `ARG0` of both `want-01` and `stop-01`. AMR deliberately does not specify how the graph is derived from the words, which makes alignment between words and nodes a separate problem [24].

Smatch evaluates AMR parsing by viewing each graph as a conjunction of triples, either `instance(variable, concept)` or `relation(variable1, variable2)`, and reporting precision, recall and F1 of matched triples under the variable mapping that maximizes the match [25]. Finding that mapping is NP-complete, so the paper provides an integer-linear-programming method and

a hill-climbing heuristic with random restarts [25]. For a four-variable graph, we can simply try every mapping:

```
import itertools

# "the operator wants to stop the pump" vs a parse that loses who stops.
gold = {("instance", "w", "want-01"), ("instance", "o", "operator"),
        ("instance", "s", "stop-01"), ("instance", "p", "pump"),
        ("ARG0", "w", "o"), ("ARG1", "w", "s"),
        ("ARG0", "s", "o"), ("ARG1", "s", "p")}
pred = {("instance", "a", "want-01"), ("instance", "b", "operator"),
        ("instance", "c", "stop-01"), ("instance", "d", "pump"),
        ("ARG0", "a", "b"), ("ARG1", "a", "c"), ("ARG1", "c", "d")}

def rename(triples, mapping):
    return {(r, mapping.get(x, x), mapping.get(y, y)) for r, x, y in triples}

pred_vars, gold_vars = sorted({t[1] for t in pred}), sorted({t[1] for t in gold})
best = max(len(rename(pred, dict(zip(pred_vars, perm))) & gold)
           for perm in itertools.permutations(gold_vars))
precision, recall = best / len(pred), best / len(gold)
print("matched triples:", best)
print(f"Smatch P={precision:.3f} R={recall:.3f}
 ↪  F1={2*precision*recall/(precision+recall):.3f}")
```

```
matched triples: 7
Smatch P=1.000 R=0.875 F1=0.933
```

The predicted graph is missing one triple — the re-entrant `ARG0` edge that says the operator is also the one stopping. Smatch still gives 0.933. For a safety application, "who stops the pump" may be the whole question. Like every aggregate metric in this chapter, Smatch should be paired with targeted checks on the relations that matter.

How pretrained models changed these tasks

Plain version. Pretrained encoders turned parsers, role labelers and coreference systems into "encoder plus small structured decoder." They did not remove the need for the decoder, the annotation scheme or careful evaluation.

Three shifts stand out. First, **features were replaced by contextual vectors**. Kitaev and Klein's parser improved from 93.55 to 95.13 F1 when it used pretrained word representations [6]; Joshi et al. applied BERT to coreference and reported a gain of 3.9 F1 over the previous state of the art on OntoNotes, noting remaining weaknesses in document-level context, conversations and mention paraphrasing [26]. SpanBERT changed the pretraining objective itself — masking contiguous spans and training span-boundary representations to predict the span content — and reported 79.6% F1 on OntoNotes coreference along with 94.6 and 88.7 F1 on SQuAD 1.1 and 2.0 [27]. Span-shaped tasks benefited from span-shaped pretraining.

Second, **the structures turned out to be partly present inside the encoders**. Hewitt and Manning's structural probe found a linear transformation under which squared distance between word vectors encodes distance in the parse tree, for ELMo and BERT but not for baselines [28]. Tenney, Das and Pavlick found that BERT represents the steps of the traditional pipeline in an interpretable order across layers — part-of-speech tagging, parsing, named entities, semantic roles, then coreference — while sometimes revising lower-level decisions using higher-level information [29]. These are probing results: they show that information is linearly recoverable, not that the model uses it the way a parser does.

Third, **the pipeline collapsed into fine-tuning and prompting**. Encoders supply evidence for tags and trees, and large language models can be prompted to return structures [30]. An output validator remains necessary: a plausible bracketed string can omit a token, reorder words or fail to form a connected tree.

Modern view

Surveys and their taxonomies. Zhang's survey of syntactic–semantic parsing focuses on the two most popular formalizations, constituent and dependency parsing, reviews representative models for each and for semantic dependency graphs, and covers cross-domain, cross-lingual and joint parsing as well as corpus development [31]. Poesio et al.'s review of computational models of anaphora traces coreference from pre-neural work through current neural models, surveys the annotated datasets, and treats less-studied phenomena such as bridging reference and discourse deixis [32]. Chen et al.'s SRL survey, accepted at Artificial Intelligence Review, proposes a four-dimensional taxonomy — model architectures, syntax feature modeling, application scenarios and multimodal extensions — analyzes when syntactic features help, and examines the complementary roles of LLMs and specialized SRL systems [34]. Zhao et al.'s LLM survey distinguishes model training, adaptation, utilization and capacity evaluation [30]; structure can play several roles within that taxonomy — a supervised output, an internal representational probe, a constraint on generation or a diagnostic challenge. Chang et al.'s evaluation survey further separates the tasks being measured from the benchmarks and protocols used [33].

Recent original work (2023–2026). Tian, Xia and Song found that LLMs are effective at shallow parsing (chunking) but struggle to produce correct full constituency trees, and improved them by prompting for chunks first and filtering them [35]. Hale and Stanojević tested a candidate syntactic universal across typologically diverse languages and found that the model they studied respected it only in some higher-resource languages [36], illustrating why an English-focused structural test cannot establish uniform multilingual competence. For coreference, Le and Ritter found that prompting instruction-tuned models can outperform unsupervised coreference systems but relies on high-quality mention detection [37]; Porada, Zou and Cheung controlled for the pretrained language model across five coreference systems and found that encoder-based models outperformed more recent decoder-based ones at equal model size, and that controlling for the encoder removes most, but not all, of the reported F1 gains of the preceding five years [38]. For semantic representations, Jin et al. found that feeding AMR into LLM chain-of-thought prompting generally hurt more than it helped across five tasks, with errors concentrated in multi-

word expressions, named entities and the final inference step [39].

Distinguish valid structure from structural generalization

These studies motivate two different tests. First, can a system produce a legal structure for this annotation scheme? Second, does its choice respond to the grammatical distinction rather than a familiar word sequence? A structural validator addresses the first and cannot establish the second. A model that always returns one conventional attachment can pass format checks while failing a controlled contrast.

Construct comparisons that hold lexical content nearly fixed while changing the role-relevant construction (active versus passive, modifier relocation, a relative clause), then vary vocabulary while holding the construction fixed. If an unfamiliar noun changes the whole structure, lexical coverage is interfering; if a grammatical change leaves the interpretation untouched, a familiar lexical pattern dominates. Finally check whether mistakes alter the semantic roles needed downstream.

Longer context changes the reference decision, not only memory size

A longer history can restore a missing antecedent, but it also adds compatible candidates; in an antecedent softmax, a new candidate changes the denominator even when existing link scores stay fixed, so thresholds calibrated on short histories need not transfer. Compare a short history, a long irrelevant history and a long history containing another device of the same type to separate useful context from merely more text.

Open problems. Annotation schemes differ; low-resource languages lack enough independently checked trees; segmentation interacts with syntax; spontaneous dialogue contains fragments and repairs; coreference across long documents and conversations remains hard; and a system can learn benchmark-specific conventions without robust compositional behavior. Preserving alternative analyses, testing structural minimal pairs, controlling for the encoder when comparing architectures [38], and measuring downstream consequences offer concrete ways to study those limits.

Key takeaways

- A grammar's failure to parse is a coverage result, not a judgment of a speaker; charts share spans, but enumerating all trees can still grow exponentially.
- Best-tree (max) and total-probability (sum) answer different questions; rule probabilities alone cannot see lexical evidence, which is what neural span and arc scorers add.
- Transition parsers make linear-time local decisions; graph parsers score all arcs and need a tree decoder, because independent head choices can form cycles.
- Each structure has its own metric — labeled brackets, UAS/LAS, argument F1, MUC/B^3/CEAF, Smatch — and each needs a stated denominator and alignment policy.

- Semantic roles survive grammatical alternations; PropBank numbers arguments per predicate and marks negation and modality as adjuncts.
- Preserve scope, attribution and provenance in extracted records; a coarse schema loses distinctions that no encoder can recover afterwards.
- Coreference is detection plus linking; one wrong bridge can merge entities, and an explicit ambiguous outcome is better than a confident wrong one.
- Pretrained encoders improved and simplified these systems, but validity still comes from the decoder and evaluation from the annotation scheme.

Exercises

1. **A grammar extension.** Extend the chart parser to accept "we saw cameras" and a new preposition. Then add an unseen noun and explain which lexical change is required.

   Hint

   The existing binary rules already support the shorter sentence. Coverage of terminals is a separate issue from phrase combination.

2. **Recognition versus enumeration.** Replace tree lists with Boolean categories in the chart. Compare what the parser can still answer and what information it loses. Then change the PCFG snippet so that the noun attachment wins with the original rule probabilities, by adding a lexicalized rule score for `robots with`.

   Hint

   Acceptance remains available without trees; recovering alternative attachments needs backpointers. For the lexicalized variant, multiply the $NP \rightarrow NP\,PP$ score by a factor that depends on the head noun.

3. **A structural validator and a non-projective tree.** Write checks for token yield, one head per word, no self-links, a unique root and absence of cycles in a basic dependency tree. Then construct a valid tree for a short sentence whose arcs cross, and explain why the arc-standard oracle cannot produce it.

   Hint

   Following head pointers from each word should reach the root without visiting a word twice. Arc-standard only attaches the top two stack items, so every arc it creates spans a contiguous, already-completed region.

4. **Two readings, two records.** Write two interpretations of "A technician inspected every sensor." Design a relation schema that can store either reading, and an evidence rule for answering "Was every sensor inspected?" from a document collection.

Hint

Compare an existential quantifier outside a universal quantifier with the reverse order. The answer requires a sensor inventory and coverage information, not only extracted inspection events.

5. **Coreference metrics.** Construct a response that merges all five mentions of Figure 5 into one entity, and another that leaves all five as singletons. Compute MUC, B^3 and $CEAF_e$ for both before running the code, then explain which metric is most lenient to each mistake.

Hint

MUC counts links, so a single giant cluster preserves every key link. B^3 penalizes each mention whose cluster contains wrong members. Singletons have no links at all.

6. **Bilingual deixis.** Write a bilingual message containing a demonstrative, a relative date and an indirect request, for example mixing English and Chinese. Describe two contexts that change its interpretation, and list the evidence a coreference or extraction system would need to distinguish them.

Hint

A forwarded quotation changes whose viewpoint controls "I," "here" and "tomorrow." An ability test and a maintenance conversation can give the same question different purposes.

References

1. Susan McRoy, *Principles of Natural Language Processing*, Chapters 3–5 and 7 and the chart-parsing appendix, open textbook, 2021. Open textbook
2. Daniel Jurafsky and James H. Martin, *Speech and Language Processing*, 3rd ed. draft (August 19, 2026), Chapters 19, 20, 22 and 24. Book site
3. E. Black, S. Abney, D. Flickenger, C. Gdaniec, R. Grishman, P. Harrison, D. Hindle, R. Ingria, F. Jelinek, J. Klavans, M. Liberman, M. Marcus, S. Roukos, B. Santorini and T. Strzalkowski, "A Procedure for Quantitatively Comparing the Syntactic Coverage of English Grammars," Speech and Natural Language Workshop, 1991. Paper
4. Mitchell P. Marcus, Beatrice Santorini and Mary Ann Marcinkiewicz, "Building a Large Annotated Corpus of English: The Penn Treebank," Computational Linguistics 19(2), 1993. Paper
5. Mitchell Stern, Jacob Andreas and Dan Klein, "A Minimal Span-Based Neural Constituency Parser," ACL, 2017. Paper
6. Nikita Kitaev and Dan Klein, "Constituency Parsing with a Self-Attentive Encoder," ACL, 2018. Paper
7. Nikita Kitaev, Steven Cao and Dan Klein, "Multilingual Constituency Parsing with Self-Attention and Pre-Training," ACL, 2019. Paper
8. Marie-Catherine de Marneffe, Christopher D. Manning, Joakim Nivre and Daniel Zeman, "Universal Dependencies," Computational Linguistics 47(2), 2021. Paper

9. Joakim Nivre, “Algorithms for Deterministic Incremental Dependency Parsing,” Computational Linguistics 34(4), 2008. Paper
10. Danqi Chen and Christopher Manning, “A Fast and Accurate Dependency Parser using Neural Networks,” EMNLP, 2014. Paper
11. Ryan McDonald, Fernando Pereira, Kiril Ribarov and Jan Hajič, “Non-Projective Dependency Parsing using Spanning Tree Algorithms,” HLT-EMNLP, 2005. Paper
12. Eliyahu Kiperwasser and Yoav Goldberg, “Simple and Accurate Dependency Parsing Using Bidirectional LSTM Feature Representations,” Transactions of the ACL 4, 2016. Paper
13. Timothy Dozat and Christopher D. Manning, “Deep Biaffine Attention for Neural Dependency Parsing,” ICLR, 2017. Paper
14. Daniel Zeman, Jan Hajič, Martin Popel, Martin Potthast, Milan Straka, Filip Ginter, Joakim Nivre and Slav Petrov, “CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies,” CoNLL Shared Task, 2018. Paper
15. Daniel Gildea and Daniel Jurafsky, “Automatic Labeling of Semantic Roles,” Computational Linguistics 28(3), 2002. Paper
16. Martha Palmer, Daniel Gildea and Paul Kingsbury, “The Proposition Bank: An Annotated Corpus of Semantic Roles,” Computational Linguistics 31(1), 2005. Paper
17. Xavier Carreras and Lluís Màrquez, “Introduction to the CoNLL-2005 Shared Task: Semantic Role Labeling,” CoNLL, 2005. Paper
18. Luheng He, Kenton Lee, Mike Lewis and Luke Zettlemoyer, “Deep Semantic Role Labeling: What Works and What’s Next,” ACL, 2017. Paper
19. Kenton Lee, Luheng He, Mike Lewis and Luke Zettlemoyer, “End-to-end Neural Coreference Resolution,” EMNLP, 2017. Paper
20. Marc Vilain, John Burger, John Aberdeen, Dennis Connolly and Lynette Hirschman, “A Model-Theoretic Coreference Scoring Scheme,” MUC-6, 1995. Paper
21. Xiaoqiang Luo, “On Coreference Resolution Performance Metrics,” HLT-EMNLP, 2005. Paper
22. Sameer Pradhan, Xiaoqiang Luo, Marta Recasens, Eduard Hovy, Vincent Ng and Michael Strube, “Scoring Coreference Partitions of Predicted Mentions: A Reference Implementation,” ACL, 2014. Paper
23. Sameer Pradhan, Alessandro Moschitti, Nianwen Xue, Olga Uryupina and Yuchen Zhang, “CoNLL-2012 Shared Task: Modeling Multilingual Unrestricted Coreference in OntoNotes,” EMNLP-CoNLL Shared Task, 2012. Paper
24. Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer and Nathan Schneider, “Abstract Meaning Representation for Sembanking,” Linguistic Annotation Workshop, 2013. Paper
25. Shu Cai and Kevin Knight, “Smatch: an Evaluation Metric for Semantic Feature Structures,” ACL, 2013. Paper
26. Mandar Joshi, Omer Levy, Luke Zettlemoyer and Daniel Weld, “BERT for Coreference Resolution: Baselines and Analysis,” EMNLP-IJCNLP, 2019. Paper
27. Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S. Weld, Luke Zettlemoyer and Omer Levy, “SpanBERT: Improving Pre-training by Representing and Predicting Spans,” Transactions of the ACL 8, 2020. Paper
28. John Hewitt and Christopher D. Manning, “A Structural Probe for Finding Syntax in Word Representations,” NAACL-HLT, 2019. Paper

29. Ian Tenney, Dipanjan Das and Ellie Pavlick, “BERT Rediscovers the Classical NLP Pipeline,” ACL, 2019. Paper
30. Wayne Xin Zhao et al., “A Survey of Large Language Models,” Frontiers of Computer Science, 2026; online manuscript updated March 2026. Survey
31. Meishan Zhang, “A survey of syntactic-semantic parsing based on constituent and dependency structures,” Science China Technological Sciences 63, 2020. Survey
32. Massimo Poesio, Juntao Yu, Silviu Paun, Abdulrahman Aloraini, Pengcheng Lu, Janosch Haber and Derya Cokal, “Computational Models of Anaphora,” Annual Review of Linguistics 9, 2023. Survey
33. Yupeng Chang et al., “A Survey on Evaluation of Large Language Models,” ACM Transactions on Intelligent Systems and Technology, 2024. Survey
34. Huiyao Chen, Meishan Zhang, Jing Li, Lilja Øvrelid, Jan Hajič, Hao Fei and Min Zhang, “A Systematic Survey of Semantic Role Labeling in the Era of Pretrained Language Models,” arXiv: 2502.08660, 2025; accepted at Artificial Intelligence Review. Survey
35. Yuanhe Tian, Fei Xia and Yan Song, “Large Language Models are No Longer Shallow Parsers,” ACL, 2024. Paper
36. John T. Hale and Miloš Stanojević, “Do LLMs learn a true syntactic universal?” EMNLP, 2024. Paper
37. Nghia T. Le and Alan Ritter, “Are Large Language Models Robust Coreference Resolvers?” arXiv:2305.14489, 2023. Paper
38. Ian Porada, Xiyuan Zou and Jackie Chi Kit Cheung, “A Controlled Reevaluation of Coreference Resolution Models,” LREC-COLING, 2024. Paper
39. Zhijing Jin, Yuen Chen, Fernando Gonzalez Adauto, Jiarui Liu, Jiayi Zhang, Julian Michael, Bernhard Schölkopf and Mona Diab, “Analyzing the Role of Semantic Representations in the Era of Large Language Models,” NAACL, 2024. Paper
40. NLTK contributors, `nltk.grammar.PCFG`, API documentation, accessed 2026-11-05. Normalization contract

Part II

Pattern Recognition

PR 01 • Patterns, Probability and Bayes Decisions

Define a recognition pipeline, separate likelihoods from posteriors, and derive minimum-error, minimum-risk and reject decisions, Gaussian discriminants and ROC analysis.

Online edition: https://blog.shyandram.dev/en/learn/pattern-recognition/bayes-decisions/

What you'll learn

- Turn a recognition problem into observations, features, predictions and actions, and decide what one "sample" means.
- Distinguish a class prior, a class likelihood and a class posterior, and see how a change in the prior moves a decision boundary.
- Derive the minimum-error rule, the minimum-risk rule for any loss matrix, and Chow's reject rule.
- Write classifiers as discriminant functions and derive the linear (LDA) and quadratic (QDA) boundaries of the Gaussian case.
- Read ROC and precision–recall curves, connect them to the Neyman–Pearson criterion, and choose an operating point from costs.

The big picture

A camera records a manufactured component. A system measures its shape, estimates whether it is defective, and decides whether to send it for manual inspection. These are different stages. The recorded pixels are **observations**; shape measurements are **features**; a defect probability is a **prediction**; sending the part to inspection is an **action**. Bayes decision theory is the part of pattern recognition that connects the last two: given what we believe, what should we do?

> **Explain like I'm five**
>
> A sorting machine looks at a fruit and guesses whether it is ripe. Guessing is one job; choosing a basket is another. If most fruit in the box is ripe, the machine should lean towards "ripe" even when the fruit looks a bit odd. If throwing away a good fruit costs much more than asking a person to check it, the machine should sometimes say "I'm not sure, please look at this one." Bayes decision theory is just the arithmetic of these three ideas: how common each kind is, what the evidence says, and how much each mistake hurts.

This track follows the classic pattern-recognition textbooks: Duda, Hart and Stork's *Pattern Classification* [1], Bishop's *Pattern Recognition and Machine Learning* [2], and Theodoridis and Koutroumbas's *Pattern Recognition* [3], with Hastie, Tibshirani and Friedman [4] for the statistical view. The chapters use original examples and derivations; the textbooks are where to go for longer proofs.

Bayes decision theory comes first in all of them because every later method can be judged against it: a classifier is good to the extent that it approximates the decision the Bayes rule would take if we knew the true distributions.

Recognition is not limited to images. An observation can be a sound recording, a sequence of pen strokes or a document. Classification predicts categories, regression predicts quantities, and clustering discovers groups without labels. This chapter is about classification, but the way of thinking (model the uncertainty, then choose the action with the lowest expected cost) carries over to all three.

Start with the experiment

Plain version. Before any formula, write down what one sample is and how the data were collected. Most surprising failures of recognition systems come from this step, not from the classifier.

One image, one patient, one video and one crop of a video are different **sampling units**. If twenty frames come from the same recording, treating them as twenty independent cases can make a test look much easier than deployment, because the test frames are near-copies of training frames.

Precise version. A recognition pipeline can be written as

$$o \longrightarrow x = \phi(o) \longrightarrow p(c \mid x) \longrightarrow a(x).$$

Here o is a raw observation (an image, a waveform); ϕ is the preprocessing and feature extraction that maps it to a feature vector $x \in \mathbb{R}^d$; $c \in \{c_1, \dots, c_K\}$ is one of K classes; $p(c \mid x)$ is the model's belief about the class; and $a(x)$ is the action finally taken. Learning adjusts the parameters of ϕ and of the probability model from **training** samples; **validation** samples choose settings such as thresholds and regularization; a held-out **test** set estimates the performance of the whole chosen procedure once (PR 09 treats this carefully: validation and ensembles).

Handcrafted features (edge histograms, shape moments) and learned embeddings (the output of a pretrained network) occupy the same slot ϕ. Changing the feature extractor changes which information the classifier receives. Information removed by cropping, quantization or pooling cannot be recovered by any classifier downstream, however clever. In decision-theoretic language: the Bayes error of the problem is defined *relative to* the features, and a poor ϕ raises the floor that no classifier can go below.

Bayes' rule connects evidence and belief

Plain version. Bayes' rule says how to update "how common is each class" into "how likely is each class for *this* object", using "how typical is this measurement for each class".

Precise version. Let the classes be $c_1, \dots, c_K$. Bayes' rule gives

$$P(c_k \mid x) = \frac{p(x \mid c_k)\, P(c_k)}{\sum_{j=1}^{K} p(x \mid c_j)\, P(c_j)}.$$

- The **prior** $P(c_k)$ is the probability of class c_k before examining this observation, for example the fraction of defective parts on the production line.
- The **likelihood** (or class-conditional density) $p(x \mid c_k)$ describes how measurements are distributed within class c_k.
- The **evidence** $p(x) = \sum_j p(x \mid c_j)P(c_j)$ is the overall density of x; it is the same for every class and only normalizes.
- The **posterior** $P(c_k \mid x)$ is the probability of class c_k after observing x.

For continuous measurements, a likelihood is a density. Its value need not lie between zero and one, and it is not the probability of observing that exact real number. Posterior class probabilities do lie in $[0, 1]$ and sum to one over k. Confusing the two directions of conditioning, $p(x \mid c)$ versus $P(c \mid x)$, is the classic **base-rate error** that the code example below makes concrete.

Because the evidence is shared, a decision only needs the products $p(x \mid c_k)P(c_k)$. Figure 1 shows what that means geometrically for two one-dimensional Gaussian classes: the boundary sits where the two *prior-weighted* likelihoods cross. When the prior of class c_1 drops from 0.5 to 0.1, its weighted curve shrinks, and the boundary moves towards c_1's mean: the system now needs stronger evidence before declaring the rarer class.

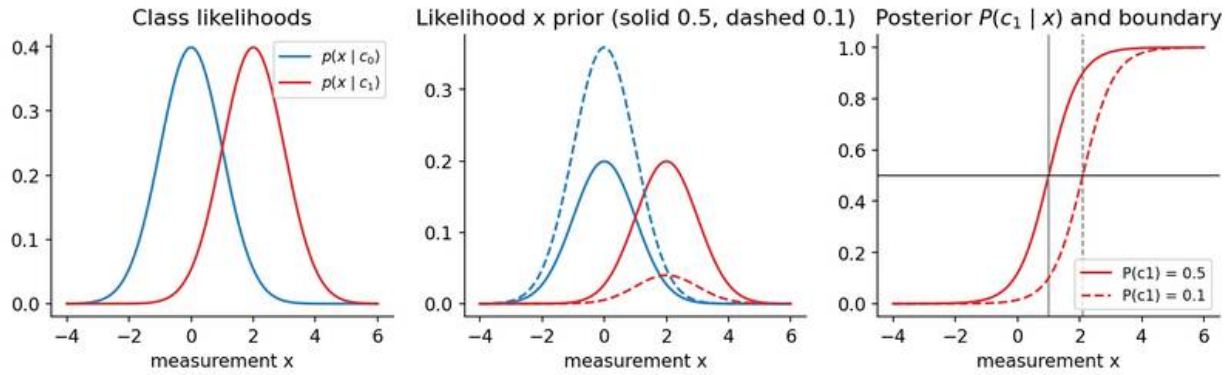


Figure 1 — Likelihoods (left) do not change when the prior changes; prior-weighted likelihoods (middle) and posteriors (right) do. With $P(c1) = 0.5$ the boundary is at $x = 1$; with $P(c1) = 0.1$ it moves to $1 + ln(9)/2 \approx 2.10$.

For the unit-variance case in Figure 1 the boundary has a closed form. With means $\mu_0 = 0$, $\mu_1 = 2$ and variance 1, the log ratio of the weighted likelihoods is $2x - 2 + \ln \frac{P(c_1)}{P(c_0)}$, which is zero at $x = 1 + \frac{1}{2} \ln \frac{P(c_0)}{P(c_1)}$. That is $x = 1$ for equal priors and $x \approx 2.10$ for $P(c_1) = 0.1$.

Code: rare events change the posterior

This NumPy example uses hypothetical inspection statistics: a detector flags 90% of defective parts (sensitivity) and 8% of good parts (false-positive rate), and 2% of parts are defective. It computes the posterior after a positive and after a negative detector output, then compares the minimum-error and minimum-risk actions (derived in the next sections).

```
import numpy as np

prior = 0.02
sensitivity = 0.90
false_positive_rate = 0.08
p_positive = (sensitivity * prior /
              (sensitivity * prior + false_positive_rate * (1 - prior)))
p_negative = ((1 - sensitivity) * prior /
              ((1 - sensitivity) * prior + (1 - false_positive_rate) * (1 - prior)))
p = np.array([p_negative, p_positive])
c_fp, c_fn = 1.0, 9.0
threshold = c_fp / (c_fp + c_fn)
risks = np.column_stack([c_fn * p, c_fp * (1 - p)])
decisions = risks.argmin(axis=1)
assert np.array_equal(decisions, (p > threshold).astype(int))
print("posteriors [negative observation, positive observation]:", p)
print("minimum-risk actions:", decisions, "threshold:", threshold)
print("minimum-error actions:", (p > 0.5).astype(int))
```

The output is a posterior of about 0.0022 after a negative flag and only 0.187 after a positive one. A positive observation does not imply a 90% defect probability: sensitivity conditions on the true class, while the posterior conditions on the observation, and with a 2% prior most positive flags come from the large pool of good parts. Under equal costs, neither observation leads to "defective" ($0.187 < 0.5$). With a missed defect costing nine times as much as an unnecessary inspection, the threshold drops to 0.1 and a positive flag does lead to inspection. The probability model did not change; only the decision rule did.

Minimum-error decisions

Plain version. If every mistake costs the same, always pick the class that is most probable for this observation. No other rule makes fewer mistakes on average.

Precise version. A decision rule $a(x)$ assigns each x to a class, which partitions feature space into **decision regions** $\mathcal{R}_1, \dots, \mathcal{R}_K$, where $\mathcal{R}_k = \{x : a(x) = c_k\}$. The probability of error is

$$P(\text{error}) = \int p(x)\,\bigl[1 - P(a(x) \mid x)\bigr]\,dx.$$

Here $P(a(x) \mid x)$ is the posterior of the class we chose. The integrand is non-negative and $p(x)$ does not depend on the rule, so the integral is minimized by minimizing $1 - P(a(x) \mid x)$ separately at every x, that is, by choosing

$$a^*(x) = \arg\max_k P(c_k \mid x) = \arg\max_k p(x \mid c_k) P(c_k).$$

This is the **maximum a posteriori (MAP) rule** or **Bayes classifier**. Its error,

$$P^*_{\text{err}} = \int p(x)\Bigl[1 - \max_k P(c_k \mid x)\Bigr]dx,$$

is the **Bayes error**: the lowest error any classifier can achieve with these features. It is zero only when the classes do not overlap in feature space. A classifier with 8% test error on a problem whose Bayes error is 7% is close to optimal; the same 8% on a problem with Bayes error 1% leaves a lot on the table. We rarely know the Bayes error of a real task, but thinking about it separates two different questions: "are my features informative enough?" (the Bayes error) and "does my classifier extract what the features contain?" (the gap above it).

For two classes the MAP rule becomes a **likelihood-ratio test**:

$$\Lambda(x) = \frac{p(x \mid c_1)}{p(x \mid c_0)} \gtrless \frac{P(c_0)}{P(c_1)}.$$

Choose c_1 when the ratio exceeds the threshold on the right, c_0 otherwise. The data enter only through $\Lambda(x)$; priors and, as we will see next, costs only move the threshold. This is why one score can serve many decision policies.

Loss matrices and minimum risk

Plain version. Some mistakes are worse than others. Assign a cost to each kind of mistake, compute the average cost of each possible action given the current beliefs, and take the cheapest action.

Precise version. Let the available actions be $\alpha_1, \dots, \alpha_A$ (usually "decide class i", possibly also "reject"). A **loss matrix** $\lambda_{ij} = L(\alpha_i, c_j)$ gives the cost of taking action α_i when the true class is c_j. The **conditional risk** of an action is its expected loss under the posterior:

$$R(\alpha_i \mid x) = \sum_{j=1}^{K} \lambda_{ij}\, P(c_j \mid x), \qquad a^*(x) = \arg\min_i R(\alpha_i \mid x).$$

The overall risk $R = \int R(a(x) \mid x)\, p(x)\, dx$ is minimized by the same pointwise argument as before, so this **Bayes decision rule** achieves the minimum possible expected loss, the **Bayes risk** [1, 2]. With the **zero-one loss** ($\lambda_{ij} = 0$ if $i = j$, 1 otherwise), $R(\alpha_i \mid x) = 1 - P(c_i \mid x)$ and the rule reduces to MAP.

For two classes, write λ_{10} for the cost of deciding c_1 when the truth is c_0 (a false positive) and λ_{01} for deciding c_0 when the truth is c_1 (a false negative). Choosing c_1 is cheaper when $R(\alpha_1 \mid x) < R(\alpha_0 \mid x)$, which rearranges to

$$\frac{p(x \mid c_1)}{p(x \mid c_0)} > \frac{\lambda_{10} - \lambda_{00}}{\lambda_{01} - \lambda_{11}} \cdot \frac{P(c_0)}{P(c_1)}.$$

This ratio form assumes positive class priors and $\lambda_{01} - \lambda_{11} > 0$, as in ordinary classification costs. If that difference is zero, compare the risks directly; if negative, dividing reverses the inequality. With free correct decisions, doubling the false-negative cost has the same effect as doubling the **prior odds** $P(c_1)/P(c_0)$, not the positive-class probability alone. The two priors must still sum to one. With correct decisions free ($\lambda_{00} = \lambda_{11} = 0$), $C_{\text{FP}} = \lambda_{10}$, $C_{\text{FN}} = \lambda_{01}$ and $p = P(c_1 \mid x)$, the rule becomes a threshold on the posterior:

$$C_{\text{FP}}(1 - p) < C_{\text{FN}}\, p \quad \Longleftrightarrow \quad p > \frac{C_{\text{FP}}}{C_{\text{FP}} + C_{\text{FN}}}.$$

When missing a defect costs nine times as much as an unnecessary inspection, the threshold is 0.1, not 0.5 (Figure 2, right).

Two remarks matter in practice. First, a loss matrix is a modelling decision, often made by people outside the machine-learning team (clinicians, safety engineers, product owners); write it down explicitly so that it can be reviewed. Second, the posterior threshold is only meaningful if the posterior is **calibrated**, meaning that among cases given probability 0.1, about 10% really are positive. Many modern classifiers are not, which is one theme of the Modern view below.

The reject option

Plain version. When the system is unsure, it can hand the case to a person. Refusing to decide has a fixed cost; deciding has an expected cost that is high when the posterior is spread out. Refuse exactly when deciding is expected to cost more.

Precise version. Add an action a_{reject} with constant loss C_R regardless of the true class, and keep the zero-one loss for the K classification actions. The best classification action has risk $1 - \max_k P(c_k \mid x)$, so the Bayes rule becomes **Chow's rule** [5]:

$$a^*(x) = \begin{cases} \arg\max_k P(c_k \mid x) & \text{if } \max_k P(c_k \mid x) \geq 1 - C_R, \\ \text{reject} & \text{otherwise.} \end{cases}$$

If rejection costs 0.2, the system accepts only when the top posterior is at least 0.8 (Figure 2, left). Two edge cases help intuition. If $C_R = 0$ (asking a person is free), the system rejects everything that is not certain. If $C_R \geq 1 - 1/K$, it never rejects, because the top posterior is always at least $1/K$.

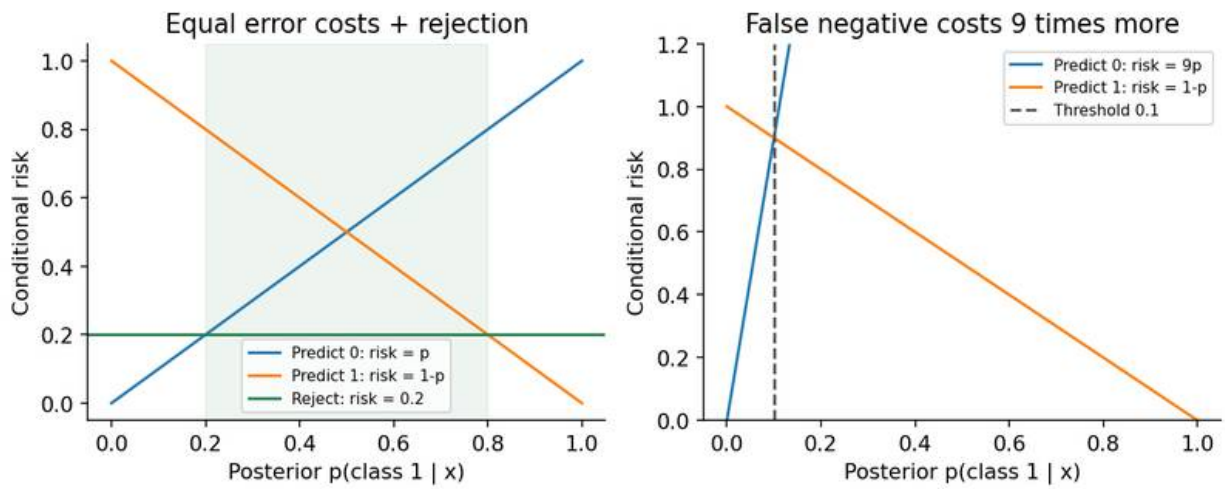


Figure 2 — Conditional risk as a function of the posterior. Left: equal costs plus a reject action costing 0.2; rejecting is cheapest when the posterior lies between 0.2 and 0.8. Right: a false negative costs nine times a false positive, so the decision threshold moves to 0.1.

Chow showed that sweeping the threshold traces an **error–reject trade-off curve**: each extra percent of rejected samples buys a reduction in the error on the remaining ones, with diminishing returns [5]. Figure 3 shows it for three overlapping Gaussian classes whose true posteriors are known. Without rejection about a quarter of all samples are misclassified; rejecting the least certain 40% cuts the error on all samples to about 8%.

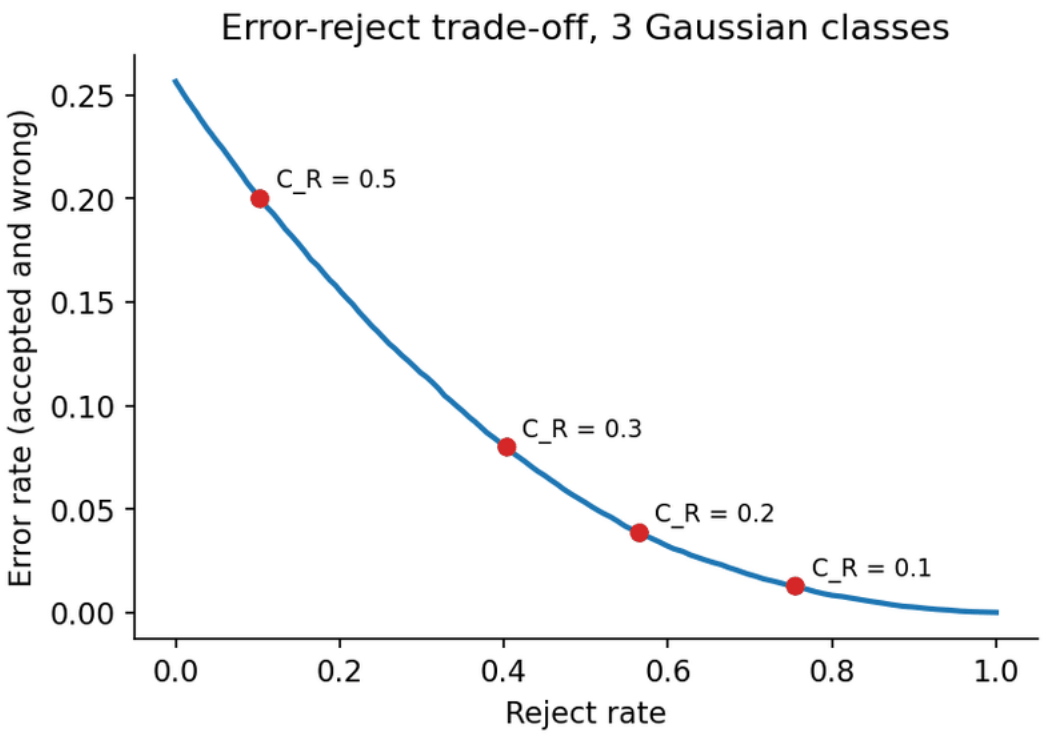


Figure 3 — Error–reject trade-off for three 2-D Gaussian classes with known posteriors (20,000 synthetic samples). Each point is Chow's rule for one reject cost C_R; the vertical axis counts samples that were accepted and misclassified.

Code: Chow's rule

```
import numpy as np

rng = np.random.default_rng(2)
means = np.array([[0, 0], [2, 0], [1, 1.7]])
y = rng.integers(0, 3, 20000)
X = means[y] + rng.normal(0, 1, (len(y), 2))

# exact posteriors: equal priors, identity covariance
logits = -0.5 * ((X[:, None, :] - means[None]) ** 2).sum(-1)
post = np.exp(logits - logits.max(1, keepdims=True))
post /= post.sum(1, keepdims=True)
pred, pmax = post.argmax(1), post.max(1)
```

```
for c_r in [0.5, 0.3, 0.2, 0.1]:
    accept = pmax >= 1 - c_r                    # Chow's rule
    reject_rate = 1 - accept.mean()
    err_accepted = np.mean(pred[accept] != y[accept])
    risk = np.mean(np.where(accept, pred != y, c_r))
    print(f"C_R={c_r}: reject {reject_rate:.3f}, "
          f"error on accepted {err_accepted:.3f}, average loss {risk:.3f}")
```

With this seed, $C_R = 0.3$ rejects about 40% of samples and leaves an error of about 13.5% *among the accepted ones*; $C_R = 0.1$ rejects about 76% and leaves about 4.4%. The average loss always falls as C_R falls because the reject action itself becomes cheaper. Subtracting the max logit before exponentiating (`logits.max`) is the standard trick that keeps the softmax numerically stable.

Rejection helps only when the uncertainty estimate is meaningful. Chow's rule handles **ambiguity**: observations that lie between known classes. It does not handle **novelty**: an object of a kind never seen in training can land far from every class and still receive a confident posterior, because posteriors are normalized over the known classes only. Detecting such inputs is the subject of open-set recognition and out-of-distribution detection (Modern view).

Discriminant functions and decision regions

Plain version. A classifier can be written as a set of scoring functions, one per class: compute every score and pick the largest. Many different scores give the same decisions.

Precise version. A classifier is defined by **discriminant functions** $g_1(x), \dots, g_K(x)$ and the rule "assign x to c_k if $g_k(x) > g_j(x)$ for all $j \neq k$" [1]. The Bayes minimum-error classifier is obtained with $g_k(x) = P(c_k \mid x)$, but any **monotonically increasing** function f applied to all discriminants leaves every decision unchanged. Three equivalent choices are therefore

$$g_k(x) = P(c_k \mid x), \qquad g_k(x) = p(x \mid c_k)P(c_k), \qquad g_k(x) = \ln p(x \mid c_k) + \ln P(c_k).$$

The log form is the one used in practice: it turns products of many small densities into sums and avoids numerical underflow. For minimum risk, use $g_k(x) = -R(\alpha_k \mid x)$.

The **decision boundary** between c_i and c_j is the set where $g_i(x) = g_j(x)$ and both are larger than all other scores. For two classes a single function suffices, a **dichotomizer**:

$$g(x) = g_1(x) - g_0(x) = \ln \frac{p(x \mid c_1)}{p(x \mid c_0)} + \ln \frac{P(c_1)}{P(c_0)}, \qquad \text{decide } c_1 \text{ if } g(x) > 0.$$

This is the log-likelihood ratio plus the log prior odds. It also equals the **logit** of the posterior, because $P(c_1 \mid x) = \sigma(g(x))$ with the logistic sigmoid $\sigma(z) = 1/(1+e^{-z})$. Remember this identity: in PR 03, logistic regression models $g(x)$ directly as a linear function instead of deriving it from densities.

The view through discriminants is useful because it separates *what* the classifier computes (the shape of the boundaries) from *how* it was obtained (densities, direct fitting, margins). Linear discriminants $g_k(x) = w_k^\top x + w_{k0}$ produce boundaries that are hyperplanes and regions that are

convex; quadratic discriminants produce conics in 2-D and hyperquadrics in general. The Gaussian case shows exactly when each arises.

The Gaussian case: LDA and QDA boundaries

Plain version. If each class looks like a "cloud" shaped like a multivariate Gaussian, the Bayes rule measures how far x is from each cloud's center, in units that respect the cloud's shape. When all clouds have the same shape, the boundaries are straight; when shapes differ, they bend.

Precise version. Suppose $p(x \mid c_k) = \mathcal{N}(x; \mu_k, \Sigma_k)$ with mean $\mu_k \in \mathbb{R}^d$ and covariance matrix $\Sigma_k \in \mathbb{R}^{d\times d}$ (symmetric, positive definite):

$$\mathcal{N}(x; \mu, \Sigma) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\!\Big(-\tfrac{1}{2}(x-\mu)^\top \Sigma^{-1}(x-\mu)\Big),$$

where $|\Sigma|$ is the determinant. The log discriminant is

$$g_k(x) = -\tfrac{1}{2}\,(x-\mu_k)^\top \Sigma_k^{-1}(x-\mu_k) - \tfrac{1}{2}\ln|\Sigma_k| + \ln P(c_k),$$

after dropping the constant $-\frac{d}{2}\ln 2\pi$ shared by all classes. The quadratic form $\Delta_k^2(x) = (x-\mu_k)^\top\Sigma_k^{-1}(x-\mu_k)$ is the squared **Mahalanobis distance** from x to μ_k: Euclidean distance after "whitening" the class so its cloud becomes a unit sphere. Three cases, shown in Figure 4, cover what happens [1, 4].

Case 1: $\Sigma_k = \sigma^2 I$ for all k. Features are uncorrelated with equal variance σ^2. Expanding $\|x-\mu_k\|^2$ and dropping the class-independent $x^\top x$ gives a linear discriminant

$$g_k(x) = w_k^\top x + w_{k0}, \qquad w_k = \frac{\mu_k}{\sigma^2}, \qquad w_{k0} = -\frac{\mu_k^\top \mu_k}{2\sigma^2} + \ln P(c_k).$$

The boundary between classes i and j is the hyperplane $w^\top(x - x_0) = 0$ with $w = \mu_i - \mu_j$ and

$$x_0 = \tfrac{1}{2}(\mu_i + \mu_j) - \frac{\sigma^2}{\|\mu_i - \mu_j\|^2}\ln\frac{P(c_i)}{P(c_j)}\,(\mu_i - \mu_j).$$

It is perpendicular to the line joining the means. With equal priors it passes through their midpoint, and the classifier is exactly the **minimum-distance (nearest-mean) classifier** of IP 13. Unequal priors slide the plane away from the more probable mean.

Case 2: $\Sigma_k = \Sigma$ shared by all classes. The quadratic term $x^\top\Sigma^{-1}x$ is again common and cancels, leaving

$$w_k = \Sigma^{-1}\mu_k, \qquad w_{k0} = -\tfrac{1}{2}\mu_k^\top\Sigma^{-1}\mu_k + \ln P(c_k).$$

The boundaries are still hyperplanes, $w^\top(x - x_0) = 0$ with $w = \Sigma^{-1}(\mu_i - \mu_j)$, but no longer perpendicular to the mean difference: they tilt to follow the shared correlation (Figure 4, middle). This is **linear discriminant analysis (LDA)** in its probabilistic form. With equal priors it is a minimum-Mahalanobis-distance classifier. The direction $\Sigma^{-1}(\mu_1 - \mu_0)$ is the same as the one Fisher derived from a projection criterion in 1936, which PR 03 re-derives.

Case 3: arbitrary Σ_k. Nothing cancels:

$$g_k(x) = x^\top W_k x + w_k^\top x + w_{k0}, \quad W_k = -\tfrac{1}{2}\Sigma_k^{-1}, \quad w_k = \Sigma_k^{-1}\mu_k, \quad w_{k0} = -\tfrac{1}{2}\mu_k^\top \Sigma_k^{-1}\mu_k - \tfrac{1}{2}\ln|\Sigma_k| + \ln P(c_k).$$

The boundaries are **hyperquadrics**: lines, parabolas, ellipses or hyperbolas in 2-D. This is **quadratic discriminant analysis (QDA)**. In Figure 4 (right) the tight blue class is enclosed by an ellipse, because a point far from the blue mean in any direction is better explained by the broad red class. A region can even be disconnected, which never happens for linear discriminants.

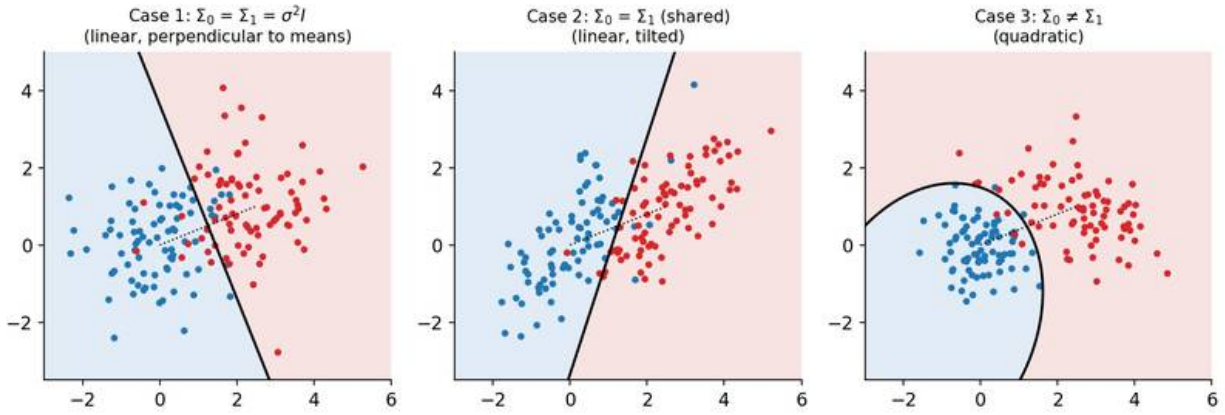


Figure 4 — Bayes boundaries (black) for two Gaussian classes with equal priors. Case 1: spherical shared covariance gives the perpendicular bisector. Case 2: a shared correlated covariance tilts the line (LDA). Case 3: different covariances give a quadratic boundary (QDA). The dotted segment joins the means.

For two classes with a shared covariance and equal priors, the Bayes error has a closed form,

$$P^*_{\text{err}} = \Phi\left(-\frac{\Delta}{2}\right), \qquad \Delta = \sqrt{(\mu_1 - \mu_0)^\top \Sigma^{-1}(\mu_1 - \mu_0)},$$

where Φ is the standard normal cumulative distribution function and Δ is the Mahalanobis distance between the means. Two classes three standard deviations apart ($\Delta = 3$) still give $\Phi(-1.5) \approx 6.7\%$ unavoidable error. This formula is a good sanity check for any classifier you train on Gaussian toy data.

Which case should you use? It is a bias–variance choice. A full covariance per class has $d(d+1)/2$ free entries per class; with $d = 100$ that is 5,050 numbers per class, which needs far more samples than many datasets have. Sharing one covariance (LDA), restricting to diagonals (which gives Gaussian naive Bayes) or shrinking towards a simple target trades flexibility for stability. Estimating these parameters is the subject of the next chapter, PR 02: density estimation.

Code: Gaussian discriminants from scratch

This example fits Gaussian class models, evaluates the log discriminants with a Cholesky factorization (never an explicit inverse), and checks that the decisions match scikit-learn's `LinearDiscriminantAnalysis` and `QuadraticDiscriminantAnalysis` on the Case 3 data of Figure 4.

```
import numpy as np
from sklearn.discriminant_analysis import (LinearDiscriminantAnalysis,
                                           QuadraticDiscriminantAnalysis)

rng = np.random.default_rng(0)
means = [np.array([0.0, 0.0]), np.array([2.5, 1.0])]
covs = [np.array([[0.4, 0.0], [0.0, 0.4]]), np.array([[2.0, -0.6], [-0.6, 1.0]])]
X = np.vstack([rng.multivariate_normal(m, S, 300) for m, S in zip(means, covs)])
y = np.repeat([0, 1], 300)

def fit_gaussians(X, y, shared):
    params = []
    pooled = sum(np.cov(X[y == k].T, bias=True) * (y == k).sum()
                 for k in (0, 1)) / len(y)
    for k in (0, 1):
        Xk = X[y == k]
        S = pooled if shared else np.cov(Xk.T, bias=True)
        params.append((Xk.mean(0), S, len(Xk) / len(y)))
    return params

def discriminants(X, params):
    g = []
    for mu, S, prior in params:
        L = np.linalg.cholesky(S)               # S = L L^T
        z = np.linalg.solve(L, (X - mu).T)      # whitened residuals
        maha = (z ** 2).sum(0)                  # squared Mahalanobis distance
        logdet = 2 * np.log(np.diag(L)).sum()
        g.append(-0.5 * maha - 0.5 * logdet + np.log(prior))
    return np.stack(g, axis=1)

for shared, ref in [(True, LinearDiscriminantAnalysis(store_covariance=True)),
                    (False, QuadraticDiscriminantAnalysis())]:
    mine = discriminants(X, fit_gaussians(X, y, shared)).argmax(1)
    theirs = ref.fit(X, y).predict(X)
    name = "LDA (shared)" if shared else "QDA (separate)"
    print(f"{name}: training error {np.mean(mine != y):.3f}, "
          f"agrees with scikit-learn on {np.mean(mine == theirs):.1%} of points")
```

Both hand-written classifiers agree with scikit-learn on every point. On these data, whose true covariances differ, QDA's training error (4.3%) is below LDA's (5.5%). That is training error, though: with few samples in high dimensions the ordering often reverses on held-out data.

ROC analysis and the Neyman–Pearson view

Plain version. A two-class score can be cut at many thresholds. Each threshold trades catching more positives against raising more false alarms. The ROC curve draws all those trade-offs at once, so you can pick the threshold later, when costs or priors are known.

Precise version. Let $s(x)$ be any score where larger values favor c_1 (the log-likelihood ratio, a posterior, a network logit), and decide c_1 when $s(x) > t$. For each threshold t define

$$\mathrm{TPR}(t) = P(s(x) > t \mid c_1), \qquad \mathrm{FPR}(t) = P(s(x) > t \mid c_0).$$

The true-positive rate (sensitivity, recall) is the fraction of positives caught; the false-positive rate ($1-$ specificity) is the fraction of negatives flagged. The **receiver operating characteristic (ROC)** curve plots TPR against FPR as t sweeps from $+\infty$ to $-\infty$ [7]. A useless score lies on the diagonal; a perfect one passes through the top-left corner. Because both rates condition on the true class, the ROC curve does not change when the class priors change.

The Neyman–Pearson criterion. Sometimes costs are hard to state but a constraint is easy: "at most 1% false alarms". The Neyman–Pearson lemma [6] says that among all decision rules with $\mathrm{FPR} \le \alpha$, the one with the highest TPR is a likelihood-ratio test $\Lambda(x) > \eta$, with η chosen so the false-positive rate equals α (with randomization at the boundary if needed). So the ROC curve of the likelihood ratio is the best achievable curve; any other score's ROC lies on or below it. Both the Bayes rule and the Neyman–Pearson rule are likelihood-ratio tests; they differ only in how the threshold is chosen.

Choosing an operating point from costs. The expected cost per sample at threshold t is

$$\mathbb{E}[\text{cost}] = C_{\mathrm{FP}}\, P(c_0)\, \mathrm{FPR}(t) + C_{\mathrm{FN}}\, P(c_1)\, (1 - \mathrm{TPR}(t)).$$

Points of equal cost lie on straight **iso-cost lines** in ROC space with slope $C_{\mathrm{FP}}P(c_0)/(C_{\mathrm{FN}}P(c_1))$; all these lines have the same fixed slope. Slide the line toward higher TPR (a larger vertical intercept) until it last touches an achievable point; that point minimizes cost [7]. For the likelihood-ratio score this tangent point is exactly the Bayes threshold derived earlier.

Area under the curve. The AUC summarizes the curve in one number. It equals the probability that a randomly chosen positive receives a higher score than a randomly chosen negative (ties counted half) [7]. For two unit-variance Gaussian score distributions separated by d' standard deviations, $\mathrm{AUC} = \Phi(d'/\sqrt{2})$: Figure 5 shows $d' = 2$ gives 0.921. AUC ignores costs, priors and calibration, so two classifiers with the same AUC can have very different costs at the threshold you actually deploy.

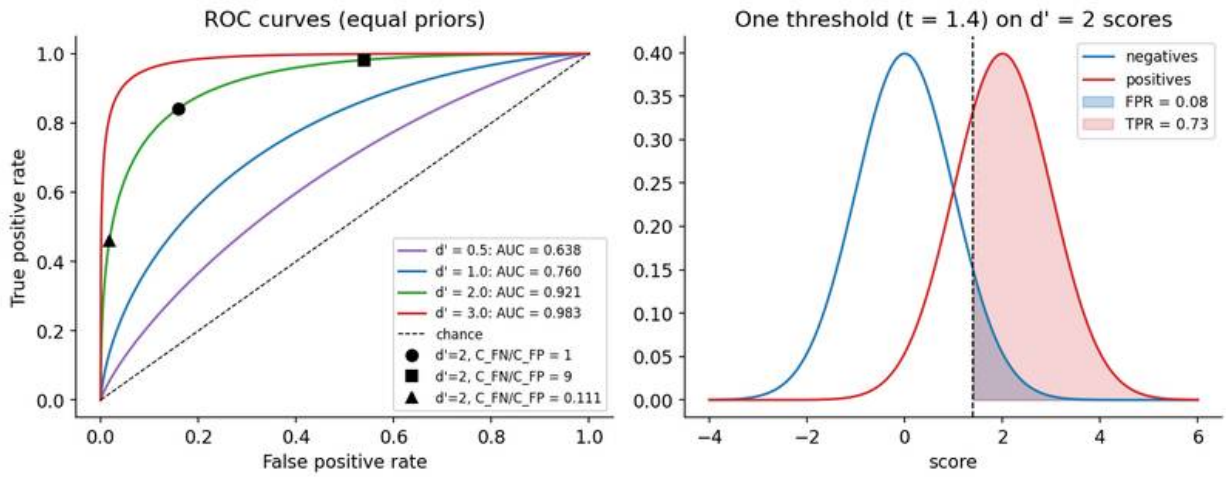


Figure 5 — Left: ROC curves of equal-variance Gaussian scores; markers show the Bayes operating points

on the $d' = 2$ curve for three cost ratios (equal priors). Right: one threshold on the $d' = 2$ scores and the FPR and TPR it produces.

Precision–recall curves. When positives are rare (fraud, defects, small objects in detection), a small FPR can still mean many false alarms in absolute numbers, because negatives are so numerous. Precision $= \mathrm{TP}/(\mathrm{TP} + \mathrm{FP})$ depends on the prior and exposes this. Davis and Goadrich [8] proved that one curve dominates another in ROC space if and only if it dominates in precision–recall space, and that linear interpolation between PR points is not valid. Saito and Rehmsmeier [9] showed with simulations and published studies that for imbalanced data the PR plot is more informative than ROC. Object-detection benchmarks report average precision for this reason (see DLCV 03).

Code: ROC, AUC and the cost-optimal threshold

```
import numpy as np
from sklearn.metrics import roc_auc_score, roc_curve

rng = np.random.default_rng(1)
n_neg, n_pos = 9000, 1000                        # prevalence 10%
scores = np.r_[rng.normal(0, 1, n_neg), rng.normal(2, 1, n_pos)]
labels = np.r_[np.zeros(n_neg), np.ones(n_pos)]

fpr, tpr, thr = roc_curve(labels, scores)
auc = roc_auc_score(labels, scores)
# Mann-Whitney view of AUC: P(score_pos > score_neg)
pos, neg = scores[labels == 1], scores[labels == 0]
auc_pairs = (pos[:, None] > neg[None, :]).mean()
print(f"AUC = {auc:.4f}, pairwise estimate = {auc_pairs:.4f}")

# choose the operating point with minimum expected cost per sample
c_fp, c_fn, prior = 1.0, 9.0, n_pos / (n_pos + n_neg)
cost = c_fp * fpr * (1 - prior) + c_fn * (1 - tpr) * prior
i = np.argmin(cost)
print(f"best threshold {thr[i]:.2f}: FPR {fpr[i]:.3f}, TPR {tpr[i]:.3f}, "
      f"expected cost {cost[i]:.3f}")
# Bayes threshold from the true model: predict 1 when the likelihood ratio
# exceeds c_fp (1 - prior) / (c_fn prior); for unit Gaussians, log LR = 2s - 2
t_bayes = (np.log(c_fp * (1 - prior) / (c_fn * prior)) + 2) / 2
print(f"Bayes threshold from the true densities: {t_bayes:.2f}")
```

The empirical AUC (0.9166) matches the pairwise-ranking estimate exactly, and is close to the theoretical $\Phi(2/\sqrt{2}) \approx 0.921$. With a 10% prior and a 9:1 cost ratio, prior odds and cost ratio cancel, so the Bayes threshold is $t = 1$; the empirical search finds 0.99. In a real project the threshold search must run on validation data, never on the test set.

Where the assumptions can fail

The Bayes rule is optimal *relative to* the assumed distributions and the specified loss. Each ingredient can be wrong in deployment.

Prior shift. Class frequencies may differ between the training set and deployment, for example between two factories or between a balanced research dataset and a real clinic. If the class-conditional distributions $p(x \mid c)$ stay fixed (**label shift**), the posterior can be corrected without retraining: multiply each posterior by the ratio of new to old priors and renormalize,

$$P_{\text{new}}(c_k \mid x) \propto P_{\text{old}}(c_k \mid x)\, \frac{P_{\text{new}}(c_k)}{P_{\text{old}}(c_k)}.$$

When the new priors are unknown, Saerens, Latinne and Decaestecker [10] estimate them with an EM procedure on the unlabeled deployment data, and Lipton, Wang and Smola's black-box shift estimation [11] estimates them from a classifier's confusion matrix. If a new camera changes the likelihoods themselves (**covariate** or **concept shift**), a prior correction is not enough.

Miscalibrated posteriors. Every threshold in this chapter assumes that $P(c \mid x)$ means what it says. Scores from SVMs, boosted trees or deep networks are often systematically over- or under-confident; recalibration on held-out data (Platt scaling, isotonic regression, temperature scaling) is a standard fix [12, 14].

Label noise and proxies. A label may record an imperfect inspection rather than physical truth. The model then learns the inspector's posterior, not the world's.

Single-number summaries. Two classifiers with identical accuracy can have different false-negative rates and very different expected loss. Report the confusion matrix, the operating point and the costs.

Modern view

Bayes decision theory is more than fifty years old, and its vocabulary (posterior, risk, reject, operating point) is now the vocabulary of trustworthy deep learning. Research since 2017 has mostly been about making the *inputs* to the Bayes rule trustworthy when the classifier is a large network.

Calibration. Guo et al. [12] found that modern deep networks were much more overconfident than the shallower networks of a decade earlier. They related this to depth, width, weight decay and batch normalization, and showed that **temperature scaling** (dividing all logits by one scalar fitted on validation data, a one-parameter version of Platt scaling) is surprisingly effective. Minderer et al. [13] revisited the question for 2021-era image classifiers and found that the most recent models, notably those not using convolutions, are among the best calibrated, and that the earlier trend of calibration decaying with model size or under distribution shift is less pronounced. The survey by Silva Filho et al. [14] is the best entry point: it covers proper scoring rules and calibration metrics, reliability diagrams, and a comprehensive account of post-hoc calibration methods for binary and multiclass problems. Gawlikowski et al. [23] review the wider field of uncertainty

in neural networks, separating reducible model uncertainty from irreducible data uncertainty and covering Bayesian networks, ensembles and test-time augmentation.

Rejection and selective classification. Chow's reject rule [5] has been rediscovered as **selective classification**. Geifman and El-Yaniv [16] showed how to wrap a trained network with a rejection threshold chosen so that the risk on accepted samples stays below a user-specified level with high probability, trading coverage for guaranteed accuracy. The survey by Hendrickx et al. [15] formalizes two kinds of rejection, **ambiguity rejection** (Chow's case) and **novelty rejection** (inputs unlike the training data), reviews how to evaluate a model's predictive and rejective quality together, and categorizes the architectures used to learn models with a reject option.

Distribution-free guarantees. Conformal prediction replaces a single label with a *set* of labels that is guaranteed to contain the true class with a chosen probability, such as 90%, under the sole assumption that calibration and test data are exchangeable. Angelopoulos and Bates [17] give a hands-on introduction with code. Large prediction sets play the role of rejection: the model is telling you that it cannot narrow things down.

Open-set recognition and out-of-distribution detection. A posterior normalized over K known classes cannot say "none of the above". Geng, Huang and Chen [18] survey open-set recognition: definitions, model families, datasets and evaluation, and its relationship to classification with reject option and to zero- and few-shot learning. Yang et al. [19] unify anomaly detection, novelty detection, open-set recognition, out-of-distribution detection and outlier detection as subtasks of one **generalized OOD detection** framework. Two results connect directly to this chapter. Hendrycks and Gimpel [20] showed that the **maximum softmax probability**, Chow's statistic, is a strong baseline for detecting both misclassified and out-of-distribution inputs. Vaze et al. [21] found that a classifier's ability to say "none of the above" is highly correlated with its closed-set accuracy, so improving the ordinary classifier improves open-set performance. Lee et al. [22] fit **class-conditional Gaussians with a shared covariance** (exactly Case 2 / LDA above) to the features of a pretrained network and use the Mahalanobis distance as a confidence score. A 1970s-style generative classifier, placed on top of learned features, remains a competitive OOD detector.

Open problems. Costs are rarely known precisely, and they change over time; calibration measured on one distribution may not transfer to another; and guarantees such as conformal coverage are marginal (on average over inputs) rather than conditional on a particular input or subgroup. Deciding *what* the loss matrix should be, and who decides, remains a human question that no estimator answers.

Key takeaways

- A recognition system maps observations to features, features to beliefs, and beliefs to actions; the sampling unit and data split are part of the problem definition.
- Prior, likelihood and posterior answer different questions. Rare classes need strong evidence: a 90%-sensitive test at 2% prevalence gives a posterior below 20%.
- Maximum posterior minimizes the error rate; minimum conditional risk handles any loss matrix. For two classes both are likelihood-ratio tests whose threshold depends on priors times costs.

- Chow's rule rejects when the top posterior falls below $1 - C_R$; it handles ambiguity, not novelty.
- Discriminant functions are defined up to a monotone transform. Gaussian classes give linear boundaries when covariances are shared (LDA) and quadratic ones when they differ (QDA).
- ROC curves separate the score from the threshold; the Neyman–Pearson lemma makes the likelihood ratio the optimal score; PR curves are more informative when positives are rare.
- All of this relies on calibrated probabilities and stable priors, which modern work on calibration, selective classification, conformal prediction and OOD detection tries to secure.

Exercises

1. **Asymmetric costs.** Derive the binary decision threshold on the posterior $p = P(c_1 \mid x)$ when a false positive costs four and a false negative costs one. Which class does the rule now favor, and why?

   Hint

   Choose c_1 when $4(1-p) < p$, so the threshold is $p > 0.8$. False alarms are now the expensive mistake, so the rule needs strong evidence before saying "positive".

2. **Reject or not.** For posterior $(0.55, 0.35, 0.10)$ and reject cost $C_R = 0.3$, choose the minimum-risk action under zero-one classification loss. For which values of C_R would the answer change?

   Hint

   The best classification risk is $1 - 0.55 = 0.45 > 0.3$, so reject. The system classifies (as c_1) when $C_R \geq 0.45$.

3. **A moving boundary.** Two 1-D classes have $p(x \mid c_0) = \mathcal{N}(0, 1)$ and $p(x \mid c_1) = \mathcal{N}(3, 1)$. Find the minimum-error boundary for equal priors and for $P(c_1) = 0.2$. Then find the minimum-risk boundary for equal priors when a false negative costs four times a false positive.

   Hint

   The log-likelihood ratio is $3x - 4.5$. Equal priors: $x = 1.5$. With $P(c_1) = 0.2$: $3x - 4.5 = \ln 4$, so $x \approx 1.96$. With $C_{\text{FN}} = 4C_{\text{FP}}$: $3x - 4.5 = \ln(1/4)$, so $x \approx 1.04$. The prior and the cost act through the same threshold, in opposite directions here.

4. **Mahalanobis versus Euclidean.** Two classes share $\Sigma = \begin{bmatrix} 4 & 0 \\ 0 & 1 \end{bmatrix}$ with means $\mu_0 = (0, 0)$ and $\mu_1 = (2, 1)$ and equal priors. Classify $x = (2, 0)$ with the Euclidean nearest-mean rule and with the Bayes (LDA) rule. Explain the disagreement.

   Hint

   Euclidean squared distances: 4 to μ_0 and 1 to μ_1, so nearest-mean says c_1. Mahalanobis: $(2^2/4) + 0 = 1$ to μ_0 and $0 + 1 = 1$ to μ_1, a tie: x lies exactly on the LDA boundary. The first feature varies a lot, so a 2-unit offset along it is weak evidence; the Euclidean rule over-weights it.

5. **ROC from scratch.** Using NumPy only, compute the ROC curve and the AUC of the scores in the ROC code example by sorting the scores, and verify against `roc_auc_score`. Then change the prevalence to 1% and explain which of ROC, AUC and precision at a fixed threshold change.

   Hint

   Sort scores in decreasing order; cumulative sums of labels give TP counts and of 1−labels give FP counts; integrate TPR over FPR with the trapezoid rule. ROC and AUC depend only on the two class-conditional score distributions, so they stay (up to sampling noise) the same; precision at a fixed threshold falls sharply because false positives now come from a much larger pool of negatives.

6. **The sampling unit.** A test set contains adjacent frames from the same videos used for training. Explain what its accuracy estimates, and propose a split that estimates performance on new videos, new cameras or new subjects.

   Hint

   It estimates accuracy on near-duplicates of training frames (an interpolation task). Split by recording (group-wise splitting), and, if deployment will meet new cameras or people, hold out entire cameras or subjects as well.

References

1. R. O. Duda, P. E. Hart and D. G. Stork, *Pattern Classification*, 2nd ed., Wiley, 2001 (Ch. 2, Bayesian decision theory). publisher page
2. C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006 (Sections 1.2 and 1.5). official page and free PDF
3. S. Theodoridis and K. Koutroumbas, *Pattern Recognition*, 4th ed., Academic Press, 2008 (Ch. 1–2). publisher page
4. T. Hastie, R. Tibshirani and J. Friedman, *The Elements of Statistical Learning*, 2nd ed., Springer, 2009 (Section 4.3, linear discriminant analysis). official page
5. C. K. Chow, "On optimum recognition error and reject tradeoff," *IEEE Transactions on Information Theory*, vol. 16, no. 1, pp. 41–46, 1970. doi
6. J. Neyman and E. S. Pearson, "On the problem of the most efficient tests of statistical hypotheses," *Philosophical Transactions of the Royal Society of London, Series A*, vol. 231, pp. 289–337, 1933. doi
7. T. Fawcett, "An introduction to ROC analysis," *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, 2006. doi
8. J. Davis and M. Goadrich, "The relationship between precision-recall and ROC curves," in *Proc. International Conference on Machine Learning (ICML)*, 2006. doi
9. T. Saito and M. Rehmsmeier, "The precision-recall plot is more informative than the ROC plot when evaluating binary classifiers on imbalanced datasets," *PLOS ONE*, vol. 10, no. 3, e0118432, 2015. doi

10. M. Saerens, P. Latinne and C. Decaestecker, “Adjusting the outputs of a classifier to new a priori probabilities: a simple procedure,” *Neural Computation*, vol. 14, no. 1, pp. 21–41, 2002. doi
11. Z. C. Lipton, Y.-X. Wang and A. Smola, “Detecting and correcting for label shift with black box predictors,” in *Proc. ICML*, PMLR 80, 2018. paper
12. C. Guo, G. Pleiss, Y. Sun and K. Q. Weinberger, “On calibration of modern neural networks,” in *Proc. ICML*, PMLR 70, 2017. paper
13. M. Minderer, J. Djolonga, R. Romijnders, F. Hubis, X. Zhai, N. Houlsby, D. Tran and M. Lucic, “Revisiting the calibration of modern neural networks,” in *Advances in Neural Information Processing Systems* (*NeurIPS*), 2021. paper
14. T. Silva Filho, H. Song, M. Perello-Nieto, R. Santos-Rodriguez, M. Kull and P. Flach, “Classifier calibration: a survey on how to assess and improve predicted class probabilities,” *Machine Learning*, vol. 112, pp. 3211–3260, 2023. doi
15. K. Hendrickx, L. Perini, D. Van der Plas, W. Meert and J. Davis, “Machine learning with a reject option: a survey,” *Machine Learning*, vol. 113, pp. 3073–3110, 2024. doi
16. Y. Geifman and R. El-Yaniv, “Selective classification for deep neural networks,” in *Advances in Neural Information Processing Systems* (*NeurIPS*), 2017. paper
17. A. N. Angelopoulos and S. Bates, “Conformal prediction: a gentle introduction,” *Foundations and Trends in Machine Learning*, vol. 16, no. 4, pp. 494–591, 2023. doi
18. C. Geng, S.-J. Huang and S. Chen, “Recent advances in open set recognition: a survey,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 10, pp. 3614–3631, 2021. doi
19. J. Yang, K. Zhou, Y. Li and Z. Liu, “Generalized out-of-distribution detection: a survey,” *International Journal of Computer Vision*, vol. 132, pp. 5635–5662, 2024. doi
20. D. Hendrycks and K. Gimpel, “A baseline for detecting misclassified and out-of-distribution examples in neural networks,” in *Proc. International Conference on Learning Representations* (*ICLR*), 2017. arXiv
21. S. Vaze, K. Han, A. Vedaldi and A. Zisserman, “Open-set recognition: a good closed-set classifier is all you need,” in *Proc. ICLR*, 2022. arXiv
22. K. Lee, K. Lee, H. Lee and J. Shin, “A simple unified framework for detecting out-of-distribution samples and adversarial attacks,” in *Advances in Neural Information Processing Systems* (*NeurIPS*), 2018. paper
23. J. Gawlikowski et al., “A survey of uncertainty in deep neural networks,” *Artificial Intelligence Review*, vol. 56, suppl. 1, pp. 1513–1589, 2023. doi

PR 02 • Density Estimation and Bayesian Learning

Estimate the densities a Bayes classifier needs: maximum likelihood, MAP and full Bayesian estimation; histograms, Parzen windows and kernel bandwidth selection; k-NN density and the k-NN classifier; and the curse of dimensionality.

Online edition: https://blog.shyandram.dev/en/learn/pattern-recognition/density-estimation/

What you'll learn

- Why a Bayes classifier needs density estimates, and the difference between parametric and non-parametric estimation.
- How to derive maximum-likelihood (ML) estimates for a Gaussian, why the covariance becomes singular with few samples, and how shrinkage fixes it.
- How MAP and full Bayesian estimation use a prior over parameters, worked out for a Gaussian mean and a Bernoulli rate.
- How histograms, Parzen windows (kernel density estimation) and k-nearest-neighbor estimates follow from one formula, and how to choose the bandwidth or k.
- How the k-NN classifier falls out of k-NN density estimation, what the Cover–Hart bound says, and why all local methods suffer from the curse of dimensionality.

The big picture

PR 01 showed that the best possible classifier needs the class priors $P(c_k)$ and the class-conditional densities $p(x \mid c_k)$. In practice nobody hands us these distributions; we have a finite set of labelled examples. This chapter is about turning examples into densities. Once we have $\hat{p}(x \mid c_k)$ and $\hat{P}(c_k)$, we can plug them into the Bayes rule. This is the **generative** route to classification. The next chapter takes the **discriminative** route and fits the decision boundary directly.

> **Explain like I'm five**
>
> Put a sticker on a park map wherever you have seen a bird. Now guess where the next bird will be. You could draw **one big oval** around most stickers (a parametric model: you only decide the oval's center and shape). You could lay a **grid** over the map and count stickers per square (a histogram). Or you could put a **little sand hill** on every sticker and look at where the sand piles up highest (a kernel estimate). Each choice is a different way to guess, and each can go wrong: the oval is too simple if birds like two different ponds, and the sand hills are too bumpy if you only have five stickers.

There are two families of estimators [1, 2, 3]:

- **Parametric** methods assume a functional form, such as a Gaussian, and estimate a fixed number of parameters. They need little data when the form is right and give systematically wrong answers when it is wrong.
- **Non-parametric** methods let the data decide the shape. Their complexity grows with the number of samples. They are flexible but need much more data, especially in high dimensions.

Both families face the same central question: how much structure can the available data support? A flexible estimator can describe the training data beautifully and still predict new data poorly. Every method below has a "smoothness knob" (the number of parameters, the prior, the bin width, the bandwidth, k), and choosing it well matters more than choosing the method.

Maximum-likelihood estimation

Plain version. Choose the parameter values under which the observed data would have been most probable.

Precise version. Let $D = \{x_1, \dots, x_N\}$ be samples drawn independently from a density $p(x \mid \theta)$ with unknown parameter vector θ. The **likelihood** of θ is $p(D \mid \theta) = \prod_n p(x_n \mid \theta)$, and the ML estimate maximizes its logarithm:

$$\hat{\theta}_{\mathrm{ML}} = \arg\max_{\theta} \ell(\theta), \qquad \ell(\theta) = \sum_{n=1}^{N} \log p(x_n \mid \theta).$$

The independence assumption is about *sampling* (different observations do not influence each other), not about independence between the coordinates of one feature vector. For a classifier we fit one model per class, using only that class's samples, and estimate the prior as the class frequency $\hat{P}(c_k) = N_k/N$.

The Gaussian case

For a d-dimensional Gaussian $\mathcal{N}(\mu, \Sigma)$ the log-likelihood is

$$\ell(\mu, \Sigma) = -\frac{N}{2}\log|\Sigma| - \frac{1}{2}\sum_n (x_n - \mu)^\top \Sigma^{-1}(x_n - \mu) + \text{const.}$$

Setting the gradient with respect to μ to zero gives $\Sigma^{-1}\sum_n (x_n - \mu) = 0$. Differentiating with respect to Σ^{-1} (using $\partial \log|A|/\partial A = A^{-\top}$ and $\partial\, a^\top A a/\partial A = a a^\top$) gives the estimates

$$\hat{\mu} = \frac{1}{N}\sum_n x_n, \qquad \hat{\Sigma} = \frac{1}{N}\sum_n (x_n - \hat{\mu})(x_n - \hat{\mu})^\top.$$

The sample mean is unbiased. The ML covariance is slightly **biased**: $\mathbb{E}[\hat{\Sigma}] = \frac{N-1}{N}\Sigma$, because $\hat{\mu}$ was fitted to the same data and sits closer to them than the true mean does. The familiar sample covariance divides by $N - 1$ to remove this bias. For large N the difference is negligible.

When ML breaks: too many parameters

A full covariance has $d(d+1)/2$ distinct entries. With $d = 64$ features that is 2,080 numbers per class, plus 64 for the mean. Worse, $\hat{\Sigma}$ is a sum of N rank-one matrices centered at $\hat{\mu}$, so its rank is at most $N - 1$. **When a class has no more samples than dimensions, its ML covariance is singular**: it has zero variance in some directions, $\log|\hat{\Sigma}| = -\infty$, and the Gaussian discriminant of PR 01 is undefined. Even when N is a bit larger than d, the smallest eigenvalues are systematically underestimated, and the inverse $\hat{\Sigma}^{-1}$ hugely amplifies noise along those directions.

Regularization **shrinks** the covariance towards a simpler target:

$$\tilde{\Sigma} = (1 - \alpha)\hat{\Sigma} + \alpha\tau I, \qquad \tau = \operatorname{tr}(\hat{\Sigma})/d, \qquad 0 \le \alpha \le 1.$$

Here τI is a spherical matrix with the same average variance, and α is the shrinkage intensity. Shrinkage pulls large eigenvalues down and small ones up, trading a little bias for a much better-conditioned estimate. Ledoit and Wolf [14] derived a closed-form estimate of the α that minimizes the expected squared error, so it needs no cross-validation. Other common restrictions are a **diagonal** covariance (ignore correlations; this is Gaussian naive Bayes, below) and a **shared** covariance across classes (pool the evidence; this is LDA from PR 01). These are different modelling assumptions, not interchangeable numerical tricks.

Code: when the ML covariance fails

```
import numpy as np
from scipy.stats import multivariate_normal
from sklearn.covariance import EmpiricalCovariance, LedoitWolf

rng = np.random.default_rng(0)
d = 40
A = rng.normal(size=(d, d))
true_cov = A @ A.T / d + 0.5 * np.eye(d)          # a well-conditioned truth
test = rng.multivariate_normal(np.zeros(d), true_cov, 2000)

for n in [30, 60, 200, 1000]:
    train = rng.multivariate_normal(np.zeros(d), true_cov, n)
    mu = train.mean(0)
    S_ml = EmpiricalCovariance().fit(train).covariance_   # divisor n
    S_lw = LedoitWolf().fit(train)
    rank = np.linalg.matrix_rank(S_ml)
    ll_ml = (multivariate_normal(mu, S_ml, allow_singular=True).logpdf(test).mean()
             if rank == d else -np.inf)
    ll_lw = multivariate_normal(mu, S_lw.covariance_).logpdf(test).mean()
    print(f"n={n:4d}: rank(S_ML)={rank:2d}, test log-lik ML={ll_ml:8.2f}, "
          f"Ledoit-Wolf={ll_lw:8.2f} (shrinkage {S_lw.shrinkage_:.2f})")
```

With $d = 40$ and $n = 30$ the ML covariance has rank 29 and assigns zero density to almost every test point (log-likelihood $-\infty$). At $n = 60$ it is invertible but poor (test log-likelihood -94.6 versus -63.0 for Ledoit–Wolf). The automatically chosen shrinkage falls from 0.65 to 0.08 as n grows,

and at $n = 1000$ the two estimates almost coincide. The rule of thumb: the fewer samples per parameter, the more you should shrink.

MAP and Bayesian estimation

Plain version. Maximum likelihood trusts the data completely. Bayesian estimation starts from a prior belief about the parameters and lets the data gradually overrule it. With little data the prior matters; with lots of data it hardly does.

Precise version. Treat θ as a random variable with a **parameter prior** $p(\theta)$. Bayes' rule gives the parameter posterior

$$p(\theta \mid D) = \frac{p(D \mid \theta)\, p(\theta)}{\int p(D \mid \theta')\, p(\theta')\, d\theta'} \;\propto\; p(D \mid \theta)\, p(\theta).$$

A parameter prior is different from the class prior of PR 01. The class prior says how common a category is; a parameter prior says which model settings are plausible before seeing this training set. Three ways to use the posterior are common:

- **MAP estimation** keeps only its mode: $\hat{\theta}_{\text{MAP}} = \arg\max_\theta \left[\log p(D \mid \theta) + \log p(\theta)\right]$. The log prior acts as a regularizer; a Gaussian prior on weights is exactly an L_2 (ridge) penalty, which PR 03 uses.
- **Full Bayesian prediction** averages over all parameter values instead of choosing one:

$$p(x_{\text{new}} \mid D) = \int p(x_{\text{new}} \mid \theta)\, p(\theta \mid D)\, d\theta.$$

This **posterior predictive** distribution carries parameter uncertainty into the prediction.

- **Plug-in prediction** uses $p(x_{\text{new}} \mid \hat{\theta})$ with $\hat{\theta}$ from ML or MAP. It ignores parameter uncertainty and is therefore overconfident when data are scarce.

One subtle point: MAP is not invariant to re-parameterization. The mode of a density over σ is not the square root of the mode of the corresponding density over σ^2, because densities pick up a Jacobian factor when the variable changes. ML estimates and full posteriors do not have this problem.

Worked example: the mean of a Gaussian

Let $x_n \sim \mathcal{N}(\mu, \sigma^2)$ with known σ^2, and put a Gaussian prior on the unknown mean, $\mu \sim \mathcal{N}(m_0, s_0^2)$. Multiplying the likelihood and the prior and completing the square in μ shows that the posterior is again Gaussian, $\mu \mid D \sim \mathcal{N}(m_N, s_N^2)$, with [1, 2]

$$\frac{1}{s_N^2} = \frac{1}{s_0^2} + \frac{N}{\sigma^2}, \qquad m_N = s_N^2 \left(\frac{m_0}{s_0^2} + \frac{N \bar{x}}{\sigma^2} \right),$$

where $\bar{x}$ is the sample mean. Two readings make this formula memorable. **Precisions add**: the posterior precision $1/s_N^2$ is the prior precision plus N times the precision of one observation. **The mean is a precision-weighted average** of the prior mean and the sample mean. As $N \to \infty$, $m_N \to \bar{x}$ (the ML estimate) and $s_N \to 0$. Since the posterior is Gaussian, its mode equals its mean, so here $\hat{\mu}_{\text{MAP}} = m_N$. The posterior predictive is $\mathcal{N}(m_N,\ \sigma^2 + s_N^2)$: wider than the plug-in $\mathcal{N}(m_N,\ \sigma^2)$ by the remaining uncertainty about μ.

A prior whose posterior stays in the same family is called **conjugate**. Conjugate pairs (Gaussian–Gaussian, Beta–Bernoulli, Dirichlet–categorical, Normal–inverse-Wishart for a mean and covariance) give closed-form updates; other priors require the approximate inference methods of PR 08.

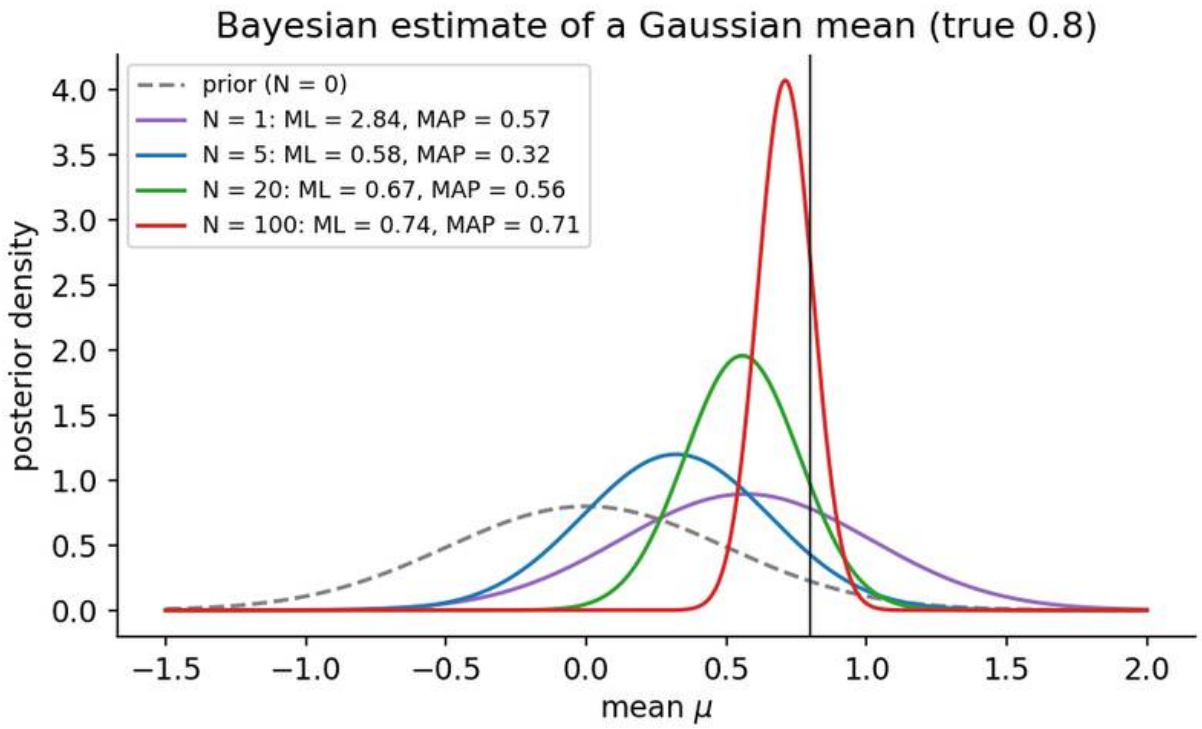


Figure 1 — Bayesian estimation of a Gaussian mean (true value 0.8, prior $N(0, 0.5^2)$). With one sample the ML estimate (2.84) is far off and the posterior mean is pulled back to 0.57; with 100 samples ML (0.74) and MAP (0.71) nearly agree and the posterior is narrow.

Worked example: a Bernoulli rate

For a Bernoulli rate q (for instance the defect rate on a line) with prior $\text{Beta}(a, b)$, observing s successes in N trials gives

$$q \mid D \sim \text{Beta}(a + s,\ b + N - s), \qquad p(x_{\text{new}} = 1 \mid D) = \frac{a + s}{a + b + N}.$$

The prior acts like a imaginary successes and b imaginary failures. With one success and no failures, ML predicts probability one, which is absurd. A uniform prior $\mathrm{Beta}(1,1)$ predicts $2/3$; this is Laplace's rule of succession. The same idea, called **additive** or **Laplace smoothing**, keeps a naive Bayes text classifier from assigning zero probability to a document because one word never appeared with a class in training.

Code: ML, MAP and the posterior predictive

```
import numpy as np

rng = np.random.default_rng(3)
sigma = 1.0                 # known noise standard deviation
m0, s0 = 0.0, 0.5           # prior on the mean: N(m0, s0^2)
x = rng.normal(0.8, sigma, 5)
n = len(x)

mu_ml = x.mean()
post_var = 1 / (1 / s0**2 + n / sigma**2)
post_mean = post_var * (m0 / s0**2 + x.sum() / sigma**2)   # = MAP for a Gaussian
pred_var = sigma**2 + post_var                              # predictive variance

# brute-force check on a grid: posterior is proportional to likelihood x prior
grid = np.linspace(-3, 3, 200001)
log_post = (-0.5 * ((x[:, None] - grid) ** 2).sum(0) / sigma**2
            - 0.5 * (grid - m0) ** 2 / s0**2)
w = np.exp(log_post - log_post.max())
w /= w.sum()
print(f"ML {mu_ml:.3f} | MAP/posterior mean {post_mean:.3f} "
      f"(grid {np.sum(w * grid):.3f}) | posterior sd {np.sqrt(post_var):.3f}")
print(f"predictive sd {np.sqrt(pred_var):.3f} vs plug-in sd {sigma:.3f}")
```

With five samples the ML mean is 0.577, the posterior mean is 0.320 (the grid integration agrees), and the posterior standard deviation is 0.333. The predictive standard deviation, 1.054, is 5% wider than the plug-in value: small here, but it grows quickly as data get scarcer or dimension grows.

Classifiers built from densities

Plain version. Fit one density per class, multiply by the class frequency, and pick the class with the largest product. How you model each density decides what the classifier can learn.

Gaussian plug-in classifiers. Estimate $\hat{\mu}_k$, $\hat{\Sigma}_k$ and $\hat{P}(c_k)$ and insert them into the Gaussian discriminant of PR 01. Separate covariances give QDA, a shared (pooled) covariance gives LDA, and the boundary geometry is exactly as derived there. For computation, use a Cholesky factorization or a linear solve rather than an explicit inverse.

Naive Bayes. Assume that the features are **conditionally independent given the class**:

$$p(x \mid c_k) = \prod_{j=1}^{d} p(x_j \mid c_k).$$

This needs only d one-dimensional densities per class instead of one d-dimensional density, so it can be estimated from very little data. It does *not* assume that features are independent in the overall population; it assumes that the class explains all of their dependence. Gaussian naive Bayes gives each coordinate a univariate Gaussian per class; its boundary is quadratic when the per-class variances differ. When features are strongly correlated (or duplicated), naive Bayes counts the same evidence several times, so its posteriors become overconfident even when its decisions remain reasonable.

Generative versus discriminative. Ng and Jordan [15] compared naive Bayes with logistic regression, its discriminative counterpart. They showed that logistic regression has a lower (or equal) error in the limit of infinite data, but naive Bayes approaches its own, higher, asymptotic error much faster: with a number of samples logarithmic in the dimension, rather than linear. So the generative model often wins with few samples and the discriminative one with many. This is a useful mental model for the trade-off between the two routes.

Non-parametric estimation: one formula, three estimators

Plain version. To estimate density at a point, look at a small region around it and count how many samples fall inside. Density is "count divided by (number of samples times region size)". You can fix the region and count, or fix the count and grow the region.

Precise version. The probability that a sample falls in a small region $\mathcal{R}$ around x is $P = \int_{\mathcal{R}} p(x')\, dx' \approx p(x)\, V$, where V is the volume of $\mathcal{R}$. If k of the N samples fall inside, $k \approx NP$, so [1, 2]

$$\hat{p}(x) = \frac{k}{N\,V}.$$

Two approximations are hidden here, and they pull in opposite directions. $P \approx p(x)V$ needs a *small* region (otherwise $\hat{p}$ is a blurred average of p: **bias**). $k \approx NP$ needs *many* samples inside (otherwise the count is noisy: **variance**). Duda, Hart and Stork [1] show that the estimate converges to $p(x)$ as $N \to \infty$ if the region shrinks ($V_N \to 0$) while the count grows ($k_N \to \infty$) more slowly than N ($k_N/N \to 0$). There are two ways to satisfy this:

- **fix V and count k**: histograms and Parzen windows (kernel density estimation);
- **fix k and measure V**: k-nearest-neighbor density estimation.

Histograms

Divide the axis into bins of width w and set $\hat{p}(x) = n_i/(Nw)$ for x in bin i with count n_i. Histograms are simple and fast, but have three weaknesses visible in Figure 2. The estimate is discontinuous at bin edges. It depends on the **origin** of the grid as well as the width: shifting all bins by 0.3 changes the picture. And in d dimensions the number of bins grows like $(1/w)^d$, so most bins

are empty long before d reaches ten. Scott's analysis [8] shows that the bin width minimizing the mean integrated squared error shrinks only like $N^{-1/3}$, so more data buy finer bins slowly.

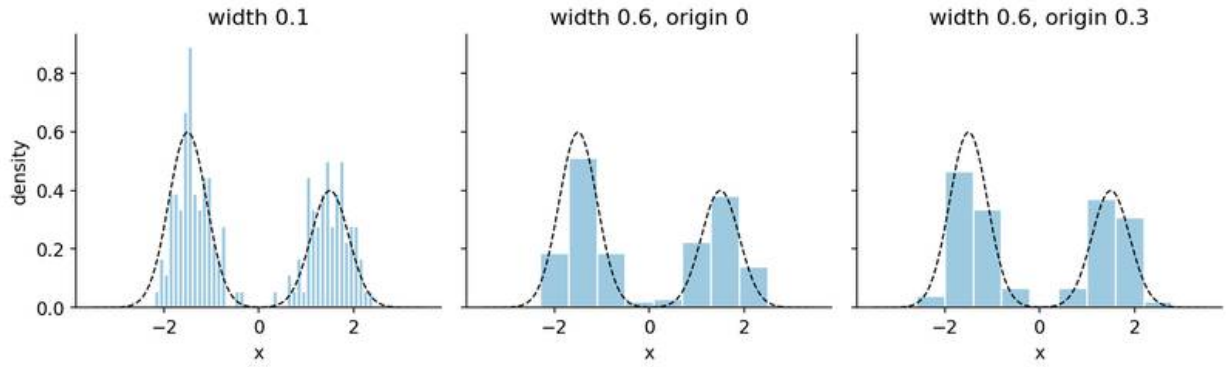


Figure 2 — The same 180 samples (dashed: true density). Narrow bins are noisy; wide bins depend on where the grid starts. The two right panels differ only in the bin origin.

Parzen windows and kernel density estimation

The **Parzen window** or **kernel density estimate (KDE)**, introduced by Rosenblatt [5] and analysed by Parzen [6], centers a window on every sample instead of fixing a grid:

$$\hat{p}_h(x) = \frac{1}{N\,h^d} \sum_{n=1}^{N} K\!\left(\frac{x - x_n}{h}\right).$$

Here K is the **kernel**, a function with $K(u) \geq 0$ and $\int K(u)\,du = 1$ (so $\hat{p}_h$ is a valid density), and $h > 0$ is the **bandwidth**, which plays the role of the bin width. With a box kernel ($K = 1$ inside the unit cube) the formula is literally "count the samples within a cube of side h around x", the fixed-V estimator above. A Gaussian kernel $K(u) = (2\pi)^{-d/2} e^{-\|u\|^2/2}$ replaces the hard count with a smooth weight, giving a smooth estimate (Figure 3, left). In practice the shape of the kernel matters much less than the bandwidth.

Bias and variance. For a smooth density in one dimension, the bias of a KDE grows like h^2 (times the curvature $p''(x)$) and the variance like $1/(Nh)$. Minimizing their sum gives an optimal bandwidth proportional to $N^{-1/5}$ and a mean integrated squared error that falls like $N^{-4/5}$, slower than the N^{-1} of a correctly specified parametric model [7, 8]. In d dimensions the optimal bandwidth scales like $N^{-1/(d+4)}$ and the error like $N^{-4/(d+4)}$. The slowdown with d is one face of the curse of dimensionality.

Choosing the bandwidth. This is the important practical decision. Common options:

- **Rules of thumb.** Silverman's rule $h = 0.9\,\min(\hat{\sigma},\ \mathrm{IQR}/1.34)\,N^{-1/5}$ [7], where $\hat{\sigma}$ is the sample standard deviation and IQR the interquartile range, is optimal if the data are Gaussian. For multimodal data it oversmooths.
- **Cross-validation.** Choose h to maximize the log-likelihood of held-out samples (likelihood

cross-validation), or to minimize an unbiased estimate of the integrated squared error (least-squares cross-validation, analysed by Scott and Terrell [9]).

- **Plug-in methods.** Sheather and Jones [10] estimate the unknown curvature term in the optimal-bandwidth formula from the data and plug it in; their selector is a widely used reliable default in one dimension.

Whatever rule you use, choose h without looking at the final test data. Figure 4 shows why the choice matters: on a bimodal sample, $h = 0.08$ produces spurious bumps, $h = 1.2$ merges the two modes into one, and the validation log-likelihood correctly prefers $h = 0.2$.

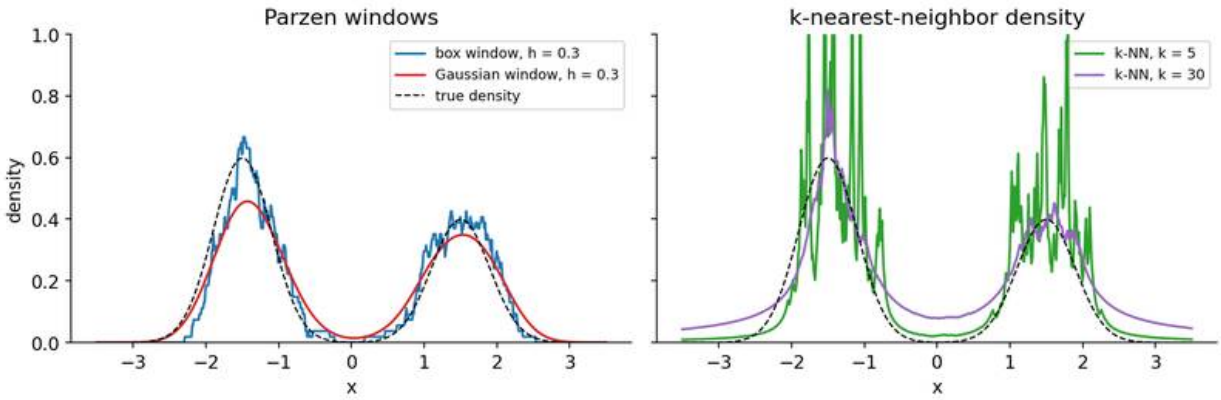


Figure 3 — Left: a box window gives a jagged estimate, a Gaussian window of the same width a smooth one. Right: k-NN density estimates adapt their width to the data but are spiky for small k and have tails that decay too slowly.

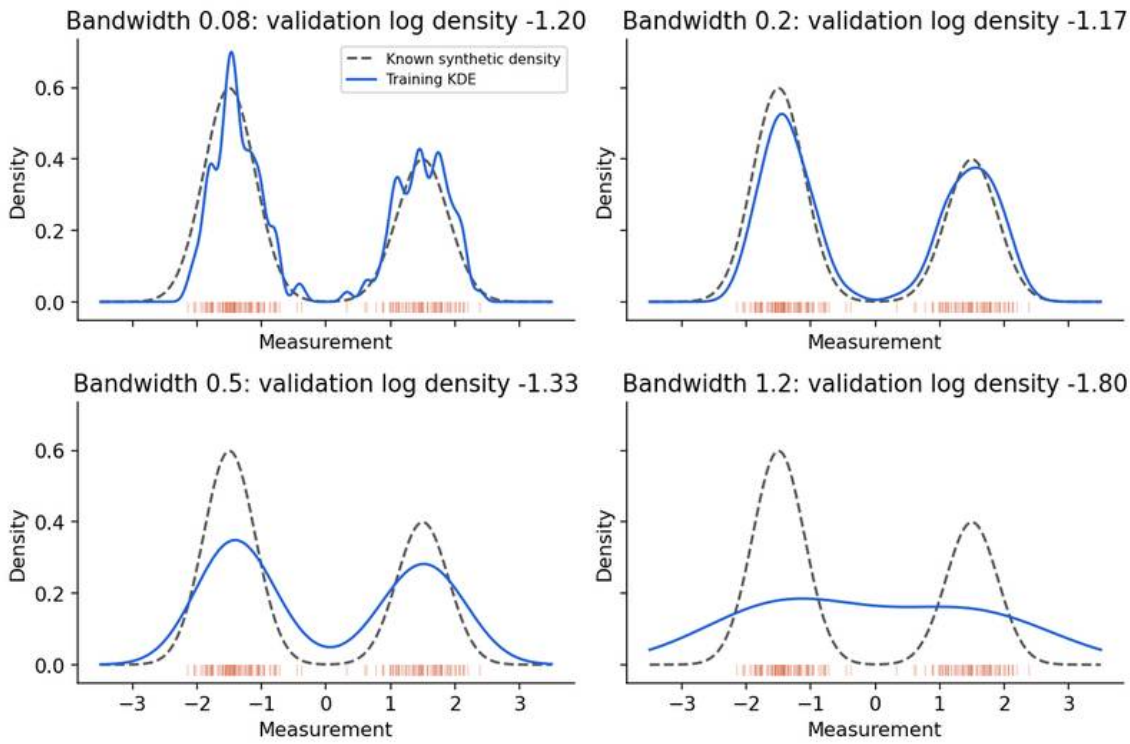


Figure 4 — KDE with four bandwidths on the training sample used in the code below. Narrow bandwidths show sampling fluctuations; wide bandwidths blur the modes. The mean validation log-density (in each title) is highest for $h = 0.2$.

Code: bandwidth selection and a Beta posterior

This example uses scikit-learn's `KernelDensity` [26] on a synthetic bimodal distribution, separate validation samples, Silverman's rule for comparison, and the closed-form Beta posterior from the previous section.

```
import numpy as np
from scipy.stats import beta
from sklearn.neighbors import KernelDensity

rng = np.random.default_rng(4)
def sample(n):
    means = np.where(rng.random(n) < 0.6, -1.5, 1.5)
    return (means + rng.normal(0, 0.4, n)).reshape(-1, 1)
train, validation = sample(180), sample(100)
bandwidths = [0.08, 0.2, 0.5, 1.2]
scores = []
for h in bandwidths:
    model = KernelDensity(bandwidth=h).fit(train)
    scores.append(model.score(validation) / len(validation))
best = bandwidths[int(np.argmax(scores))]
```

```
# Silverman's rule of thumb for comparison
x = train.ravel()
iqr = np.subtract(*np.percentile(x, [75, 25]))
h_silverman = 0.9 * min(x.std(ddof=1), iqr / 1.34) * len(x) ** (-1 / 5)

posterior = beta(2, 1)  # uniform prior + one success
print("validation mean log densities:",
      {h: round(float(s), 3) for h, s in zip(bandwidths, scores)})
print("selected bandwidth:", best, "| Silverman's rule:", round(h_silverman, 3))
print("posterior mean:", posterior.mean())
print("95% credible interval:", posterior.ppf([0.025, 0.975]))
assert np.isclose(posterior.mean(), 2 / 3)
```

Validation selects $h = 0.2$, while Silverman's rule gives $h \approx 0.475$. The rule assumes a single Gaussian; the spread of this sample comes mostly from the gap *between* the two modes, so the rule oversmooths. The Beta posterior after one success under a uniform prior has mean $2/3$ and a 95% credible interval of roughly $[0.16, 0.99]$: one observation tells us very little.

A credible interval describes uncertainty under the assumed prior and likelihood; it is not automatically a calibrated frequentist coverage guarantee. Likewise, a good density score does not by itself guarantee a good classification boundary: some of the modelled variation may be irrelevant to the labels.

k-nearest-neighbor density estimation

Fix k instead of V: around x, grow a ball until it contains exactly k samples, and let $V_k(x)$ be its volume. Then [11]

$$\hat{p}_k(x) = \frac{k}{N\, V_k(x)}.$$

Loftsgaarden and Quesenberry [11] showed that this estimate is consistent when $k \to \infty$ and $k/N \to 0$. Its bandwidth adapts automatically: narrow where data are dense, wide where they are sparse. But Figure 3 (right) shows its flaws. For small k it is very spiky, because V_k is tiny near clusters of samples. In the tails, V_k grows only linearly with the distance in 1-D, so $\hat{p}_k$ decays like $1/|x|$, and the estimate **does not integrate to one** (its integral diverges). It is therefore a poor *density* estimate, yet, as the next section shows, an excellent basis for *classification*.

The k-nearest-neighbor classifier

Plain version. To classify a new point, find the k training points closest to it and take a majority vote. No model is trained; the training set *is* the model.

Precise version. Draw a ball around x that contains k samples in total, of which k_c belong to class c. Applying the $k/(NV)$ estimate separately to each class, with N_c samples in class c, gives

$$\hat{p}(x \mid c) = \frac{k_c}{N_c V}, \qquad \hat{P}(c) = \frac{N_c}{N}, \qquad \hat{p}(x) = \frac{k}{NV} \quad \Longrightarrow \quad \hat{P}(c \mid x) = \frac{\hat{p}(x \mid c)\hat{P}(c)}{\hat{p}(x)} = \frac{k_c}{k}.$$

The volume cancels, and the posterior estimate is simply the fraction of neighbors from each class. The Bayes rule then becomes the **majority vote**. With $k = 1$ it is the **nearest-neighbor rule**.

How good is it? Cover and Hart [12] proved that, as $N \to \infty$, the error $P_{1\mathrm{NN}}$ of the 1-NN rule satisfies

$$P^* \le P_{1\mathrm{NN}} \le P^*\Big(2 - \frac{K}{K-1}P^*\Big) \le 2P^*,$$

where P^* is the Bayes error and K the number of classes. This is an error-rate bound under the theorem's sampling assumptions, not a statement that a neighbor contains half the information in the information-theoretic sense. As k grows (with $k/N \to 0$), the k-NN error approaches P^* itself. This is an asymptotic statement; with finite data and many dimensions, things are much less rosy.

Choosing k. Small k gives low bias but high variance: the 1-NN boundary in Figure 5 wraps around individual noisy points. Large k smooths the boundary but eventually ignores real structure. Choose k by cross-validation. Two practical details: use an odd k for two classes to avoid ties, and **standardize the features** first, because the Euclidean distance is dominated by whichever feature has the largest numeric range. In PR 04, kernels and nearest neighbors are compared with other nonlinear classifiers, and learned distance metrics are discussed there and in later chapters.

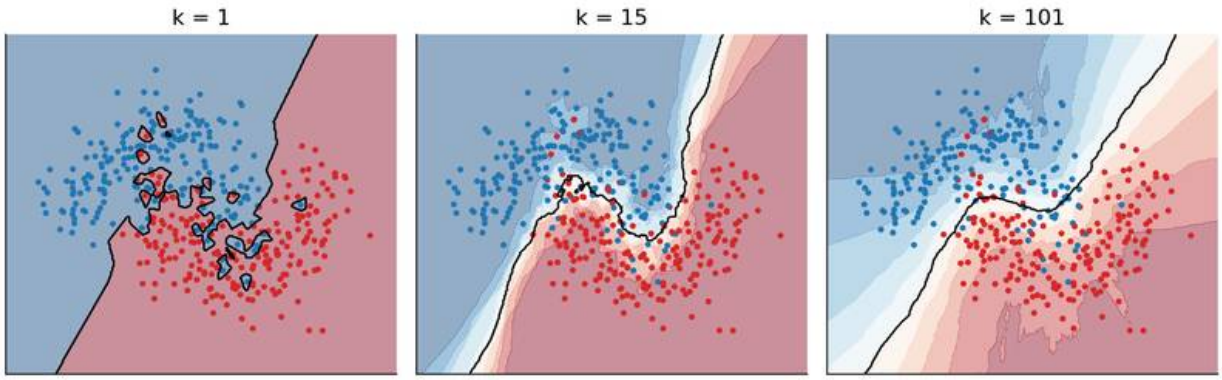


Figure 5 — k-NN on 400 noisy two-moons points. Shading shows the vote fraction k_c/k; the black curve is the 0.5 boundary. k = 1 overfits individual points, k = 101 starts to straighten the moons.

Code: k-NN versus Gaussian classifiers

```
import numpy as np
from sklearn.datasets import make_moons
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
```

```
X, y = make_moons(1000, noise=0.3, random_state=0)
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.5, random_state=0)

search = GridSearchCV(KNeighborsClassifier(),
                      {"n_neighbors": [1, 3, 5, 9, 15, 31, 61, 101]}, cv=5)
search.fit(Xtr, ytr)                    # k chosen on training folds only
print("cross-validated k:", search.best_params_["n_neighbors"])
models = {"1-NN": KNeighborsClassifier(1),
          f"{search.best_params_['n_neighbors']}-NN": search.best_estimator_,
          "Gaussian naive Bayes": GaussianNB(),
          "QDA": QuadraticDiscriminantAnalysis()}
for name, m in models.items():
    m.fit(Xtr, ytr)
    print(f"{name:22s} test error {1 - m.score(Xte, yte):.3f}")
```

Cross-validation picks $k = 31$, with a test error of 10.6%, against 12.4% for 1-NN. The Gaussian models, whose single-ellipse class shapes cannot follow the moons, are worse (17.4% for naive Bayes, 17.6% for QDA). On this 2-D problem, the non-parametric method wins because the parametric form is wrong. The next section shows why the advantage shrinks as dimension grows.

The curse of dimensionality

Plain version. Local methods need neighbors that are close. When many coordinates contain diffuse or irrelevant variation, distance can become a poor guide: to collect even a small fraction of the data you must reach across most of the space, and the nearest and farthest points end up at almost the same distance.

Precise version. The phrase "curse of dimensionality" covers several related effects; Bishop [2, Section 1.4] gives geometric illustrations. Three are easy to check (Figure 6):

1. **Neighborhoods must be huge.** For data uniform in the unit cube $[0,1]^d$, a sub-cube that contains a fraction r of the data has edge length $r^{1/d}$. To capture 10% of the data, the edge is 0.32 in 2-D, 0.79 in 10-D and 0.98 in 100-D. A "local" neighborhood spans almost the full range of every feature, so the estimate is no longer local.
2. **Volume is in the corners and the shell.** The volume of the unit ball divided by the volume of its enclosing cube $[-1,1]^d$ tends to zero rapidly, and for a ball almost all volume lies in a thin shell near its surface. Samples from high-dimensional Gaussians sit at nearly the same distance from the mean, not near it.
3. **Distances concentrate.** Beyer et al. [13] proved that, under broad conditions on the data distribution, the ratio of the farthest to the nearest neighbor distance tends to one as d grows, so "nearest neighbor" stops being meaningful. In the simulation below, the ratio of nearest to farthest distance rises from 0.02 in 2-D to about 0.7 in 100-D.

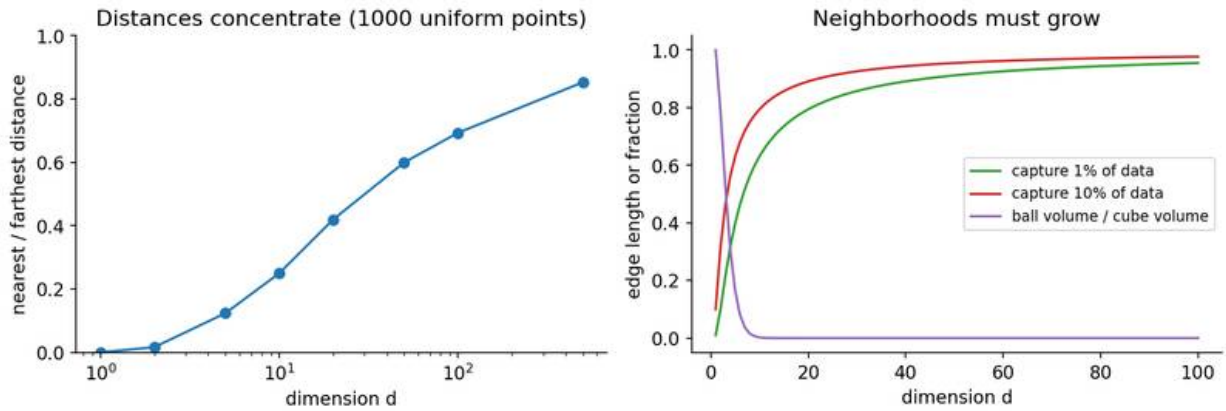


Figure 6 — Left: distance concentration for 1,000 uniform points and 50 queries. Right: edge length of a sub-cube that captures 1% or 10% of uniform data, and the fraction of the cube [-1, 1]^d occupied by the unit ball.

Code: distance concentration

```
import numpy as np

rng = np.random.default_rng(0)
for d in [1, 2, 10, 100, 1000]:
    pts = rng.random((1000, d))          # 1000 uniform points in [0, 1]^d
    q = rng.random((50, d))              # 50 query points
    dist = np.sqrt(((q[:, None, :] - pts[None]) ** 2).sum(-1))
    ratio = (dist.min(1) / dist.max(1)).mean()
    edge = 0.1 ** (1 / d)                # sub-cube edge holding 10% of the volume
    print(f"d={d:5d}: nearest/farthest = {ratio:.3f}, "
          f"edge for 10% of data = {edge:.3f}")
```

The nearest/farthest ratio is 0.017 in 2-D, 0.27 in 10-D, 0.70 in 100-D and 0.89 in 1000-D.

Why does anything work in high dimensions, then? Real data are not uniform. Images, sounds and text lie close to low-dimensional structures (manifolds, clusters) inside their huge ambient spaces, and their informative directions are far fewer than their raw coordinates. Practical defences are therefore: reduce or learn the representation first (PR 05: features and dimensionality), constrain the model (shared or diagonal covariances, naive Bayes), and regularize. Nearest-neighbor methods work very well today precisely *because* they are applied to learned embeddings rather than raw pixels, as the Modern view explains.

Modern view

Density estimation has split into two streams since the textbooks were written: classical statistics refined KDE and covariance estimation, while deep learning built density models that scale to

images, and nearest-neighbor search became infrastructure.

Kernel density estimation, revisited. Chen's tutorial [16] is a gentle modern introduction: convergence rates, bandwidth selection, confidence bands and how to handle their bias, and recent work on estimating geometric and topological features of a density (modes, level sets, ridges) with KDE. Wang and Scott [17] review non-parametric density estimators designed for high-dimensional data, including applications such as modal clustering, and list open research directions. Scott's monograph [8] and Silverman's classic [7] remain the references for the theory.

High-dimensional covariance. The singular-covariance problem of this chapter is a whole research area. Fan, Liao and Liu [18] review estimators of large covariance and precision (inverse covariance) matrices, focusing on rank-based methods and factor-model-based methods; Ledoit–Wolf shrinkage [14] is still the default regularizer in libraries.

Deep density models. Normalizing flows build a flexible density by passing a simple base distribution through a chain of invertible, differentiable maps; the change-of-variables formula then gives an exact log-likelihood. Papamakarios et al. [19] and Kobyzev, Prince and Brubaker [20] are the two standard reviews: the first emphasizes design principles, expressive power and computational trade-offs, the second the construction of flows and open questions. Autoregressive models, VAEs and diffusion models are other ways to learn high-dimensional densities; see ML 07: generative models and ML 08: diffusion and flow matching. Murphy's two-volume text [4] covers both the classical estimators of this chapter and these modern models in one notation.

Nearest neighbors at scale. k-NN is now everywhere, but on learned embeddings instead of raw features. Johnson, Douze and Jégou [21] showed how to run billion-scale similarity search on GPUs, and the Faiss library paper [22] describes the trade-off space of vector search (accuracy, speed, memory) and the indexing methods behind modern vector databases. Papernot and McDaniel's **deep k-NN** [23] compares a test input with its training neighbors in every layer's representation, using label disagreement among neighbors as a confidence and robustness signal. For out-of-distribution detection, Sun et al. [24] use the plain k-th nearest-neighbor distance in feature space, without any distributional assumption, and report large improvements over a parametric Mahalanobis-distance baseline. That parametric baseline, from Lee et al. [25], fits class-conditional Gaussians with shared covariance to network features, which is LDA from PR 01. The parametric versus non-parametric choice of this chapter is being replayed on top of deep features.

Open problems. Reliable density estimation in high dimension remains hard. Nalisnick et al. [27] showed that flow-based models, VAEs and PixelCNNs trained on one image dataset can assign *higher* likelihood to a different dataset (for example, a model trained on a dataset of everyday objects scores a dataset of house-number photos higher), so likelihood alone is not a safe novelty score. Bandwidth or k selection for embeddings, and calibrated uncertainty for non-parametric estimates, are active questions.

Key takeaways

- Generative classifiers need $\hat{p}(x \mid c_k)$ and $\hat{P}(c_k)$; the estimator's assumptions decide the boundary's shape.
- ML estimates for a Gaussian are the sample mean and the divisor-N covariance; with $N \leq d$ samples the covariance is singular, and shrinkage (e.g. Ledoit–Wolf) or shared/diagonal structure is needed.
- MAP keeps the posterior mode; full Bayesian prediction integrates over parameters and is wider than plug-in prediction when data are scarce. Conjugate priors give closed forms: precisions add for a Gaussian mean, counts add for a Beta–Bernoulli rate.
- Non-parametric estimates all follow from $\hat{p} = k/(NV)$: histograms and Parzen/KDE fix V, k-NN fixes k. The bandwidth or k matters more than the kernel; choose it by cross-validation or a justified rule, never on test data.
- The k-NN classifier estimates $P(c \mid x) \approx k_c/k$; asymptotically 1-NN is within a factor two of the Bayes error.
- In high dimensions neighborhoods stop being local and distances concentrate; whether local methods work depends on intrinsic structure, feature scaling and sample size, not ambient dimension alone.

Exercises

1. **Counting parameters.** Count the mean and covariance parameters of one 64-dimensional Gaussian. Explain why fitting it to 30 samples fails, and give two ways to fix it.

   Hint

   64 means plus $64 \cdot 65/2 = 2{,}080$ covariance entries. With 30 samples, $\hat{\Sigma}$ has rank at most 29, so it is singular. Fixes: shrinkage towards τI (Ledoit–Wolf), a diagonal covariance (naive Bayes), a covariance shared across classes, or dimensionality reduction first.

2. **Beta–Bernoulli update.** Start with $\mathrm{Beta}(2, 3)$, observe three successes and two failures, and compute the posterior, the MAP estimate and the posterior predictive probability of success.

   Hint

   Posterior $\mathrm{Beta}(5, 5)$. MAP (mode) $= (5-1)/(5+5-2) = 0.5$. Predictive $= 5/10 = 0.5$. Compare with ML, $3/5 = 0.6$.

3. **Precision-weighted means.** For the Gaussian mean with prior $\mathcal{N}(0, 1)$ and noise variance $\sigma^2 = 4$, after how many samples does the data weight equal the prior weight? What is the posterior standard deviation then?

   Hint

   Prior precision 1; each sample adds $1/4$. Equal weight at $N = 4$. Posterior precision 2, so $s_N = 1/\sqrt{2} \approx 0.71$.

4. **Duplicated features.** Duplicate one feature in Gaussian naive Bayes. Predict what happens to accuracy and to the confidence of the posteriors, then check with scikit-learn on `load_breast_cancer`.

   Hint

   The extra coordinate adds no new information, but the factorized likelihood counts it twice, squaring its likelihood ratio. Decisions may change a little; posteriors move closer to 0 or 1 (more overconfident). Compare `predict_proba` histograms and the log-loss before and after.

5. **k-NN density is not a density.** Show that in 1-D the k-NN estimate $\hat{p}_k(x)$ decays like $1/|x|$ far from the data, and conclude that $\int \hat{p}_k(x)\,dx$ diverges. Why does this not hurt the k-NN classifier?

   Hint

   Far from all samples, the distance to the k-th neighbor is about $|x|$, so $V_k \approx 2|x|$ and $\hat{p}_k \approx k/(2N|x|)$, whose integral is logarithmically divergent. The classifier only uses ratios k_c/k at the same point, where the common volume cancels.

6. **Bandwidth rules.** Generate 500 samples from a mixture of $\mathcal{N}(-2, 0.5^2)$ and $\mathcal{N}(2, 0.5^2)$. Compare Silverman's rule, likelihood cross-validation (`GridSearchCV` over `KernelDensity`) and, if you have `statsmodels`, a plug-in selector. Which oversmooths, and why?

   Hint

   Silverman's rule uses the overall spread (about 2.1 here), which is dominated by the gap between the modes, so it gives a bandwidth several times too large for each mode. Cross-validated likelihood picks something close to the within-mode optimum.

References

1. R. O. Duda, P. E. Hart and D. G. Stork, *Pattern Classification*, 2nd ed., Wiley, 2001 (Ch. 3, maximum-likelihood and Bayesian estimation; Ch. 4, non-parametric techniques). publisher page
2. C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006 (Sections 1.4, 2.1–2.3, 2.5 and 4.2). official page and free PDF
3. S. Theodoridis and K. Koutroumbas, *Pattern Recognition*, 4th ed., Academic Press, 2008 (Ch. 2, density estimation and Bayesian classification). publisher page
4. K. P. Murphy, *Probabilistic Machine Learning: An Introduction*, MIT Press, 2022. official page
5. M. Rosenblatt, "Remarks on some nonparametric estimates of a density function," *The Annals of Mathematical Statistics*, vol. 27, no. 3, pp. 832–837, 1956. doi
6. E. Parzen, "On estimation of a probability density function and mode," *The Annals of Mathematical Statistics*, vol. 33, no. 3, pp. 1065–1076, 1962. doi

7. B. W. Silverman, *Density Estimation for Statistics and Data Analysis*, Chapman and Hall, 1986 (reissued by Routledge, 2018). doi
8. D. W. Scott, *Multivariate Density Estimation: Theory, Practice, and Visualization*, 2nd ed., Wiley, 2015. doi
9. D. W. Scott and G. R. Terrell, "Biased and unbiased cross-validation in density estimation," *Journal of the American Statistical Association*, vol. 82, no. 400, pp. 1131–1146, 1987. doi
10. S. J. Sheather and M. C. Jones, "A reliable data-based bandwidth selection method for kernel density estimation," *Journal of the Royal Statistical Society, Series B*, vol. 53, no. 3, pp. 683–690, 1991. doi
11. D. O. Loftsgaarden and C. P. Quesenberry, "A nonparametric estimate of a multivariate density function," *The Annals of Mathematical Statistics*, vol. 36, no. 3, pp. 1049–1051, 1965. doi
12. T. M. Cover and P. E. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, 1967. doi
13. K. Beyer, J. Goldstein, R. Ramakrishnan and U. Shaft, "When is 'nearest neighbor' meaningful?," in *Database Theory — ICDT'99*, Lecture Notes in Computer Science, Springer, 1999. doi
14. O. Ledoit and M. Wolf, "A well-conditioned estimator for large-dimensional covariance matrices," *Journal of Multivariate Analysis*, vol. 88, no. 2, pp. 365–411, 2004. doi
15. A. Y. Ng and M. I. Jordan, "On discriminative vs. generative classifiers: a comparison of logistic regression and naive Bayes," in *Advances in Neural Information Processing Systems* (*NIPS*), 2001. paper
16. Y.-C. Chen, "A tutorial on kernel density estimation and recent advances," *Biostatistics & Epidemiology*, vol. 1, no. 1, pp. 161–187, 2017. doi
17. Z. Wang and D. W. Scott, "Nonparametric density estimation for high-dimensional data — algorithms and applications," *WIREs Computational Statistics*, vol. 11, no. 4, 2019. doi
18. J. Fan, Y. Liao and H. Liu, "An overview of the estimation of large covariance and precision matrices," *The Econometrics Journal*, vol. 19, no. 1, pp. C1–C32, 2016. doi
19. G. Papamakarios, E. Nalisnick, D. J. Rezende, S. Mohamed and B. Lakshminarayanan, "Normalizing flows for probabilistic modeling and inference," *Journal of Machine Learning Research*, vol. 22, no. 57, pp. 1–64, 2021. arXiv
20. I. Kobyzev, S. J. D. Prince and M. A. Brubaker, "Normalizing flows: an introduction and review of current methods," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 11, pp. 3964–3979, 2021. doi
21. J. Johnson, M. Douze and H. Jégou, "Billion-scale similarity search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2021. doi
22. M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P.-E. Mazaré, M. Lomeli, L. Hosseini and H. Jégou, "The Faiss library," arXiv:2401.08281, 2024. arXiv
23. N. Papernot and P. McDaniel, "Deep k-nearest neighbors: towards confident, interpretable and robust deep learning," arXiv:1803.04765, 2018. arXiv
24. Y. Sun, Y. Ming, X. Zhu and Y. Li, "Out-of-distribution detection with deep nearest neighbors," in *Proc. International Conference on Machine Learning* (*ICML*), PMLR 162, 2022. paper
25. K. Lee, K. Lee, H. Lee and J. Shin, "A simple unified framework for detecting out-of-distribution samples and adversarial attacks," in *Advances in Neural Information Processing Systems* (*NeurIPS*), 2018. paper
26. scikit-learn developers, "Density estimation," scikit-learn user guide. documentation

27. E. Nalisnick, A. Matsukawa, Y. W. Teh, D. Gorur and B. Lakshminarayanan, “Do deep generative models know what they don’t know?,” in *Proc. International Conference on Learning Representations (ICLR)*, 2019. arXiv

PR 03 • Linear Models for Regression and Classification

Linear discriminants from first principles: least squares, Fisher's discriminant, the perceptron, logistic regression and the maximum-margin SVM with its dual, unified through their loss functions.

Online edition: https://blog.shyandram.dev/en/learn/pattern-recognition/linear-models/

What you'll learn

- Read the geometry of a linear discriminant (weight vector, bias, signed distance) and the pitfalls of building multiclass classifiers from two-class ones.
- Derive least squares, ridge regression and their Bayesian interpretation, and see why least squares is a fragile classifier.
- Derive Fisher's linear discriminant and connect it to Gaussian LDA from PR 01.
- Train a perceptron, prove its convergence theorem, and explain why it can stop at any separating line.
- Fit logistic regression by Newton's method, and explain why separable data push its weights to infinity.
- Derive the maximum-margin SVM, its Lagrangian dual and its support vectors, and see all of these methods as different losses on the same linear score.

The big picture

Linear models are the first serious classifiers in every pattern-recognition course, and they remain the first thing to try on a new representation. If a straight boundary separates two categories in feature space, a large nonlinear classifier may add cost without useful accuracy. If a linear baseline fails, its errors help identify missing features or nonlinear structure. In modern deep learning, the final layer of almost every classifier is a linear model on learned features, and "how well does a linear classifier do on these features?" is the standard test of a representation.

> **Explain like I'm five**
>
> Each measurement gets a vote. Multiply each measurement by how strongly it votes, add everything up, and compare the total with a threshold: above means "yes", below means "no". All the methods in this chapter use exactly this voting machine. They only disagree about **how to choose the vote strengths**: make the totals match the right answers (least squares), spread the two groups apart (Fisher), fix mistakes one at a time (perceptron), predict honest probabilities (logistic regression), or leave the widest possible gap between the groups (SVM).

A linear score is $f(x) = w^\top \phi(x) + b$. The **feature map** ϕ might contain raw intensities, squared values, gradients or a frozen neural embedding. "Linear" refers to the parameters w and b, not

necessarily to the original measurement x: with $\phi(x) = (x, x^2)$ a linear model draws a curved boundary in x. This chapter follows Duda, Hart and Stork [1, Ch. 5], Bishop [2, Ch. 3–4, 7] and Theodoridis and Koutroumbas [3, Ch. 3], with Hastie et al. [4] for the statistical view. Kernels, which make ϕ implicit and very rich, are the topic of PR 04.

Linear discriminant functions

Plain version. A two-class linear classifier is a flat cut through feature space: a line in 2-D, a plane in 3-D, a hyperplane in general. The weight vector says which way the cut faces; the bias says where it sits.

Precise version. For two classes, decide c_1 if $g(x) = w^\top x + b > 0$ and c_0 otherwise. The decision boundary $g(x) = 0$ is a hyperplane H. Three facts follow from the geometry of linear classifiers [1, 2, 3]:

- w is **normal** to H: for two points x_a, x_b on H, $w^\top(x_a - x_b) = 0$.
- The **signed distance** from any x to H is $r = g(x)/\|w\|$. So $g(x)$ is proportional to how far, and on which side, x lies.
- For $w \neq 0$, the ordinary distance from the origin to H is $|b|/\|w\|$. Its signed distance under the convention above is $b/\|w\|$; the closest point on H is $-bw/\|w\|^2$. The bias shifts the plane without rotating it.

It is convenient to absorb the bias by **augmenting** the input with a constant: $\tilde{x} = (x, 1)$ and $\tilde{w} = (w, b)$, so that $g(x) = \tilde{w}^\top \tilde{x}$. We do this silently in code. With labels $y \in \{-1, +1\}$, a sample is correctly classified exactly when its **functional margin** $y\,g(x)$ is positive. Every method below can be written in terms of this quantity.

More than two classes. Two-class machines can be combined in two obvious ways: **one-versus-rest** (K classifiers, each separating one class from all others) and **one-versus-one** ($K(K-1)/2$ classifiers, one per pair, followed by voting). Both can leave regions of input space where the votes are ambiguous [2, Section 4.1.2]. The clean solution is a **linear machine**: one discriminant per class, $g_k(x) = w_k^\top x + b_k$, and assign x to the class with the largest score. Its decision regions are always convex and simply connected, because if x_a and x_b both have g_k largest, so does every point between them (each g_j is linear along the segment). The Gaussian LDA of PR 01, multiclass logistic regression and the final layer of a neural network are all linear machines.

Least squares and ridge regression

Plain version. For predicting a quantity, choose the weights that make the squared prediction errors as small as possible. If there are too many weights for the data, also penalize large weights.

Precise version. Let row n of the **design matrix** $\Phi \in \mathbb{R}^{N \times M}$ be the feature vector $\phi(x_n)^\top$, including a constant coordinate for the intercept, and let $t \in \mathbb{R}^N$ hold the targets. Least squares minimizes

$$\mathcal{L}(w) = \frac{1}{2}\|\Phi w - t\|_2^2,$$

whose gradient $\Phi^\top(\Phi w - t)$ vanishes at the **normal equations** $\Phi^\top\Phi w = \Phi^\top t$. Forming $\Phi^\top\Phi$ squares the 2-norm condition number for a full-column-rank design, and explicitly inverting it is unnecessary; QR- or SVD-based solvers (`np.linalg.lstsq`) handle near-dependent columns more reliably. Assuming independent Gaussian noise of common variance on the targets, this objective equals the negative log-likelihood up to a positive scale and an additive constant, so least squares is the maximum-likelihood estimate (PR 02). The noise assumption explains the statistical interpretation; it does not make squared error suitable for every target.

Ridge regression [8] adds a penalty $\frac{\lambda}{2}\|w\|^2$, giving

$$(\Phi^\top\Phi + \lambda I)\, w = \Phi^\top t.$$

The matrix on the left is invertible for any $\lambda > 0$, even when $M > N$. Usually the intercept is excluded from the penalty. Feature scaling matters, because the same penalty has different effects on coordinates measured in different units; fit scaling statistics on training data only.

A Bayesian connection

With a Gaussian prior $w \sim \mathcal{N}(0, \alpha^{-1}I)$ (precision α) and Gaussian noise of precision β, the posterior over weights is Gaussian with covariance and mean [2, Section 3.3]

$$S = (\alpha I + \beta\Phi^\top\Phi)^{-1}, \qquad m = \beta S\Phi^\top t.$$

For a new feature vector ϕ_*, the predictive distribution is

$$p(t_* \mid x_*, D) = \mathcal{N}(t_* \mid m^\top\phi_*,\ \beta^{-1} + \phi_*^\top S\phi_*).$$

The first variance term is observation noise; the second expresses uncertainty about the weights, which is larger along feature directions that the training data rarely visited. The posterior mean m equals the ridge solution with $\lambda = \alpha/\beta$: ridge is a MAP estimate, the same story as the Gaussian mean in PR 02.

Least squares for classification

Plain version. You can treat class labels as numbers ($+1$ and -1, or one-hot vectors) and fit them with least squares. It is fast and has a closed form, but it is the wrong tool: it punishes points for being *too* correct.

Precise version. For K classes, encode the targets as **one-hot** vectors $t_n \in \{0,1\}^K$ and fit a linear machine $g(x) = W^\top\tilde{x}$ by least squares, $W = (\tilde{X}^\top\tilde{X})^{-1}\tilde{X}^\top T$, then classify by the largest output. For two classes with suitable target values (N/N_1 for class 1 and $-N/N_0$ for class 0), the least-squares direction is exactly Fisher's direction derived in the next section [2, Section 4.1.5], so the method is not unreasonable.

It has two well-known failure modes:

1. **Sensitivity to points far on the correct side.** Squared error penalizes $g(x) = 5$ for a target of $+1$ as severely as $g(x) = -3$. Figure 1 adds twenty red points that are correctly classified but far away; least squares rotates its boundary to reduce their "error" and starts misclassifying blue points. Logistic regression barely moves.
2. **Masking with three or more classes.** When classes lie roughly along a line, the output for the middle class can be dominated everywhere by one of its neighbors, so the middle class is never predicted [4, Section 4.2].

In probabilistic terms, the Gaussian noise model behind least squares is simply wrong for binary targets. The remedy is to use a loss designed for classification, which the rest of the chapter does.

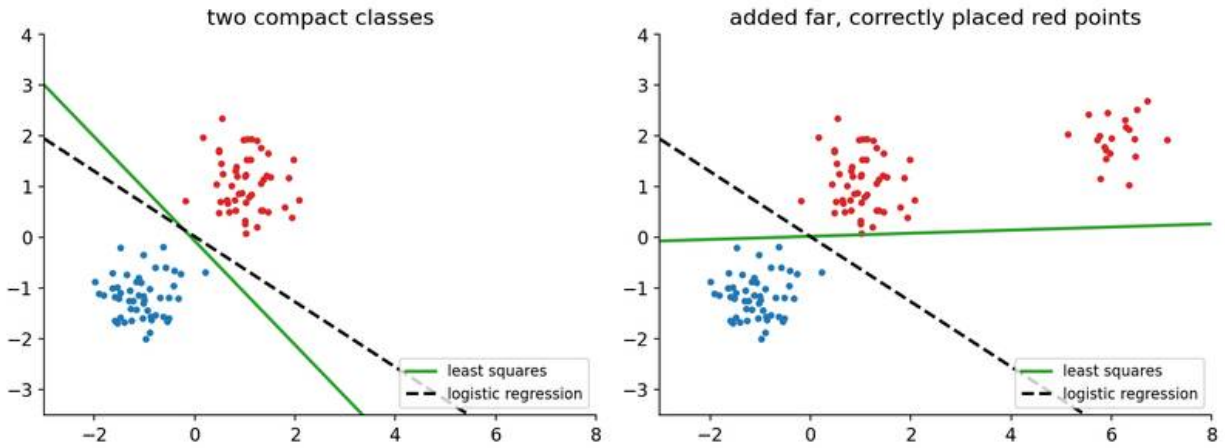


Figure 1 — Least squares versus logistic regression. Adding red points that are already correctly classified (right) pulls the least-squares boundary (green) into the blue class; logistic regression (dashed) is unaffected.

Fisher's linear discriminant

Plain version. Project all points onto one line and look at the two classes' 1-D histograms. A good line makes the class means far apart *and* each class narrow. Fisher's direction is the line that maximizes this ratio.

Precise version. Fisher [5] proposed choosing a projection $z = w^\top x$ that maximizes the separation of the projected class means relative to the spread within each class. Let $m_k = \frac{1}{N_k} \sum_{n \in c_k} x_n$ be the class means. The projected means are $w^\top m_k$, and the projected within-class scatter of class k is $\sum_{n \in c_k} (w^\top x_n - w^\top m_k)^2 = w^\top S_k w$, with $S_k = \sum_{n \in c_k} (x_n - m_k)(x_n - m_k)^\top$. **Fisher's criterion** is

$$J(w) = \frac{(w^\top m_1 - w^\top m_0)^2}{w^\top S_0 w + w^\top S_1 w} = \frac{w^\top S_B w}{w^\top S_W w},$$

where

$$S_B = (m_1 - m_0)(m_1 - m_0)^\top, \qquad S_W = S_0 + S_1$$

are the **between-class** and **within-class scatter matrices**. J does not depend on the length of w, only on its direction.

Derivation. Setting the gradient of the ratio to zero gives

$$(w^\top S_B w)\, S_W w = (w^\top S_W w)\, S_B w.$$

Both bracketed terms are scalars. Moreover $S_B w = (m_1 - m_0)\,[(m_1 - m_0)^\top w]$ always points along $m_1 - m_0$. Hence $S_W w \propto m_1 - m_0$, and since only the direction matters,

$$w^\star \propto S_W^{-1}(m_1 - m_0).$$

If S_W is a multiple of the identity (isotropic, uncorrelated classes), Fisher's direction is just the mean difference. Otherwise S_W^{-1} rotates it away from directions in which the classes are spread out. Figure 2 shows the effect for two elongated, correlated classes: projecting on the mean difference gives heavily overlapping histograms ($J = 0.70$), while Fisher's direction separates them well ($J = 3.18$).

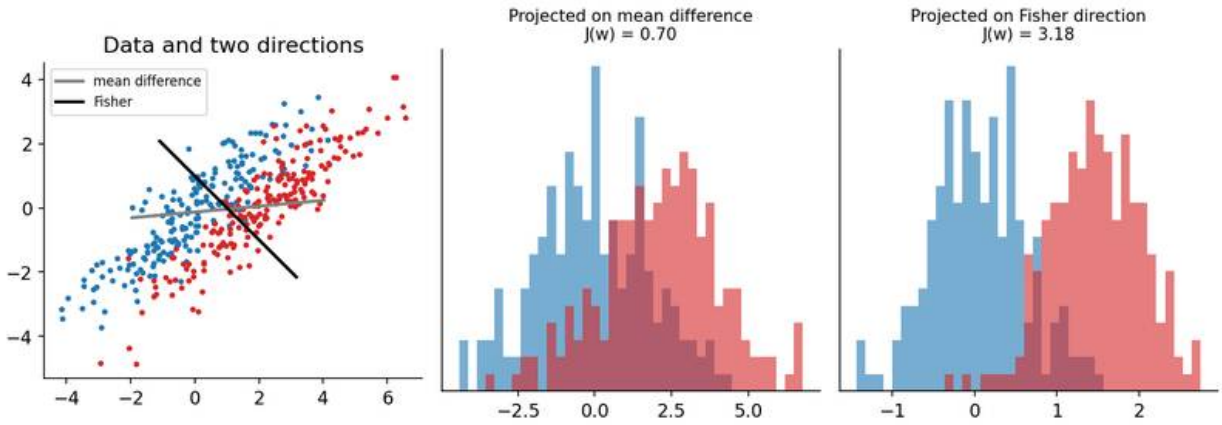


Figure 2 — Two correlated classes projected onto the mean difference (middle, $J = 0.70$) and onto Fisher's direction (right, $J = 3.18$). Per-class covariances were used for S_W; with equal class sizes this only rescales J.

Relationship with Gaussian LDA. Fisher's criterion makes no distributional assumption; it is a criterion for a *representation*. Yet its direction coincides with the Gaussian LDA direction $\Sigma^{-1}(\mu_1 - \mu_0)$ of PR 01 when S_W is (proportional to) the pooled within-class covariance. The two differ only in the threshold: Fisher leaves the cut point on the projected axis open, while Gaussian LDA fixes it from the class means and priors. So "LDA" is used for both. (In topic modelling, LDA also means latent Dirichlet allocation, which is unrelated.) Fisher projections do not by themselves provide calibrated probabilities.

More than two classes. With K classes and overall mean m, define $S_W = \sum_k S_k$ and $S_B = \sum_k N_k (m_k - m)(m_k - m)^\top$. The projection $W \in \mathbb{R}^{d \times (K-1)}$ that maximizes the multiclass criterion is spanned by the leading eigenvectors of $S_W^{-1} S_B$ [1, 2]. Because S_B is a sum of K rank-one terms constrained by the overall mean, its rank is at most $K - 1$, so Fisher can produce at most $K - 1$ useful features. This is the supervised dimensionality reduction of PR 05.

Code: Fisher's direction

```
import numpy as np
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

rng = np.random.default_rng(2)
cov = np.array([[3.0, 2.4], [2.4, 2.4]])
X0 = rng.multivariate_normal([0, 0], cov, 200)
X1 = rng.multivariate_normal([2, 0], cov, 200)
m0, m1 = X0.mean(0), X1.mean(0)
S_W = np.cov(X0.T) + np.cov(X1.T)       # within-class scatter (per-class covariances)
w = np.linalg.solve(S_W, m1 - m0)                          # Fisher direction

def J(v):                         # Fisher criterion for two classes
    return (v @ (m1 - m0)) ** 2 / (v @ S_W @ v)

lda = LinearDiscriminantAnalysis().fit(np.vstack([X0, X1]), np.repeat([0, 1], 200))
cos = w @ lda.coef_[0] / np.linalg.norm(w) / np.linalg.norm(lda.coef_[0])
angles = np.linspace(0, np.pi, 3601)
best = max(J(np.array([np.cos(a), np.sin(a)])) for a in angles)
print(f"J(mean difference) = {J(m1 - m0):.4f}, J(Fisher) = {J(w):.4f}, "
      f"grid maximum = {best:.4f}")
print(f"cosine between Fisher direction and LDA coef_: {cos:.6f}")
```

The closed form attains the grid maximum ($J = 3.1841$), more than four times the mean-difference value (0.6953), and its direction agrees with scikit-learn's Gaussian LDA coefficients to six decimals (cosine 1.000000).

The perceptron

Plain version. Start with any line. Go through the training points; whenever one is on the wrong side, nudge the line towards classifying it correctly. If the classes can be separated by a line, this process is guaranteed to stop.

Precise version. Rosenblatt's perceptron [6] uses labels $y_n \in \{-1, +1\}$ and augmented features $\tilde{x}_n$. It minimizes the **perceptron criterion**

$$J_P(w) = -\sum_{n \in \mathcal{M}} y_n \, w^\top \tilde{x}_n,$$

where $\mathcal{M}$ is the set of misclassified samples (those with $y_n w^\top \tilde{x}_n \le 0$). Each term is non-negative, and $J_P = 0$ exactly when every sample is correctly classified. Stochastic gradient descent on one misclassified sample at a time gives the update

$$w \leftarrow w + \eta \, y_n \tilde{x}_n.$$

With w initialized at zero, the learning rate $\eta > 0$ only rescales w and does not change the sequence of decisions, so $\eta = 1$ is common. The IP 13 perceptron section presents the same rule in the language of image patterns and connects it to multilayer networks.

Convergence theorem. Suppose the data are **linearly separable with margin** $\gamma > 0$: some unit vector u satisfies $y_n u^\top \tilde{x}_n \geq \gamma$ for all n. Let $R = \max_n \|\tilde{x}_n\|$. Then the perceptron (started at $w_0 = 0$) makes at most $(R/\gamma)^2$ updates. The proof takes three lines. After t updates:

- each update adds $y_n \tilde{x}_n^\top u \geq \gamma$ to $w^\top u$, so $w_t^\top u \geq t\gamma$;
- each update happens on a mistake, where $y_n w^\top \tilde{x}_n \leq 0$, so $\|w_t\|^2 = \|w_{t-1}\|^2 + 2y_n w_{t-1}^\top \tilde{x}_n + \|\tilde{x}_n\|^2 \leq \|w_{t-1}\|^2 + R^2$, hence $\|w_t\|^2 \leq tR^2$;
- by Cauchy–Schwarz, $t\gamma \leq w_t^\top u \leq \|w_t\| \leq \sqrt{t}\, R$, so $t \leq R^2/\gamma^2$.

The bound does not depend on the number of samples or the dimension, only on how "fat" the gap between the classes is relative to the data's size.

Limitations. If the classes overlap, the perceptron never stops; the weights keep cycling (variants that keep the best weights seen so far are a common practical fix). Even when it converges, it stops at the *first* separating hyperplane it reaches, which depends on initialization and data order (Figure 3, left). And its score is not a probability. In terms of losses, the perceptron uses $\max(0, -y\, g(x))$, which is zero for any correct classification, however marginal. Both logistic regression and the SVM fix this by asking for more than "just barely correct".

Code: perceptron updates versus the bound

```
import numpy as np
from sklearn.svm import SVC

rng = np.random.default_rng(1)
X = np.vstack([rng.normal([-1.0, -0.7], 0.6, (30, 2)),
               rng.normal([1.0, 0.7], 0.6, (30, 2))])
y = np.r_[-np.ones(30), np.ones(30)]
Phi = np.c_[X, np.ones(len(X))]          # augmented features (bias as last weight)

def perceptron(Phi, y, rng, max_epochs=1000):
    w, updates = np.zeros(Phi.shape[1]), 0
    for epoch in range(max_epochs):
        errors = 0
        for i in rng.permutation(len(y)):
            if y[i] * (w @ Phi[i]) <= 0:  # misclassified or on the boundary
                w += y[i] * Phi[i]        # learning rate 1 (scale is irrelevant)
                errors += 1
        updates += errors
        if errors == 0:
            return w, updates
    return w, updates

runs = [perceptron(Phi, y, np.random.default_rng(s)) for s in range(200)]
updates = np.array([u for _, u in runs])
print(f"updates over 200 random orders: mean {updates.mean():.1f}, "
      f"max {updates.max()}")
print("all runs separate the data:",
      all(np.all(np.sign(Phi @ w) == y) for w, _ in runs))
```

```
# convergence bound (R / gamma)^2, with gamma from the max-margin separator
svm = SVC(kernel="linear", C=1e6).fit(X, y)
u = np.r_[svm.coef_[0], svm.intercept_[0]]
gamma = np.min(y * (Phi @ u)) / np.linalg.norm(u)
R = np.max(np.linalg.norm(Phi, axis=1))
print(f"bound (R/gamma)^2 = {(R / gamma) ** 2:.1f}")
```

Over 200 random presentation orders the perceptron needs on average 2.7 updates (at most 10), and every run separates the data. The worst-case bound is 691: it is a guarantee, not a prediction. The bound needs a valid margin γ; we take it from a hard-margin SVM (the subject of the SVM section below). Because the SVM does not penalize the bias, the true maximum margin in the augmented space may be slightly larger, which would give a slightly tighter bound.

Logistic regression

Plain version. Instead of a hard yes/no, output a probability by squashing the linear score through an S-shaped curve. Fit the weights so that the probabilities assigned to the true labels are as high as possible.

Precise version. With labels $y \in \{0, 1\}$, binary logistic regression [7] models

$$P(y = 1 \mid x) = \sigma(w^\top \tilde{x}), \qquad \sigma(z) = \frac{1}{1 + e^{-z}}.$$

The score $w^\top \tilde{x}$ is the **log-odds** $\ln \frac{P(y=1|x)}{P(y=0|x)}$. This form is not arbitrary. PR 01 showed that for Gaussian classes with a shared covariance the posterior is exactly $\sigma(w^\top x + w_0)$ with $w = \Sigma^{-1}(\mu_1 - \mu_0)$; the same holds for many other class-conditional families. Logistic regression keeps the functional form of that posterior but fits w directly, with $d + 1$ parameters instead of the $2d + d(d+1)/2 + 1$ of the Gaussian model. Ng and Jordan [18] analysed this generative–discriminative trade-off (PR 02).

Training. The negative log-likelihood of the Bernoulli model is the **cross-entropy**

$$E(w) = -\sum_{n=1}^{N} \Big[y_n \ln p_n + (1 - y_n) \ln(1 - p_n) \Big], \qquad p_n = \sigma(w^\top \tilde{x}_n).$$

Because $\sigma'(z) = \sigma(z)(1 - \sigma(z))$, its gradient and Hessian are remarkably simple [2, Section 4.3]:

$$\nabla E = \sum_n (p_n - y_n)\, \tilde{x}_n = \tilde{X}^\top (p - y), \qquad H = \tilde{X}^\top R\, \tilde{X}, \quad R = \operatorname{diag}(p_n(1 - p_n)).$$

The gradient has the same "prediction minus target, times input" form as least squares. H is positive semidefinite, so E is convex and has no spurious local minima. Newton's method, $w \leftarrow w - H^{-1}\nabla E$, can be rewritten as a weighted least-squares problem with weights R that change every iteration: **iteratively reweighted least squares (IRLS)**. In code, compute losses from logits with a stable formula such as $\ln(1 + e^z) =$ `np.logaddexp(0, z)` rather than taking logarithms of rounded probabilities.

Separable data. If a hyperplane separates the training classes, scaling w up pushes every p_n closer to its label, so E keeps decreasing towards zero and the unregularized solution does not exist: $\|w\| \to \infty$. In practice, always regularize, e.g. add $\frac{\lambda}{2}\|w\|^2$ (scikit-learn's `C` is $1/\lambda$). Remarkably, *how* the weights diverge is well defined: plain gradient descent on separable data converges in direction to the maximum-margin SVM solution [19], which the Modern view returns to.

Multiclass. Replace the sigmoid with the **softmax** $P(y = k \mid x) = \exp(w_k^\top \tilde{x}) / \sum_j \exp(w_j^\top \tilde{x})$ and the cross-entropy with its K-class version. This is a linear machine with probabilistic outputs, and it is exactly the last layer plus loss of a standard neural classifier. Probability outputs are not automatically calibrated under label noise or distribution shift (PR 01).

Code: Newton's method (IRLS) and separable data

```
import numpy as np
from sklearn.linear_model import LogisticRegression

rng = np.random.default_rng(0)
X = np.vstack([rng.normal(-1, 1, (100, 2)), rng.normal(1, 1, (100, 2))])
y = np.r_[np.zeros(100), np.ones(100)]
Phi = np.c_[np.ones(len(X)), X]

def nll(w, lam):
    z = Phi @ w
    # log(1 + e^z) - y z, written stably with logaddexp
    return np.sum(np.logaddexp(0, z) - y * z) + 0.5 * lam * np.sum(w[1:] ** 2)

# Newton's method (iteratively reweighted least squares)
lam, w = 1.0, np.zeros(3)
for it in range(20):
    p = 1 / (1 + np.exp(-(Phi @ w)))
    grad = Phi.T @ (p - y) + lam * np.r_[0, w[1:]]
    H = Phi.T @ (Phi * (p * (1 - p))[:, None]) + lam * np.diag([0, 1, 1])
    step = np.linalg.solve(H, grad)
    w -= step
    if np.linalg.norm(step) < 1e-10:
        break
sk = LogisticRegression(C=1 / lam, tol=1e-10, max_iter=1000).fit(X, y)
print(f"Newton converged in {it + 1} iterations, loss {nll(w, lam):.4f}")
print("Newton:      ", np.round(w, 4))
print("scikit-learn:", np.round(np.r_[sk.intercept_, sk.coef_[0]], 4))
```

Newton's method converges in 8 iterations to the same weights as scikit-learn, $(0.0777, 1.7205, 1.5300)$ for (intercept, w_1, w_2). The penalty excludes the intercept, matching scikit-learn's convention.

The next snippet runs plain gradient descent on *unregularized* logistic loss for data that a line through the origin separates. It tracks the weight norm and the cosine between w and the max-margin direction.

```
import numpy as np
from sklearn.svm import LinearSVC

rng = np.random.default_rng(1)
X = np.vstack([rng.normal([2.0, 1.0], 0.6, (40, 2)),
               rng.normal([-2.0, -1.0], 0.6, (40, 2))])
y = np.r_[np.ones(40), -np.ones(40)]              # separable through the origin

# max-margin direction without bias (hard margin approximated by large C)
svm = LinearSVC(fit_intercept=False, loss="hinge", C=1e4, max_iter=200000).fit(X, y)
u = svm.coef_[0] / np.linalg.norm(svm.coef_[0])

w, lr = np.zeros(2), 0.1
for t in range(1, 100001):
    m = y * (X @ w)
    grad = -(X * (y / (1 + np.exp(m)))[:, None]).mean(0)   # logistic loss gradient
    w -= lr * grad
    if t in (10, 100, 1000, 10000, 100000):
        print(f"step {t:6d}: ||w|| = {np.linalg.norm(w):6.2f}, "
              f"cos(w, SVM) = {w @ u / np.linalg.norm(w):.5f}")
```

The norm grows without bound but only slowly (0.68 after 10 steps, 8.64 after 10^5), while the cosine with the SVM direction creeps from 0.87 to 0.98. This is the logarithmically slow convergence to the max-margin direction predicted by Soudry et al. [19].

Support vector machines: maximum margin

Plain version. Among all lines that separate the classes, pick the one with the widest empty "street" around it. Only the points on the edges of the street matter; move any other point (without crossing the street) and the answer does not change.

Precise version. Use labels $y_n \in \{-1, +1\}$ and score $f(x) = w^\top x + b$ (bias kept separate here). For a separating hyperplane, the distance of x_n to the boundary is $y_n f(x_n)/\|w\|$. Since (w, b) and (cw, cb) define the same plane, fix the scale so that the closest points satisfy $y_n f(x_n) = 1$ (the **canonical** form). The margin on each side is then $1/\|w\|$ and the street has width $2/\|w\|$. Maximizing it is equivalent to the convex quadratic program [9, 10]

$$\min_{w,b} \frac{1}{2}\|w\|^2 \quad \text{subject to} \quad y_n(w^\top x_n + b) \geq 1, \quad n = 1, \dots, N.$$

Lagrangian and dual. Introduce multipliers $a_n \geq 0$, one per constraint:

$$\mathcal{L}(w, b, a) = \frac{1}{2}\|w\|^2 - \sum_n a_n[y_n(w^\top x_n + b) - 1].$$

Setting $\partial\mathcal{L}/\partial w = 0$ and $\partial\mathcal{L}/\partial b = 0$ gives

$$w = \sum_n a_n y_n x_n, \qquad \sum_n a_n y_n = 0.$$

Substituting back eliminates w and b and leaves the **dual problem**

$$\max_{a} \sum_{n} a_n - \frac{1}{2} \sum_{n} \sum_{m} a_n a_m y_n y_m \, x_n^{\top} x_m \quad \text{subject to} \quad a_n \geq 0, \;\; \sum_{n} a_n y_n = 0.$$

Three consequences matter [2, 11]:

- **Support vectors.** The Karush–Kuhn–Tucker conditions include **complementary slackness**, $a_n[y_n f(x_n) - 1] = 0$. So $a_n > 0$ only for points exactly on the margin. These are the **support vectors**; all other points have $a_n = 0$ and can be removed without changing the solution. The bias follows from any support vector s: $b = y_s - w^{\top} x_s$ (average over all of them for numerical stability).
- **Sparsity of the prediction.** $f(x) = \sum_{n \in \mathcal{S}} a_n y_n \, x_n^{\top} x + b$ depends only on support vectors $\mathcal{S}$.
- **Only inner products.** The data appear in the dual and in the prediction only through inner products $x_n^{\top} x_m$. Replacing them with a kernel $k(x_n, x_m)$ gives a nonlinear SVM without ever computing ϕ. That is the **kernel trick**, developed in PR 04.

Soft margin and the hinge loss. Real classes overlap, so the constraints cannot all hold. Cortes and Vapnik [10] added **slack variables** $\xi_n \geq 0$, constraints $y_n f(x_n) \geq 1 - \xi_n$, and a penalty $C \sum_n \xi_n$. At the optimum $\xi_n = \max(0, 1 - y_n f(x_n))$, so the soft-margin SVM is equivalent to the unconstrained problem

$$\min_{w,b} \; \frac{1}{2} \|w\|^2 + C \sum_{n} \max(0, \; 1 - y_n f(x_n)).$$

The second term is the **hinge loss**: zero for points beyond the margin, linear inside it and on the wrong side. In the dual the only change is a box constraint $0 \leq a_n \leq C$. So a linear SVM is "ridge-regularized hinge-loss minimization", directly comparable to regularized logistic regression. The roles of C, kernel bandwidth and feature scaling, and the fact that an SVM score is not a probability, are treated in PR 04.

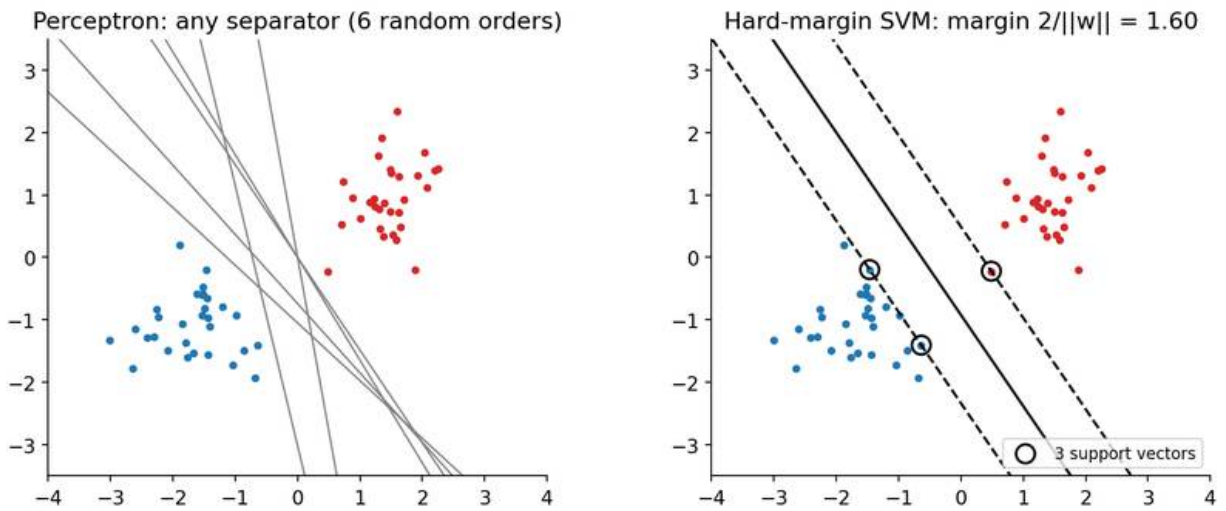


Figure 3 — Left: the perceptron stops at whichever separating line it reaches first (six random orders).

Right: the hard-margin SVM picks the unique line with the widest margin; only the three circled support vectors determine it.

Code: the dual in practice

This check fits a (nearly) hard-margin linear SVM and verifies the dual relations numerically: $w = \sum_n a_n y_n x_n$, $\sum_n a_n y_n = 0$, and $y_n f(x_n) = 1$ on the support vectors.

```
import numpy as np
from sklearn.svm import SVC

rng = np.random.default_rng(7)
X = np.vstack([rng.normal([-1.5, -1.0], 0.6, (30, 2)),
               rng.normal([1.5, 1.0], 0.6, (30, 2))])
y = np.r_[-np.ones(30), np.ones(30)]

svm = SVC(kernel="linear", C=1e6).fit(X, y)          # huge C ~ hard margin
alpha_y = svm.dual_coef_[0]                          # alpha_n * y_n for SVs
w_dual = alpha_y @ svm.support_vectors_              # w = sum_n alpha_n y_n x_n
f = X @ svm.coef_[0] + svm.intercept_[0]

print("support vectors:", len(svm.support_), "of", len(y))
print("w from dual:", np.round(w_dual, 4), " coef_:", np.round(svm.coef_[0], 4))
print("sum_n alpha_n y_n =", round(float(alpha_y.sum()), 8))
print("y f(x) on support vectors:", np.round(y[svm.support_] * f[svm.support_], 4))
print("smallest y f(x) over all points:", round(float(np.min(y * f)), 4))
print("margin width 2/||w|| =", round(2 / np.linalg.norm(svm.coef_[0]), 4))
```

Only 3 of the 60 points are support vectors (the circled points in Figure 3). The weight vector rebuilt from the dual coefficients matches `coef_` exactly, the dual coefficients sum to zero, the support vectors sit at $yf(x) \approx 1$ (to solver tolerance), no point is inside the margin, and the margin width is 1.60. scikit-learn's `SVC` wraps LIBSVM [16]; for large sparse problems, `LinearSVC` and `LogisticRegression(solver="liblinear")` use LIBLINEAR [14], and stochastic sub-gradient solvers such as Pegasos [15] scale to very large datasets.

One picture: linear classifiers as losses

Plain version. All the methods in this chapter score a point with the same linear function. They differ in how much they charge for each value of the margin $y\,g(x)$: nothing once correct (perceptron), nothing once safely correct (SVM), a smoothly decreasing amount (logistic), or a parabola centred at exactly 1 (least squares).

Precise version. With labels $y \in \{-1, +1\}$ and margin $m = y\,g(x)$, each method minimizes a (possibly regularized) average of a **surrogate loss** $\varphi(m)$ that stands in for the non-convex zero-one loss $\mathbb{1}[m \le 0]$ (Figure 4):

Method	Loss $\varphi(m)$	Behaviour
Perceptron	$\max(0, -m)$	zero for any correct point; scale-free, so $w = 0$ is a trivial minimizer
Hinge (SVM)	$\max(0, 1 - m)$	zero beyond the margin; sparse solutions (support vectors)
Logistic	$\ln(1 + e^{-m})$	smooth, never exactly zero; gives probabilities
Squared	$(1 - m)^2 = (y - g(x))^2$	penalizes $m > 1$: "too correct" points pull the boundary

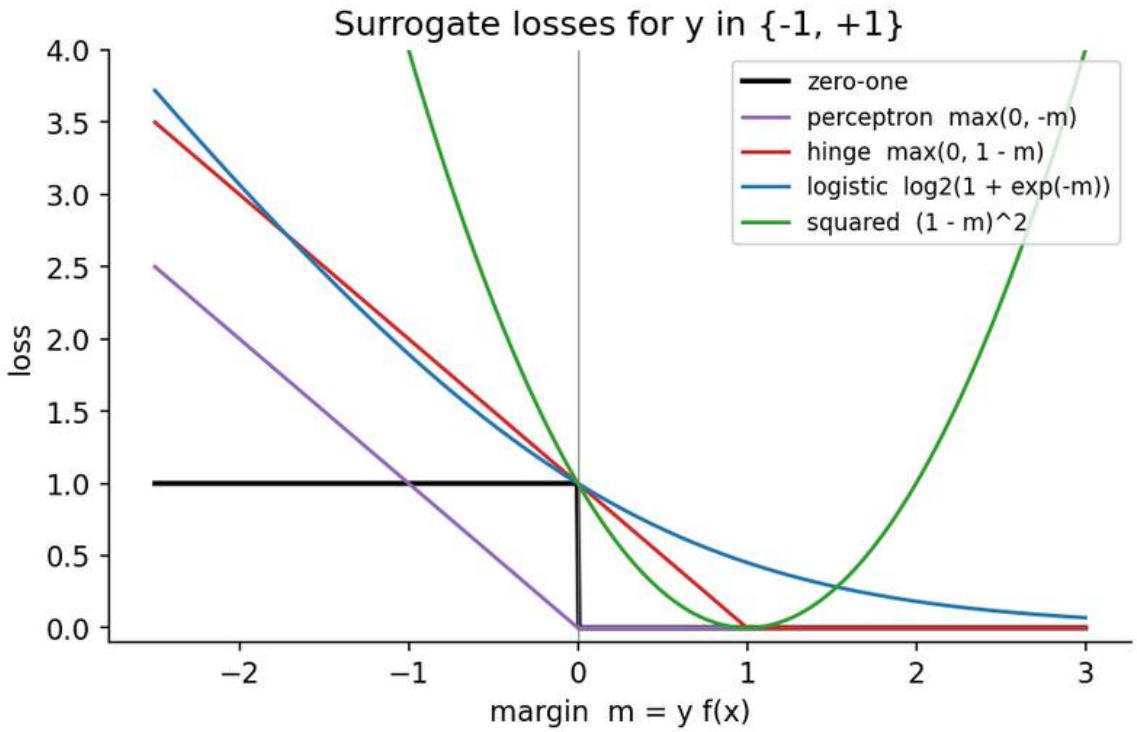


Figure 4 — Surrogate losses as functions of the margin $m = y\,f(x)$. The logistic loss is scaled by $1/\ln 2$ so that it passes through $(0, 1)$. Only the squared loss increases for $m > 1$, which explains Figure 1.

Which surrogates lead to the Bayes-optimal classifier when there is enough data? Bartlett, Jordan and McAuliffe [12] answered this precisely: a convex surrogate is **classification-calibrated** (minimizing its expected value yields the Bayes decision $\operatorname{sign}(2\eta(x) - 1)$, with $\eta(x) = P(y = 1 \mid x)$) if and only if it is differentiable at 0 with $\varphi'(0) < 0$. Hinge, logistic, squared and exponential losses all qualify; the perceptron loss does not. They also relate the excess surrogate risk to the excess classification risk. Rosasco et al. [13] compared the convergence rates implied by different convex losses and concluded that, for classification, the hinge loss converges about as fast as the logistic loss and much faster than the square loss. The losses still differ in *what* they estimate: minimizing the expected logistic loss pointwise gives $g(x) = \ln \frac{\eta(x)}{1-\eta(x)}$, so its scores are log-odds (probabilities after a sigmoid), whereas the pointwise minimizer of the expected hinge loss is $\operatorname{sign}(2\eta(x) - 1)$,

which carries no probability information. That is why SVM scores need separate calibration.

Code: compare linear classifiers on digits

The built-in digits dataset (8×8 images, 10 classes) requires no download. This is a single reproducible hold-out demonstration, not a benchmark ranking. Every scaler is fitted inside a pipeline, on training data only.

```
from sklearn.datasets import load_digits
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.linear_model import LogisticRegression, Perceptron, RidgeClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

X, y = load_digits(return_X_y=True)
Xtr, Xte, ytr, yte = train_test_split(
    X, y, test_size=0.25, stratify=y, random_state=7)
models = {
    "least squares (ridge)": make_pipeline(StandardScaler(), RidgeClassifier(alpha=1.0)),
    "perceptron": make_pipeline(StandardScaler(),
                                Perceptron(max_iter=1500, random_state=7)),
    "logistic": make_pipeline(StandardScaler(),
                              LogisticRegression(C=1, max_iter=1500)),
    "linear SVM": make_pipeline(StandardScaler(),
                                LinearSVC(C=0.1, max_iter=20000, random_state=7)),
    "Gaussian LDA": LinearDiscriminantAnalysis(solver="lsqr", shrinkage="auto"),
}
for name, model in models.items():
    model.fit(Xtr, ytr)
    print(f"{name:22s} test accuracy: {accuracy_score(yte, model.predict(Xte)):.4f}")
```

On this split, logistic regression reaches 97.1%, the linear SVM 96.4%, the perceptron 95.1%, and ridge-regularized least squares and shrinkage LDA 94.7% each. All five are linear machines on the same 64 pixels; the differences come from the losses and regularization alone. Changing regularization after seeing these test results would turn this test set into a validation set. A real model-selection study needs a separate validation split or cross-validation inside the training partition (PR 09).

Modern view

Linear classifiers have not been replaced by deep learning; they have moved to the top of it. Three lines of recent work explain why they still matter, and the classic survey literature remains useful for the methods themselves.

Surveys and solvers. Burges's tutorial [11] remains the most readable derivation of SVMs, from VC-dimension motivation through the dual and kernels. Cervantes et al. [17] survey SVM classification as of 2020: variants, applications, and challenges such as training cost on large datasets, class imbalance and model selection. On the computational side, LIBLINEAR [14] made linear SVMs and logistic regression with millions of sparse features routine, LIBSVM [16] is the reference kernel-SVM implementation behind scikit-learn's `SVC`, and Pegasos [15] showed that stochastic sub-gradient descent on the SVM primal reaches a given accuracy with a run time that, for linear kernels, does not depend directly on the training-set size, the same idea as training neural networks with SGD.

Linear probes as the yardstick of representations. Alain and Bengio [21] proposed training independent linear classifiers ("probes") on every layer of a network, and observed that linear separability increases monotonically with depth. Kornblith, Shlens and Le [22] compared 16 networks on 12 datasets and found that, with features used as fixed extractors under a logistic-regression classifier, ImageNet accuracy and transfer accuracy are strongly correlated, though some regularizers that slightly improve ImageNet accuracy produce much worse penultimate-layer features. CLIP [23] compares its zero-shot classifier with logistic-regression probes on frozen features, and DINOv2 [24] evaluates its self-supervised features largely with linear classifiers on frozen backbones. Kumar et al. [25] showed that full fine-tuning can distort good pretrained features and underperform a linear probe out of distribution, and proposed fitting a linear probe first and then fine-tuning (LP-FT). In other words: a well-regularized logistic regression on top of a frozen foundation model is a strong baseline, and often the right first experiment.

Implicit bias: logistic regression secretly finds the SVM. Soudry et al. [19] proved that gradient descent on unregularized logistic (or cross-entropy) loss over linearly separable data converges *in direction* to the hard-margin SVM solution, extending the result to other losses with exponential tails, to multiclass problems and to a restricted deep-network setting. Convergence is only logarithmic, which helps explain why continuing to train after zero training error can keep improving the margin. The gradient-descent code above shows the effect in two dimensions.

Neural collapse: Fisher's picture at the end of training. Papyan, Han and Donoho [20] observed, across three architectures and seven datasets, that in the terminal phase of training (after zero training error) the last-layer features of each class collapse to their class mean, the class means arrange themselves into a maximally separated, symmetric configuration (a simplex equiangular tight frame), the classifier weights align with the class means, and the decision rule becomes the nearest class mean. In this chapter's language, the within-class scatter S_W shrinks towards zero, Fisher's criterion grows without bound, and the network's last layer becomes the minimum-distance classifier of PR 01.

Open problems. When should one trust a linear probe as a measure of representation quality, given that probe results depend on regularization and data size? How do implicit-bias results extend to realistic, non-separable, data-augmented training? And when do probabilities from linear heads stay calibrated under distribution shift? These questions link the oldest classifier in this track to current research.

Key takeaways

- A linear discriminant $g(x) = w^\top x + b$ is a hyperplane with normal w; $g(x)/\|w\|$ is the signed distance. Use one discriminant per class (a linear machine) for multiclass problems.
- Least squares and ridge have closed forms and a Bayesian MAP interpretation, but least squares is a poor classifier: it punishes points that are "too correct" and can mask classes.
- Fisher's direction $w \propto S_W^{-1}(m_1 - m_0)$ maximizes between-class over within-class spread; it equals the Gaussian LDA direction, which adds a threshold from means and priors.
- The perceptron converges in at most $(R/\gamma)^2$ updates on separable data, but stops at an arbitrary separator and cycles on overlapping data.
- Logistic regression fits the log-odds by convex cross-entropy minimization (Newton/IRLS); without regularization its weights diverge on separable data, in the direction of the max-margin solution.
- The hard-margin SVM maximizes the margin $2/\|w\|$; its dual shows that only support vectors matter and that data enter only through inner products, which opens the door to kernels (PR 04). The soft margin is ridge-regularized hinge loss.
- All these methods are one linear score with different surrogate losses; hinge, logistic and squared losses are classification-calibrated, but only the logistic loss estimates probabilities.

Exercises

1. **Linear in the parameters.** Show that fitting $g(x) = w_0 + w_1 x + w_2 x^2$ by least squares is a linear regression problem, and describe the decision regions in x if the same features are used for a two-class linear classifier.

   Hint

   Build rows $(1, x, x^2)$ of a design matrix; the model is linear in (w_0, w_1, w_2). A quadratic in x changes sign at most twice, so the region for one class can be an interval and the other class its complement (two half-lines).

2. **Units and ridge.** Two features measure the same distance, one in metres and one in millimetres. Explain what ridge regression does without feature scaling, and why standardizing fixes it.

   Hint

   The millimetre coefficient can be 1,000 times smaller while describing the same effect, so it is penalized far less; ridge will load almost all weight onto the millimetre feature. After standardization both coordinates are penalized on the same scale.

3. **Fisher by hand.** Two classes have means $m_0 = (0, 0)$, $m_1 = (2, 0)$ and within-class scatter $S_W = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$. Compute Fisher's direction and compare it with the mean difference. When would the two directions coincide?

Hint

$S_W^{-1} = \frac{1}{3}\begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix}$, so $w \propto S_W^{-1}(2,0)^\top = \frac{1}{3}(4,-2) \propto (2,-1)$, rotated away from the mean difference $(1,0)$ because the classes are stretched along $(1,1)$. The directions coincide exactly when $m_1 - m_0$ is an eigenvector of S_W, e.g. for $m_1 = (2,2)$.

4. **A perceptron that never stops.** Construct a one-dimensional dataset on which the perceptron updates forever, and explain which assumption of the convergence theorem fails.

Hint

Put two points at the same coordinate with opposite labels (or any overlapping classes). No u with $\gamma > 0$ exists, so the bound does not apply; each update that fixes one point breaks the other.

5. **SVM by hand.** Find the hard-margin SVM for the two points $x_1 = (1,1)$ with $y_1 = +1$ and $x_2 = (-1,-1)$ with $y_2 = -1$. Give w, b, the dual variables and the margin width, and check the KKT conditions.

Hint

By symmetry $b = 0$ and $w = c(1,1)$. The constraint $y_1 w^\top x_1 = 1$ gives $2c = 1$, so $w = (\frac{1}{2}, \frac{1}{2})$ and the width is $2/\|w\| = 2\sqrt{2}$, the distance between the points. From $w = a_1 x_1 - a_2 x_2$ and $a_1 = a_2$, $a_1 = a_2 = \frac{1}{4}$. Both points are support vectors.

6. **Logistic versus hinge.** Using the digits pipeline, fit `LogisticRegression` and `LinearSVC` for several values of `C` with 5-fold cross-validation on the training set only. Compare their best validation accuracies, then compare their scores: which one gives meaningful probabilities, and how would you obtain probabilities from the other?

Hint

Accuracies are typically close, as the loss analysis predicts. Only logistic regression's `predict_proba` estimates posteriors; for the SVM, fit a calibration map (Platt scaling or isotonic regression, e.g. `CalibratedClassifierCV`) on held-out data.

References

1. R. O. Duda, P. E. Hart and D. G. Stork, *Pattern Classification*, 2nd ed., Wiley, 2001 (Ch. 5, linear discriminant functions). publisher page
2. C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006 (Sections 3.1–3.3, 4.1–4.3 and 7.1). official page and free PDF
3. S. Theodoridis and K. Koutroumbas, *Pattern Recognition*, 4th ed., Academic Press, 2008 (Ch. 3, linear classifiers). publisher page

4. T. Hastie, R. Tibshirani and J. Friedman, *The Elements of Statistical Learning*, 2nd ed., Springer, 2009 (Ch. 3–4 and 12). official page
5. R. A. Fisher, “The use of multiple measurements in taxonomic problems,” *Annals of Eugenics*, vol. 7, no. 2, pp. 179–188, 1936. doi
6. F. Rosenblatt, “The perceptron: a probabilistic model for information storage and organization in the brain,” *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958. doi
7. D. R. Cox, “The regression analysis of binary sequences,” *Journal of the Royal Statistical Society, Series B*, vol. 20, no. 2, pp. 215–232, 1958. doi
8. A. E. Hoerl and R. W. Kennard, “Ridge regression: biased estimation for nonorthogonal problems,” *Technometrics*, vol. 12, no. 1, pp. 55–67, 1970. doi
9. B. E. Boser, I. M. Guyon and V. N. Vapnik, “A training algorithm for optimal margin classifiers,” in *Proc. 5th Annual Workshop on Computational Learning Theory* (*COLT*), pp. 144–152, 1992. doi
10. C. Cortes and V. Vapnik, “Support-vector networks,” *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995. doi
11. C. J. C. Burges, “A tutorial on support vector machines for pattern recognition,” *Data Mining and Knowledge Discovery*, vol. 2, no. 2, pp. 121–167, 1998. doi
12. P. L. Bartlett, M. I. Jordan and J. D. McAuliffe, “Convexity, classification, and risk bounds,” *Journal of the American Statistical Association*, vol. 101, no. 473, pp. 138–156, 2006. doi
13. L. Rosasco, E. De Vito, A. Caponnetto, M. Piana and A. Verri, “Are loss functions all the same?,” *Neural Computation*, vol. 16, no. 5, pp. 1063–1076, 2004. doi
14. R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang and C.-J. Lin, “LIBLINEAR: a library for large linear classification,” *Journal of Machine Learning Research*, vol. 9, pp. 1871–1874, 2008. paper
15. S. Shalev-Shwartz, Y. Singer, N. Srebro and A. Cotter, “Pegasos: primal estimated sub-gradient solver for SVM,” *Mathematical Programming*, vol. 127, no. 1, pp. 3–30, 2011. doi
16. C.-C. Chang and C.-J. Lin, “LIBSVM: a library for support vector machines,” *ACM Transactions on Intelligent Systems and Technology*, vol. 2, no. 3, article 27, 2011. doi
17. J. Cervantes, F. Garcia-Lamont, L. Rodríguez-Mazahua and A. Lopez, “A comprehensive survey on support vector machine classification: applications, challenges and trends,” *Neurocomputing*, vol. 408, pp. 189–215, 2020. doi
18. A. Y. Ng and M. I. Jordan, “On discriminative vs. generative classifiers: a comparison of logistic regression and naive Bayes,” in *Advances in Neural Information Processing Systems* (*NIPS*), 2001. paper
19. D. Soudry, E. Hoffer, M. S. Nacson, S. Gunasekar and N. Srebro, “The implicit bias of gradient descent on separable data,” *Journal of Machine Learning Research*, vol. 19, no. 70, pp. 1–57, 2018. paper
20. V. Papyan, X. Y. Han and D. L. Donoho, “Prevalence of neural collapse during the terminal phase of deep learning training,” *Proceedings of the National Academy of Sciences*, vol. 117, no. 40, pp. 24652–24663, 2020. doi
21. G. Alain and Y. Bengio, “Understanding intermediate layers using linear classifier probes,” arXiv:1610.01644, 2016. arXiv
22. S. Kornblith, J. Shlens and Q. V. Le, “Do better ImageNet models transfer better?,” in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition* (*CVPR*), 2019. paper
23. A. Radford et al., “Learning transferable visual models from natural language supervision,” in *Proc. International Conference on Machine Learning* (*ICML*), PMLR 139, 2021. paper

24. M. Oquab et al., “DINOv2: learning robust visual features without supervision,” *Transactions on Machine Learning Research*, 2024. arXiv
25. A. Kumar, A. Raghunathan, R. Jones, T. Ma and P. Liang, “Fine-tuning can distort pretrained features and underperform out-of-distribution,” in *Proc. International Conference on Learning Representations* (*ICLR*), 2022. arXiv

PR 04 • Neighbors, Kernels and Nonlinear Classifiers

From nearest neighbors and the kernel trick to soft-margin SVMs, RBF networks, decision trees and multilayer perceptrons: how classifiers bend the decision boundary, and what each bend costs.

Online edition: https://blog.shyandram.dev/en/learn/pattern-recognition/kernels-nonlinear-classifiers/

What you'll learn

- How nearest-neighbor rules produce nonlinear decision regions without fitting any parameters, and why the distance metric matters as much as k.
- What the kernel trick is, what makes a function a valid (Mercer) kernel, and how to check it on a finite Gram matrix.
- The soft-margin support vector machine: its primal and dual problems, support vectors, and how the hyperparameters C and γ shape the boundary.
- How Gaussian processes and relevance vector machines reuse kernels with a probabilistic interpretation.
- Three other routes to nonlinearity: radial basis function (RBF) networks, decision trees grown with impurity measures (CART), and multilayer perceptrons.
- How to compare these families fairly, and where they stand in 2026.

The big picture

PR 03 built classifiers whose decision boundary is a straight line, a plane or a hyperplane. Many real problems are not like that. Points inside a ring cannot be separated from points outside it by any straight line. Handwritten "1"s and "7"s overlap in pixel space along curved, twisted regions. We need boundaries that bend.

There are only a few basic ways to make them bend:

1. **Compare with stored examples.** Label a new point by looking at the most similar training points (nearest neighbors).
2. **Change the representation, then stay linear.** Map each input x to new features $\phi(x)$ in which a hyperplane does work. Kernels do this implicitly; RBF networks and multilayer perceptrons do it explicitly, with learned or fixed basis functions.
3. **Cut the space into pieces.** A decision tree asks a sequence of yes/no questions, and each answer path ends in a small box with its own label.

Explain like I'm five

> Imagine sorting leaves you find in a park. You could hold each new leaf next to your labeled collection and copy the label of the leaves it looks most like (neighbors). You could look at the leaf through special glasses that make the differences jump out, so a simple rule works again (kernels and features). Or you could play twenty questions: "Is it wider than my thumb? Does it have jagged edges?" until only one kind of leaf is left (decision trees).

One warning applies to every method in this chapter. A nonlinear classifier can only use information that the measurements contain. If two classes look identical in the features you recorded, no amount of boundary-bending will separate them. Feature design is the subject of PR 05. The textbook background for this chapter is Theodoridis and Koutroumbas, Chapters 2 and 4 [1], and Bishop, Sections 2.5.2, 5.1–5.3, 6.1–6.4, 7.1–7.2 and 14.4 [2].

Nearest-neighbor classification

In plain words. To classify a new point, find the k training points closest to it and let them vote. Nothing is "learned" beyond storing the data, yet the resulting boundary can follow any shape the data have.

The rule and its guarantee

Given training pairs (x_n, y_n), $n = 1, \dots, N$, and a query x, let $\mathcal{N}_k(x)$ be the indices of the k training points nearest to x under some distance d. The k-nearest-neighbor (k-NN) rule predicts

$$\hat{y}(x) = \arg\max_{c} \sum_{n \in \mathcal{N}_k(x)} w_n \, \mathbf{1}[y_n = c],$$

where c runs over the classes, $\mathbf{1}[\cdot]$ is 1 when its argument is true and 0 otherwise, and w_n is a vote weight: $w_n = 1$ for plain majority voting, or a decreasing function of $d(x, x_n)$ for distance-weighted voting.

The vote fractions are a crude estimate of the class posteriors $P(c \mid x)$: k-NN is a nonparametric density method in disguise (PR 02), with the neighborhood size set by k rather than by a fixed bandwidth [2]. Cover and Hart proved the classic result for the 1-NN rule: as the number of training samples grows without bound, its error rate is at most twice the Bayes error rate [3]. This is an asymptotic statement. With finite data, especially in high dimension, k-NN can be far from that bound.

The metric is a modeling decision

Euclidean distance treats a one-unit difference in every coordinate as equally important. If one feature is measured in millimeters and another in kilometers, the first will dominate every neighborhood. Standardizing features, learning a Mahalanobis metric $d(x, z)^2 = (x - z)^\top M (x - z)$ with a positive semidefinite matrix M, or comparing normalized embeddings by cosine similarity all change who counts as a neighbor.

For vectors u and v normalized to unit length,

$$\|u - v\|^2 = \|u\|^2 + \|v\|^2 - 2u^\top v = 2 - 2u^\top v.$$

So the nearest neighbor by Euclidean distance is exactly the neighbor with the *largest* cosine similarity: the two criteria rank candidates in reverse order of each other and pick the same neighbors. This is why nearest-neighbor search over normalized deep features (ML 06) can use either.

Interactive demonstration (online): open the online edition of this tutorial to use this demonstration.Open demonstration

Choosing k, and the cost of prediction

A small k (such as 1) gives a jagged boundary that follows every noisy training point: low bias, high variance. A large k averages over a wide region and smooths the boundary, but it can wash out a small class entirely. Choose k, the metric and the scaling by cross-validation on training data only.

k-NN shifts all the cost to prediction time. A naive query compares x with all N training points, which costs $O(ND)$ for D-dimensional inputs. Tree-based indices and approximate nearest-neighbor search reduce this in practice. In high dimension, a further problem appears: when many coordinates carry irrelevant variation, distances between random points concentrate around a common value, and "nearest" stops meaning "most similar." The right panel of Figure 3 later in this chapter shows a 7-NN boundary on a two-dimensional toy problem, where none of these problems arise.

Kernels: the kernel trick and Mercer's condition

In plain words. Many linear algorithms only ever use inputs through inner products $x^\top z$. If we replace every inner product by a "similarity function" $k(x, z)$ that equals an inner product in some richer feature space, we get a nonlinear algorithm for the price of a linear one, without ever computing the rich features.

Lifting the data

Take the ring data on the left of Figure 1. No line separates the inner disc from the outer ring. Add a third feature, the squared radius $r^2 = x_1^2 + x_2^2$. In the new three-dimensional space the inner points sit low and the outer points sit high, and a flat plane separates them perfectly. A linear classifier in the lifted space is a circle in the original space.

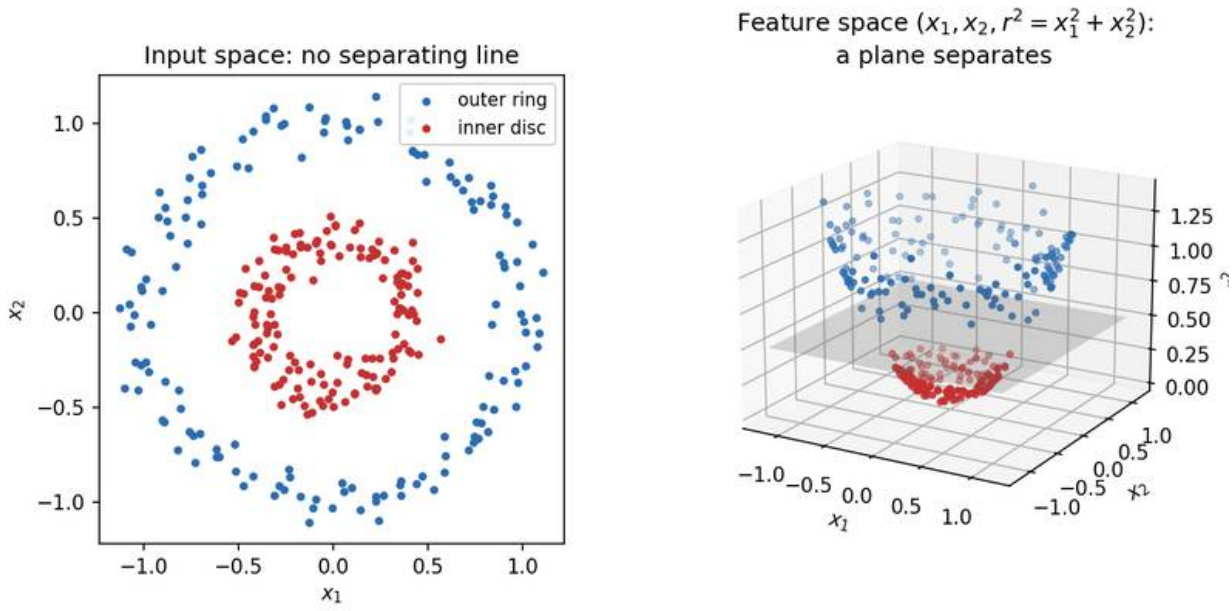


Figure 1 — Lifting to a feature space. Adding the feature $r^2 = x_1^2 + x_2^2$ turns a circular boundary into a flat plane.

Explicit lifting gets expensive quickly. All monomials of degree up to m in D variables number $\binom{D+m}{m}$; for a 16×16 image ($D = 256$) and $m = 3$ that is about 2.9 million features. Kernels avoid building them.

The kernel trick

A **kernel** is a function $k(x, z)$ for which there exists a feature map ϕ into some inner-product space with

$$k(x, z) = \phi(x)^{\top} \phi(z).$$

Here x and z are inputs, $\phi(x)$ is the (possibly very long, even infinite) feature vector of x, and the right-hand side is their inner product. For example, in two dimensions, with $x = (a, b)$,

$$(x^{\top} z + 1)^2 = \phi(x)^{\top} \phi(z), \qquad \phi(x) = (1,\ \sqrt{2}a,\ \sqrt{2}b,\ a^2,\ b^2,\ \sqrt{2}ab).$$

Evaluating the left side costs one inner product and one square. The right side needs six features per point. For degree m in D dimensions, the left side still costs $O(D)$.

The **kernel trick** is the observation that any algorithm written entirely in terms of inner products between data points can be "kernelized" by replacing $x_i^{\top} x_j$ with $k(x_i, x_j)$ [7]. The SVM dual, kernel ridge regression, kernel PCA (PR 05) and Gaussian processes are all of this kind. The result is collected in the **Gram matrix** $K \in \mathbb{R}^{N \times N}$, $K_{ij} = k(x_i, x_j)$.

The code below checks the identity numerically, and then shows a popular "similarity" that is *not* a valid kernel.

```
import numpy as np

rng = np.random.default_rng(0)
x, z = rng.normal(size=2), rng.normal(size=2)

def phi(v):  # explicit feature map of (v.w + 1)^2 in 2-D
    a, b = v
    s = np.sqrt(2)
    return np.array([1, s * a, s * b, a * a, b * b, s * a * b])

lhs = (x @ z + 1) ** 2
rhs = phi(x) @ phi(z)
print("kernel:", round(lhs, 6), " explicit:", round(rhs, 6))
assert np.isclose(lhs, rhs)

X = rng.normal(size=(40, 2))
K_sig = np.tanh(2.0 * X @ X.T - 1.0)        # "sigmoid kernel"
print("sigmoid Gram min eigenvalue:", np.linalg.eigvalsh(K_sig).min().round(3))
```

The two numbers agree. The tanh "sigmoid kernel" Gram matrix has a clearly negative eigenvalue (about -15 here), so for these parameters no feature map ϕ can reproduce it.

Which functions are kernels? Mercer's condition

A function k is a valid kernel exactly when it is **symmetric**, $k(x, z) = k(z, x)$, and **positive semidefinite**: for every finite set of points $x_1, \dots, x_N$ and every real vector $a \in \mathbb{R}^N$,

$$a^\top K a = \sum_{i=1}^{N} \sum_{j=1}^{N} a_i a_j \, k(x_i, x_j) \;\geq\; 0.$$

Equivalently, every Gram matrix has nonnegative eigenvalues. Symmetry alone is not enough, as the sigmoid example shows.

Mercer's theorem [4] gives the continuous version. For a continuous symmetric k on a compact domain $\mathcal{X}$, positive semidefiniteness in the integral sense,

$$\int_{\mathcal{X}} \int_{\mathcal{X}} g(x)\, k(x, z)\, g(z)\, dx\, dz \geq 0 \quad \text{for all square-integrable } g,$$

guarantees an expansion $k(x, z) = \sum_{i=1}^{\infty} \lambda_i \, e_i(x) \, e_i(z)$ with eigenvalues $\lambda_i \geq 0$ and orthonormal eigenfunctions e_i. The feature map is then $\phi(x) = (\sqrt{\lambda_1} e_1(x), \sqrt{\lambda_2} e_2(x), \dots)$. In modern terms, every positive semidefinite kernel defines a **reproducing kernel Hilbert space** (RKHS) of functions, and the representer theorem says that regularized fits in that space are finite combinations $f(x) = \sum_n \alpha_n k(x_n, x)$ [7].

Valid kernels can be built from valid kernels. If k_1 and k_2 are kernels, so are $k_1 + k_2$, $c\,k_1$ for $c > 0$, $k_1 k_2$, $\exp(k_1)$, and $f(x) k_1(x, z) f(z)$ for any function f [2]. The common kernels follow:

Kernel	Formula	Parameters	Notes
Linear	$x^\top z$	none	Recovers the linear model
Polynomial	$(\gamma x^\top z + r)^m$	$\gamma > 0$, $r > 0$, integer $m \geq 1$	All monomials up to degree m ($r = 0$: degree m only)
Gaussian RBF	$\exp(-\gamma\|x - z\|^2)$	$\gamma > 0$	Infinite-dimensional feature space; local similarity
Laplacian	$\exp(-\gamma\|x - z\|_1)$	$\gamma > 0$	Rougher functions than the Gaussian
Sigmoid	$\tanh(\gamma x^\top z + r)$	γ, r	Not positive semidefinite in general

The Gaussian RBF kernel is the default nonlinear choice. Its parameter γ (sometimes written $1/(2\sigma^2)$ with a bandwidth σ) sets how far similarity reaches. A large γ makes each training point influence only a tiny neighborhood; a small γ makes the kernel nearly constant over the data, so the model approaches a very smooth, almost linear function. Because γ multiplies squared distances, it is tied to the scale of the features: scaling is part of the model. Standardization is a useful default for mixed units, but features already sharing meaningful units may work better without it, as the PR 10 capstone illustrates. Compare these choices inside validation, and fit any scaler only on each training fold [27].

Kernel ridge regression: the trick in one line

Ridge regression with features ϕ minimizes $\sum_n (t_n - w^\top \phi(x_n))^2 + \lambda\|w\|^2$. Its solution can be written as $w = \sum_n \alpha_n \phi(x_n)$, which leads to

$$\alpha = (K + \lambda I)^{-1} t, \qquad f(x) = k_x^\top \alpha,$$

where $t = (t_1, \ldots, t_N)^\top$ are the targets, $\lambda > 0$ is the regularization strength, I is the $N \times N$ identity, and $k_x = (k(x_1, x), \ldots, k(x_N, x))^\top$ holds the similarities of a query x to the training points. The D-dimensional feature-space solve has become an $N \times N$ sample-space solve: attractive when features are many and samples few, expensive ($O(N^3)$ time, $O(N^2)$ memory) when N is large.

```
import numpy as np
from sklearn.kernel_ridge import KernelRidge
from sklearn.metrics.pairwise import rbf_kernel

rng = np.random.default_rng(1)
X = rng.uniform(-3, 3, size=(60, 1))
t = np.sin(X[:, 0]) + 0.1 * rng.normal(size=60)
lam, gamma = 0.1, 0.5

K = rbf_kernel(X, X, gamma=gamma)
alpha = np.linalg.solve(K + lam * np.eye(len(X)), t)
Xq = np.linspace(-3, 3, 5)[:, None]
f_ours = rbf_kernel(Xq, X, gamma=gamma) @ alpha
f_skl = KernelRidge(alpha=lam, kernel="rbf", gamma=gamma).fit(X, t).predict(Xq)
```

```
print(np.round(f_ours, 4))
assert np.allclose(f_ours, f_skl)
```

The hand-written solve and scikit-learn's `KernelRidge` agree, and the predictions follow $\sin x$ at the five query points.

Support vector machines

In plain words. Among all boundaries that separate the classes, pick the one with the widest empty "street" between them. Only the points on the edges of the street, or inside it, matter; they are the support vectors. Allowing a few points into the street, at a price C, makes the method work on overlapping data. Replacing inner products by a kernel makes the street curve.

Hard and soft margins

Use labels $y_n \in \{-1, +1\}$ and a score $f(x) = w^\top \phi(x) + b$, with weight vector w and bias b; predict the sign of f. If the data are separable, scale w and b so that the closest points satisfy $y_n f(x_n) = 1$. The distance between the two lines $f = \pm 1$, the **margin**, is then $2/\|w\|$. Maximizing it means minimizing $\frac{1}{2}\|w\|^2$ subject to $y_n f(x_n) \geq 1$ for all n: the hard-margin SVM of Boser, Guyon and Vapnik [5].

Real data overlap, so Cortes and Vapnik introduced slack variables $\xi_n \geq 0$ that measure how far each point violates the margin [6]. The **soft-margin SVM** solves

$$\min_{w,b,\xi} \ \frac{1}{2}\|w\|^2 + C \sum_{n=1}^{N} \xi_n \quad \text{subject to} \quad y_n f(x_n) \geq 1 - \xi_n, \ \ \xi_n \geq 0.$$

A point with $\xi_n = 0$ is on the correct side with margin at least one. At an optimum, $0 < \xi_n < 1$ means a point is inside the street but correctly classified; $\xi_n = 1$ lies on the decision boundary, and $\xi_n > 1$ is misclassified. The constant $C > 0$ sets the price of violations. Eliminating ξ gives the equivalent unconstrained form

$$\min_{w,b} \ \frac{1}{2}\|w\|^2 + C \sum_{n} \max\left(0,\ 1 - y_n f(x_n)\right),$$

regularized **hinge loss**. A point with $y_n f(x_n) \geq 1$ contributes no loss at all; this is the source of the SVM's sparsity. (Some texts average the losses and put the constant on the regularizer instead, writing $\frac{\lambda}{2}\|w\|^2 + \frac{1}{N}\sum_n \text{hinge}$. The two forms match when $C = 1/(\lambda N)$.)

The dual problem and support vectors

Introducing a Lagrange multiplier $\alpha_n \geq 0$ for each margin constraint and eliminating w, b and ξ gives the **dual problem** [2], [6]:

$$\max_{\alpha} \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \alpha_n \alpha_m y_n y_m \, k(x_n, x_m) \quad \text{subject to} \quad 0 \le \alpha_n \le C, \ \sum_{n=1}^{N} \alpha_n y_n = 0.$$

It is a convex quadratic program in the N multipliers α_n, and the data appear *only* through kernel values $k(x_n, x_m)$. That is where the kernel trick enters. The optimal weight vector is $w = \sum_n \alpha_n y_n \phi(x_n)$, so predictions need only kernels too:

$$f(x) = \sum_{n=1}^{N} \alpha_n y_n \, k(x_n, x) + b.$$

The Karush–Kuhn–Tucker (KKT) optimality conditions sort the training points into three groups:

- $\alpha_n = 0$: the point is outside the street ($y_n f(x_n) \ge 1$) and has no influence on f;
- $0 < \alpha_n < C$: the point lies exactly on the margin ($y_n f(x_n) = 1$); these points determine b;
- $\alpha_n = C$: the point is inside the street or misclassified ($y_n f(x_n) \le 1$).

Points with $\alpha_n > 0$ are the **support vectors**. Deleting any other point and retraining gives the same classifier. How many there are depends on the data: on a heavily overlapping problem, a large fraction of the training set can end up as support vectors, and prediction becomes as expensive as nearest-neighbor search.

How C and γ shape an RBF-SVM

With the Gaussian kernel, two numbers control everything. C sets how much the training errors cost relative to a wide margin; γ sets how local the kernel is. Figure 2 sweeps both on the same noisy data.

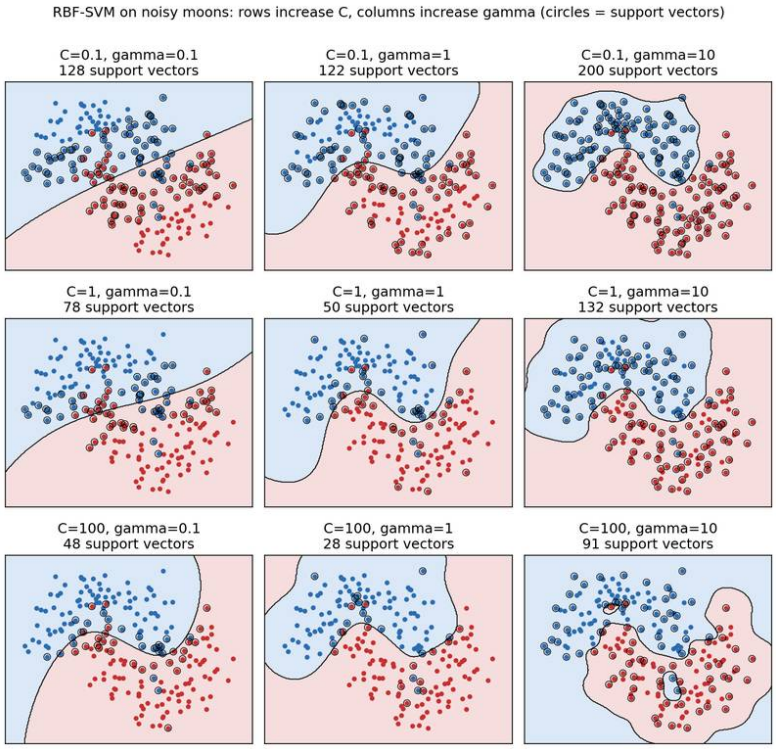


Figure 2 — RBF-SVM on noisy two-moons data (standardized). Small γ gives almost linear boundaries; large γ gives islands around individual points. Small C tolerates many margin violations (many support vectors); large C fits the training points harder. Generated by scripts/figures/pr_04_kernels_nonlinear_classifiers_extra.py.

Read the grid like this. Moving down (larger C) makes the boundary chase individual training points, and the number of support vectors drops because fewer points are allowed inside the margin. Moving right (larger γ) shrinks each point's sphere of influence; at $\gamma = 10$ the classifier draws little islands, which is overfitting. The good region is a diagonal band: when γ grows, a smaller C is usually needed. In practice one searches C and γ on a logarithmic grid with cross-validation.

```
import numpy as np
from sklearn.datasets import make_moons
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC

X, y = make_moons(n_samples=500, noise=0.18, random_state=2)
Xtr, Xte, ytr, yte = train_test_split(
    X, y, test_size=0.3, stratify=y, random_state=2)
models = {
    "linear SVM": SVC(kernel="linear", C=1),
    "RBF SVM": SVC(kernel="rbf", C=3, gamma=0.7),
```

```
    "7-neighbor": KNeighborsClassifier(n_neighbors=7),
}
for name, estimator in models.items():
    model = make_pipeline(StandardScaler(), estimator).fit(Xtr, ytr)
    print(name, "test accuracy:", round(model.score(Xte, yte), 4))

search = GridSearchCV(
    make_pipeline(StandardScaler(), SVC(kernel="rbf")),
    {"svc__C": [0.1, 1, 10, 100], "svc__gamma": [0.01, 0.1, 1, 10]},
    cv=5)
search.fit(Xtr, ytr)
best = search.best_estimator_
print("best:", search.best_params_,
      "CV acc:", round(search.best_score_, 4),
      "test acc:", round(best.score(Xte, yte), 4),
      "support vectors:", best[-1].n_support_.sum(), "of", len(Xtr))
```

On this split the linear SVM reaches 0.827 test accuracy, the hand-picked RBF-SVM and the 7-NN rule both 0.96. The grid search chooses $C = 1$, $\gamma = 1$ (cross-validated accuracy 0.971) with 59 of 350 training points as support vectors, and scores 0.953 on the test set. The tuned model is not "worse" than the hand-picked one in any meaningful sense: with 150 test points, one test point is 0.0067 of accuracy, and the two differ by one point. Treat single-split differences of this size as noise (PR 09). The scaler lives inside the pipeline so that it is refitted on each training fold.

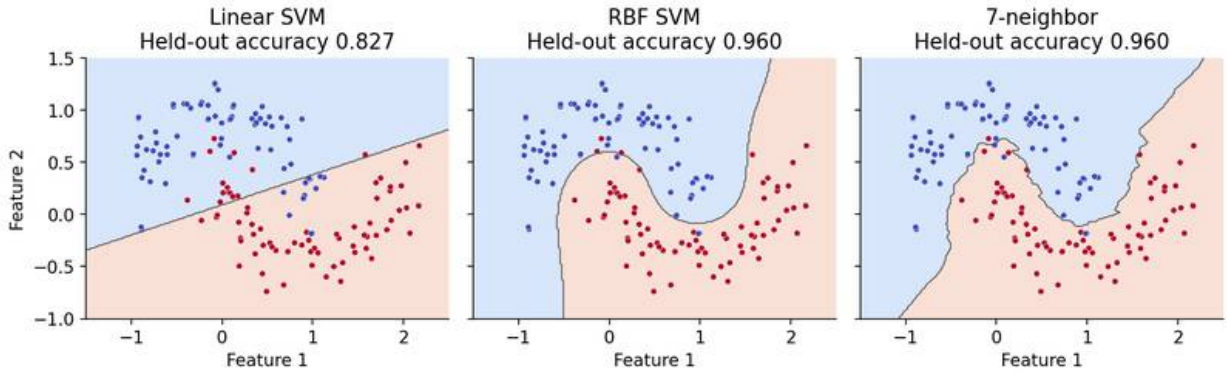


Figure 3 — Linear SVM, RBF-SVM and 7-NN on the same split as the code above. Scalers are fitted on training data only; dots and scores use the held-out partition. A synthetic example, not a general benchmark.

Scores are not probabilities

The SVM output $f(x)$ is a signed distance-like score, not $P(y = +1 \mid x)$. Its training objective never asks it to be calibrated. If probabilities are needed (for a reject option, PR 01, or for combining models), fit a separate monotone mapping from scores to probabilities on held-out data, and check it with a reliability diagram (PR 09). Multiclass SVMs are usually built from binary ones, either one-versus-rest or one-versus-one (scikit-learn's `SVC` uses one-versus-one).

Gaussian processes and relevance vectors

In plain words. A Gaussian process uses the same kernels, but instead of returning one best function it returns a whole distribution of plausible functions, so each prediction comes with an error bar.

A **Gaussian process** (GP) is a prior over functions f such that any finite set of values $(f(x_1), \dots, f(x_N))$ is jointly Gaussian with mean zero (typically) and covariance $K_{ij} = k(x_i, x_j)$ [8]. The kernel must therefore be a valid covariance, that is, positive semidefinite. For regression with Gaussian observation noise of variance σ^2, the posterior at a query x is Gaussian with

$$\mu(x) = k_x^\top (K + \sigma^2 I)^{-1} t, \qquad s^2(x) = k(x,x) - k_x^\top (K + \sigma^2 I)^{-1} k_x,$$

where $\mu(x)$ is the predictive mean and $s^2(x)$ the variance of the latent function. The mean is exactly kernel ridge regression with $\lambda = \sigma^2$. The variance is new: it is small near training inputs and grows back to the prior variance $k(x,x)$ far from them. Kernel hyperparameters (such as γ and σ^2) can be chosen by maximizing the marginal likelihood instead of cross-validation [2], [8].

Two cautions. The uncertainty is conditional on the chosen kernel; a confident GP with the wrong kernel can be confidently wrong. And GP *classification* is harder: a Bernoulli likelihood on top of a Gaussian latent function makes the posterior non-Gaussian, so Laplace, expectation propagation or variational approximations are needed (PR 08).

The **relevance vector machine** (RVM), introduced by Tipping [9] and presented in Bishop, Section 7.2 [2], uses the same functional form as the SVM, $f(x) = \sum_n w_n k(x_n, x) + b$, but places a separate zero-mean Gaussian prior on each weight w_n with its own precision hyperparameter. Maximizing the marginal likelihood drives many precisions to infinity, which switches the corresponding basis functions off. The surviving training points, the "relevance vectors," are typically far fewer than an SVM's support vectors, the outputs are probabilistic, and the basis functions do not need to satisfy Mercer's condition. The price is a non-convex training problem. "Sparse" thus has different meanings: few stored basis functions (SVM, RVM), few nonzero input weights (lasso), or sparse input data. Say which one you mean.

Radial basis function networks

In plain words. Put a few Gaussian "bumps" at representative places in the input space, measure how strongly a new point activates each bump, and feed those activations to a linear classifier. It is the lifting idea of Figure 1, but with a small, explicit set of features.

An RBF network computes

$$f(x) = \sum_{j=1}^{M} w_j \exp\!\left(-\gamma \|x - \mu_j\|^2\right) + w_0,$$

where $\mu_1, \dots, \mu_M$ are M **centers**, γ sets the width of each bump, w_j are output weights and w_0 is a bias. Moody and Darken proposed the classic two-stage training [10]: first place the centers without labels, for example by k-means clustering (PR 07); then, with centers fixed, the output weights

solve an ordinary linear least-squares or logistic regression problem. Park and Sandberg showed that RBF networks are universal approximators: with enough centers they can approximate any continuous function on a compact set arbitrarily well [11].

Compare this with the kernel machines above. A kernel SVM effectively places one bump on every support vector, chosen by the margin criterion. An RBF network uses $M \ll N$ bumps whose positions come from clustering (or from gradient descent if all parameters are trained jointly), which makes prediction cheap. Its weakness is that unsupervised center placement ignores the labels, so centers may be wasted where the classes do not meet.

```
import numpy as np
from sklearn.cluster import KMeans
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

X, y = make_moons(n_samples=500, noise=0.18, random_state=2)
Xtr, Xte, ytr, yte = train_test_split(
    X, y, test_size=0.3, stratify=y, random_state=2)
scaler = StandardScaler().fit(Xtr)
Xtr, Xte = scaler.transform(Xtr), scaler.transform(Xte)

M, gamma, lam = 12, 1.0, 1e-3
centers = KMeans(M, n_init=10, random_state=0).fit(Xtr).cluster_centers_

def design(X):  # N x (M+1): Gaussian bumps plus a bias column
    d2 = ((X[:, None, :] - centers[None, :, :]) ** 2).sum(-1)
    return np.c_[np.exp(-gamma * d2), np.ones(len(X))]

Phi = design(Xtr)
t = 2.0 * ytr - 1.0                      # targets in {-1, +1}
w = np.linalg.solve(Phi.T @ Phi + lam * np.eye(M + 1), Phi.T @ t)
pred = (design(Xte) @ w > 0).astype(int)
print("RBF network test accuracy:", round((pred == yte).mean(), 4))
```

Twelve bumps and a least-squares output layer give 0.953 test accuracy on the moons split, on par with the RBF-SVM that keeps dozens of support vectors. Figure 5 (left) shows the same network with a logistic output layer.

Decision trees

In plain words. A decision tree plays twenty questions with the data. Each question looks at one feature and asks "is it at most this threshold?" Each answer sends the point left or right, until it reaches a leaf that holds a label. The tree is grown greedily: at every step, pick the question that makes the two resulting groups as "pure" as possible.

Impurity measures

Let a node contain training points with class proportions $p_1, \dots, p_C$ (fraction of the node's points in each class c). An **impurity** function is zero when one class has all the points and largest when the classes are evenly mixed. The three standard choices are

$$\text{Gini:} \quad G = 1 - \sum_{c=1}^{C} p_c^2 = \sum_{c} p_c(1 - p_c),$$

$$\text{Entropy:} \quad H = -\sum_{c=1}^{C} p_c \log_2 p_c,$$

$$\text{Misclassification:} \quad E = 1 - \max_c p_c.$$

The Gini index is the probability that two points drawn at random (with replacement) from the node have different labels. Entropy is the information-theoretic uncertainty of the label, in bits. Misclassification error is the error rate if the node predicted its majority class. Figure 4 (left) plots the three for two classes.

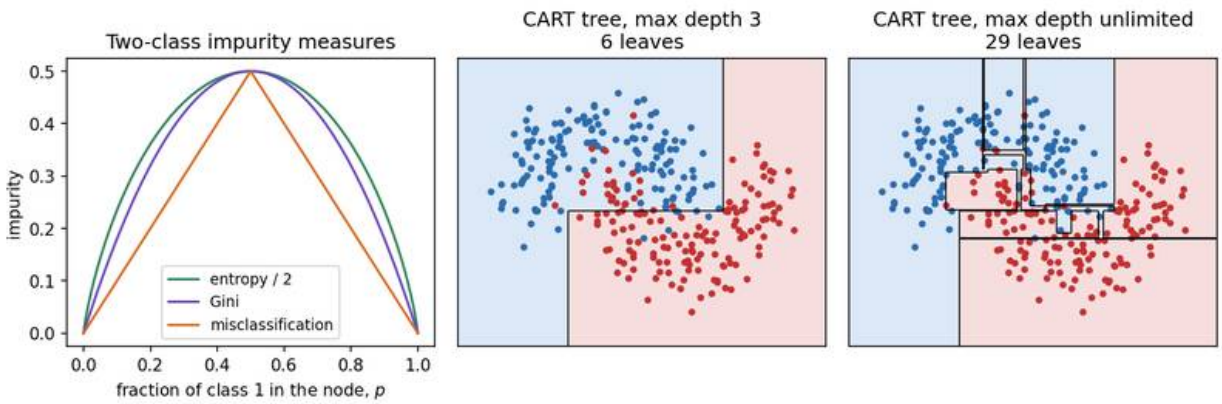


Figure 4 — Left: two-class impurity measures (entropy scaled by ½ to share the axis). Middle: a depth-3 CART tree gives a coarse, axis-aligned staircase. Right: an unlimited tree carves tiny boxes around noisy points, overfitting the training data.

Gini and entropy are strictly concave, so splitting a node never increases their weighted average. Misclassification error is piecewise linear and often fails to reward a split that makes the children purer without changing either child's majority label. That is why trees are grown with Gini or entropy and evaluated with error rate.

Growing a tree: the CART algorithm

The CART procedure of Breiman, Friedman, Olshen and Stone [12] grows a binary tree top-down. At a node with point set S, it considers every feature j and every threshold τ between consecutive observed values, splitting S into $S_L = \{x \in S : x_j \le \tau\}$ and $S_R = S \setminus S_L$. It chooses the split with the largest impurity decrease

$$\Delta i(j, \tau) = i(S) - \frac{|S_L|}{|S|} i(S_L) - \frac{|S_R|}{|S|} i(S_R),$$

where $i(\cdot)$ is the chosen impurity and $|S|$ counts the points in a set. It then recurses on both children until a stopping rule fires (pure node, minimum node size, maximum depth). Quinlan's ID3 [13] is a close relative that uses entropy, calls Δi the **information gain**, and makes multiway splits on categorical features.

The code finds the best root split by brute force and checks it against scikit-learn, whose `DecisionTreeClassifier` implements an optimized version of CART.

```
import numpy as np
from sklearn.datasets import make_moons
from sklearn.tree import DecisionTreeClassifier

def gini(labels):
    p = np.bincount(labels, minlength=2) / len(labels)
    return 1.0 - (p ** 2).sum()

def best_split(X, y):
    best = (np.inf, None, None)
    for j in range(X.shape[1]):
        values = np.unique(X[:, j])
        for thr in (values[:-1] + values[1:]) / 2:
            left = X[:, j] <= thr
            n_l, n_r = left.sum(), (~left).sum()
            score = (n_l * gini(y[left]) + n_r * gini(y[~left])) / len(y)
            if score < best[0]:
                best = (score, j, thr)
    return best

X, y = make_moons(n_samples=300, noise=0.25, random_state=1)
score, j, thr = best_split(X, y)
root = DecisionTreeClassifier(max_depth=1).fit(X, y).tree_
print(f"ours: feature {j}, threshold {thr:.4f}, weighted Gini {score:.4f}")
print(f"sklearn: feature {root.feature[0]}, threshold {root.threshold[0]:.4f}")
assert j == root.feature[0] and np.isclose(thr, root.threshold[0])
```

Both choose feature 1 with threshold 0.0547, reducing the impurity from 0.5 at the root to a weighted 0.287.

Pruning and the character of trees

A tree grown until every leaf is pure memorizes the training set (Figure 4, right). CART's answer is **cost-complexity pruning** [12]: grow a large tree T_0, then for each subtree T obtained by collapsing internal nodes, measure

$$R_\alpha(T) = R(T) + \alpha\,|\widetilde{T}|,$$

where $R(T)$ is the training misclassification rate, $|\widetilde{T}|$ is the number of leaves and $\alpha \geq 0$ is a complexity price. Increasing α traces a nested sequence of smaller and smaller trees; choose α by cross-validation (`ccp_alpha` in scikit-learn).

Trees have a distinctive profile. They need no feature scaling, handle mixed numeric and categorical inputs, ignore irrelevant features if they never help a split, and produce rules a person can read. Their boundaries are axis-aligned staircases, so a diagonal boundary costs many splits. And they are unstable: a small change in the data can change the root split and hence the whole tree. Averaging many randomized trees (bagging, random forests) or adding them sequentially (boosting) fixes the instability, and that is the topic of PR 09.

Multilayer perceptrons as nonlinear classifiers

In plain words. A multilayer perceptron (MLP) is an RBF network whose bumps are replaced by learned "ramps," and whose ramps are themselves learned from the labels. Each hidden unit draws one line; the next layer combines the lines into curved regions.

A one-hidden-layer MLP for C classes computes

$$h = \sigma(W_1 x + b_1), \qquad P(y = c \mid x) = \operatorname{softmax}(W_2 h + b_2)_c,$$

where W_1, b_1 map the D inputs to H hidden units, σ is an elementwise nonlinearity (ReLU $\max(0, \cdot)$, logistic or tanh), and W_2, b_2 map hidden units to class scores. All weights are trained jointly by minimizing cross-entropy with gradient descent, the gradients being computed by backpropagation [14]. Unlike every method above, both the features (h) and the classifier are learned from the labels. The universal approximation theorems of Cybenko [15] and Hornik [16] guarantee that one hidden layer with enough units can approximate any continuous function on a compact set; they say nothing about how many units are needed or whether gradient descent will find them. The full story, from neurons and activations to backpropagation and training tricks, is in ML 02; the image-classification version is in IP 13.

Geometrically, with ReLU units each hidden unit splits the input space with a hyperplane, and the network output is piecewise linear on the resulting cells, which is why the MLP boundary in Figure 5 is made of straight segments.

```
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

X, y = make_moons(n_samples=500, noise=0.18, random_state=2)
Xtr, Xte, ytr, yte = train_test_split(
    X, y, test_size=0.3, stratify=y, random_state=2)
for hidden in [(2,), (16,), (16, 16)]:
    mlp = make_pipeline(
        StandardScaler(),
        MLPClassifier(hidden_layer_sizes=hidden, alpha=1e-3,
```

```
                    max_iter=3000, random_state=0))
mlp.fit(Xtr, ytr)
print(hidden, "test accuracy:", round(mlp.score(Xte, yte), 4))
```

With two hidden units the network can only bend the boundary once and reaches 0.833. Sixteen units reach 0.947, and a second layer of 16 does not help on this easy two-dimensional problem. Depth pays off on high-dimensional inputs such as images, where it lets the network build features hierarchically (ML 03).

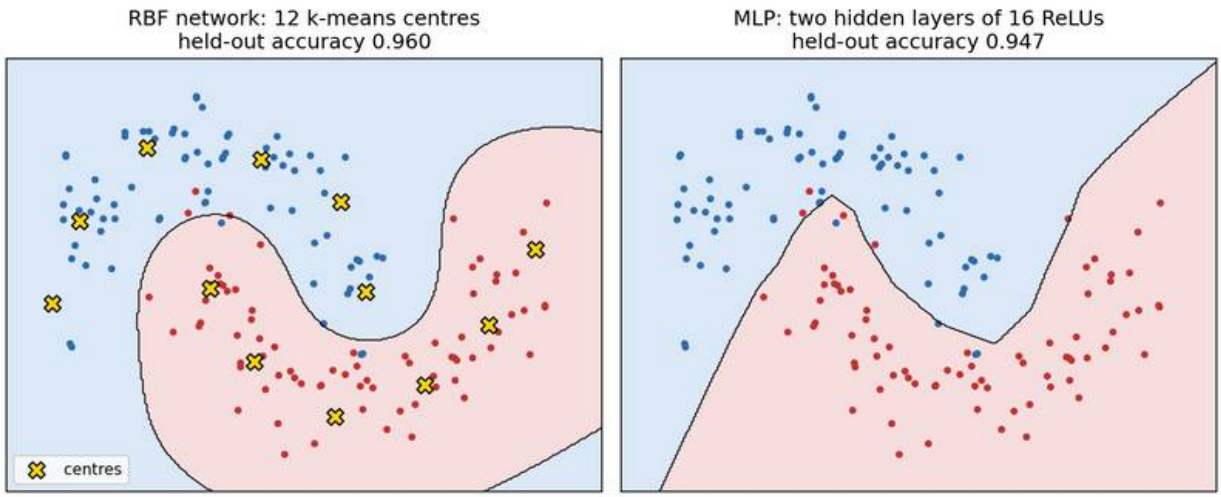


Figure 5 — Two explicit feature maps. Left: an RBF network with 12 k-means centers and a logistic output layer gives smooth, local curves. Right: an MLP with ReLU units gives a piecewise-linear boundary. Held-out points from the same split as the code.

Choosing among nonlinear classifiers

Method	Boundary shape	Training cost	Prediction cost	Key hyperparameters	Typical strengths
k-NN	Any; follows data	None (store data)	$O(ND)$ per query	k, metric, scaling	Simple baseline; good with good embeddings
Kernel SVM	Smooth (RBF)	Roughly $O(N^2)$–$O(N^3)$	$O(N_{\text{sv}}D)$	C, kernel, γ	Small/medium data, strong theory
Gaussian process	Smooth, with uncertainty	$O(N^3)$ exact	$O(N)$ mean, $O(N^2)$ variance	Kernel; set by marginal likelihood	Calibrated uncertainty (regression)
RBF network	Smooth, local	Clustering + linear solve	$O(MD)$	M, γ	Cheap prediction

Method	Boundary shape	Training cost	Prediction cost	Key hyperparameters	Typical strengths
Decision tree	Axis-aligned boxes	$O(DN \log N)$ per level	$O(\text{depth})$	Depth, leaf size, α	Interpretable; mixed feature types
MLP	Piecewise linear (ReLU)	Iterative SGD	$O(\text{weights})$	Width, depth, decay, learning rate	Scales to large data; learns features

Here N_{sv} is the number of support vectors and M the number of RBF centers. The complexity entries are orders of magnitude, not exact counts. No row wins everywhere. Compare candidates under one validation protocol, and remember the lesson of the grid-search example: differences of a fraction of a percent on one split are rarely real.

Modern view

Kernel methods dominated pattern recognition from the late 1990s until deep networks took over large-scale perception around 2012. They did not disappear; they changed roles. Six strands summarize the current picture.

Surveys of the classical field. Hofmann, Schölkopf and Smola's review in the *Annals of Statistics* [7] is the standard map of kernel methods: positive definite kernels and their RKHS, the representer theorem, kernel construction for structured data such as strings and graphs, and the statistical learning theory behind large margins. Cervantes et al. [19] survey SVM classification from the application side: variants developed where the basic SVM struggles (very large data sets, many classes, imbalanced classes), combinations with evolutionary parameter search, a long catalogue of applications, and the method's limitations. For trees, Costa and Pedreira [20] review recent decision-tree research around the three goals of any predictive learner: fitting the training data, generalizing beyond it, and staying interpretable.

Scaling kernels. The $O(N^2)$ memory of a Gram matrix was the main barrier. Rahimi and Recht's random features [17] approximate a shift-invariant kernel such as the RBF by an explicit random map $z(x) \in \mathbb{R}^R$ with $z(x)^\top z(x') \approx k(x, x')$, so that a linear model on R random features stands in for the kernel machine. Liu et al.'s survey [18] organizes a decade of follow-ups: sampling schemes for the random features, theory on how many features are needed for a given accuracy, and connections to deep networks. Combining such approximations with GPU linear algebra, the FALKON line of work trains kernel ridge regression on billions of points [23], and GPyTorch [24] makes exact and approximate Gaussian process inference practical on GPUs with iterative matrix–matrix solvers.

Kernels as a lens on deep learning. Jacot, Gabriel and Hongler showed that, in an infinite-width limit with suitable scaling, training a neural network by gradient descent is equivalent to kernel regression with a fixed **neural tangent kernel** [21]. Belkin, Ma and Mandal argued that phenomena puzzling in deep learning, such as fitting the training data exactly yet still generalizing, already appear in kernel machines, so understanding kernels is a prerequisite for understanding

deep networks [22]. Real networks of finite width do learn their features, which a fixed kernel cannot, so this analogy is a theoretical tool, not a replacement.

Trees still rule tabular data. For data in tables (rows of heterogeneous measurements rather than pixels or words), Grinsztajn, Oyallon and Varoquaux benchmarked tree ensembles against deep architectures on 45 medium-sized datasets and found the tree-based models still ahead, attributing the gap to neural networks' sensitivity to uninformative features, their rotation invariance, and their bias toward smooth functions [25]. The newest challenger is a **tabular foundation model**: TabPFN, a transformer trained across millions of synthetic datasets, predicts on a new small dataset in a single forward pass. Its authors report that it outperforms previous methods, including tuned gradient-boosted tree ensembles, on datasets with up to 10,000 samples, with far less training time [26].

Neighbors on learned features. The nearest-neighbor rule has had a revival as a classifier on top of large pretrained embeddings. Once features are good, a k-NN or cosine-similarity classifier is often the simplest strong baseline, and it can add new classes without retraining. IP 13 and ML 06 discuss this pattern.

Open problems. Exact kernel methods and GPs still scale poorly in N; approximation quality versus speed remains an active trade-off [18]. Choosing kernels for structured data is still largely manual. Single trees remain unstable, and globally optimal trees are expensive to compute [20]. And for tabular data, whether foundation-style models will displace tuned tree ensembles beyond small datasets is an open question as of 2026.

Key takeaways

- Nonlinear boundaries come from three ideas: comparing with stored examples (k-NN), lifting to a feature space where a linear rule works (kernels, RBF networks, MLPs), or partitioning the space (trees).
- k-NN's behavior is set by the metric and scaling as much as by k; for unit vectors, Euclidean and cosine neighbors coincide.
- A kernel is valid exactly when every Gram matrix is symmetric positive semidefinite (Mercer's condition). The kernel trick replaces inner products by kernel values in any algorithm written in inner products.
- The soft-margin SVM minimizes $\frac{1}{2}\|w\|^2$ plus C times hinge loss. Its dual depends on the data only through the kernel, and only support vectors ($\alpha_n > 0$) affect predictions. Tune C and γ jointly, on standardized features, by cross-validation.
- GPs give kernel predictions with uncertainty; the RVM (Tipping) gives sparse probabilistic kernel models. SVM scores are not probabilities.
- RBF networks use a few explicit Gaussian bumps; trees grow greedily by impurity decrease (Gini or entropy) and need pruning; MLPs learn their own features and are covered in depth in ML 02.
- No family wins everywhere: kernels for small/medium smooth problems, trees and their ensembles for tabular data, deep networks for raw perceptual data, neighbors on top of good embeddings.

Exercises

1. **Distance and cosine.** Derive $\|u - v\|^2 = 2 - 2u^\top v$ for unit vectors, and show that the 1-NN under Euclidean distance equals the 1-NN under cosine similarity. Does this still hold if the vectors are not normalized?

   Hint

   Expand $\|u-v\|^2 = u^\top u - 2u^\top v + v^\top v$. With unit norms, distance is a decreasing function of $u^\top v$, so the minimizer of one is the maximizer of the other. Without normalization the norms enter: a long vector pointing in a slightly different direction can be closer in Euclidean distance than a short, perfectly aligned one.

2. **Building a kernel.** Show that $k(x,z) = (x^\top z)^2$ is a valid kernel in $\mathbb{R}^2$ by writing down its feature map. Then use the closure rules to argue that the Gaussian kernel $\exp(-\gamma\|x - z\|^2)$ is valid.

   Hint

   $\phi(x) = (x_1^2, x_2^2, \sqrt{2}x_1x_2)$. For the Gaussian, write
   $\exp(-\gamma\|x - z\|^2) = \exp(-\gamma\|x\|^2)\,\exp(2\gamma x^\top z)\,\exp(-\gamma\|z\|^2)$. The middle factor is exp of a valid (scaled linear) kernel, and the outer factors have the form $f(x)\,k(x,z)\,f(z)$.

3. **The limits of γ.** For a fixed finite data set, what does the RBF Gram matrix tend to as $\gamma \to 0$ and as $\gamma \to \infty$? Relate each limit to the leftmost and rightmost columns of Figure 2.

   Hint

   As $\gamma \to 0$ every entry tends to 1 (the all-ones matrix): all points look alike and the boundary becomes nearly linear or constant. As $\gamma \to \infty$ the off-diagonal entries tend to 0 and $K \to I$: each point is only similar to itself, so the SVM can memorize every training label, which is the "islands" regime.

4. **Support vectors and cost.** An RBF-SVM trained on 100,000 images keeps 90% of them as support vectors. What does this imply about the data and about prediction time? Name two things you could try.

   Hint

   Many points sit inside the margin or are misclassified: the classes overlap heavily in this feature space, or C is small or γ is badly chosen. Each prediction needs about 90,000 kernel evaluations. Try re-tuning C and γ, better features, random features with a linear SVM, or a different model family.

5. **Impurity by hand.** A node has 40 points of class A and 40 of class B. Split 1 gives children (30 A, 10 B) and (10 A, 30 B). Split 2 gives (40 A, 20 B) and (0 A, 20 B). Compute the misclassification-error decrease and the Gini decrease for both splits. Which split does each measure prefer?

Hint

Root Gini = 0.5, root error = 0.5. Split 1: both children have error 0.25 and Gini $2 \cdot 0.75 \cdot 0.25 = 0.375$; weighted error 0.25, weighted Gini 0.375. Split 2: left child error 1/3, Gini $2 \cdot \frac{2}{3} \cdot \frac{1}{3} = 4/9$; right child pure. Weighted error $\frac{60}{80} \cdot \frac{1}{3} = 0.25$, weighted Gini $\frac{60}{80} \cdot \frac{4}{9} = 1/3$. Error cannot tell them apart (both 0.25); Gini prefers split 2, which creates a pure node.

6. **RBF network design.** In the RBF-network code, vary $M \in \{2, 4, 12, 50\}$ and $\gamma \in \{0.1, 1, 10\}$. Which combinations underfit and which overfit? Why does a small M with a large γ behave badly?

Hint

With few centers and a large γ, the bumps are narrow and most points activate none of them strongly, so the design matrix is nearly zero away from the centers and the classifier falls back to the bias. Many centers with a large γ start to behave like a kernel machine with too little regularization.

References

1. S. Theodoridis and K. Koutroumbas, *Pattern Recognition*, 4th ed., Academic Press, 2009, Chapters 2 and 4. publisher page
2. C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006, Sections 2.5.2, 5.1–5.3, 6.1–6.4, 7.1–7.2 and 14.4. official book page and PDF
3. T. Cover and P. Hart, "Nearest Neighbor Pattern Classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, 1967. doi:10.1109/TIT.1967.1053964
4. J. Mercer, "Functions of Positive and Negative Type, and Their Connection with the Theory of Integral Equations," *Philosophical Transactions of the Royal Society of London, Series A*, vol. 209, pp. 415–446, 1909. doi:10.1098/rsta.1909.0016
5. B. E. Boser, I. M. Guyon, and V. N. Vapnik, "A Training Algorithm for Optimal Margin Classifiers," *Proceedings of the Fifth Annual Workshop on Computational Learning Theory (COLT)*, pp. 144–152, 1992. doi:10.1145/130385.130401
6. C. Cortes and V. Vapnik, "Support-Vector Networks," *Machine Learning*, vol. 20, pp. 273–297, 1995. doi:10.1007/BF00994018
7. T. Hofmann, B. Schölkopf, and A. J. Smola, "Kernel Methods in Machine Learning," *The Annals of Statistics*, vol. 36, no. 3, 2008. doi:10.1214/009053607000000677
8. C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning*, MIT Press, 2006. book page
9. M. E. Tipping, "Sparse Bayesian Learning and the Relevance Vector Machine," *Journal of Machine Learning Research*, vol. 1, pp. 211–244, 2001. JMLR
10. J. Moody and C. J. Darken, "Fast Learning in Networks of Locally-Tuned Processing Units," *Neural Computation*, vol. 1, no. 2, pp. 281–294, 1989. doi:10.1162/neco.1989.1.2.281

11. J. Park and I. W. Sandberg, "Universal Approximation Using Radial-Basis-Function Networks," *Neural Computation*, vol. 3, no. 2, pp. 246–257, 1991. doi:10.1162/neco.1991.3.2.246
12. L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*, Chapman and Hall/CRC, 2017 edition. doi:10.1201/9781315139470
13. J. R. Quinlan, "Induction of Decision Trees," *Machine Learning*, vol. 1, pp. 81–106, 1986. doi:10.1007/BF00116251
14. D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Representations by Back-Propagating Errors," *Nature*, vol. 323, pp. 533–536, 1986. doi:10.1038/323533a0
15. G. Cybenko, "Approximation by Superpositions of a Sigmoidal Function," *Mathematics of Control, Signals, and Systems*, vol. 2, pp. 303–314, 1989. doi:10.1007/BF02551274
16. K. Hornik, "Approximation Capabilities of Multilayer Feedforward Networks," *Neural Networks*, vol. 4, no. 2, pp. 251–257, 1991. doi:10.1016/0893-6080(91)90009-T
17. A. Rahimi and B. Recht, "Random Features for Large-Scale Kernel Machines," *Advances in Neural Information Processing Systems 20 (NIPS)*, 2007. paper
18. F. Liu, X. Huang, Y. Chen, and J. A. K. Suykens, "Random Features for Kernel Approximation: A Survey on Algorithms, Theory, and Beyond," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 10, pp. 7128–7148, 2022. doi:10.1109/TPAMI.2021.3097011
19. J. Cervantes, F. Garcia-Lamont, L. Rodríguez-Mazahua, and A. Lopez, "A Comprehensive Survey on Support Vector Machine Classification: Applications, Challenges and Trends," *Neurocomputing*, vol. 408, pp. 189–215, 2020. doi:10.1016/j.neucom.2019.10.118
20. V. G. Costa and C. E. Pedreira, "Recent Advances in Decision Trees: An Updated Survey," *Artificial Intelligence Review*, vol. 56, pp. 4765–4800, 2023. doi:10.1007/s10462-022-10275-5
21. A. Jacot, F. Gabriel, and C. Hongler, "Neural Tangent Kernel: Convergence and Generalization in Neural Networks," *NeurIPS*, 2018. arXiv:1806.07572
22. M. Belkin, S. Ma, and S. Mandal, "To Understand Deep Learning We Need to Understand Kernel Learning," arXiv:1802.01396, 2018. arXiv
23. G. Meanti, L. Carratino, L. Rosasco, and A. Rudi, "Kernel Methods Through the Roof: Handling Billions of Points Efficiently," *Advances in Neural Information Processing Systems 33 (NeurIPS)*, 2020. paper
24. J. R. Gardner, G. Pleiss, D. Bindel, K. Q. Weinberger, and A. G. Wilson, "GPyTorch: Blackbox Matrix-Matrix Gaussian Process Inference with GPU Acceleration," *NeurIPS*, 2018. arXiv:1809.11165
25. L. Grinsztajn, E. Oyallon, and G. Varoquaux, "Why Do Tree-Based Models Still Outperform Deep Learning on Typical Tabular Data?," *Advances in Neural Information Processing Systems 35 (NeurIPS), Datasets and Benchmarks Track*, 2022. paper
26. N. Hollmann, S. Müller, L. Purucker, A. Krishnakumar, et al., "Accurate Predictions on Small Data with a Tabular Foundation Model," *Nature*, vol. 637, pp. 319–326, 2025. doi:10.1038/s41586-024-08328-6
27. scikit-learn developers, "Support Vector Machines," user guide, accessed 2026-11-05. documentation

PR 05 • Feature Selection, Transforms and Dimensionality

Choose and build features: filter, wrapper and embedded selection, class-separability criteria, PCA, Fisher LDA, kernel PCA, ICA and manifold learning (Isomap, LLE, t-SNE, UMAP), and how to validate a feature pipeline without leakage.

Online edition: https://blog.shyandram.dev/en/learn/pattern-recognition/features-dimensionality/

What you'll learn

- Why fewer, better features often beat more features, and the difference between feature **selection**, feature **extraction** and feature **learning**.
- Filter, wrapper and embedded selection, and the most common mistake: selecting features outside cross-validation.
- How to score features and feature subsets: Fisher ratio, scatter-matrix criteria, Bhattacharyya distance and mutual information.
- Linear extraction: PCA (with whitening) and Fisher's linear discriminant, derived and compared.
- Nonlinear and non-Gaussian extraction: kernel PCA, ICA (and NMF), and manifold learning with Isomap, LLE, t-SNE and UMAP, including what their pictures can and cannot prove.
- Classic hand-designed image and signal descriptors, and where all of this stands in 2026.

The big picture

Two recognition systems can use exactly the same classifier and still behave very differently, because they describe the input differently. A texture histogram throws away where things are. A shape descriptor may ignore color. An embedding learned from web photos may quietly encode the background. Every representation keeps some variation and discards the rest, and the classifier can only use what was kept.

> **Explain like I'm five**
>
> Describe a toy to a friend so they can pick it out of a toy box. "It's red, round and the size of an orange" works. "It's made of atoms" is true but useless. Telling them all 500 things you can see is too much to remember. A good description is short and keeps exactly the clues that tell this toy apart from the others.

This chapter is about building that short, useful description. There are three ways to get one:

- **Selection** keeps a subset of the original measurements, for example 20 of 5,000 gene-expression levels. The kept features remain interpretable.
- **Extraction** computes new coordinates as functions of all measurements, for example the projection onto a few principal components.
- **Learning** estimates an extraction rule from data, often with a deep network (ML 06). PCA is already a (very simple) learned extraction rule; the categories overlap.

Why bother, if a flexible classifier could use all the features? Three reasons. With a fixed number of training samples, adding weakly informative features can make a trained classifier *worse*, because each extra dimension adds parameters to estimate and room to overfit: the "curse of dimensionality" and the peaking phenomenon discussed by Theodoridis and Koutroumbas [1]. Fewer features are cheaper to measure, store and compute. And a small, meaningful feature set helps people understand what the classifier relies on. Guyon and Elisseeff list the same three motives in their classic introduction to feature selection [3]. The textbook background for this chapter is Theodoridis and Koutroumbas, Chapters 5–7 [1], and Bishop, Sections 4.1.4–4.1.6 and 12.1–12.4 [2]. For hand-designed image features, see also IP 12.

Feature selection

In plain words. Feature selection asks: which of the measurements I already have should I keep? You need a way to score features (or groups of features) and a way to search, because trying every subset is impossible.

With D features there are $2^D - 1$ non-empty subsets: about 10^{30} for $D = 100$. Exhaustive search is out of the question except for tiny D. The approaches differ in how they score and how they search [3], [22], [23].

Filter methods

A **filter** scores each feature (or subset) with a criterion computed directly from the data, independently of the final classifier, then keeps the best. Typical criteria are variance (drop near-constant features), a class-separation statistic such as the ANOVA F statistic or the Fisher ratio (next section), correlation with the label, and the mutual information

$$I(x_j; y) = \sum_{v} \sum_{c} p(x_j = v, y = c) \, \log \frac{p(x_j = v, y = c)}{p(x_j = v)\, P(y = c)},$$

where x_j is feature j (discretized to values v), y is the class label with values c, and p, P are joint and marginal probabilities estimated from the training set. $I(x_j; y) = 0$ exactly when the feature and the label are statistically independent.

Filters are fast and classifier-agnostic, but univariate filters have two blind spots. **Redundancy**: two copies of the same good feature both score high, yet the second adds nothing. Peng, Long and Ding's mRMR criterion addresses this by rewarding relevance to the label while penalizing mutual information with already-selected features [6]. **Interactions**: two features can each be

useless alone and decisive together. In the exclusive-or pattern $y = \mathbf{1}[x_1 > 0] \oplus \mathbf{1}[x_2 > 0]$, each feature on its own is independent of y, so every univariate filter discards both.

Wrapper methods

A **wrapper** scores a candidate subset by training the actual classifier on it and measuring validation performance [4]. Because exhaustive search is impossible, wrappers use heuristics such as:

- **sequential forward selection**: start empty, repeatedly add the feature that most improves the validation score;
- **sequential backward elimination**: start with all features, repeatedly remove the least useful one;
- floating variants that may undo earlier steps [1].

Wrappers see interactions and redundancy through the classifier's eyes, but they are expensive (a full training run per candidate) and, because they try many subsets against the same validation data, they can overfit the validation set itself. Kohavi and John's paper that introduced the term is still the standard reference for these trade-offs [4].

Embedded methods

An **embedded** method selects features as a side effect of training. The best-known example is the lasso, which adds an ℓ_1 penalty to a linear model [5]:

$$\hat{w} = \arg\min_{w} \frac{1}{2N} \sum_{n=1}^{N} \left(t_n - w^{\top} x_n\right)^2 + \lambda \lVert w \rVert_1, \qquad \lVert w \rVert_1 = \sum_{j=1}^{D} |w_j|,$$

with targets t_n, inputs x_n and penalty strength $\lambda \geq 0$. The corners of the ℓ_1 ball make many coordinates of $\hat{w}$ exactly zero, so the surviving coordinates are the selected features. The same idea applies to logistic regression for classification. Decision trees and their ensembles (PR 04, PR 09) also embed selection: a feature that never helps a split is never used.

The selection-bias trap

Supervised selection uses the labels. If you select features on the full data set and *then* cross-validate only the classifier, the held-out folds have already influenced which features were chosen, and the error estimate is optimistic. Ambroise and McLachlan showed that this mistake made published gene-expression classifiers look nearly perfect; with selection performed inside each cross-validation fold, the error was no longer zero [7]. The code makes the effect concrete with features that are pure noise.

```
import numpy as np
```

```
from sklearn.feature_selection import SelectKBest, f_classif
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score
from sklearn.pipeline import make_pipeline

rng = np.random.default_rng(0)
X = rng.normal(size=(60, 5000))          # 60 samples, 5000 pure-noise features
y = rng.integers(0, 2, size=60)          # labels unrelated to X

# Wrong: select on all data, then cross-validate the classifier only.
X_sel = SelectKBest(f_classif, k=20).fit_transform(X, y)
wrong = cross_val_score(LogisticRegression(max_iter=1000), X_sel, y, cv=5)

# Right: selection is part of the model and is refitted in every fold.
model = make_pipeline(SelectKBest(f_classif, k=20),
                      LogisticRegression(max_iter=1000))
right = cross_val_score(model, X, y, cv=5)
print(f"selection outside CV: {wrong.mean():.2f}")
print(f"selection inside CV:  {right.mean():.2f}")
```

The labels are random, so no method can truly beat 50% accuracy. Selecting outside cross-validation reports 0.98. Putting the selector inside the pipeline reports 0.58, which is chance level up to the noise of a 60-sample estimate. The rule generalizes: every step that looks at labels (selection, supervised projection such as LDA, choosing the number of features) belongs inside the training procedure. Even unsupervised steps such as PCA and scaling should be fitted on training folds only, so that the procedure you evaluate is the procedure you will deploy.

Evaluating features and feature subsets

In plain words. A good feature puts the classes far apart compared with how spread out each class is. Several formulas turn this idea into a number, from a simple per-feature ratio to criteria for whole subsets.

Fisher's discriminant ratio for a single feature compares the spread *between* class means with the spread *within* classes:

$$\mathrm{FDR}_j = \frac{\sum_c P_c\,(\mu_{cj} - \mu_j)^2}{\sum_c P_c\,\sigma_{cj}^2},$$

where P_c is the proportion of class c, μ_{cj} and σ_{cj}^2 are the mean and variance of feature j in class c, and μ_j is the overall mean [1]. (For two classes, Theodoridis and Koutroumbas write it as $(\mu_1 - \mu_2)^2/(\sigma_1^2 + \sigma_2^2)$; the idea is the same.)

For a **subset** of features, use the scatter matrices. With x the feature vector restricted to the subset and μ_c, μ the class and overall mean vectors,

$$S_W = \sum_c P_c\,\Sigma_c, \qquad S_B = \sum_c P_c\,(\mu_c - \mu)(\mu_c - \mu)^\top, \qquad J = \mathrm{tr}(S_W^{-1} S_B),$$

where Σ_c is the covariance of class c, S_W is the within-class scatter, S_B is the between-class scatter, and tr is the trace. J is large when class means are far apart relative to the within-class spread, measured in the directions that matter [1]. Unlike summing per-feature ratios, J accounts for correlation between the features.

Two further criteria come from probability. The **Bhattacharyya distance** between two Gaussian classes $\mathcal{N}(\mu_1, \Sigma_1)$ and $\mathcal{N}(\mu_2, \Sigma_2)$ is

$$B = \frac{1}{8}(\mu_1 - \mu_2)^\top \Big(\frac{\Sigma_1 + \Sigma_2}{2}\Big)^{-1}(\mu_1 - \mu_2) + \frac{1}{2}\ln \frac{|\frac{1}{2}(\Sigma_1 + \Sigma_2)|}{\sqrt{|\Sigma_1||\Sigma_2|}},$$

with $|\cdot|$ the determinant. The first term measures mean separation; the second measures covariance difference. It gives an upper bound on the Bayes error for equal priors, $P_e \le \frac{1}{2}e^{-B}$ [1]. **Mutual information** with the label, introduced above, is the distribution-free counterpart.

The code compares single-feature scores with a subset criterion on the 13-feature wine data.

```
import numpy as np
from sklearn.datasets import load_wine
from sklearn.preprocessing import StandardScaler

X, y = load_wine(return_X_y=True)
X = StandardScaler().fit_transform(X)
classes = np.unique(y)
mu = X.mean(axis=0)

# Per-feature Fisher ratio: between-class over within-class variance.
between = sum((y == c).mean() * (X[y == c].mean(0) - mu) ** 2 for c in classes)
within = sum((y == c).mean() * X[y == c].var(0) for c in classes)
fdr = between / within
print("top-3 features by Fisher ratio:", np.argsort(fdr)[::-1][:3],
      np.round(np.sort(fdr)[::-1][:3], 2))

def J(cols):  # scatter-matrix criterion trace(Sw^-1 Sb) on a feature subset
    Z = X[:, cols]
    m = Z.mean(0)
    Sw = sum((y == c).mean() * np.cov(Z[y == c].T, bias=True).reshape(len(cols), -1)
             for c in classes)
    Sb = sum((y == c).mean() * np.outer(Z[y == c].mean(0) - m, Z[y == c].mean(0) - m)
             for c in classes)
    return np.trace(np.linalg.solve(Sw, Sb))

top2 = list(np.argsort(fdr)[::-1][:2])
print("J(top-2 by ratio) =", round(J(top2), 2))
best_pair = max(((i, j) for i in range(13) for j in range(i + 1, 13)),
                key=lambda p: J(list(p)))
print("best pair by J:", best_pair, "J =", round(J(list(best_pair)), 2))
```

Features 6, 12 and 11 have the highest individual ratios (2.67, 2.38, 2.17). Yet the best *pair* by J is features 6 and 9 ($J = 5.39$), not the top two individual features ($J = 4.93$): feature 12 overlaps in

information with feature 6, while feature 9 complements it. This is the redundancy problem of univariate filters, seen in numbers.

All of these criteria are proxies. The final judge of a feature set is the cross-validated performance of the complete pipeline, estimated without leakage (PR 09).

Principal component analysis

In plain words. PCA rotates the coordinate system so that the first axis points in the direction where the data vary most, the second in the direction of most remaining variation, and so on. Keeping only the first few axes gives the best low-dimensional summary in the least-squares sense. It never looks at the labels.

Derivation

Let $x_1, \dots, x_N \in \mathbb{R}^D$ have mean $\bar{x}$ and covariance $\Sigma = \frac{1}{N-1}\sum_n (x_n - \bar{x})(x_n - \bar{x})^\top$. The variance of the data projected onto a unit vector u is $u^\top \Sigma u$. Maximizing it subject to $u^\top u = 1$ with a Lagrange multiplier λ gives

$$\Sigma u = \lambda u,$$

so the best direction is an eigenvector of Σ, and the variance it captures is its eigenvalue λ. The first principal direction is the eigenvector with the largest eigenvalue; the next ones are the remaining eigenvectors in decreasing order of eigenvalue, each orthogonal to the previous ones [2], [8], [9]. Keeping the top r directions also minimizes reconstruction error [2]. With the divisor-$N-1$ covariance used here, the **mean squared Euclidean error per sample** is $\frac{1}{N}\sum_n \|x_n - \hat{x}_n\|^2 = \frac{N-1}{N}\sum_{i>r}\lambda_i$. Dividing by $N-1$ instead gives exactly the discarded eigenvalue sum; a per-coordinate mean additionally divides by D. State the normalization before comparing a plotted error with the eigenvalues.

In practice PCA is computed with the singular value decomposition (SVD) of the centered data matrix $X_c \in \mathbb{R}^{N\times D}$ (row n is $x_n - \bar{x}$):

$$X_c = USV^\top, \qquad \lambda_i = \frac{s_i^2}{N-1}, \qquad Z = X_c V_r,$$

where U and V have orthonormal columns, $S = \operatorname{diag}(s_1, s_2, \dots)$ holds the singular values in decreasing order, V_r contains the first r columns of V (the principal directions), and $Z \in \mathbb{R}^{N\times r}$ are the **scores**. The fraction $\lambda_i / \sum_k \lambda_k$ is the **explained variance ratio** of component i. To reconstruct, compute $\hat{x} = \bar{x} + V_r z$.

Whitening

Whitening rescales each score to unit variance:

$$z_i^{\text{white}} = \frac{z_i}{\sqrt{\lambda_i + \epsilon}},$$

where z_i is the score on component i, λ_i is that component's variance from the training data, and $\epsilon \geq 0$ is a small constant that keeps tiny eigenvalues from blowing up. For retained components with $\lambda_i > 0$, $\epsilon = 0$ gives identity covariance. With positive ϵ, the diagonal entries are instead $\lambda_i/(\lambda_i + \epsilon)$, so tiny-variance directions are deliberately suppressed. Drop zero-variance components or use positive ϵ; do not divide by zero. This makes all directions equally loud, which helps some algorithms (ICA below, k-means, some neural networks) but also amplifies directions that were small because they were mostly noise. Whitening is a modeling choice, not a free improvement.

```
import numpy as np
from sklearn.datasets import load_digits
from sklearn.decomposition import PCA

X, _ = load_digits(return_X_y=True)          # 1797 images of 8x8 = 64 pixels
Xc = X - X.mean(axis=0)
U, S, Vt = np.linalg.svd(Xc, full_matrices=False)
eigvals = S ** 2 / (len(X) - 1)               # variances along each direction
ratio = eigvals / eigvals.sum()
r = int(np.searchsorted(np.cumsum(ratio), 0.90) + 1)
print("components for 90% variance:", r)

Z = Xc @ Vt[:r].T                             # scores, N x r
X_hat = Z @ Vt[:r] + X.mean(axis=0)           # reconstruction
print("relative reconstruction error:",
      round(np.linalg.norm(X - X_hat) / np.linalg.norm(Xc), 3))

mse_per_sample = np.mean(np.sum((X - X_hat) ** 2, axis=1))
assert np.isclose(mse_per_sample, (len(X) - 1) / len(X) * eigvals[r:].sum())

pca = PCA(n_components=r).fit(X)
assert np.allclose(pca.explained_variance_, eigvals[:r])
assert np.allclose(np.abs(pca.components_), np.abs(Vt[:r]))   # equal up to sign
Zw = Z / np.sqrt(eigvals[:r])                 # whitened scores
print("whitened covariance is identity:",
      np.allclose(np.cov(Zw.T), np.eye(r)))
```

Twenty-one of the 64 pixel dimensions hold 90% of the variance of the digits, and the reconstruction from them has a relative error of 0.31. The hand-written SVD agrees with scikit-learn up to the sign of each component (an eigenvector's sign is arbitrary). Figure 1 shows the cumulative curve and reconstructions.

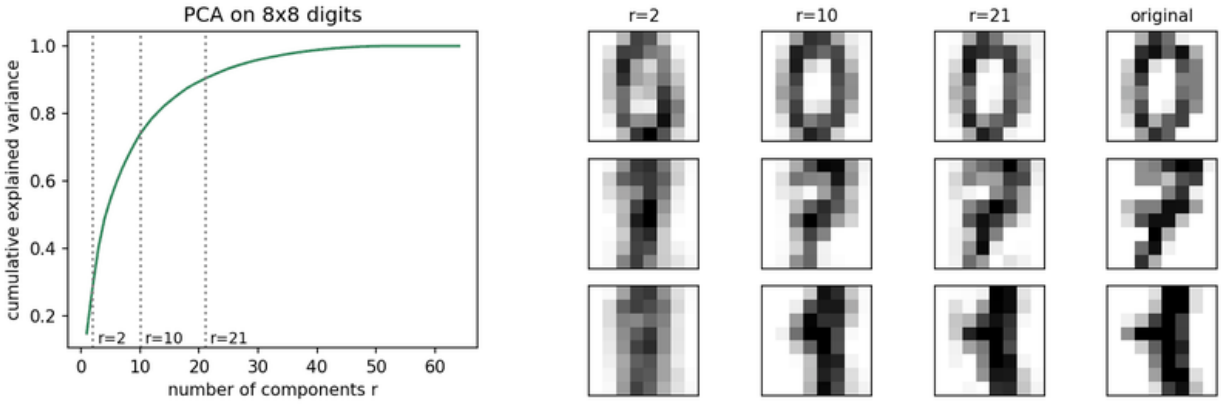


Figure 1 — PCA on the scikit-learn digits. Two components give blurry blobs; ten already show the digit; 21 components (90% of the variance) are close to the original.

Variance is not class information

PCA ranks directions by variance, and variance is not the same as usefulness for classification. In Figure 2, a large nuisance variation (think lighting) dominates the first principal component, while the class difference lives in a direction with small variance. Projecting onto PC 1 keeps 96% of the variance and throws away essentially all the class information. A supervised projection (Fisher's LDA, next section) finds the useful direction immediately.

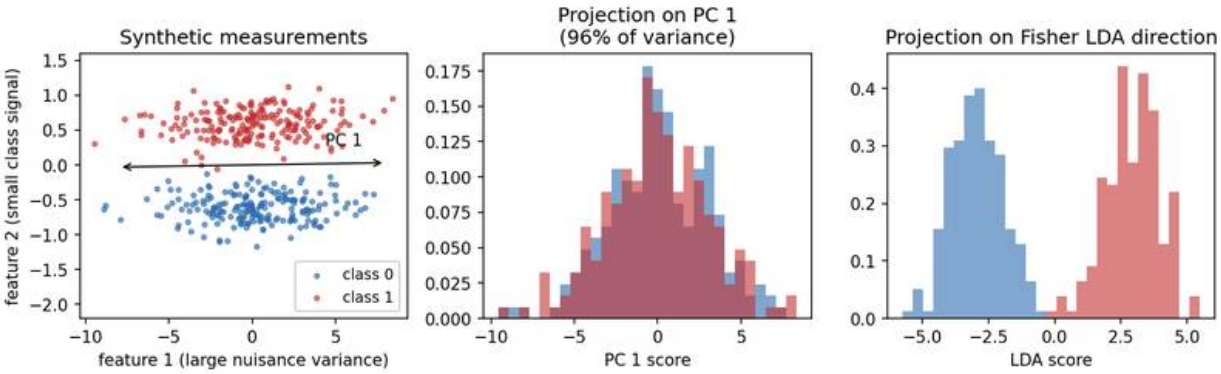


Figure 2 — Variance is not class information. PC 1 (96% of the variance) follows the nuisance axis and mixes the classes; the Fisher LDA direction separates them. Synthetic data; generated by scripts/figures/pr_05_features_dimensionality_extra.py.

Jolliffe and Cadima's review [10] covers what PCA can and cannot do, and its many variants (for example sparse and robust PCA). Probabilistic PCA and factor analysis, which turn PCA into a latent-variable model, are developed in Bishop, Chapter 12 [2], and in ML 06A.

Fisher's linear discriminant analysis

In plain words. LDA looks for the directions in which the class means are far apart *relative to* how spread out each class is. It uses the labels, so it finds directions that matter for classification, but it can produce at most one fewer direction than there are classes.

Fisher's criterion for a projection direction v is the ratio of between-class to within-class scatter along v [11]:

$$J(v) = \frac{v^\top S_B v}{v^\top S_W v},$$

with S_W and S_B as defined in the previous section. Setting the gradient to zero gives the **generalized eigenproblem**

$$S_B v = \lambda S_W v,$$

and the best directions are the eigenvectors with the largest eigenvalues λ (equivalently, eigenvectors of $S_W^{-1} S_B$ when S_W is invertible) [1], [2]. For two classes there is a closed form, $v \propto S_W^{-1}(\mu_1 - \mu_2)$, which reappears as the Gaussian LDA classifier in PR 03.

Why at most $K-1$ directions. S_B is a weighted sum of K rank-one matrices $(\mu_c - \mu)(\mu_c - \mu)^\top$. The vectors $\mu_c - \mu$ are linearly dependent, because $\sum_c P_c(\mu_c - \mu) = 0$. So S_B has rank at most $K-1$, and at most $K-1$ eigenvalues are nonzero. Ten digit classes give at most nine discriminant directions, whatever the input dimension.

When S_W is singular. With fewer samples than features (common with images), S_W cannot be inverted. Standard fixes are to apply PCA first and then LDA in the reduced space, or to shrink S_W toward a multiple of the identity (regularized LDA).

```
from sklearn.datasets import load_digits
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from sklearn.model_selection import cross_val_score
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import make_pipeline
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

X, y = load_digits(return_X_y=True)
lda = LDA().fit(X, y)
print("discriminant directions for 10 classes:", lda.transform(X).shape[1])
for name, reducer in [("PCA", PCA(n_components=2)),
                      ("LDA", LDA(n_components=2))]:
    pipe = make_pipeline(StandardScaler(), reducer,
                         KNeighborsClassifier(n_neighbors=5))
    acc = cross_val_score(pipe, X, y, cv=5).mean()
    print(f"{name} to 2-D, then 5-NN: CV accuracy {acc:.3f}")
```

LDA returns nine directions for ten classes, as predicted. Squeezed to two dimensions, a 5-NN classifier on LDA coordinates reaches a cross-validated accuracy of 0.624, against 0.512 on the first

two principal components. Because LDA uses labels, it sits inside the pipeline and is refitted in every fold. Figure 3 shows why: LDA pulls whole digit classes apart, while PCA mostly shows the directions of largest pixel variation.

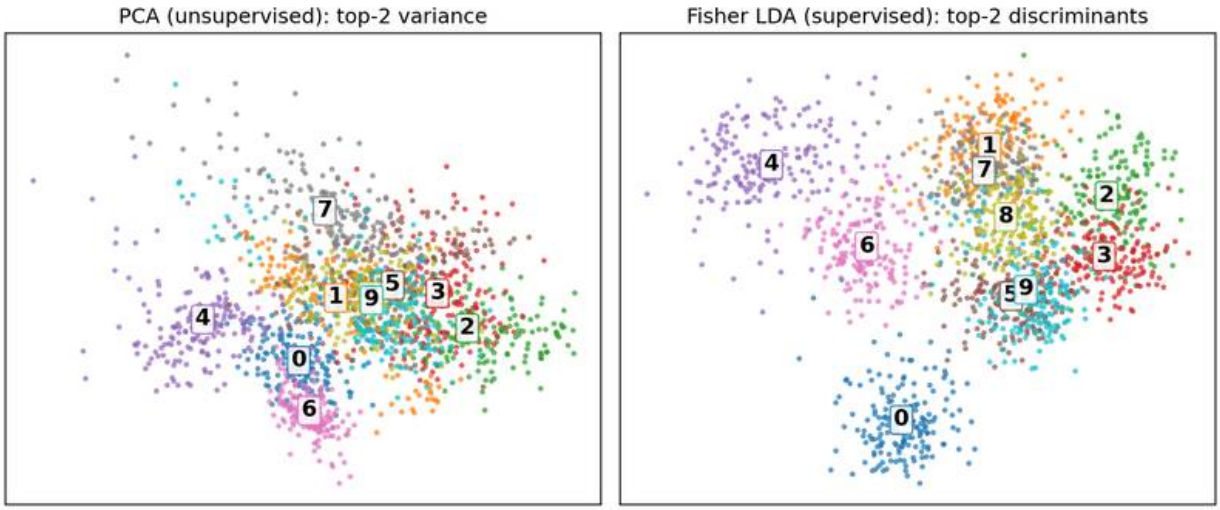


Figure 3 — Digits projected to two dimensions. Left: the top-2 principal components (no labels used). Right: the top-2 Fisher discriminants (labels used). Class labels mark each class mean.

Kernel PCA

In plain words. Kernel PCA is PCA performed in the rich feature space of a kernel (PR 04), without ever computing the features. It can unroll structures that no linear projection can.

Schölkopf, Smola and Müller showed that PCA in the feature space $\phi(x)$ only needs the kernel matrix $K_{ij} = k(x_i, x_j)$ [12]. Center it in feature space,

$$\tilde{K} = K - \mathbf{1}_N K - K\mathbf{1}_N + \mathbf{1}_N K \mathbf{1}_N,$$

where $\mathbf{1}_N$ is the $N \times N$ matrix with every entry $1/N$. Solve the eigenproblem $\tilde{K}a_i = N\lambda_i a_i$ for coefficient vectors $a_i \in \mathbb{R}^N$, normalized so that $N\lambda_i a_i^\top a_i = 1$. The i-th kernel principal component of a point x is then

$$z_i(x) = \sum_{n=1}^{N} a_{in}\, \tilde{k}(x_n, x),$$

where $\tilde{k}$ is the kernel centered in the same way. The cost is that of an $N \times N$ eigenproblem, and a new point needs N kernel evaluations.

Kernel PCA is still unsupervised: it ranks components by variance in feature space, not by class separation. On the two-ring data, the first two RBF kernel components describe the *angle* around the rings and mix the classes; the third component describes the *radius* and separates them perfectly (Figure 4).

```
import numpy as np
from sklearn.datasets import make_circles
from sklearn.decomposition import PCA, KernelPCA
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score
from sklearn.pipeline import make_pipeline

X, y = make_circles(n_samples=400, factor=0.4, noise=0.07, random_state=0)
Z = KernelPCA(n_components=4, kernel="rbf", gamma=2).fit_transform(X)
print("|corr(component, label)|:",
      np.round([abs(np.corrcoef(Z[:, i], y)[0, 1]) for i in range(4)], 3))

for name, reducer in [("linear PCA, 2 comps", PCA(n_components=2)),
                      ("kernel PCA, 3 comps", KernelPCA(n_components=3,
                                                        kernel="rbf", gamma=2))]:
    acc = cross_val_score(make_pipeline(reducer, LogisticRegression()),
                          X, y, cv=5).mean()
    print(f"{name} -> logistic regression: {acc:.3f}")
```

Only the third component correlates with the label (0.982; the others are below 0.05). A logistic regression on the two linear principal components cannot beat chance (0.432), while on three kernel components it is perfect (1.000).

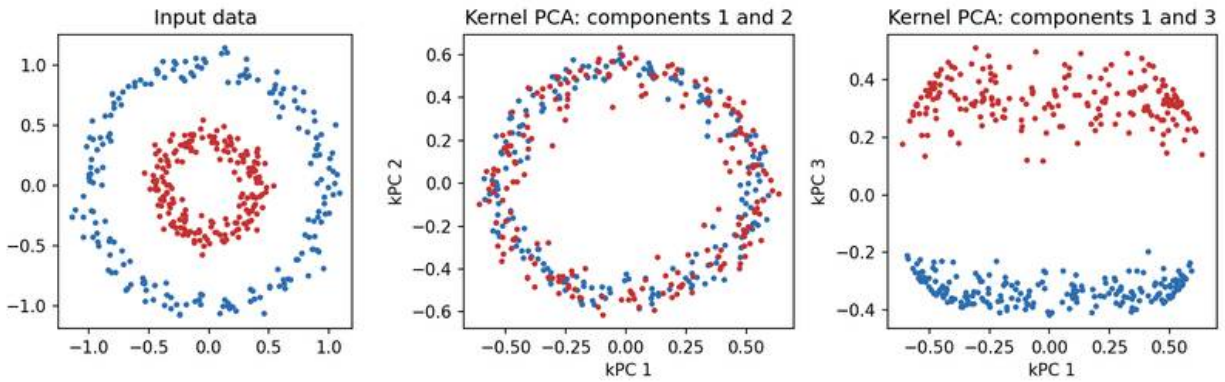


Figure 4 — Kernel PCA (RBF, $\gamma = 2$) on two rings. The leading components capture the angle and mix the classes; component 3 captures the radius and separates them. High variance in feature space still does not mean high class information.

Independent component analysis

In plain words. At a party, two microphones each record a mixture of two voices. ICA tries to recover the individual voices from the mixtures alone, using the fact that the voices are statistically independent and not Gaussian.

ICA assumes the observed vector is a linear mixture of independent sources [13]:

$$x = As,$$

where $s = (s_1, \dots, s_M)$ are statistically independent, non-Gaussian source signals, A is an unknown mixing matrix and x are the observed signals. The goal is an unmixing matrix $W \approx A^{-1}$ such that $\hat{s} = Wx$ has independent components. Why non-Gaussian? By the central limit theorem, mixtures tend to be "more Gaussian" than their sources, so maximizing the non-Gaussianity of $w^\top x$ finds a source. FastICA measures non-Gaussianity with approximations of negentropy and runs a fixed-point iteration after whitening the data [13]. Two ambiguities can never be resolved: the order of the sources and their scale (including sign).

The contrast with PCA is the point. PCA only makes the components *uncorrelated* (second-order statistics); ICA makes them *independent*, using higher-order statistics. If the sources were Gaussian, uncorrelated would already mean independent, and any rotation of the whitened data would be equally good, so ICA would fail.

```
import numpy as np
from sklearn.decomposition import FastICA

t = np.linspace(0, 8, 2000)
S = np.c_[np.sin(3 * t),                      # source 1: sine
          np.sign(np.sin(5 * t)),             # source 2: square wave
          ((2 * t) % 2) - 1]                  # source 3: sawtooth
A = np.array([[1.0, 0.5, 0.3],
              [0.6, 1.0, 0.4],
              [0.4, 0.3, 1.0]])               # unknown mixing matrix
X = S @ A.T                                   # what the sensors record

S_hat = FastICA(n_components=3, whiten="unit-variance",
                random_state=0).fit_transform(X)
corr = np.abs(np.corrcoef(S.T, S_hat.T)[:3, 3:])
print(np.round(corr, 3))                      # one entry near 1 per row
```

Each true source matches exactly one recovered component with absolute correlation 0.999 or 1.000, and the off-diagonal matches are below 0.05. Figure 5 shows that PCA, applied to the same mixtures, returns uncorrelated signals that are still mixtures.

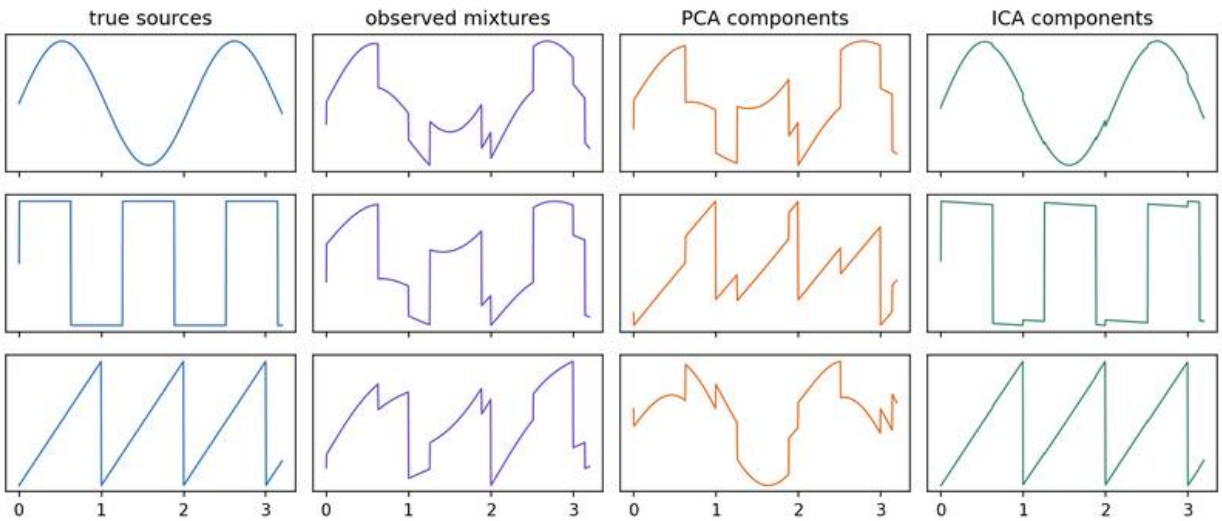


Figure 5 — Blind source separation. PCA decorrelates the mixtures but does not unmix them; FastICA recovers the sine, square and sawtooth sources up to order and scale.

Non-negative matrix factorization (NMF) is a related decomposition for non-negative data such as pixel intensities or word counts: it writes $X \approx WH$ with $W, H \geq 0$ elementwise. Because nothing can be subtracted, the factors tend to be additive "parts," as Lee and Seung showed for face images [14].

Manifold learning

In plain words. Sometimes high-dimensional data lie on a curved, low-dimensional sheet, like a rolled-up poster. Manifold learning tries to unroll the sheet, keeping points that were neighbors along the sheet close together, without being fooled by points that are close in 3-D only because the sheet is rolled up.

The Swiss roll in Figure 6 is the standard example: a two-dimensional rectangle rolled up in three dimensions. PCA flattens it by projection and puts different layers of the roll on top of each other. Neighborhood-based methods do better:

- **Isomap** [15] builds a k-nearest-neighbor graph, estimates *geodesic* distances along the manifold as shortest-path lengths in the graph, and then finds low-dimensional coordinates whose Euclidean distances match those geodesic distances (classical multidimensional scaling).
- **Locally linear embedding** (LLE) [16] writes each point as a weighted combination of its neighbors, $x_n \approx \sum_{m \in \mathcal{N}(n)} W_{nm} x_m$ with $\sum_m W_{nm} = 1$, and then finds low-dimensional points y_n that are reconstructed by the *same* weights.
- **t-SNE** [17] converts distances into neighbor probabilities p_{ij} in the input (Gaussian, with a bandwidth set by a "perplexity" parameter) and q_{ij} in the embedding (heavy-tailed Student-t), and minimizes the Kullback–Leibler divergence $\sum_{i \neq j} p_{ij} \log(p_{ij}/q_{ij})$. The heavy tails let well-separated groups spread out.

- **UMAP** [18] builds a weighted k-nearest-neighbor graph (a "fuzzy" topological representation) and optimizes a low-dimensional layout with a cross-entropy objective. It is typically faster than t-SNE and is often used for the same purpose.

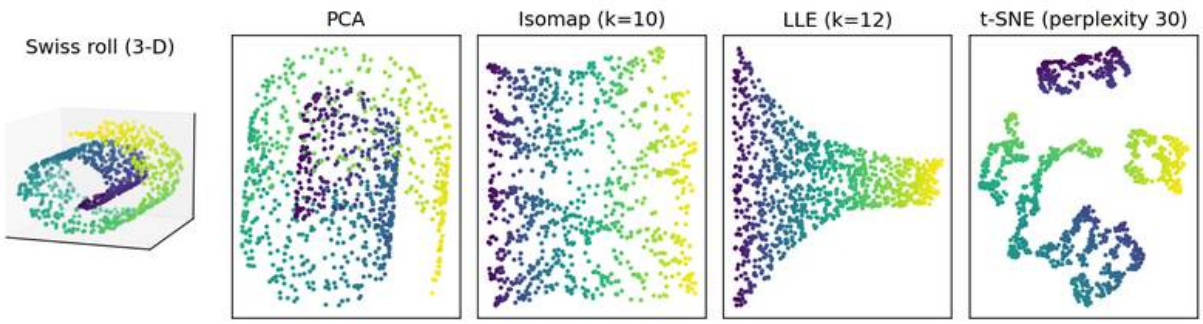


Figure 6 — The Swiss roll and four 2-D embeddings, colored by position along the roll. Isomap and LLE unroll it; PCA cannot; t-SNE keeps neighbors together but tears the continuous sheet into separate islands.

One quantitative check is **trustworthiness**: the extent to which the k nearest neighbors of each point in the embedding were also near neighbors in the input (1.0 is perfect).

```
from sklearn.datasets import make_swiss_roll
from sklearn.decomposition import PCA
from sklearn.manifold import Isomap, LocallyLinearEmbedding, TSNE, trustworthiness

X, t = make_swiss_roll(n_samples=1000, noise=0.05, random_state=0)
methods = {
    "PCA": PCA(n_components=2),
    "Isomap": Isomap(n_neighbors=10, n_components=2),
    "LLE": LocallyLinearEmbedding(n_neighbors=12, n_components=2,
                                  random_state=0),
    "t-SNE": TSNE(n_components=2, perplexity=30, init="pca",
                  random_state=0),
}
for name, method in methods.items():
    Y = method.fit_transform(X)
    print(f"{name:7s} trustworthiness (k=10): "
          f"{trustworthiness(X, Y, n_neighbors=10):.3f}")
```

PCA scores 0.905; Isomap and t-SNE 0.999; LLE 0.996. Note what the number does *not* see: t-SNE scores as high as Isomap even though Figure 6 shows it tearing the roll into islands. Local neighborhoods are preserved; global structure is not.

What an embedding picture can and cannot prove

These methods are mainly tools for looking at data. Three cautions follow from how they work [19]:

- **Cluster sizes and gaps are not meaningful by default.** t-SNE normalizes local densities, so a tight cluster and a diffuse one can look the same size, and distances between clusters may say little.
- **Results depend on settings.** Perplexity, number of neighbors, initialization and random seed all change the picture. Wattenberg, Viégas and Johnson's interactive article shows the same data producing very different t-SNE maps [19].
- **Two dimensions may not be enough.** A projection to 2-D must distort high-dimensional distances; well-separated islands in a picture are not proof of separate populations.

There is also a practical gap for classification: t-SNE and standard LLE learn coordinates for the training points only, with no natural rule for embedding a new point. Isomap, kernel PCA and UMAP provide out-of-sample transforms, but if a nonlinear embedding is part of a classifier, it must be fitted inside the training folds like any other step.

Image and signal descriptors

In plain words. Before features were learned, people designed them by hand to measure exactly the properties they believed mattered: texture, edges, shape, frequency content. They remain useful as baselines and as interpretable checks.

Theodoridis and Koutroumbas devote two chapters to transform-based and application-specific features [1]; IP 12 treats image descriptors in detail. Consider what each one keeps and what it is sensitive to:

Descriptor	Preserves	Typical sensitivity
Intensity histogram	Distribution of values	Loses spatial arrangement entirely
Co-occurrence (GLCM) statistics [20]	Pairwise intensity relations at an offset	Quantization levels and offset direction
Histogram of oriented gradients (HOG) [21]	Local edge directions in a grid of cells	Cropping, scale and alignment
Area, perimeter, compactness	Region geometry	Segmentation errors and resolution
Fourier/DCT coefficients	Frequency content	Alignment and which frequencies are kept
Wavelet coefficients	Location and scale of variation	Basis choice and boundary handling

For a quantized image I and an offset δ, a gray-level co-occurrence matrix counts pairs $(I(p), I(p+\delta))$ over pixel positions p; normalized to probabilities $P(i,j)$, Haralick's texture features include contrast $\sum_{i,j}(i-j)^2 P(i,j)$ and energy $\sum_{i,j} P(i,j)^2$ [20]. Always state the offset and the number of gray levels, because the numbers depend on both. For a segmented object with area A and perimeter P, compactness $4\pi A/P^2$ equals 1 for an ideal continuous circle and is smaller for irregular shapes; pixel-based perimeter estimates distort it for small objects. Dalal and Triggs' HOG descriptor, built from normalized histograms of gradient orientation in small cells, was introduced for pedestrian detection and became one of the most widely used hand-designed image features [21].

For audio and other 1-D signals, features are computed on short windows in which the signal is approximately stationary: spectral energy in frequency bands, zero-crossing rate, and so on. The window length trades time resolution against frequency resolution.

Code: one classifier, three feature sets

This example compares fixed feature strategies with the same classifier on the built-in digits data.

```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.feature_selection import SelectKBest, f_classif
from sklearn.linear_model import LogisticRegression

X, y = load_digits(return_X_y=True)
Xtr, Xte, ytr, yte = train_test_split(
    X, y, test_size=0.25, stratify=y, random_state=9)
transforms = {
    "all features": "passthrough",
    "24 PCA coordinates": PCA(n_components=24, random_state=9),
    "24 selected coordinates": SelectKBest(f_classif, k=24),
}
for name, transform in transforms.items():
    model = make_pipeline(
        StandardScaler(), transform, LogisticRegression(max_iter=1500))
    model.fit(Xtr, ytr)
    print(name, "test accuracy:", round(model.score(Xte, yte), 4))
```

All 64 pixels give 0.969 test accuracy, 24 principal components 0.956, and the 24 pixels with the best ANOVA F scores 0.953. Here a reduction to about a third of the dimensions costs little accuracy. Some border pixels of the digits are constant, so the F test warns about constant features; the warning reflects zero within-class variance, not leakage, and those pixels simply receive no useful score.

Modern view

Feature engineering by hand has largely given way to learned representations, but the ideas of this chapter are everywhere in modern pipelines: deep features are selected, projected, whitened, visualized and evaluated with the same tools. Here is how the survey literature maps the field.

Feature selection. Guyon and Elisseeff's introduction [3] set the vocabulary (filters, wrappers, embedded methods), discussed feature construction, ranking and validation, and stressed that a feature can be useless alone and useful in combination. Chandrashekar and Sahin [23] provide a compact survey organized along the same filter/wrapper/embedded lines. Li et al. [22]

revisit selection "from a data perspective": they group algorithms for conventional data into similarity-based, information-theoretic, sparse-learning-based and statistical methods, add methods for structured, heterogeneous and streaming data, and release an open-source repository for benchmarking. The leakage warning of Ambroise and McLachlan [7] remains the single most important practical lesson.

PCA and its descendants. Jolliffe and Cadima's review [10] explains PCA as an adaptive, data-defined technique and surveys its variants tailored to different data types. In modern deep learning, PCA survives as a diagnostic (how many directions carry most variance in a layer's activations) and as a preprocessing and initialization step, for example the PCA initialization recommended for t-SNE [26].

Dimensionality reduction for visualization. Espadoto et al. [24] ran a large quantitative survey of projection techniques: they characterized data sets, techniques and projection quality with several quantitative metrics (such as neighborhood and distance preservation), measured how the results depend on the technique and the data, and turned the observations into guidance for choosing a projection in a given context, with all code and results released as an open benchmark. Wang, Huang, Rudin and Shaposhnik [25] analyzed why t-SNE, UMAP and TriMAP behave as they do, focused on the trade-off between preserving local and global structure, and proposed PaCMAP to balance the two.

Pitfalls of 2-D embeddings. Single-cell biology, where t-SNE and UMAP are routine, produced the sharpest critiques. Kobak and Berens [26] gave a protocol for more faithful t-SNE maps: PCA initialization, a high learning rate and multi-scale similarities. Kobak and Linderman [27] showed that the better global structure often credited to UMAP comes largely from its initialization; with the same informative initialization, t-SNE preserves global structure similarly. Chari and Pachter [28] went further, arguing that extreme reduction to two dimensions inevitably distorts high-dimensional data and that such pictures should not be used for quantitative conclusions.

Variance versus usefulness, revisited. Figure 2's lesson reappears at scale in self-supervised learning. Balestriero and LeCun [29] argue that training a representation to reconstruct its input emphasizes high-variance components of the data, which are often less informative for perception tasks than low-variance details; reconstruction-trained features therefore need extra steps (such as fine-tuning) to be useful for classification. This is PCA's limitation, in a deep-learning setting.

Open problems. Selection with many correlated features and few samples remains unstable: different subsets can perform equally well, which complicates interpretation. Evaluating embeddings still lacks agreed metrics that capture both local and global fidelity [24], [25]. And for learned features, deciding which directions carry task-relevant rather than merely high-variance information is an active research question [29].

Key takeaways

- Feature selection keeps original measurements; extraction builds new coordinates; learning fits the extraction rule from data. All three decide what the classifier can see.
- Filters are fast but miss interactions and redundancy; wrappers see them but cost more and can

overfit validation data; embedded methods (lasso, trees) select during training.

- Any step that uses labels (selection, LDA, choosing the number of features) must be refitted inside each cross-validation fold. Selecting on the full data can report 98% accuracy on pure noise.
- Fisher ratio, $J = \mathrm{tr}(S_W^{-1} S_B)$, Bhattacharyya distance and mutual information score features; subsets must be scored jointly because of redundancy.
- PCA keeps maximum variance (eigenvectors of the covariance, computed by SVD); whitening equalizes variances. Variance is not class information.
- Fisher LDA solves $S_B v = \lambda S_W v$ and gives at most $K - 1$ directions. Kernel PCA, ICA and NMF extend linear extraction to nonlinear, independent or non-negative structure.
- Isomap, LLE, t-SNE and UMAP are mainly visualization tools: their pictures depend on settings, and gaps between clusters are not proof of separate populations.

Exercises

1. **Same histogram, different image.** Construct two 4×4 binary images with identical intensity histograms but different texture. Which descriptor from the table distinguishes them?

   Hint

   Use eight 1-pixels arranged as a checkerboard versus the left half filled. The histograms are identical. A co-occurrence matrix with horizontal offset $(0, 1)$ distinguishes them: the checkerboard has only unequal neighbor pairs, the half-filled image mostly equal ones.

2. **The $K - 1$ limit.** Prove that the between-class scatter S_B of K classes has rank at most $K - 1$. What does this imply for LDA on a two-class problem?

   Hint

   $S_B = \sum_c P_c d_c d_c^\top$ with $d_c = \mu_c - \mu$, and $\sum_c P_c d_c = 0$, so the K vectors d_c span at most $K - 1$ dimensions. With two classes, LDA gives a single direction, $v \propto S_W^{-1}(\mu_1 - \mu_2)$.

3. **When 99% of the variance is not enough.** Describe a realistic imaging setup in which keeping 99% of the PCA variance discards the class signal. How would you detect the problem, and what would you do?

   Hint

   Large independent lighting or pose changes plus a small, consistent defect mark (Figure 2 in disguise). Detect it by comparing cross-validated accuracy with and without PCA, or by checking class separation (Fisher ratio) per component. Use a supervised projection, normalize the nuisance (for example illumination) first, or skip PCA.

4. **The XOR trap.** Generate x_1, x_2 uniform on $[-1, 1]$ and eight irrelevant noise features, with $y = \mathbf{1}[x_1 > 0] \oplus \mathbf{1}[x_2 > 0]$. Rank all ten features with `f_classif` and with `mutual_info_classif`. Then run sequential forward selection with a decision tree. Which methods find x_1 and x_2?

   Hint

   Both univariate filters give x_1 and x_2 scores near zero, like the noise features, because each alone is independent of y. A wrapper that adds features one at a time can also struggle at the first step, for the same reason; a tree-based wrapper that evaluates pairs, or backward elimination starting from all features, finds them.

5. **Leakage, quantified.** In the selection-bias code, vary the number of noise features (100, 1,000, 5,000) and the number of samples (30, 60, 200). How does the "wrong" accuracy change? Explain the trend.

   Hint

   The optimism grows with the number of candidate features and shrinks with the number of samples: with more noise features, more of them correlate with the labels by chance in a small sample, and selecting the best 20 picks exactly those.

6. **Reading a t-SNE map.** Run t-SNE on the digits data with perplexity 5, 30 and 100, and with two random seeds each. Which features of the plots are stable (which digits are neighbors?) and which change (cluster sizes, gaps, positions)? What would you be willing to claim from one such plot?

   Hint

   Membership of local neighborhoods is quite stable; absolute positions, cluster areas and between-cluster distances are not. A defensible claim is "these points are locally similar in the input space," not "these two groups are far apart" or "this group is twice as diverse."

References

1. S. Theodoridis and K. Koutroumbas, *Pattern Recognition*, 4th ed., Academic Press, 2009, Chapters 5–7. publisher page
2. C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006, Sections 4.1.4–4.1.6 and 12.1–12.4. official book page and PDF
3. I. Guyon and A. Elisseeff, "An Introduction to Variable and Feature Selection," *Journal of Machine Learning Research*, vol. 3, pp. 1157–1182, 2003. JMLR
4. R. Kohavi and G. H. John, "Wrappers for Feature Subset Selection," *Artificial Intelligence*, vol. 97, pp. 273–324, 1997. doi:10.1016/S0004-3702(97)00043-X
5. R. Tibshirani, "Regression Shrinkage and Selection via the Lasso," *Journal of the Royal Statistical Society, Series B*, vol. 58, no. 1, pp. 267–288, 1996. doi:10.1111/j.2517-6161.1996.tb02080.x

6. H. Peng, F. Long, and C. Ding, "Feature Selection Based on Mutual Information: Criteria of Max-Dependency, Max-Relevance, and Min-Redundancy," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, no. 8, pp. 1226–1238, 2005. doi:10.1109/TPAMI.2005.159
7. C. Ambroise and G. J. McLachlan, "Selection Bias in Gene Extraction on the Basis of Microarray Gene-Expression Data," *Proceedings of the National Academy of Sciences*, vol. 99, no. 10, pp. 6562–6566, 2002. doi:10.1073/pnas.102102699
8. K. Pearson, "On Lines and Planes of Closest Fit to Systems of Points in Space," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 2, no. 11, pp. 559–572, 1901. doi:10.1080/14786440109462720
9. H. Hotelling, "Analysis of a Complex of Statistical Variables into Principal Components," *Journal of Educational Psychology*, vol. 24, no. 6, pp. 417–441, 1933. doi:10.1037/h0071325
10. I. T. Jolliffe and J. Cadima, "Principal Component Analysis: A Review and Recent Developments," *Philosophical Transactions of the Royal Society A*, vol. 374, 20150202, 2016. doi:10.1098/rsta.2015.0202
11. R. A. Fisher, "The Use of Multiple Measurements in Taxonomic Problems," *Annals of Eugenics*, vol. 7, no. 2, pp. 179–188, 1936. doi:10.1111/j.1469-1809.1936.tb02137.x
12. B. Schölkopf, A. Smola, and K.-R. Müller, "Nonlinear Component Analysis as a Kernel Eigenvalue Problem," *Neural Computation*, vol. 10, no. 5, pp. 1299–1319, 1998. doi:10.1162/089976698300017467
13. A. Hyvärinen and E. Oja, "Independent Component Analysis: Algorithms and Applications," *Neural Networks*, vol. 13, no. 4–5, pp. 411–430, 2000. doi:10.1016/S0893-6080(00)00026-5
14. D. D. Lee and H. S. Seung, "Learning the Parts of Objects by Non-Negative Matrix Factorization," *Nature*, vol. 401, pp. 788–791, 1999. doi:10.1038/44565
15. J. B. Tenenbaum, V. de Silva, and J. C. Langford, "A Global Geometric Framework for Nonlinear Dimensionality Reduction," *Science*, vol. 290, no. 5500, pp. 2319–2323, 2000. doi:10.1126/science.290.5500.2319
16. S. T. Roweis and L. K. Saul, "Nonlinear Dimensionality Reduction by Locally Linear Embedding," *Science*, vol. 290, no. 5500, pp. 2323–2326, 2000. doi:10.1126/science.290.5500.2323
17. L. van der Maaten and G. Hinton, "Visualizing Data using t-SNE," *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 2008. JMLR
18. L. McInnes, J. Healy, and J. Melville, "UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction," arXiv:1802.03426, 2018. arXiv
19. M. Wattenberg, F. Viégas, and I. Johnson, "How to Use t-SNE Effectively," *Distill*, 2016. doi:10.23915/distill.00002
20. R. M. Haralick, K. Shanmugam, and I. Dinstein, "Textural Features for Image Classification," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-3, no. 6, pp. 610–621, 1973. doi:10.1109/TSMC.1973.4309314
21. N. Dalal and B. Triggs, "Histograms of Oriented Gradients for Human Detection," *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, vol. 1, pp. 886–893, 2005. doi:10.1109/CVPR.2005.177
22. J. Li, K. Cheng, S. Wang, F. Morstatter, R. P. Trevino, J. Tang, and H. Liu, "Feature Selection: A Data Perspective," *ACM Computing Surveys*, vol. 50, no. 6, 2018. doi:10.1145/3136625
23. G. Chandrashekar and F. Sahin, "A Survey on Feature Selection Methods," *Computers & Electrical Engineering*, vol. 40, no. 1, pp. 16–28, 2014. doi:10.1016/j.compeleceng.2013.11.024

24. M. Espadoto, R. M. Martins, A. Kerren, N. S. T. Hirata, and A. C. Telea, “Toward a Quantitative Survey of Dimension Reduction Techniques,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 27, no. 3, pp. 2153–2173, 2021. doi:10.1109/TVCG.2019.2944182
25. Y. Wang, H. Huang, C. Rudin, and Y. Shaposhnik, “Understanding How Dimension Reduction Tools Work: An Empirical Approach to Deciphering t-SNE, UMAP, TriMAP, and PaCMAP for Data Visualization,” *Journal of Machine Learning Research*, vol. 22, 2021. arXiv:2012.04456
26. D. Kobak and P. Berens, “The Art of Using t-SNE for Single-Cell Transcriptomics,” *Nature Communications*, vol. 10, 2019. doi:10.1038/s41467-019-13056-x
27. D. Kobak and G. C. Linderman, “Initialization Is Critical for Preserving Global Data Structure in Both t-SNE and UMAP,” *Nature Biotechnology*, vol. 39, pp. 156–157, 2021. doi:10.1038/s41587-020-00809-z
28. T. Chari and L. Pachter, “The Specious Art of Single-Cell Genomics,” *PLOS Computational Biology*, vol. 19, e1011288, 2023. doi:10.1371/journal.pcbi.1011288
29. R. Balestriero and Y. LeCun, “Learning by Reconstruction Produces Uninformative Features for Perception,” arXiv:2402.11337, 2024. arXiv

PR 06 • Template Matching and Structured Recognition

Match patterns with correlation, normalized cross-correlation and dynamic time warping, describe structure with strings, grammars and graphs, and model context with graphical models, HMMs (forward algorithm, Viterbi) and CRFs.

Online edition: https://blog.shyandram.dev/en/learn/pattern-recognition/templates-structured-models/

What you'll learn

- How template matching works with sum of squared differences, cross-correlation and normalized cross-correlation (NCC), and which nuisance changes each one tolerates.
- How dynamic time warping (DTW) aligns sequences that unfold at different speeds, and how to constrain it.
- Syntactic and structural pattern recognition: describing patterns as strings, grammars and graphs, and comparing them with edit distance and graph matching.
- How directed and undirected graphical models factorize a joint distribution, and why that makes inference efficient.
- Hidden Markov models: the forward algorithm, forward–backward smoothing, Viterbi decoding and Baum–Welch learning, with tested code.
- Conditional random fields as the discriminative counterpart of HMMs, and where all of these ideas live on in 2026.

The big picture

The classifiers of PR 03 and PR 04 look at one fixed-length feature vector at a time and decide its class alone. Many recognition problems are not like that:

- To find a part in an image, you slide a reference picture over the image and look for the best match (**templates**).
- A spoken word or a signature unfolds in time, and two people produce it at different speeds (**alignment**).
- A character, a molecule or a road network is best described by its parts and how they connect (**structure**).
- An isolated image patch might be road or roof; a blurry handwritten letter might be "c" or "e". Neighboring regions or letters resolve the ambiguity (**context**).

Explain like I'm five

Reading messy handwriting, you don't stare at each letter alone. You compare its shape with letters you know (templates), you allow some letters to be stretched or squashed (alignment), you know

how strokes join to make a letter (structure), and you use the neighboring letters to guess the word (context). This chapter turns each of these habits into an algorithm.

A common thread runs through the whole chapter: **dynamic programming**. DTW, edit distance, the HMM forward algorithm, Viterbi decoding and CRF inference are all the same trick. Break a huge search over all alignments or label sequences into small, reusable subproblems, and the cost drops from exponential to polynomial.

One caution belongs at the start. Context helps only when its assumptions match reality. A strong "neighbors agree" rule can erase small objects or brief events, so more context is not automatically better. The textbook background is Theodoridis and Koutroumbas, Chapters 8 and 9 [1], and Bishop, Sections 8.1–8.4 and 13.1–13.3 [2].

Template matching

In plain words. Keep a small reference image, the template. Slide it over the scene, and at every position measure how similar the patch underneath is. The best-scoring position is the match. The interesting question is *how* to measure similarity so that irrelevant changes, such as a brighter or dimmer exposure, do not fool you.

Three similarity measures

Write the template as a vector t of its M pixel values and the image patch under it as x, also of length M.

Sum of squared differences (SSD). $\mathrm{SSD}(x,t) = \|x-t\|^2 = \sum_i (x_i - t_i)^2$. Small is good. It is the natural distance, but any brightness offset or contrast change in the scene increases it.

Cross-correlation. Expanding the square, $\|x-t\|^2 = \|x\|^2 - 2x^\top t + \|t\|^2$. The template energy $\|t\|^2$ is constant, so if the patch energy $\|x\|^2$ were roughly constant too, minimizing SSD would be the same as maximizing the correlation $x^\top t$. In practice $\|x\|^2$ varies a lot, and plain correlation is drawn to bright regions.

Normalized cross-correlation (NCC). Subtract the means and divide by the norms:

$$\mathrm{NCC}(x,t) = \frac{(x-\bar{x}\mathbf{1})^\top (t-\bar{t}\mathbf{1})}{\|x-\bar{x}\mathbf{1}\|\,\|t-\bar{t}\mathbf{1}\|},$$

where $\bar{x}$ and $\bar{t}$ are the mean pixel values of the patch and the template and $\mathbf{1}$ is the all-ones vector. NCC is the cosine of the angle between the mean-subtracted vectors, so it lies in $[-1,1]$, and it equals 1 exactly when $x = a\,t + b\mathbf{1}$ for some $a > 0$ and any b. It is therefore invariant to positive changes of contrast (a) and to brightness offsets (b). It is undefined for a constant template or patch (zero denominator); implementations return a convention such as 0 or NaN. NCC is the "matching by correlation" of IP 13.

Computing it efficiently

Evaluating NCC at every position of an $N \times N$ image with an $m \times m$ template costs on the order of $N^2 m^2$ operations. The numerator is a correlation, which can be computed for all positions at once with the FFT (the correlation theorem, IP 04). The local means and energies in the denominator vary from position to position; Lewis showed how to get them cheaply from running-sum tables (integral images) of the image and of its square, which makes NCC nearly as fast as plain correlation [3].

```
import cv2
import numpy as np
from skimage import data

img = data.camera().astype(np.float32)          # 512 x 512 grayscale
r0, c0, h, w = 300, 200, 40, 40
template = img[r0:r0 + h, c0:c0 + w].copy()

def ncc_at(window, t):
    a, b = window - window.mean(), t - t.mean()
    return float((a * b).sum() / (np.linalg.norm(a) * np.linalg.norm(b)))

# The scene is captured again under different exposure: darker and lower contrast.
scene = 0.5 * img + 20.0
ncc = cv2.matchTemplate(scene, template, cv2.TM_CCOEFF_NORMED)
ssd = cv2.matchTemplate(scene, template, cv2.TM_SQDIFF)
r, c = np.unravel_index(ncc.argmax(), ncc.shape)
print("NCC best (row, col):", (int(r), int(c)), "score", round(float(ncc.max()), 4))
r, c = np.unravel_index(ssd.argmin(), ssd.shape)
print("SSD best (row, col):", (int(r), int(c)))
print("hand-computed NCC at true location:",
      round(ncc_at(scene[r0:r0 + h, c0:c0 + w], template), 4))
```

NCC finds the template at its true position (300, 200) with a score of 1.0, and the hand-computed value agrees with OpenCV's `TM_CCOEFF_NORMED`. SSD, fooled by the exposure change, picks (130, 413), a patch of sky whose darkened intensities (mean about 129) are closer to the template's original values (mean about 143) than the darkened true patch is (mean about 92).

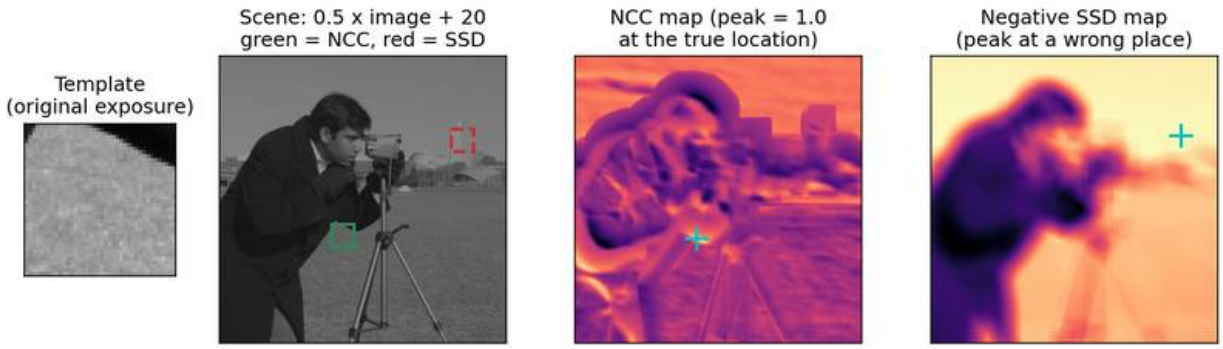


Figure 1 — Template matching under an exposure change. NCC is invariant to the gain 0.5 and offset 20 and finds the true location; SSD does not. Generated by scripts/figures/pr_06_templates_structured_models_extra.py.

What templates cannot handle

NCC removes brightness and contrast, nothing else. Rotation, scale change, perspective, occlusion and non-rigid deformation all lower the score. The classical remedies are to store a family of templates (rotated and scaled copies), to search over an image pyramid, or to match small parts and then check their geometric arrangement, which leads to the structural methods below. Template matching remains the right tool when appearance is tightly controlled, as in industrial inspection, and it is still a core step in tracking and image registration [4].

Dynamic time warping

In plain words. Two people say the same word, one slowly, one quickly. Comparing the recordings sample by sample fails because the sounds do not line up in time. DTW stretches and compresses the time axis of one sequence to line it up with the other as well as possible, then measures the remaining difference.

The recursion

Let $x = (x_1, \dots, x_n)$ and $t = (t_1, \dots, t_m)$ be two sequences, and $d(x_i, t_j)$ a local cost such as $|x_i - t_j|$ or $\|x_i - t_j\|^2$ for vector-valued samples. A **warping path** is a sequence of index pairs $(i_1, j_1), \dots, (i_L, j_L)$ that starts at $(1,1)$, ends at (n,m), and moves by one of the steps $(1,0)$, $(0,1)$ or $(1,1)$ at a time, so that it never goes backward in either sequence. DTW is the smallest total cost over all warping paths. It is computed by filling a table:

$$D(i,j) = d(x_i, t_j) + \min\{D(i-1,j),\ D(i,j-1),\ D(i-1,j-1)\},$$

with $D(0,0) = 0$ and $D(i,0) = D(0,j) = \infty$ for $i, j \geq 1$. Then $\mathrm{DTW}(x,t) = D(n,m)$. Each cell needs only its three neighbors, so the cost is $O(nm)$ time. Tracing back which neighbor gave the minimum recovers the optimal alignment. This is the algorithm Sakoe and Chiba optimized for spoken-word recognition [5].

Constraints and caveats

An unconstrained path can match a single sample of one sequence to a long stretch of the other ("pathological warping"), which lets unrelated sequences look similar. Two standard fixes are:

- a **global constraint**, such as the Sakoe–Chiba band $|i - j| \leq w$, which forbids cells far from the diagonal and also reduces the cost to $O(nw)$ [5];
- **slope constraints** or step weights that penalize long horizontal or vertical runs [5].

The band width w must allow the real timing differences. In the code and in Figure 2, a band of 3 samples is narrower than the true shift of about 6 samples, so the banded DTW cannot fully align the two sine waves.

Two further facts matter in practice. DTW is not a metric: it can violate the triangle inequality, so data structures that assume a metric need care. And sequences are usually **z-normalized** (zero mean, unit variance) before comparison, so that DTW compares shapes rather than offsets and scales.

```
import numpy as np

def dtw(x, t, band=None):
    """DTW distance between 1-D sequences x (length n) and t (length m)."""
    n, m = len(x), len(t)
    D = np.full((n + 1, m + 1), np.inf)
    D[0, 0] = 0.0
    for i in range(1, n + 1):
        lo, hi = 1, m
        if band is not None:                    # Sakoe-Chiba band
            lo, hi = max(1, i - band), min(m, i + band)
        for j in range(lo, hi + 1):
            cost = abs(x[i - 1] - t[j - 1])
            D[i, j] = cost + min(D[i - 1, j], D[i, j - 1], D[i - 1, j - 1])
    return D[n, m]

s = np.linspace(0, 2 * np.pi, 60)
a = np.sin(s)
b = np.sin(s - 0.6)                             # same shape, shifted in time
c = 0.25 * np.sign(np.sin(3 * s))               # a different shape
print("lock-step a-b:", round(np.abs(a - b).sum(), 2),
      " a-c:", round(np.abs(a - c).sum(), 2))
print("DTW       a-b:", round(dtw(a, b), 2),
      " a-c:", round(dtw(a, c), 2))
print("DTW a-b with band 3:", round(dtw(a, b, band=3), 2))
```

Both comparisons use the same local cost $|x_i - t_j|$. Sample by sample ("lock-step"), the shifted copy b costs 22.76 and the different shape c costs 35.18. DTW reduces the shifted copy to 5.05 while the different shape stays at 34.57: warping absorbs timing differences, not shape differences. With a band of 3 samples, the cost of the shifted copy rises to 12.76 because the band is too narrow for a shift of about 6 samples.

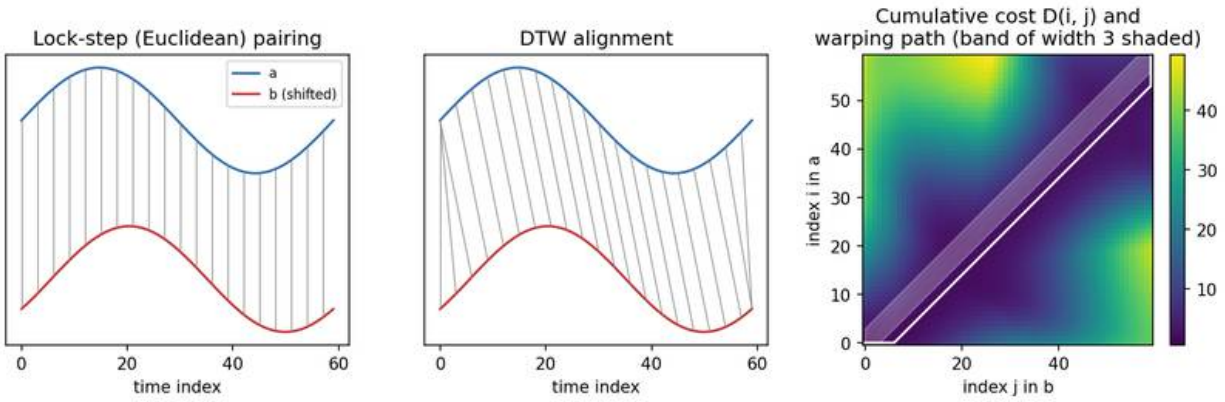


Figure 2 — Dynamic time warping. Lock-step comparison pairs equal time indices; DTW pairs corresponding parts of the two curves. In the cost matrix, the optimal path runs about 6 samples off the diagonal, outside a band of width 3.

A 1-nearest-neighbor classifier with DTW distance was for many years a strong, hard-to-beat baseline for time-series classification [16]. Searching for the nearest neighbor under DTW in very long series became practical once cheap lower bounds and early abandoning of hopeless candidates were combined [6].

Syntactic and structural pattern recognition

In plain words. Some patterns are best described not by a list of numbers but by their parts and how the parts fit together, the way a sentence is words arranged by grammar. Structural pattern recognition describes a pattern as a string, tree or graph of primitive parts, and compares or parses those structures.

Strings and grammars

Choose a small alphabet of **primitives**, for example the four directions of a step along an object's boundary. Then a shape becomes a string. A 4-direction **chain code** writes 0 for a step right, 1 up, 2 left and 3 down while tracing the boundary (see IP 12). Figure 3 shows three outlines and their codes.

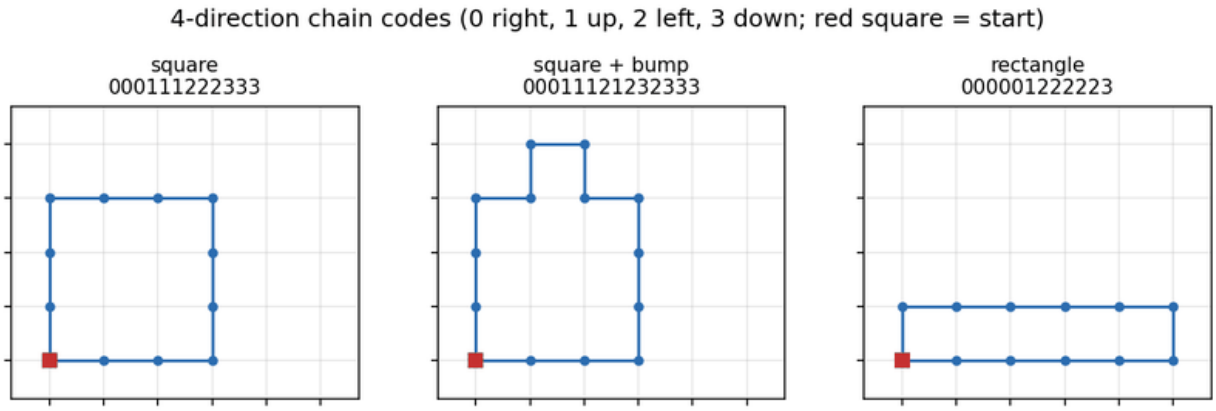


Figure 3 — Shapes as strings. Each outline is traced counter-clockwise from the red start point, writing one symbol per boundary step.

In **syntactic** pattern recognition each class is described by a formal grammar: a set of rewrite rules that generates exactly the valid strings of that class. Recognition is **parsing**: a string belongs to a class if that class's grammar can generate it. For example, the rule set $S \to 0S2 \mid 1$ generates strings $0^k 1\, 2^k$, a "class" in which every rightward run is matched by an equally long leftward run. Grammars can express recursive structure (nested parts, repeated sub-shapes) that fixed-length feature vectors cannot. Their weaknesses are that real data are noisy, so a single wrong primitive makes a string ungrammatical, and that grammars are hard to learn from examples. Stochastic grammars, which attach probabilities to rules, address the first problem; a hidden Markov model, below, is the simplest stochastic grammar of all [1].

Edit distance

A more forgiving approach compares strings by the cheapest way to turn one into the other. The **edit distance** (Levenshtein distance) counts the minimum number of insertions, deletions and substitutions. Wagner and Fischer gave the dynamic program [7]:

$$E(i,j) = \min\{E(i-1,j)+1,\; E(i,j-1)+1,\; E(i-1,j-1)+\mathbf{1}[s_i \neq u_j]\},$$

with $E(i,0)=i$ and $E(0,j)=j$, for strings s (length n) and u (length m); the answer is $E(n,m)$. The structure is exactly that of DTW, with the operations "insert", "delete" and "substitute" instead of "stretch". Costs can be weighted, for example to make substituting neighboring directions cheaper than opposite ones.

```
import numpy as np

def edit_distance(s, t):
    """Levenshtein distance: unit-cost insertions, deletions, substitutions."""
    n, m = len(s), len(t)
    D = np.zeros((n + 1, m + 1), dtype=int)
    D[:, 0] = np.arange(n + 1)
    D[0, :] = np.arange(m + 1)
```

```
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            sub = 0 if s[i - 1] == t[j - 1] else 1
            D[i, j] = min(D[i - 1, j] + 1,          # delete s[i-1]
                          D[i, j - 1] + 1,          # insert t[j-1]
                          D[i - 1, j - 1] + sub)    # match or substitute
    return D[n, m]

# 4-direction chain codes (0=right, 1=up, 2=left, 3=down) of three outlines
square = "000111222333"           # 3 x 3 square, traced counter-clockwise
square_bump = "00011121232333"   # the same square with a one-pixel bump on top
rectangle = "000001222223"       # a flat 5 x 1 rectangle
for name, code in [("square + bump", square_bump), ("rectangle", rectangle)]:
    print(f"edit distance(square, {name}) = {edit_distance(square, code)}")
```

The bumped square is 2 edits from the square (insert one "up" and one "down"); the flat rectangle is 4 edits away. A nearest-neighbor classifier over strings with this distance is the structural analogue of k-NN. Chain codes depend on the starting point and on rotation; normalizing them (for example by choosing the starting point that gives the smallest string, and by using differences between successive directions) is part of the representation design.

Graphs and graph matching

When parts connect in more than a chain, a pattern becomes an **attributed graph**: nodes are parts (strokes, regions, atoms) with attributes such as size or color, and edges are relations (touching, above, bonded). Recognition then means **graph matching**: finding a correspondence between the nodes of two graphs. Exact matching (graph or subgraph isomorphism) tolerates no differences and is computationally hard in general; error-tolerant matching, such as **graph edit distance**, generalizes string edit distance with node and edge insertions, deletions and substitutions, at an even higher computational price. Conte, Foggia, Sansone and Vento surveyed three decades of graph matching in pattern recognition, from exact algorithms to inexact ones and their applications [8], and Foggia, Percannella and Vento updated the picture with a decade of graph matching and graph learning methods [9].

Graphical models describe dependence

In plain words. A graphical model is a drawing of which variables directly influence which. The drawing tells you how a huge joint probability breaks into small local pieces, and small pieces are what make computation possible.

A **directed** acyclic graph (a Bayesian network) factorizes the joint distribution of variables $z_1, \dots, z_M$ as

$$p(z_1, \dots, z_M) = \prod_{i=1}^{M} p(z_i \mid \mathrm{pa}(z_i)),$$

where $\mathrm{pa}(z_i)$ are the parents of z_i in the graph [2]. An **undirected** graph (a Markov random field) uses nonnegative potential functions ψ_C on its cliques C (fully connected subsets of nodes):

$$p(z) = \frac{1}{Z} \prod_C \psi_C(z_C), \qquad Z = \sum_z \prod_C \psi_C(z_C).$$

A potential is a compatibility score, not necessarily a probability. The **partition function** Z normalizes the product; it sums over all joint configurations and is often the hardest quantity to compute.

Conditional independence is read from the graph by its rules (d-separation for directed graphs, graph separation for undirected ones), not from the mere absence of an arrow. A classic surprise: two independent causes become dependent once you condition on their common effect ("explaining away") [2]. **Factor graphs** draw variables and factors as two kinds of nodes and make local message passing explicit; on trees, message passing (the sum-product algorithm) computes all marginals exactly, and the HMM algorithms below are its special case on a chain [2].

For image labeling, a common undirected model is an **energy** with unary data terms and pairwise smoothness terms:

$$E(z; x) = \sum_i U_i(z_i; x) + \lambda \sum_{(i,j)\in\mathcal{E}} \mathbf{1}[z_i \neq z_j],$$

where z_i is the label of pixel i, U_i is the cost of that label given the image x (for example the negative log-probability from a pixel classifier), $\mathcal{E}$ is the set of neighboring pixel pairs, and $\lambda \geq 0$ is the price of a label change between neighbors (a Potts model). The distribution is $p(z \mid x) \propto \exp(-E(z; x))$. It encourages neighboring labels to agree, which cleans up noise but can also erase thin structures if λ is too large. Exact minimization is hard on general grids, and PR 08 covers approximate inference.

Hidden Markov models

In plain words. A system moves between a few hidden states (silence, vowel, consonant; walking, running) and at each time step emits an observation that depends only on the current state. You see the observations, never the states. An HMM lets you compute how likely an observation sequence is, which hidden states probably produced it, and how to learn the model's probabilities from data.

The model

An HMM with K hidden states has initial probabilities $\pi_j = p(z_1 = j)$, a **transition matrix** $A_{ij} = p(z_t = j \mid z_{t-1} = i)$, and **emission** probabilities (or densities) $B_j(x_t) = p(x_t \mid z_t = j)$. The joint probability of states $z_{1:T} = (z_1, \ldots, z_T)$ and observations $x_{1:T}$ is

$$p(z_{1:T}, x_{1:T}) = \pi_{z_1} B_{z_1}(x_1) \prod_{t=2}^{T} A_{z_{t-1}, z_t} B_{z_t}(x_t).$$

Two assumptions are built in: the hidden state is first-order Markov (the future depends on the past only through the present state), and each observation depends only on the current state. The observations themselves are *not* Markov: integrating out the hidden states creates long-range dependence among them. Rabiner's tutorial, still the standard introduction, organizes HMM computation around three problems [10]: **evaluation** (compute $p(x_{1:T})$), **decoding** (find the hidden states) and **learning** (estimate π, A, B). Figure 4 contrasts the HMM's directed graph with the CRF discussed later.

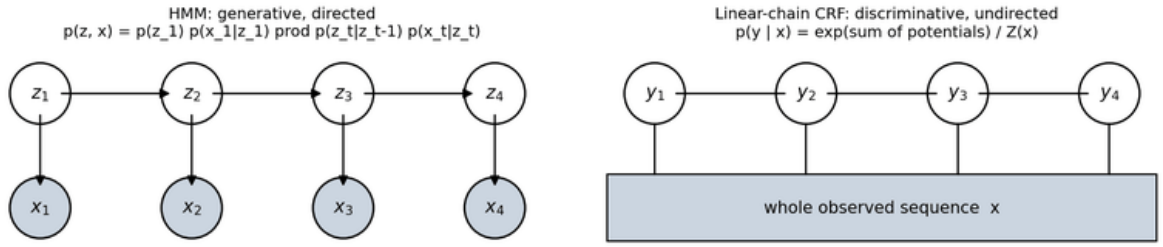


Figure 4 — HMM versus linear-chain CRF. The HMM is a directed generative model of states and observations; the CRF is an undirected model of labels conditioned on the whole observation sequence.

Evaluation: the forward algorithm

Summing the joint over all K^T state sequences is hopeless for realistic T. Define the **forward variable** $\alpha_t(j) = p(x_{1:t}, z_t = j)$. Because z_t summarizes everything the past can tell about the future, it satisfies

$$\alpha_1(j) = \pi_j B_j(x_1), \qquad \alpha_t(j) = B_j(x_t) \sum_{i=1}^{K} \alpha_{t-1}(i) A_{ij},$$

and $p(x_{1:T}) = \sum_j \alpha_T(j)$. The cost is $O(TK^2)$ instead of $O(TK^T)$. Since α_t is a product of many probabilities, it underflows for long sequences. The standard fix is to normalize at each step: store $c_t = \sum_j \tilde{\alpha}_t(j)$ and $\hat{\alpha}_t = \tilde{\alpha}_t / c_t$, where $\tilde{\alpha}_t$ is the unnormalized update. Then $\hat{\alpha}_t(j) = p(z_t = j \mid x_{1:t})$ is exactly the **filtered** state probability, $c_t = p(x_t \mid x_{1:t-1})$, and

$$\log p(x_{1:T}) = \sum_{t=1}^{T} \log c_t.$$

Evaluation gives a classifier immediately: train one HMM per class (one per spoken word, say) and assign a new sequence to the class whose HMM gives it the highest likelihood, times the prior.

Smoothing: the forward–backward algorithm

Filtering computes $p(z_t \mid x_{1:t})$ from past and present only; it is what an online system can know at time t. **Smoothing** computes $p(z_t \mid x_{1:T})$, using later observations as well. A backward recursion

$\beta_t(i) = p(x_{t+1:T} \mid z_t = i)$,

$$\beta_T(i) = 1, \qquad \beta_t(i) = \sum_{j=1}^{K} A_{ij}\, B_j(x_{t+1})\, \beta_{t+1}(j),$$

combines with the forward variable to give $\gamma_t(j) = p(z_t = j \mid x_{1:T}) \propto \alpha_t(j)\beta_t(j)$ [2], [10]. With scaling, divide each backward step by c_{t+1}, and then $\gamma_t = \hat{\alpha}_t \odot \hat{\beta}_t$ (elementwise product) exactly.

Decoding: the Viterbi algorithm

Often we want the single most probable *sequence* of states, $\arg\max_{z_{1:T}} p(z_{1:T} \mid x_{1:T})$. Viterbi's algorithm [11] replaces the sum in the forward recursion by a maximum. In log space,

$$\delta_1(j) = \log \pi_j + \log B_j(x_1), \qquad \delta_t(j) = \log B_j(x_t) + \max_i \{\delta_{t-1}(i) + \log A_{ij}\},$$

where $\delta_t(j)$ is the log-probability of the best path that ends in state j at time t. Store the maximizing predecessor $\psi_t(j) = \arg\max_i\{\cdots\}$, pick the best final state, and trace the pointers backward. The cost is again $O(TK^2)$.

The most probable path is *not* the same as choosing, at each time, the state with the highest smoothed probability $\gamma_t(j)$. The per-step choice maximizes the expected number of correct states but can string together states that the model says cannot follow each other. The code shows this on a model whose states must cycle $0 \to 1 \to 2 \to 0$, so the transition $0 \to 2$ has probability zero.

```
import itertools
import numpy as np

pi = np.array([1 / 3, 1 / 3, 1 / 3])            # initial state probabilities
A = np.array([[0.5, 0.5, 0.0],                  # cyclic model: 0 -> 1 -> 2 -> 0
              [0.0, 0.5, 0.5],
              [0.5, 0.0, 0.5]])
B = np.array([[0.7, 0.3],                       # P(symbol | state)
              [0.5, 0.5],
              [0.3, 0.7]])
obs = np.array([1, 0, 1, 0, 0])
T, K = len(obs), len(pi)

# Forward pass with scaling: alpha[t] = p(z_t | x_1..t), c[t] = p(x_t | x_1..t-1).
alpha, c = np.zeros((T, K)), np.zeros(T)
alpha[0] = pi * B[:, obs[0]]
c[0] = alpha[0].sum(); alpha[0] /= c[0]
for t in range(1, T):
    alpha[t] = B[:, obs[t]] * (alpha[t - 1] @ A)
    c[t] = alpha[t].sum(); alpha[t] /= c[t]
log_lik = np.log(c).sum()

# Backward pass (scaled) and smoothed marginals gamma[t] = p(z_t | x_1..T).
beta = np.ones((T, K))
for t in range(T - 2, -1, -1):
    beta[t] = A @ (B[:, obs[t + 1]] * beta[t + 1]) / c[t + 1]
```

```
gamma = alpha * beta

# Viterbi in log space (log 0 = -inf marks forbidden transitions).
with np.errstate(divide="ignore"):
    lA, lB, lpi = np.log(A), np.log(B), np.log(pi)
delta = lpi + lB[:, obs[0]]
back = []
for t in range(1, T):
    cand = delta[:, None] + lA
    back.append(cand.argmax(axis=0))
    delta = cand.max(axis=0) + lB[:, obs[t]]
path = [int(delta.argmax())]
for b in reversed(back):
    path.append(int(b[path[-1]]))
path = path[::-1]

# Brute force over all K**T paths as an independent check.
def joint(z):
    p = pi[z[0]] * B[z[0], obs[0]]
    for t in range(1, T):
        p *= A[z[t - 1], z[t]] * B[z[t], obs[t]]
    return p
paths = list(itertools.product(range(K), repeat=T))
assert np.isclose(log_lik, np.log(sum(joint(z) for z in paths)))
assert tuple(path) == max(paths, key=joint)

marginal = gamma.argmax(axis=1)
print("log p(x):", round(log_lik, 4))
print("Viterbi path:         ", path, " p =", round(joint(path), 5))
print("per-step argmax gamma:", marginal.tolist(),
      " p =", round(joint(marginal), 5))
```

The scaled forward pass gives $\log p(x) = -3.5449$, and both the likelihood and the Viterbi path agree with brute-force enumeration of all $3^5 = 243$ paths. Viterbi returns $1 \to 1 \to 2 \to 0 \to 0$ with joint probability 0.00179. Picking the most probable state at each step gives $2, 0, 2, 0, 0$, which contains the forbidden jump $0 \to 2$ and has probability exactly zero under the model. Which decoder is right depends on the loss: per-step accuracy favors the marginals; a valid, coherent sequence needs Viterbi.

Figure 5 shows the same computations on a longer simulated sequence: the filtered probabilities react only to the past, the smoothed ones use the whole sequence and are more confident, and the Viterbi path tracks the true states except where observations are ambiguous.

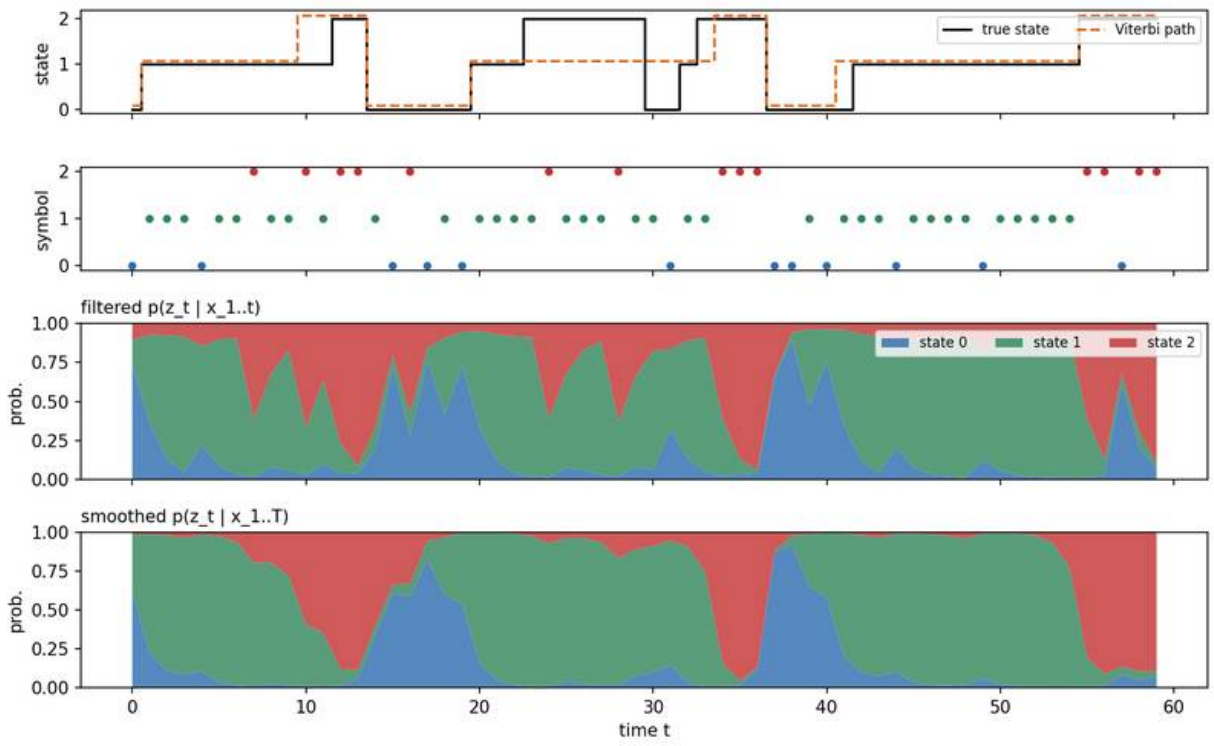


Figure 5 — A simulated 3-state cyclic HMM. Filtering uses observations up to time t; smoothing uses the whole sequence. On this run the Viterbi path matches the true state at 78% of time steps, the per-step smoothed choice at 77%, and the per-step filtered choice at 73%.

A note on evaluation. If a system must make decisions online, report filtered (or fixed-lag) performance. A smoothed result uses observations that would not yet exist at decision time, and it overstates what the deployed system can do.

Learning: Baum–Welch

When the hidden states are not labeled, the parameters are estimated by expectation–maximization, known for HMMs as the **Baum–Welch** algorithm [12]. The E-step runs forward–backward to get the expected state occupancies $\gamma_t(j)$ and the expected transitions $\xi_t(i, j) = p(z_{t-1} = i, z_t = j \mid x_{1:T})$. The M-step re-estimates

$$\hat{A}_{ij} = \frac{\sum_{t=2}^{T} \xi_t(i,j)}{\sum_{t=2}^{T} \sum_{k} \xi_t(i,k)}, \qquad \hat{\pi}_j = \gamma_1(j),$$

and the emission parameters from observations weighted by $\gamma_t(j)$ (for example, weighted means and covariances for Gaussian emissions). Each iteration does not decrease the likelihood, but the result depends on the initialization and can be a local maximum; PR 07 develops EM in general. If the hidden states are continuous and everything is linear-Gaussian, the same forward recursion

becomes the Kalman filter [13]. Neural recurrent models, which replace the discrete state by a learned vector, are covered in ML 04.

Conditional random fields

In plain words. An HMM models how states *generate* observations, which forces simple assumptions about the observations. When all you need is the labels given the observations, a CRF models that directly, and it can then use any features of the whole input, overlapping or not.

An HMM is **generative**: it models $p(z, x)$, including how observations arise. That requires the emission model $B_j(x_t)$ to describe x_t fully, which is awkward when the features are many, overlapping and dependent (the word itself, its suffix, its capitalization, the neighboring words...). Lafferty, McCallum and Pereira introduced conditional random fields to model the **conditional** distribution of a label sequence $y = (y_1, \dots, y_T)$ given the observations x directly [14]. A **linear-chain CRF** is

$$p(y \mid x) = \frac{1}{Z(x)} \exp\Big(\sum_{t=1}^{T} \sum_{k} \theta_k f_k(y_{t-1}, y_t, x, t) \Big), \qquad Z(x) = \sum_{y'} \exp\Big(\sum_{t,k} \theta_k f_k(y'_{t-1}, y'_t, x, t) \Big),$$

where the f_k are feature functions that may look at the current and previous label and at *any* part of the input sequence, θ_k are learned weights, and $Z(x)$ normalizes over all label sequences y' for this particular input [14], [15]. An HMM, written in this form, is a special case in which the features are only "previous label, current label" and "current label, current observation" indicators. The CRF can add arbitrary input features without modeling their distribution.

Inference uses the same dynamic programs as the HMM: the forward algorithm (with log-sum-exp in place of sums) computes $Z(x)$ and the marginals, and Viterbi finds the best labeling. Training maximizes the conditional log-likelihood $\sum_n \log p(y^{(n)} \mid x^{(n)})$, usually with an ℓ_2 penalty. For a linear-chain CRF that objective is concave, so there are no bad local optima; its gradient is "observed feature counts minus expected feature counts," where the expectations come from forward–backward [15]. The code checks the forward computation of $\log Z$ against brute force.

```
import itertools
import numpy as np

rng = np.random.default_rng(0)
T, K = 5, 3
U = rng.normal(size=(T, K))       # unary scores  theta . f(y_t, x, t)
P = rng.normal(size=(K, K))       # transition scores  theta . f(y_{t-1}, y_t)

def score(y):
    return U[np.arange(T), y].sum() + sum(P[y[t - 1], y[t]] for t in range(1, T))

# log Z by the forward recursion in log space (log-sum-exp instead of sum).
log_alpha = U[0].copy()
for t in range(1, T):
    m = log_alpha[:, None] + P
    log_alpha = U[t] + np.logaddexp.reduce(m, axis=0)
```

```
log_Z = np.logaddexp.reduce(log_alpha)

brute = np.logaddexp.reduce([score(np.array(y))
                             for y in itertools.product(range(K), repeat=T)])
print("log Z forward:", round(log_Z, 6), " brute force:", round(brute, 6))
assert np.isclose(log_Z, brute)
```

Both give $\log Z = 6.173403$. Here the scores are random numbers standing in for $\sum_k \theta_k f_k$; in a real CRF they come from features of x, and in modern systems from a neural network.

On images, the same idea gives grid-structured CRFs over pixels, with unary terms from a pixel classifier and pairwise terms like the Potts energy above. Exact inference is no longer possible on loopy grids, so approximate methods are used (PR 08).

Modern view

Templates, alignment, grammars and Markov chains look old-fashioned next to deep networks. In fact each survives, sometimes inside the networks. These are the main threads in the review literature and recent work.

Template matching. Hashemi et al. [4] review the basic concept of matching, advances such as invariant features and deformable models, and applications including medical image analysis. They conclude that template matching works best combined with other recognition steps, for example NCC followed by a robust decision stage. Correlation-based matching is fast, simple and exact when appearance is controlled, and it remains standard in industrial inspection, image registration and as a building block of trackers. When appearance varies, matching moved to learned feature spaces; the "template" is now a feature map, and correlation is computed between deep features, but the operation is the same.

Time-series classification and DTW. Bagnall et al.'s "bake off" [16] reimplemented 18 time-series classification algorithms in one framework and compared them over 100 resamples of each of 85 UCR datasets. Only nine were significantly better than both benchmarks (one of which was 1-NN with DTW), and a collective of transformation-based ensembles (COTE) beat all others. The "bake off redux" by Middlehurst, Schäfer and Bagnall [17] repeated the exercise on the expanded 112-dataset archive plus 30 new problems, added convolution-based, feature-based and deep-learning categories to the taxonomy, and found two recent algorithms, MultiROCKET+Hydra and HIVE-COTE 2, significantly better than the rest. Ismail Fawaz et al. [18] reviewed deep learning for time-series classification with a unified taxonomy and trained 8,730 networks on 97 datasets in one open framework. Meanwhile DTW itself became a building block: soft-DTW [19] replaces the minimum in the DTW recursion by a smooth soft-minimum, which makes it differentiable so it can serve as a loss for training networks and for averaging time series.

Alignment inside neural networks. Connectionist temporal classification (CTC) [20] trains a recurrent network to output label sequences without a frame-by-frame alignment, by summing over all alignments with a forward–backward recursion closely related to the HMM's. The dynamic program survived; the emission model became a network.

Online only: Connectionist temporal classification (CTC) — https://blog.shyandram.dev/en/learn/pattern-recognition/templates-structured-models/notes/connectionist-temporal-classification-ctc/

CRFs with deep features. Sutton and McCallum's tutorial [15] remains the standard reference for CRF modeling, inference and training. In vision, fully connected CRFs with Gaussian edge potentials made mean-field inference over all pixel pairs efficient [21] and became a popular refinement step for semantic segmentation; Zheng et al. then wrote that mean-field inference as a recurrent network layer and trained it end to end with the CNN [22]. In language, the BiLSTM-CRF of Lample et al. [23] put a CRF output layer on top of a bidirectional LSTM for named-entity recognition, so that label transitions are modeled explicitly while features are learned. In both areas, later architectures with large receptive fields or attention often model enough context that the explicit CRF adds little, but the structured output layer is still used where label constraints matter.

Structure and graphs. The graph-matching surveys of Conte et al. [8] and Foggia et al. [9] trace the move from exact and error-tolerant matching to graph kernels, graph embeddings and learning-based methods. The modern descendants, graph neural networks, learn node and graph representations by message passing, a direct generalization of the local computations of factor graphs.

State-space models. The idea that a compact hidden state summarizes the past, the core of HMMs and the Kalman filter [13], is central again in sequence modeling. Selective state-space models such as Mamba [24] use input-dependent linear recurrences that run in time linear in the sequence length, offered as an alternative to attention for long sequences.

Open problems. DTW and edit distances are quadratic per comparison, so search over large collections relies on bounds and pruning [6]. Graph edit distance remains expensive to compute exactly. HMM and CRF results depend on the independence assumptions built into the graph; when those are wrong, smoothing can confidently erase short events. And for any online system, the distinction between filtered and smoothed evaluation is easy to get wrong.

Key takeaways

- NCC subtracts means and divides by norms, so it is invariant to brightness offsets and positive contrast changes; SSD and raw correlation are not. Neither handles rotation, scale or deformation.
- DTW finds the cheapest monotone alignment by dynamic programming in $O(nm)$; a Sakoe–Chiba band limits pathological warping and cost, but must be wider than the real timing differences.
- Structural recognition describes patterns as strings, grammars or graphs; edit distance compares strings with the same dynamic-programming pattern as DTW.
- Graphical models factorize joint distributions into local pieces; message passing on chains and trees is exact and efficient.
- HMM algorithms: forward ($p(x)$ and filtering), forward–backward (smoothing), Viterbi (best path), Baum–Welch (learning), all $O(TK^2)$. Scale or work in logs to avoid underflow.

- The Viterbi path and per-step most-probable states answer different questions; the latter can be an impossible sequence.
- CRFs model $p(y \mid x)$ directly, can use arbitrary overlapping input features, share the HMM's inference algorithms and have a concave training objective in the linear-chain case.

Exercises

1. **NCC and negatives.** Show that NCC is unchanged when every pixel of a patch is replaced by $ax_i + b$ with $a > 0$. What happens when $a < 0$, and why can NCC not tell you the value of b?

   Hint

   Subtracting the mean removes b entirely, so no information about it survives. The factor a appears in the numerator and as $|a|$ in the denominator; it cancels for $a > 0$ and flips the sign for $a < 0$ (a photographic negative of a perfect match scores -1).

2. **DTW by hand.** Compute the DTW distance with cost $|x_i - t_j|$ between $x = (0, 1, 2)$ and $t = (0, 0, 1, 2)$. Write out the 3×4 table and the optimal path. What is the lock-step distance, and why is it not even defined here?

   Hint

   The optimal path matches x_1 to both zeros, then $x_2 \leftrightarrow t_3$ and $x_3 \leftrightarrow t_4$, with total cost 0. Lock-step comparison needs equal lengths, which these sequences do not have; DTW handles unequal lengths naturally.

3. **A forbidden transition.** Construct your own two- or three-state HMM with one transition probability set to zero and an observation sequence for which the per-step argmax of γ_t is an impossible state sequence. Verify with the code above.

   Hint

   You need two neighboring time steps whose individually most likely states are joined by the forbidden transition. A symmetric or cyclic model with ambiguous emissions (probabilities close to 0.5) makes this easy; then check that `joint(marginal)` is 0.

4. **Online versus offline.** An activity recognizer reports 95% accuracy computed from smoothed state probabilities on recorded sequences. It will run on a phone and must decide at each second. What information advantage does the reported number include, and what would you report instead?

   Hint

   Smoothing uses observations from the future of each decision. Report filtered accuracy, or fixed-lag smoothing with the lag the application can tolerate (for example decisions delayed by 2 seconds use observations up to $t + 2$).

5. **From HMM to CRF.** Write an HMM's $\log p(z, x)$ as a sum of feature functions times weights, in the form of the linear-chain CRF. Which features and weights do you get? What extra features could a CRF add for part-of-speech tagging that an HMM cannot easily use?

   Hint

   Use indicator features $\mathbf{1}[z_{t-1} = i, z_t = j]$ with weight $\log A_{ij}$, $\mathbf{1}[z_t = j, x_t = v]$ with weight $\log B_j(v)$, and $\mathbf{1}[t = 1, z_1 = j]$ with weight $\log \pi_j$. A CRF can add features such as "the next word ends in -ly" or "the current word is capitalized," which overlap with and depend on the word identity.

6. **Edit distance with weights.** Modify `edit_distance` so that substituting two directions that differ by 90 degrees costs 1 and by 180 degrees costs 2. Recompute the distances in the chain-code example. Does the ranking change?

   Hint

   The substitution cost is the circular difference $\min(|a - b|, 4 - |a - b|)$ between direction codes. The bumped square needs only insertions, so its distance stays 2; check whether the rectangle's distance changes, given which substitutions its cheapest edit sequence uses.

References

1. S. Theodoridis and K. Koutroumbas, *Pattern Recognition*, 4th ed., Academic Press, 2009, Chapters 8 and 9. publisher page
2. C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006, Sections 8.1–8.4 and 13.1–13.3. official book page and PDF
3. J. P. Lewis, "Fast Normalized Cross-Correlation," *Vision Interface*, 1995. paper
4. N. S. Hashemi, R. B. Aghdam, A. S. B. Ghiasi, and P. Fatemi, "Template Matching Advances and Applications in Image Analysis," arXiv:1610.07231, 2016. arXiv
5. H. Sakoe and S. Chiba, "Dynamic Programming Algorithm Optimization for Spoken Word Recognition," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 26, no. 1, pp. 43–49, 1978. doi:10.1109/TASSP.1978.1163055
6. T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh, "Searching and Mining Trillions of Time Series Subsequences under Dynamic Time Warping," *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 262–270, 2012. doi:10.1145/2339530.2339576
7. R. A. Wagner and M. J. Fischer, "The String-to-String Correction Problem," *Journal of the ACM*, vol. 21, no. 1, pp. 168–173, 1974. doi:10.1145/321796.321811
8. D. Conte, P. Foggia, C. Sansone, and M. Vento, "Thirty Years of Graph Matching in Pattern Recognition," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 18, no. 3, pp. 265–298, 2004. doi:10.1142/S0218001404003228
9. P. Foggia, G. Percannella, and M. Vento, "Graph Matching and Learning in Pattern Recognition in the Last 10 Years," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 28, no. 1, 1450001, 2014. doi:10.1142/S0218001414500013

10. L. R. Rabiner, “A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition,” *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989. doi:10.1109/5.18626
11. A. J. Viterbi, “Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm,” *IEEE Transactions on Information Theory*, vol. 13, no. 2, pp. 260–269, 1967. doi:10.1109/TIT.1967.1054010
12. L. E. Baum, T. Petrie, G. Soules, and N. Weiss, “A Maximization Technique Occurring in the Statistical Analysis of Probabilistic Functions of Markov Chains,” *The Annals of Mathematical Statistics*, vol. 41, no. 1, pp. 164–171, 1970. doi:10.1214/aoms/1177697196
13. R. E. Kalman, “A New Approach to Linear Filtering and Prediction Problems,” *Journal of Basic Engineering*, vol. 82, no. 1, pp. 35–45, 1960. doi:10.1115/1.3662552
14. J. D. Lafferty, A. McCallum, and F. C. N. Pereira, “Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data,” *Proceedings of the 18th International Conference on Machine Learning (ICML)*, pp. 282–289, 2001. paper page
15. C. Sutton and A. McCallum, “An Introduction to Conditional Random Fields,” *Foundations and Trends in Machine Learning*, vol. 4, no. 4, pp. 267–373, 2012. doi:10.1561/2200000013
16. A. Bagnall, J. Lines, A. Bostrom, J. Large, and E. Keogh, “The Great Time Series Classification Bake Off: A Review and Experimental Evaluation of Recent Algorithmic Advances,” *Data Mining and Knowledge Discovery*, vol. 31, pp. 606–660, 2017. doi:10.1007/s10618-016-0483-9
17. M. Middlehurst, P. Schäfer, and A. Bagnall, “Bake Off Redux: A Review and Experimental Evaluation of Recent Time Series Classification Algorithms,” *Data Mining and Knowledge Discovery*, vol. 38, pp. 1958–2031, 2024. doi:10.1007/s10618-024-01022-1
18. H. Ismail Fawaz, G. Forestier, J. Weber, L. Idoumghar, and P.-A. Muller, “Deep Learning for Time Series Classification: A Review,” *Data Mining and Knowledge Discovery*, vol. 33, pp. 917–963, 2019. doi:10.1007/s10618-019-00619-1
19. M. Cuturi and M. Blondel, “Soft-DTW: A Differentiable Loss Function for Time-Series,” *ICML*, 2017. arXiv:1703.01541
20. A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber, “Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks,” *Proceedings of the 23rd International Conference on Machine Learning (ICML)*, pp. 369–376, 2006. doi:10.1145/1143844.1143891
21. P. Krähenbühl and V. Koltun, “Efficient Inference in Fully Connected CRFs with Gaussian Edge Potentials,” *NIPS*, 2011. arXiv:1210.5644
22. S. Zheng, S. Jayasumana, B. Romera-Paredes, V. Vineet, Z. Su, D. Du, C. Huang, and P. H. S. Torr, “Conditional Random Fields as Recurrent Neural Networks,” *ICCV*, 2015. arXiv:1502.03240
23. G. Lample, M. Ballesteros, S. Subramanian, K. Kawakami, and C. Dyer, “Neural Architectures for Named Entity Recognition,” *NAACL*, 2016. arXiv:1603.01360
24. A. Gu and T. Dao, “Mamba: Linear-Time Sequence Modeling with Selective State Spaces,” arXiv:2312.00752, 2023. arXiv

PR 07 • Clustering, Mixtures and EM

Group unlabeled data with k-means and k-means++, hierarchical linkage, DBSCAN and spectral methods; derive Gaussian-mixture EM; choose the number of clusters with BIC and validity indices.

Online edition: https://blog.shyandram.dev/en/learn/pattern-recognition/clustering-mixtures/

What you'll learn

- What a clustering algorithm actually optimizes, and why the choice of features, scaling and distance decides the answer before any algorithm runs.
- How Lloyd's k-means works, why it always converges but only to a local optimum, and how k-means++ seeding gives a provable $O(\log K)$ guarantee.
- How hierarchical clustering (single, complete, average and Ward linkage), DBSCAN and spectral clustering encode different definitions of "a group".
- How to derive the E and M steps of expectation–maximization (EM) for a Gaussian mixture from scratch, why EM never decreases the likelihood, and where it can fail.
- How to choose the number of clusters with BIC and AIC, and how to read internal indices (silhouette, Davies–Bouldin, Calinski–Harabasz), external indices (ARI, NMI) and stability.

The big picture

So far in this track every training example came with a label. Clustering removes the labels and asks: are there natural groups in the data? The answer is never fully determined by the data alone. The same collection of images can be grouped by object, by color, by background or by the camera that took them. Each grouping is "correct" under its own notion of similarity. A clustering algorithm is therefore a precise statement of *which* notion of similarity we mean, plus a search procedure for the best grouping under that notion.

> **Explain like I'm five**
>
> You pour a big box of buttons on the floor and want to tidy them into piles. You could pile them by color, by size, or by how many holes they have. Every way makes neat piles, but which one is useful depends on what you want to do next. A clustering method is a rule for making piles. **k-means** says "pick a few pile-centers and put each button on the closest pile." **DBSCAN** says "buttons that are crowded together belong to the same pile, and lonely buttons are leftovers." A **Gaussian mixture** says "each pile is a fuzzy cloud, and a button can be 70% sure it belongs to one cloud and 30% to another."

Why it matters in recognition: clustering builds visual vocabularies (local image descriptors quantized with k-means into "visual words"), initializes and compresses models (vector quantization,

color palettes), discovers sub-classes inside a labeled class, finds outliers, segments images (pixel clustering, see IP 10), and, in modern self-supervised learning, produces pseudo-labels for training deep networks. Gaussian mixtures are also the standard example of a **latent-variable model**, and EM is the template for many inference algorithms we will meet in PR 08.

This chapter follows the clustering part of Theodoridis and Koutroumbas (Chapters 11–16) [1] and Bishop's Chapter 9 on mixtures and EM [2]. It organizes the main algorithm families, derives the two most important ones in full, and ends with how to decide whether a clustering is any good.

Similarity, scaling and what "a cluster" means

Plain version. Before choosing an algorithm, decide what "close" means. Change the features or their units and the clusters change.

Precise version. Most methods start from a dissimilarity $d(x, x')$ between feature vectors $x, x' \in \mathbb{R}^D$. Common choices are

$$d_2(x,x') = \|x - x'\|_2, \qquad d_M(x,x') = \sqrt{(x-x')^\top S^{-1}(x-x')}, \qquad d_{\cos}(x,x') = 1 - \frac{x^\top x'}{\|x\|\,\|x'\|},$$

where d_2 is the Euclidean distance, d_M is the Mahalanobis distance with a positive-definite matrix S (often a covariance estimate), and $d_{\cos}$ is the cosine dissimilarity, which ignores vector length and is popular for deep-network embeddings.

Three practical consequences follow.

1. **Scaling is a modeling decision.** If one coordinate is measured in millimeters and another in kilometers, Euclidean distance is dominated by the first. Standardizing every feature to unit variance removes that, but it also removes any genuinely meaningful difference in spread. For classification we can let validation decide; for clustering there are no labels, so the choice must come from domain knowledge.
2. **Hard versus soft assignments.** A *hard* partition assigns each point x_n to exactly one cluster $z_n \in \{1, \dots, K\}$. A *soft* assignment stores degrees of membership $r_{nk} \geq 0$ with $\sum_k r_{nk} = 1$. In a Gaussian mixture these are posterior probabilities; in fuzzy clustering [1] they are memberships defined by an objective and should not be read as probabilities.
3. **Clusters are not classes.** Labels are attached to clusters afterwards and arbitrarily. Cluster "3" in one run may be cluster "1" in the next, so any comparison with class labels must be invariant to permutations, which is exactly what ARI and NMI (below) are.

Theodoridis and Koutroumbas classify algorithms into sequential, hierarchical, objective-function-based (k-means, mixtures, fuzzy), and graph/density-based families [1]. We follow roughly that order.

K-means and Lloyd's algorithm

Plain version. Choose K centers. Repeat two steps: give every point to its nearest center, then move every center to the average of its points. Stop when nothing changes.

The objective

With one-hot indicators $z_{nk} \in \{0, 1\}$ ($z_{nk} = 1$ if point n is in cluster k, and $\sum_k z_{nk} = 1$) and centers $\mu_1, \dots, \mu_K \in \mathbb{R}^D$, k-means minimizes the within-cluster sum of squares

$$J(z, \mu) = \sum_{n=1}^{N} \sum_{k=1}^{K} z_{nk} \, \|x_n - \mu_k\|^2.$$

Here N is the number of points and D their dimension. J is also called *inertia* or *distortion*.

Two alternating steps

Lloyd's algorithm [3] (developed for quantizing signals in pulse-code modulation) minimizes J by **block coordinate descent**:

- **Assignment step.** Fix μ. Each term involves a single z_n, so set $z_{nk} = 1$ for $k = \arg\min_j \|x_n - \mu_j\|^2$.
- **Update step.** Fix z. J is a convex quadratic in each μ_k; setting the gradient to zero,

$$\frac{\partial J}{\partial \mu_k} = -2 \sum_n z_{nk}(x_n - \mu_k) = 0 \quad \Longrightarrow \quad \mu_k = \frac{\sum_n z_{nk} x_n}{\sum_n z_{nk}},$$

the mean of the points currently in cluster k. The name "k-means" was coined by MacQueen [4], whose version updates a center immediately after each point arrives, $\mu_k \leftarrow \mu_k + (x_n - \mu_k)/N_k$, where N_k is the running count of points in cluster k.

Why it converges, and why only locally

Each step can only lower J or leave it unchanged: the assignment step picks the best z for the current μ, and the update step picks the best μ for the current z. There are finitely many partitions of N points into K groups, and (with a consistent tie-breaking rule) the algorithm never revisits a partition with a higher J, so it stops after finitely many iterations. The fixed point is a **local** minimum: a partition where no single step helps. Different initial centers can lead to very different fixed points.

Two implementation details matter. An **empty cluster** (no point assigned) has no mean; common rules keep the old center or re-seed it at the point with the largest current error. And every iteration costs $O(NKD)$ distance computations, which motivates the mini-batch variant below.

What k-means assumes

Because the objective is a sum of squared Euclidean distances to a center, the boundary between two clusters is the perpendicular bisector of their centers: k-means partitions space into **Voronoi cells**. It works well for compact, roughly spherical groups of similar size. It splits elongated groups, merges small groups that are close to large ones, and cannot follow curved shapes (Figure 1, middle).

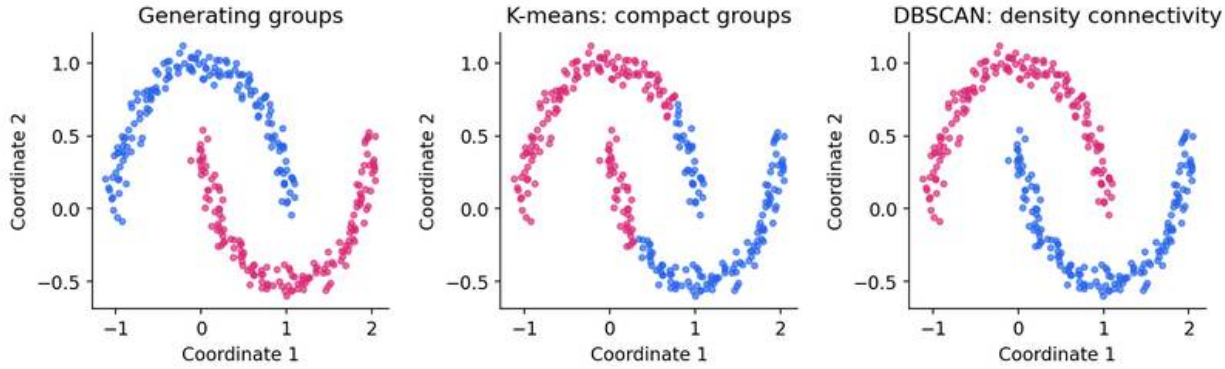


Figure 1 — The same two-moons data grouped by the generating labels (left), k-means (middle) and DBSCAN (right). K-means and DBSCAN fit without labels. Colors are arbitrary cluster identifiers.

k-means++ seeding

Plain version. Start with one random center, then pick each new center far from the ones you already have, but randomly, so outliers do not always win.

Arthur and Vassilvitskii [5] proposed **D² seeding**:

1. Pick the first center uniformly at random from the data.
2. For every point compute $D(x_n)$, the distance to the nearest center chosen so far.
3. Pick the next center as x_n with probability $D(x_n)^2 / \sum_m D(x_m)^2$.
4. Repeat steps 2–3 until K centers are chosen, then run Lloyd's algorithm.

Their main theorem states that the seeding alone already achieves, in expectation,

$$\mathbb{E}[J] \leq 8(\ln K + 2)\, J_{\mathrm{OPT}},$$

where J_{OPT} is the optimal k-means objective. Lloyd iterations can only lower J further. Uniform random seeding has no such guarantee. The bound is a worst-case statement; in practice the benefit shows up mostly as fewer catastrophic runs. Libraries combine k-means++ with several restarts (`n_init` in scikit-learn) and keep the run with the lowest J.

Code: k-means++ and Lloyd from scratch

```
import numpy as np
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans

def kmeans_pp_init(X, k, rng):
    """D^2 seeding: each new center is drawn with probability proportional
    to the squared distance to the nearest center chosen so far."""
    centers = [X[rng.integers(len(X))]]
    for _ in range(k - 1):
        d2 = np.min(((X[:, None, :] - np.array(centers)[None]) ** 2).sum(-1), axis=1)
        centers.append(X[rng.choice(len(X), p=d2 / d2.sum())])
    return np.array(centers)

def lloyd(X, centers, max_iter=100):
    history = []
    for _ in range(max_iter):
        d2 = ((X[:, None, :] - centers[None]) ** 2).sum(-1)   # N x K
        z = d2.argmin(axis=1)                                 # assignment step
        history.append(d2[np.arange(len(X)), z].sum())        # objective J
        new = np.array([X[z == k].mean(axis=0) if np.any(z == k) else centers[k]
                        for k in range(len(centers))])        # update step
        if np.allclose(new, centers):
            break
        centers = new
    return centers, z, history

X, _ = make_blobs(n_samples=600, centers=4, cluster_std=1.8, random_state=7)
rng = np.random.default_rng(0)
best = None
for restart in range(10):                       # several seeds, keep the best J
    c, z, hist = lloyd(X, kmeans_pp_init(X, 4, rng))
    if best is None or hist[-1] < best[2][-1]:
        best = (c, z, hist)
c, z, hist = best
print("J per iteration:", np.round(hist, 1))
assert all(a >= b - 1e-9 for a, b in zip(hist, hist[1:]))   # monotone
ref = KMeans(n_clusters=4, n_init=10, random_state=0).fit(X)
print("from scratch J:", round(hist[-1], 2), " sklearn inertia:", round(ref.inertia_, 2))

# single runs (no restarts): uniform random seeding vs D^2 seeding, 15 clusters
X2, _ = make_blobs(n_samples=1500, centers=15, cluster_std=0.6, random_state=2)
for name, init in [("random", lambda r: X2[r.choice(len(X2), 15, replace=False)]),
                   ("k-means++", lambda r: kmeans_pp_init(X2, 15, r))]:
    finals = [lloyd(X2, init(np.random.default_rng(s)))[2][-1] for s in range(20)]
    print(f"{name:9s} final J over 20 runs: median {np.median(finals):7.1f}, "
          f"worst {np.max(finals):7.1f}")
```

Output (NumPy 2.0, scikit-learn 1.6):

```
J per iteration: [5865.6 3633.4 3625. ]
from scratch J: 3625.0  sklearn inertia: 3625.0
```

```
random    final J over 20 runs: median  1305.1, worst  4750.8
k-means++ final J over 20 runs: median  1147.5, worst  1525.2
```

The from-scratch version reaches the same objective as scikit-learn, and J decreases monotonically. On the 15-cluster problem, uniform seeding sometimes puts two seeds in one blob and none in another; Lloyd's algorithm cannot repair that, and the worst run is about three times worse. Figure 2 shows the two worst runs.

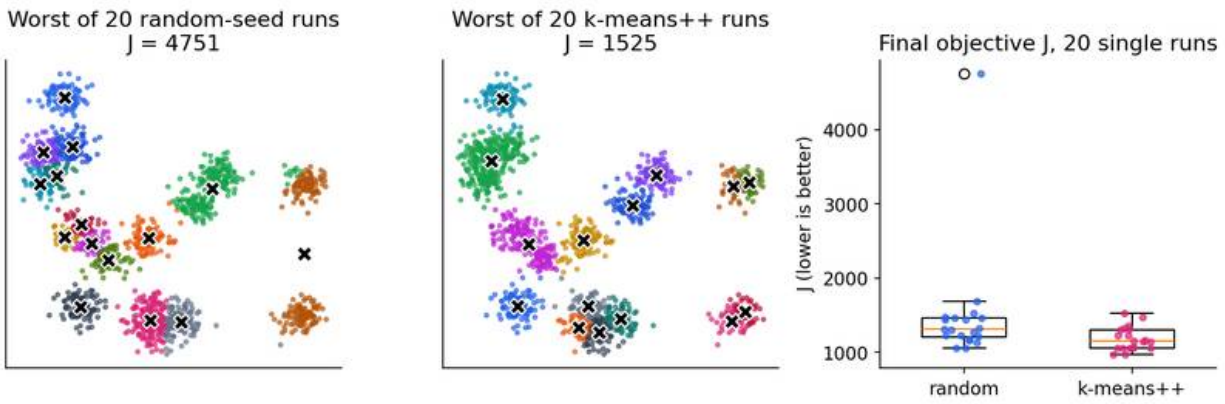


Figure 2 — Seeding matters. Fifteen synthetic blobs, 20 single Lloyd runs per seeding rule (no restarts). Random seeding leaves some runs stuck with merged and split blobs; D^2 (k-means++) seeding gives lower and less variable objectives. Black crosses are final centers.

Mini-batch k-means

For millions of points, Sculley's mini-batch k-means [6] updates the centers using small random batches: each center moves toward the batch points assigned to it with a per-center learning rate that decays as 1/(count of points seen). It trades a slightly higher final objective for a large speed-up, and is available as `MiniBatchKMeans` in scikit-learn. It is the usual choice for building large visual vocabularies.

Hierarchical clustering

Plain version. Start with every point as its own group. Repeatedly merge the two closest groups. The record of merges is a tree; cut it at some height to get clusters.

Agglomerative algorithm and linkages

Agglomerative clustering needs a definition of the distance between two *groups* A and B. The four classical **linkages** are

$$d_{\text{single}}(A,B) = \min_{a\in A,\, b\in B} d(a,b),$$
$$d_{\text{complete}}(A,B) = \max_{a\in A,\, b\in B} d(a,b),$$
$$d_{\text{average}}(A,B) = \frac{1}{|A|\,|B|}\sum_{a\in A}\sum_{b\in B} d(a,b),$$
$$\Delta_{\text{Ward}}(A,B) = \frac{|A|\,|B|}{|A|+|B|}\,\|\bar{a}-\bar{b}\|^2,$$

where $|A|$ is the number of points in A and $\bar{a}, \bar{b}$ are the group means. Ward's criterion [7] is the increase of the within-cluster sum of squares J caused by merging A and B, so Ward agglomeration is a greedy, bottom-up way of keeping J small, the same objective as k-means.

Each linkage has a personality (Figure 3):

- **Single linkage** merges whenever *any* two points are close. It follows arbitrary shapes, but a thin "bridge" of points can chain two groups together. It is equivalent to cutting the longest edges of a minimum spanning tree, one of the graph-based methods in [1].
- **Complete linkage** requires *all* pairs to be close. It produces compact groups of similar diameter and is sensitive to outliers.
- **Average linkage** sits between the two.
- **Ward** produces compact, similarly sized, roughly spherical groups, much like k-means.

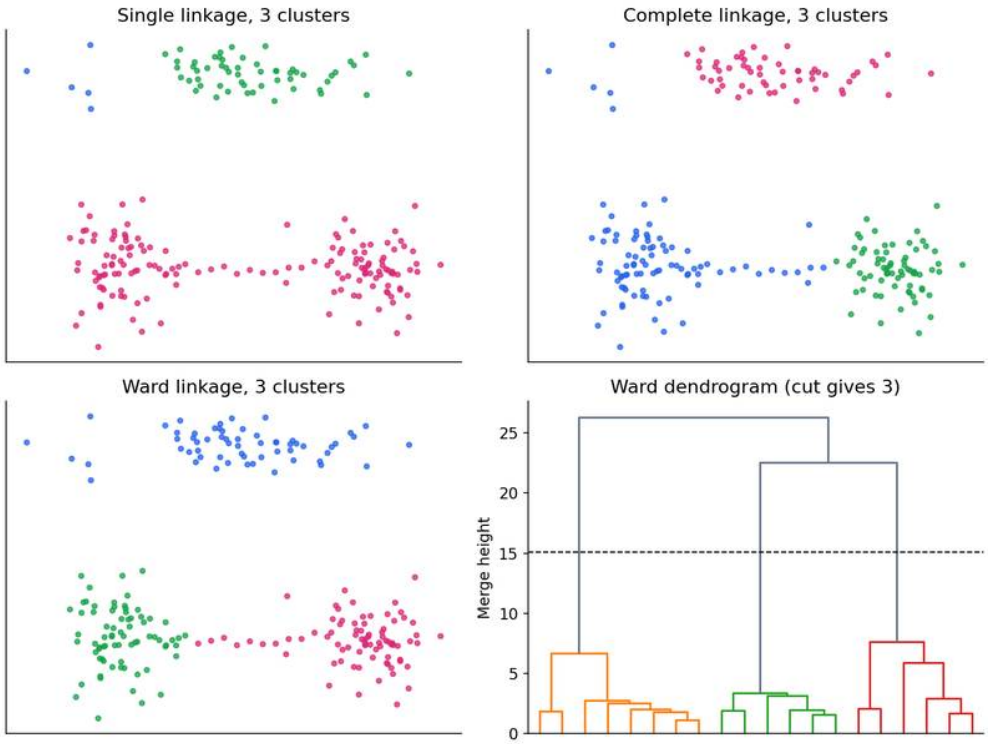


Figure 3 — One data set, three linkages. Single linkage chains along the thin bridge, merges both blobs into

one cluster and spends a cluster on four stray points; complete and Ward linkage keep the two blobs apart and split the bridge between them. Bottom right: the Ward dendrogram, where cutting at the dashed line yields three clusters.

The dendrogram

The merge history is a binary tree called a **dendrogram**; the height of each junction is the linkage value at which the merge happened. Cutting the tree at height h yields the partition at that level, and a large gap between consecutive merge heights suggests a natural number of clusters. The cut height is still a choice.

Two practical limitations: merges are never undone, so an early mistake propagates; and the standard algorithm stores all pairwise distances, $O(N^2)$ memory, which limits it to tens of thousands of points. Divisive (top-down) hierarchical methods exist but are less common.

Density-based clustering: DBSCAN

Plain version. A cluster is a crowded region. Points in crowded places grow the cluster; points in empty places are labeled noise.

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) [8] has two parameters, a radius ε and a count MinPts. Let $N_\varepsilon(x) = \{x' : d(x, x') \le \varepsilon\}$ be the ε-neighborhood of x (including x itself). Then

- x is a **core point** if $|N_\varepsilon(x)| \ge$ MinPts;
- x is a **border point** if it is not core but lies in the ε-neighborhood of some core point;
- every other point is **noise**.

A point is *density-reachable* from a core point if there is a chain of core points, each within ε of the next, ending in its neighborhood. A cluster is a maximal set of mutually density-connected points. DBSCAN finds clusters of any shape, does not need K, and explicitly refuses to assign isolated points.

Choosing ε. The original paper suggests the sorted **k-distance graph**: for each point compute the distance to its k-th nearest neighbor with $k =$ MinPts, sort these values, and pick ε near the "knee" where the curve rises sharply (Figure 4, left). In the descending plot of Figure 4 the large values to the left of the knee are the candidates for noise. scikit-learn's `min_samples` counts the point itself, which matches the definition above.

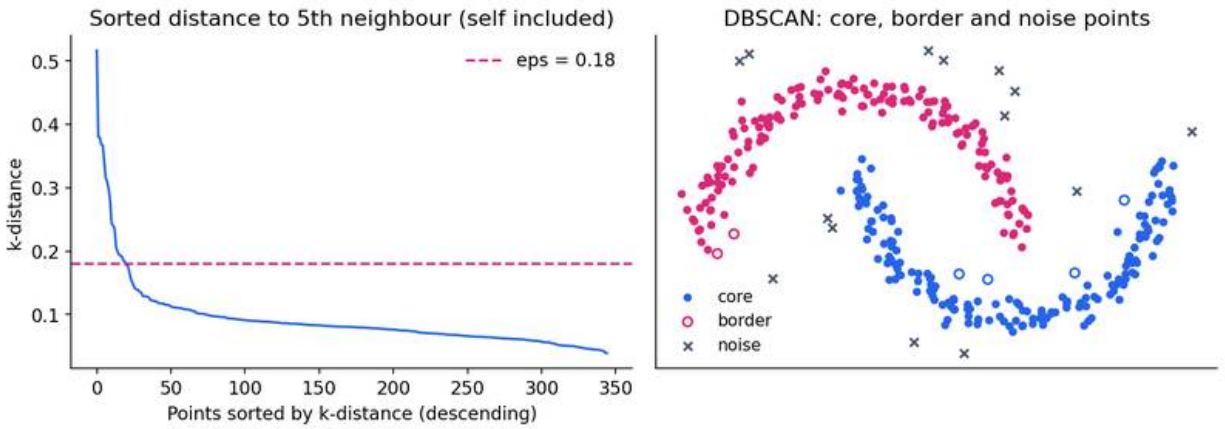


Figure 4 — DBSCAN on two moons plus 25 uniform outliers. Left: the sorted 5-distance curve; the chosen $\varepsilon = 0.18$ sits where the curve starts to rise. Right: core points (filled), border points (hollow) and points rejected as noise (crosses).

Limitations. A single ε means a single density level. If one cluster is dense and another sparse, no ε separates both. In high dimensions, distances concentrate and the neighborhood counts lose contrast. **HDBSCAN** [9] addresses the first issue by building a hierarchy over all density levels and extracting the most stable clusters, replacing ε with a minimum cluster size; it is available as `HDBSCAN` in scikit-learn.

Spectral clustering in one page

Plain version. Build a graph that connects nearby points, then find the cuts that sever the fewest strong connections.

Given an affinity matrix W ($W_{ij} \geq 0$, large for similar points, for example a k-nearest-neighbor graph or $W_{ij} = \exp(-\|x_i - x_j\|^2/2\sigma^2)$) and the diagonal degree matrix $D_{ii} = \sum_j W_{ij}$, one normalized graph Laplacian is

$$L_{\text{sym}} = I - D^{-1/2} W D^{-1/2}.$$

If the graph has K connected components, L_{sym} has eigenvalue 0 with multiplicity K, and its eigenvectors indicate the components. For a nearly disconnected graph, the K eigenvectors with the smallest eigenvalues still nearly indicate the groups. Spectral clustering stacks these eigenvectors as columns, normalizes the rows, and runs k-means on the rows [10]. The result can follow curved shapes that defeat k-means in the original space, because the eigenvectors measure *connectivity*, not distance to a center.

The graph is part of the model. The neighborhood size or bandwidth σ must be chosen; a graph that is accidentally disconnected (an isolated vertex, say) produces a trivial eigenvector that dominates the result. Von Luxburg's tutorial [10] explains the graph-cut interpretation (ratio cut and normalized cut) and gives practical advice on choosing the graph.

Code: one geometry, different assumptions

The two-moons data set below is generated with labels, but the labels are used only to score each result afterwards with the adjusted Rand index (ARI, 1 = perfect agreement up to relabeling, about 0 = chance).

```
import numpy as np
from sklearn.datasets import make_moons
from sklearn.neighbors import NearestNeighbors
from sklearn.cluster import (KMeans, AgglomerativeClustering, DBSCAN,
                             HDBSCAN, SpectralClustering)
from sklearn.mixture import GaussianMixture
from sklearn.metrics import adjusted_rand_score

X, truth = make_moons(n_samples=320, noise=0.06, random_state=5)

# k-distance curve for choosing DBSCAN's eps (k = min_samples)
dist, _ = NearestNeighbors(n_neighbors=5).fit(X).kneighbors(X)
kdist = np.sort(dist[:, -1])                    # 5th neighbour, counting the point itself
print("k-distance quantiles 50/90/99%:", np.round(np.quantile(kdist, [0.5, 0.9, 0.99]), 3))

models = {
    "k-means": KMeans(n_clusters=2, n_init=10, random_state=5),
    "single linkage": AgglomerativeClustering(n_clusters=2, linkage="single"),
    "Ward linkage": AgglomerativeClustering(n_clusters=2, linkage="ward"),
    "Gaussian mixture": GaussianMixture(n_components=2, n_init=3, random_state=5),
    "DBSCAN": DBSCAN(eps=0.18, min_samples=5),
    "HDBSCAN": HDBSCAN(min_cluster_size=15),
    "spectral": SpectralClustering(n_clusters=2, affinity="nearest_neighbors",
                                   n_neighbors=20, assign_labels="kmeans",
                                   random_state=5),
}
for name, model in models.items():
    labels = model.fit_predict(X)
    print(f"{name:17s} ARI = {adjusted_rand_score(truth, labels):.3f}"
          f"   noise points = {int((labels == -1).sum())}")
```

```
k-distance quantiles 50/90/99%: [0.079 0.11  0.144]
k-means           ARI = 0.260   noise points = 0
single linkage    ARI = 1.000   noise points = 0
Ward linkage      ARI = 0.242   noise points = 0
Gaussian mixture  ARI = 0.488   noise points = 0
DBSCAN            ARI = 1.000   noise points = 0
HDBSCAN           ARI = 0.988   noise points = 2
spectral          ARI = 1.000   noise points = 0
```

Figure 1 shows two of these results. Methods built on centers or ellipses (k-means, Ward, a two-component Gaussian mixture) cut each moon in half; connectivity-based methods (single linkage, DBSCAN, HDBSCAN, spectral) recover the moons. The chosen $\varepsilon = 0.18$ lies above the 99th percentile of the k-distance, so no moon point is called noise. A k-nearest-neighbor graph with

too few neighbors can be disconnected, and scikit-learn then warns that the spectral embedding may be unreliable; 20 neighbors avoids that here. None of this makes connectivity methods "better": on overlapping Gaussian blobs single linkage chains everything together and k-means wins. Comparisons must vary geometry, density, noise and dimension.

Gaussian mixtures and the EM algorithm

Plain version. Pretend every point was produced by first rolling a K-sided die to pick a cloud, then sampling from that Gaussian cloud. We never see the die. EM guesses, for every point, how likely each cloud produced it (E step), then refits each cloud using those guesses as weights (M step), and repeats.

The model

A Gaussian mixture model (GMM) with K components has density

$$p(x \mid \theta) = \sum_{k=1}^{K} \pi_k \, \mathcal{N}(x \mid \mu_k, \Sigma_k), \qquad \pi_k \geq 0, \quad \sum_{k=1}^{K} \pi_k = 1,$$

where π_k is the **mixing weight** (prior probability of component k), $\mu_k \in \mathbb{R}^D$ its mean, Σ_k its $D \times D$ covariance, $\theta = \{\pi_k, \mu_k, \Sigma_k\}_{k=1}^{K}$ collects all parameters, and

$$\mathcal{N}(x \mid \mu, \Sigma) = \frac{1}{(2\pi)^{D/2} |\Sigma|^{1/2}} \exp\Big(-\tfrac{1}{2} (x-\mu)^{\top} \Sigma^{-1} (x-\mu) \Big).$$

Introduce a **latent** one-hot variable z_n with $p(z_{nk} = 1) = \pi_k$ and $p(x_n \mid z_{nk} = 1) = \mathcal{N}(x_n \mid \mu_k, \Sigma_k)$. Summing out z_n gives back the mixture density. Given data $X = \{x_1, \dots, x_N\}$, the log-likelihood is

$$\ln p(X \mid \theta) = \sum_{n=1}^{N} \ln \sum_{k=1}^{K} \pi_k \, \mathcal{N}(x_n \mid \mu_k, \Sigma_k).$$

The logarithm of a sum couples all the parameters, so setting its gradient to zero has no closed-form solution. EM is the standard way around this [11], [2].

Step 1: the complete-data log-likelihood

If we *knew* the assignments $Z = \{z_n\}$, the log-likelihood of (X, Z) would be

$$\ln p(X, Z \mid \theta) = \sum_{n=1}^{N} \sum_{k=1}^{K} z_{nk} \Big[\ln \pi_k + \ln \mathcal{N}(x_n \mid \mu_k, \Sigma_k) \Big].$$

The logarithm now acts on a single Gaussian, and the problem splits into K separate weighted Gaussian fits: easy.

Step 2 (E step): the expected complete-data log-likelihood

We do not know Z, but with current parameters θ^{old} we know its posterior. By Bayes' rule,

$$r_{nk} \equiv p(z_{nk} = 1 \mid x_n, \theta^{\text{old}}) = \frac{\pi_k \, \mathcal{N}(x_n \mid \mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \, \mathcal{N}(x_n \mid \mu_j, \Sigma_j)},$$

evaluated at θ^{old}. The number r_{nk} is the **responsibility** of component k for point n; each row sums to one. Because the complete-data log-likelihood is linear in z_{nk}, its expectation under this posterior simply replaces z_{nk} by r_{nk}:

$$Q(\theta, \theta^{\text{old}}) = \sum_{n} \sum_{k} r_{nk} \Big[\ln \pi_k - \tfrac{D}{2} \ln 2\pi - \tfrac{1}{2} \ln |\Sigma_k| - \tfrac{1}{2} (x_n - \mu_k)^{\top} \Sigma_k^{-1} (x_n - \mu_k) \Big].$$

Step 3 (M step): maximize Q in closed form

Define the **effective count** $N_k = \sum_n r_{nk}$.

Mixing weights. Only $\sum_n \sum_k r_{nk} \ln \pi_k$ depends on π, under the constraint $\sum_k \pi_k = 1$. With a Lagrange multiplier λ,

$$\frac{\partial}{\partial \pi_k} \Big[\sum_{n,k} r_{nk} \ln \pi_k + \lambda \Big(\sum_k \pi_k - 1 \Big) \Big] = \frac{N_k}{\pi_k} + \lambda = 0 \;\Rightarrow\; \pi_k = -\frac{N_k}{\lambda}.$$

Summing over k gives $\lambda = -N$, so $\pi_k = N_k / N$. (The constraint $\pi_k \geq 0$ is satisfied automatically.)

Means. The gradient with respect to μ_k is $\sum_n r_{nk} \Sigma_k^{-1} (x_n - \mu_k)$. Setting it to zero and multiplying by Σ_k,

$$\mu_k = \frac{1}{N_k} \sum_{n} r_{nk} \, x_n.$$

Covariances. Write $\Lambda_k = \Sigma_k^{-1}$, so $-\tfrac{1}{2} \ln |\Sigma_k| = \tfrac{1}{2} \ln |\Lambda_k|$ and the quadratic term is $-\tfrac{1}{2} \operatorname{tr}\big(\Lambda_k (x_n - \mu_k)(x_n - \mu_k)^{\top}\big)$. Using $\partial \ln |\Lambda| / \partial \Lambda = \Lambda^{-1}$ and $\partial \operatorname{tr}(\Lambda A) / \partial \Lambda = A^{\top}$ for symmetric A,

$$\frac{\partial Q}{\partial \Lambda_k} = \frac{1}{2} \sum_{n} r_{nk} \Big[\Sigma_k - (x_n - \mu_k)(x_n - \mu_k)^{\top} \Big] = 0 \;\Rightarrow\; \Sigma_k = \frac{1}{N_k} \sum_{n} r_{nk} (x_n - \mu_k)(x_n - \mu_k)^{\top}.$$

These are the familiar maximum-likelihood formulas for one Gaussian (see PR 02), with every point weighted by its responsibility. EM alternates the E step and the M step until the log-likelihood stops improving (Figure 5).

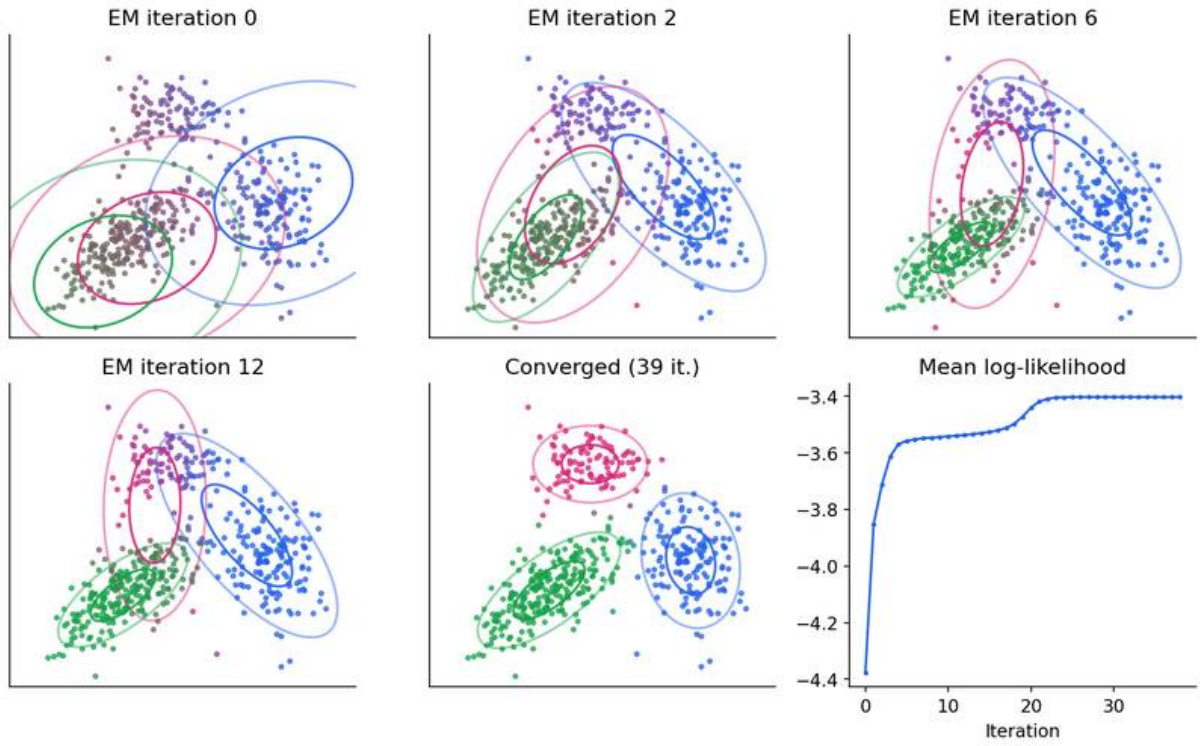


Figure 5 — EM for a three-component Gaussian mixture. The components start at random data points with the overall covariance; point colors mix the component colors by responsibility. Right: the mean log-likelihood never decreases.

Why EM never decreases the likelihood

For any distribution $q(Z)$ over the latent assignments,

$$\ln p(X \mid \theta) = \underbrace{\sum_{Z} q(Z) \ln \frac{p(X, Z \mid \theta)}{q(Z)}}_{\mathcal{L}(q,\theta)} + \underbrace{\sum_{Z} q(Z) \ln \frac{q(Z)}{p(Z \mid X, \theta)}}_{\mathrm{KL}(q \,\|\, p(Z \mid X, \theta)) \geq 0}.$$

So $\mathcal{L}(q, \theta)$ is a lower bound on the log-likelihood, the **evidence lower bound** (ELBO). The E step sets $q(Z) = p(Z \mid X, \theta^{\text{old}})$, which makes the KL term zero and the bound tight at θ^{old}. The M step maximizes $\mathcal{L}$ over θ; since $\mathcal{L}(q, \theta) = Q(\theta, \theta^{\text{old}}) + \text{const}$, this is the same as maximizing Q. The new likelihood is at least the new bound, which is at least the old bound, which equals the old likelihood. Hence $\ln p(X \mid \theta^{\text{new}}) \geq \ln p(X \mid \theta^{\text{old}})$ [11], [2]. When the exact posterior is intractable, we restrict q to a tractable family and get **variational inference**, the subject of PR 08.

Monotone does not mean globally optimal or safely convergent: under suitable regularity conditions EM can approach a stationary point, while an unconstrained Gaussian mixture can instead head toward the singularity described next. Practical implementations initialize from k-means and run several restarts.

Singularities and how to avoid them

The likelihood of an unconstrained GMM is **unbounded**. Put one component's mean exactly on a data point, $\mu_j = x_n$, with $\Sigma_j = \sigma^2 I$. That term contributes $\pi_j (2\pi\sigma^2)^{-D/2}$ to $p(x_n)$, which goes to infinity as $\sigma \to 0$, while the other components keep the remaining points' likelihood finite. The unrestricted maximum likelihood estimate therefore need not exist, and EM itself does not prevent collapse. Use a positive covariance floor (`reg_covar` in scikit-learn [26]), explicit variance constraints, or an appropriate prior. **Diagonal or spherical covariance alone is not a cure**: the collapsing covariance $\sigma^2 I$ already has both forms. A simpler covariance reduces parameter count; it still needs protection against zero variance. Restarting a component is an operational response, not a proof that the unconstrained objective is bounded.

Identifiability and label switching

Permuting the component labels leaves the density unchanged, so there are $K!$ equivalent maxima. "Component 2" has no inherent meaning, and comparing component indices with class labels requires a matching rule.

K-means as the limit of EM

Fix $\pi_k = 1/K$ and $\Sigma_k = \epsilon I$ for all k. The responsibilities become

$$r_{nk} = \frac{\exp(-\|x_n - \mu_k\|^2/2\epsilon)}{\sum_j \exp(-\|x_n - \mu_j\|^2/2\epsilon)}.$$

As $\epsilon \to 0$, the term with the smallest distance dominates, so $r_{nk} \to 1$ for the nearest center and 0 otherwise: the E step becomes the k-means assignment step and the M step for μ_k becomes the k-means update [2]. A general GMM therefore extends k-means with soft assignments, unequal sizes (π_k) and elliptical shapes (Σ_k).

Code: EM from scratch, checked against scikit-learn

```
import numpy as np
from scipy.special import logsumexp
from scipy.stats import multivariate_normal
from sklearn.mixture import GaussianMixture

rng = np.random.default_rng(3)
true_means = np.array([[0.0, 0.0], [4.0, 1.0], [1.0, 4.0]])
true_covs = [np.array([[1.0, 0.6], [0.6, 0.8]]),
             np.array([[0.5, -0.2], [-0.2, 1.2]]),
             np.array([[0.7, 0.0], [0.0, 0.3]])]
counts = [250, 150, 100]
X = np.vstack([rng.multivariate_normal(m, S, n)
               for m, S, n in zip(true_means, true_covs, counts)])
N, D = X.shape
```

```
K, reg = 3, 1e-6                                # reg: small covariance floor

# initialise with K random points, shared data covariance, equal weights
mu = X[rng.choice(N, K, replace=False)]
Sigma = np.array([np.cov(X.T) for _ in range(K)])
pi = np.full(K, 1.0 / K)
loglik = []
for it in range(200):
    # E step: log of pi_k N(x_n | mu_k, Sigma_k), normalised with log-sum-exp
    log_joint = np.column_stack([np.log(pi[k]) +
                                 multivariate_normal(mu[k], Sigma[k]).logpdf(X)
                                 for k in range(K)])
    log_norm = logsumexp(log_joint, axis=1)
    R = np.exp(log_joint - log_norm[:, None])     # responsibilities r_nk
    loglik.append(log_norm.sum())
    # M step: closed-form weighted updates
    Nk = R.sum(axis=0)
    pi = Nk / N
    mu = (R.T @ X) / Nk[:, None]
    for k in range(K):
        diff = X - mu[k]
        Sigma[k] = (R[:, k, None] * diff).T @ diff / Nk[k] + reg * np.eye(D)
    if it > 0 and loglik[-1] - loglik[-2] < 1e-8:
        break

assert np.all(np.diff(loglik) > -1e-6)           # EM never decreases log-likelihood
print("iterations:", len(loglik), " final mean log-lik:", round(loglik[-1] / N, 4))
print("weights:", np.round(np.sort(pi), 3))
gm = GaussianMixture(3, covariance_type="full", n_init=5, random_state=0).fit(X)
print("sklearn mean log-lik:", round(gm.score(X), 4))
print("sklearn weights:", np.round(np.sort(gm.weights_), 3))
```

```
iterations: 40  final mean log-lik: -3.4032
weights: [0.202 0.299 0.499]
sklearn mean log-lik: -3.4033
sklearn weights: [0.203 0.3   0.497]
```

The E step works in log space: densities in high dimension underflow, and `logsumexp` computes $\ln \sum_k e^{a_k}$ stably. The recovered weights match the generating proportions $0.5, 0.3, 0.2$ and scikit-learn's solution; the tiny difference comes from scikit-learn's own stopping tolerance.

Choosing the number of clusters

Plain version. More clusters always fit the training data better. A model-selection criterion charges a price for every extra parameter, and the best K balances fit against that price.

BIC and AIC for mixtures

For a model with maximized likelihood $\hat{L}$, p free parameters and N data points,

$$\mathrm{BIC} = -2\ln\hat{L} + p\ln N, \qquad \mathrm{AIC} = -2\ln\hat{L} + 2p,$$

and smaller is better. The Bayesian information criterion (BIC) [12] comes from a large-sample approximation of the log marginal likelihood (model evidence, see PR 08); Akaike's information criterion (AIC) [13] estimates expected predictive log-loss. Because $\ln N > 2$ once $N \geq 8$, BIC penalizes complexity more and tends to choose fewer components.

For a full-covariance GMM in D dimensions,

$$p = \underbrace{(K-1)}_{\text{weights}} + \underbrace{KD}_{\text{means}} + \underbrace{K\,\tfrac{D(D+1)}{2}}_{\text{covariances}}.$$

The covariances dominate in high dimension: with $D = 128$ (a SIFT-like descriptor) each full covariance has 8,256 free numbers, which is why diagonal covariances are standard for large descriptors. The covariance *type* is therefore part of model selection too.

```
import numpy as np
from sklearn.datasets import make_blobs
from sklearn.mixture import GaussianMixture

X, _ = make_blobs(n_samples=600, centers=4, cluster_std=1.0, random_state=7)
X = X @ np.array([[1.0, 0.0], [0.6, 0.5]])        # shear: make clusters elliptical
results = []
for cov in ["spherical", "diag", "full"]:
    for K in range(1, 9):
        gm = GaussianMixture(K, covariance_type=cov, n_init=3, random_state=0).fit(X)
        results.append((gm.bic(X), gm.aic(X), cov, K))
for bic, aic, cov, K in sorted(results)[:5]:
    print(f"BIC={bic:8.1f}  AIC={aic:8.1f}  {cov:9s} K={K}")

# check the parameter count used by BIC for a full-covariance model
gm = GaussianMixture(4, covariance_type="full", random_state=0).fit(X)
D, K = X.shape[1], 4
p = (K - 1) + K * D + K * D * (D + 1) // 2
manual = -2 * gm.score(X) * len(X) + p * np.log(len(X))
print("free parameters:", p, " manual BIC:", round(manual, 1), " sklearn:", round(gm.bic(X),
 ↪  1))
```

```
BIC=  4301.1  AIC=  4200.0  full      K=4
BIC=  4320.7  AIC=  4193.2  full      K=5
BIC=  4357.6  AIC=  4203.7  full      K=6
BIC=  4388.4  AIC=  4208.2  full      K=7
BIC=  4414.9  AIC=  4243.4  diag      K=8
free parameters: 23  manual BIC: 4301.1  sklearn: 4301.1
```

BIC picks the true $K = 4$ with full covariances (the shear made the clusters elliptical, so spherical and diagonal models need extra components). AIC slightly prefers $K = 5$: its lighter penalty lets a fifth component buy a small likelihood gain. Neither criterion is "right" in general; both assume the model family is adequate, and for non-Gaussian clusters BIC will happily use several Gaussians to describe one curved group. If the goal is density estimation, that is fine; if each component is supposed to be one semantic group, it is not.

Other heuristics

The **elbow** method plots k-means J against K and looks for a bend; J always decreases, so the bend is subjective. The silhouette and stability approaches in the next section give alternatives. A Bayesian treatment can also start with more components than needed and let a prior on the mixing weights switch the unnecessary ones off; Bishop works through this for a variational Gaussian mixture [2].

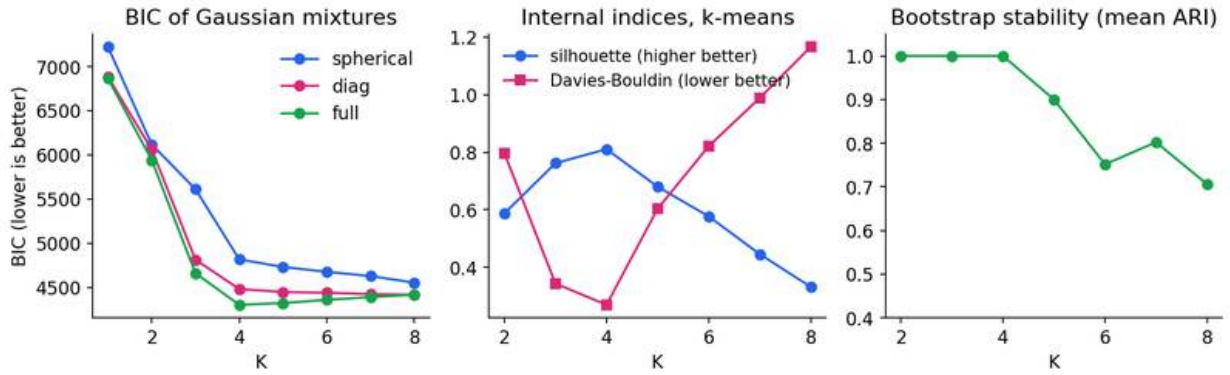


Figure 6 — Three ways to choose K on synthetic four-blob data. Left: BIC (sheared data). Middle: silhouette (higher is better) and Davies–Bouldin (lower is better) for k-means. Right: bootstrap stability is perfect for both $K = 3$ and $K = 4$, so stability alone cannot decide.

Cluster validity: is the clustering any good?

Plain version. Without labels, "good" has to be defined: tight and well separated (internal indices), agreeing with some trusted reference (external indices), or reproducible when the data are perturbed (stability). Theodoridis and Koutroumbas devote Chapter 16 to these questions [1]; PR 09 returns to validation in the supervised setting.

Internal indices

Silhouette [14]. For point i in cluster C, let $a(i)$ be its mean distance to the other points in C, and $b(i)$ the smallest mean distance from i to the points of any other cluster. Then

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}} \in [-1, 1].$$

$s(i)$ near 1 means i sits well inside its cluster; near 0, on a boundary; negative, probably in the wrong cluster. The mean silhouette over all points summarizes a partition, and the per-point silhouette plot shows which clusters are weak.

Davies–Bouldin [15]. Let s_i be the average distance of cluster i's points to its centroid c_i, and $d_{ij} = \|c_i - c_j\|$. Then

$$\mathrm{DB} = \frac{1}{K}\sum_{i=1}^{K}\max_{j\neq i}\frac{s_i + s_j}{d_{ij}},$$

the average "worst-case similarity" of each cluster to another. Lower is better.

Calinski–Harabasz [16]. With between-cluster scatter $B = \sum_k N_k (c_k - \bar{x})(c_k - \bar{x})^\top$ and within-cluster scatter $W = \sum_k \sum_{n \in C_k} (x_n - c_k)(x_n - c_k)^\top$ ($\bar{x}$ is the overall mean),

$$\mathrm{CH} = \frac{\operatorname{tr}(B)/(K-1)}{\operatorname{tr}(W)/(N-K)},$$

a variance-ratio statistic. Higher is better.

All three reward compact, convex, well-separated clusters in the chosen geometry. They will rate k-means' wrong answer on the two moons higher than the correct curved answer. They measure agreement with an assumed cluster *shape*, not semantic correctness.

External indices

When reference labels exist (for benchmarking, or for a separate held-out annotation), compare partitions with permutation-invariant indices.

Adjusted Rand index (ARI) [17]. The Rand index counts the fraction of point pairs on which two partitions agree (both "same cluster" or both "different cluster"). ARI subtracts the value expected under random labelings with the same cluster sizes and rescales:

$$\mathrm{ARI} = \frac{\mathrm{RI} - \mathbb{E}[\mathrm{RI}]}{\max \mathrm{RI} - \mathbb{E}[\mathrm{RI}]},$$

so random partitions score about 0 and identical partitions (up to relabeling) score 1.

Normalized mutual information (NMI). Treat the two labelings U and V as random variables over points; then

$$\mathrm{NMI}(U, V) = \frac{I(U;V)}{\frac{1}{2}\left(H(U) + H(V)\right)},$$

with mutual information I and entropies H (the arithmetic-mean normalization shown is scikit-learn's default; other normalizations exist). Vinh, Epps and Bailey [18] showed that unadjusted NMI is biased upward when the number of clusters is large relative to N, and recommend chance-adjusted versions such as adjusted mutual information (AMI).

If the reference labels were used to tune the algorithm or K, they are no longer an independent check.

Stability

Refit the clustering on bootstrap samples or perturbed data and compare each new partition with the original on the *same* points, using ARI. A K that gives unstable partitions is suspect. But

stability is necessary, not sufficient: merging two nearby blobs is perfectly reproducible (Figure 6, right, $K = 3$), and a stable partition can still reflect an irrelevant factor such as the camera or the lighting.

Code: validity indices and stability

```
import numpy as np
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
from sklearn.metrics import (silhouette_score, davies_bouldin_score,
                             calinski_harabasz_score, adjusted_rand_score,
                             normalized_mutual_info_score)

X, truth = make_blobs(n_samples=600, centers=4, cluster_std=1.0, random_state=7)
print(" K  silhouette(+)  Davies-Bouldin(-)  Calinski-Harabasz(+)")
for K in range(2, 8):
    z = KMeans(K, n_init=10, random_state=0).fit_predict(X)
    print(f"{K:2d}  {silhouette_score(X, z):12.3f}  {davies_bouldin_score(X, z):17.3f}"
          f"  {calinski_harabasz_score(X, z):20.1f}")

z = KMeans(4, n_init=10, random_state=0).fit_predict(X)
print("external ARI:", round(adjusted_rand_score(truth, z), 3),
      " NMI:", round(normalized_mutual_info_score(truth, z), 3))
perm = np.array([2, 0, 3, 1])[z]                       # relabel clusters
print("ARI after relabelling:", round(adjusted_rand_score(truth, perm), 3))

# stability: refit on bootstrap samples, compare on the original points
rng = np.random.default_rng(1)
for K in [3, 4, 5, 6]:
    base = KMeans(K, n_init=10, random_state=0).fit(X)
    scores = []
    for b in range(10):
        idx = rng.integers(0, len(X), len(X))
        boot = KMeans(K, n_init=10, random_state=b).fit(X[idx])
        scores.append(adjusted_rand_score(base.predict(X), boot.predict(X)))
    print(f"K={K}: bootstrap stability (mean ARI) = {np.mean(scores):.3f}")
```

```
 K  silhouette(+)  Davies-Bouldin(-)  Calinski-Harabasz(+)
 2         0.588              0.796                850.9
 3         0.762              0.343               2235.5
 4         0.811              0.270               7641.8
 5         0.680              0.603               6341.0
 6         0.577              0.821               5620.2
 7         0.446              0.988               5304.4
external ARI: 1.0  NMI: 1.0
ARI after relabelling: 1.0
K=3: bootstrap stability (mean ARI) = 1.000
K=4: bootstrap stability (mean ARI) = 1.000
K=5: bootstrap stability (mean ARI) = 0.858
K=6: bootstrap stability (mean ARI) = 0.699
```

On these well-separated blobs every internal index agrees on $K = 4$, and the external indices confirm a perfect match regardless of how the clusters are numbered. Real data are rarely this kind: indices frequently disagree, and the disagreement is informative, because it tells you the data do not have one unambiguous cluster structure at the chosen scale.

Modern view

Plain version. The algorithms in this chapter are decades old and still everywhere. What changed is the *representation*: today we usually cluster learned embeddings rather than raw pixels or hand-made descriptors, and deep networks are sometimes trained *by* clustering.

Surveys worth reading

- **Xu and Wunsch, "Survey of clustering algorithms" (2005)** [19] is the classic broad survey from a pattern-recognition and neural-network viewpoint. It reviews proximity measures, then organizes algorithms into hierarchical, squared-error (k-means family), mixture-density, graph-theoretic, fuzzy, neural (self-organizing maps, adaptive resonance), kernel, sequential-data, large-scale and high-dimensional methods, and closes with cluster validity. It is a good map of the classical landscape.
- **Jain, "Data clustering: 50 years beyond K-means" (2010)** [20] is a shorter, opinionated overview. Its main messages remain current: there is no universally best algorithm; the representation and the definition of a cluster matter more than the algorithm; the number of clusters and validity are open questions; and side information (pairwise constraints, semi-supervised clustering) and clustering ensembles are valuable directions.
- **Ikotun et al., "K-means clustering algorithms: A comprehensive review, variants analysis, and advances in the era of big data" (2023)** [21] reviews the many k-means variants: initialization (including k-means++), automatic choice of K, handling of mixed or high-dimensional data, and scalable and distributed implementations.
- **Zhou et al., "A comprehensive survey on deep clustering: Taxonomy, challenges, and future directions" (ACM Computing Surveys, 2024)** [22] organizes deep clustering by how the representation-learning module and the clustering module interact: multi-stage methods (learn features, then cluster), iterative methods (alternate pseudo-labels and feature learning), generative methods (latent-variable models such as mixture-prior autoencoders), and simultaneous methods (optimize both jointly). It also discusses benchmarks, metrics and applications.
- **Ren et al., "Deep clustering: A comprehensive survey" (IEEE TNNLS, 2025)** [23] organizes the field by data source and prior knowledge: single-view, semi-supervised, multi-view and transfer deep clustering. Read together with [22], it shows how wide the deep-clustering design space has become.

From visual words to self-labeling

Two influential vision papers show how classical clustering became a training signal. **DeepCluster** [24] alternates between running k-means on the features of a convolutional network and training that network to predict the cluster assignments as pseudo-labels; repeated over the whole data set, this learns useful features from unlabeled images. **SCAN** [25] separates representation learning from clustering: it first learns features with a self-supervised pretext task, then mines each image's nearest neighbors in that feature space and trains a clustering head so that neighbors receive the same cluster, followed by a self-labeling refinement. Both are applications of the ideas in this chapter (k-means as quantization, neighborhoods as evidence of similarity) to learned representations; ML 06 explains where such representations come from.

State of practice (2023–2026) and open problems

In current practice, a common recipe for grouping images is: extract embeddings from a pretrained (often self-supervised) encoder, optionally reduce dimension, then run k-means, a Gaussian mixture or HDBSCAN, and validate on a labeled subset. The chapter's warnings apply unchanged:

- **The representation defines similarity.** Embeddings trained for one purpose may cluster by style, background or camera rather than by the category you care about.
- K **and validity remain open.** Information criteria assume the model family; internal indices assume a shape; external indices need labels. No survey above claims a general solution.
- **Evaluation of deep clustering usually relies on labeled benchmarks** and permutation-invariant scores such as clustering accuracy after optimally matching clusters to classes, NMI and ARI [22]. Those scores measure agreement with one chosen labeling, not the "true" structure.
- **Scale and imbalance.** Long-tailed data, many small clusters, and streaming data stress both classical and deep methods; [21] and [22] list these as active problems.

Key takeaways

- Clustering is defined by its notion of similarity. Features, scaling and distance are modeling decisions made before any algorithm runs.
- K-means (Lloyd) alternates nearest-center assignment and mean update; J decreases monotonically to a local minimum. k-means++ seeding gives $\mathbb{E}[J] \leq 8(\ln K + 2)J_{\mathrm{OPT}}$ and far fewer bad runs; use several restarts anyway.
- Linkage defines hierarchical clustering: single linkage chains along thin bridges, complete and Ward prefer compact groups. The dendrogram must still be cut.
- DBSCAN defines clusters as density-connected regions, finds arbitrary shapes and labels noise; ε comes from the k-distance curve, and HDBSCAN handles varying density.
- EM for Gaussian mixtures follows from the complete-data log-likelihood: E step responsibilities, M step weighted means and covariances, $\pi_k = N_k/N$ via a Lagrange multiplier. The ELBO

decomposition proves monotone improvement; singularities and label switching need explicit care; k-means is the hard, isotropic limit.
- BIC $= -2\ln\hat{L} + p\ln N$ and AIC $= -2\ln\hat{L} + 2p$ trade fit against parameter count; covariance type is part of the choice.
- Internal indices (silhouette, Davies–Bouldin, Calinski–Harabasz) reward an assumed geometry; external indices (ARI, NMI) need independent labels; stability is necessary but not sufficient.

Exercises

1. **Lloyd's algorithm never increases J.** Prove that both the assignment step and the update step cannot increase J, and use this to argue that the algorithm terminates. What goes wrong with the termination argument if ties are broken randomly?

   Hint

   The assignment step minimizes each term $\|x_n - \mu_{z_n}\|^2$ separately; the update step uses $\sum_{n\in C_k}\|x_n - \mu\|^2 = \sum_{n\in C_k}\|x_n - \bar{x}_k\|^2 + N_k\|\bar{x}_k - \mu\|^2$. There are at most K^N partitions. With random tie-breaking, two partitions of equal J could alternate forever; a fixed rule (for example, keep the current cluster on ties) prevents that.

2. **k-means++ by hand.** One-dimensional data are $\{0, 1, 2, 10, 11\}$ and $K = 2$. The first seed is 0. Compute the probability that each point becomes the second seed. Then compare with uniform sampling of the second seed.

   Hint

   D^2 values are $0, 1, 4, 100, 121$, summing to 226. So $P(10) = 100/226$ and $P(11) = 121/226$; together about 0.98. Under uniform sampling the second seed lands in the right-hand group with probability only $2/5$ (or $2/4$ if the first seed is excluded).

3. **The M step for a 1-D mixture.** For a one-dimensional two-component mixture with variances σ_1^2, σ_2^2, derive the M-step update of σ_k^2 from Q directly, without matrix calculus. Then show that if a component's responsibilities concentrate on one point x_n (that is, $r_{nk} \approx 1$ for that point and ≈ 0 for the others), the update drives σ_k^2 toward 0. What does scikit-learn's `reg_covar` do about it?

   Hint

   Differentiate $\sum_n r_{nk}[-\frac{1}{2}\ln\sigma_k^2 - (x_n-\mu_k)^2/(2\sigma_k^2)]$ with respect to σ_k^2 to get $\sigma_k^2 = \sum_n r_{nk}(x_n-\mu_k)^2/N_k$. With all weight on one point, $\mu_k = x_n$ and the sum is zero. `reg_covar` adds a small constant to the diagonal after each M step, so the variance stays bounded away from zero.

4. **Counting parameters.** For local descriptors with $D = 64$ and $K = 32$ components, count the free parameters of a GMM with full, diagonal and spherical covariances. With $N = 10^5$ descriptors, how large is the BIC penalty $p \ln N$ in each case?

   Hint

   Full: $31 + 32 \cdot 64 + 32 \cdot 64 \cdot 65/2 = 31 + 2048 + 66560 = 68{,}639$. Diagonal: $31 + 2048 + 2048 = 4{,}127$. Spherical: $31 + 2048 + 32 = 2{,}111$. Multiply by $\ln 10^5 \approx 11.51$. The full model has to improve $-2 \ln \hat{L}$ by roughly 7.4×10^5 over the diagonal one just to break even.

5. **When indices mislead.** Run k-means with $K = 2$ and DBSCAN on the two-moons data from the comparison code. Compute the silhouette score of each partition and their ARI against the generating labels. Explain the disagreement in terms of what each index measures.

   Hint

   Silhouette is computed from Euclidean distances and favors convex, separated groups, so the k-means cut usually scores higher than the two moons even though its ARI is about 0.26 versus 1.0 for DBSCAN. Internal indices encode a shape assumption; the "right" answer here is defined by connectivity.

6. **Stability is not truth.** Generate four Gaussian blobs where two are very close (for example centers $(0, 0)$ and $(1.2, 0)$ with standard deviation 0.5). Compute bootstrap stability for $K = 3$ and $K = 4$. Which is more stable? Which matches the generating process? What extra information would settle it?

   Hint

   Merging the close pair is very reproducible, so $K = 3$ can be as stable as or more stable than $K = 4$. The generating process has four groups, but whether they *should* be separated depends on the task. Domain knowledge, a labeled subset, or a density-based criterion such as BIC on a GMM gives additional evidence.

References

1. S. Theodoridis and K. Koutroumbas, *Pattern Recognition*, 4th ed., Academic Press, 2009, Ch. 11–16 (clustering basic concepts, sequential, hierarchical, function-optimization and graph-based algorithms, cluster validity). doi
2. C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006, Ch. 9 (k-means, mixtures of Gaussians, EM, the general EM algorithm). book page
3. S. P. Lloyd, "Least squares quantization in PCM," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982. doi
4. J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proc. Fifth Berkeley Symposium on Mathematical Statistics and Probability*, vol. 1, pp. 281–297, 1967. Project Euclid

5. D. Arthur and S. Vassilvitskii, “k-means++: The advantages of careful seeding,” in *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2007, pp. 1027–1035. pdf
6. D. Sculley, “Web-scale k-means clustering,” in *Proc. 19th International Conference on World Wide Web (WWW)*, 2010, pp. 1177–1178. doi
7. J. H. Ward, “Hierarchical grouping to optimize an objective function,” *Journal of the American Statistical Association*, vol. 58, pp. 236–244, 1963. doi
8. M. Ester, H.-P. Kriegel, J. Sander and X. Xu, “A density-based algorithm for discovering clusters in large spatial databases with noise,” in *Proc. 2nd International Conference on Knowledge Discovery and Data Mining (KDD)*, 1996. pdf
9. R. J. G. B. Campello, D. Moulavi and J. Sander, “Density-based clustering based on hierarchical density estimates,” in *Advances in Knowledge Discovery and Data Mining (PAKDD)*, LNCS, 2013, pp. 160–172. doi
10. U. von Luxburg, “A tutorial on spectral clustering,” *Statistics and Computing*, vol. 17, pp. 395–416, 2007. doi
11. A. P. Dempster, N. M. Laird and D. B. Rubin, “Maximum likelihood from incomplete data via the EM algorithm,” *Journal of the Royal Statistical Society: Series B*, vol. 39, no. 1, pp. 1–22, 1977. doi
12. G. Schwarz, “Estimating the dimension of a model,” *The Annals of Statistics*, vol. 6, no. 2, 1978. doi
13. H. Akaike, “A new look at the statistical model identification,” *IEEE Transactions on Automatic Control*, vol. 19, no. 6, pp. 716–723, 1974. doi
14. P. J. Rousseeuw, “Silhouettes: A graphical aid to the interpretation and validation of cluster analysis,” *Journal of Computational and Applied Mathematics*, vol. 20, pp. 53–65, 1987. doi
15. D. L. Davies and D. W. Bouldin, “A cluster separation measure,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-1, no. 2, pp. 224–227, 1979. doi
16. T. Caliński and J. Harabasz, “A dendrite method for cluster analysis,” *Communications in Statistics*, vol. 3, no. 1, pp. 1–27, 1974. doi
17. L. Hubert and P. Arabie, “Comparing partitions,” *Journal of Classification*, vol. 2, pp. 193–218, 1985. doi
18. N. X. Vinh, J. Epps and J. Bailey, “Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance,” *Journal of Machine Learning Research*, vol. 11, pp. 2837–2854, 2010. JMLR
19. R. Xu and D. Wunsch II, “Survey of clustering algorithms,” *IEEE Transactions on Neural Networks*, vol. 16, no. 3, pp. 645–678, 2005. doi
20. A. K. Jain, “Data clustering: 50 years beyond K-means,” *Pattern Recognition Letters*, vol. 31, no. 8, pp. 651–666, 2010. doi
21. A. M. Ikotun, A. E. Ezugwu, L. Abualigah, B. Abuhaija and J. Heming, “K-means clustering algorithms: A comprehensive review, variants analysis, and advances in the era of big data,” *Information Sciences*, vol. 622, pp. 178–210, 2023. doi
22. S. Zhou, H. Xu, Z. Zheng, J. Chen, Z. Li, J. Bu, J. Wu, X. Wang, W. Zhu and M. Ester, “A com-

prehensive survey on deep clustering: Taxonomy, challenges, and future directions," *ACM Computing Surveys*, vol. 57, no. 3, pp. 1–38, 2024 (arXiv:2206.07579). doi

23. Y. Ren, J. Pu, Z. Yang, J. Xu, G. Li, X. Pu, P. S. Yu and L. He, "Deep clustering: A comprehensive survey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 36, no. 4, pp. 5858–5878, 2025 (arXiv:2210.04142). doi
24. M. Caron, P. Bojanowski, A. Joulin and M. Douze, "Deep clustering for unsupervised learning of visual features," in *Proc. European Conference on Computer Vision (ECCV)*, LNCS, 2018, pp. 139–156. doi
25. W. Van Gansbeke, S. Vandenhende, S. Georgoulis, M. Proesmans and L. Van Gool, "SCAN: Learning to classify images without labels," in *Proc. European Conference on Computer Vision (ECCV)*, LNCS, 2020, pp. 268–285. doi
26. scikit-learn developers, "GaussianMixture," API documentation, accessed 2026-11-05. documentation

PR 08 • Approximate Inference and Bayesian Model Complexity

Why posterior integrals are hard, and the two big families of answers: sampling (Monte Carlo, importance sampling, Metropolis–Hastings, Gibbs, HMC) and optimization (variational inference, Laplace, EP), with links to VAEs and diffusion models.

Online edition: https://blog.shyandram.dev/en/learn/pattern-recognition/approximate-inference/

What you'll learn

- Why Bayesian posteriors are easy to *evaluate* up to a constant but hard to *integrate*, and why grids fail beyond a handful of dimensions.
- How plain Monte Carlo, rejection sampling and importance sampling turn integrals into averages, and how to tell when their weights have collapsed.
- How Markov chain Monte Carlo works: detailed balance, Metropolis–Hastings (including the asymmetric-proposal correction), Gibbs sampling, Hamiltonian Monte Carlo, and the diagnostics ($\hat{R}$, effective sample size) that keep you honest.
- How variational inference turns integration into optimization: the evidence lower bound (ELBO) derived two ways, mean-field coordinate ascent, reverse versus forward KL, and stochastic gradients with the reparameterization trick.
- Where Laplace approximation and expectation propagation fit, how model evidence penalizes complexity, and how the same ideas power VAEs and diffusion models.

The big picture

Earlier chapters fitted one "best" parameter setting: maximum likelihood, MAP, or EM for mixtures. A Bayesian recognizer asks a bigger question. After seeing the data, *how plausible is every parameter setting*, and what do they predict *on average*? Answering it requires integrals over all unknowns. For almost every useful model those integrals have no closed form. This chapter is about the two families of workarounds: **draw samples** that behave like the posterior, or **fit a simpler distribution** that sits close to it.

> **Explain like I'm five**
>
> Imagine a huge dark field with treasure buried everywhere in different amounts, and you want to know the *total* treasure. Finding the single richest spot is easy: walk uphill. That is optimization. Knowing the total is harder. One way is to walk around randomly but cleverly, spending more time where treasure is rich, and count what you see: that is **sampling**. Another way is to lay a simple, smooth blanket over the field shaped as much like the treasure map as possible, then measure the

blanket instead: that is **variational inference**. Walking is slow but honest. The blanket is fast but can hide bumps it was not shaped to show.

Why it matters for recognition: uncertainty tells a system when to abstain (see PR 01), integrated evidence compares models without a held-out set, and the machinery of this chapter, especially the ELBO and Langevin-style sampling, is exactly what modern generative models use. We follow Bishop's chapters on approximate inference and sampling [1]; Theodoridis and Koutroumbas supply the estimation and optimization background (Chapter 2 and Appendix C) rather than a one-to-one treatment of these methods [2].

Why exact inference is hard

Plain version. Bayes' rule says the posterior is "prior times likelihood, divided by a normalizing number". The top is usually easy to compute for any single guess. The bottom is a sum over *every possible* guess, and that is where the trouble starts.

Precise version. Collect all unknowns, latent variables and parameters, into z, and write D for the observed data. Then

$$p(z \mid D) = \frac{p(D, z)}{p(D)}, \qquad p(D) = \int p(D, z)\, dz,$$

where $p(D, z) = p(D \mid z)\, p(z)$ is the joint density (likelihood times prior) and $p(D)$ is the **evidence** or marginal likelihood. Almost everything we want is an expectation under the posterior:

$$\mathbb{E}_{p(z|D)}[f(z)] = \int f(z)\, p(z \mid D)\, dz,$$

for example the posterior mean ($f(z) = z$), a predictive probability ($f(z) = p(y^* \mid x^*, z)$) or the probability that a parameter is positive ($f(z) = \mathbf{1}[z > 0]$). We write $\tilde{p}(z) = p(D, z)$ for the **unnormalized** posterior: it is cheap to evaluate pointwise, but its normalizer $Z = p(D)$ is unknown.

Three things make the integral hard:

1. **Dimension.** A grid with G points per axis needs G^d evaluations in d dimensions. For $G = 201$ and $d = 50$ that is about 10^{115}, far beyond any computer.
2. **Concentration.** With a lot of data the posterior occupies a tiny region of a huge space. Random guesses almost never land in it.
3. **Discrete structure.** For a mixture with K components and N points, the latent assignments have K^N configurations; summing over them exactly is hopeless even when each term is simple.

MAP estimation sidesteps all of this by maximizing $\tilde{p}$, but a maximum is not an integral. The highest point of the posterior can sit in a narrow spike that carries little mass, and a single point carries no information about spread. EM, from PR 07, sits in between: it integrates over latent assignments in its E step (which is tractable for Gaussian mixtures) but still returns a point estimate of the parameters.

A worked example: Bayesian logistic regression

Take a two-class problem with inputs $x_n \in \mathbb{R}^2$ (a bias feature and one measurement), labels $y_n \in \{0, 1\}$, weights w and the model

$$p(y_n = 1 \mid x_n, w) = \sigma(w^\top x_n), \qquad \sigma(s) = \frac{1}{1 + e^{-s}}, \qquad p(w) = \mathcal{N}(0, 4I).$$

The log joint is $\sum_n \left[y_n w^\top x_n - \log(1 + e^{w^\top x_n}) \right] - \frac{1}{8}\|w\|^2$ plus a constant. The Gaussian prior is not conjugate to the logistic likelihood, so the posterior is *not* Gaussian and its normalizer has no formula. With only two weights we can still brute-force it on a grid, which gives us a ground truth to compare every approximation against.

```
import numpy as np

rng = np.random.default_rng(0)
n = 40
X = np.c_[np.ones(n), rng.normal(size=n)]
w_true = np.array([-0.5, 1.5])
y = rng.random(n) < 1 / (1 + np.exp(-X @ w_true))

def log_joint(w):                       # log p(y | X, w) + log p(w), w ~ N(0, 2^2 I)
    s = X @ w
    return np.sum(y * s - np.logaddexp(0, s)) - 0.5 * np.sum(w**2) / 4.0

# Brute-force normalization on a grid: fine in 2-D, hopeless in 50-D.
g = np.linspace(-4, 4, 201)
W0, W1 = np.meshgrid(g, g, indexing="ij")
logp = np.array([[log_joint(np.array([a, b])) for b in g] for a in g])
cell = (g[1] - g[0])**2
log_evidence = np.log(np.sum(np.exp(logp - logp.max())) * cell) + logp.max()
post = np.exp(logp - log_evidence)
mean = np.array([np.sum(W0 * post), np.sum(W1 * post)]) * cell
sd = np.sqrt(np.array([np.sum((W0 - mean[0])**2 * post),
                       np.sum((W1 - mean[1])**2 * post)]) * cell)
print("log evidence:", round(log_evidence, 3))
print("posterior mean:", mean.round(3), " sd:", sd.round(3))
print("grid points for 201 per axis in 50-D:", f"{201.0**50:.2e}")
```

It prints a log evidence of -22.117, a posterior mean of $(-0.303, 1.631)$ and standard deviations of $(0.366, 0.578)$. Note the subtraction of `logp.max()` before exponentiating: log densities of real data are large negative numbers, and working in log space (the "log-sum-exp" trick) is the first habit of every practitioner. We will reuse this model to judge the Laplace and variational approximations later in the chapter.

Monte Carlo: integrals as averages

Plain version. If you can draw random examples from a distribution, the average of a quantity over those examples estimates its expected value. More draws give a better estimate, and the error

shrinks in a predictable way that does not depend on the dimension.

Precise version. Given independent draws $z^{(1)}, \dots, z^{(S)} \sim p$, the Monte Carlo estimator of $I = \mathbb{E}_p[f(z)]$ is

$$\hat{I}_S = \frac{1}{S}\sum_{s=1}^{S} f(z^{(s)}), \qquad \mathbb{E}[\hat{I}_S] = I, \qquad \mathrm{Var}[\hat{I}_S] = \frac{\mathrm{Var}_p[f(z)]}{S}.$$

Here S is the number of samples and $\mathrm{Var}_p[f]$ is the variance of f under p. Assuming $\mathbb{E}_p[|f|]$ is finite, the estimator is **unbiased**; the displayed variance and the $1/\sqrt{S}$ standard-error statement additionally require finite $\mathrm{Var}_p[f]$: to halve the error you need four times as many samples. The striking part is that d does not appear. The rate exponent is dimension independent, but the variance of f and the cost of obtaining each draw can grow with dimension. Heavy-tailed quantities with infinite variance do not have this standard-error guarantee.

The catch is the assumption "draws from p". For a posterior we usually cannot sample directly. The rest of the sampling half of this chapter is about manufacturing such draws, or correcting for draws from the wrong distribution.

Rejection sampling

Suppose we can sample from a simple **proposal** $q(z)$ and know a constant M with $M\,q(z) \geq \tilde{p}(z)$ for all z. Draw $z \sim q$ and $u \sim \mathrm{Uniform}(0,1)$, and keep z if $u < \tilde{p}(z)/(Mq(z))$. The kept points are exact draws from p, because the accepted region under the envelope Mq has exactly the shape of $\tilde{p}$. The acceptance probability is Z/M, where $Z = \int \tilde{p}$. In high dimensions a usable envelope must be enormous compared with the target, so almost everything is rejected. Rejection sampling is a building block for one-dimensional conditionals (for example inside Gibbs samplers), not a general posterior engine [1].

Importance sampling

Plain version. Draw from a convenient distribution, then correct each draw with a weight that says how much more (or less) likely it is under the distribution you actually care about. Works well when the convenient distribution covers the important region; fails quietly when it does not.

Precise version. For any q with $q(z) > 0$ wherever $p(z)f(z) \neq 0$,

$$\mathbb{E}_p[f(z)] = \int f(z)\,\frac{p(z)}{q(z)}\,q(z)\,dz \approx \frac{1}{S}\sum_{s=1}^{S} w_s f(z^{(s)}), \qquad w_s = \frac{p(z^{(s)})}{q(z^{(s)})}, \quad z^{(s)} \sim q.$$

The ratio w_s is the **importance weight**. When only $\tilde{p}$ is available, the unknown Z cancels if we normalize the weights:

$$\hat{I}_{\mathrm{SN}} = \sum_{s=1}^{S} \bar{w}_s f(z^{(s)}), \qquad \bar{w}_s = \frac{\tilde{w}_s}{\sum_t \tilde{w}_t}, \qquad \tilde{w}_s = \frac{\tilde{p}(z^{(s)})}{q(z^{(s)})}.$$

For self-normalization and evidence estimation, strengthen the support condition to $q(z) > 0$ wherever $\tilde{p}(z) > 0$, including regions where $f(z) = 0$: the denominator must account for the whole target mass. Otherwise normalization estimates a truncated target rather than the intended posterior.

This **self-normalized** estimator is generally biased for finite S but consistent as $S \to \infty$. As a by-product, $\frac{1}{S}\sum_s \tilde{w}_s$ is an unbiased estimate of the evidence Z, which is how importance sampling is used for model comparison.

The quality of the estimate depends on how uneven the weights are. A widely used summary is the **effective sample size** of the weights,

$$\mathrm{ESS} = \frac{1}{\sum_s \bar{w}_s^2},$$

which equals S when all weights are equal and 1 when one weight takes everything. Two failure modes matter:

- **Proposal too narrow or misplaced.** Regions where p has mass but q is tiny produce rare, gigantic weights. The estimate is dominated by a handful of draws and its variance can even be infinite.
- **Light tails.** The ratio p/q must stay bounded in the tails. A rule of thumb is to make q *heavier-tailed* than p (a Student-t instead of a Gaussian, or a wider Gaussian).

```
import numpy as np
from scipy.stats import norm

rng = np.random.default_rng(1)
N = 10_000
exact = norm.sf(3.0)                      # P(X > 3) for X ~ N(0, 1)

# Plain Monte Carlo: most samples never reach the tail.
x = rng.normal(0, 1, N)
plain = np.mean(x > 3)

# Importance sampling with a proposal centered on the tail, q = N(3, 1).
z = rng.normal(3, 1, N)
w = norm.pdf(z) / norm.pdf(z, loc=3)      # p(z) / q(z)
is_est = np.mean(w * (z > 3))
print(f"exact {exact:.6f}  plain MC {plain:.6f}  importance {is_est:.6f}")

# Self-normalized IS for an unnormalized target, with effective sample size.
def log_p_tilde(t):                       # N(1.5, 0.7^2) without its constant
    return -0.5 * ((t - 1.5) / 0.7) ** 2

for scale in [1.0, 3.0, 0.3]:
    t = rng.normal(0, scale, N)
    log_w = log_p_tilde(t) - norm.logpdf(t, scale=scale)
    wn = np.exp(log_w - log_w.max())
    wn /= wn.sum()
    ess = 1.0 / np.sum(wn**2)
    print(f"proposal N(0,{scale}^2): mean {np.sum(wn * t):.3f}  ESS {ess:7.1f} of {N}")
```

The rare-event estimate shows why importance sampling exists: the exact tail probability is 0.001350; plain Monte Carlo with 10,000 draws gives 0.001900 (only 19 draws landed in the tail), while the shifted proposal gives 0.001372. In the second part the target mean is 1.5. The wide proposals give 1.509 and 1.492 with ESS of about 1,870 and 2,830. The narrow proposal $\mathcal{N}(0, 0.3^2)$ gives 0.921 with an ESS of 13, a confident-looking but wrong answer. A tiny ESS is the alarm bell.

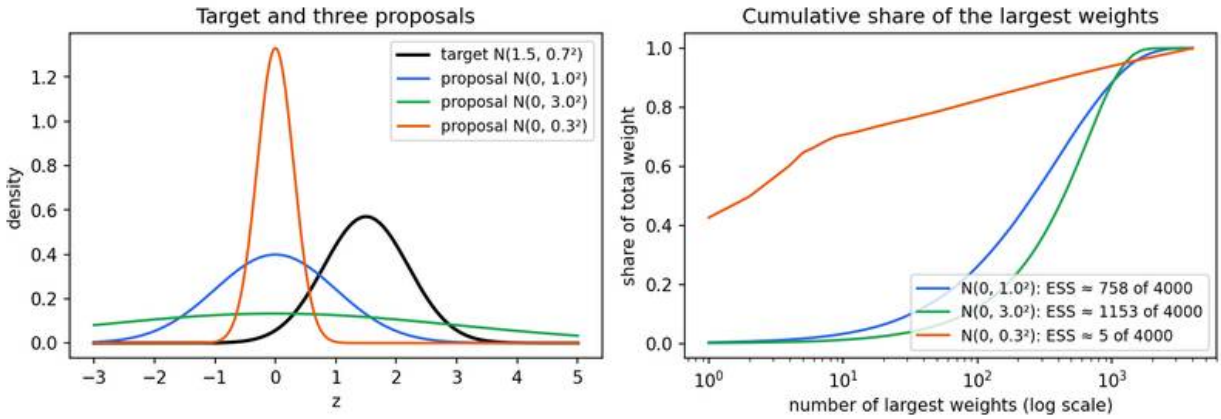


Figure 1 — Importance sampling with 4,000 draws per proposal. The narrow proposal (orange) barely reaches the target, so its largest single weight already holds over 40% of the total and the ESS is about 5. The wide proposals spread the weight over hundreds of draws. Computed by this tutorial's figure script.

Importance sampling stays important even when it is not the main engine: it re-weights samples from an old model to a new one, it estimates evidence, and it powers **sequential Monte Carlo** (particle filters), where weights are refreshed and resampled step by step as data arrive.

Markov chain Monte Carlo

Plain version. Instead of drawing independent samples, take a random walk whose rules are designed so that, in the long run, the fraction of time spent in each region equals the posterior probability of that region. Then time averages along the walk estimate posterior expectations. Successive steps are correlated, so a long walk is worth fewer independent draws than its length suggests.

Precise version. A Markov chain with transition kernel $T(z' \mid z)$ has p as a **stationary** (invariant) distribution if

$$\int T(z' \mid z)\, p(z)\, dz = p(z').$$

A simple sufficient condition is **detailed balance** (reversibility):

$$p(z)\, T(z' \mid z) = p(z')\, T(z \mid z') \quad \text{for all } z, z'.$$

Integrating both sides over z gives stationarity. If the chain can also reach every region (irreducibility) and does not cycle (aperiodicity), the distribution of the state converges to p from

any starting point, and the ergodic average $\frac{1}{S}\sum_s f(z^{(s)})$ converges to $\mathbb{E}_p[f]$. Importantly, detailed balance involves only *ratios* $p(z')/p(z)$, so the unknown normalizer cancels.

Metropolis–Hastings

The Metropolis algorithm [3], generalized by Hastings [4], builds such a kernel from any proposal $q(z' \mid z)$:

1. From the current state z, propose $z' \sim q(z' \mid z)$.
2. Accept with probability

$$A(z', z) = \min\left(1,\ \frac{\tilde{p}(z')\, q(z \mid z')}{\tilde{p}(z)\, q(z' \mid z)}\right).$$

3. If accepted, move to z'; otherwise **stay at** z **and record** z **again**.

Here $q(z \mid z')$ is the density of proposing the reverse move. The factor $q(z \mid z')/q(z' \mid z)$ is the **Hastings correction**: it cancels any bias the proposal has toward certain directions. For a symmetric proposal such as a Gaussian random walk, $q(z' \mid z) = q(z \mid z')$ and the rule reduces to Metropolis's original $\min(1, \tilde{p}(z')/\tilde{p}(z))$.

To see why it works, take $z' \neq z$ and suppose $\tilde{p}(z')q(z \mid z') \leq \tilde{p}(z)q(z' \mid z)$. Then $p(z)T(z' \mid z) = p(z)\, q(z' \mid z)\, \frac{p(z')q(z|z')}{p(z)q(z'|z)} = p(z')q(z \mid z')$, and in the reverse direction the acceptance is 1, so $p(z')T(z \mid z') = p(z')q(z \mid z')$. Both sides match: detailed balance holds.

Forgetting step 3 (only recording accepted states) is a classic bug: it over-represents regions the chain moves through easily and biases every estimate.

A Gaussian target with known moments lets us check a random-walk chain. The assertions are deliberately loose: a numerical sanity check, not a convergence proof.

```
import numpy as np

rng = np.random.default_rng(12)
def log_target(x):
    return -0.5 * ((x - 1.5) / 0.7)**2
x, accepted = -3.0, 0
samples = []
for step in range(18000):
    proposal = x + rng.normal(0, 0.8)
    if np.log(rng.random()) < min(0.0, log_target(proposal) - log_target(x)):
        x = proposal
        accepted += 1
    if step >= 2000:                      # discard warm-up, keep repeats
        samples.append(x)
samples = np.asarray(samples)
rho1 = np.corrcoef(samples[:-1], samples[1:])[0, 1]
print("mean, std:", samples.mean(), samples.std())
print("acceptance:", accepted / 18000, "lag-one correlation:", rho1)
```

```
assert abs(samples.mean() - 1.5) < 0.12
assert abs(samples.std() - 0.7) < 0.12
```

The chain recovers mean 1.50 and standard deviation 0.69, with acceptance 0.67 and lag-one correlation 0.74. The step size is the main tuning knob, and acceptance alone does not tell you whether it is good. Tiny steps are almost always accepted but crawl; huge steps are mostly rejected and the chain sits still. Both produce strongly correlated samples.

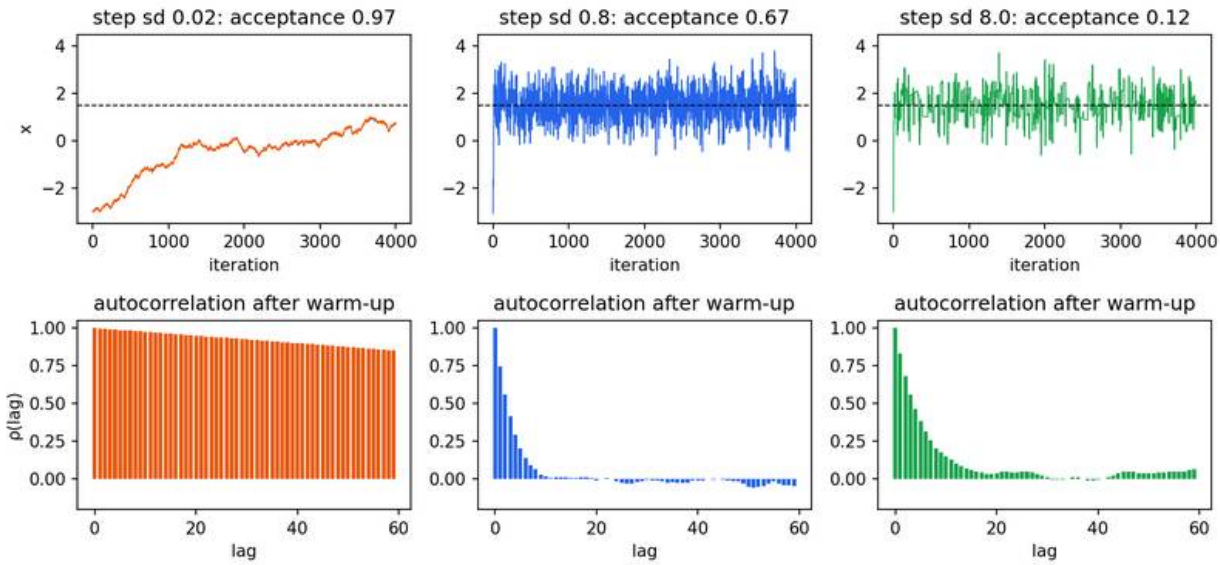


Figure 2 — Random-walk Metropolis on $N(1.5, 0.7^2)$ started at -3. With step 0.02 (acceptance 0.97) the chain has not even reached the target after 4,000 steps and its autocorrelation barely decays. Step 0.8 mixes well. Step 8 (acceptance 0.12) gets stuck for long flat stretches. Computed by this tutorial's figure script.

Gibbs sampling

Plain version. Update one variable (or block) at a time by drawing it from its distribution *given all the others*. If those conditional distributions are easy, as they often are in conjugate models, every move is accepted.

Precise version. Gibbs sampling, introduced for image restoration with Markov random fields by Geman and Geman [5], cycles through blocks $j = 1, \dots, J$ and draws

$$z_j \sim p(z_j \mid z_{-j}, D),$$

where z_{-j} denotes all blocks except j. It is a special case of Metropolis–Hastings with proposal $q(z' \mid z) = p(z'_j \mid z_{-j})$ and $z'_{-j} = z_{-j}$; plugging into the acceptance ratio gives exactly 1 [1]. In a Gaussian mixture, for example, one alternates between sampling each point's component label from its responsibilities and sampling the component parameters from their conjugate posteriors: a stochastic cousin of EM.

For a zero-mean bivariate Gaussian with unit variances and correlation ρ, the conditionals are $z_1 \mid z_2 \sim \mathcal{N}(\rho z_2,\, 1-\rho^2)$ and symmetrically for z_2. The code below runs four chains from dispersed corners and computes the **split-$\hat{R}$** convergence diagnostic described in the next subsection.

```
import numpy as np

rng = np.random.default_rng(3)
rho = 0.85                      # target: zero-mean bivariate Gaussian, unit variances
sd = np.sqrt(1 - rho**2)        # conditional standard deviation

def gibbs(n_steps, start):
    z = np.array(start, dtype=float)
    out = np.empty((n_steps, 2))
    for t in range(n_steps):
        z[0] = rng.normal(rho * z[1], sd)   # z1 | z2 ~ N(rho z2, 1 - rho^2)
        z[1] = rng.normal(rho * z[0], sd)   # z2 | z1 ~ N(rho z1, 1 - rho^2)
        out[t] = z
    return out

def split_rhat(chains):         # chains: (m, n) draws of one scalar quantity
    m, n = chains.shape
    halves = chains[:, : n // 2 * 2].reshape(2 * m, n // 2)
    W = halves.var(axis=1, ddof=1).mean()          # within-chain variance
    B = (n // 2) * halves.mean(axis=1).var(ddof=1) # between-chain variance
    var_plus = (n // 2 - 1) / (n // 2) * W + B / (n // 2)
    return np.sqrt(var_plus / W)

starts = [(-4, -4), (4, 4), (-4, 4), (4, -4)]
chains = np.stack([gibbs(3000, s)[500:] for s in starts])   # drop warm-up
draws = chains.reshape(-1, 2)
print("sample correlation:", np.corrcoef(draws.T)[0, 1].round(3))
print("sample variances:", draws.var(axis=0).round(3))
lag1 = np.mean([np.corrcoef(c[:-1, 0], c[1:, 0])[0, 1] for c in chains])
print("lag-1 autocorrelation of z1:", lag1.round(3), " (theory rho^2 =", rho**2, ")")
print("split R-hat for z1:", split_rhat(chains[:, :, 0]).round(4))
```

Output: sample correlation 0.849, variances near 0.98, lag-one autocorrelation 0.717 (theory $\rho^2 = 0.7225$) and split-$\hat{R}$ 1.0009. The autocorrelation is the price of axis-aligned moves: when the target is a long diagonal ridge, each coordinate can only move a short distance before hitting the walls. As $\rho \to 1$ the chain slows to a crawl. Reparameterizing (rotating or whitening the variables) or updating correlated variables together as a block fixes this.

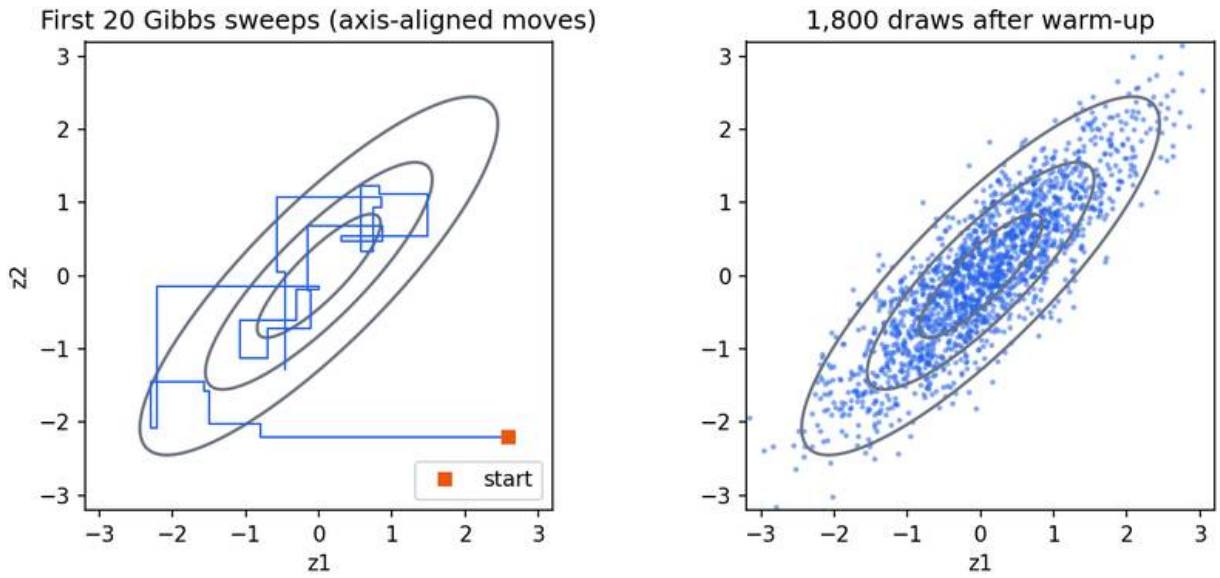


Figure 3 — Gibbs sampling on a Gaussian with correlation 0.85. Each half-step moves parallel to an axis (left), so progress along the diagonal ridge is slow, yet after warm-up the draws fill the target correctly (right). Computed by this tutorial's figure script.

Hamiltonian Monte Carlo

Plain version. Random walks waste effort going back and forth. Hamiltonian Monte Carlo (HMC) uses the *gradient* of the log posterior to glide along the distribution like a frictionless puck on a curved surface, making long, informed moves that are still accepted with high probability.

Precise version. Define potential energy $U(z) = -\log \tilde{p}(z)$, add an auxiliary momentum $r \sim \mathcal{N}(0, M)$ with mass matrix M, and form the Hamiltonian

$$H(z, r) = U(z) + \tfrac{1}{2} r^{\top} M^{-1} r.$$

Hamiltonian dynamics conserve H and preserve volume. In practice we simulate them with L **leapfrog** steps of size ϵ:

$$r \leftarrow r - \tfrac{\epsilon}{2} \nabla U(z), \qquad z \leftarrow z + \epsilon M^{-1} r, \qquad r \leftarrow r - \tfrac{\epsilon}{2} \nabla U(z).$$

The leapfrog integrator is reversible and volume-preserving but not exactly energy-conserving, so the endpoint (z', r') is accepted with probability $\min(1, \exp(H(z, r) - H(z', r')))$, a Metropolis step on the joint space [6]. Because the trajectory follows the geometry of the target, HMC can traverse a correlated ridge that would trap Gibbs or a random walk. Its tuning parameters, ϵ and L, are delicate; the **No-U-Turn Sampler** (NUTS) chooses the trajectory length automatically by stopping when the path starts to double back, and adapts ϵ during warm-up [7]. Betancourt's conceptual introduction explains why HMC scales to high dimensions in terms of the **typical set**, the thin shell where posterior mass actually lives [8]. HMC needs differentiable, continuous targets; discrete latent variables must be summed out or handled by other moves.

A close relative is the **Langevin** update

$$z \leftarrow z + \tfrac{\epsilon}{2}\nabla \log \tilde{p}(z) + \sqrt{\epsilon}\,\xi, \qquad \xi \sim \mathcal{N}(0, I),$$

which is one leapfrog step with freshly drawn momentum. Without an accept/reject step it is only approximately correct for finite ϵ; with the Metropolis correction it becomes the Metropolis-adjusted Langevin algorithm. Keep this update in mind: it reappears in diffusion models.

Diagnostics: is the chain telling the truth?

No finite run *proves* convergence. Diagnostics can only reveal problems. Standard practice:

- **Run several chains from dispersed starting points** and discard an initial warm-up portion of each.
- **Compare chains with $\hat{R}$.** With m chains (split in halves) of length n, let W be the average within-chain variance and B/n the variance of the chain means. Then

$$\hat{R} = \sqrt{\frac{\frac{n-1}{n}W + \frac{1}{n}B}{W}}.$$

If all chains explore the same distribution, $\hat{R} \approx 1$; values noticeably above 1 mean the chains disagree. Vehtari et al. showed that the classic version misses heavy tails and differences in scale, and proposed a rank-normalized, folded split-$\hat{R}$ together with bulk and tail effective sample sizes; this is the version current software reports [9].

- **Report effective sample size.** For a stationary chain with lag-ℓ autocorrelation ρ_ℓ of the quantity of interest,

$$N_{\text{eff}} \approx \frac{N}{1 + 2\sum_{\ell \geq 1} \rho_\ell},$$

where the sum is truncated once estimated autocorrelations become noisy. The Monte Carlo standard error is roughly $\text{sd}/\sqrt{N_{\text{eff}}}$, not $\text{sd}/\sqrt{N}$. For the Gibbs chain above, $\rho_\ell = \rho^{2\ell}$, so $N_{\text{eff}}/N = (1-\rho^2)/(1+\rho^2) \approx 0.16$: 10,000 stored draws are worth about 1,600 independent ones.

- **Look at traces** and, for gradient-based samplers, at divergence warnings, which flag regions of high curvature the integrator cannot follow.

Different quantities mix at different speeds: a mean can look fine while a tail quantile has not converged. And a perfectly mixing chain still only describes the posterior of the *model you wrote down*; it says nothing about whether that model is right.

Variational inference

Plain version. Pick a family of simple distributions (for example, independent Gaussians), and search within it for the member closest to the posterior. Integration becomes optimization, which is fast and scales to huge datasets, at the cost of a bias you cannot remove by running longer.

Precise version. Choose a family $\mathcal{Q}$ of tractable densities $q(z)$, indexed by **variational parameters** ϕ. Variational inference (VI) [10] solves

$$q^* = \arg\min_{q \in \mathcal{Q}} D_{\mathrm{KL}}(q(z) \,\|\, p(z \mid D)), \qquad D_{\mathrm{KL}}(q\|p) = \mathbb{E}_q\left[\log \frac{q(z)}{p(z)}\right].$$

We cannot evaluate this KL directly because it contains the unknown $p(D)$. The trick is the ELBO.

The ELBO, derived two ways

Derivation 1: decomposition. Substitute $\log p(z \mid D) = \log p(D, z) - \log p(D)$ into the KL:

$$D_{\mathrm{KL}}(q\|p(\cdot \mid D)) = \mathbb{E}_q[\log q(z)] - \mathbb{E}_q[\log p(D, z)] + \log p(D).$$

Rearranging,

$$\log p(D) = \mathcal{L}(q) + D_{\mathrm{KL}}(q(z) \,\|\, p(z \mid D)), \qquad \mathcal{L}(q) = \mathbb{E}_q[\log p(D, z)] - \mathbb{E}_q[\log q(z)].$$

Because $\log p(D)$ does not depend on q, maximizing $\mathcal{L}$ is *exactly* minimizing the KL. Because KL is nonnegative, $\mathcal{L}(q) \le \log p(D)$: it is the **evidence lower bound**, ELBO.

Derivation 2: Jensen's inequality. Multiply and divide by q inside the evidence integral and use the concavity of log:

$$\log p(D) = \log \mathbb{E}_q\left[\frac{p(D, z)}{q(z)}\right] \ge \mathbb{E}_q\left[\log \frac{p(D, z)}{q(z)}\right] = \mathcal{L}(q).$$

The first equality assumes q covers the support of $p(D, z)$. If it omits positive joint mass, the expectation integrates only over the support of q; it is no longer equal to the full evidence. The ELBO remains a lower bound via derivation 1 whenever the terms are well-defined. Under the coverage assumption, the gap in Jensen's inequality is exactly that KL. A useful reading of the ELBO is $\mathcal{L} = \mathbb{E}_q[\log p(D \mid z)] - D_{\mathrm{KL}}(q(z)\|p(z))$: fit the data, but do not stray too far from the prior. EM is the special case where q over latent variables is set to the exact posterior in the E step (making the bound tight) and the bound is then maximized over parameters in the M step [1].

Mean field and coordinate ascent (CAVI)

The most common family is **mean field**: $q(z) = \prod_j q_j(z_j)$, independent factors over blocks of z. Holding all factors except q_j fixed and maximizing the ELBO over q_j gives

$$\log q_j^*(z_j) = \mathbb{E}_{q_{-j}}[\log p(D, z)] + \text{const},$$

where $\mathbb{E}_{q_{-j}}$ averages over all other factors and the constant normalizes q_j^*. (To derive it, write $\mathcal{L}$ as a function of q_j only; it equals $-D_{\mathrm{KL}}(q_j \| \tilde{q}_j)$ plus a constant, where $\log \tilde{q}_j = \mathbb{E}_{q_{-j}}[\log p(D, z)]$.) Cycling through j is **coordinate ascent variational inference** (CAVI). Each update cannot decrease the ELBO, so CAVI converges, but only to a local optimum [11]. For conditionally conjugate models each q_j^* stays in the same family as the prior and the updates are closed-form.

Worked example: a Gaussian with unknown mean and precision. Observations $x_n \sim \mathcal{N}(\mu, \tau^{-1})$, prior $\mu \mid \tau \sim \mathcal{N}(\mu_0, (\lambda_0 \tau)^{-1})$, $\tau \sim \mathrm{Gamma}(a_0, b_0)$. The exact posterior is known (Normal–Gamma), so we can grade the approximation. With $q(\mu, \tau) = q(\mu) q(\tau)$ the mean-field update rule gives $q(\mu) = \mathcal{N}(\mu_N, \lambda_N^{-1})$ and $q(\tau) = \mathrm{Gamma}(a_N, b_N)$ with [1]

$$\mu_N = \frac{\lambda_0 \mu_0 + N \bar{x}}{\lambda_0 + N}, \qquad \lambda_N = (\lambda_0 + N)\, \mathbb{E}[\tau],$$
$$a_N = a_0 + \frac{N+1}{2}, \qquad b_N = b_0 + \tfrac{1}{2}\, \mathbb{E}_{\mu}\Big[\textstyle\sum_n (x_n - \mu)^2 + \lambda_0 (\mu - \mu_0)^2 \Big],$$

where $\bar{x}$ is the sample mean and $\mathbb{E}[\tau] = a_N / b_N$. The two factors depend on each other through $\mathbb{E}[\tau]$ and $\mathbb{E}[\mu]$, $\mathbb{E}[\mu^2]$, so we iterate.

```
import numpy as np

# Data x_n ~ N(mu, 1/tau); prior mu | tau ~ N(mu0, 1/(lam0 tau)), tau ~ Gamma(a0, b0).
rng = np.random.default_rng(4)
x = rng.normal(2.0, 0.5, size=30)
N, xbar = len(x), x.mean()
mu0, lam0, a0, b0 = 0.0, 1.0, 1.0, 1.0

# Mean field q(mu, tau) = q(mu) q(tau) with q(mu) = N(mu_N, 1/lam_N), q(tau) = Gamma(a_N,
 ↪  b_N).
E_tau = a0 / b0
for it in range(50):
    mu_N = (lam0 * mu0 + N * xbar) / (lam0 + N)
    lam_N = (lam0 + N) * E_tau
    a_N = a0 + (N + 1) / 2
    E_mu, E_mu2 = mu_N, mu_N**2 + 1 / lam_N
    b_N = b0 + 0.5 * (np.sum(x**2) - 2 * E_mu * np.sum(x) + N * E_mu2
                      + lam0 * (E_mu2 - 2 * mu0 * E_mu + mu0**2))
    E_tau = a_N / b_N

# Exact posterior is Normal-Gamma: compare the marginal moments of mu.
lam_post = lam0 + N
mu_post = (lam0 * mu0 + N * xbar) / lam_post
a_post = a0 + N / 2
b_post = b0 + 0.5 * (np.sum((x - xbar) ** 2) + lam0 * N * (xbar - mu0) ** 2 / lam_post)
var_mu_exact = b_post / ((a_post - 1) * lam_post)       # Student-t marginal variance
print("E[mu]   VI %.4f  exact %.4f" % (mu_N, mu_post))
print("Var[mu] VI %.5f  exact %.5f" % (1 / lam_N, var_mu_exact))
print("E[tau]  VI %.3f   exact %.3f" % (a_N / b_N, a_post / b_post))
```

The means agree (1.8668 both) and so does $\mathbb{E}[\tau]$ (2.146), but the variance of μ is 0.01503 under VI against 0.01603 exactly: about 6% too small. Under the exact posterior, μ and τ are dependent (the

spread of μ scales with $1/\tau$) and its marginal is a heavier-tailed Student-t; a factorized Gaussian cannot represent either property.

This underestimation is systematic. For a Gaussian target with precision matrix Λ, the optimal mean-field factor q_j has variance $1/\Lambda_{jj}$, the *conditional* variance, which is never larger than the marginal variance. Figure 4 shows the extreme case.

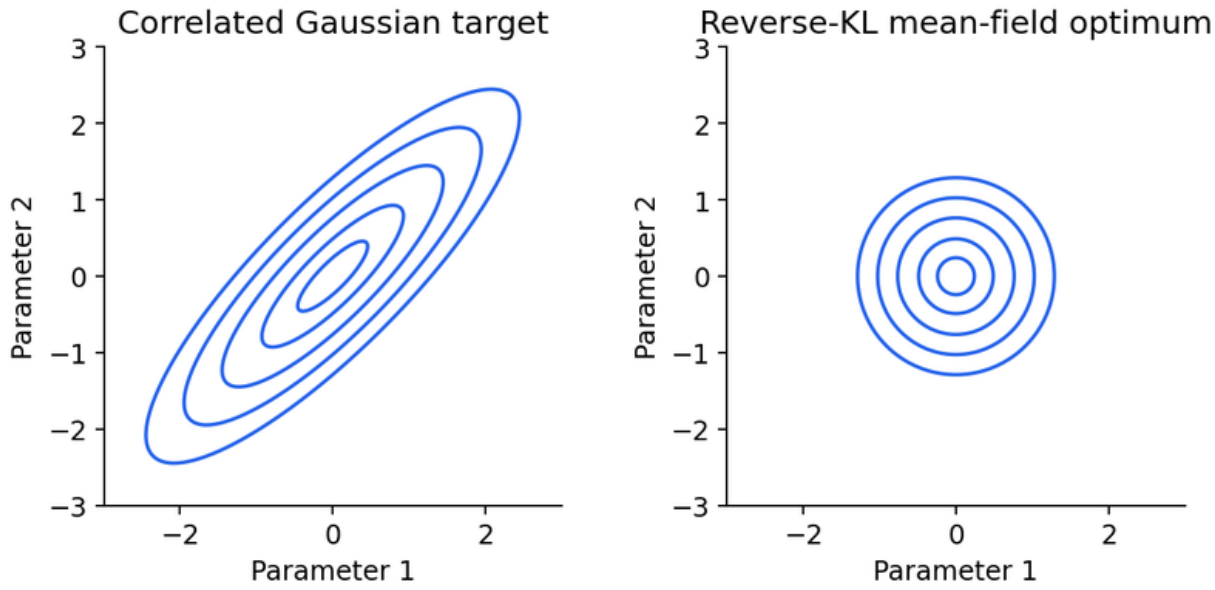


Figure 4 — Original analytical illustration with unit marginal target variances and correlation 0.85. The optimal Gaussian mean-field factors under reverse KL have variance $1 - 0.85^2 \approx 0.28$ each, illustrating lost correlation and underestimated marginal uncertainty. Contours show relative density, not measured model performance.

Which KL? Mode seeking versus mass covering

KL is not symmetric, and the direction changes the answer. **Reverse KL**, $D_{\mathrm{KL}}(q\|p)$, used by standard VI, takes expectations under q: it heavily penalizes putting q-mass where p is tiny, but does not mind if q ignores some of p. The result is **mode seeking** and **under-dispersed**. **Forward KL**, $D_{\mathrm{KL}}(p\|q)$, takes expectations under p: it heavily penalizes missing any region where p has mass, so q stretches to cover everything, which is **mass covering** and can put mass in the valleys between modes. Within an exponential family such as the Gaussians, minimizing forward KL is exactly **moment matching**: q takes the mean and covariance of p.

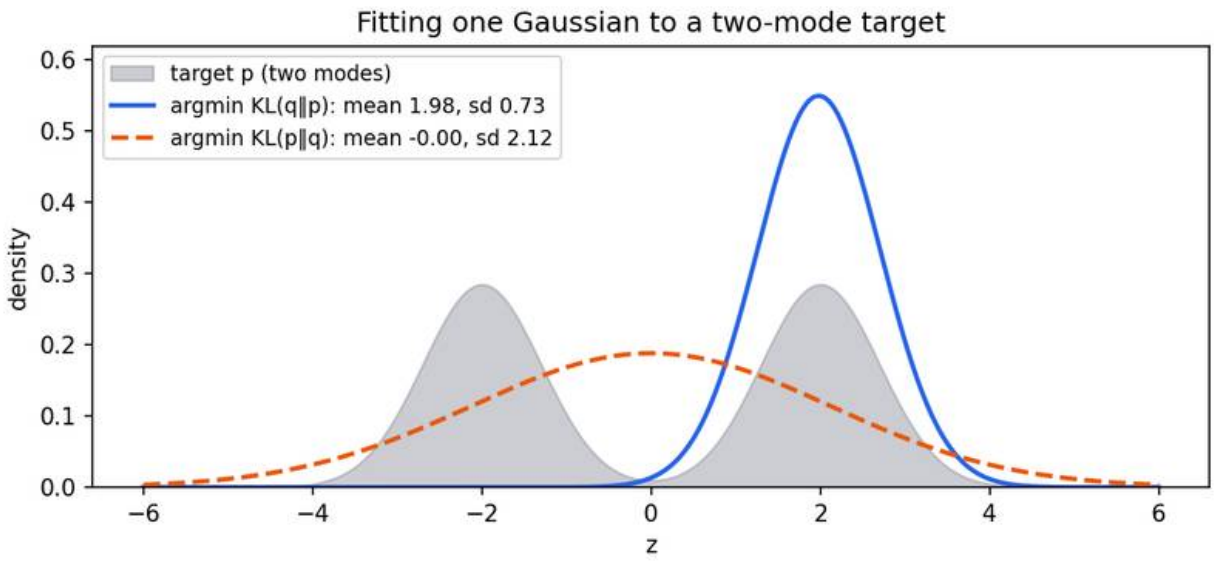


Figure 5 — One Gaussian fitted to an equal mixture of $N(-2, 0.7^2)$ and $N(2, 0.7^2)$. Minimizing reverse $KL(q\|p)$ locks onto one mode (blue; which mode depends on initialization). Minimizing forward $KL(p\|q)$ matches the mixture's mean and variance and covers both modes, placing its peak where the target has almost no mass (orange, dashed). Computed numerically by this tutorial's figure script.

Neither direction is "right". Reverse KL is tractable and gives sharp, usable answers but understates uncertainty. Forward KL is the objective behind expectation propagation and behind maximum-likelihood fitting of a model to data, and it is safer when missing a mode is costly.

Stochastic and black-box VI

CAVI needs conjugacy and passes over the whole dataset per update. Modern VI instead takes **stochastic gradients** of the ELBO, which works for almost any model and scales to large data by subsampling minibatches [11, 12]. There are two main gradient estimators for $\nabla_\phi \mathcal{L}(\phi)$:

- **Score-function (REINFORCE) estimator**, used by black-box VI [13]:

$$\nabla_\phi \mathcal{L} = \mathbb{E}_{q_\phi}\Big[\nabla_\phi \log q_\phi(z)\,\big(\log p(D, z) - \log q_\phi(z)\big)\Big].$$

It needs only evaluations of $\log p(D, z)$ and works for discrete z, but has high variance and needs control variates.

- **Reparameterization (pathwise) estimator.** Write $z = g_\phi(\varepsilon)$ with noise ε from a fixed distribution, for example $z = m + \sigma \odot \varepsilon$ with $\varepsilon \sim \mathcal{N}(0, I)$. Then

$$\nabla_\phi \mathcal{L} = \mathbb{E}_{\varepsilon}\Big[\nabla_\phi\big(\log p(D, g_\phi(\varepsilon)) - \log q_\phi(g_\phi(\varepsilon))\big)\Big].$$

It requires differentiable $\log p$ and continuous z but typically has much lower variance. This is the estimator the VAE made famous [15].

For a diagonal Gaussian with $\phi = (m, s)$, $\sigma = e^s$, the entropy term is $\sum_j s_j$ plus a constant, so its gradient with respect to s is simply 1. Applied to the logistic-regression posterior:

```
import numpy as np

rng = np.random.default_rng(0)
n = 40
X = np.c_[np.ones(n), rng.normal(size=n)]
y = (rng.random(n) < 1 / (1 + np.exp(-X @ np.array([-0.5, 1.5])))).astype(float)
prior_var = 4.0

def grad_log_joint(W):                  # W: (S, 2) -> gradients (S, 2)
    p = 1 / (1 + np.exp(-W @ X.T))      # (S, n)
    return (y - p) @ X - W / prior_var

# Laplace: Newton's method to the MAP, then invert the Hessian.
w = np.zeros(2)
for _ in range(25):
    p = 1 / (1 + np.exp(-X @ w))
    H = X.T @ (X * (p * (1 - p))[:, None]) + np.eye(2) / prior_var
    w = w + np.linalg.solve(H, grad_log_joint(w[None])[0])
print("Laplace mean:", w.round(3), " sd:", np.sqrt(np.diag(np.linalg.inv(H))).round(3))

# Mean-field Gaussian VI with reparameterization: w = m + exp(s) * eps.
m, s = np.zeros(2), np.full(2, -1.0)
lr, S = 0.02, 64
for step in range(3000):
    eps = rng.normal(size=(S, 2))
    W = m + np.exp(s) * eps
    g = grad_log_joint(W)
    grad_m = g.mean(axis=0)                                 # d ELBO / d m
    grad_s = (g * eps).mean(axis=0) * np.exp(s) + 1.0       # + entropy term
    m += lr * grad_m
    s += lr * grad_s
print("VI mean:     ", m.round(3), " sd:", np.exp(s).round(3))
```

Compare with the grid truth, mean $(-0.303, 1.631)$ and sd $(0.366, 0.578)$. The code prints a Laplace mean of $(-0.291, 1.500)$ with sd $(0.355, 0.556)$, and a VI mean of $(-0.303, 1.630)$ with sd $(0.358, 0.562)$. VI recovers the posterior mean almost exactly, because it averages over the skewed posterior; Laplace reports the *mode*, which differs from the mean when the posterior is skewed. Both slightly understate the spread. With 40 data points the posterior is nearly Gaussian, so the differences are small; with fewer points or separable classes they grow quickly.

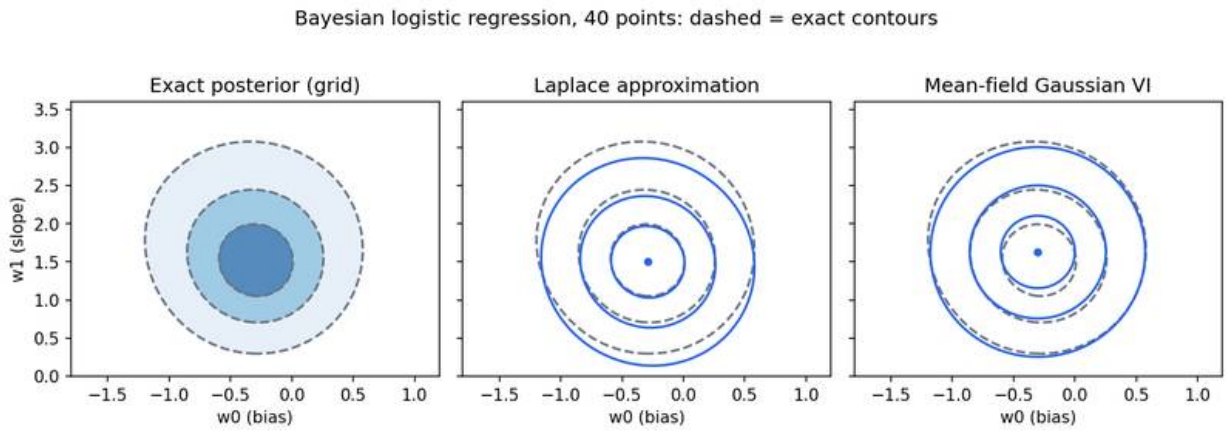


Figure 6 — Two-weight Bayesian logistic regression on 40 synthetic points. Dashed gray: exact posterior contours from the grid. Blue: Laplace (centered at the MAP, shifted toward smaller slope) and mean-field Gaussian VI (centered near the true posterior mean, slightly too narrow). Computed by this tutorial's figure script.

Laplace approximation and expectation propagation

Laplace: a Gaussian at the peak

Plain version. Find the highest point of the posterior and fit a Gaussian whose curvature matches the curvature there.

Precise version. Expand $\log \tilde{p}(z)$ to second order around a mode $\hat{z}$, where the gradient is zero. With $H = -\nabla^2 \log \tilde{p}(\hat{z})$ the Hessian of the negative log posterior,

$$q(z) = \mathcal{N}(z \mid \hat{z}, H^{-1}), \qquad \log p(D) \approx \log \tilde{p}(\hat{z}) + \frac{d}{2} \log 2\pi - \frac{1}{2} \log |H|,$$

where d is the dimension of z. The second formula is the **Laplace evidence approximation**; keeping only the terms that grow with N gives the BIC criterion used for choosing the number of mixture components in PR 07 [1]. Laplace is cheap (one optimization and one Hessian) and becomes accurate as data accumulate, because posteriors of regular models become increasingly Gaussian. It fails for skewed posteriors, for modes on a boundary (such as a variance near zero), when the Hessian is singular, and for multi-modal posteriors, where it describes only one mode. Modern "Laplace for deep networks" methods apply it to the last layer or with structured Hessian approximations to obtain cheap uncertainty for trained neural networks.

Expectation propagation

Expectation propagation (EP) [14] targets posteriors that factor as $p(z \mid D) \propto \prod_i f_i(z)$, for example one factor per data point. It approximates each factor by a simple **site** $\tilde{f}_i(z)$ from an exponential family, so $q(z) \propto \prod_i \tilde{f}_i(z)$, and refines sites one at a time:

1. Remove site i to form the **cavity** $q^{\setminus i}(z) \propto q(z)/\tilde{f}_i(z)$.
2. Form the **tilted** distribution $\hat{p}_i(z) \propto f_i(z)\, q^{\setminus i}(z)$, the cavity times the *exact* factor.
3. Set the new q to the member of the family that matches the moments of $\hat{p}_i$, i.e. minimizes the *forward* KL $D_{\mathrm{KL}}(\hat{p}_i \| q)$.
4. Update the site: $\tilde{f}_i \propto q^{\text{new}}/q^{\setminus i}$.

Because each step uses forward KL locally, EP tends to cover mass and often gives better-calibrated variances than mean-field VI, for example in Gaussian-process classification. In exchange, it has no global objective that is guaranteed to improve, can fail to converge, and may produce invalid (negative-variance) sites that need damping [1, 14].

Evidence and complexity: Bayesian Occam's razor

Plain version. A model that can explain *anything* spreads its prior bets thin, so it gets little credit for explaining the data you actually saw. A simpler model that bet on this kind of data gets more credit.

Precise version. The evidence of model $\mathcal{M}$ integrates the likelihood against the prior,

$$p(D \mid \mathcal{M}) = \int p(D \mid \theta, \mathcal{M})\, p(\theta \mid \mathcal{M})\, d\theta.$$

A flexible model can reach a high peak likelihood, yet if most of its prior mass sits on parameters that fit badly, the integral is small. The Laplace formula makes this explicit: the evidence is the peak value times an "Occam factor" that compares posterior volume with prior volume. Two cautions follow. First, the evidence depends on the prior, even on aspects that barely affect the posterior, such as how wide a vague prior is. Second, it answers "which model would have predicted this data set?", which is related to but not identical to "which model will predict *new* data best?". Held-out validation (PR 09) answers the second. Comparing ELBOs across models is also delicate: each ELBO undershoots its evidence by a different, unknown KL gap.

From the ELBO to VAEs and diffusion models

The machinery in this chapter is the backbone of two families of deep generative models covered in the ML track.

VAEs are amortized variational inference. A variational autoencoder models images x with a latent code z, a prior $p(z)$, and a neural decoder $p_\theta(x \mid z)$. Instead of optimizing separate variational parameters for every image, an encoder network outputs them: $q_\phi(z \mid x) = \mathcal{N}(\mu_\phi(x), \mathrm{diag}\, \sigma^2_\phi(x))$. This is **amortized inference**: one network learns to perform inference for any input. The per-image objective is exactly the ELBO,

$$\mathcal{L}(\theta, \phi; x) = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x \mid z)] - D_{\mathrm{KL}}(q_\phi(z \mid x) \,\|\, p(z)),$$

trained jointly over θ and ϕ with the reparameterization gradient [15, 16]. Its known weaknesses are the ones above: an **amortization gap** (the encoder's answer is worse than per-image optimiza-

tion), the under-dispersion of reverse KL, and a Gaussian decoder that blurs. See ML 07 for a full treatment and code.

Diffusion models are trained with an ELBO and sampled with Langevin-style dynamics. A denoising diffusion probabilistic model fixes a forward process that gradually adds Gaussian noise, $x_0 \to x_1 \to \ldots \to x_T$, and learns a reverse process $p_\theta(x_{t-1} \mid x_t)$. The forward noisy states play the role of latent variables, with the forward process as a *fixed* variational posterior; the training objective is a variational bound on $-\log p_\theta(x_0)$, and the popular noise-prediction loss is a reweighted form of that bound [17]. The score-based view [18] instead learns $s_\theta(x, t) \approx \nabla_x \log p_t(x)$, the gradient of the log density of noisy data, and generates samples by integrating a reverse-time stochastic differential equation; its predictor–corrector samplers explicitly interleave Langevin MCMC steps that use s_θ in place of $\nabla \log \tilde{p}$. The only change from the Langevin update earlier in this chapter is that the score is *learned from data* rather than computed from a formula. See ML 08.

Choosing a method

Method	What it returns	Strengths	Main failure mode
Grid / quadrature	numerical approximation	low-dimensional reference after refinement checks	truncation/discretization error; cost grows rapidly with dimension
Importance sampling	weighted samples, evidence	simple, parallel, gives $p(D)$	weight collapse when q misses p
Random-walk MH	correlated samples	needs only $\tilde{p}$	slow in high dimension; tuning
Gibbs	correlated samples	no tuning when conditionals are easy	slow on correlated variables
HMC / NUTS	correlated samples	scales to many continuous parameters	needs gradients; discrete variables; divergences
Mean-field VI	a distribution q	fast, scalable, minibatch-friendly	under-dispersion; local optima
Laplace	Gaussian at the mode	cheap; evidence estimate	skew, boundaries, multiple modes
EP	Gaussian / exponential family	often good variances	may not converge

A sensible workflow is to start with an optimizer (MAP or VI) for speed, check it against a careful sampler on a smaller version of the problem, and use simulation from known parameters to verify that the whole pipeline recovers what it should.

Modern view

Surveys worth reading. Blei, Kucukelbir and McAuliffe's review for statisticians [11] is the standard entry point to VI: it derives the ELBO and CAVI, works through Gaussian mixtures, introduces stochastic VI, and is explicit that VI trades MCMC's asymptotic exactness for speed, tending to underestimate posterior variance. Zhang, Bütepage, Kjellström and Mandt [12] organize

the later literature into four axes: scalable VI (stochastic and distributed), generic VI for non-conjugate models (black-box and reparameterization gradients), accurate VI (alternative divergences beyond reverse KL, richer families) and amortized VI (inference networks, VAEs). Jordan, Ghahramani, Jaakkola and Saul [10] is the classic introduction to variational methods for graphical models and remains the clearest source for the convex-duality view. On the sampling side, Neal's chapter [6] is the canonical derivation of HMC, Betancourt [8] gives the geometric picture of why it works in high dimensions, and Hoffman and Gelman [7] describe NUTS, the default sampler in modern probabilistic programming systems. Vehtari et al. [9] set the current standard for convergence diagnostics. Kingma and Welling's monograph [16] surveys VAEs as amortized VI, and Yang et al. [19] survey diffusion models, including their variational and score-based formulations.

State of the art, 2023–2026. Three trends stand out.

1. *Machine learning inside Bayesian computation.* Winter, Campbell, Lin, Srivastava and Dunson [20] review how normalizing flows, Bayesian coresets, distributed inference and variational methods are being used to speed up posterior computation, and where the guarantees are still weaker than for classical MCMC.
2. *Diffusion ideas used for sampling, not just generation.* When only an unnormalized density is available (no data), one can still train a diffusion-style sampler. Denoising Diffusion Samplers [21] learn to reverse a noising process so that it ends at a target known only up to a constant, giving approximate samples and an estimate of the normalizing constant; they connect to optimal control and Schrödinger bridges and are a representative member of a growing family of learned samplers for Bayesian and physics problems.
3. *Workflows over single algorithms.* Practice has moved toward probabilistic programming systems that run NUTS by default, report rank-normalized $\hat{R}$ and bulk/tail ESS [9], and use VI or Laplace as fast first passes rather than final answers.

Open problems. Reliable uncertainty for very large models remains hard: full-batch MCMC is too expensive, and cheap approximations (mean-field VI, last-layer Laplace, ensembles) disagree. Multi-modal posteriors defeat both local VI and single chains. Evaluating the quality of an approximation without ground truth is still largely heuristic, and comparing models through ELBOs is confounded by unknown approximation gaps.

Key takeaways

- Posteriors are easy to evaluate up to a constant and hard to integrate; grids cost G^d, so approximation is unavoidable beyond a few dimensions.
- Monte Carlo error falls as $1/\sqrt{S}$ regardless of dimension, but only if you can draw from the right distribution. Importance sampling corrects for the wrong one, and the weight ESS tells you when the correction has collapsed.
- Metropolis–Hastings needs only ratios of $\tilde{p}$ and a Hastings correction for asymmetric proposals. Gibbs is MH with acceptance 1. HMC uses gradients to move far. Always record rejected states, run several chains, and check $\hat{R}$ and ESS.

- VI maximizes the ELBO, which equals minimizing reverse KL to the posterior. It is fast and scalable, mode seeking, and tends to underestimate variance, especially under mean field.
- Laplace fits a Gaussian at the mode and gives a cheap evidence estimate (and BIC); EP matches moments locally and often gives better variances but may not converge.
- The evidence implements Occam's razor but depends on priors and answers a different question from held-out prediction.
- VAEs are amortized VI with the reparameterization gradient; diffusion models are trained with a variational bound and sampled with Langevin-like dynamics driven by a learned score.

Exercises

1. **ELBO by hand.** Starting from $D_{\mathrm{KL}}(q(z) \| p(z \mid D))$, derive $\log p(D) = \mathcal{L}(q) + D_{\mathrm{KL}}$. Then show that when $q(z) = p(z \mid D)$ the bound is tight, and explain how this makes EM a special case of maximizing the ELBO.

   Hint

   Substitute $\log p(z \mid D) = \log p(D, z) - \log p(D)$ and pull $\log p(D)$ out of the expectation because it does not depend on z. With q equal to the posterior, the KL is zero. EM's E step sets q to the posterior of the latent variables at the current parameters (tight bound); its M step maximizes $\mathcal{L}$ over the parameters with q fixed.

2. **The Hastings correction.** You propose from a log-normal random walk on a positive parameter, $z' = z\,e^{\eta}$ with $\eta \sim \mathcal{N}(0, s^2)$. Compute $q(z' \mid z)$ and the ratio $q(z \mid z')/q(z' \mid z)$. What goes wrong if you use the plain Metropolis rule?

   Hint

   $q(z' \mid z) = \frac{1}{z'\, s\sqrt{2\pi}} \exp(-(\log z' - \log z)^2/(2s^2))$. The exponential parts are symmetric, so the ratio is z'/z. Omitting it makes the chain sample from a density proportional to $\tilde{p}(z)/z$, biased toward small values.

3. **Step size and mixing.** In the Metropolis code, change the proposal standard deviation to 0.02 and then to 8. For each, report acceptance, lag-one autocorrelation and an estimate of N_{eff} using the formula in the diagnostics section (truncate the sum at the first negative autocorrelation). Which step size gives the most effective samples per iteration?

   Hint

   Compare with Figure 2. Very small steps give acceptance near 1 but autocorrelations near 1, so N_{eff} is tiny and the chain may not even reach the target. Very large steps give low acceptance and long runs of repeated values. An intermediate value, around the target's standard deviation, gives far more effective samples.

4. **Gibbs on a ridge.** Re-run the Gibbs sampler with $\rho = 0.99$. Predict the lag-one autocorrelation of z_1 and N_{eff}/N, then check by simulation. Propose and implement a fix.

   Hint

   The z_1 chain is autoregressive with coefficient $\rho^2 \approx 0.98$, so $N_{\text{eff}}/N = (1-\rho^2)/(1+\rho^2) \approx 0.01$. Fixes: sample (z_1, z_2) jointly as one block, or rotate to $u = (z_1 + z_2)/\sqrt{2}$, $v = (z_1 - z_2)/\sqrt{2}$, which are independent under the target.

5. **Mean field underestimates variance.** For a Gaussian target $\mathcal{N}(0, \Sigma)$ with precision $\Lambda = \Sigma^{-1}$, use the mean-field update rule to show that the optimal q_j is Gaussian with variance $1/\Lambda_{jj}$. Prove $1/\Lambda_{jj} \le \Sigma_{jj}$ and evaluate both for unit variances and correlation 0.85.

   Hint

   $\log p = -\frac{1}{2} z^\top \Lambda z + \text{const}$; take the expectation over q_{-j} and keep terms in z_j: $-\frac{1}{2}\Lambda_{jj} z_j^2 - z_j \sum_{k \ne j} \Lambda_{jk} \mathbb{E}[z_k]$, a Gaussian with precision Λ_{jj}. $1/\Lambda_{jj}$ is the conditional variance of z_j given the rest, which by the Schur complement is Σ_{jj} minus a nonnegative term. For $\rho = 0.85$: $1 - \rho^2 \approx 0.28$ versus 1.

6. **Importance sampling the evidence.** For the two-weight logistic regression, estimate $\log p(D)$ by importance sampling with (a) the prior as proposal and (b) the Laplace Gaussian (with its covariance inflated by 1.5) as proposal. Compare both with the grid value -22.117 and with the weight ESS. Which proposal would you trust in 50 dimensions, and why?

   Hint

   The estimate is $\log \frac{1}{S} \sum_s \tilde{p}(z_s)/q(z_s)$, computed with log-sum-exp. The prior is much wider than the posterior, so most draws get negligible weight and the ESS is a small fraction of S; the Laplace-based proposal sits on the posterior and gives a much larger ESS. In high dimensions the mismatch between prior and posterior grows exponentially, so the prior proposal becomes useless while a well-fitted, slightly heavy-tailed proposal can still work.

References

1. C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006, Sections 4.4 (Laplace), 10.1–10.2 and 10.7 (variational inference, EP) and 11.1–11.5 (sampling). Official book page and PDF
2. S. Theodoridis and K. Koutroumbas, *Pattern Recognition*, 4th ed., Academic Press, 2009, Chapter 2 (Bayes decision theory and parameter estimation) and Appendix C (cost function optimization). These are prerequisites rather than identical coverage of this chapter. doi
3. N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller and E. Teller, "Equation of state calculations by fast computing machines," *The Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087–1092, 1953. doi

4. W. K. Hastings, “Monte Carlo sampling methods using Markov chains and their applications,” *Biometrika*, vol. 57, no. 1, pp. 97–109, 1970. doi
5. S. Geman and D. Geman, “Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-6, no. 6, pp. 721–741, 1984. doi
6. R. M. Neal, “MCMC using Hamiltonian dynamics,” in S. Brooks, A. Gelman, G. Jones and X.-L. Meng (eds.), *Handbook of Markov Chain Monte Carlo*, Chapman and Hall/CRC, 2011 (arXiv: 1206.1901). arXiv
7. M. D. Hoffman and A. Gelman, “The No-U-Turn Sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo,” *Journal of Machine Learning Research*, vol. 15, pp. 1593–1623, 2014. JMLR
8. M. Betancourt, “A conceptual introduction to Hamiltonian Monte Carlo,” arXiv:1701.02434, 2017. arXiv
9. A. Vehtari, A. Gelman, D. Simpson, B. Carpenter and P.-C. Bürkner, “Rank-normalization, folding, and localization: An improved $\hat{R}$ for assessing convergence of MCMC (with discussion),” *Bayesian Analysis*, vol. 16, no. 2, 2021. doi
10. M. I. Jordan, Z. Ghahramani, T. S. Jaakkola and L. K. Saul, “An introduction to variational methods for graphical models,” *Machine Learning*, vol. 37, no. 2, pp. 183–233, 1999. doi
11. D. M. Blei, A. Kucukelbir and J. D. McAuliffe, “Variational inference: A review for statisticians,” *Journal of the American Statistical Association*, vol. 112, no. 518, pp. 859–877, 2017. doi
12. C. Zhang, J. Bütepage, H. Kjellström and S. Mandt, “Advances in variational inference,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 41, no. 8, pp. 2008–2026, 2019. doi
13. R. Ranganath, S. Gerrish and D. M. Blei, “Black box variational inference,” in *Proc. International Conference on Artificial Intelligence and Statistics* (*AISTATS*), PMLR vol. 33, 2014. PMLR
14. T. P. Minka, “Expectation propagation for approximate Bayesian inference,” in *Proc. Conference on Uncertainty in Artificial Intelligence* (*UAI*), 2001 (arXiv:1301.2294). arXiv
15. D. P. Kingma and M. Welling, “Auto-encoding variational Bayes,” *International Conference on Learning Representations* (*ICLR*), 2014; arXiv:1312.6114. arXiv
16. D. P. Kingma and M. Welling, “An introduction to variational autoencoders,” *Foundations and Trends in Machine Learning*, vol. 12, no. 4, pp. 307–392, 2019. arXiv
17. J. Ho, A. Jain and P. Abbeel, “Denoising diffusion probabilistic models,” in *Advances in Neural Information Processing Systems* (*NeurIPS*), 2020 (arXiv:2006.11239). arXiv
18. Y. Song, J. Sohl-Dickstein, D. P. Kingma, A. Kumar, S. Ermon and B. Poole, “Score-based generative modeling through stochastic differential equations,” in *Proc. International Conference on Learning Representations* (*ICLR*), 2021. arXiv
19. L. Yang, Z. Zhang, Y. Song, S. Hong, R. Xu, Y. Zhao, W. Zhang, B. Cui and M.-H. Yang, “Diffusion models: A comprehensive survey of methods and applications,” *ACM Computing Surveys*, vol. 56, no. 4, 2023. doi
20. S. Winter, T. Campbell, L. Lin, S. Srivastava and D. B. Dunson, “Machine learning and the future of Bayesian computation,” arXiv:2304.11251, 2023. arXiv
21. F. Vargas, W. Grathwohl and A. Doucet, “Denoising diffusion samplers,” in *Proc. International Conference on Learning Representations* (*ICLR*), 2023 (arXiv:2302.13834). arXiv

PR 09 • Ensembles, Validation and Cluster Validity

Derive the bias–variance decomposition, bagging, random forests, AdaBoost and gradient boosting, then estimate and compare recognizers honestly with cross-validation, the bootstrap, nested and group-aware validation, and paired statistical tests.

Online edition: https://blog.shyandram.dev/en/learn/pattern-recognition/ensembles-validation/

What you'll learn

- Derive the bias–variance decomposition of squared error and use it to explain why averaging helps unstable learners.
- Explain how decision trees partition feature space, and how bagging and random forests reduce their variance, including out-of-bag estimates.
- Derive AdaBoost from exponential loss, connect it to gradient boosting, and distinguish boosting, voting, Bayesian model averaging and stacking.
- Estimate the performance of a whole recognition procedure with hold-out, k-fold, group-aware and nested cross-validation and with the bootstrap (.632 and .632+).
- Compare two or more classifiers with McNemar's test, the 5×2cv and corrected resampled t-tests, and the Friedman/Nemenyi procedure, and know when each is valid.
- Evaluate probabilities as well as decisions, and treat cluster validity as a model-selection question.

The big picture

A single recognizer makes mistakes for two very different reasons. Sometimes it is too simple to express the true decision rule; sometimes it is so flexible that it memorizes accidents of the particular training set. Ensembles attack the second problem (bagging, random forests) or the first (boosting) by combining many models. Validation is the other half of the story: once we can build many models and tune many knobs, we need a trustworthy way to estimate how the *chosen* procedure will perform on new data, and to decide whether one procedure is really better than another.

> **Explain like I'm five**
>
> Imagine guessing how many sweets are in a jar. One friend's guess can be way off. If ten friends guess *independently* and you average, the wild guesses cancel out and the average is usually closer. But if all ten friends just copy the loudest friend, averaging does not help at all. Now suppose you want to know which friend is the best guesser. You must test them on a *new* jar they have never seen; testing on the jar they already studied would flatter everyone.

Why it matters: tree ensembles remain among the strongest methods for tabular features, including hand-crafted image descriptors [23], deep-network ensembles are a standard way to obtain uncertainty estimates [24], and almost every published comparison between recognizers rests on cross-validation and a statistical test. Theodoridis and Koutroumbas treat decision trees, classifier combination and boosting in Chapter 4, error estimation and comparison in Chapter 10 ("Supervised Learning: The Epilogue"), and cluster validity in Chapter 16 [1]. Bishop's Chapter 14 separates several mathematically different ways to combine models, and Section 3.2 derives the bias–variance decomposition [2]. Classifiers themselves were covered in PR 03 and PR 04.

Bias, variance and noise

Plain version. Imagine retraining the same learning algorithm on many different training sets drawn from the same source. The *average* prediction may still miss the truth: that systematic miss is **bias**. The individual predictions scatter around their average: that scatter is **variance**. Even a perfect predictor cannot remove the randomness in the labels themselves: that is **noise**.

Precise version. Let the target be $y = f(x) + \varepsilon$, where f is the unknown regression function and ε is noise with $\mathbb{E}[\varepsilon] = 0$ and $\mathrm{Var}(\varepsilon) = \sigma^2$, independent of everything else. A learning algorithm maps a random training set D to a predictor $\hat{f}_D$. Write $\bar{f}(x) = \mathbb{E}_D[\hat{f}_D(x)]$ for the average prediction over training sets. At a fixed input x, the expected squared error over training sets and test noise is

$$\begin{aligned}\mathbb{E}_{D,\varepsilon}[(y - \hat{f}_D(x))^2] &= \mathbb{E}[(\varepsilon + f(x) - \bar{f}(x) + \bar{f}(x) - \hat{f}_D(x))^2] \\ &= \underbrace{\sigma^2}_{\text{noise}} + \underbrace{(f(x) - \bar{f}(x))^2}_{\text{bias}^2} + \underbrace{\mathbb{E}_D[(\hat{f}_D(x) - \bar{f}(x))^2]}_{\text{variance}}.\end{aligned}$$

The three cross terms vanish: ε has zero mean and is independent of D, $f(x) - \bar{f}(x)$ is a constant, and $\mathbb{E}_D[\bar{f}(x) - \hat{f}_D(x)] = 0$ by the definition of $\bar{f}$. Averaging over the input distribution $p(x)$ gives the expected test error. Geman, Bienenstock and Doursat named the resulting tension the *bias/variance dilemma* [3]: making a model more flexible typically lowers bias and raises variance, and the best complexity balances the two.

The code below estimates each term by brute force for regression trees of growing depth on a noisy sine curve. Two hundred training sets of 60 points each are drawn; the empirical average and spread of the 200 fitted trees give bias² and variance.

```
import numpy as np
from sklearn.tree import DecisionTreeRegressor

rng = np.random.default_rng(0)
f = lambda x: np.sin(2 * np.pi * x)          # true regression function
sigma = 0.3                                   # noise standard deviation
x_test = np.linspace(0, 1, 200)[:, None]

def bias_variance(depth, n_sets=200, n=60):
    preds = np.empty((n_sets, len(x_test)))
    for s in range(n_sets):                   # many independent training sets
```

```
        x = rng.random((n, 1))
        y = f(x[:, 0]) + rng.normal(0, sigma, n)
        tree = DecisionTreeRegressor(max_depth=depth, random_state=0)
        preds[s] = tree.fit(x, y).predict(x_test)
    mean_pred = preds.mean(axis=0)
    bias2 = np.mean((mean_pred - f(x_test[:, 0])) ** 2)
    var = np.mean(preds.var(axis=0))
    return bias2, var

for depth in [1, 2, 3, 5, 8, None]:
    b2, v = bias_variance(depth)
    print(f"depth={str(depth):>4}  bias^2={b2:.4f}  variance={v:.4f}  "
          f"expected test MSE~{b2 + v + sigma**2:.4f}")
```

On our run, depth 1 gives bias² ≈ 0.074 and variance ≈ 0.035, depth 3 gives the lowest total (bias² ≈ 0.003, variance ≈ 0.045), and fully grown trees have almost no bias but variance ≈ 0.092, as large as the noise variance $\sigma^2 = 0.09$ itself. Figure 1 shows the same experiment visually.

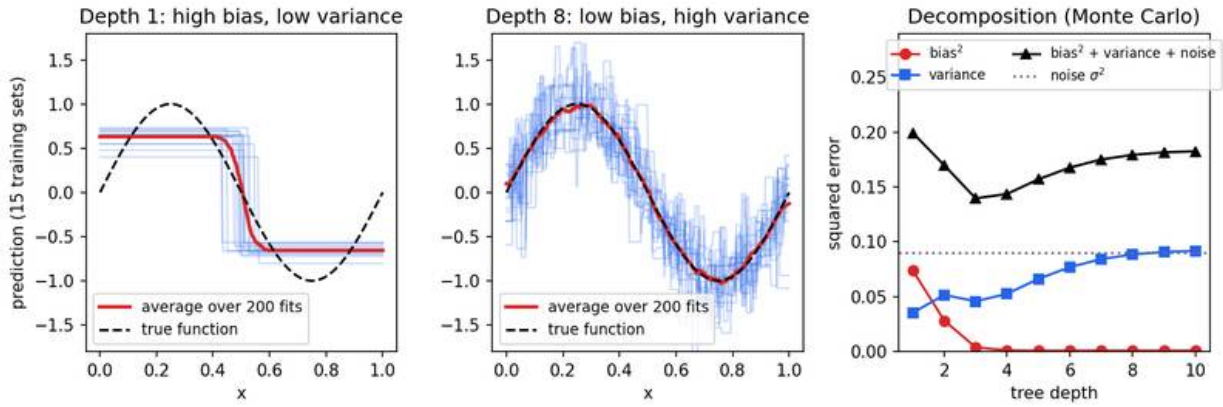


Figure 1 — Bias–variance decomposition for regression trees on $y = \sin(2\pi x) + \text{noise}$ ($\sigma = 0.3$, 60 points per training set, 200 training sets). Shallow trees are consistently wrong (bias); deep trees are right on average but each one is noisy (variance). Generated by this chapter's script.

Three cautions keep the picture honest. First, the clean decomposition is specific to squared error; for 0–1 classification loss there are several competing definitions and bias and variance can interact non-additively, so use the decomposition as intuition rather than as an exact accounting for classifiers. Second, "complexity" is not just parameter count: depth, regularization strength, number of boosting rounds and early stopping all move a model along the bias–variance axis. Third, very large over-parameterized models can show a test-error curve that rises and then falls again as capacity grows past the interpolation point, the "double descent" shape documented by Belkin and colleagues [25]. The decomposition still holds pointwise; what changes is how variance behaves as capacity grows.

Online only: Double descent: reading Belkin, Hsu, Ma and Mandal (PNAS 2019) — https://blog.shyandram.dev/en/learn/pattern-recognition/ensembles-validation/papers/double-descent-reading-belkin-hsu-ma-and-mandal-pnas-2019/

Trees and partitions

Plain version. A decision tree asks a sequence of yes/no questions about the features, such as "is feature 3 below 0.7?", and each final answer box (a leaf) stores a prediction. It is the classical example of a low-bias, high-variance learner, which is exactly why it is the favorite building block for ensembles.

Precise version. A tree recursively divides observations by conditions such as $x_j \le b$. Each leaf estimates a class distribution (classification) or a constant (regression). For a classification node with class proportions p_k, the **Gini impurity** is

$$G = 1 - \sum_k p_k^2,$$

and the **entropy** is $H = -\sum_k p_k \log p_k$. Both are zero for a pure node and largest for a uniform mix. A split of a parent node P into children L and R, holding n_L and n_R of its n_P samples, is scored by the impurity decrease

$$\Delta = G(P) - \frac{n_L}{n_P} G(L) - \frac{n_R}{n_P} G(R),$$

and the learner chooses the feature and threshold with the largest Δ. This is a greedy local choice: the best single split need not lead to the best tree. A deep tree can capture interactions and nonlinear, axis-aligned boundaries, but it can change substantially when a few training points change; that instability is exactly the high variance seen in Figure 1.

Depth limits, minimum leaf sizes and cost-complexity pruning constrain complexity. Because splits depend only on the ordering of each feature, trees are invariant to monotone rescaling, which makes scaling less critical than for distance-based models; missing values, feature definitions and train-only transformations still require care. Theodoridis and Koutroumbas (Chapter 4) and Bishop (Section 14.4) treat trees as one family of nonlinear classifiers [1], [2].

Bagging and random forests

Plain version. Bagging ("bootstrap aggregating") builds many training sets by sampling the original data *with replacement*, trains one unstable model on each, and averages or votes. Random forests add one more source of randomness, a random subset of features at every split, so that the trees disagree more and their average improves more.

Precise version. Breiman introduced bagging as a way to improve *unstable* procedures, those whose output changes a lot under small perturbations of the training set [4]. Given B bootstrap samples $D^{(1)}, \dots, D^{(B)}$, each of size n, the bagged regressor is $\hat{f}_{\text{bag}}(x) = \frac{1}{B}\sum_b \hat{f}_{D^{(b)}}(x)$ and the bagged classifier takes a majority vote (or averages class probabilities).

Why does averaging help? Suppose M scalar predictors at a fixed x each have variance σ^2 and every pair has correlation ρ. Expanding the variance of a sum,

$$\operatorname{Var}\Big(\frac{1}{M}\sum_{m=1}^{M} f_m\Big) = \frac{1}{M^2}\Big[M\sigma^2 + M(M-1)\rho\sigma^2\Big] = \sigma^2\left[\rho + \frac{1-\rho}{M}\right].$$

The second term vanishes as M grows, but the first term $\rho\sigma^2$ does not: correlated errors set a floor (Figure 2, left). Averaging leaves the bias essentially unchanged, so bagging helps most for low-bias, high-variance learners such as deep trees, and little for stable learners such as linear discriminants or k-nearest neighbors with large k. This illustrative formula explains why correlated errors limit gains; it does not promise that every classification ensemble improves accuracy.

Random forests reduce ρ directly [5]. When growing each tree on its bootstrap sample, every split considers only a random subset of m_{try} features (a common default for classification is $\sqrt{d}$ of d features). Breiman showed that the forest's generalization error is bounded in terms of the *strength* of the individual trees and the *correlation* between them [5], which is the same trade-off as the variance formula: weaker but less correlated trees can make a stronger forest.

Out-of-bag (OOB) estimates. The probability that a given observation is *not* drawn in n draws with replacement is $(1-1/n)^n \to e^{-1} \approx 0.368$. Each tree therefore leaves out about 37% of the data, and each observation can be predicted by the roughly one third of trees that never saw it. Aggregating those predictions gives an OOB error estimate essentially for free [5]. In Figure 2 (right) the OOB estimate tracks the test error once there are enough trees. But ordinary bootstrapping resamples *rows*: if several rows share a subject, writer or recording, a tree can train on one image of a person and be "tested" OOB on another image of the same person. The OOB score is then optimistic for deployment to new groups.

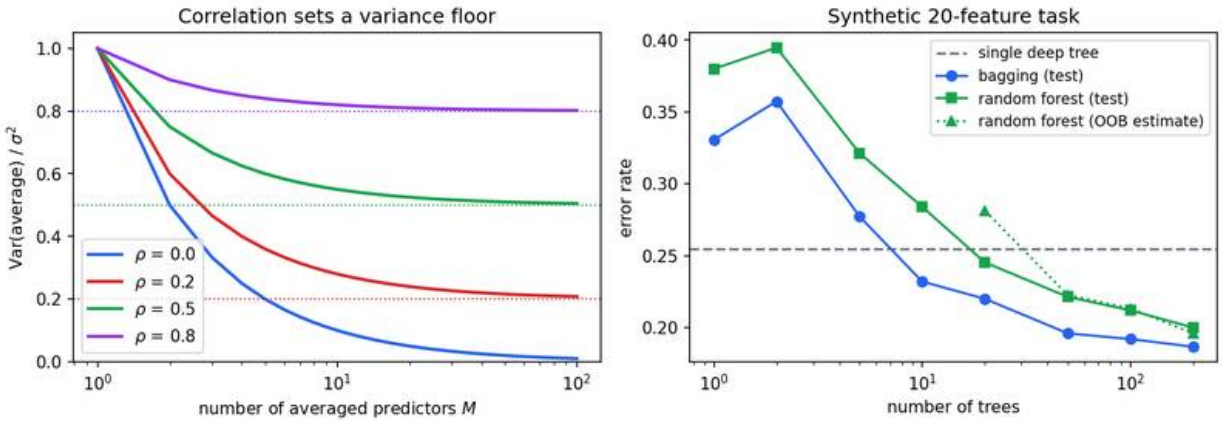


Figure 2 — Left: analytic variance of an average of M equally correlated predictors. Right: one synthetic 20-feature task (750 training, 750 test samples). Both ensembles beat a single deep tree once they have tens of trees; on this particular task plain bagging is slightly better than the forest, a reminder that random feature subsets are not guaranteed to help. Votes with very few trees (1–2) are unstable and can tie.

Interactive demonstration (online): open the online edition of this tutorial to use this demonstration.Open demonstration

Boosting changes the next learner's job

Plain version. Bagging trains its members independently and in parallel. Boosting trains them *in sequence*: each new weak learner concentrates on the examples the current ensemble still gets wrong, and the final prediction is a weighted sum. Bagging mainly reduces variance; boosting mainly reduces bias, turning many "slightly better than chance" rules into a strong classifier.

Precise version. Boosting builds an additive model

$$F_M(x) = \sum_{m=1}^{M} \alpha_m h_m(x),$$

where each h_m is a weak learner (often a depth-1 tree, a *stump*) and α_m its weight. The class prediction for labels $y \in \{-1, +1\}$ is $\operatorname{sign} F_M(x)$.

The AdaBoost algorithm

Freund and Schapire's AdaBoost [6] works as follows, for training pairs (x_i, y_i), $i = 1, \dots, n$:

1. Start with uniform weights $w_i^{(1)} = 1/n$.
2. For $m = 1, \dots, M$:
   - fit h_m to the training data using weights $w^{(m)}$;
   - compute the weighted error $\epsilon_m = \sum_i w_i^{(m)} \mathbf{1}[h_m(x_i) \neq y_i] / \sum_i w_i^{(m)}$;
   - set $\alpha_m = \frac{1}{2} \log \frac{1-\epsilon_m}{\epsilon_m}$;
   - update $w_i^{(m+1)} \propto w_i^{(m)} \exp\big(-\alpha_m y_i h_m(x_i)\big)$ and renormalize.
3. Output $\operatorname{sign} \sum_m \alpha_m h_m(x)$.

Misclassified points have $y_i h_m(x_i) = -1$, so their weight is multiplied by e^{α_m}; correctly classified points are multiplied by $e^{-\alpha_m}$. A learner with $\epsilon_m < 0.5$ gets a positive vote, and the better it is, the larger the vote.

Deriving AdaBoost from exponential loss

Friedman, Hastie and Tibshirani showed that AdaBoost is **forward stagewise additive modeling** with the **exponential loss** $L(y, F) = \exp(-yF)$ [7]. Suppose F_{m-1} is fixed and we add one term αh. The training loss is

$$\sum_{i=1}^{n} \exp\big(-y_i[F_{m-1}(x_i) + \alpha h(x_i)]\big) = \sum_{i=1}^{n} w_i^{(m)} \exp\big(-\alpha y_i h(x_i)\big), \qquad w_i^{(m)} = \exp\big(-y_i F_{m-1}(x_i)\big).$$

The weights $w_i^{(m)}$ do not depend on α or h. Split the sum into correctly and incorrectly classified points:

$$\sum_i w_i^{(m)} e^{-\alpha y_i h(x_i)} = e^{-\alpha} \sum_{y_i = h(x_i)} w_i^{(m)} + e^{\alpha} \sum_{y_i \neq h(x_i)} w_i^{(m)} = (e^{\alpha} - e^{-\alpha}) \sum_i w_i^{(m)} \mathbf{1}[y_i \neq h(x_i)] + e^{-\alpha} \sum_i w_i^{(m)}.$$

Step 1, choose h. For any $\alpha > 0$, the first coefficient is positive, so the best h minimizes the **weighted training error**: exactly "fit h_m with weights $w^{(m)}$."

Step 2, choose α. Divide by $\sum_i w_i^{(m)}$ and write ϵ for the normalized weighted error. We minimize $g(\alpha) = e^{-\alpha}(1-\epsilon) + e^{\alpha}\epsilon$. Setting $g'(\alpha) = -e^{-\alpha}(1-\epsilon) + e^{\alpha}\epsilon = 0$ gives $e^{2\alpha} = (1-\epsilon)/\epsilon$, that is,

$$\alpha_m = \frac{1}{2}\log\frac{1-\epsilon_m}{\epsilon_m}.$$

Step 3, update weights. The next weights are $w_i^{(m+1)} = \exp(-y_i F_m(x_i)) = w_i^{(m)}\exp(-\alpha_m y_i h_m(x_i))$, the AdaBoost update. Every step of the algorithm is thus recovered from greedy minimization of exponential loss.

Two consequences follow. Since $\mathbf{1}[yF < 0] \le e^{-yF}$, the training error is bounded by the average exponential loss; with the optimal α_m, each round multiplies that bound by $2\sqrt{\epsilon_m(1-\epsilon_m)} < 1$ whenever $\epsilon_m < 0.5$, so training error falls exponentially fast if every weak learner beats chance by a margin [6]. And the minimizer of expected exponential loss is $F^*(x) = \frac{1}{2}\log\frac{P(y=1|x)}{P(y=-1|x)}$, half the log-odds [7], so $\sigma(2F_M(x))$ is a (not necessarily well-calibrated) probability estimate. The exponential weight also explains a known weakness: a mislabeled point receives exponentially growing weight, so AdaBoost can be sensitive to label noise and outliers.

The code implements AdaBoost from scratch with weighted stumps and checks it against scikit-learn's `AdaBoostClassifier`. For two classes scikit-learn's SAMME variant uses $\alpha = \log\frac{1-\epsilon}{\epsilon}$, twice ours, which rescales F but leaves every sign, and every renormalized weight, unchanged.

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier

X, y01 = make_classification(n_samples=600, n_features=10, n_informative=4,
                             flip_y=0.05, random_state=1)
y = 2 * y01 - 1                                   # labels in {-1, +1}
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.3, random_state=1)

def adaboost_fit(X, y, M=100):
    n = len(y)
    w = np.full(n, 1 / n)                         # step 1: uniform weights
    stumps, alphas = [], []
    for m in range(M):
        h = DecisionTreeClassifier(max_depth=1, random_state=0)
        h.fit(X, y, sample_weight=w)              # step 2a: weighted fit
        pred = h.predict(X)
        err = np.sum(w * (pred != y)) / np.sum(w) # step 2b: weighted error
        err = np.clip(err, 1e-12, 1 - 1e-12)
        alpha = 0.5 * np.log((1 - err) / err)     # step 2c: learner weight
        w = w * np.exp(-alpha * y * pred)         # step 2d: reweight
        w /= w.sum()
        stumps.append(h); alphas.append(alpha)
    return stumps, np.array(alphas)
```

```
def adaboost_score(stumps, alphas, X):
    return sum(a * h.predict(X) for h, a in zip(stumps, alphas))

stumps, alphas = adaboost_fit(Xtr, ytr)
F_tr = adaboost_score(stumps, alphas, Xtr)
F_te = adaboost_score(stumps, alphas, Xte)
print("from scratch: train err", np.mean(np.sign(F_tr) != ytr),
      " test err", np.mean(np.sign(F_te) != yte))
print("exp-loss bound on train err:", np.mean(np.exp(-ytr * F_tr)))

ref = AdaBoostClassifier(DecisionTreeClassifier(max_depth=1, random_state=0),
                         n_estimators=100, random_state=0).fit(Xtr, ytr)
agree = np.mean(ref.predict(Xte) == np.sign(F_te))
print("sklearn test err", np.mean(ref.predict(Xte) != yte),
      " agreement with ours:", agree)
```

With scikit-learn 1.6 our run prints a training error of 0.076, an exponential-loss bound of 0.384, a test error of 0.156 for both implementations and 100% agreement on test predictions. Figure 3 shows the curves over 300 rounds and the staircase boundary that stumps build in two dimensions.

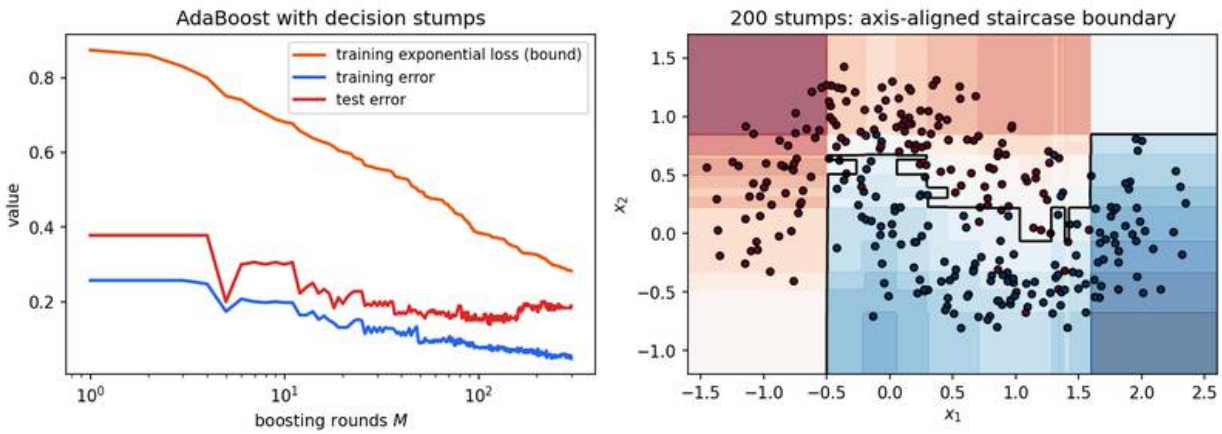


Figure 3 — AdaBoost with decision stumps (our from-scratch implementation). Left: the exponential loss upper-bounds the training error and keeps shrinking; test error plateaus and can drift upward slightly with label noise. Right: summing axis-aligned stumps yields a staircase decision boundary on the two-moons data.

Gradient boosting

Plain version. Gradient boosting generalizes the AdaBoost idea to any differentiable loss: at each round, compute in which direction each training prediction should move to reduce the loss, and fit a small tree to those directions.

Precise version. Friedman's gradient boosting machine [8] performs gradient descent in function space. At round m, compute the **pseudo-residuals**

$$r_{im} = -\left.\frac{\partial L(y_i, F)}{\partial F}\right|_{F=F_{m-1}(x_i)},$$

fit a regression tree h_m to the pairs (x_i, r_{im}) by least squares, choose a step ρ_m by line search (or one optimal constant per leaf, Friedman's TreeBoost), and update $F_m = F_{m-1} + \nu\rho_m h_m$ with a **learning rate** (shrinkage) $0 < \nu \leq 1$. For squared error $L = \frac{1}{2}(y - F)^2$ the pseudo-residual is the ordinary residual $y_i - F_{m-1}(x_i)$. For binary log loss with $y \in \{0, 1\}$ and F the log-odds, it is $y_i - \sigma(F_{m-1}(x_i))$: the label minus the *predicted probability*, not minus the predicted class. With exponential loss and a line search, the procedure reduces to a version of AdaBoost.

The learning rate, the number of rounds and the tree size interact: a smaller ν needs more rounds but usually generalizes better, and the number of rounds is chosen by early stopping on validation data, never on the final test. XGBoost [9] made gradient-boosted trees fast and scalable through a regularized objective, second-order (Newton) leaf values, sparsity-aware split finding and a weighted quantile sketch; histogram-based implementations such as scikit-learn's `HistGradientBoostingClassifier` follow the same principles [26].

Averaging is not always the same operation

Many methods "combine models," but they answer different questions.

Method	What supplies the combination?	Main caution
Voting or probability averaging (bagging, forests)	fixed equal combination of independently trained predictors	correlated errors and incompatible calibration
Boosting	sequential fitting; weights from a loss	label noise; needs early stopping
Bayesian model averaging	posterior probability over model hypotheses	evidence and priors must be meaningful
Stacking	a learned predictor of base-model outputs	train it on out-of-fold predictions
Conditional mixtures (mixtures of experts)	input-dependent gates or responsibilities	experts and gate can overfit together

Bayesian model averaging (Bishop Section 14.1 [2]) uses

$$p(y \mid x, D) = \sum_m p(y \mid x, D, M_m)\, p(M_m \mid D),$$

where $p(M_m \mid D) \propto p(D \mid M_m)p(M_m)$ is the posterior probability of model M_m. Those weights are posterior model probabilities, not arbitrary validation accuracies, and with plenty of data they tend to concentrate on one model. BMA expresses uncertainty about *which* model generated the data; a committee or ensemble instead builds a richer model out of several parts [2]. The model evidence $p(D \mid M_m)$ connects to the approximate-inference tools of PR 08.

Stacking (stacked generalization, Wolpert [10]) instead *learns* a mapping from base-model predictions to the target. Its training features must come from base models that did not train on those same examples; otherwise the stacker learns from unrealistically easy, overfitted predictions. The standard recipe: split the training set into K folds, fit each base model on $K - 1$ folds and predict

the held-out fold, collect these out-of-fold predictions as features, fit a simple meta-learner (often regularized logistic regression) on them, and finally refit the base models on all training data.

The following comparison on scikit-learn's bundled breast-cancer dataset (569 samples, 30 features) runs every method inside the same stratified 5-fold cross-validation, so all numbers are comparable. `StackingClassifier` builds its own inner out-of-fold predictions (`cv=5`) inside each outer training fold.

```
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import StratifiedKFold, cross_val_score
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import (BaggingClassifier, RandomForestClassifier,
                              HistGradientBoostingClassifier, StackingClassifier)

X, y = load_breast_cancer(return_X_y=True)
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=0)
logreg = make_pipeline(StandardScaler(), LogisticRegression(max_iter=2000))
models = {
    "single tree": DecisionTreeClassifier(random_state=0),
    "bagged trees": BaggingClassifier(DecisionTreeClassifier(), n_estimators=100,
                                      random_state=0),
    "random forest": RandomForestClassifier(n_estimators=100, random_state=0),
    "hist. gradient boosting": HistGradientBoostingClassifier(random_state=0),
    "logistic regression": logreg,
}
models["stacking"] = StackingClassifier(
    estimators=[("rf", models["random forest"]),
                ("gb", models["hist. gradient boosting"]),
                ("lr", logreg)],
    final_estimator=LogisticRegression(max_iter=2000),
    cv=5, stack_method="predict_proba")
for name, model in models.items():
    s = cross_val_score(model, X, y, cv=cv, scoring="accuracy")
    print(f"{name:>24}: {s.mean():.3f} +/- {s.std():.3f}")
```

Our run gives mean accuracies of 0.926 (single tree), 0.960 (bagging), 0.965 (forest), 0.970 (gradient boosting), 0.979 (logistic regression) and 0.979 (stacking), with fold standard deviations between 0.009 and 0.022. Two lessons follow. Ensembling clearly fixes the unstable single tree. But a well-regularized *linear* model is already as good as the stack on this small, nearly linearly separable dataset, and the differences among the top four are within one fold standard deviation. Whether any of them is *significantly* better is a question for the statistical tests later in this chapter, not for eyeballing means.

Validation estimates a procedure

Plain version. A test score is only trustworthy if the test data played no part in building the model. Every choice we make while looking at data, including features, hyperparameters, thresholds, and even which algorithm to use, is part of the "model," and must be made without the data used to judge it.

Training fits parameters. Validation selects hyperparameters, features, thresholds and calibration choices. Testing assesses the selected procedure. Every adaptive choice consumes information from whichever data it examines. Strictly speaking, cross-validation estimates the performance of a *procedure* (an algorithm plus its tuning rules) trained on data sets of a given size, rather than the error of the one fitted model you will ship; Bates, Hastie and Tibshirani make this precise and show that naive cross-validation confidence intervals can undercover [19].

Hold-out, k-fold and leave-one-out

- **Hold-out.** Split once into training and test parts. Simple and cheap, but with small data the estimate has high variance and depends on the split, and the model is trained on less data than you have.
- **k-fold cross-validation.** Partition the data into k folds; train on $k-1$ and score the remaining one, rotating so each fold is held out once (Figure 4, top). The CV estimate is $\widehat{\mathrm{Err}}_{\mathrm{CV}} = \frac{1}{n}\sum_{i} L(y_i, \hat{f}^{(-\kappa(i))}(x_i))$, where $\kappa(i)$ is the fold containing sample i and $\hat{f}^{(-k)}$ is the model trained without fold k. Values $k = 5$ or 10 are common compromises: each model sees 80–90% of the data, so the pessimistic bias is small, and the cost is k fits.
- **Leave-one-out (LOO).** The case $k = n$. Nearly unbiased for the size-n procedure, but it requires n fits (unless a closed form exists, as for linear smoothers or k-NN) and can have high variance because the n training sets are almost identical.
- **Stratification** keeps class proportions similar across folds, which matters for rare classes. It does not fix dependence between samples.

Arlot and Celisse survey the theory of these choices for both estimating risk and selecting models [18].

Groups and time

For observations sharing a patient, writer, camera or recording session, hold **whole groups** together (Figure 4, middle). Otherwise near-duplicate images of the same person land on both sides of the split and the score measures memorization of that person, not recognition of new people. Time-ordered data needs chronological splits (train on the past, test on the future). The rule is to make the split mirror the deployment question: "new images of known writers" and "images of new writers" are different questions and deserve different splits.

The bootstrap, .632 and .632+

Plain version. Instead of folds, draw many bootstrap samples, train on each, and test on the points each one left out.

Precise version. Draw B bootstrap samples of size n with replacement. For each, fit the model and evaluate it on the out-of-bootstrap points (about 36.8% of the data). The average is the leave-one-out bootstrap error $\widehat{\mathrm{Err}}^{(1)}$. Because each bootstrap model is trained on only about 63.2% of *distinct* points, $\widehat{\mathrm{Err}}^{(1)}$ is pessimistic, while the resubstitution (training) error $\overline{\mathrm{err}}$ is optimistic. Efron's **.632 estimator** blends them:

$$\widehat{\mathrm{Err}}^{(.632)} = 0.368\,\overline{\mathrm{err}} + 0.632\,\widehat{\mathrm{Err}}^{(1)}.$$

It fails for heavily overfitting learners such as 1-nearest neighbor, whose training error is zero. Efron and Tibshirani's **.632+ estimator** [11] fixes this by moving weight toward $\widehat{\mathrm{Err}}^{(1)}$ as overfitting grows. Let $\hat{\gamma}$ be the *no-information error rate* (the error if inputs and labels were independent: $\hat{\gamma} = \sum_k \hat{p}_k(1-\hat{q}_k)$, with $\hat{p}_k$ the observed proportion of class k and $\hat{q}_k$ the proportion of predictions equal to k), and define the relative overfitting rate $\hat{R} = (\widehat{\mathrm{Err}}^{(1)} - \overline{\mathrm{err}})/(\hat{\gamma} - \overline{\mathrm{err}})$, clipped to $[0,1]$. Then

$$\hat{w} = \frac{0.632}{1 - 0.368\hat{R}}, \qquad \widehat{\mathrm{Err}}^{(.632+)} = (1-\hat{w})\,\overline{\mathrm{err}} + \hat{w}\,\widehat{\mathrm{Err}}^{(1)}.$$

When $\hat{R} = 0$ (no overfitting) it reduces to .632; when $\hat{R} = 1$ it becomes $\widehat{\mathrm{Err}}^{(1)}$.

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score, KFold

X, y = make_classification(n_samples=200, n_features=6, n_informative=3,
                           flip_y=0.1, random_state=3)
model = KNeighborsClassifier(n_neighbors=1)          # resubstitution error is 0
rng = np.random.default_rng(3)
n, B = len(y), 200
resub = np.mean(model.fit(X, y).predict(X) != y)
oob_errs = []
for b in range(B):
    idx = rng.integers(0, n, n)                       # draw n with replacement
    out = np.setdiff1d(np.arange(n), idx)             # ~36.8% left out
    model.fit(X[idx], y[idx])
    oob_errs.append(np.mean(model.predict(X[out]) != y[out]))
err_boot1 = np.mean(oob_errs)                          # leave-one-out bootstrap
err_632 = 0.368 * resub + 0.632 * err_boot1
cv10 = 1 - cross_val_score(model, X, y,
                           cv=KFold(10, shuffle=True, random_state=3)).mean()
print(f"resubstitution {resub:.3f}  OOB bootstrap {err_boot1:.3f}  "
      f".632 {err_632:.3f}  10-fold CV {cv10:.3f}")

# .632+ : shrink toward the out-of-bootstrap error when overfitting is large.
```

```
pred = model.fit(X, y).predict(X)
gamma = sum(np.mean(y == k) * (1 - np.mean(pred == k)) for k in np.unique(y))
R = np.clip((err_boot1 - resub) / (gamma - resub), 0, 1)
w = 0.632 / (1 - 0.368 * R)
print(f"no-information rate {gamma:.3f}  R {R:.3f}  "
      f".632+ {(1 - w) * resub + w * err_boot1:.3f}")
```

For 1-NN the resubstitution error is 0.000, the out-of-bootstrap error 0.332, the .632 estimate only 0.210 (pulled down by the meaningless zero), the .632+ estimate 0.278 and 10-fold CV 0.325. The .632 estimator is badly optimistic here; .632+ repairs most of the gap. In practice, 10-fold CV (possibly repeated) and .632+ are both reasonable; the bootstrap is attractive for very small samples and for confidence intervals.

Nested cross-validation

Plain version. If you tune hyperparameters by cross-validation and then report that same cross-validation score, you report the score of the *luckiest* configuration, which is optimistic. Nested CV puts the whole tuning process inside an outer loop.

Nested cross-validation has an **inner loop** for selection and an **outer loop** for evaluating that selection procedure (Figure 4, bottom). Within each outer training set, an inner CV chooses hyperparameters; the chosen configuration is refit on the whole outer training set and scored on the outer held-out fold, which the inner loop never saw. Varma and Simon showed that reporting the inner (tuning) CV score can be substantially biased, and that nested CV gives nearly unbiased estimates [12]. Cawley and Talbot went further: overfitting the model-selection criterion itself can cost as much as the differences between learning algorithms, so a low-variance selection criterion matters as much as an unbiased one [13]. Refit preprocessing (scaling, feature selection, PCA) inside each inner fold, typically by placing it in a `Pipeline`. Report variability across outer folds, but do not treat correlated fold scores as independent measurements for a naive confidence interval.

After estimating the procedure, select on development data and evaluate once on an untouched test. Repeatedly inspecting test errors and redesigning the model turns that test into another validation set.

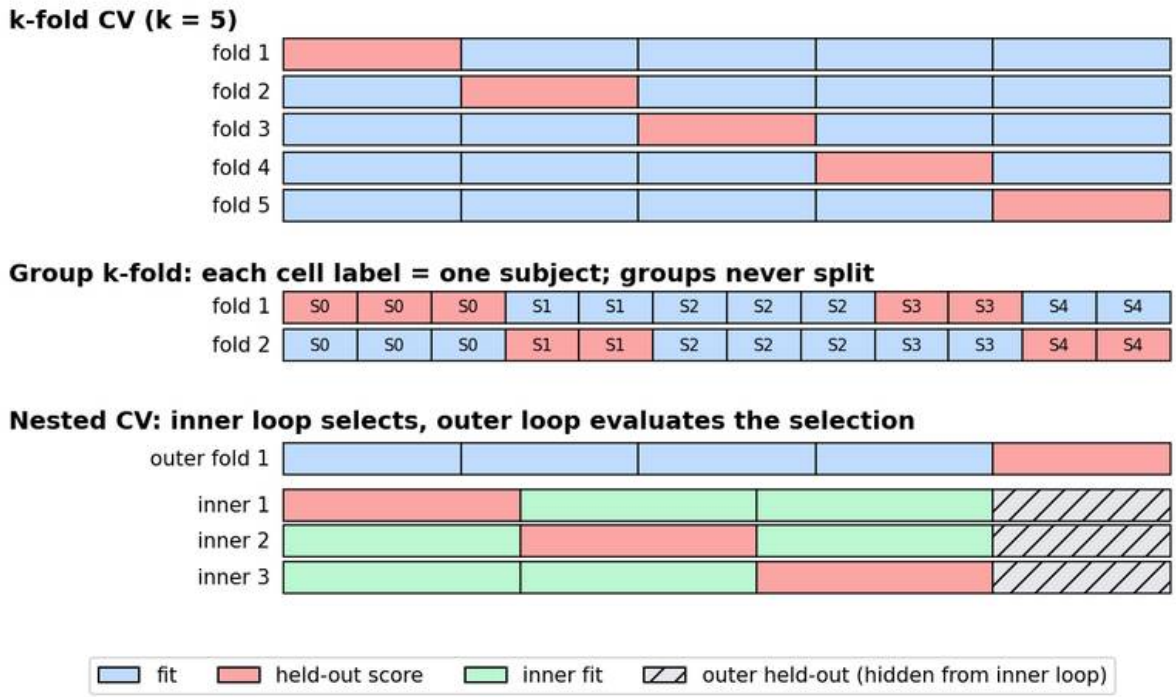


Figure 4 — Three validation layouts. Top: 5-fold CV. Middle: group k-fold keeps every observation of a subject (S0–S4) on one side of each split. Bottom: nested CV; the inner loop tunes on the outer training data only, while the outer held-out fold (hatched) stays hidden until the tuned model is scored.

Code: nested selection without group leakage

This CPU example creates three noisy repeated measurements of each synthetic subject. Group labels describe subjects, not classes. The outer loop estimates the selected forest procedure; a separate group-disjoint test set remains untouched until final reporting.

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split, GroupKFold, GridSearchCV
from sklearn.metrics import balanced_accuracy_score, log_loss

rng = np.random.default_rng(9)
subject_x, subject_y = make_classification(
    n_samples=180, n_features=8, n_informative=5, n_redundant=1,
    class_sep=1.0, random_state=9)
groups = np.repeat(np.arange(180), 3)
X = subject_x[groups] + rng.normal(0, 0.03, (len(groups), 8))
y = subject_y[groups]
dev_groups, test_groups = train_test_split(
    np.arange(180), test_size=0.25, stratify=subject_y, random_state=9)
dev = np.flatnonzero(np.isin(groups, dev_groups))
```

```
test = np.flatnonzero(np.isin(groups, test_groups))
assert not set(groups[dev]) & set(groups[test])

def search():
    return GridSearchCV(
        RandomForestClassifier(n_estimators=60, random_state=9, n_jobs=1),
        {"max_depth": [3, None], "min_samples_leaf": [1, 3]},
        cv=GroupKFold(3), scoring="balanced_accuracy")

outer_scores = []
for fit, valid in GroupKFold(3).split(X[dev], y[dev], groups[dev]):
    chosen = search().fit(X[dev[fit]], y[dev[fit]], groups=groups[dev[fit]])
    pred = chosen.predict(X[dev[valid]])
    outer_scores.append(balanced_accuracy_score(y[dev[valid]], pred))
final = search().fit(X[dev], y[dev], groups=groups[dev])
p = final.predict_proba(X[test])
pred = final.predict(X[test])
positive = p[:, np.flatnonzero(final.classes_ == 1)[0]]
print("Nested outer balanced accuracy:", np.round(outer_scores, 3))
print("Selected parameters:", final.best_params_)
print("Independent group test:", balanced_accuracy_score(y[test], pred))
print("Test log loss:", log_loss(y[test], p, labels=final.classes_))
print("Test Brier:", np.mean((positive - y[test]) ** 2))
```

Our run prints outer balanced accuracies of 0.869, 0.784 and 0.861, selects `max_depth=None, min_samples_leaf=3`, and scores 0.853 balanced accuracy, 0.378 log loss and 0.118 Brier score on the independent group test. The spread of the outer scores (0.78 to 0.87) is a useful reminder of how noisy a single split would be. These results describe the generated subjects; they do not establish performance on images. Replacing `GroupKFold` by shuffled row-level folds would let near-copies of a subject appear on both sides and inflate the scores (Exercise 3).

Comparing classifiers statistically

Plain version. If classifier A scores 97.9% and B scores 96.5%, is A really better, or did it get lucky with this particular data? A statistical test asks how surprising the observed difference would be if the two were equally good. The catch is that the usual textbook t-test assumes independent measurements, and cross-validation scores are not independent.

Dietterich's study of approximate tests [14] is the classic starting point; Raschka's tutorial collects the main options with code [20].

One test set: McNemar's test

Train both classifiers once and evaluate them on the *same* test set. Count n_{01}, the test points A gets right and B gets wrong, and n_{10}, the reverse. Points that both get right or both get wrong carry no information about which is better. Under the null hypothesis of equal error rates, each

disagreement is equally likely to go either way, so $n_{01} \sim \text{Binomial}(n_{01}+n_{10}, \tfrac{1}{2})$. The continuity-corrected statistic

$$\chi^2 = \frac{(|n_{01} - n_{10}| - 1)^2}{n_{01} + n_{10}}$$

is approximately χ^2 with one degree of freedom; for small counts use the exact binomial test. Dietterich found McNemar's test has acceptable Type I error and recommended it when the algorithms can be run only once [14]. It does not account for variability due to the choice of training set.

Repeated splits: why the naive t-test fails

A tempting alternative is to repeat random train/test splits J times, record the error differences d_j, and run a paired t-test on them. Dietterich showed this **resampled t-test** has a badly inflated Type I error [14]: the training sets overlap heavily, so the d_j are positively correlated and the usual standard error $\hat{\sigma}/\sqrt{J}$ is far too small. Two fixes are common.

- **5×2cv paired t-test** [14]. Do five replications of 2-fold CV. In replication i, let $p_i^{(1)}, p_i^{(2)}$ be the error differences on the two folds, $\bar{p}_i$ their mean and $s_i^2 = (p_i^{(1)} - \bar{p}_i)^2 + (p_i^{(2)} - \bar{p}_i)^2$. The statistic $t = p_1^{(1)} / \sqrt{\tfrac{1}{5}\sum_{i=1}^{5} s_i^2}$ is compared with a t distribution with 5 degrees of freedom.
- **Corrected resampled t-test** (Nadeau and Bengio [15]). Keep J random splits with n_{test} test and n_{train} training points, but inflate the variance to account for overlap:

$$t = \frac{\bar{d}}{\sqrt{\left(\dfrac{1}{J} + \dfrac{n_{\text{test}}}{n_{\text{train}}}\right)\hat{\sigma}_d^2}},$$

where $\bar{d}$ and $\hat{\sigma}_d^2$ are the mean and sample variance of the d_j, compared with a t distribution with $J-1$ degrees of freedom. The extra $n_{\text{test}}/n_{\text{train}}$ term does not shrink as J grows, which is the point: more resamples of the same data cannot create more information.

```
import numpy as np
from scipy import stats
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split, ShuffleSplit
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier

X, y = load_breast_cancer(return_X_y=True)
A = make_pipeline(StandardScaler(), LogisticRegression(max_iter=2000))
B = RandomForestClassifier(n_estimators=200, random_state=0)

# 1) McNemar's test on one held-out test set (paired predictions).
```

```
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.4, stratify=y,
                                      random_state=0)
ca = A.fit(Xtr, ytr).predict(Xte) == yte
cb = B.fit(Xtr, ytr).predict(Xte) == yte
n01 = int(np.sum(ca & ~cb))      # A right, B wrong
n10 = int(np.sum(~ca & cb))      # A wrong, B right
p_exact = stats.binomtest(n01, n01 + n10, 0.5).pvalue if n01 + n10 else 1.0
chi2 = (abs(n01 - n10) - 1) ** 2 / (n01 + n10) if n01 + n10 else 0.0
print(f"McNemar: n01={n01} n10={n10} chi2_cc={chi2:.3f} "
      f"p_chi2={stats.chi2.sf(chi2, 1):.3f} p_exact={p_exact:.3f}")

# 2) Corrected resampled t-test (Nadeau & Bengio) over J random splits.
J, test_frac = 30, 0.2
d = []
for tr, te in ShuffleSplit(J, test_size=test_frac, random_state=0).split(X):
    ea = np.mean(A.fit(X[tr], y[tr]).predict(X[te]) != y[te])
    eb = np.mean(B.fit(X[tr], y[tr]).predict(X[te]) != y[te])
    d.append(ea - eb)
d = np.array(d)
naive_se = np.sqrt(d.var(ddof=1) / J)
corr_se = np.sqrt(d.var(ddof=1) * (1 / J + test_frac / (1 - test_frac)))
for name, se in [("naive", naive_se), ("corrected", corr_se)]:
    t = d.mean() / se
    print(f"{name:>9} t={t:.3f}  p={2 * stats.t.sf(abs(t), J - 1):.3f}")
```

Our run gives $n_{01} = 9$, $n_{10} = 5$ and McNemar $p \approx 0.42$: no evidence of a difference on one test set of 228 cases. Over 30 random 80/20 splits, logistic regression has on average about 1.8 percentage points lower error. The naive t-test reports $t \approx -5.4, p < 0.001$; the corrected test reports $t \approx -1.85$, $p \approx 0.075$ (Figure 5). Same data, opposite conclusions at the 5% level: the naive test mistakes 30 heavily overlapping resamples for 30 independent experiments.

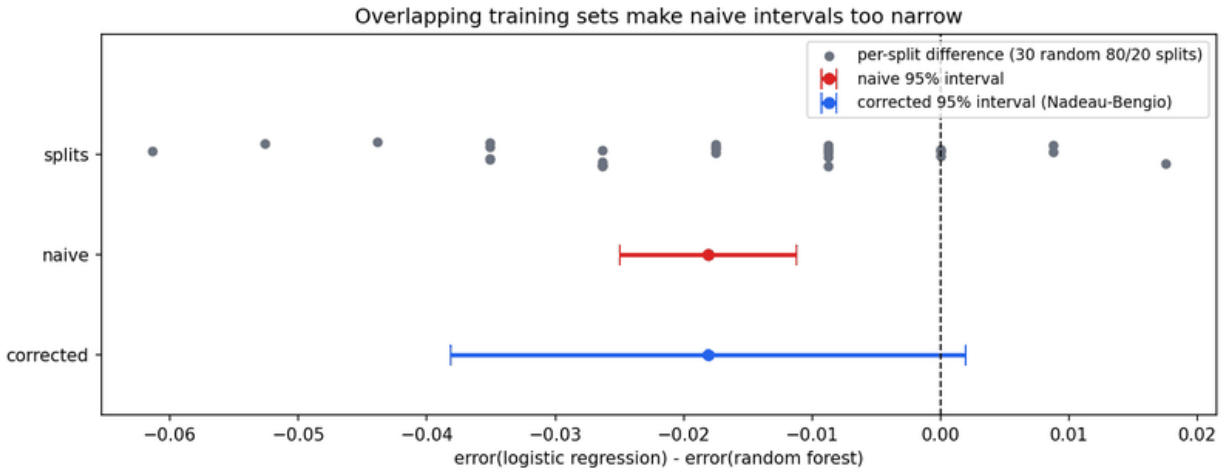


Figure 5 — Error differences (logistic regression minus random forest) on the breast-cancer data over 30 random 80/20 splits. The naive paired t interval excludes zero; the Nadeau–Bengio corrected interval, which accounts for overlapping training sets, does not.

Many classifiers on many datasets

When comparing k algorithms over N datasets, Demšar recommends non-parametric tests on ranks [16]: the Wilcoxon signed-rank test for two classifiers, and for more the **Friedman test**. Rank the algorithms on each dataset (1 = best) and let R_j be the average rank of algorithm j. Under the null hypothesis that all are equivalent,

$$\chi_F^2 = \frac{12N}{k(k+1)}\left[\sum_{j=1}^{k} R_j^2 - \frac{k(k+1)^2}{4}\right]$$

is approximately χ^2 with $k-1$ degrees of freedom (Demšar also gives a less conservative F-distributed variant). If the null is rejected, the **Nemenyi post-hoc test** declares two algorithms different when their average ranks differ by more than the critical difference $\mathrm{CD} = q_\alpha \sqrt{k(k+1)/(6N)}$, where q_α comes from the Studentized range statistic divided by $\sqrt{2}$; Demšar's *critical difference diagrams* display the result compactly [16]. The datasets, not the folds, are the independent units here.

Benavoli, Corani, Demšar and Zaffalon argue for going further and replacing null-hypothesis significance testing with **Bayesian analysis** of the differences, which reports the probability that one classifier is practically better, practically equivalent within a region of practical equivalence, or worse [17]. Whatever the test, remember that a p-value says nothing about *how large* the difference is; report effect sizes and intervals as well.

Measure decisions and probabilities

Accuracy measures the proportion correct; **balanced accuracy** averages per-class recall, so it is not dominated by the majority class. **Macro-F1** weights classes equally when averaging classwise F1. A confusion matrix shows which errors occur. For rare positives, examine precision and recall at an operational threshold, and the full precision–recall or ROC curve when no threshold is fixed yet.

Log loss and the Brier score assess probability predictions:

$$\mathrm{LogLoss} = -\frac{1}{N}\sum_{n} \log p_n(y_n), \qquad \mathrm{Brier} = \frac{1}{N}\sum_{n} (p_n - y_n)^2,$$

where $p_n(y_n)$ is the probability the model gave to the true class of example n, and in the Brier score (binary case) p_n is the predicted probability of the positive class and $y_n \in \{0, 1\}$. Both are *proper scoring rules*: they are optimized in expectation by reporting the true probabilities. A **reliability diagram** compares predicted probability with observed frequency in bins. Good ranking, good accuracy and good calibration are different properties: random forests average votes and tend to push probabilities toward the middle, boosting with many rounds can push them toward the extremes, and a model can have excellent ROC AUC with poorly calibrated scores.

Fit a calibrator (scikit-learn offers sigmoid and isotonic mappings [26]) on predictions from data excluded from the corresponding base-model fitting. If observations are grouped, the calibration

folds must respect those groups too. Calibration does not automatically repair domain shift.

Cluster validity as model selection

Clustering has no labels, so "validation" means something different. Theodoridis and Koutroumbas devote Chapter 16 to cluster validity because a tidy partition need not serve the task [1]. The full set of indices (silhouette, Davies–Bouldin, Calinski–Harabasz, gap statistic, BIC for mixtures) is developed alongside the algorithms in PR 07; here we place them in the validation framework of this chapter.

Internal criteria use only features and assignments. The silhouette of point i compares its mean distance to its own cluster, $a(i)$, with the smallest mean distance to another cluster, $b(i)$:

$$s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))} \in [-1, 1].$$

It favors separation in the selected geometry. Choosing the number of clusters by maximizing an internal index is a model-selection step, exactly like tuning a hyperparameter, and a high score is not evidence that groups represent the intended semantic categories.

External criteria compare assignments with independent reference labels. The adjusted Rand index (ARI) corrects pairwise agreement for chance and is invariant to label permutations, so it does not matter which cluster is called "1." If labels select the algorithm or the number of clusters, those labels are no longer an untouched external assessment; the separation between selection and evaluation applies here too.

Stability compares partitions after resampling or perturbation, much like the bootstrap: cluster several subsamples and measure agreement (for example ARI) on the observations they share, accounting for label permutation. Stable groups can still reflect an irrelevant camera or acquisition artifact.

Modern view

Surveys of ensemble learning. Dong, Yu, Cao, Shi and Ma survey ensemble learning broadly [21]: they review the mainstream ensemble approaches, classify them by their characteristics, list challenges and research directions for each, and discuss combining ensembles with deep learning and reinforcement learning. Ganaie, Hu, Malik, Tanveer and Suganthan review **ensemble deep learning** [22], organizing methods into bagging-, boosting- and stacking-based deep ensembles, negative-correlation learning, explicit versus implicit ensembles (for example snapshot or dropout-style ensembles that share weights), homogeneous versus heterogeneous members, and decision-fusion strategies. Across all of these categories the practical question is the one posed by the variance formula above: how to obtain *diverse* members cheaply.

Tree ensembles on tabular data. Despite the rise of deep networks, gradient-boosted trees (XGBoost [9] and its histogram-based successors) and random forests [5] remain the default for

medium-sized tabular problems, which in recognition include hand-crafted descriptors, metadata and the outputs of other models. Grinsztajn, Oyallon and Varoquaux benchmarked tree-based models and deep architectures on 45 datasets and found that tree-based models remained state of the art at around 10,000 samples, attributing the gap to neural networks' sensitivity to uninformative features, their rotation invariance, and their difficulty learning irregular functions [23].

Ensembles for uncertainty. Lakshminarayanan, Pritzel and Blundell showed that **deep ensembles**, several networks trained from different random initializations with a proper scoring rule, give predictive uncertainty that matched or beat approximate Bayesian methods in their experiments and degrade more gracefully on out-of-distribution inputs [24]. This links the bagging intuition (disagreement signals variance) to the Bayesian model averaging of PR 08, and it is the baseline that newer uncertainty methods are compared against.

Bias–variance revisited. Belkin, Hsu, Ma and Mandal documented **double descent** [25]: as capacity grows past the point where a model can interpolate the training data, test error can fall again. It does not overturn the decomposition, but it warns that "more parameters means more variance" is not a law, and that the classical U-shaped curve describes only part of the capacity range.

What cross-validation estimates. Arlot and Celisse's survey [18] remains the reference on CV's theory for risk estimation and model selection. Bates, Hastie and Tibshirani showed that CV estimates the average error of models fit on other training sets of the same size, not the error of the particular model in hand, and that standard CV confidence intervals can have poor coverage; they proposed a nested-CV estimate of the variance that gives intervals with better coverage [19]. Cawley and Talbot's warning about overfitting the selection criterion [13] has grown more relevant with automated hyperparameter search, where thousands of configurations are compared on the same folds.

Comparing algorithms. Practice has moved from naive resampled t-tests toward McNemar and corrected tests for one dataset [14], [15], rank-based Friedman/Nemenyi tests with critical-difference diagrams across datasets [16], and Bayesian comparisons with a region of practical equivalence [17]. Raschka's tutorial [20] is a practical guide to all of these.

Open problems. Honest evaluation under distribution shift (new cameras, sites or populations), where no random split mirrors deployment; leakage through large pretrained models whose pretraining data may contain the test images; reliable uncertainty from ensembles that are cheap at inference time; and statistical comparison when only a single large benchmark exists. ML 11 continues these themes for deep vision models.

Key takeaways

- For squared error, expected test error = noise + bias² + variance; flexible learners such as deep trees trade bias for variance.
- Averaging M predictors with pairwise correlation ρ leaves variance $\sigma^2[\rho + (1-\rho)/M]$: bagging and random forests help unstable learners, and decorrelating members (random feature

subsets) is the lever.
- AdaBoost is greedy forward-stagewise minimization of exponential loss; gradient boosting fits trees to the negative gradient of any differentiable loss, with shrinkage and early stopping as the main regularizers.
- Voting, boosting, Bayesian model averaging and stacking combine models differently; stacking must be trained on out-of-fold predictions.
- Validation estimates a *procedure*: every adaptive choice belongs inside the loop, splits must respect groups and time, and nested CV is needed to report a tuned model's performance.
- The bootstrap's .632+ estimator corrects .632's optimism for overfitting learners; out-of-bag scores ignore group structure.
- Do not run a naive t-test on overlapping CV folds; use McNemar, 5×2cv, the corrected resampled t-test, or Friedman/Nemenyi across datasets, and report effect sizes.
- Internal cluster indices, external agreement (ARI) and stability answer different questions; choosing K with them is model selection.

Exercises

1. **Correlation floor.** Derive the averaging-variance formula by expanding the variance of a sum. With $\rho = 0.8$, how many predictors are needed to bring the variance within 1% of its limit? What does this say about bagging very similar models?

   Hint

   The sum has M variance terms and $M(M-1)$ ordered covariance terms $\rho\sigma^2$. The limit is $\rho\sigma^2 = 0.8\sigma^2$; the excess is $(1-\rho)\sigma^2/M = 0.2\sigma^2/M$. Requiring $0.2/M \le 0.008$ gives $M \ge 25$. Beyond a few dozen highly correlated models almost nothing is gained; the only way to go lower is to reduce ρ.

2. **AdaBoost by hand.** Five points have weights 0.2 each. The first stump misclassifies exactly one of them. Compute ϵ_1, α_1 and the new normalized weights. What weighted error does the same stump have under the new weights, and why is that no accident?

   Hint

   $\epsilon_1 = 0.2$, $\alpha_1 = \frac{1}{2}\log 4 = \log 2 \approx 0.693$. Unnormalized weights: $0.2 \cdot 2 = 0.4$ for the misclassified point and $0.2/2 = 0.1$ for the others; total 0.8, so the new weights are 0.5 and 0.125 (four times). The stump's new weighted error is exactly 0.5: with the optimal α, AdaBoost always reweights so that the last learner is no better than chance, forcing the next learner to find something new.

3. **Leaky folds.** In the nested-CV code, replace every `GroupKFold` by `KFold(3, shuffle=True, random_state=0)` (and drop the `groups` arguments). Compare the outer scores with the group-disjoint test score, and explain which deployment question each split answers.

   Hint

   With row-level folds, two of a subject's three near-identical measurements are often in training when the third is validated, so the forest can recognize the subject rather than the class; inner and outer

scores should rise above the independent group test. Row-level folds answer "new measurements of known subjects"; group folds answer "new subjects."

4. **Bootstrap vs. CV for a stable learner.** Rerun the bootstrap code with `KNeighborsClassifier(n_neighbors=15)`. How do resubstitution, out-of-bootstrap, .632, .632+ and 10-fold CV compare now, and why does the gap between .632 and .632+ shrink?

   Hint

   A 15-NN classifier no longer has zero training error, so the relative overfitting rate $\hat{R}$ is much smaller, $\hat{w}$ is close to 0.632, and the two estimators nearly coincide. All estimates should land within a few points of each other.

5. **Choosing a test.** You have (a) one fixed benchmark split and the predictions of two detectors; (b) one small dataset and enough compute to retrain each model ten times; (c) five algorithms evaluated on twelve public datasets. Which test fits each case, and what are the independent units in each?

   Hint

   (a) McNemar's test (or an exact binomial test) on the paired disagreements; the units are test images, and training-set variability is not captured. (b) The 5×2cv t-test or the corrected resampled t-test; the units are resamples, corrected for overlap. (c) Friedman test with Nemenyi post-hoc (or a Bayesian hierarchical comparison); the units are datasets.

6. **Clusters that mean the wrong thing.** Construct (on paper or with `make_blobs`) a dataset of object images that clusters strongly by camera while the class labels describe object identity. Which validity measures would expose the mismatch, and which would not?

   Hint

   Internal indices such as silhouette would score the camera clusters highly, because they are well separated. External ARI against object labels would be low, while ARI against camera metadata would be high. Stability would also be high, so stability alone cannot reveal that the grouping is semantically wrong.

References

1. S. Theodoridis and K. Koutroumbas, *Pattern Recognition*, 4th ed., Academic Press, 2009. Chapter 4 (decision trees, combining classifiers, boosting), Chapter 10 ("Supervised Learning: The Epilogue") and Chapter 16 ("Cluster Validity"). doi
2. C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006. Sections 1.3 and 3.2 (model selection, bias–variance) and 14.1–14.5 (Bayesian model averaging, committees, boosting, trees, conditional mixtures). official book and PDF

3. S. Geman, E. Bienenstock and R. Doursat, “Neural networks and the bias/variance dilemma,” *Neural Computation*, vol. 4, no. 1, pp. 1–58, 1992. doi
4. L. Breiman, “Bagging predictors,” *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996. doi
5. L. Breiman, “Random forests,” *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001. doi
6. Y. Freund and R. E. Schapire, “A decision-theoretic generalization of on-line learning and an application to boosting,” *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 119–139, 1997. doi
7. J. Friedman, T. Hastie and R. Tibshirani, “Additive logistic regression: A statistical view of boosting (with discussion),” *The Annals of Statistics*, vol. 28, no. 2, 2000. doi
8. J. H. Friedman, “Greedy function approximation: A gradient boosting machine,” *The Annals of Statistics*, vol. 29, no. 5, 2001. doi
9. T. Chen and C. Guestrin, “XGBoost: A scalable tree boosting system,” in *Proc. 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 785–794, 2016. doi
10. D. H. Wolpert, “Stacked generalization,” *Neural Networks*, vol. 5, no. 2, pp. 241–259, 1992. doi
11. B. Efron and R. Tibshirani, “Improvements on cross-validation: The .632+ bootstrap method,” *Journal of the American Statistical Association*, vol. 92, no. 438, pp. 548–560, 1997. doi
12. S. Varma and R. Simon, “Bias in error estimation when using cross-validation for model selection,” *BMC Bioinformatics*, vol. 7, 91, 2006. doi
13. G. C. Cawley and N. L. C. Talbot, “On over-fitting in model selection and subsequent selection bias in performance evaluation,” *Journal of Machine Learning Research*, vol. 11, pp. 2079–2107, 2010. JMLR
14. T. G. Dietterich, “Approximate statistical tests for comparing supervised classification learning algorithms,” *Neural Computation*, vol. 10, no. 7, pp. 1895–1923, 1998. doi
15. C. Nadeau and Y. Bengio, “Inference for the generalization error,” *Machine Learning*, vol. 52, no. 3, pp. 239–281, 2003. doi
16. J. Demšar, “Statistical comparisons of classifiers over multiple data sets,” *Journal of Machine Learning Research*, vol. 7, pp. 1–30, 2006. JMLR
17. A. Benavoli, G. Corani, J. Demšar and M. Zaffalon, “Time for a change: A tutorial for comparing multiple classifiers through Bayesian analysis,” *Journal of Machine Learning Research*, vol. 18, no. 77, pp. 1–36, 2017. JMLR
18. S. Arlot and A. Celisse, “A survey of cross-validation procedures for model selection,” *Statistics Surveys*, vol. 4, pp. 40–79, 2010. doi
19. S. Bates, T. Hastie and R. Tibshirani, “Cross-validation: What does it estimate and how well does it do it?,” *Journal of the American Statistical Association*, vol. 119, no. 546, pp. 1434–1445, 2024. doi
20. S. Raschka, “Model evaluation, model selection, and algorithm selection in machine learning,” arXiv:1811.12808, 2018. arXiv
21. X. Dong, Z. Yu, W. Cao, Y. Shi and Q. Ma, “A survey on ensemble learning,” *Frontiers of Computer Science*, vol. 14, no. 2, pp. 241–258, 2020. doi

22. M. A. Ganaie, M. Hu, A. K. Malik, M. Tanveer and P. N. Suganthan, “Ensemble deep learning: A review,” *Engineering Applications of Artificial Intelligence*, vol. 115, 105151, 2022. doi

23. L. Grinsztajn, E. Oyallon and G. Varoquaux, “Why do tree-based models still outperform deep learning on typical tabular data?,” in *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2022 (arXiv:2207.08815). arXiv

24. B. Lakshminarayanan, A. Pritzel and C. Blundell, “Simple and scalable predictive uncertainty estimation using deep ensembles,” in *Advances in Neural Information Processing Systems (NIPS)*, 2017 (arXiv:1612.01474). arXiv

25. M. Belkin, D. Hsu, S. Ma and S. Mandal, “Reconciling modern machine-learning practice and the classical bias–variance trade-off,” *Proceedings of the National Academy of Sciences*, vol. 116, no. 32, pp. 15849–15854, 2019. doi

26. scikit-learn developers, *scikit-learn User Guide*: ensemble methods, cross-validation, probability calibration and clustering performance evaluation.

27. NumPy developers, “numpy.linalg.lstsq,” API documentation, accessed 2026-11-05. documentation

PR 10 • Recognition Capstone: Rejection, Shift and Learned Features

Build an end-to-end, reproducible recognition project on scikit-learn's digits: frame the task, lock a test set, compare baselines and classifiers with nested validation, analyze errors, then test rejection, unknown classes and distribution shift.

Online edition: https://blog.shyandram.dev/en/learn/pattern-recognition/recognition-capstone/

What you'll learn

- How to turn a vague goal ("recognize handwritten digits") into a written protocol: observation, sampling unit, loss, reject option and what each data split may be used for.
- How to run an honest comparison of the track's classifiers (templates, naive Bayes, k-NN, LDA, logistic regression, kernel SVM, random forest) with one shared pipeline and nested cross-validation.
- How to decide whether a difference between two models is real, using a corrected resampled t-test instead of eyeballing fold means.
- How to do error analysis on development data, read a learning curve, and open the locked test set exactly once.
- How to separate closed-set accuracy from selective classification, unknown-class rejection and robustness to distribution shift, and how all of this carries over to pretrained neural features.

The big picture

The first nine chapters each studied one part of a recognizer: decisions and risk (PR 01), density estimation (PR 02), linear and kernel classifiers (PR 03, PR 04), features (PR 05), templates (PR 06), clustering (PR 07), approximate inference (PR 08) and ensembles and validation (PR 09). A real project needs all of them at once, in the right order, with discipline about which data each decision is allowed to see.

> **Explain like I'm five**
>
> Imagine you are training a friend to sort a big jar of buttons into ten boxes. First you agree on the rules: what counts as a "button", which boxes exist, and what to do with something weird, like a coin. Then you hide a handful of buttons in your pocket and do not show them during practice. Your friend practices on the rest, tries different sorting tricks, and looks closely at every mistake. Only at the very end do you pull the hidden buttons out of your pocket and count how many land in the right box. If you peeked at the hidden buttons during practice, the final count would make your friend look better than they really are.

Why it matters: most disappointing recognition systems do not fail because someone picked the "wrong" classifier. They fail because the evaluation answered a different question from the deployment: the test set leaked into model selection, the split mixed samples from the same person, the system was never asked what to do with something it had not seen, or nobody checked what happens when the camera moves by one pixel. Theodoridis and Koutroumbas treat system design and evaluation as part of pattern recognition itself [1], and Bishop frames the final step as a decision under a loss, with an explicit reject option [2]. This chapter builds one complete project around those ideas.

The project at a glance

Plain version. We follow nine steps. The first seven use only development data. Step 8 opens the locked test set once. Step 9 asks harder questions than "how accurate is it?"

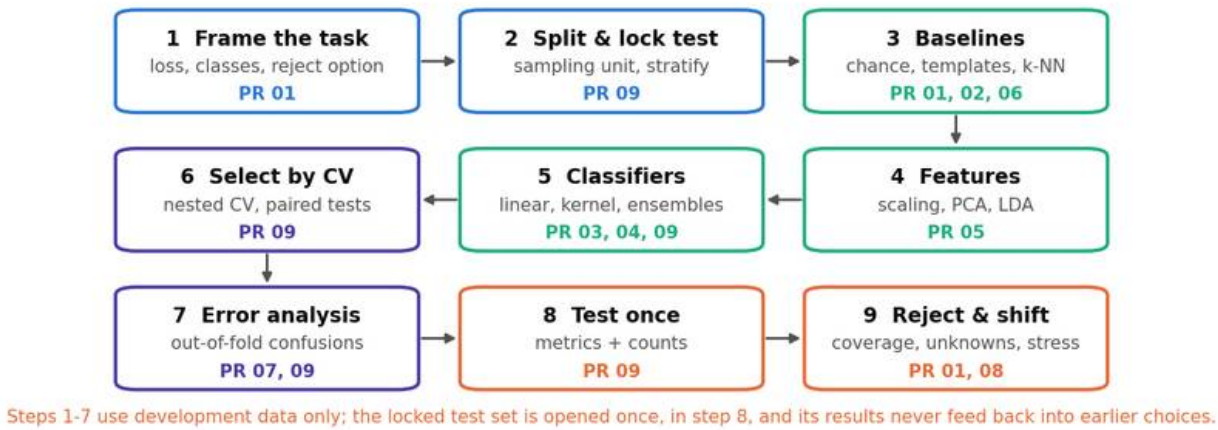


Figure 1 — The capstone workflow and the chapters each step draws on. Steps 1–7 see only development data; the test set is opened once in step 8.

Step	Question it answers	Main tools	Track chapter
1 Frame	What is an observation, a class, a mistake?	loss matrix, reject option	PR 01
2 Split	Which data may inform which decision?	stratified, grouped splits	PR 09
3 Baselines	What does "good" mean here?	chance, class means, k-NN, naive Bayes	PR 01, 02, 06
4 Features	What representation do classifiers see?	scaling, PCA, LDA	PR 05
5 Classifiers	Which model families are plausible?	linear, kernel, ensemble	PR 03, 04, 09
6 Select	Which configuration, and is the gap real?	nested CV, corrected t-test	PR 09
7 Analyze	Why does it fail, and would more data help?	out-of-fold confusions, learning curve	PR 07, 09

Step	Question it answers	Main tools	Track chapter
8 Test	How well does the frozen procedure do?	accuracy with interval, per-class report	PR 09
9 Stress	What about doubt, novelty and shift?	coverage–risk, unknown classes, perturbations	PR 01, 08

All code below runs on a laptop CPU in well under a minute and uses only scikit-learn's bundled data [4]. The snippets in steps 2 to 8 form one script: run them in order in the same Python session.

Step 1 — Frame the task

Plain version. Before touching a model, write down what one example is, what the possible answers are, what a mistake costs, and when the system is allowed to say "I don't know".

A written protocol

A protocol is a short document you write *before* experimenting. It protects you from changing the question after seeing the answer. For any recognition project, answer at least these questions:

1. **Observation.** What is one input? An image, a crop, a video clip, a whole document?
2. **Sampling unit.** Which observations share a source (a person, a camera, a patient, a day)? Observations from the same source are not independent and must stay on the same side of every split.
3. **Label space.** Which classes are known during training? What should happen to inputs from other categories?
4. **Loss.** Are all mistakes equally bad? Is abstaining allowed, and what does it cost?
5. **Data roles.** What may training data, validation data and test data each be used for?
6. **Deployment conditions.** Which devices, writers, lighting or time periods must the evaluation cover?

The decision rule we are aiming for

PR 01 showed that under zero–one loss the Bayes classifier picks the class with the largest posterior probability. Adding a reject option with cost λ_r (where a correct decision costs 0 and an error costs 1) gives the reject-option rule that Bishop presents in his decision-theory section [2]:

$$\hat{y}(x) = \begin{cases} \arg\max_k P(\omega_k \mid x) & \text{if } \max_k P(\omega_k \mid x) \geq 1 - \lambda_r, \\ \text{reject} & \text{otherwise.} \end{cases}$$

Here x is the feature vector, ω_k is class k, $P(\omega_k \mid x)$ is its posterior probability, and $0 \leq \lambda_r \leq 1$ is the cost of rejecting relative to the cost of an error. If $\lambda_r = 0$, rejecting is free and we reject everything that is not certain; if $\lambda_r \geq 1 - 1/K$ for K classes, we never reject. Everything that follows (classifier

choice, validation, threshold setting) is an attempt to approximate this rule with finite data. Keep in mind one limitation: the rule only protects against doubt *among known classes*. A completely new category can still receive a large posterior, which is why step 9 treats unknowns separately.

Our project and its data card

We will recognize handwritten digits using `sklearn.datasets.load_digits` [4]. Writing a short "data card" for it is part of the framing (this is the spirit of datasheets for datasets [24]):

- **Origin.** The data are a copy of the *test* portion of the UCI "Optical Recognition of Handwritten Digits" dataset by Alpaydin and Kaynak [3].
- **Observations.** 1,797 images. Each started as a 32×32 binary bitmap that was divided into non-overlapping 4×4 blocks; each of the resulting 8×8 cells counts the "on" pixels, so values are integers from 0 to 16 [3].
- **Classes.** Ten digits, roughly balanced (174 to 183 images per class).
- **Sources.** According to the dataset description, this portion was written by 13 people, different from the 30 who wrote the UCI training portion [3]. **Writer IDs are not provided.**

The last point shapes every claim we can make. Because we cannot group by writer, our random splits place images by the same 13 people on both sides. The test accuracy we report will therefore estimate performance on *new images from these writers*, not on *new writers*. The UCI training portion, written by different people, would allow a writer-disjoint test, but it is not bundled with scikit-learn and we do not use it here. Stating this limitation openly is part of an honest result.

Step 2 — Split the data and lock the test set

Plain version. Put a quarter of the data in a "locked drawer" right away. Every decision until step 8 is made without it.

```
import numpy as np
import warnings
warnings.filterwarnings("ignore", message=".*within_class_std_dev_.*")
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

digits = load_digits()
X, y = digits.data, digits.target          # 1797 x 64, integer intensities 0..16
print("shape:", X.shape, "range:", X.min(), X.max())
print("class counts:", np.bincount(y))

idx = np.arange(len(y))
dev, test = train_test_split(idx, test_size=0.25, stratify=y, random_state=0)
assert not set(dev) & set(test)
print("development:", len(dev), "test (locked):", len(test))
```

Output: `shape: (1797, 64) range: 0.0 16.0`, class counts between 174 and 183, `development: 1347 test (locked): 450`. (The warnings filter only silences a harmless notice that `NearestCentroid` prints because some border pixels are constant within a class.)

Why stratify? With 450 test images, a random split could easily leave one digit under-represented. Stratification keeps the class proportions the same in both parts. It does *not* fix dependence between observations; only grouping by the sampling unit does that (PR 09).

Leakage: the most common way to fool yourself

Kapoor and Narayanan surveyed reproducibility failures in machine-learning-based science and found data leakage across many fields; they organize it into a taxonomy that includes no clean train–test separation, preprocessing fitted on all data, duplicates across splits, illegitimate features, and test sets that do not match the population of interest [5]. In a recognition project, the most frequent forms are:

- fitting a scaler, PCA or feature selector on all data before splitting;
- choosing hyperparameters, features or thresholds by looking at test scores;
- near-duplicate images (consecutive video frames, augmented copies, the same person) on both sides of a split;
- a test set from a different population than the deployment (for example, one writer group when deployment involves new writers).

The remedy is structural: keep every fitted transformation inside a scikit-learn `Pipeline`, so cross-validation refits it on each training fold, and keep the test indices out of every function call until step 8.

Step 3 — Look at the data and set baselines

Plain version. Before training anything clever, look at examples and compute what simple methods achieve. A fancy model that barely beats a trivial one is not a success.

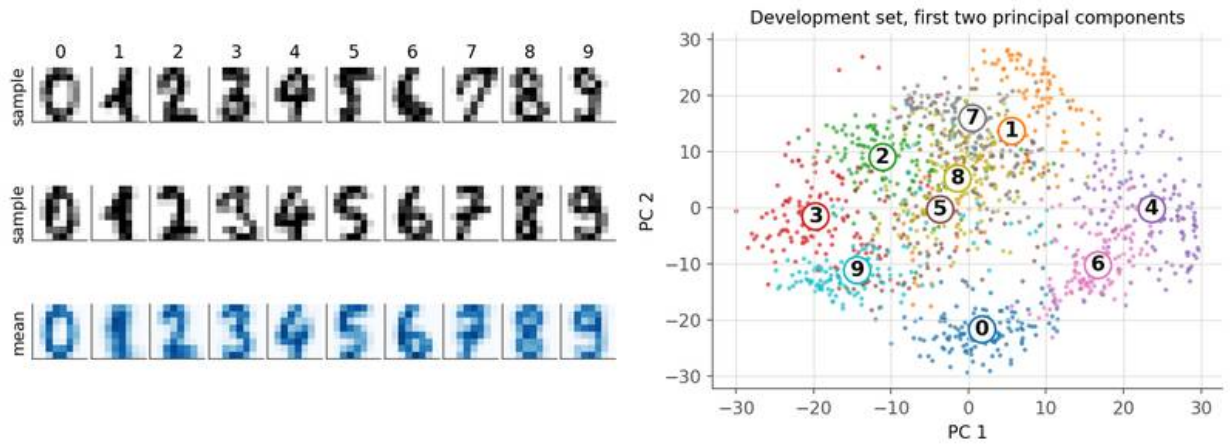


Figure 2 — Left: two development images per class and each class's mean image (the templates of a nearest-class-mean classifier). Right: the development set projected on its first two principal components. Digits 0, 4 and 6 already largely separate; 1, 7, 8, 5 and 9 overlap heavily in this 2-D view.

Figure 2 already suggests the difficulty structure. The class means look like clean digits, so template matching (PR 06) should work reasonably. In two principal components, some classes are well separated and others overlap; PR 05 explained why high-variance directions need not be the most discriminative, so overlap in two components says little about overlap in 64 dimensions.

We use four baselines, each tied to a chapter:

- **Majority class.** Always predict the most common digit. With ten balanced classes this gives about 10% accuracy. It is the floor.
- **Nearest class mean.** Compare an image with each class mean and pick the closest: the simplest template matcher.
- **Gaussian naive Bayes.** A Bayes classifier (PR 01) with class-conditional Gaussian densities whose covariance is diagonal. Pixels are strongly correlated, so the independence assumption is badly wrong here, and we expect it to suffer.
- **k-nearest neighbors.** A nonparametric density-based rule (PR 02): vote among the k closest training images.

Step 4 — Decide on features

Plain version. A classifier only sees numbers. Decide which numbers, how they are scaled, and whether to compress them, and fit any of those transformations on training folds only.

Our raw features are the 64 block counts. Three common transformations are worth comparing:

Standardization. Subtract each feature's training mean and divide by its standard deviation:

$$z_j = \frac{x_j - \bar{x}_j}{s_j},$$

where x_j is pixel j, $\bar{x}_j$ and s_j are its mean and standard deviation over the training fold, and z_j is the standardized value. This matters for distance-based and gradient-based methods when features have very different units. Here all features already share the same unit (a count from 0 to 16). Worse, the border pixels are almost always 0, so their s_j is tiny: standardizing them *amplifies* rare stray strokes into large values. We will see that this choice measurably hurts the kernel SVM.

Principal component analysis. PCA projects onto the directions of largest variance (PR 05). With d' components the representation is $u = W^\top(x - \bar{x})$, where W holds the top d' eigenvectors of the training covariance. It removes noise directions and speeds up kernels, but variance is not the same as class information.

Linear discriminant analysis. LDA (PR 03, PR 05) finds at most $K - 1 = 9$ directions that maximize between-class over within-class scatter. We use it as a classifier with shrinkage, which regularizes the within-class covariance estimate.

> **Features are hyperparameters**
>
> Whether to scale, how many PCA components to keep, or which hand-made features to add are choices made with data. Treat them exactly like a regularization constant: put them in the pipeline and let validation choose them. Fitting them once on all the data before cross-validation is leakage.

For real images, this is also where you would add engineered features (HOG, LBP, moments, see the feature extraction chapter) or pretrained embeddings. At 8×8 resolution, the raw counts are a reasonable starting representation.

Step 5 — Compare candidate classifiers

Plain version. Run every candidate through the *same* cross-validation splits of the development set and compare their score distributions, not just their best fold.

```
from sklearn.model_selection import RepeatedStratifiedKFold, cross_val_score
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.dummy import DummyClassifier
from sklearn.neighbors import NearestCentroid, KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier

candidates = {
    "majority class (PR 01)": DummyClassifier(strategy="most_frequent"),
    "nearest class mean (PR 06)": NearestCentroid(),
    "Gaussian naive Bayes (PR 01)": GaussianNB(),
    "k-NN, k=3 (PR 02)": KNeighborsClassifier(n_neighbors=3),
    "LDA, shrinkage (PR 03/05)": LinearDiscriminantAnalysis(
        solver="lsqr", shrinkage="auto"),
```

```
    "logistic regression (PR 03)": make_pipeline(
        StandardScaler(), LogisticRegression(C=1.0, max_iter=3000)),
    "PCA-30 + RBF SVM (PR 04/05)": make_pipeline(
        StandardScaler(), PCA(n_components=30, random_state=0),
        SVC(C=10, gamma="scale")),
    "random forest (PR 09)": RandomForestClassifier(
        n_estimators=300, random_state=0, n_jobs=1),
}
cv = RepeatedStratifiedKFold(n_splits=5, n_repeats=2, random_state=0)
cv_scores = {}
for name, model in candidates.items():
    s = cross_val_score(model, X[dev], y[dev], cv=cv, scoring="accuracy")
    cv_scores[name] = s
    print(f"{name:30s} {s.mean():.3f} +/- {s.std():.3f}")
```

Output (mean $\pm$ standard deviation over the 10 folds):

Candidate	Mean CV accuracy	Std
majority class	0.100	0.000
Gaussian naive Bayes	0.827	0.017
nearest class mean	0.897	0.011
LDA with shrinkage	0.951	0.007
logistic regression (scaled)	0.966	0.012
random forest, 300 trees	0.975	0.010
scaled PCA-30 + RBF SVM	0.976	0.010
3-nearest neighbors	0.985	0.007

Several lessons are already visible:

- **The ranking of simple models follows their assumptions.** Naive Bayes, whose independence assumption is violated by correlated neighboring pixels, is the weakest real model. A single template per class (nearest mean) cannot capture different writing styles of the same digit. LDA, which models a shared full covariance, does much better.
- **The best untuned model is the simplest nonparametric one.** 3-NN beats every untuned parametric model. With 1,347 small images and only 13 writers, many test images have a near twin in the training folds, which is exactly the situation where nearest neighbors shine. This also reminds us that our split measures "new images from known writers".
- **Defaults are not a fair comparison.** The SVM and the logistic regression ran with a standardization step and default-ish settings. Before concluding anything, the strong candidates deserve tuning, under a protocol that does not bias the comparison.

Step 6 — Tune and select without fooling yourself

Plain version. Tuning is part of the method. To estimate how good "SVM *with* tuning" is, you must repeat the whole tuning inside each outer fold. Then ask whether the remaining gap to the next-best model is bigger than the noise.

Nested cross-validation

The inner loop chooses hyperparameters; the outer loop scores the *whole procedure*. If you report the best inner score directly, you report the maximum of many noisy numbers, which is optimistically biased. Varma and Simon showed this bias empirically for cross-validated model selection [6], and Cawley and Talbot showed that over-fitting the model-selection criterion can produce performance differences as large as the differences between learning algorithms [7]. Raschka's tutorial review collects the practical recipes [9].

```
from sklearn.model_selection import GridSearchCV, StratifiedKFold
from sklearn.pipeline import Pipeline

svm = Pipeline([("scale", StandardScaler()), ("pca", PCA(random_state=0)),
                ("svc", SVC())])
grid = {"scale": [StandardScaler(), "passthrough"],
        "pca": [PCA(30, random_state=0), "passthrough"],
        "svc__C": [1, 10],
        "svc__gamma": ["scale", 0.001]}
inner = StratifiedKFold(5, shuffle=True, random_state=1)
outer = StratifiedKFold(5, shuffle=True, random_state=2)
search = GridSearchCV(svm, grid, cv=inner, scoring="accuracy")
nested = cross_val_score(search, X[dev], y[dev], cv=outer)
print("nested-CV accuracy of the tuning procedure:",
      nested.round(3), "mean", nested.mean().round(3))
search.fit(X[dev], y[dev])
print("chosen on all development data:", search.best_params_)
```

Output: outer-fold accuracies `[0.993 0.993 0.985 0.985 0.993]`, mean `0.99`. The configuration chosen on all development data is **no standardization, PCA with 30 components,** $C = 10$**,** $\gamma = 0.001$.

Two observations. First, the grid includes the preprocessing decision itself (`"passthrough"` means "skip this step"), and validation rejected standardization, confirming the argument of step 4. Second, the nested estimate (0.990) describes the *procedure* "run this grid search, then refit", which is what we will deploy. It is not a property of one particular hyperparameter setting.

Is the winner really better?

The tuned SVM (0.990 nested) and 3-NN (0.985) differ by half a percentage point. Is that real? A naive paired t-test on fold scores treats the folds as independent samples, but the training sets of different folds overlap heavily, so the fold scores are positively correlated and the naive test is too confident. Nadeau and Bengio proposed a corrected variance for resampled designs [8]:

$$t = \frac{\bar{d}}{\sqrt{\left(\frac{1}{k} + \frac{n_{\text{te}}}{n_{\text{tr}}}\right)\hat{\sigma}_d^2}},$$

where d_i is the score difference between the two models on resample i, $\bar{d}$ and $\hat{\sigma}_{\bar{d}}^2$ are the mean and sample variance of the k differences, and n_{tr} and n_{te} are the training and test sizes of each resample. The extra term $n_{\text{te}}/n_{\text{tr}}$ inflates the variance to account for overlap. The statistic is compared with a t distribution with $k-1$ degrees of freedom. PR 09 derives this and discusses alternatives such as McNemar's test.

```
from scipy import stats

def corrected_t(a, b, n_train, n_test):
    """Nadeau-Bengio corrected resampled t-test on paired fold scores."""
    d = np.asarray(a) - np.asarray(b)
    k = len(d)
    var = d.var(ddof=1) * (1 / k + n_test / n_train)
    t = d.mean() / np.sqrt(var)
    return t, 2 * stats.t.sf(abs(t), df=k - 1)

n_test_fold = len(dev) // 5
tuned = cross_val_score(search.best_estimator_, X[dev], y[dev], cv=cv)
t, p = corrected_t(tuned,
                   cv_scores["k-NN, k=3 (PR 02)"],
                   n_train=len(dev) - n_test_fold, n_test=n_test_fold)
print(f"tuned SVM vs k-NN: corrected t = {t:.2f}, p = {p:.3f}")
```

Output: `tuned SVM vs k-NN: corrected t = 1.03, p = 0.331`.

Treat this output as an **exploratory illustration of variance correction**, not a confirmatory significance test. The SVM configuration was selected using all development labels before these repeated folds were made; the correction addresses overlapping folds, not that selection bias [7, 8]. A nonsignificant number also does not prove equivalence. For a fresh comparison, run `cross_val_score(search, ...)` so tuning is repeated inside every evaluation fold, evaluate the predeclared comparator on the same outer splits, and state the remaining approximation in the statistical test. Here we retain 3-NN as a strong alternative and proceed with the tuned SVM based on the earlier nested evaluation and practical storage/prediction requirements, without claiming a statistically established tie.

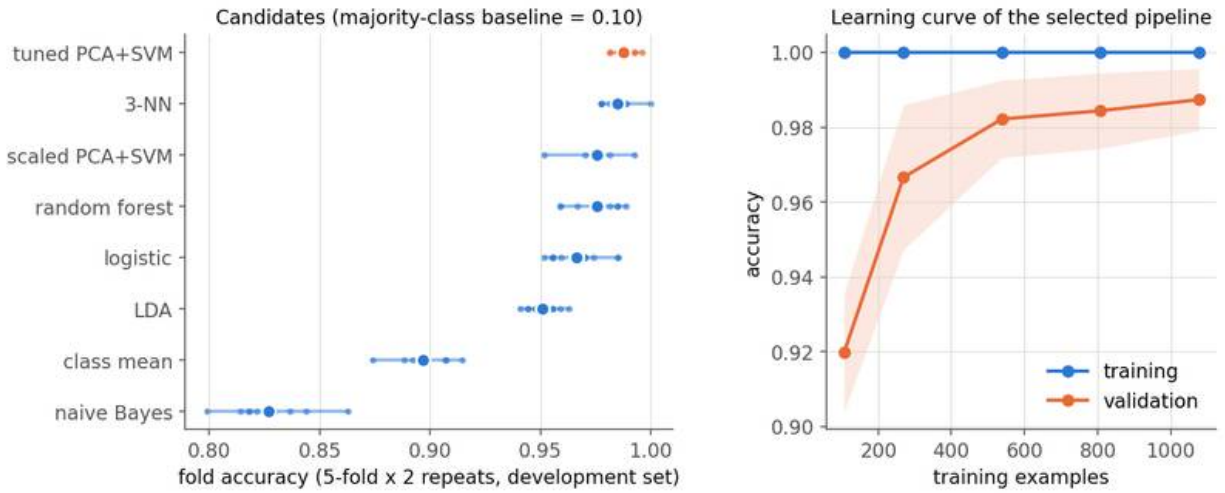


Figure 3 — Left: accuracy on each of the 10 development folds (small dots), with the mean (large dot) and range (bar). The tuned pipeline (orange) uses the configuration chosen by the search. Right: learning curve of the selected pipeline; the shaded band is ±1 standard deviation across folds.

Step 7 — Error analysis on development data

Plain version. Look at the mistakes, but look at *development* mistakes. Once you start redesigning the system because of test errors, the test set stops being a test.

Out-of-fold predictions

`cross_val_predict` returns, for each development image, the prediction of a model that did not train on it. These predictions are useful for development error analysis. However, `chosen` was selected using all development labels: each image was excluded from coefficient fitting, not from hyperparameter selection. Do not report this error count as an unbiased evaluation of tuning; pass the full search into the outer prediction loop when that is the goal [7].

```
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import confusion_matrix

chosen = search.best_estimator_
oof = cross_val_predict(chosen, X[dev], y[dev],
                        cv=StratifiedKFold(5, shuffle=True, random_state=3))
cm_dev = confusion_matrix(y[dev], oof)
off = cm_dev.copy()
np.fill_diagonal(off, 0)
order = np.argsort(off.ravel())[::-1][:5]
print("out-of-fold errors:", int(off.sum()), "of", len(dev))
for t_, p_ in zip(*np.unravel_index(order, off.shape)):
    print(f"  true {t_} -> predicted {p_}: {off[t_, p_]}")
```

```
dev_errors = dev[oof != y[dev]]
print("first misclassified dataset indices:", dev_errors[:8])
```

Output: 17 errors out of 1,347. The most frequent confusions are 9 → 8 (3 times), 5 → 9 (3), 8 → 1 (2), then single errors such as 9 → 5 and 6 → 8.

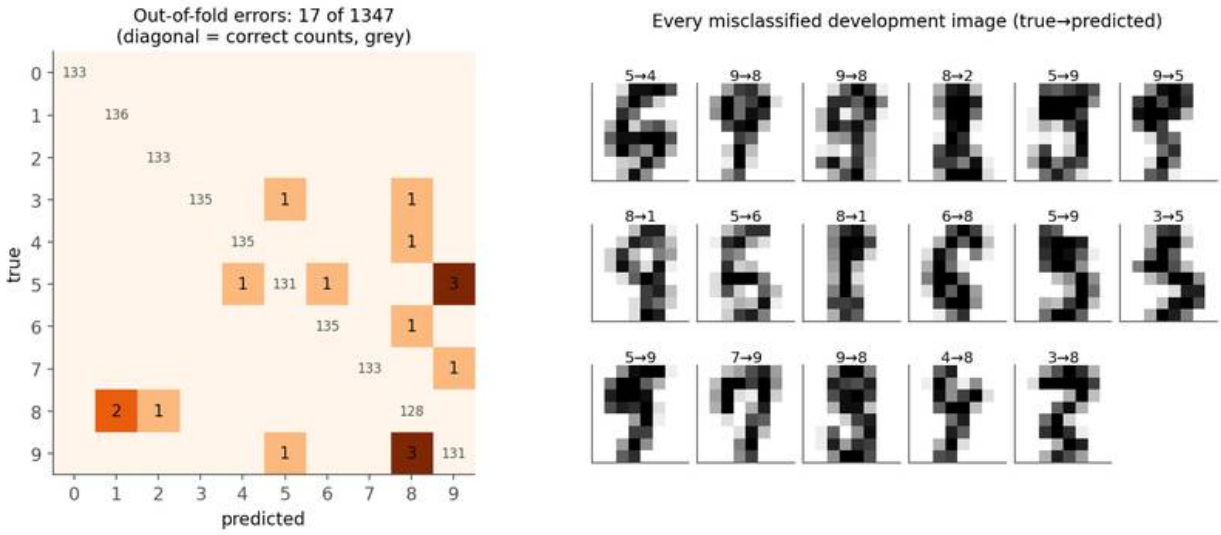


Figure 4 — Out-of-fold error analysis of the selected pipeline. Left: off-diagonal counts (correct counts shown small on the diagonal). Right: every misclassified development image, labeled true→predicted.

Reading Figure 4 is where domain knowledge enters:

- **Systematic confusions.** The errors are concentrated among digits with closed upper loops (5, 8, 9) and thin vertical strokes (1, 8). At 8 × 8 resolution, the gap that distinguishes a 9 from an 8 may be a single cell.
- **Label or quality issues.** Some misclassified images are ambiguous even to a human. In a larger project you would review such cases and decide whether they are labeling errors (to fix), genuinely ambiguous inputs (candidates for rejection, step 9), or ordinary failures (to address with features or data).
- **Grouping errors.** With hundreds of errors you cannot inspect them one by one. A common trick is to cluster the misclassified examples in a feature space (PR 07) and inspect a few members of each cluster; clusters often correspond to one failure mode, such as a writing style or an acquisition problem.

A good error analysis ends with *hypotheses*, each paired with an experiment that can be run on development data: "a one-pixel shift invariance might help 5/9 confusions; try augmenting training folds with shifted copies and re-run the nested comparison."

Would more data help? The learning curve

```
from sklearn.model_selection import learning_curve

sizes, tr, va = learning_curve(
    chosen, X[dev], y[dev], cv=StratifiedKFold(5, shuffle=True, random_state=3),
    train_sizes=[0.1, 0.25, 0.5, 0.75, 1.0])
for n, a, b in zip(sizes, tr.mean(1), va.mean(1)):
    print(f"n={n:4d}  train={a:.3f}  validation={b:.3f}")
```

Training images	Training accuracy	Validation accuracy
107	1.000	0.920
269	1.000	0.967
538	1.000	0.982
807	1.000	0.984
1,077	1.000	0.987

The training accuracy is perfect at every size, and the validation accuracy is still rising slowly at the right edge (Figure 3, right). In bias–variance terms (PR 09) the remaining error is mostly *variance*: the model is flexible enough, and more (or more varied) training data would likely help a little. Had both curves plateaued together at a low value, the advice would be the opposite: a more expressive model or better features.

Step 8 — Open the test set, once

Plain version. Freeze every choice, run the final pipeline on the locked test images, and report the result with its uncertainty and its limits.

```
from sklearn.metrics import (accuracy_score, balanced_accuracy_score,
                             classification_report)

final = search.best_estimator_            # already refit on all of dev
pred = final.predict(X[test])
print("TEST accuracy:", round(accuracy_score(y[test], pred), 4),
      "balanced:", round(balanced_accuracy_score(y[test], pred), 4),
      "errors:", int((pred != y[test]).sum()), "of", len(test))
cm = confusion_matrix(y[test], pred)
print(cm)
print(classification_report(y[test], pred, digits=3))

k, n, z = int((pred == y[test]).sum()), len(test), 1.96   # Wilson interval
phat = k / n
centre = (phat + z**2 / (2 * n)) / (1 + z**2 / n)
half = z * np.sqrt(phat * (1 - phat) / n + z**2 / (4 * n**2)) / (1 + z**2 / n)
print(f"95% Wilson interval for accuracy: [{centre - half:.3f}, {centre + half:.3f}]")
```

Output: test accuracy **0.9933** (3 errors out of 450: one each of 3 → 7, 8 → 1 and 9 → 5), balanced accuracy 0.9933, per-class recall between 0.977 and 1.000, and a 95% Wilson interval of **[0.981, 0.998]**.

How to read and report this:

- **Report counts, not just percentages.** "3 errors out of 450" tells the reader immediately how much the third decimal can be trusted.
- **Report an interval.** For an accuracy close to 1, the usual normal approximation $\hat{p} \pm 1.96\sqrt{\hat{p}(1-\hat{p})/n}$ can exceed 1 and is unreliable; the Wilson score interval used above stays inside $[0, 1]$ and behaves better near the boundary. With n test examples, k correct and $\hat{p} = k/n$, its center is $(\hat{p} + z^2/2n)/(1 + z^2/n)$ with $z = 1.96$ for 95% coverage.
- **Compare with the development estimate.** The test result (0.993) is consistent with the nested estimate (0.990). If it had been much lower, the likely causes would be leakage in development or a test population that differs from development, and either deserves investigation, but a fix would then need a *new* untouched test set.
- **State the scope.** This is accuracy on new images from the same 13 writers (step 1). It is not evidence about new writers, scanners or preprocessing.

The protocol note in the code matters: `search.best_estimator_` was refit on all development data once, before testing. Refitting again on development plus test and then reporting the test score would be meaningless. For deployment you *may* refit the frozen procedure on all labeled data, but the reported number must come from the run above.

Step 9 — Beyond closed-set accuracy

A recognizer that is 99% accurate on a balanced test of known digits can still be useless if it confidently labels a letter as a digit, or collapses when the scanner shifts the image. This step tests three questions the closed-set number cannot answer.

Four tasks that are easy to confuse

Task	What the system must do	What the evaluation must contain
Closed-set classification	choose among trained classes	independent examples of known classes
Selective classification	abstain on difficult known examples	coverage and error on accepted known examples
Open-set recognition	classify known classes and reject unknown categories	held-out categories as well as known examples
Anomaly detection	flag deviations from a reference population	a defined normal population and relevant deviations

A low-confidence known example is not necessarily an unknown category; an unfamiliar category

can receive a high confidence; and a valid known object photographed badly may be rejected. These tasks need different data and different metrics.

Selective classification: coverage and risk

Plain version. Let the classifier skip the inputs it is least sure about and measure two things: how many inputs it still answers, and how often those answers are wrong.

For a confidence score $q(x)$ and an acceptance threshold τ, over N known-class test examples with predictions $\hat{y}_n$ and labels y_n:

$$\text{coverage}(\tau) = \frac{1}{N}\sum_{n=1}^{N} \mathbf{1}[q(x_n) \geq \tau],$$

$$\text{selective risk}(\tau) = \frac{\sum_n \mathbf{1}[q(x_n) \geq \tau]\,\mathbf{1}[\hat{y}_n \neq y_n]}{\sum_n \mathbf{1}[q(x_n) \geq \tau]}.$$

Here $\mathbf{1}[\cdot]$ is 1 when its condition holds and 0 otherwise. Selective risk is undefined when nothing is accepted. Plotting risk against coverage while sweeping τ gives the risk–coverage curve; Geifman and El-Yaniv showed how to choose a threshold for deep networks that guarantees a desired risk with high probability [11]. A useful threshold must be chosen on validation data and then only *reported* on test data.

Open-set recognition: held-out categories

Plain version. Pretend some digits do not exist during training, then check whether the system rejects them when they show up.

Scheirer and colleagues formalized open-set recognition: a classifier must limit the "open space" it labels as known, because a model trained on K classes will otherwise assign every input on Earth to one of them [13]. The simplest baseline score is the maximum softmax (or class) probability, proposed by Hendrycks and Gimpel for detecting misclassified and out-of-distribution examples [12].

We redo the experiment with a reduced label space. Digits 0–5 are **known**; digits 6–7 are **proxy unknowns** that development may look at; digits 8–9 are **test unknowns** reserved for the final evaluation. Validation chooses between two predefined pipelines, and the rejection threshold is the 5th percentile of validation confidences, which targets roughly 95% acceptance of known digits. All choices are frozen before the final block. This script is self-contained.

This is a separate six-class experiment. Its variables live inside `open_set_demo()` so the original ten-class model, development indices and locked test indices remain unchanged for the later stress tests.

```
def open_set_demo():
    import numpy as np
```

```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.metrics import balanced_accuracy_score, confusion_matrix, roc_auc_score

digits = load_digits()
X, y = digits.data, digits.target
known = np.flatnonzero(y < 6)
train, rest = train_test_split(known, test_size=0.4,
                               stratify=y[known], random_state=10)
valid, test = train_test_split(rest, test_size=0.5,
                               stratify=y[rest], random_state=10)
proxy_unknown = np.flatnonzero((y == 6) | (y == 7))
test_unknown = np.flatnonzero(y >= 8)
assert not set(train) & (set(valid) | set(test))
assert not set(proxy_unknown) & set(test_unknown)

candidates = {
    "linear": make_pipeline(
        StandardScaler(), LogisticRegression(C=1.0, max_iter=1500)),
    "PCA + RBF": make_pipeline(
        StandardScaler(), PCA(n_components=24, random_state=10),
        SVC(C=4.0, gamma=0.02, probability=True, random_state=10)),
}
scores = {}
for name, model in candidates.items():
    model.fit(X[train], y[train])
    validation_p = model.predict_proba(X[valid])
    validation_pred = model.classes_[validation_p.argmax(axis=1)]
    scores[name] = balanced_accuracy_score(y[valid], validation_pred)
name = max(scores, key=scores.get)
model = candidates[name]
threshold = np.quantile(model.predict_proba(X[valid]).max(axis=1), 0.05)
proxy_conf = model.predict_proba(X[proxy_unknown]).max(axis=1)
print("Validation candidates:", scores, "selected:", name)
print("Fixed threshold:", threshold)
print("Proxy-unknown acceptance:", np.mean(proxy_conf >= threshold))

# Final evaluation begins here; do not revise choices from these results.
known_p = model.predict_proba(X[test])
known_conf = known_p.max(axis=1)
known_pred = model.classes_[known_p.argmax(axis=1)]
accept = known_conf >= threshold
unknown_conf = model.predict_proba(X[test_unknown]).max(axis=1)
print("Known test balanced accuracy:",
      balanced_accuracy_score(y[test], known_pred))
print("Known coverage:", accept.mean())
print("Accepted known accuracy:",
      np.mean(known_pred[accept] == y[test][accept]) if accept.any() else np.nan)
print("Unknown-test acceptance:", np.mean(unknown_conf >= threshold))
is_unknown = np.r_[np.zeros(len(test)), np.ones(len(test_unknown))]
unknown_score = 1 - np.r_[known_conf, unknown_conf]
```

```
    print("Unknown detection AUROC:", roc_auc_score(is_unknown, unknown_score))
    print("Known-class confusion matrix:\n", confusion_matrix(y[test], known_pred))

open_set_demo()
```

Output, in the order printed: validation balanced accuracy 0.982 (linear) versus 0.995 (PCA + RBF), so the RBF pipeline is selected; threshold 0.741; **proxy-unknown acceptance 0.083**; known test balanced accuracy 0.981; known coverage 0.949; accepted known accuracy 0.995; **unknown-test acceptance 0.350**; unknown-detection AUROC 0.944.

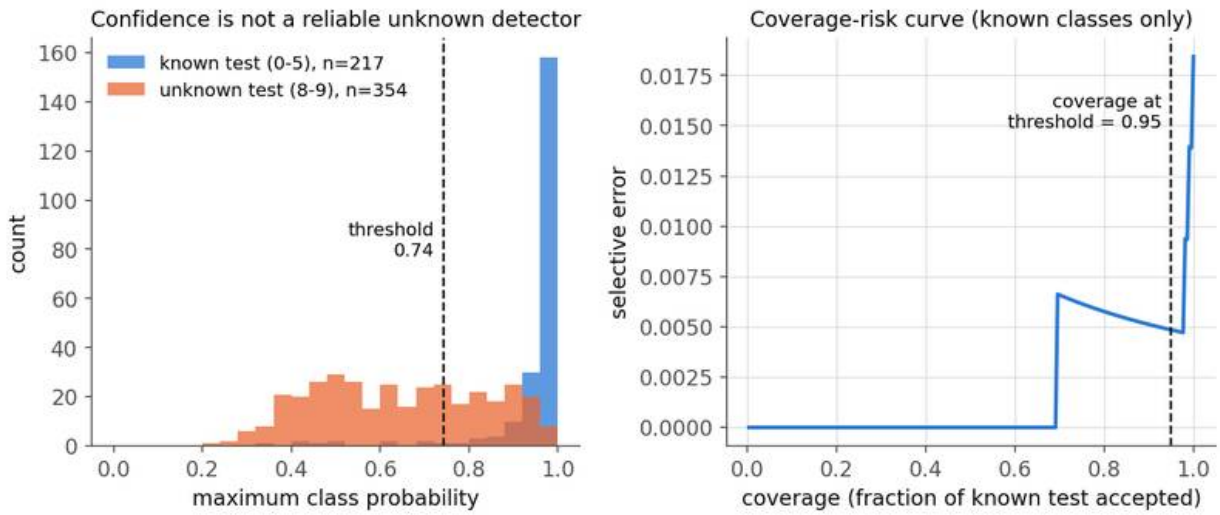


Figure 5 — Left: maximum class probability for known test digits (0–5) and unknown test digits (8–9), with the threshold fixed on validation data. Right: risk–coverage curve for known test digits; the dashed line marks the coverage of the fixed threshold.

The numbers carry the main lesson of open-set evaluation:

- On known digits, rejection works as intended: 95% coverage and 99.5% accuracy on what is accepted.
- The proxy unknowns (6 and 7) were rejected 92% of the time, which might have suggested that confidence thresholding "solves" unknowns.
- The real test unknowns (8 and 9) were **accepted 35% of the time**. An 8 looks like a 3 or a 5 with extra strokes, and a 9 looks like a 5 or a 3; the classifier confidently maps them onto those known digits.
- AUROC of 0.944 summarizes ranking over these particular unknowns. It does not fix an operating threshold and does not describe other kinds of novelty.

Performance on one set of proxy unknowns does not transfer to other unknowns. Report which categories were held out, and never claim rejection of "all unfamiliar inputs". Vaze and colleagues found that a stronger closed-set classifier tends to be a better open-set recognizer, with closed-set and open-set performance strongly correlated across methods [16], so improving the base model

and using better scores (feature distances, energy or logit-based scores) are complementary.

Other families of scores model the reference population directly: a one-class SVM estimates a region that contains most of the training data [17], and an Isolation Forest flags points that random axis-aligned splits isolate quickly [18]. An "outlier" may be an acquisition error, a rare valid case or a new class; statistical unusualness does not tell you which.

Distribution shift and stress tests

Plain version. The world changes after you deploy. Ask "what if" questions with controlled perturbations and, better, with real data from new conditions.

Let $p_{\text{train}}(x, y)$ and $p_{\text{deploy}}(x, y)$ be the joint distributions of features and labels during training and deployment. Moreno-Torres and colleagues organized the vocabulary of dataset shift in classification [19]. Three cases matter most:

- **Prior (label) shift:** class frequencies $p(y)$ change, while $p(x \mid y)$ stays the same. Example: one digit becomes much more common in a new form layout.
- **Covariate shift:** $p(x)$ changes, while $p(y \mid x)$ stays the same. Example: a new scanner makes strokes thicker.
- **Concept shift:** $p(y \mid x)$ itself changes. Example: a new convention for writing 7 with a bar.

These are modeling assumptions, not diagnoses you can read off the data's name. A camera change can alter class-conditional features; adding a new class breaks the label space entirely.

We cannot collect new handwriting here, but we can probe sensitivity with controlled perturbations of the *test* images. The model is not retrained; the perturbations were decided before looking at their results.

```
rng = np.random.default_rng(0)
def shift_right(X, k=1):
    imgs = X.reshape(-1, 8, 8)
    out = np.zeros_like(imgs)
    out[:, :, k:] = imgs[:, :, :-k]
    return out.reshape(len(X), 64)

conditions = {
    "clean": X[test],
    "noise sd=2": np.clip(X[test] + rng.normal(0, 2, X[test].shape), 0, 16),
    "noise sd=4": np.clip(X[test] + rng.normal(0, 4, X[test].shape), 0, 16),
    "shift 1 px right": shift_right(X[test], 1),
    "contrast x0.5": X[test] * 0.5,
}
for name, Xc in conditions.items():
    print(f"{name:18s} accuracy = {accuracy_score(y[test], final.predict(Xc)):.3f}")
```

(Run this after the step 8 block, not after the open-set script, because it uses the 10-class `X`, `test` and `final`.)

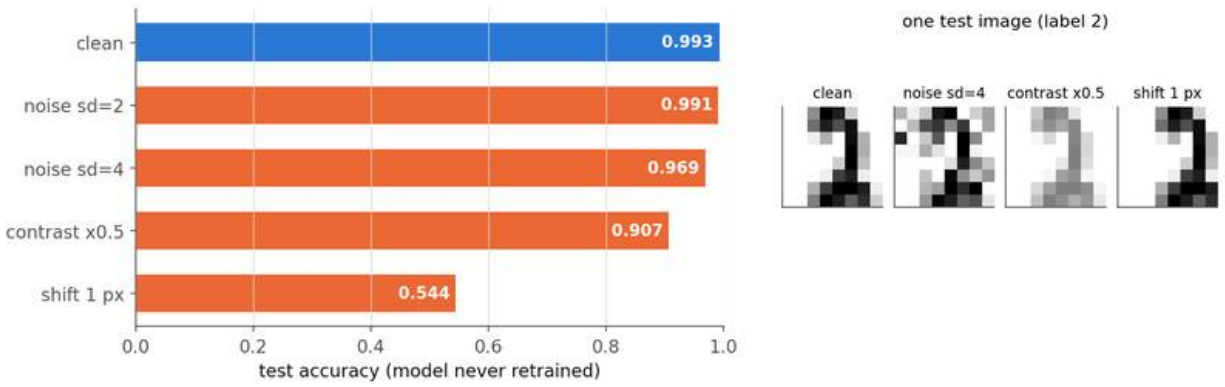


Figure 6 — Test accuracy of the frozen pipeline under perturbations it never saw in training, and one test image under each condition. Intensities stay in the dataset's 0–16 range.

Condition	Test accuracy
clean	0.993
additive noise, sd = 2	0.991
additive noise, sd = 4	0.969
contrast halved	0.907
shifted right by one cell	**0.544**

Additive noise barely matters. Halving the contrast costs about 9 points, because the RBF kernel compares raw intensities and γ was tuned for full-contrast strokes. A **one-cell translation** cuts accuracy almost in half: a pixel-vector classifier has no notion that a shifted 3 is still a 3. One cell corresponds to a 4-pixel block of the original 32×32 bitmap, an eighth of the image width, so this is a realistic misalignment for a sloppy crop. This is the classic argument for building translation tolerance into the model, by augmentation or by convolution and pooling, which LeCun and colleagues demonstrated for document recognition [10] (see ML 03).

Synthetic perturbations are a cheap first check, not a substitute for real new-condition data. Benchmarks such as WILDS collect real distribution shifts across hospitals, cameras and regions and show sizable drops from in-distribution to out-of-distribution performance [21]; Recht and colleagues rebuilt the CIFAR-10 and ImageNet test sets following the original procedures and observed accuracy drops for many models, even without any deliberate shift [20]. Report results per meaningful condition (writer, device, site, time), not just pooled.

Probabilities that mean what they say

If downstream users act on confidence values, they must be *calibrated*: among predictions made with confidence 0.8, about 80% should be correct. Guo and colleagues showed that modern deep networks are often overconfident and that a single scalar *temperature* fitted on validation data is a strong fix [22]. For SVMs, `SVC(probability=True)` fits Platt scaling internally with cross-validation; `CalibratedClassifierCV` gives explicit control. Calibration must be fitted on data not used for training the base model, and it does not repair distribution shift.

Bridge to learned and foundation features

Plain version. Swap the hand-made features for a pretrained network's features. Everything else in this chapter stays the same.

Replace the feature map with a pretrained encoder:

$$x = \phi_{\text{pretrained}}(I), \qquad \hat{y} = g_\theta(x),$$

where I is the raw image, $\phi_{\text{pretrained}}$ is a frozen network (for example a self-supervised vision transformer such as DINOv2 [25]), x is its embedding, and g_θ is a classifier with parameters θ trained on your labels. A **linear probe** (logistic regression on frozen embeddings) is the standard baseline; compare it with k-NN on embeddings, a regularized nonlinear head and fine-tuning, holding splits and metrics fixed. ML 06 explains how such features are learned and ML 10 how they are adapted.

The protocol extends naturally:

- record the checkpoint, input resizing, normalization, pooling and embedding dimension;
- fit downstream preprocessing (scaling, PCA) only on training embeddings;
- count the choice of checkpoint, prompt or layer as model selection, made on validation data;
- if evaluation images could have appeared in the pretraining data, state what evidence you have about overlap;
- distance-based unknown scores (k-NN distance to training embeddings) often work much better on good embeddings than on raw pixels, but they still need held-out unknown categories to evaluate.

Representation learning changes *how measurements are obtained*. It does not change minimum-risk decisions, honest validation or the separation of known-class accuracy from rejection and shift, which is the framework of this track [1], [2].

What to submit

A finished recognition project should include, at minimum:

1. **Task definition:** observation, label space, loss, reject policy.
2. **Data card:** provenance, licensing, sampling unit, class counts, known limitations (here: 13 writers, no writer IDs) [24].
3. **Split description:** what each split was used for, the unit of splitting, seeds.
4. **Baselines and candidates:** all models tried, not just the winner, with development scores and their variability.
5. **Selection evidence:** nested estimate of the chosen procedure and a statistical comparison with the runner-up.
6. **Error analysis:** confusion structure, examples, hypotheses, learning curve.
7. **Final test result:** counts, interval, per-class results, scope statement.

8. **Stress results:** rejection (coverage and risk), held-out unknown categories, shift or perturbation tests, calibration if probabilities are used.
9. **Reproducibility:** environment and library versions, the full pipeline object, class ordering, runtime and memory if they matter.

Model cards formalize the deployment-facing part of this list (intended use, evaluation conditions, performance across relevant groups, caveats) [23]. If the original test set informed any redesign, the revised system needs a new protected test.

Modern view

The capstone workflow is old in its principles and still evolving in practice. The following reviews and studies are good entry points.

Raschka, "Model evaluation, model selection, and algorithm selection in machine learning" (2018) [9]. A long tutorial review that organizes evaluation into three distinct goals: estimating generalization performance, selecting hyperparameters and models, and comparing algorithms. It covers holdout, bootstrap and cross-validation, nested cross-validation and statistical tests (McNemar, 5×2cv and related tests), with practical recommendations by dataset size. It is the closest single reference to steps 2 and 6 of this chapter.

Kapoor and Narayanan, "Leakage and the reproducibility crisis in machine-learning-based science" (*Patterns*, 2023) [5]. Collects published leakage failures across many scientific fields and proposes a taxonomy of leakage types together with "model info sheets" that ask authors to justify the absence of each type. Its main takeaway is that leakage is common, often invisible in the reported numbers, and best prevented by protocol rather than detected afterwards.

Geng, Huang and Chen, "Recent advances in open set recognition: a survey" (TPAMI, 2021) [14]. Distinguishes known known, known unknown, unknown known and unknown unknown classes and organizes open-set methods into discriminative (traditional ML and deep network based) and generative (instance and non-instance generation) families. It also relates open-set recognition to zero-shot, one-shot and classification-with-reject settings and reviews the datasets and evaluation criteria in use.

Yang, Zhou, Li and Liu, "Generalized out-of-distribution detection: a survey" (IJCV, 2024) [15]. Unifies anomaly detection, novelty detection, open-set recognition, out-of-distribution detection and outlier detection in one framework, distinguishing them by whether shift is semantic or covariate and by what the in-distribution task is. This is the vocabulary to use when deciding which of the "four tasks" table rows your project belongs to.

Vaze, Han, Vedaldi and Zisserman, "Open-set recognition: a good closed-set classifier is all you need?" (ICLR, 2022) [16]. Shows a strong correlation between closed-set accuracy and open-set performance and that a well-trained baseline with a maximum-logit score is competitive with specialized methods; it also proposes the "Semantic Shift Benchmark" to separate semantic novelty from low-level shift.

Koh et al., "WILDS" (ICML, 2021) [21] and Recht et al. (ICML, 2019) [20]. Two complementary

views of evaluation under shift: WILDS curates real deployment shifts across domains, while Recht et al. show that even a careful re-collection of a test set "from the same distribution" can lower accuracy. Together they argue for reporting results per condition and treating a single i.i.d. test score as an upper bound.

Guo et al., "On calibration of modern neural networks" (ICML, 2017) [22] and **Geifman and El-Yaniv (NeurIPS, 2017) [11]** are the starting points for confidence quality and selective prediction in deep models.

State of the art (2023–2026). In practice, recognition projects increasingly begin from frozen or lightly fine-tuned foundation features (for example DINOv2 [25]) with a linear or k-NN head, and spend the saved effort on data quality, evaluation design and documentation. Out-of-distribution evaluation has moved toward unified benchmarks and the semantic-versus-covariate distinction of [15]. Datasheets [24] and model cards [23] provide ready-made templates for documenting data and models alongside these results.

Open problems.

- **Unknown unknowns.** Any held-out set of "unknown" categories is a sample; there is no general way to certify rejection of all novel inputs.
- **Evaluation under shift.** Collecting labeled data for every deployment condition is expensive, and synthetic perturbations do not reliably predict real shifts.
- **Pretraining overlap.** With web-scale pretraining, it is increasingly hard to guarantee that test images were never seen, which complicates the "locked test" principle.
- **Small-sample comparisons.** Many applied projects have hundreds, not millions, of labeled examples, where the variance of every estimate in this chapter is large and statistical comparison is essential.

Key takeaways

- Write the protocol first: observation, sampling unit, label space, loss, reject policy and the role of each split. Most evaluation failures are protocol failures.
- Lock the test set before any modeling, keep every fitted transformation inside a pipeline, and treat features and preprocessing as hyperparameters.
- Compare all candidates on identical folds; baselines tell you what "good" means, and on these digits a 3-NN baseline is hard to beat.
- Estimate the tuned procedure with nested cross-validation, and use a corrected resampled test before claiming one model is better.
- Do error analysis on out-of-fold development predictions; use learning curves to decide between more data and a richer model.
- Open the test set once and report counts, an interval and the scope (here: new images from the same 13 writers).
- Closed-set accuracy, selective risk, unknown-class rejection and robustness to shift are different properties: a 99%-accurate digit classifier still accepted 35% of unseen digits and lost almost half its accuracy under a one-cell shift.

Exercises

1. **Distance-based rejection.** In the open-set experiment, replace maximum class probability with the distance from a test image to its nearest training image in the train-fitted PCA space. Choose the threshold with validation known digits only, then report known coverage and unknown acceptance on the test sets. Does it beat confidence thresholding?

   Hint

   Fit `StandardScaler` and `PCA` on `X[train]` only, then use `NearestNeighbors(n_neighbors=1)` on the transformed training set. The score is the distance (larger means "more unknown"); set the threshold at the 95th percentile of validation distances so about 95% of known validation digits are accepted. Report both numbers; do not change the threshold after seeing the test results.

2. **Risk–coverage without cheating.** Draw the risk–coverage curve of the 10-class tuned SVM using its decision values as confidence, first with out-of-fold development predictions, then (for reporting only) on the test set. Explain how selective accuracy can improve while useful coverage collapses, and what happens at zero coverage.

   Hint

   Sort examples by decreasing confidence; at each cut, coverage is the fraction kept and risk is the error rate among them. At very low coverage only a few examples remain, so risk is noisy, and at zero coverage it is undefined. Choose any operating point on the development curve, then read off its test value.

3. **Fix the shift weakness.** Augment each *training fold* with copies of its images shifted by one cell left, right, up and down. Re-run the nested comparison of step 6 and the stress test of step 9. Which accuracies change, and why must augmentation happen inside the folds?

   Hint

   Write a small transformer or a custom loop that augments only the training part of each split; augmenting before splitting puts shifted copies of validation images into training, which is leakage. Expect the shifted-test accuracy to rise sharply and clean accuracy to stay similar or change slightly.

4. **Writer-disjoint evaluation by design.** The UCI dataset's training portion was written by 30 people and its test portion by 13 different people [3]. Design (on paper) an experiment that measures generalization to new writers. Which numbers in this chapter would you expect to change, and which conclusions would you no longer be able to draw from the bundled data alone?

   Hint

   Train and select on one writer group and test on the other. Expect lower accuracy for k-NN in particular, since its advantage relies on near-duplicates from the same writers. Without writer IDs inside a portion, you still cannot do grouped cross-validation within development, so validation remains writer-mixed.

5. **Is it significant?** Using the 10 fold scores from step 5, apply the corrected resampled t-test to logistic regression versus random forest, and to LDA versus logistic regression. Then compute the naive paired t-test without the $n_{\text{te}}/n_{\text{tr}}$ term. In which case would the two tests lead to different conclusions?

Hint

With 5-fold CV, $n_{\text{te}}/n_{\text{tr}} \approx 0.25$, which is much larger than $1/k = 0.1$, so the corrected variance is about 3.5 times the naive one and the t statistic shrinks by a factor of about $\sqrt{3.5} \approx 1.9$. Small but consistent differences are where the conclusions diverge.

6. **Frozen features versus fine-tuning.** Design an experiment comparing a linear probe on frozen pretrained features with full fine-tuning for a real image dataset of your choice. List every decision that could leak test information.

Hint

Include checkpoint and layer choice, image resizing and normalization, augmentation, learning-rate and epoch selection (early stopping), threshold and calibration fitting, prompt design if a vision–language model is used, and possible overlap between pretraining data and your test images.

References

1. S. Theodoridis and K. Koutroumbas, *Pattern Recognition*, 4th ed., Academic Press (Elsevier), 2009; Ch. 1 “Introduction”, Ch. 5–7 on feature selection and generation, Ch. 10 “Supervised Learning: The Epilogue”. doi
2. C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2006; Sec. 1.5 (decision theory and the reject option), Sec. 1.3 (model selection), Ch. 14 (combining models). official page and PDF
3. E. Alpaydin and C. Kaynak, “Optical Recognition of Handwritten Digits,” UCI Machine Learning Repository, 1998. doi
4. F. Pedregosa et al., “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011. JMLR; dataset loader: load_digits
5. S. Kapoor and A. Narayanan, “Leakage and the reproducibility crisis in machine-learning-based science,” *Patterns*, vol. 4, no. 9, 2023. doi
6. S. Varma and R. Simon, “Bias in error estimation when using cross-validation for model selection,” *BMC Bioinformatics*, vol. 7, 91, 2006. doi
7. G. C. Cawley and N. L. C. Talbot, “On over-fitting in model selection and subsequent selection bias in performance evaluation,” *Journal of Machine Learning Research*, vol. 11, pp. 2079–2107, 2010. JMLR
8. C. Nadeau and Y. Bengio, “Inference for the generalization error,” *Machine Learning*, vol. 52, pp. 239–281, 2003. doi
9. S. Raschka, “Model evaluation, model selection, and algorithm selection in machine learning,” arXiv:1811.12808, 2018. arXiv

10. Y. LeCun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. doi
11. Y. Geifman and R. El-Yaniv, "Selective classification for deep neural networks," in *Advances in Neural Information Processing Systems 30* (*NIPS*), 2017. proceedings
12. D. Hendrycks and K. Gimpel, "A baseline for detecting misclassified and out-of-distribution examples in neural networks," in *Proc. ICLR*, 2017. arXiv
13. W. J. Scheirer, A. de Rezende Rocha, A. Sapkota and T. E. Boult, "Toward open set recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 7, pp. 1757–1772, 2013. doi
14. C. Geng, S.-J. Huang and S. Chen, "Recent advances in open set recognition: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 10, 2021. doi
15. J. Yang, K. Zhou, Y. Li and Z. Liu, "Generalized out-of-distribution detection: A survey," *International Journal of Computer Vision*, 2024. doi
16. S. Vaze, K. Han, A. Vedaldi and A. Zisserman, "Open-set recognition: A good closed-set classifier is all you need?," in *Proc. ICLR*, 2022. arXiv
17. B. Schölkopf, J. C. Platt, J. Shawe-Taylor, A. J. Smola and R. C. Williamson, "Estimating the support of a high-dimensional distribution," *Neural Computation*, vol. 13, no. 7, pp. 1443–1471, 2001. doi
18. F. T. Liu, K. M. Ting and Z.-H. Zhou, "Isolation forest," in *Proc. IEEE International Conference on Data Mining* (*ICDM*), 2008. doi
19. J. G. Moreno-Torres, T. Raeder, R. Alaiz-Rodríguez, N. V. Chawla and F. Herrera, "A unifying view on dataset shift in classification," *Pattern Recognition*, vol. 45, no. 1, pp. 521–530, 2012. doi
20. B. Recht, R. Roelofs, L. Schmidt and V. Shankar, "Do ImageNet classifiers generalize to ImageNet?," in *Proc. ICML*, PMLR 97, pp. 5389–5400, 2019. PMLR
21. P. W. Koh, S. Sagawa, H. Marklund et al., "WILDS: A benchmark of in-the-wild distribution shifts," in *Proc. ICML*, PMLR 139, pp. 5637–5664, 2021. PMLR
22. C. Guo, G. Pleiss, Y. Sun and K. Q. Weinberger, "On calibration of modern neural networks," in *Proc. ICML*, 2017. arXiv
23. M. Mitchell, S. Wu, A. Zaldivar, P. Barnes, L. Vasserman, B. Hutchinson, E. Spitzer, I. D. Raji and T. Gebru, "Model cards for model reporting," in *Proc. Conference on Fairness, Accountability, and Transparency* (*FAT**), 2019. doi
24. T. Gebru, J. Morgenstern, B. Vecchione, J. W. Vaughan, H. Wallach, H. Daumé III and K. Crawford, "Datasheets for datasets," *Communications of the ACM*, vol. 64, no. 12, 2021. doi
25. M. Oquab, T. Darcet, T. Moutakanni et al., "DINOv2: Learning robust visual features without supervision," arXiv:2304.07193, 2023. arXiv