IAIO Training

Eljakim Schrijvers

January 2026

Feedback: `eljakim+ai@gmail.com`

Preface

> I am still mine, and therefore
> yours to hold.
>
> Eljakim Schrijvers, *I Am Still Mine*

I should begin with a confession: I am not an AI researcher.

What I am is someone who loves algorithms, programming, and helping young people discover the joy of computational thinking. I serve as one of the coaches for the Dutch delegation to the International Olympiad in Informatics (IOI), and in 2024 I had the privilege of bringing the European Girls' Olympiad in Informatics (EGOI) to The Netherlands. Competitive programming has been a passion of mine for years. There is something deeply satisfying about watching a student's face light up when an elegant solution clicks into place.

I studied neural networks in the 1990s, when a "deep" network had three layers. Returning to the field decades later, I find the foundations surprisingly unchanged. What has changed, dramatically, is everything else: the computational power, the data, and the sheer scale at which these old ideas now operate. The difference between a neural network in 1995 and one in 2025 is not the architecture diagram on a whiteboard; it is the thousands of GPUs and billions of parameters that bring that diagram to life.

This book is my attempt to distill what I have learned into a training guide for students. This is not a scientific paper: no bibliography, no citation list at the end. The goal is to help you understand and apply AI concepts, not to trace their intellectual lineage. I have done my best to present ideas accurately and fairly, but I have prioritised clarity over academic convention.

I used AI tools as a writing assistant throughout the process of writing this book. The ideas, structure, and pedagogical choices are mine; the AI helped me express them more clearly and consistently than I could have managed alone within my own time constraints.

While writing, I found myself thinking about AI, not just as exam material but personally. What does it mean when an algorithm predicts what you will say before you say it? When it learns your patterns and then finishes your thoughts? I wrote a song about it. The lyrics and melody are mine; I used AI only as a very poor rhyme dictionary. You will find the score in Appendix I. It should not be part of any exam. But I believe every good story deserves a song. It makes the most important part easier to remember. So perform it, play it, hum it. Make it ours.

If you have feedback, corrections, or suggestions, I would love to hear from you. Send an email to `eljakim+ai@gmail.com`. I cannot guarantee a response, but I will read every message.

Good luck.

Eljakim Schrijvers
January 2026

Contents

Introduction

> Everything should be made as simple as possible, but not simpler.[1]
>
> Albert Einstein

What This Book Is About

This book prepares you for the International AI Olympiad (IAIO), a competition testing AI's mathematical foundations, algorithmic techniques, and ethical implications. The IAIO is not about memorizing formulas. You will face problems requiring reasoning about algorithms you've never seen, applying concepts in novel situations, and thinking critically about AI systems.

How the IAIO Tests Thinking

IAIO differs fundamentally from school exams.

Pre-Practice Checklist: Habits to Avoid

- ☐ Memorizing formulas without understanding derivations
- ☐ Studying only what was explicitly taught
- ☐ Solving only problems similar to homework
- ☐ Writing everything you know (graders reward precision)
- ☐ Skipping proofs to focus on calculations
- ☐ Stopping when you get "an answer" (IAIO asks: when does this fail?)

[1] This wording is a widely used paraphrase. It traces back to a 1933 lecture in which Einstein stated: "The supreme goal of all theory is to make the irreducible basic elements as simple and as few as possible without having to surrender the adequate representation of a single datum of experience."

Pre-Practice Checklist: Habits to Build

- ☐ Understanding *why* each algorithm works
- ☐ Constructing counterexamples where methods fail
- ☐ Deriving bounds from first principles
- ☐ Reading problem statements twice (quantifiers, edge cases)
- ☐ Stating assumptions explicitly (partial credit)
- ☐ Checking answers with sanity tests

Core difference: School tests what you learned. IAIO tests whether you can *think* with what you learned.

Chapter Classification

Core IAIO Material (60% study time)

Chapters 3 (Data), 4 (Supervised), 5 (Evaluation), 6 (Unsupervised), 8 (Search/CSP), 11 (Learning Theory)

Important Supporting (30% study time)

Chapters 1 (Intro), 2 (Ethics), 7 (Kernels), 9 (RL), 10 (Generative), 12 (VC Dimension)

Reference / Enrichment (10% study time)

Chapters 13 (Matrix Factorization), 14 (Modern DL), Appendices

About the Appendices

The appendices are reference material, not additional chapters to study. Appendix A covers competition strategy. Appendix B summarizes mathematical notation. Appendices C through F review probability, calculus, logic, and linear algebra. Appendix G covers computational complexity. Appendix H contains solutions to all practice problems. I finish with a song in Appendix I.

If you encounter an unfamiliar concept in a chapter (say, Bayes' theorem in Chapter 4, or eigenvalues in Chapter 6), look it up in the relevant appendix, understand it well enough to continue, and move on. Do not read the appendices front-to-back before starting the main chapters.

Chapter Dependencies

Not all chapters must be read in strict order. The diagram below shows which chapters build on which others. An arrow from Chapter A to Chapter B means you should read A before B.

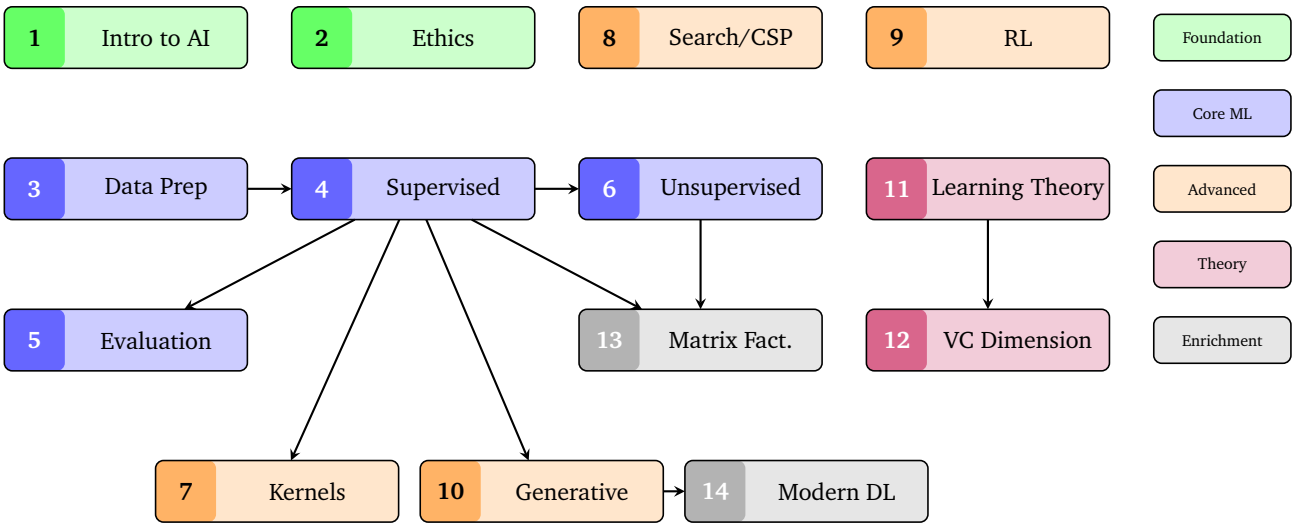


Figure 1: Chapter dependencies. Follow arrows to find prerequisites. Chapters 1, 2, 8, 9, and 11 can be read independently.

Recommended Reading Paths

If you have time, you can go through everything. If you do not have enough time I recommend you always read chapters 1 and 2, and then follow one of the following paths:

The Core ML Path (essential machine learning):

$$\text{Ch } 1{+}2 \rightarrow \text{Ch } 3 \rightarrow \text{Ch } 4 \rightarrow \text{Ch } 5 \rightarrow \text{Ch } 6$$

The Theory Path (mathematical foundations):

$$\text{Ch } 1{+}2 \rightarrow \text{Ch } 4 \rightarrow \text{Ch } 7 \rightarrow \text{Ch } 11 \rightarrow \text{Ch } 12$$

The Modern AI Path (deep learning and generative AI):

$$\text{Ch } 1{+}2 \rightarrow \text{Ch } 4 \rightarrow \text{Ch } 10 \rightarrow \text{Ch } 14$$

The Search and Reasoning Path:

$$\text{Ch } 1{+}2 \rightarrow \text{Ch } 8 \rightarrow \text{Appendix E (Logic)}$$

What You Will Need

This book uses high school mathematics: algebra, basic calculus (derivatives and integrals), and probability (basic rules, conditional probability, Bayes' theorem). If you haven't encountered all of these yet, don't worry — the appendices cover everything you need. Look things up as you go.

How to Use This Book

Work through each chapter's practice problems before moving on. They are designed to expose gaps in understanding, not to confirm what you already know. When you get stuck, resist the urge to check the solution immediately; the discomfort of being stuck is where the deepest learning happens. When you do check, focus on *why* the approach works, not just on the final answer.

The section below describes the most common ways students lose points at IAIO. Read it now, then revisit it after completing a few chapters.

Common Failure Modes at IAIO

Failure Mode 1: Shallow Understanding

Symptom: You can explain what an algorithm does but not *why* it works or *when* it fails.

Example: "K-means minimizes within-cluster variance" is shallow. Deep understanding: "K-means alternates between assigning points to nearest centroids and updating centroids to cluster means. Each step decreases or maintains the objective, guaranteeing convergence to a local minimum. The non-convex objective means different initializations yield different solutions."

Fix: For every algorithm or theorem, derive it yourself (not just read the derivation), construct an input where it fails, and ask: "What changes if I modify this assumption?"

Failure Mode 2: Execution Errors

Symptom: You understand the concept but lose points on computation or misread the problem statement.

Common traps: forgetting to normalize probabilities, sign errors in gradients, confusing **log** vs **ln**, off-by-one errors in VC dimension, and misreading "for all" vs "there exists." The last one is especially dangerous: "Prove no linear classifier can solve XOR" requires showing **all** linear classifiers fail, not just that you could not find one that works.

Fix: Before writing, state exactly what you are proving and what type of proof is needed. After computing, run sanity checks: do probabilities sum to 1? Is the entropy non-negative? Does a counterexample break your claim?

Failure Mode 3: Partial Credit Traps

Symptom: You get part (a) wrong, which propagates to parts (b), (c), (d).

Strategies:

- If unsure about (a), state your assumption clearly and proceed: "Assuming $\boldsymbol{X}$ from part (a), we have..."
- Check if later parts give hints about (a). Sometimes (c) reveals what (a) should be.
- Budget time to revisit early parts after seeing the full problem.

Failure Mode 4: Running Out of Time

Symptom: You solve problems you know well but never attempt harder ones.

IAIO scoring: Partial credit on hard problems often exceeds full marks on easy ones.

Strategy:

- First pass: Solve problems you're confident about (but don't perfect them yet).
- Second pass: Attempt all remaining problems, writing at least the setup.
- Third pass: Return to polish and verify.

Good luck at the IAIO!

Part I

Foundations

Chapter 1

Introduction to Artificial Intelligence

> AI is whatever hasn't been done yet.
>
> Larry Tesler

1.1 What is Artificial Intelligence?

Imagine you could build a machine that thinks. Not just calculates (we have had calculators for centuries), but actually *thinks*: recognizes faces in a crowd, understands spoken words, makes complex decisions, learns from mistakes, and solves problems it has never encountered before. For most of human history, this was the stuff of dreams and science fiction. Today, it is reality.

Every time you unlock your phone with your face, ask Siri a question, or watch a movie recommended by Netflix, you are interacting with artificial intelligence. When a doctor uses a computer to help diagnose a disease, when a car drives itself down a highway, when a program defeats the world champion at chess or Go; these are all examples of AI at work. What seemed impossible just decades ago has become part of everyday life.

But what exactly *is* artificial intelligence? The term itself was coined in 1956, and people have been debating its definition ever since. At its heart, AI is about creating computer systems that can perform tasks we typically associate with human intelligence. These include perceiving the world through vision and hearing, understanding and generating language, learning from experience, reasoning about complex situations, and making decisions

under uncertainty.

What makes AI different from ordinary computer programs? Consider a traditional calculator: you give it numbers and operations, and it follows precise rules to compute a result. The programmer specifies exactly what the calculator should do in every situation. AI systems work differently. Instead of being explicitly programmed with rules for every situation, they *learn* patterns from data. Show an AI system thousands of pictures of cats and dogs, tell it which is which, and it will learn to distinguish them, even for pictures it has never seen before.

This is the key insight: AI systems discover patterns on their own. They are not explicitly told "cats have pointed ears" or "dogs have wet noses." Instead, they examine thousands of examples and figure out what distinguishes cats from dogs. The patterns they discover might be subtle ones that humans never consciously noticed.

Definition: Artificial Intelligence

Artificial Intelligence (AI) is the field of computer science dedicated to creating systems that can perform tasks typically associated with human intelligence. These tasks include recognizing images and objects, understanding and generating speech, making decisions and predictions, solving complex problems, and learning from experience. The distinguishing feature of AI systems is that they learn from data rather than being explicitly programmed with rules.

AI is not magic, and it is not the sentient robots of science fiction (at least, not yet). At its core, AI is mathematics and data working together. The mathematics provides frameworks for learning patterns and making decisions. The data provides the raw material from which those patterns are learned. The quality of what an AI system learns depends entirely on the quality of the data it learns from.

Reading Guide: Skim This Section

Section 1.2 provides historical context for AI development. While interesting, this material is **rarely tested on IAIO**. Skim for general awareness; focus your study time on Chapters 3–6 and 8–9.

1.2 A Brief History of AI

The dream of creating intelligent machines is ancient, appearing in myths and stories throughout human history. But the scientific pursuit of artificial

intelligence began surprisingly recently, and its history is a dramatic tale of soaring hopes, crushing disappointments, and remarkable breakthroughs.

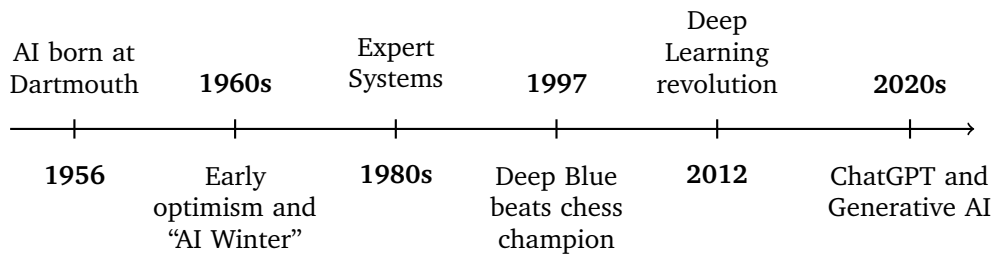


Figure 1.1: Key milestones in the history of artificial intelligence.

1.2.1 The Birth of AI

The field of AI officially began in the summer of 1956 at a small conference at Dartmouth College in New Hampshire, USA. A group of researchers proposed an ambitious project: "We propose that a 2-month, 10-man study of artificial intelligence be carried out... The study is to proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it."[1]

This was an audacious claim: that human intelligence could be precisely described and then replicated in a machine. The researchers at Dartmouth were optimistic. Some predicted that machines would be capable of doing any work a human could do within twenty years. This optimism would prove dramatically premature.

1.2.2 The Perceptron and Its Limits

One of the earliest and most influential AI systems was the **perceptron**, invented by Frank Rosenblatt in 1958.[2] The perceptron was a simple neural network: it took multiple inputs, multiplied each by a weight, summed the results, and produced an output. Crucially, it could *learn* its weights from examples.

The perceptron generated enormous excitement. Rosenblatt made bold predictions about machines that could "perceive, recognize, and identify their surroundings without human training." The New York Times reported

[1] McCarthy, J., Minsky, M. L., Rochester, N., & Shannon, C. E. (1955). *A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence*. `http://jmc.stanford.edu/articles/dartmouth/dartmouth.pdf`

[2] Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6), 386–408.

that the Navy expected the perceptron to eventually "walk, talk, see, write, reproduce itself, and be conscious of its existence."[3]

Then came a devastating blow. In 1969, Marvin Minsky and Seymour Papert published *Perceptrons*,[4] a mathematical analysis proving that single-layer perceptrons could not learn certain simple functions, including XOR (exclusive or). The limitation was fundamental: without multiple layers, perceptrons could only learn linearly separable patterns.

The book's impact was chilling. Funding for neural network research dried up almost overnight. The first "AI winter" had begun.

1.2.3 Expert Systems and the Second Winter

AI experienced a resurgence in the 1980s with the rise of **expert systems**, programs designed to capture the knowledge of human experts in specific domains. These systems encoded rules like "IF the patient has fever AND cough AND recent travel to tropical regions THEN consider malaria." By chaining together many such rules, expert systems could provide useful advice for medical diagnosis, equipment repair, mineral exploration, and other specialized tasks.

But expert systems had limitations. The rules had to be manually encoded by human programmers working with domain experts. This was a slow, expensive process. The systems were brittle: they worked well within their narrow domain but failed spectacularly when confronted with situations their rules did not cover. By the late 1980s, interest waned, and AI entered another winter.

1.2.4 The Backpropagation Revival

While expert systems rose and fell, a quieter revolution was brewing. In 1986, David Rumelhart, Geoffrey Hinton, and Ronald Williams published a landmark paper showing how to train *multi-layer* neural networks using an algorithm called **backpropagation**.[5] This solved the problem that had killed the perceptron: with multiple layers and backpropagation, neural networks could learn XOR and far more complex patterns.

The idea was not entirely new (versions had been discovered earlier), but this paper made it accessible and demonstrated its power. Neural networks were back, though it would take decades for them to reach their full potential.

---

[3] *New York Times*, July 8, 1958, p. 25: "New Navy Device Learns by Doing."

[4] Minsky, M. & Papert, S. (1969). *Perceptrons: An Introduction to Computational Geometry*. MIT Press.

[5] Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533–536.

1.2.5 The Statistical Turn

The 1990s saw AI take a more rigorous, mathematical direction. Researchers moved away from hand-coded rules toward **statistical methods** that could learn patterns from data. Support Vector Machines (SVMs), Bayesian networks, and ensemble methods like Random Forests emerged during this period.

These methods had solid theoretical foundations and worked well on medium-sized datasets. They became the workhorses of practical machine learning, and many remain important today. Much of this book covers techniques from this statistical tradition.

1.2.6 Big Data and GPUs

Two developments in the 2000s set the stage for the deep learning revolution. First, the internet created unprecedented amounts of data: billions of images, documents, and recordings that could be used to train learning algorithms. Second, graphics processing units (GPUs), originally designed for video games, turned out to be extraordinarily good at the parallel matrix operations that neural networks require.

Neural networks are computationally hungry. Training them on CPUs was painfully slow. GPUs made it feasible to train much larger networks on much larger datasets.

1.2.7 The Deep Learning Revolution

The watershed moment came in 2012 when a deep learning system called AlexNet won the ImageNet competition, an annual contest to recognize objects in images. AlexNet did not just win; it crushed the competition, reducing the error rate by more than 10 percentage points compared to the next best system.[6] This was not incremental progress; it was a leap that made the entire field take notice.

Within a few years, deep learning systems were achieving human-level or superhuman performance on task after task: recognizing faces, transcribing speech, translating languages, playing games. The AI winter was definitively over.

[6]Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 25, 1097–1105. Top-5 error dropped from 26.2% to 15.3%.

1.2.8 Transformers and Large Language Models

In 2017, researchers at Google published a paper titled "Attention Is All You Need,"[7] introducing the **transformer** architecture. Transformers processed sequences using a mechanism called *attention* that allowed them to capture long-range dependencies more effectively than previous approaches.

Transformers proved remarkably scalable. Making them larger and training them on more data consistently improved performance. This led to the development of **Large Language Models** (LLMs): GPT, BERT, and their successors, trained on vast amounts of text from the internet.

In November 2022, OpenAI released ChatGPT, bringing large language models to the general public. The name reveals its architecture: Chat**GPT** stands for "Chat Generative Pre-trained **Transformer**." ChatGPT demonstrated that AI could engage in fluent, knowledgeable conversations on almost any topic. Within two months, it had over 100 million users.[8]

Today, we are witnessing the rise of **generative AI**: systems that can create new content rather than just analyzing existing data. Large language models write essays, answer questions, and explain concepts. Image generators like DALL-E and Midjourney create original artwork from text descriptions. We are living through one of the most exciting periods in the history of technology.

1.3 Three Types of AI

Scientists distinguish between three levels of artificial intelligence, each representing a fundamentally different degree of capability. This classification helps us understand where we are today and where we might be heading.

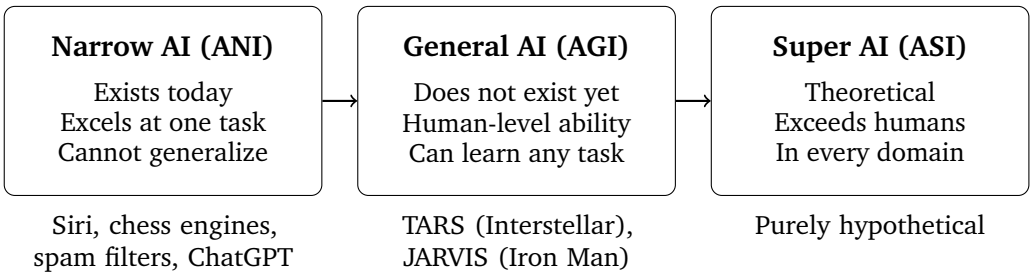


Figure 1.2: The three types of AI: only Narrow AI exists today.

[7] Vaswani, A. et al. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, 30, 5998–6008. https://arxiv.org/abs/1706.03762

[8] Hu, K. (2023, February 1). ChatGPT sets record for fastest-growing user base — analyst note. *Reuters*. https://www.reuters.com/technology/chatgpt-sets-record-fastest-growing-user-base-analyst-note-2023-02-01/

1.3.1 Narrow AI (Artificial Narrow Intelligence)

Every AI system that exists today, no matter how impressive, is an example of **Narrow AI**. Narrow AI systems excel at specific, well-defined tasks but cannot transfer their abilities to other domains. They are specialists, not generalists.

Consider some examples. Voice assistants like Siri and Alexa can understand speech and answer questions, but they cannot drive a car or diagnose a disease. Recommendation systems on Netflix and YouTube are remarkably good at predicting what you might want to watch, but they know nothing about anything else. Chess programs can defeat any human player, but they cannot play checkers, let alone hold a conversation. Even ChatGPT, despite its multimodal capabilities, is a narrow AI: it excels at language-related tasks but does not have general intelligence, and it cannot control robots or do anything beyond what it was trained for.

The key limitation of Narrow AI is the lack of generalization. A system trained to recognize cats cannot automatically recognize dogs. A system trained on English cannot automatically understand French. Each new task requires new training data, new model design, and often significant effort from human engineers. Narrow AI is powerful, but it is powerful in the same way that a screwdriver is powerful: excellent for its intended purpose, useless for others.

1.3.2 Artificial General Intelligence (AGI)

Artificial General Intelligence refers to AI systems that could perform any intellectual task a human can do. An AGI system would be able to learn new skills without being specifically programmed for them, reason about unfamiliar problems, transfer knowledge from one domain to another, understand context and nuance, and adapt to unexpected situations.

AGI does not yet exist. No one knows when, or even whether, it will be achieved. Some researchers believe we are just years away; others think it will take decades or longer; some doubt it is possible at all.

Science fiction has imagined many visions of AGI. Think of TARS from *Interstellar*, who can have natural conversations, tell jokes, pilot spacecraft, and adapt to completely unexpected situations on alien planets. Or JARVIS from *Iron Man*, who serves as an intelligent assistant capable of helping with any task, from managing a household to supporting superhero operations. These fictional examples capture what AGI might look like: systems that are truly intelligent in the way humans are intelligent, not just very good at specific tasks.

The path from today's Narrow AI to AGI remains unclear. Some believe it will require fundamentally new approaches we have not yet discovered. Others think scaling up current techniques (more data, more computing power, larger models) will eventually produce AGI. This is one of the most important open questions in the field.

1.3.3 Artificial Superintelligence (ASI)

Artificial Superintelligence is a theoretical concept: AI that surpasses human intelligence in every domain. A superintelligent AI would be better than humans at scientific research, social reasoning, creativity, and every other cognitive task.

ASI remains purely hypothetical. We have no idea how to build it, whether it is possible, or what it would be like. Some researchers worry that superintelligent AI could pose existential risks if its goals were not carefully aligned with human values. Others argue these concerns are premature given that we have not even achieved AGI. Regardless of where one stands on these debates, ASI represents the far horizon of AI research, a possibility that may or may not be achievable, but that shapes how many people think about the field's trajectory.

1.4 Key Technologies in AI

Artificial intelligence encompasses many specialized technologies and subfields. Understanding the relationships between them is essential for navigating the field.

1.4.1 Machine Learning

Machine learning is the technology at the heart of modern AI. Instead of programming computers with explicit rules, we give them examples and let them discover patterns automatically.

The traditional approach to software development involves a programmer analyzing a problem, designing an algorithm, and writing code that implements that algorithm step by step. The program then follows these instructions exactly. Machine learning inverts this process. The programmer provides data (examples of inputs and desired outputs), and the learning algorithm discovers a function that maps inputs to outputs. The rules are learned, not programmed.

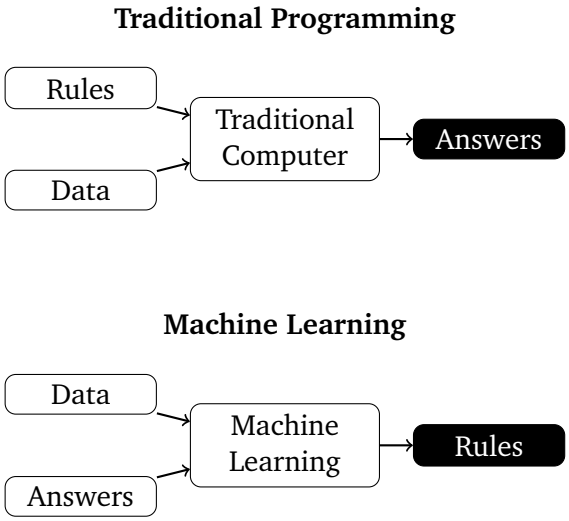


Figure 1.3: Traditional programming requires humans to specify rules explicitly. Machine learning discovers rules automatically from data and examples.

This approach is powerful because many tasks are hard to describe with explicit rules but easy to demonstrate with examples. How would you write rules to recognize a face? It is nearly impossible to articulate the precise features that make a face recognizable. But you can easily provide thousands of example photos labeled "face" or "not face," and a machine learning algorithm will figure out the patterns.

1.4.2 Deep Learning

Deep learning is a subset of machine learning that uses neural networks with many layers. The term "deep" refers to the depth of these networks (the number of processing layers between input and output).

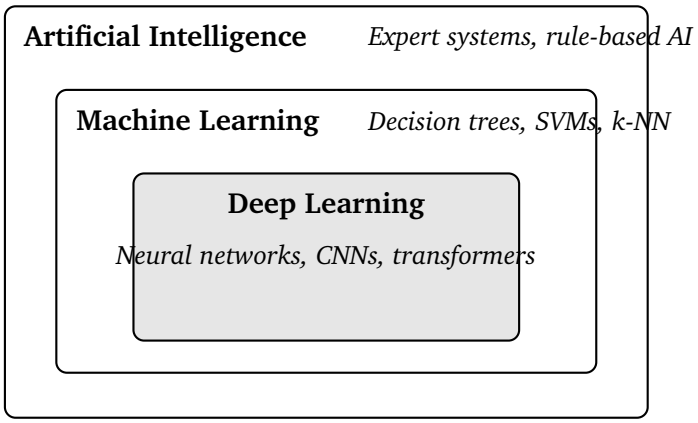


Figure 1.4: The relationship between AI, Machine Learning, and Deep Learning. Deep learning is a subset of machine learning, which is itself a subset of AI.

Neural networks are computational models loosely inspired by the structure of biological brains. They consist of interconnected nodes ("neurons") organized in layers. Information flows from input nodes through hidden layers to output nodes, with each connection having a weight that determines its strength. Learning involves adjusting these weights so that the network produces correct outputs for given inputs.

Deep networks can learn extremely complex patterns because each layer builds upon the representations learned by previous layers. Early layers might detect simple features like edges and colors; middle layers might combine these into shapes and textures; late layers might recognize complete objects. This hierarchical feature learning is what gives deep learning its power.

The deep learning revolution has produced most of the AI advances you hear about today: systems that recognize faces, transcribe speech, translate languages, generate images, and converse in natural language. When people talk about AI in the news, they are usually talking about deep learning.

1.4.3 Computer Vision

Computer vision is the field concerned with enabling computers to interpret and understand visual information from the world. When your phone recognizes faces in photos, when a self-driving car detects pedestrians, when a quality control system spots defective products on an assembly line; these are all computer vision applications.

The core challenge of computer vision is bridging the gap between raw pixel values and meaningful understanding. A digital image is just a grid of

numbers representing color intensities. A human looking at the same image instantly perceives objects, people, actions, and relationships. Teaching computers to make this leap from numbers to understanding has been one of AI's great challenges.

Computer vision encompasses many specific tasks. **Image classification** determines what category an image belongs to: is this a photo of a cat or a dog? **Object detection** finds and identifies multiple objects within an image: there is a cat in the lower left, a dog in the center, and a person in the background. **Image segmentation** labels every pixel according to what object it belongs to. Each task is progressively more detailed and challenging.

1.4.4 Natural Language Processing

Natural Language Processing (NLP) enables computers to understand and generate human language. This includes both written text and spoken language.

Language is extraordinarily complex. The same words can have different meanings in different contexts. Sentences can be ambiguous. Meaning often depends on cultural knowledge, common sense, and understanding of the speaker's intent. For decades, NLP systems could only handle narrow, well-defined tasks like keyword search. The deep learning revolution changed this, producing systems that can engage in nuanced conversation, translate between languages, summarize documents, and answer complex questions.

Large language models like GPT represent the current frontier of NLP. Trained on enormous amounts of text from the internet, these models learn patterns in how language is used. They can then generate coherent, contextually appropriate text on virtually any topic. The results are impressive enough that many people find it difficult to distinguish AI-generated text from human writing.

Reading Guide: Skim This Section

The following section explains the AI development pipeline. This provides useful context for later chapters but is **not directly tested on IAIO**. Skim for conceptual understanding.

1.5 The AI Development Pipeline

Building an AI system is not a single step but a process involving multiple stages. Understanding this pipeline is essential for working in AI.

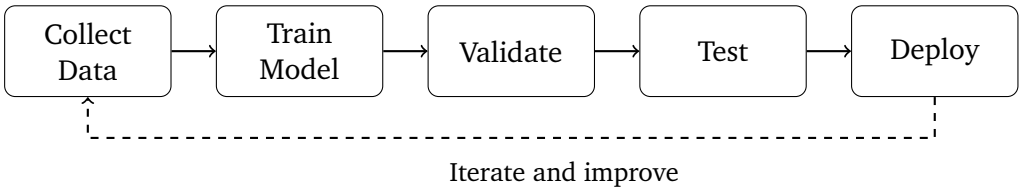


Figure 1.5: The AI development pipeline: collect data, train, validate, test, deploy, then iterate to improve.

1.5.1 Step 1: Collect Data

Everything in AI begins with data. If you want to build a system that recognizes cats, you need thousands of images of cats (and non-cats) with labels indicating which is which. If you want to build a system that predicts stock prices, you need historical data on prices and relevant factors. The data is the raw material from which the AI will learn.

Key Insight: Data is Everything

In AI, data is everything. The quality of an AI system depends critically on the quality of its training data. More data generally leads to better results, but the data must also be accurate (labels must be correct), representative (covering the full range of cases the system will encounter), and relevant (matching the actual use case). Poor data produces poor AI, no matter how sophisticated the algorithm.

Collecting good data is often the hardest part of AI development. Data must be gathered, cleaned, labeled, and verified. Labeling alone can require enormous human effort; someone must look at each image and mark what it contains. Companies sometimes employ thousands of labelers working around the clock.

1.5.2 Step 2: Train the Model

Training is the process where the AI actually learns. The algorithm examines the training data, looking for patterns that allow it to predict the correct outputs from the inputs.

Imagine teaching a child to recognize letters. You show an "A" and say "This is an A." You show a "B" and say "This is a B." At first, the child guesses randomly. Over time, as you correct mistakes, the child starts to notice distinguishing features: an "A" has a peak at the top and a horizontal bar; a "B" has bumps on the right side. Eventually, the child can recognize letters they have never seen before.

Machine learning works similarly. The algorithm makes predictions on training examples, computes how wrong it was (the "error" or "loss"), and adjusts its internal parameters to reduce that error. This process repeats millions or billions of times. Gradually, the model improves, learning to make accurate predictions.

1.5.3 Step 3: Validate

During training, we need to check whether the model is actually learning useful patterns or just memorizing the training data. This is where validation comes in.

We set aside some data; the validation set, which the model never trains on directly. Periodically during training, we test the model on this validation data to see how it performs on examples it has not seen before. If performance on training data is good but validation performance is poor, the model is memorizing rather than learning; a problem called overfitting.

Validation also helps us tune the model's settings. There are many choices to make when training a model: how fast should learning proceed? How complex should the model be? How long should training continue? By testing different choices on validation data, we can find settings that produce the best generalization to new examples.

1.5.4 Step 4: Test

Once training is complete and we have chosen our best model configuration, we perform a final evaluation on test data. This data has been held out from the entire development process; the model has never seen it, and we have not used it to make any decisions about the model.

Testing gives us an unbiased estimate of how well the model will perform in the real world. It is like the final exam that determines the grade: no more chances to improve based on the results.

A critical rule: never train on test data. If you look at test performance and then make changes to improve it, the test set no longer provides an unbiased estimate. It becomes just another form of validation data.

1.5.5 Step 5: Deploy

Deployment means putting the model into actual use. This might mean running it on a cloud server where millions of users can access it, embedding it in a smartphone app, or integrating it into a medical device. The model is now doing real work in the real world.

Deployment is not the end of the process. Once in production, the model should be monitored to ensure it continues performing well. Real-world data might differ from training data in unexpected ways, causing performance to degrade. New data should be collected to improve future versions. AI development is an ongoing cycle of improvement, not a one-time effort.

1.6 Prerequisites for Learning AI

To succeed in artificial intelligence, whether in this Olympiad or in a future career; you will need foundations in three areas.

1.6.1 Mathematics

AI is fundamentally mathematical. The algorithms that power machine learning are built on concepts from several branches of mathematics.

Linear algebra provides the language for describing data and transformations. Data points are vectors; operations on data are matrix multiplications. Neural networks are essentially sequences of matrix operations, and understanding linear algebra helps you understand what neural networks actually compute.

Calculus is essential because learning happens through optimization. When we train a model, we are searching for parameters that minimize error. The gradient; a concept from calculus, tells us which direction to adjust parameters to reduce error most quickly. This process, called gradient descent, is how neural networks learn.

Probability and statistics help us reason about uncertainty. Real-world data is noisy and incomplete. Models make predictions with varying confidence. Understanding probability allows us to quantify uncertainty, make principled decisions, and evaluate whether patterns in data are meaningful or just random chance.

You do not need to be a math prodigy to work in AI. What matters is building intuition about these concepts: understanding what they mean and why they matter, even if you cannot prove every theorem. The mathematics becomes clearer as you apply it to real problems.

1.6.2 Programming

You will write code. AI systems do not build themselves; someone must implement the algorithms, process the data, and orchestrate the training.

Python has become the dominant language for AI because of its readability

and the rich ecosystem of libraries available. PyTorch and TensorFlow provide tools for building and training neural networks. NumPy enables efficient numerical computation. Pandas helps with data manipulation. Matplotlib creates visualizations. These tools allow you to focus on the AI concepts rather than low-level implementation details.

For the IAIO, you should be comfortable with basic Python programming: loops, functions, conditionals, and data structures like lists and dictionaries. You should understand how to use NumPy arrays for numerical operations and how to load and manipulate data with Pandas. Experience with machine learning libraries like scikit-learn is valuable but not strictly required.

1.6.3 Understanding Computers

Knowing how computers work helps you write efficient code and make good engineering decisions. Modern AI training requires massive computation, and understanding what makes computation fast or slow matters.

CPUs (Central Processing Units) are general-purpose processors that handle most computer tasks. GPUs (Graphics Processing Units) were originally designed for rendering graphics but turned out to be extraordinarily good at the matrix operations neural networks require. Modern AI training almost always uses GPUs, and large-scale training uses thousands of them working together.

Understanding concepts like memory, parallel processing, and computational complexity will help you design systems that are not just correct but also practical to run.

1.7 Summary

Key Takeaways

Artificial Intelligence is the field of creating systems that perform tasks associated with human intelligence. The key feature of modern AI is learning from data rather than explicit programming.
Machine learning discovers patterns in data automatically. **Deep learning** uses neural networks with many layers to learn complex patterns hierarchically.
Only **Narrow AI** exists today; systems that excel at specific tasks but cannot generalize. **Artificial General Intelligence** (human-level AI) and **Artificial Superintelligence** remain future possibilities, not current realities.
The AI development pipeline involves collecting data, training models, validating and testing performance, and deploying to real-world use. This cycle repeats as systems are improved.
Success in AI requires foundations in mathematics (linear algebra, calculus, probability), programming (especially Python), and ethical reasoning about fairness, privacy, and accountability.
In AI, data is everything. The quality of an AI system depends fundamentally on the quality of the data it learns from.

1.8 Practice Problems

1. Classify each example as Narrow AI, AGI, or ASI:

   (a) A chess program that can defeat any human player

   (b) A robot that can learn any new physical task as quickly as a human apprentice

   (c) An AI that consistently produces better scientific discoveries than any human researcher

   (d) A spam filter for your email

2. A color image has dimensions $\mathbf{640} \times \mathbf{480}$ pixels. How many total numbers does a computer need to store this image? (Assume RGB color representation with one byte per color channel.)

3. Explain why it is important to keep test data separate from training data. What goes wrong if you use the same data for both?

4. A model achieves 99% accuracy on training data but only 60% accuracy

on test data. What problem does this indicate? What might have caused it, and how might you address it?

5. Explain in your own words the difference between the computer vision tasks of classification, localization, and object detection.

6. An AI system for screening job applications rejects 40% of female applicants but only 15% of male applicants with identical qualifications. What ethical principle is being violated? How might this have happened, and what could be done to address it? (See Chapter 2 for fairness definitions.)

Chapter 2

The Societal Impact of AI

> Technology is neither good nor bad; nor is it neutral.[1]
>
> Melvin Kranzberg

2.1 Introduction: Why Ethics Matters

Imagine you are building an AI system to help doctors diagnose diseases. After months of development and testing, the system performs brilliantly. In clinical trials, it outperforms human doctors at identifying certain conditions. You deploy it to hospitals across the country, confident that it will save lives.

A year later, you discover something troubling. The system works wonderfully for some patients but fails systematically for others. Patients from certain ethnic backgrounds are being misdiagnosed at alarming rates. Some have suffered because diseases were caught too late. What went wrong?

When you investigate, the answer becomes clear. The training data came primarily from hospitals in wealthy urban areas that served predominantly one demographic group. The AI learned patterns that work for that population but do not generalize to others. The system was not designed to be discriminatory. It simply learned from data that did not represent everyone.

This scenario is not hypothetical. Variations of this story have played out repeatedly as AI systems have been deployed in healthcare, criminal justice, hiring, lending, and countless other domains. AI systems can produce outcomes that **nobody planned or wanted**, simply by learning from data that

[1] Kranzberg, M. (1986). Technology and History: "Kranzberg's Laws." *Technology and Culture*, 27(3), 544–560.

contains the biases, assumptions, and blind spots of human society.

Understanding these risks (and learning how to prevent them) is not optional for anyone working with AI. Ethics is not a side topic or an afterthought; it is central to building AI systems that actually work for everyone.

Key Principle

AI amplifies what it learns from. If the training data is biased, the AI will be biased. If the data underrepresents certain groups, the AI will perform poorly for those groups. The algorithm itself has no inherent moral compass. It faithfully reproduces whatever patterns exist in its training data, whether those patterns are helpful or harmful.

2.2 Case Study 1: Bias in Image Search

Let us begin with a simple example that reveals how deeply societal biases can be embedded in AI systems.

Consider what happens when you search for "school boy" on an image search engine. You will find ordinary pictures: boys in uniforms, carrying backpacks, sitting in classrooms, playing at recess. The images depict students engaged in typical educational activities.

Now search for "school girl." The results often look remarkably different. Instead of ordinary depictions of female students, you may find images that objectify or sexualize young women. The character of the results differs fundamentally from the "school boy" search, despite both searches asking for the same concept. Just with different genders.

Why does this happen? The engineers who built the search algorithm almost certainly did not intend this result. The algorithm simply learned patterns from the data available to it; billions of images uploaded, labeled, and linked by people around the world.

The problem is that the internet reflects human society, including all of its biases and problematic content. There is far more objectifying content depicting women than men. Search algorithms learn these patterns and reproduce them at scale.

The lesson here is important: AI systems do not create bias from nothing. They expose and amplify biases that already exist in human behavior and the data that behavior produces.

IAIO Abstraction: Image Search Bias

What variables were biased? The training data (web images) contained gender-asymmetric content, more objectifying images for female-related queries.
Where in the pipeline did failure occur? Data collection. The algorithm learned from biased web content.
How would IAIO test this?

- Describe how training data composition can create disparate outputs for semantically equivalent queries.
- If search results for "school boy" vs "school girl" differ systematically, at what pipeline stage(s) could intervention occur?

2.3 Case Study 2: Bias in Hiring

Consider two names: **Tamika** and **Brandon**. In the United States, "Tamika" is a name associated primarily with African-American women, while "Brandon" is associated primarily with white men.

Researchers conducted a famous experiment:[2] they created pairs of **identical resumes**, same education, same work experience, same qualifications, but with different names at the top. They sent these resumes to actual employers.

The Resume Study Results

Resumes with white-sounding names received **50% more callbacks** than identical resumes with African-American-sounding names.
The qualifications were exactly the same. Only the name differed.

Now imagine building an AI system to screen job applications, trained on historical hiring data. If historical decisions were influenced by such biases, the AI will learn those biases.

The algorithm might learn: "Candidates named Brandon tend to be hired more often than candidates named Tamika." It learns this not because names predict job performance, but because the training data reflected human prejudice.

Worse, once deployed, such a system creates a feedback loop. Biased AI produces biased hiring decisions, which become the training data for future AI systems, perpetuating and amplifying the original bias.

[2]Bertrand, M. & Mullainathan, S. (2004). Are Emily and Greg more employable than Lakisha and Jamal? A field experiment on labor market discrimination. *American Economic Review*, 94(4), 991–1013.

IAIO Abstraction: Hiring Bias

What variables were biased? Names (proxy for demographics) correlated with hiring outcomes in historical data.
Where in the pipeline did failure occur? Training data reflected historical human bias; model learned and reproduced it.
Key concept: Feedback loops. Biased AI produces biased hiring, which becomes biased future training data, amplifying the original bias.
How would IAIO test this?

- Define disparate impact. How would you measure it given a confusion matrix by demographic group?
- An AI trained on historical hiring data uses "years since graduation" as a feature. Why might this be problematic?

2.4 Defining Fairness: A Fundamentally Hard Problem

What does it mean for an AI system to be "fair"? This seemingly simple question has no single answer. Different definitions of fairness capture different moral intuitions, and they often conflict with each other mathematically. This is not a failure of AI research; it reflects genuine disagreement about what fairness means.

The Core Problem

There is no mathematically "correct" definition of fairness. Choosing a fairness criterion is an **ethical choice**, not a technical one. Different stakeholders may reasonably disagree about which definition should apply.

2.4.1 Competing Definitions

Demographic Parity: The system should produce positive outcomes at equal rates across groups. If 30% of Group A receives loans, then 30% of Group B should also receive loans.

Equalized Odds: The system should have equal error rates across groups. If the false positive rate is 5% for Group A, it should also be 5% for Group B. Similarly for false negative rates.

Calibration: Among people the system assigns probability p, the actual positive rate should be p, regardless of group membership. If the system says

"70% chance of default," then 70% of those people should actually default, whether they are from Group A or Group B.

Individual Fairness: Similar individuals should receive similar predictions, regardless of group membership.

2.4.2 The Impossibility Result

Mathematical Impossibility

When base rates differ between groups (e.g., if historical default rates differ), it is **mathematically impossible** to simultaneously satisfy demographic parity, equalized odds, and calibration.[a]
This is a theorem, not a limitation of current technology. No algorithm, however sophisticated, can satisfy all three when base rates differ.

[a] Proved independently by Kleinberg, J., Mullainathan, S., & Raghavan, M. (2016), *Inherent Trade-Offs in the Fair Determination of Risk Scores*, ITCS 2017; and Chouldechova, A. (2017), Fair prediction with disparate impact, *Big Data*, 5(2), 153–163.

This impossibility result has profound implications. It means that:

- Fairness requires **choices** about which definition to prioritize
- These choices reflect **values**, not just technical considerations
- Different stakeholders may legitimately prefer different definitions
- There is no "fair" system that everyone will agree is fair

2.4.3 Which Definition When?

Different contexts may call for different fairness definitions:

Demographic parity may be appropriate when: historical data is known to be biased, or when equal representation is a policy goal (e.g., diverse hiring).

Equalized odds may be appropriate when: errors have serious consequences, and we want to ensure no group bears a disproportionate burden of mistakes.

Calibration may be appropriate when: the system's probability estimates are used directly for decisions, and accuracy of those estimates matters.

The choice is not purely technical. It requires input from affected communities, domain experts, ethicists, and policymakers.

2.5 The Bias-Detection-Mitigation Pipeline

Addressing bias requires systematic effort throughout the AI lifecycle:

Detection: Compare AI system outputs across different groups. Are error rates equal? Do certain groups face worse consequences from mistakes?

Analysis: Identify the root cause. Is the problem in the training data? The algorithm? The deployment context?

Mitigation: Address the identified problems, more representative data, removing problematic features, adjusting the algorithm, or adding human review.

Monitoring: Continue auditing after deployment. Bias can emerge or re-emerge over time.

Framework: Analyzing AI Ethics Problems

When analyzing any AI ethics scenario, systematically identify:
1. The bias source:
- Historical bias (training data reflects past discrimination)
- Representation bias (some groups underrepresented)
- Measurement bias (features measured differently across groups)
- Aggregation bias (single model for heterogeneous populations)

2. The pipeline stage: Data collection, labeling, feature selection, training, evaluation, deployment
3. The fairness definition violated: Demographic parity, equalized odds, calibration, individual fairness
4. The stakeholders: Who benefits? Who is harmed? Who decides?

2.6 Systematic Approach to AI Ethics Problems

IAIO ethics questions require structured reasoning, not narrative recall.

The Five-Point Analysis Template

Given any AI system, systematically identify:

1. **DATA**: What training data was used? Who is represented? Who is missing?
2. **OBJECTIVE**: What does the system optimize? (accuracy? engagement? profit?)
3. **METRIC**: How is success measured? What does the metric miss?
4. **HARM**: Who could be harmed? How? (false positives vs. false negatives)
5. **MITIGATION**: What interventions are possible? At what tradeoff?

Use this template for every ethics problem. Write one sentence per point. Whenever an IAIO problem asks "is this system acceptable?" or "what could go wrong?," this is the framework to apply.

Worked Example

System: A bank's loan approval AI trained on 10 years of historical decisions.

1. **DATA**: Historical decisions; reflects past biases; underrepresents groups historically denied credit.
2. **OBJECTIVE**: Minimize default rate (equivalently: maximize profit).
3. **METRIC**: AUC on historical data; does not measure fairness across groups.
4. **HARM**: Qualified applicants from underrepresented groups denied; perpetuates inequality.
5. **MITIGATION**: Remove proxies (zip code), add fairness constraints, audit by group. Tradeoff: may slightly increase default rate.

2.7 Privacy and Data Protection

AI systems are hungry for data. The more data they have, the better they typically perform. But this creates fundamental tensions with privacy.

2.7.1 Collection and Consent

Most AI training data was not collected with AI training in mind. Photos uploaded to social media, text written in emails, medical records created for treatment: all of this data is now being used to train AI systems. Did the people who created this data consent to this use?

The legal frameworks vary by jurisdiction. The European Union's General Data Protection Regulation (GDPR) requires explicit consent for many uses of personal data. Other regions have weaker protections. But legality aside, there are ethical questions: Should companies be able to use data for purposes the original creators never imagined?

2.7.2 De-anonymization

"We anonymized the data" is often offered as a solution to privacy concerns. But research has repeatedly shown that supposedly anonymous data can be re-identified.

De-anonymization Example

Researchers showed that 87% of Americans can be uniquely identified by just three pieces of information: zip code, birth date, and gender.[a] Data that seems anonymous often is not.

[a] Sweeney, L. (2000). Simple demographics often identify people uniquely. Carnegie Mellon University, Data Privacy Working Paper 3. `https://dataprivacylab.org/projects/identifiability/paper1.pdf`. Based on 1990 U.S. Census data. A replication using 2000 Census data found 63% (Golle, P., 2006, *Proceedings of the 5th ACM Workshop on Privacy in Electronic Society*), still remarkably high for just three fields.

Location data is particularly sensitive. Even without names attached, movement patterns can identify individuals (home location, work location, daily routines) and reveal sensitive information (visits to medical clinics, religious sites, political gatherings).

2.7.3 Training on Personal Data

Large language models are trained on vast amounts of text from the internet, which includes personal information, private conversations that were leaked, and copyrighted material. These models can sometimes regurgitate this training data, including personal details about real people.

This raises difficult questions: Who owns the patterns learned from personal data? If a model "knows" something about you because it was in the training data, do you have a right to have it removed?

2.7.4 Differential Privacy

Differential privacy is a mathematical framework that provides provable privacy guarantees. The idea is to add carefully calibrated noise to data or

computations, making it impossible to determine whether any individual was in the dataset.

Formally, an algorithm is ϵ-differentially private if its output distribution changes by at most a factor of e^{ϵ} when any single individual's data is added or removed. Smaller ϵ means stronger privacy but typically reduces accuracy.

Differential privacy is increasingly used for sensitive applications, but it involves tradeoffs: stronger privacy guarantees typically mean less accurate models.

2.8 Accountability and Responsibility

When an AI system makes a mistake, who is responsible? This question becomes urgent in high-stakes domains.

2.8.1 Medical AI

AI systems are increasingly used in healthcare: analyzing medical images, suggesting diagnoses, recommending treatments. When these systems make errors, patients can be harmed.

If an AI system misses a cancer diagnosis and the patient dies, who is liable? The doctor who relied on the AI? The hospital that deployed it? The company that built it? The engineers who trained it? Current legal frameworks were not designed for this situation.

2.8.2 Hiring and Employment

AI hiring tools screen millions of job applications. When these systems discriminate, who is accountable? Companies often purchase AI tools from vendors and may not understand how they work. Can a company be held responsible for discrimination by a system they did not build and cannot explain?

2.8.3 Criminal Justice and Credit

AI systems are used to make bail decisions, predict recidivism, and determine credit scores. These decisions have profound impacts on people's lives. Yet the affected individuals often have no way to understand why a decision was made, let alone challenge it.

The Accountability Gap

As AI systems become more autonomous and more opaque, a gap opens between the harm they can cause and the ability to hold anyone responsible. Addressing this gap requires both technical solutions (explainability, auditing) and legal/regulatory frameworks.

2.9 Transparency and Explainability

2.9.1 Black Boxes vs. Interpretable Models

Some AI models are inherently interpretable. A decision tree shows exactly which features led to a decision and why. A linear regression reveals the weight given to each factor.

Other models, particularly deep neural networks, are "black boxes." They may have millions of parameters, and even their creators cannot explain why they make specific decisions. The model works, but nobody knows exactly how.

Model Type	Interpretability	Typical Accuracy
Decision Tree	High	Moderate
Linear/Logistic Regression	High	Moderate
Random Forest	Medium	High
Deep Neural Network	Low	Highest

There is often a tradeoff: the most accurate models are the least interpretable. This creates difficult choices in high-stakes domains.

2.9.2 The Right to Explanation

The GDPR includes provisions for a "right to explanation": individuals affected by automated decisions have the right to "meaningful information about the logic involved." But what counts as "meaningful"? Can a neural network's decision ever be meaningfully explained?

Some researchers argue that explanations of black-box models are inherently unreliable: they are post-hoc rationalizations, not true explanations of how the model works. Others argue that even imperfect explanations are better than none.

2.9.3 When Is Interpretability Required?

Not all AI applications require interpretability. For a movie recommendation system, "the algorithm thought you'd like this" may be sufficient. But for a

system that denies someone a loan, a job, or parole, the stakes are different.

Consider: Would you accept a medical diagnosis from a system that cannot explain its reasoning? Would you accept a prison sentence based on an algorithm that nobody can interpret?

2.10 Feedback Loops and Distributed Harm

2.10.1 Reinforcing Inequality Over Time

AI systems do not just make predictions; they shape the world they are predicting. When a hiring AI screens out candidates from certain backgrounds, those candidates do not get jobs, do not gain experience, and become even less likely to be selected by future AI systems. The prediction becomes self-fulfilling.

This dynamic can entrench and amplify existing inequalities. A system that is only slightly biased at deployment can become severely biased over time as its decisions feed back into its training data.

2.10.2 Predictive Policing: A Case Study in Feedback Loops

Predictive policing systems use historical crime data to forecast where crimes will occur. Police are then deployed to those predicted "hot spots."

The Feedback Loop Problem

1. Historical data shows more crime in certain neighborhoods (often due to more policing there, not more actual crime)
2. AI predicts those neighborhoods are high-crime areas
3. Police are deployed there more heavily
4. More arrests are made there (because more police are present)
5. This confirms the prediction and reinforces the pattern
6. The cycle continues, intensifying over time

The system may be accurately predicting *where arrests will occur*, but this is not the same as predicting where crime occurs. The AI is modeling policing patterns, not crime patterns.

2.10.3 Distributed Harm

AI harms are often distributed: no single person suffers catastrophically, but millions of people are each harmed a little. A recommendation algorithm that slightly increases political polarization affects billions of users. A hiring

algorithm that slightly disadvantages certain groups affects millions of job seekers.

These distributed harms are hard to see, hard to measure, and hard to attribute. No individual can point to the AI and say "this system harmed me" in a way that would be legally actionable. Yet the aggregate harm may be enormous.

2.11 Environmental and Resource Costs

2.11.1 Energy Consumption

Training large AI models requires enormous computational resources. Training GPT-3 was estimated to consume approximately 1,287 MWh of electricity, equivalent to the annual consumption of over 100 American homes. The carbon footprint was estimated at 552 tonnes of CO_2, comparable to 500 round-trip flights between New York and San Francisco.[3]

As models grow larger, these costs increase. Training costs are paid once, but inference costs (actually running the model) accumulate with every use. A model serving millions of users consumes substantial ongoing resources.

2.11.2 Centralization of Power

The computational resources required to train frontier AI models are available only to a handful of wealthy organizations. This creates a concentration of power: a few companies control the most capable AI systems, the data they are trained on, and the decisions about how they are deployed.

This centralization raises concerns about:

- Democratic accountability: Who elected these companies to shape the future of AI?
- Competition: Can smaller organizations compete if AI requires billion-dollar investments?
- Access: Who gets to use the most powerful AI systems, and at what cost?

2.11.3 Should You Use This Data? Should You Build This System?

Beyond bias and fairness, there are broader ethical questions that AI practitioners must ask:

[3] Patterson, D. et al. (2021). Carbon emissions and large neural network training. *arXiv:2104.10350*. `https://arxiv.org/abs/2104.10350`

- Even if my model is unbiased, should this decision be automated at all?
- Even if the data was legally obtained, was it ethically obtained?
- Even if I can build this system, should I?
- Who benefits from this system? Who is harmed? Are the benefits worth the harms?
- What happens if this system is misused? Can I prevent that?

Reasoning About "Should This Exist?"

When evaluating whether an AI system should be built or deployed, consider:
1. Necessity: Is AI the right solution? Would simpler approaches work? Would human judgment be more appropriate?
2. Proportionality: Do the benefits justify the risks? Who bears the risks vs. who receives the benefits?
3. Reversibility: Can harms be undone? A wrongly rejected job application can be reconsidered; a wrongly denied medical treatment may cause irreversible harm.
4. Alternatives: What happens without this system? Is the status quo better or worse? Are there intermediate options?
5. Consent: Did affected people agree to be subject to this system? Do they have meaningful alternatives?
There are no universal answers. But structured reasoning helps identify the ethical dimensions at stake.

2.12 Generative AI: Earlier Issues Amplified

The preceding sections analyzed ethical challenges that apply to all AI systems. This final section explains why generative models (Chapter 10) and modern deep learning (Chapter 14) make those same challenges significantly harder to manage. The problems are not new, but their scale is.

Scale of influence. A biased hiring algorithm affects individual applicants. A biased generative model produces text or images consumed by millions, shaping cultural norms and public understanding. When a text-to-image system consistently generates scientists as white men, it does not just reflect a biased dataset; it produces new media that reinforces the stereotype. The feedback loop from Section 2.10 now operates at the scale of culture, not just individual decisions.

Training data encompasses everything. Traditional ML systems are trained

on curated datasets collected for a specific task. Large generative models are trained on substantial portions of the internet, inheriting every bias, stereotype, and piece of misinformation present in that data. The curation step that might catch problems in a smaller system is largely absent.

Reduced human oversight. When a classifier flags a loan application, a human reviewer can check the decision. When a generative model produces millions of text completions or images per day, meaningful human review of each output is impossible. Harmful outputs can propagate before anyone notices.

These amplification effects produce several concerns specific to generative systems:

Generative-Specific Ethical Concerns

Deepfakes: Generative models can produce realistic images, audio, or video of real people in fabricated situations. This enables misinformation, fraud, and harassment at a scale that was previously impossible.
Copyright and data ownership: Models trained on billions of creative works can produce outputs that closely resemble specific training examples. Who owns the output? Were the original creators compensated or consulted? These questions remain legally and ethically unresolved.
Environmental cost: Training a single large model can consume as much energy as hundreds of households use in a year. The concentration of compute in a few wealthy organizations raises questions about who bears the environmental cost and who receives the benefits.

The analysis tools from this chapter apply directly. Use the five-point template (DATA, OBJECTIVE, METRIC, HARM, MITIGATION) on any generative system. Ask where in the pipeline bias enters. Identify the feedback loops. The technology is new; the ethical reasoning is the same.

2.13 Summary

Key Takeaways

Bias: AI systems amplify patterns in their training data, including biases and historical discrimination.
Fairness: Multiple definitions of fairness exist (demographic parity, equalized odds, calibration), they mathematically conflict, and choosing between them is an ethical choice, not a technical one.
Privacy: Data collection, de-anonymization risks, and training on personal data raise fundamental privacy concerns. Differential privacy offers mathematical guarantees but involves tradeoffs.
Accountability: As AI systems make more consequential decisions, gaps emerge between the harms they cause and the ability to hold anyone responsible.
Transparency: Black-box models may be more accurate but less interpretable. High-stakes decisions may require explainable AI.
Feedback loops: AI systems shape the world they predict, potentially reinforcing and amplifying existing inequalities over time.
Environmental costs: Training large models consumes enormous energy and concentrates power in wealthy organizations.
Critical questions: Who benefits, whose data, who decides, what is the alternative, who is accountable, and should this system exist at all?

2.14 Practice Problems

Formal Definitions

1. **(Definition)** Define *disparate impact* precisely. If Group A has approval rate p_A and Group B has approval rate p_B, write the mathematical condition for the 80% rule.

2. **(Comparison)** Compare *demographic parity* and *equalized odds*. Give a concrete example where a system satisfies one but not the other.

3. **(Calculation)** A classifier has these confusion matrices:

Group A	Pred +	Pred –
Actual +	80	20
Actual –	10	90

Group B	Pred +	Pred –
Actual +	60	40
Actual –	10	90

   (a) Calculate TPR and FPR for each group.

   (b) Does the system satisfy equalized odds? Justify mathematically.

(c) Does it satisfy demographic parity? Show your calculation.

Proof-Style Reasoning

4. **(Prove)** Show that if base rates differ between groups ($P(Y = 1|G = A) \neq P(Y = 1|G = B)$), then a perfectly calibrated classifier cannot satisfy demographic parity.

   Hint: Start from the definition of calibration: $P(Y = 1|\hat{Y} = 1, G) = \hat{p}$ for all groups.

5. **(Construct)** Create a simple example (2 features, 2 groups, binary outcome) where removing a biased feature *increases* error for both groups. Explain why this happens.

6. **(Analyze)** A hiring model uses: job title, years of experience, salary at previous job (gender is excluded).

   (a) Prove or disprove: excluding gender guarantees the model cannot discriminate by gender.

   (b) Identify which features might act as proxies for gender.

   (c) Propose a statistical test to detect proxy discrimination.

Template Application

7. Apply the five-point template (DATA, OBJECTIVE, METRIC, HARM, MITIGATION) to each:

   (a) A content recommendation system that maximizes watch time

   (b) A predictive policing system that forecasts crime hotspots

   (c) A medical triage AI that prioritizes emergency room patients

Privacy and Accountability

8. **(Privacy)** A fitness app collects GPS data to track running routes. The company claims the data is "anonymized" before being sold to advertisers.

   (a) Explain why GPS data is particularly difficult to truly anonymize.

   (b) Describe how this data could be re-identified.

   (c) Would differential privacy help? What would be the tradeoff?

9. **(Accountability)** A hospital uses an AI system to prioritize patients for

a new treatment. The system denies treatment to a patient who later dies. The hospital claims they "just followed the algorithm."

(a) Identify all parties who might bear responsibility.

(b) What information would you need to determine if the system was at fault?

(c) Should hospitals be allowed to use AI systems they cannot explain? Argue both sides.

10. **(Feedback Loops)** A bank's loan approval AI is trained on historical data showing lower repayment rates in certain zip codes. The AI learns to reject applications from those areas.

(a) Explain how this creates a feedback loop.

(b) After 5 years of deployment, would you expect the disparity to increase, decrease, or stay the same? Why?

(c) Propose an intervention to break the feedback loop.

11. **(Transparency vs. Accuracy)** You must choose between two medical diagnosis systems:

- System A: Decision tree, 85% accuracy, fully interpretable
- System B: Deep neural network, 92% accuracy, black box

(a) For which medical conditions might you prefer System A despite lower accuracy?

(b) For which might you prefer System B?

(c) A patient demands to know why System B flagged them for additional testing. What can you tell them?

12. **(Environmental)** Training a large language model costs €10 million in compute and emits 500 tonnes of CO_2. The model will be used by 100 million people.

(a) Calculate the per-user training cost and carbon footprint.

(b) If each user query costs €0.001 in compute, what is the total inference cost for 1 billion queries?

(c) How would you decide if the benefits justify these costs?

13. **(Generative AI)** Text-to-image systems can generate realistic images of real people in situations that never occurred.

(a) What potential harms does this enable?

(b) What safeguards do current systems implement?

(c) Apply the five-point template (DATA, OBJECTIVE, METRIC, HARM, MITIGATION) to a text-to-image system trained on internet images.

Part II

Core Machine Learning

Chapter 3

Data Preparation and Feature Engineering

> More data beats clever algorithms, but better data beats more data.
>
> Peter Norvig

3.1 Introduction: Why Data Preparation Matters

You have collected a dataset, chosen a promising algorithm, and you are ready to train your first machine learning model. You run the code, wait for the results, and... the model performs terribly. What went wrong?

Nine times out of ten, the problem is not the algorithm. The problem is the data.

Machine learning algorithms are remarkably powerful pattern-finders, but they are also remarkably literal. They find whatever patterns exist in the data you give them. Including patterns you did not intend. If your data contains errors, the algorithm learns those errors. If your data has missing values, the algorithm either crashes or makes up numbers. If one feature is measured in millions while another is measured in decimals, the algorithm pays attention only to the large numbers.

The Golden Rule of Machine Learning

The quality of your model is limited by the quality of your data. "Garbage in, garbage out." No algorithm, no matter how sophisticated, can extract meaningful patterns from bad data.

This chapter covers the essential steps of **data preparation**. Transforming raw, messy data into a clean format that algorithms can learn from effectively.

3.2 Understanding Data Structure

Machine learning works with tabular data: rows represent **objects** (also called instances or samples), and columns represent **attributes** (also called features or variables).

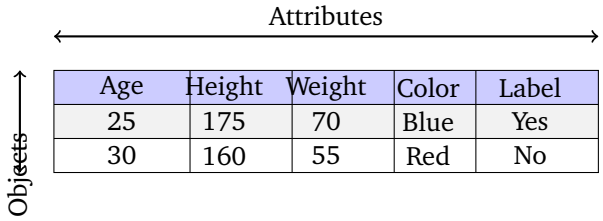


Figure 3.1: Tabular data: rows are objects, columns are attributes.

Numerical attributes take continuous values (age, height, temperature). **Categorical attributes** take values from a fixed set (color, country, size). This distinction determines how you process each attribute.

3.3 Data Quality Problems

Real-world data is messy. Before building any model, you must identify and address quality problems.

Noise refers to random errors that obscure true patterns. Measurement imprecision, data entry errors, and natural variation all contribute noise that makes learning harder.

Outliers are data points dramatically different from others. A single outlier can distort learned patterns severely, one mansion priced at €50 million in a dataset of €300 000 homes can completely skew a regression model.

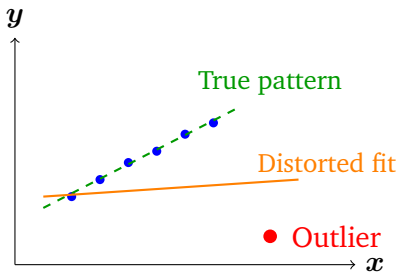


Figure 3.2: One outlier can dramatically distort the learned pattern.

Missing values occur when some attributes are not recorded. You must either delete incomplete rows (losing data) or impute missing values using estimates like the column mean or median.

Duplicates give extra weight to certain examples, skewing learned patterns.

3.4 Feature Scaling

Many algorithms are sensitive to feature scales. If income ranges from €20 000 to €200 000 while age ranges from 18 to 80, income differences will dominate any distance calculations, effectively making the algorithm ignore age.

Standard scaling transforms each feature to have mean 0 and standard deviation 1:

$$\tilde{x}_i = \frac{x_i - \mu}{\sigma}$$

Example: Standard Scaling

Data: $[10, 20, 30, 40, 50]$
Mean: $\mu = 30$, Standard deviation: $\sigma \approx 14.14$
Scaled: $[-1.41, -0.71, 0, 0.71, 1.41]$

Min-max scaling transforms features to range $[0, 1]$:

$$\tilde{x}_i = \frac{x_i - x_{\min}}{x_{\max} - x_{\min}}$$

3.5 Encoding Categorical Variables

Algorithms need numbers, but assigning arbitrary numbers (red=1, blue=2) implies false ordering. **One-hot encoding** converts a categorical feature with k values into k binary columns.

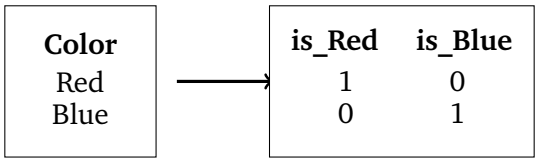


Figure 3.3: One-hot encoding creates binary columns for each category.

3.6 Train-Validation-Test Split

To evaluate how well a model generalizes to new data, we split the dataset into portions with distinct roles.

The Exam Analogy

Think of the **training set** as your textbook, the **validation set** as practice exams, and the **test set** as the final exam. See Chapter 5 for a detailed discussion of evaluation methodology and data splitting.

The training set (typically 70%) is where learning happens. The validation set (15%) is used to tune hyperparameters and make modeling decisions. The test set (15%) is held completely aside until final evaluation. It represents truly unseen data.

The Cardinal Rule

Never train on test data. Never use test data to make decisions. Once you peek at test performance and adjust your model, the test set no longer represents new data. See Chapter 5 for detailed discussion of evaluation methodology.

3.7 Feature Engineering

Raw data rarely comes in the ideal form for machine learning. **Feature engineering** is the process of transforming raw data into features that better represent the underlying patterns, often dramatically improving model performance.

3.7.1 Creating New Features

Sometimes the raw features are not the most predictive representation. Consider predicting house prices:

- Raw features: length, width of the lot

- Better feature: area = length $\times$ width

The area captures what actually matters (how much land) more directly than the raw dimensions.

Common feature engineering techniques:

- **Polynomial features**: Create x^2, $x_1 \cdot x_2$, etc. to capture non-linear relationships
- **Binning**: Convert continuous variables to categories (age $\rightarrow$ age groups)
- **Log transforms**: Compress the scale of highly skewed variables
- **Date/time features**: Extract day of week, month, year, is_weekend from timestamps

3.7.2 Domain Knowledge

The best features often come from understanding the problem domain. A machine learning engineer working on credit scoring should consult with finance experts to understand which ratios (debt-to-income, credit utilization) are meaningful predictors.

Example: Feature Engineering for Fraud Detection

Raw features: transaction amount, merchant category, timestamp
Engineered features:

- Transaction amount relative to user's average
- Time since last transaction
- Number of transactions in last hour
- Distance from user's typical locations
- Whether merchant category is unusual for this user

These engineered features capture behavioral patterns that raw features cannot.

3.7.3 Data Leakage: A Critical Pitfall

Data leakage occurs when information that would not be available at prediction time is used during training. This creates artificially high validation scores that collapse when the model encounters real data. Leakage comes in two distinct forms.

Target Leakage

Target leakage occurs when a feature is derived from, or is a proxy for, the label you are trying to predict.

Example: Target Leakage

You are predicting whether patients will be readmitted to the hospital within 30 days. Your model achieves 99% accuracy on validation data. Then you notice one of your features: `readmission_date`. This field is only populated for patients who *were* readmitted. The feature encodes the label directly.

Other examples:

- Predicting loan default using "number of missed payments" (this *is* default)
- Predicting disease using "prescribed medication for that disease"
- Predicting churn using "cancellation reason"

The pattern: any feature that could only exist *because of* the outcome is target leakage.

Feature Leakage (Train-Test Contamination)

Feature leakage occurs when information from the test set contaminates the training process, even though no individual feature is derived from the label.

Example: Feature Leakage

Common sources of feature leakage:

- Fitting a scaler on the *entire* dataset (including test data) before splitting
- Computing the mean for missing value imputation using all rows, then splitting
- Duplicate or near-duplicate rows appearing in both train and test sets
- Time series data shuffled randomly instead of split chronologically

The pattern: any preprocessing step that allows test-set statistics to influence training transforms is feature leakage. The correct approach is to fit all transformations on the training set only, then apply those same transformations to validation and test sets.

Target Leakage vs. Feature Leakage

	Target Leakage	**Feature Leakage**
What leaks?	Information from label into a feature	Information from test set into training
Cause	Feature derived from outcome	Preprocessing uses future/test data
Detection	"Would I have this before the outcome?"	"Did test data influence training?"
Fix	Remove the feature	Fix preprocessing order

Connection to Bias-Variance (Chapter 5)

Feature engineering directly affects the bias-variance tradeoff:

- **Too few features**: High bias (underfitting), model cannot capture important patterns
- **Too many features**: High variance (overfitting), model memorizes noise
- **Leaked features**: Artificially low training *and* validation error, but catastrophic test error

The goal is features that capture true signal without introducing noise or leakage.

Critical Link: Leakage and Validation Error

Both forms of leakage cause **optimistic validation error**: your cross-validation score looks great, but real-world performance is terrible. Target leakage inflates scores because the model has access to the answer. Feature leakage inflates scores because the validation set is no longer truly independent.
Only the final test set (or real deployment) reveals the truth.
See Chapter 5 for why proper validation methodology is essential and how leakage undermines it.

3.8 Sampling Bias: Problems Before Modeling Begins

Even before you choose an algorithm, your dataset may be flawed because of *how* the data was collected. Sampling bias means the data does not represent the population you want to make predictions about.

3.8.1 Selection Bias

Selection bias occurs when the process of collecting data systematically excludes certain groups or situations. The resulting dataset does not represent the full population.

A medical study conducted only at urban hospitals will underrepresent rural patients. A survey conducted only in English will miss non-English speakers. An AI trained on data from one country may fail in another.

3.8.2 Survivorship Bias

Survivorship bias occurs when your data only contains examples that "survived" some selection process, and you draw conclusions as if they were representative of all cases.

Classic Example: Survivorship Bias

During World War II, engineers studied bullet holes in planes returning from combat and proposed reinforcing the most-hit areas. Statistician Abraham Wald pointed out the flaw: they were only seeing planes that *survived*. The planes that were hit in other areas never came back. The military should reinforce the areas with *no* bullet holes on surviving planes.

In machine learning, survivorship bias appears when you train on data from customers who stayed (ignoring those who left), companies that succeeded (ignoring those that failed), or patients who completed a study (ignoring those who dropped out).

3.8.3 Convenience Sampling

Convenience sampling means collecting data from whatever source is easiest to access. Training a facial recognition system on photos of university students (because they are nearby) produces a model biased toward young, educated, campus-dwelling populations. Training a language model on internet text overrepresents English-speaking, technically literate users.

Most large-scale AI training datasets suffer from some degree of convenience sampling.

3.8.4 Non-Response Bias

When you collect data through surveys or voluntary participation, the people who respond may differ systematically from those who do not. Product reviews skew toward people who feel strongly (either very satisfied or very

dissatisfied). Medical studies lose participants who become too ill to continue. Job satisfaction surveys underrepresent those who already quit.

IAIO Connection: Identifying Sampling Bias

IAIO problems may present a dataset and ask you to identify potential sampling biases. For each dataset, ask:

- Who or what is *missing* from this data?
- What process generated this data, and who does that process exclude?
- Would results change if the missing data were included?

3.9 Distribution Shift: When the World Changes

A model trained on one distribution of data may encounter a different distribution in deployment. This is called **distribution shift**, and it is one of the most common reasons deployed models fail.

3.9.1 Covariate Shift

The input distribution changes, but the relationship between inputs and outputs stays the same.

Example: A model trained to diagnose diseases from X-rays at Hospital A is deployed at Hospital B, which uses different X-ray machines. The images look different (shifted inputs), but the underlying relationship between pathology and image features has not changed. The model may still fail because it learned features specific to Hospital A's equipment.

Formally: $\boldsymbol{P}_{\text{train}}(\boldsymbol{X}) \neq \boldsymbol{P}_{\text{deploy}}(\boldsymbol{X})$, but $\boldsymbol{P}(\boldsymbol{Y}|\boldsymbol{X})$ remains the same.

3.9.2 Label Shift

The distribution of outcomes changes, but the relationship from outcomes to features stays the same.

Example: A spam filter trained when 5% of emails were spam is deployed when 40% are spam. The filter's threshold, calibrated for a 5% spam rate, will miss many spam emails. The spam itself looks the same; there is just more of it.

Formally: $\boldsymbol{P}_{\text{train}}(\boldsymbol{Y}) \neq \boldsymbol{P}_{\text{deploy}}(\boldsymbol{Y})$, but $\boldsymbol{P}(\boldsymbol{X}|\boldsymbol{Y})$ remains the same.

3.9.3 Concept Drift

The fundamental relationship between inputs and outputs changes over time.

Example: A model predicting "creditworthy" borrowers was trained before a major recession. During the recession, the same income and employment features no longer predict repayment in the same way because the economy itself has changed. The concept of "creditworthy" has shifted.

Formally: $P_{\text{train}}(Y|X) \neq P_{\text{deploy}}(Y|X)$.

Three Types of Distribution Shift

Type	What changes	Example
Covariate shift	Input distribution $P(X)$	New hospital, different equipment
Label shift	Output distribution $P(Y)$	Spam rate increases
Concept drift	Relationship $P(Y\|X)$	Recession changes default patterns

Concept drift is the most dangerous because the model's learned rules become wrong, not just miscalibrated. Models deployed in changing environments need ongoing monitoring.

3.10 When Is a Test Set Valid?

A test set measures generalization: how well the model will perform on new data. But this measurement is only meaningful if the test set represents the data the model will actually encounter in deployment.

3.10.1 The Representativeness Requirement

If you train and test on hospital data from 2020, your test accuracy tells you how well the model performs on *2020 hospital data*. It says nothing about performance in 2025, at a different hospital, in a different country, or on a different patient population.

A test set is valid only if:

- It was drawn from the same distribution the model will encounter in deployment
- It was not used to make any modeling decisions (no peeking)
- It is large enough that the accuracy estimate is statistically reliable

3.10.2 When Test Sets Lie

A Test Set Can Give a False Sense of Security

Consider a model trained and tested on data from one hospital. The test accuracy is 95%. The model is then deployed at a second hospital, where accuracy drops to 70%.
What went wrong? The test set was not *wrong*, it accurately measured performance on that hospital's data. But it was *misleading* because it did not represent the deployment setting.
A test score is not a guarantee. It is an estimate, valid only under the assumption that deployment data resembles test data.

3.10.3 Temporal Splits

For time-dependent data, random splitting is invalid. If you randomly shuffle stock prices from 2020–2024 into train and test sets, the model can use future prices to predict past ones.

The correct approach: train on earlier data, test on later data. This simulates the real deployment scenario, where you always predict the future using only the past.

3.10.4 Connecting the Pieces

Sampling bias, distribution shift, and test set validity are all facets of the same problem: *does your data represent the situation where the model will be used?*

- Sampling bias means the training data was not representative from the start
- Distribution shift means the world changed after the data was collected
- An invalid test set means you cannot even measure the problem

3.11 Summary

Key Takeaways

Data Quality: Address noise, outliers, missing values, and duplicates before modeling.
Feature Scaling: Use standard or min-max scaling so all features contribute equally.
Categorical Encoding: Use one-hot encoding to avoid implying false ordering.
Data Splits: Training for learning, validation for tuning, test for final evaluation. Never peek at test data.
Feature Engineering: Transform raw data into features that better represent the underlying patterns. Domain knowledge is invaluable.
Leakage: Target leakage (features derived from the label) and feature leakage (test data contaminating training) both produce falsely optimistic results.
Sampling Bias: Selection bias, survivorship bias, convenience sampling, and non-response bias can all produce unrepresentative datasets.
Distribution Shift: Covariate shift, label shift, and concept drift cause models to fail when the world changes after training.
Test Set Validity: A test score is only meaningful if the test set represents the deployment setting.

3.12 Practice Problems

1. Given data $[\mathbf{2, 4, 6, 8, 10}]$, calculate the mean and standard deviation, then apply standard scaling.

2. A feature "Size" has values {Small, Medium, Large}. How many columns does one-hot encoding create? What is the encoding for "Medium"?

3. With 5000 samples and a 70-15-15 split, how many samples are in each set?

4. You have a dataset with columns: timestamp, user_id, purchase_amount, product_category. Suggest three engineered features that might help predict whether a user will make a purchase next week.

5. A dataset has 10% missing values in the "income" column. Describe two different approaches to handling this, and discuss when you might prefer each.

6. Why might applying a log transform to a "house price" feature improve model performance?

7. **(Adversarial thinking)** You receive a dataset where applying standard preprocessing achieves 98% accuracy, but the model fails completely on new real-world data. Describe three specific ways the preprocessing pipeline could have introduced this problem.

8. **(What-if)** A dataset has features with very different scales: age (0–100), income ($0–$1,000,000), and number of children (0–10).

   (a) What problems might arise if you train a k-NN classifier without scaling?

   (b) What problems might arise if you train a decision tree without scaling?

   (c) Based on your answers, which algorithms are "scale-invariant" and why?

9. **(Multi-part)** A dataset has 10% missing values in the "income" column.

   (a) Describe three different strategies for handling these missing values.

   (b) For each strategy, describe a scenario where it would be the best choice.

   (c) For each strategy, describe a scenario where it would be harmful.

10. **(Feature Leakage Detection)** You're predicting customer churn (whether a customer will cancel their subscription). Your colleague proposes these features:

    (a) Days since last login

    (b) Total purchases in lifetime

    (c) "Cancellation reason" (text field)

    (d) Average session duration

    (e) Number of support tickets filed

    Which feature(s) might be leakage? For each potential leak, explain why and how you would verify.

11. **(Adversarial: When NOT to Act)** You're building a model to predict house prices. A colleague is concerned about these features:

    (a) Square footage of the house

(b) Number of bedrooms

(c) Year the house was built

(d) Latitude and longitude coordinates

They worry these might be "leakage" because they correlate strongly with price. Explain why these are **not** leakage and should be kept in the model. What distinguishes legitimate predictive features from actual leakage?

12. **(Target vs. Feature Leakage)** Classify each of the following as target leakage, feature leakage, or no leakage. Justify each answer.

    (a) Predicting exam scores using "number of hours spent studying" (collected before the exam)

    (b) Predicting exam scores using "student's satisfaction with their grade" (collected after)

    (c) Using the mean of the entire dataset (train + test) to impute missing values

    (d) Predicting hospital readmission using "length of initial stay" (known at discharge)

13. **(Sampling Bias)** For each scenario, identify the type of sampling bias and explain what population is underrepresented:

    (a) A self-driving car trained on dashcam footage from California

    (b) A sentiment analysis model trained on product reviews

    (c) A study on the effectiveness of a new drug, where 30% of participants dropped out before completion

    (d) A dataset of "successful startups" used to predict which new startups will succeed

14. **(Distribution Shift)** A model predicting taxi demand in New York City was trained on 2019 data and deployed in March 2020.

    (a) Which type(s) of distribution shift occurred? Justify your answer.

    (b) Would retraining on January–February 2020 data help? Why or why not?

    (c) Propose a monitoring strategy that would detect this failure early.

15. **(Test Set Validity)** A team trains a skin cancer detection model on images from dermatology clinics in Northern Europe. The test set is a

random 15% split from the same clinics. Test accuracy is 94%.

(a) The model is deployed in clinics in Sub-Saharan Africa. Why might performance drop?

(b) Is the 94% test accuracy "wrong"? Explain what it does and does not tell you.

(c) How should the team have constructed their test set if they knew the model would be deployed globally?

Chapter 4

Supervised Learning

> All models are wrong, but some are useful.
>
> George Box

4.1 Introduction to Supervised Learning

Imagine teaching a child to recognize animals. You show them pictures one by one: “This is a cat,” “This is a dog,” “This is a cat,” “This is a dog.” After seeing enough examples, something remarkable happens. When shown a completely new picture (an animal they have never seen before) the child can correctly identify whether it is a cat or a dog. They have learned to generalize from specific examples to a broader concept.

This is precisely how **supervised learning** works. We provide an algorithm with a collection of examples where we already know the correct answers, and the algorithm discovers the underlying pattern that connects the inputs to the outputs. Once it has learned this pattern, it can make predictions about new examples it has never encountered.

The term “supervised” comes from the idea that we are supervising the learning process by providing the correct answers. Think of it like a teacher grading homework: for each problem, the student (the algorithm) can see both the question and the correct answer, learning from any mistakes along the way.

4.1.1 The Structure of Supervised Learning

In supervised learning, our data consists of input-output pairs. The inputs are called **features**, they are the pieces of information we use to make predictions. For an image, the features might be the pixel values. For a house, the features might include square footage, number of bedrooms, and location. For a patient, the features might include age, blood pressure, and cholesterol level.

The outputs are called **labels**, they are what we want to predict. For the image, the label might be "cat" or "dog." For the house, the label might be its price. For the patient, the label might be whether they have a particular disease.

The collection of input-output pairs we use to train our algorithm is called the **training set**. This is the data from which the algorithm learns. Separately, we keep a **test set**, data that the algorithm never sees during training. Which we use to evaluate how well the algorithm has learned to generalize.

Key Terminology

Features (denoted $\mathbf{x}$): The input variables used to make predictions. Also called predictors, attributes, or independent variables.
Label (denoted $\boldsymbol{y}$): The output variable we want to predict. Also called the target, response, or dependent variable.
Training set: The collection of (feature, label) pairs used to train the model. The algorithm learns from these examples.
Test set: Data held out from training, used to evaluate how well the model generalizes to new, unseen examples.

4.1.2 Why Supervised Learning Works

The fundamental assumption underlying supervised learning is that the data contains patterns. Regularities that connect inputs to outputs. If house prices were completely random and had nothing to do with size, location, or any other measurable feature, no algorithm could predict them. But in reality, larger houses tend to cost more, houses in desirable neighborhoods tend to cost more, and so on. These patterns exist in the data, waiting to be discovered.

A supervised learning algorithm's job is to find a function $\boldsymbol{h}$ (called the **hypothesis**) that maps inputs to outputs: given features $\mathbf{x}$, the function produces a prediction $\boldsymbol{h}(\mathbf{x})$ that should be close to the true label $\boldsymbol{y}$. The better the algorithm discovers the true underlying pattern, the more accurate its predictions will be on new data.

4.2 Classification vs. Regression

Supervised learning problems come in two main flavors, depending on the nature of what we are trying to predict.

4.2.1 Regression: Predicting Continuous Values

In **regression** problems, the label is a continuous real number. It can take on any value within some range. When you predict tomorrow's temperature, you might get 22.4°C or 20.4°C or any other number. When you predict a house price, you might get €247 500 or €312 000 or anything in between.

Common regression problems include predicting house prices based on their characteristics, forecasting stock prices or sales figures, estimating a person's age from their photograph, and predicting how long a machine will operate before failing. In each case, the output is a number that can vary continuously.

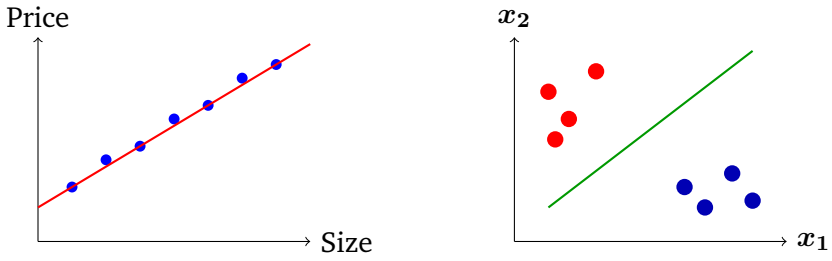
The key characteristic of regression is that small changes in the input can lead to small changes in the output. A house that is slightly larger might be predicted to cost slightly more. This smooth relationship between inputs and outputs is what makes regression different from classification.

4.2.2 Classification: Predicting Categories

In **classification** problems, the label is a discrete category. It belongs to one of a finite set of possible classes. When you classify an email, it is either spam or not spam; there is no "37% spam." When you diagnose a disease, the patient either has it or does not. When you recognize a handwritten digit, it is 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9, there is no "digit 3.7."

When there are exactly two possible classes, we call it **binary classification**. Examples include spam detection (spam vs. not spam), medical diagnosis (disease vs. no disease), and fraud detection (fraudulent vs. legitimate). When there are more than two classes, we call it **multi-class classification**. Examples include handwritten digit recognition (10 classes), identifying animal species (many classes), and categorizing news articles by topic.

The fundamental difference from regression is that small changes in input can lead to abrupt changes in output. An email that is very similar to spam might still be classified as not spam if it falls on the right side of the decision boundary. This discrete, all-or-nothing nature of classification requires different algorithms and evaluation methods than regression.



Regression
Predict a continuous value

Classification
Predict a discrete category

Figure 4.1: In regression, we fit a line (or curve) to predict continuous values. In classification, we find a boundary that separates different categories.

4.3 Linear Regression

4.3.1 Loss Functions: What the Algorithm Optimizes

Every supervised learning algorithm works by minimizing a **loss function**: a measure of how wrong its predictions are. The choice of loss function determines what kinds of errors the model cares about most, and different tasks require different losses.

Squared loss (used by regression) penalizes large errors quadratically:

$$L(y, \hat{y}) = (y - \hat{y})^2$$

A prediction that is off by 10 is penalized 100 times more than a prediction off by 1. This makes regression models sensitive to outliers: a single wildly wrong prediction dominates the loss.

Log loss / cross-entropy (used by logistic regression) measures how well predicted probabilities match actual classes:

$$L(y, \hat{p}) = -[y \log(\hat{p}) + (1 - y) \log(1 - \hat{p})]$$

A confident wrong prediction (predicting 0.99 when the true label is 0) incurs enormous loss. A confident correct prediction incurs almost none. This drives classifiers to produce well-separated probabilities.

Hinge loss (used by SVMs) only cares whether the prediction is on the correct side of the margin:

$$L(y, f(x)) = \max(0, 1 - y \cdot f(x))$$

If the prediction is correct and confident (far from the boundary), hinge loss is zero. It only penalizes predictions that are wrong or too close to the

boundary. This is why SVMs produce maximum-margin classifiers: hinge loss explicitly rewards large margins.

Why the Loss Function Matters

The loss function shapes the model's behavior:

- **Squared loss**: Sensitive to outliers; predictions are continuous
- **Log loss**: Drives predictions toward 0 or 1; outputs are probabilities
- **Hinge loss**: Ignores "easy" examples; focuses on the decision boundary

Choosing the wrong loss for your task can produce a model that optimizes for the wrong thing.

4.3.2 The Hypothesis and Cost Function

Linear regression is the simplest and most foundational algorithm in supervised learning. The core idea: if there is a straight-line relationship between input and output, find that line.

For predicting house prices from size, we model the relationship as:

$$h_\theta(x) = \theta_0 + \theta_1 x \tag{4.1}$$

This is simply a straight line: θ_0 is the y-intercept, θ_1 is the slope. Different parameter values give different lines; the algorithm finds the values that best fit the training data.

To measure how well a line fits, we use the **Mean Squared Error (MSE)**:

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 \tag{4.2}$$

This **cost function** measures how poorly predictions match actual values. Our goal is to minimize it.

4.3.3 Gradient Descent

Gradient descent finds the minimum by iteratively stepping in the direction that reduces cost:[1]

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \tag{4.3}$$

[1] See Appendix D for partial derivatives and the chain rule.

The **learning rate** α controls step size. Too small: slow convergence. Too large: overshoot or diverge.

A key property: linear regression's cost function is **convex**, shaped like a bowl with a single minimum. Gradient descent is guaranteed to find it.

4.3.4 Multiple Features

So far, we have considered predicting house prices from size alone. But in reality, many features affect price: the number of bedrooms, the age of the house, the quality of the neighborhood, and so on. Linear regression easily extends to multiple features:

$$h_\theta(\mathrm{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n \tag{4.4}$$

Here, $x_1, x_2, \ldots, x_n$ are the n different features, and each has its own parameter θ_j determining how much that feature contributes to the prediction. The parameter θ_0 is still the intercept.

Using vector notation, we can write this more compactly as $h_\theta(\mathrm{x}) = \theta^T \mathrm{x}$, where θ and x are both vectors. This notation becomes essential when working with many features.

4.3.5 Polynomial Regression

What if the relationship between input and output is not a straight line? Perhaps house prices increase faster for larger houses, suggesting a curved relationship. We can capture this by adding polynomial features:

$$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \cdots \tag{4.5}$$

Interestingly, this is still called "linear" regression because the hypothesis is linear in the parameters θ. We have simply created new features (x^2, x^3, etc.) from the original feature x. The same gradient descent algorithm works to find the optimal parameters.

The power of polynomial regression comes with a warning: high-degree polynomials can fit the training data very closely but may perform poorly on new data. This problem, called overfitting, will be addressed later in this chapter.

4.4 Logistic Regression

Despite its name containing "regression," logistic regression is actually a **classification** algorithm. It is used when the output is categorical rather

than continuous, most commonly for binary classification where we want to predict one of two classes.

4.4.1 Why Linear Regression Fails for Classification

You might wonder: why not just use linear regression for classification? Suppose we label spam emails as 1 and legitimate emails as 0. We could train a linear regression model and then classify emails as spam if the prediction is greater than 0.5.

The problem is that linear regression can produce predictions less than 0 or greater than 1. It might predict $-\mathbf{0.3}$ for one email and $\mathbf{1.7}$ for another, which do not make sense as probabilities. Furthermore, a single outlier can dramatically shift the regression line, changing classifications for many other examples.

What we need is a function that always outputs values between 0 and 1, which we can interpret as probabilities. This is exactly what logistic regression provides.

4.4.2 The Sigmoid Function

The key ingredient in logistic regression is the **sigmoid function** (also called the logistic function):

$$\sigma(z) = \frac{\mathbf{1}}{\mathbf{1} + e^{-z}} \tag{4.6}$$

This function has a beautiful S-shaped curve. No matter what value of z you input, the output is always between 0 and 1. When z is a large positive number, $\sigma(z)$ is close to 1. When z is a large negative number, $\sigma(z)$ is close to 0. When $z = \mathbf{0}$, $\sigma(z) = \mathbf{0.5}$ exactly.

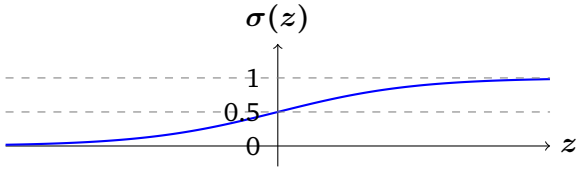


Figure 4.2: The sigmoid function smoothly maps any real number to a value between 0 and 1, making it perfect for representing probabilities.

4.4.3 The Logistic Regression Hypothesis

In logistic regression, we take the linear combination of features (just like in linear regression) and pass it through the sigmoid function:

$$h_\theta(\mathbf{x}) = \sigma(\theta^T \mathbf{x}) = \frac{1}{1 + e^{-\theta^T \mathbf{x}}} \tag{4.7}$$

The output $h_\theta(\mathbf{x})$ is now always between 0 and 1, and we interpret it as the probability that the input belongs to class 1. Specifically, $h_\theta(\mathbf{x}) = P(y = 1|\mathbf{x}; \theta)$, the probability that y equals 1, given the features $\mathbf{x}$ and parameters θ.

To make a final classification, we use a threshold, typically 0.5:

$$\text{Predict } y = \begin{cases} 1 & \text{if } h_\theta(\mathbf{x}) \geq 0.5 \\ 0 & \text{if } h_\theta(\mathbf{x}) < 0.5 \end{cases} \tag{4.8}$$

This means we predict class 1 if the model believes there is at least a 50% chance of class 1. The threshold can be adjusted depending on the application, for example, in medical diagnosis, we might use a lower threshold to avoid missing diseases.

4.4.4 The Decision Boundary

The **decision boundary** is the surface in feature space where the prediction switches from one class to the other. For logistic regression with two features, this boundary is where $h_\theta(\mathbf{x}) = 0.5$, which happens exactly when $\theta^T \mathbf{x} = 0$.

With two features x_1 and x_2, the equation $\theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0$ defines a straight line. Points on one side are classified as class 0; points on the other side are classified as class 1. The algorithm's job is to find the line (determined by θ_0, θ_1, θ_2) that best separates the two classes.

4.4.5 The Cost Function for Logistic Regression

We cannot use mean squared error for logistic regression because the resulting cost function would be non-convex, with many local minima where gradient descent could get stuck.

Instead, we use **binary cross-entropy loss**, which has a beautiful intuition. If the true label is $y = 1$, we want $h_\theta(\mathbf{x})$ to be close to 1. The term $-\log(h_\theta(\mathbf{x}))$ is small when $h_\theta(\mathbf{x})$ is near 1, but becomes very large as $h_\theta(\mathbf{x})$ approaches 0. Similarly, if $y = 0$, the term $-\log(1 - h_\theta(\mathbf{x}))$ penalizes predictions near 1.

Combining these cases:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log(h_\theta(x^{(i)})) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\right] \tag{4.9}$$

This cost function is convex, so gradient descent finds the global minimum. The update rules look similar to linear regression, and many software libraries make training logistic regression straightforward.

4.5 Regularization: Preventing Overfitting

One of the most important concepts in machine learning is the distinction between fitting the training data and generalizing to new data. A model that performs wonderfully on training data but poorly on new data has **overfit**, it has memorized the training examples rather than learning the underlying pattern.

4.5.1 Understanding Overfitting

Imagine fitting a polynomial to a small dataset. A straight line (degree 1) might not capture the pattern well, this is **underfitting**. A degree-2 polynomial might fit nicely. But a degree-20 polynomial could pass exactly through every training point, achieving zero training error. This sounds great, but the polynomial would wiggle wildly between the points, making terrible predictions for new data.

The degree-20 polynomial has too many parameters relative to the amount of training data. It has enough flexibility to fit not just the true pattern but also the random noise in the data. This is overfitting.

4.5.2 Regularization: Adding a Penalty for Complexity

Regularization prevents overfitting by adding a penalty term to the cost function that discourages large parameter values. The intuition is that extreme parameter values often indicate the model is trying too hard to fit the training data.

L2 Regularization (also called Ridge regression) adds the sum of squared parameters:

$$J_{\text{L2}}(\theta) = J(\theta) + \lambda\sum_{j=1}^{n}\theta_j^2 \tag{4.10}$$

The hyperparameter λ controls the strength of regularization. A larger λ applies more penalty, pushing parameters toward smaller values. When

$\lambda = 0$, there is no regularization. Note that we typically do not regularize θ_0 (the intercept).

L2 regularization shrinks all parameters toward zero, but they rarely become exactly zero. The model retains all features but reduces their influence.

L1 Regularization (also called Lasso) adds the sum of absolute values:

$$J_{\text{L1}}(\theta) = J(\theta) + \lambda \sum_{j=1}^{n} |\theta_j| \tag{4.11}$$

L1 regularization has a remarkable property: it tends to make some parameters exactly zero. This performs automatic feature selection, unimportant features are eliminated entirely. This can be valuable when you have many features and want to identify which ones truly matter.

Choosing the right value of λ is crucial. Too small, and regularization has little effect; too large, and the model becomes too simple, underfitting the data. Cross-validation (discussed later) is typically used to select λ.

Conceptual Pause: Linear Models Compared

Before moving to more complex algorithms, let us consolidate what we have learned about linear models.

Method	Use When	Strengths	Weaknesses
Linear Regression	Predicting continuous values; relationship is approximately linear	Simple, interpretable, fast; closed-form solution	Cannot capture nonlinear patterns
Logistic Regression	Binary/multi-class classification; want probabilities	Outputs calibrated probabilities; works with many features	Assumes linear decision boundary
Ridge (L2)	Many features; want to prevent overfitting	Shrinks all coefficients; stable	No feature selection
Lasso (L1)	Many features; want sparsity	Automatic feature selection	May select arbitrarily among correlated features

4.6 Support Vector Machines

Support Vector Machines (SVMs) approach classification from a different perspective: instead of modeling probabilities like logistic regression, they focus on finding the boundary that separates classes with the **maximum margin**.

4.6.1 The Maximum Margin Idea

Consider a binary classification problem where the two classes are linearly separable. Meaning we can draw a straight line (or hyperplane in higher dimensions) that perfectly separates them. There are infinitely many such lines. Which one should we choose?

SVMs choose the line that is farthest from the nearest training points of each class. This distance is called the **margin**. The intuition is that a line with a large margin is more robust. Small perturbations or noise in the data are less likely to cause misclassifications.

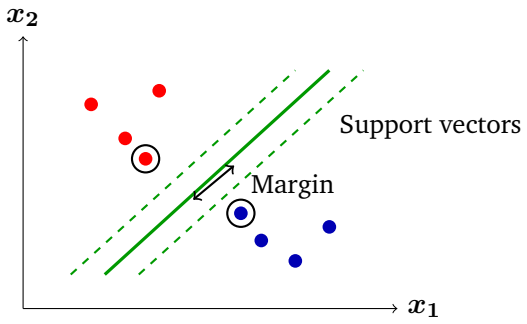


Figure 4.3: The SVM finds the decision boundary (solid line) that maximizes the margin (distance to the dashed lines). The circled points are support vectors. The training examples closest to the boundary.

4.6.2 Support Vectors

The points that lie exactly on the margin boundaries are called **support vectors**. These are the training examples that "support" the decision boundary. If you removed any other training point, the optimal boundary would not change. Only the support vectors matter for determining the classifier.

This is a beautiful property: the SVM solution depends only on a subset of the training data. In contrast, methods like logistic regression use all training points to determine the parameters.

4.6.3 Soft Margins and the Parameter C

In practice, data is rarely perfectly separable. There might be a few points from one class mixed in with the other, or the classes might overlap significantly. A "hard margin" SVM that requires perfect separation would fail on such data.

Soft margin SVMs allow some points to violate the margin or even be on the wrong side of the boundary. The optimization becomes a trade-off: we

want a large margin, but we also want to minimize misclassifications. The parameter C controls this trade-off:

$$\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{m} \xi_i \tag{4.12}$$

Here, ξ_i represents how much point i violates the margin (zero if it does not violate). A large C heavily penalizes violations, leading to a smaller margin but fewer misclassifications. This might overfit noisy data. A small C tolerates more violations in exchange for a larger margin. This might underfit if the data is cleanly separable.

4.6.4 Kernel Methods: Going Non-Linear

What if no straight line can separate the classes? SVMs handle this through the **kernel trick**, one of the most elegant ideas in machine learning.

The kernel trick implicitly maps the data to a higher-dimensional space where linear separation might be possible. For example, data that forms two concentric circles in 2D cannot be separated by a line. But if we add a third dimension $z = x_1^2 + x_2^2$ (the distance from the origin squared), the two circles become two parallel planes that *can* be separated by a hyperplane.

The remarkable part is that we never actually compute this high-dimensional mapping. Instead, we use a **kernel function** $K(\mathbf{x}, \mathbf{z})$ that computes the dot product in the higher-dimensional space directly from the original features. Common kernels include:

The **linear kernel** $K(\mathbf{x}, \mathbf{z}) = \mathbf{x}^T\mathbf{z}$ corresponds to no transformation, it gives a standard linear SVM.

The **polynomial kernel** $K(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^T\mathbf{z} + c)^d$ corresponds to polynomial features of degree d.

The **RBF (Gaussian) kernel** $K(\mathbf{x}, \mathbf{z}) = \exp(-\gamma\|\mathbf{x} - \mathbf{z}\|^2)$ is the most popular, corresponding to an infinite-dimensional space. It can create arbitrarily complex decision boundaries.

4.7 Decision Trees

Decision trees take a completely different approach to classification: they learn a series of if-then rules that partition the feature space into regions, each assigned to a class. The result is highly interpretable, you can trace exactly why any prediction was made.

4.7.1 The Tree Structure

A decision tree is built from three types of nodes. The **root node** is at the top and contains the first decision. **Internal nodes** represent subsequent decisions as you move down the tree. **Leaf nodes** are at the bottom and contain the final class predictions.

At each internal node, the tree asks a question about one feature, typically of the form "Is feature x_j greater than some threshold?" Based on the answer, we follow one branch or the other. Eventually, we reach a leaf node that tells us the predicted class.

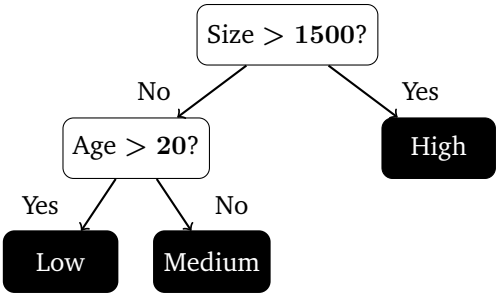


Figure 4.4: A simple decision tree for predicting house price category. Starting at the root, we follow branches based on the answers until reaching a leaf with the prediction.

4.7.2 Choosing the Best Splits

The key question in building a decision tree is: at each node, which feature should we split on, and at what threshold? We want splits that best separate the classes.

To measure how "mixed" or "impure" a set of examples is, we use **entropy**:

$$H(S) = -\sum_{c} p_c \log_2(p_c) \tag{4.13}$$

Here, p_c is the proportion of examples in class c. If all examples belong to one class ($p_c = 1$ for that class), entropy is 0. There is no uncertainty about the class. If examples are evenly split between two classes ($p_c = 0.5$ for each), entropy is 1. Maximum uncertainty.

A good split should reduce entropy, after splitting, the resulting subsets should be more pure than the original set. **Information gain** measures this reduction:

$$\text{Gain}(S, A) = H(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} H(S_v) \tag{4.14}$$

This computes the entropy before the split minus the weighted average entropy after the split. The attribute with the highest information gain is chosen for the split.

Example: Computing Information Gain

Consider a dataset with 10 examples: 5 positive and 5 negative. The entropy is $H(S) = -0.5\log_2(0.5) - 0.5\log_2(0.5) = 1.0$ (maximum uncertainty).
Suppose splitting on attribute A produces two branches: one with 4 positive and 1 negative, another with 1 positive and 4 negative.
For the first branch: $H = -\frac{4}{5}\log_2\frac{4}{5} - \frac{1}{5}\log_2\frac{1}{5} \approx 0.72$
For the second branch: same calculation gives $H \approx 0.72$
Weighted average entropy after split: $\frac{5}{10}(0.72) + \frac{5}{10}(0.72) = 0.72$
Information Gain $= 1.0 - 0.72 = 0.28$
This split reduces uncertainty by 0.28 bits.

4.7.3 Building the Tree: The ID3 Algorithm

The classic algorithm for building decision trees, called ID3, works recursively:

First, if all examples in the current set have the same class, create a leaf node with that class label. Second, if there are no more attributes to split on, create a leaf node with the majority class. Otherwise, select the attribute with the highest information gain, create a node for that attribute, and for each possible value of the attribute, recursively build a subtree for the examples with that value.

This greedy approach builds the tree top-down, making the locally optimal choice at each step. While it does not guarantee the globally optimal tree, it works well in practice.

4.7.4 Pruning: Avoiding Overfitting

Decision trees have a tendency to overfit, especially when grown deep. A tree with many levels can create very specific rules that fit the training data perfectly but fail to generalize.

Pruning addresses this by removing branches that do not improve performance on validation data. Pre-pruning stops growing the tree early, while post-pruning grows the full tree and then removes unhelpful branches. Parameters like maximum depth, minimum samples per leaf, and minimum information gain for splitting help control tree complexity.

4.7.5 Ensemble Methods: Combining Multiple Trees

A single decision tree is prone to overfitting and high variance: small changes in the training data can produce a very different tree. **Ensemble methods** address this weakness by combining many trees into a single, more robust predictor.

Bagging (Bootstrap Aggregating) trains many trees on random subsamples of the training data (drawn with replacement) and averages their predictions. Because each tree sees a slightly different dataset, they make different errors, and averaging reduces variance.

A **Random Forest** extends bagging by also randomizing the features considered at each split. At each node, only a random subset of features is evaluated, forcing the trees to be diverse. Random Forests are among the most reliable off-the-shelf classifiers in practice: they handle non-linear patterns, require little tuning, resist overfitting, and work well on both small and large datasets.

Boosting takes a different approach: it trains trees sequentially, where each new tree focuses on the examples that previous trees got wrong. **AdaBoost** increases the weight of misclassified examples before training the next tree. **Gradient Boosting** fits each new tree to the residual errors of the ensemble so far. Boosting often achieves higher accuracy than bagging but is more sensitive to noisy data and requires more careful tuning.

Key Idea: Why Ensembles Work

A single tree is like asking one person for directions: you might get a great answer or a terrible one. An ensemble is like asking many people and taking a vote. Individual errors cancel out, and the consensus is usually more reliable than any single opinion.

4.8 Naïve Bayes Classification

2

Naïve Bayes is a probabilistic classifier based on Bayes' theorem with a simplifying assumption that makes computation tractable. Despite its simplicity, it often performs surprisingly well, especially for text classification.

[2] See Appendix C for a review of Bayes' theorem and conditional probability.

4.8.1 Bayes' Theorem

Bayes' theorem relates the probability of a hypothesis given evidence to the probability of evidence given the hypothesis:

$$P(y|\mathbf{x}) = \frac{P(\mathbf{x}|y) \cdot P(y)}{P(\mathbf{x})} \tag{4.15}$$

In classification terms: $P(y|\mathbf{x})$ is the probability of class y given features $\mathbf{x}$ (what we want). $P(\mathbf{x}|y)$ is the probability of seeing features $\mathbf{x}$ given class y (the likelihood). $P(y)$ is the prior probability of class y before seeing any features. $P(\mathbf{x})$ is the probability of the features, which serves as a normalizing constant.

To classify, we compute $P(y|\mathbf{x})$ for each possible class y and predict the class with the highest probability.

4.8.2 The Naïve Independence Assumption

The challenge is computing $P(\mathbf{x}|y)$, the probability of a particular combination of features given a class. With many features, the number of possible combinations is enormous.

The "naïve" assumption solves this by assuming that features are conditionally independent given the class. In other words, once we know the class, knowing the value of one feature tells us nothing about the values of other features. This allows us to factor the probability:

$$P(\mathbf{x}|y) = P(x_1|y) \cdot P(x_2|y) \cdot \ldots \cdot P(x_n|y) = \prod_{i=1}^{n} P(x_i|y) \tag{4.16}$$

Each $P(x_i|y)$ can be estimated directly from the training data by counting how often each feature value appears in each class.

4.8.3 Classification Rule

The final classification chooses the class that maximizes the posterior probability:

$$\hat{y} = \arg\max_y P(y) \prod_{i=1}^{n} P(x_i|y) \tag{4.17}$$

Since $P(\mathbf{x})$ is the same for all classes, we can ignore it and just compare the numerators.

The independence assumption is almost always false, features are usually correlated. Yet Naïve Bayes often works remarkably well. One reason is that

classification only requires ranking the classes correctly, not computing exact probabilities. Even if the probabilities are wrong, the ranking might still be right.

Naïve Bayes is particularly popular for text classification (spam filtering, sentiment analysis) where features are word counts. It is fast to train, handles high-dimensional data well, and provides a strong baseline.

4.9 Algorithm Comparison

Choosing the right algorithm requires understanding their assumptions, strengths, and failure modes.

Algorithm	**Assumptions**	**Failure Modes**	**Strengths**	**Interpretable?**
Linear Regression	Linear relationship; homoscedastic errors	Non-linear patterns; outliers distort fit	Fast; closed-form solution; coefficients meaningful	Yes
Logistic Regression	Linear decision boundary; features independent-ish	Non-linear boundaries; multicollinearity	Probabilistic output; regularization easy	Yes
Decision Tree	None (non-parametric)	Overfits with depth; axis-aligned only	Handles mixed types; no scaling needed	Yes
SVM (linear)	Linearly separable (soft margin helps)	Very non-linear data; large datasets slow	Max-margin = good generalization	Somewhat
SVM (RBF)	Data has local structure	Too many features; sensitive to γ, C	Captures complex boundaries	No
Naïve Bayes	Features conditionally independent	Correlated features; continuous features tricky	Fast; works with little data; handles many features	Yes

Table 4.1: Supervised learning algorithms: assumptions, failure modes, and interpretability.

When to Use What

Start simple: Linear/logistic regression as baseline.
Need interpretability: Decision trees or linear models with regularization.
Non-linear patterns: SVM with RBF kernel, or tree ensembles such as Random Forests.
High-dimensional sparse data (e.g., text): Naïve Bayes or linear SVM.
Mixed feature types: Decision trees handle categorical features naturally.

Ethics Checkpoint: Classifier Fairness

You have trained a decision tree to predict loan defaults. The tree achieves 92% accuracy overall.
Technical-ethical questions:

1. You discover the tree uses "zip code" as a top split. Why might this be problematic even if zip code is predictive?
2. The tree has 85% recall for Group A but only 60% recall for Group B. Both groups have the same base rate of default. Is this fair? Which definition of fairness is violated?
3. You remove zip code from features. Accuracy drops to 89%. A manager says "we're sacrificing performance for political correctness." How do you respond?

This is the type of technical-ethical hybrid problem that appears on IAIO. See Chapter 2 for fairness definitions.

Conceptual Pause: Classifiers Compared

We have now seen several classification algorithms. Here is how they compare:

Method	Decision Boundary	Best For	Many Features?	Interpretable?
Logistic Regression	Linear (hyperplane)	Linearly separable data; probability outputs	Yes, with regularization	Yes (coefficients)
SVM (linear)	Linear (max margin)	High-dimensional data; clear margin	Yes	Somewhat
SVM (kernel)	Non-linear	Complex boundaries; moderate data	Depends	No
Decision Tree	Axis-aligned rectangles	Mixed feature types; interpretability	Can struggle	Yes (rules)
Naïve Bayes	Quadratic	Text classification; independent features	Yes	Yes (probabilities)

Common Mistakes

- **Using SVM without scaling**: SVM is sensitive to feature scales. Always standardize first.
- **Deep decision trees without pruning**: They will memorize training data.
- **Naïve Bayes with correlated features**: Independence assumption is violated; probabilities will be wrong (but classification may still work).
- **Logistic regression for non-linear boundaries**: It will underfit. Consider kernel SVM or trees.

Model Evaluation

Training a model is only half the battle. Rigorous evaluation on held-out data is essential.
Key principle: Never evaluate on training data, it gives misleadingly optimistic results.
For comprehensive coverage of evaluation methods, metrics, and cross-validation, see **Chapter 5**.

4.10 Summary

Linear regression models continuous targets as weighted sums of features. Gradient descent finds optimal weights by iteratively reducing squared error.[3]

Logistic regression handles classification by applying the sigmoid function to a linear combination, producing probabilities. Cross-entropy loss enables gradient-based learning.

Regularization (L1/L2) prevents overfitting by penalizing large weights. L1 produces sparse models; L2 shrinks all weights.

SVMs find maximum-margin decision boundaries. Kernels enable non-linear classification without explicit feature computation.

Decision trees learn axis-aligned splits that recursively partition the feature space. Entropy/information gain guide split selection.

Naïve Bayes applies Bayes' theorem[4] with conditional independence assumptions. Despite being "wrong," it often works well.

Algorithm choice depends on data characteristics, interpretability needs, and computational constraints. No single algorithm is best for all problems.

4.11 Practice Problems

1. A linear regression model predicts house prices: $h(x) = 50{,}000 + 100x$, where x is size in square feet. What price does the model predict for a 2000 square foot house? What is the interpretation of the coefficients?

2. A logistic regression model outputs $h_\theta(\mathbf{x}) = 0.73$. What class is predicted at threshold 0.5? What does 0.73 represent?

3. In building a decision tree, splitting on attribute A produces branches with $(4+, 0-)$ and $(1+, 5-)$. If initial entropy was 1.0, calculate the information gain.

4. Explain why SVM is called a "large margin classifier." How does C affect the margin?

5. Compare L1 and L2 regularization. When would you prefer each?

6. **(IAIO-style: Algorithm You've Never Seen)**

   A colleague proposes the "Weighted Neighbor Classifier" (WNC):

[3] See Appendix D for calculus of gradients.
[4] See Appendix C for probability review.

(a) For a test point $\mathbf{x}$, compute distance d_i to each training point $\mathbf{x}_i$

(b) Assign weight $w_i = \exp(-d_i^2/\sigma^2)$ to each training point

(c) Predict: $\hat{y} = \text{sign}\left(\sum_i w_i \cdot y_i\right)$ where $y_i \in \{-1, +1\}$

Answer the following:

(a) What happens as $\sigma \to 0$? As $\sigma \to \infty$?

(b) How does this relate to k-NN? To kernel methods?

(c) What is the decision boundary shape? Is it linear?

(d) What are the failure modes of this algorithm?

(e) Propose a modification to handle class imbalance.

7. **(Multi-part)** Consider the XOR problem with points $(0, 0)$, $(0, 1)$, $(1, 0)$, $(1, 1)$ and labels $0, 1, 1, 0$.

   (a) Prove no linear classifier can solve this.

   (b) Show that adding feature $x_3 = x_1 \cdot x_2$ makes it linearly separable.

   (c) Draw a decision tree that solves XOR.

   (d) How does this relate to the kernel trick?

8. **(Technical-Ethical)** A bank's logistic regression uses "years at current address" (negative coefficient = lower default risk). This correlates with age.

   (a) Why might this constitute age discrimination even though age isn't a feature?

   (b) Removing the feature drops AUC from 0.82 to 0.79. Quantify the tradeoff.

   (c) What fairness metrics would you compute?

9. **(IAIO-style: Another Algorithm You've Never Seen)**

   Consider the "Prototype Classifier":

   (a) For each class c, compute the centroid (mean) of all training points in that class: $\boldsymbol{\mu}_c = \frac{1}{|C_c|} \sum_{\mathbf{x} \in C_c} \mathbf{x}$

   (b) To classify a new point $\mathbf{x}$, assign it to the class whose centroid is closest: $\hat{y} = \arg\min_c \|\mathbf{x} - \boldsymbol{\mu}_c\|$

   Answer the following:

(a) What is the shape of the decision boundary between two classes? (Hint: where is $\|\mathbf{x} - \boldsymbol{\mu_1}\| = \|\mathbf{x} - \boldsymbol{\mu_2}\|$?)

(b) How does this compare to k-NN with $\boldsymbol{k} =$ all points in each class?

(c) What assumption about the data would make this classifier work well?

(d) Construct a simple 2D dataset where this classifier fails badly. What property does your dataset have?

(e) How many parameters does this classifier store? Compare to k-NN and logistic regression.

Chapter 5

Model Evaluation

> The purpose of computing is insight, not numbers.
>
> — Richard Hamming

5.1 Introduction: The Problem of Generalization

Imagine you are preparing a student for an important exam. You give them 100 practice problems, and they study diligently. When you test them on those same 100 problems, they answer every single one correctly. Impressive! But when the actual exam arrives with different questions, they fail miserably. What happened?

The student memorized the specific answers to the practice problems rather than learning the underlying principles. They could reproduce answers they had seen before but could not apply their knowledge to new situations. This is the educational equivalent of a fundamental problem in machine learning: the difference between **memorization** and **generalization**.

Machine learning models face exactly this challenge. A model that performs perfectly on training data but fails on new, unseen data has not truly learned. It has merely memorized. Such a model is useless for practical purposes, since the whole point of machine learning is to make predictions about data we have not yet seen.

The Core Challenge

A model that performs perfectly on training data but fails on new data is **useless**. The entire value of machine learning lies in generalization. The ability to apply learned patterns to new situations. Our evaluation methods must assess generalization, not just training performance.

5.2 Parameters vs. Hyperparameters

Before diving into evaluation methods, we need to clarify a distinction that confuses many beginners.

Parameters are values that a model learns automatically from data during training. In a neural network, these are the weights connecting neurons. The learning algorithm adjusts these to fit the training data.

Hyperparameters are settings that humans choose before training begins. They control how learning works but are not learned from data. Examples include the learning rate, number of epochs, network architecture, and regularization strength.

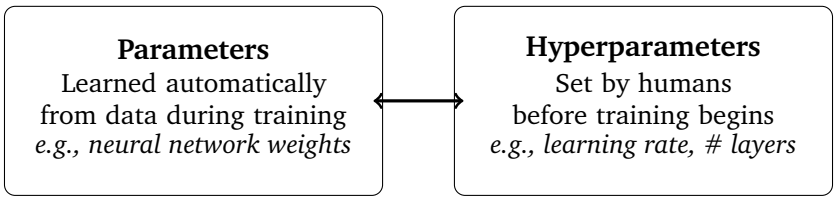


Figure 5.1: Parameters are learned; hyperparameters are chosen by humans.

5.3 The Importance of Data Splitting

The most fundamental principle in model evaluation is: **never evaluate a model on the same data used to train it.**

If we train on certain data and test on that same data, the model might achieve high accuracy simply by memorizing training examples. This tells us nothing about generalization.

The Golden Rule

Never test on training data!
Evaluating on training data is like giving students their exam questions in advance. High performance tells you nothing about true understanding.

We split data into separate sets with different purposes:

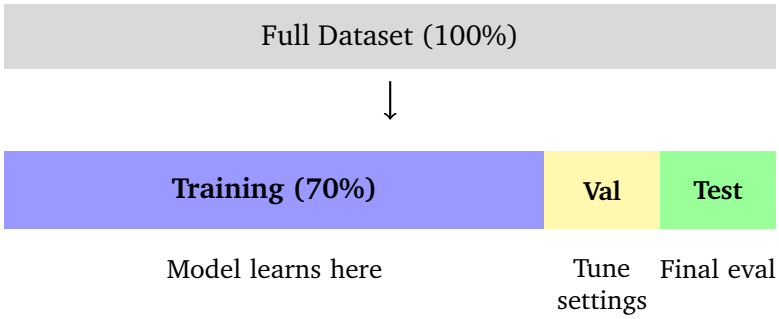


Figure 5.2: Standard train-validation-test split.

The **training set** (60-80%) is where learning happens. The **validation set** (10-20%) is used to check progress during development and tune hyperparameters. The **test set** (10-20%) is kept completely untouched until final evaluation, it provides an unbiased estimate of real-world performance.

Analogy: Preparing for an Exam

Training set = Textbook (learn from it)
Validation set = Practice exams (check progress, adjust study strategy)
Test set = Final exam (true measure of learning)

When splitting data, select examples **randomly**. If data is sorted (by date, category, etc.), taking the first 70% creates bias.

5.4 Bias and Variance: Understanding Errors

Every model makes mistakes. Understanding *why* helps us fix them.

Underfitting (high bias) occurs when the model is too simple. It cannot capture patterns even in training data. Solution: increase complexity.

Overfitting (high variance) occurs when the model is too complex. It memorizes training data but fails on new data. Solution: reduce complexity, add

regularization, or collect more data.

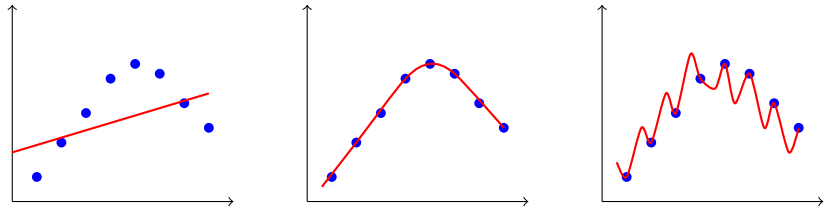


Figure 5.3: Underfitting (too simple), good fit (just right), and overfitting (too complex).

Diagnosing: If training accuracy is low, the model is underfitting. If training accuracy is high but validation accuracy is much lower, the model is overfitting.

5.5 Classification Metrics

For classification problems, we need appropriate metrics. The **confusion matrix** shows predictions versus actual labels:

		Predicted	
		Positive	Negative
Actual	Positive	True Positive (TP)	False Negative (FN)
	Negative	False Positive (FP)	True Negative (TN)

Accuracy is the fraction correct: $(\boldsymbol{TP} + \boldsymbol{TN})/(\boldsymbol{TP} + \boldsymbol{TN} + \boldsymbol{FP} + \boldsymbol{FN})$

But accuracy can mislead with imbalanced classes! A smuggling detector that always says "not smuggling" gets 99.5% accuracy if only 0.5% of containers have contraband. Yet catches zero smugglers.

Precision answers "of my positive predictions, how many were correct?"

$$\text{Precision} = \frac{\boldsymbol{TP}}{\boldsymbol{TP} + \boldsymbol{FP}}$$

Use when false positives are costly (e.g., spam filter: you do not want to lose real emails).

Recall answers "Of all actual positives, how many did I catch?"

$$\text{Recall} = \frac{\boldsymbol{TP}}{\boldsymbol{TP} + \boldsymbol{FN}}$$

Use when false negatives are costly (e.g., cancer screening, don't miss cases).

F1 Score balances precision and recall:

$$\text{F1} = \mathbf{2} \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Worked Example

A disease detector: TP=80, FP=20, FN=10, TN=890
Accuracy = $\mathbf{(80+890)/(80+890+20+10) = 970/1000 = 97\%}$
Precision = $\mathbf{80/(80+20) = 80/100 = 80\%}$
Recall = $\mathbf{80/(80+10) = 80/90 \approx 88.9\%}$
F1 = $\mathbf{2 \cdot (0.80 \times 0.889)/(0.80+0.889) \approx 84.2\%}$

There is typically a **trade-off** between precision and recall, controlled by the **decision threshold**. Most classifiers output a probability or score; you choose a threshold above which you predict positive.

Raising the threshold (e.g., from 0.5 to 0.8) means you only predict positive when the model is very confident. Fewer false positives, so precision increases. But you miss more true positives, so recall decreases.

Lowering the threshold (e.g., from 0.5 to 0.2) means you flag more cases as positive. You catch more true positives, so recall increases. But more negatives are incorrectly flagged, so precision decreases.

Threshold Choice Is a Business Decision

A cancer screening test might use a low threshold (say 0.3): missing a cancer is far worse than ordering an unnecessary follow-up test. High recall matters more than high precision.
A spam filter might use a high threshold (say 0.8): deleting a legitimate email is worse than letting some spam through. High precision matters more than high recall.
There is no "correct" threshold. The right choice depends on the relative costs of false positives and false negatives in your specific application.

5.5.1 ROC Curves and AUC

The **Receiver Operating Characteristic (ROC) curve** provides a comprehensive view of classifier performance across all possible thresholds.

Most classifiers output a probability or score rather than a hard class prediction. We convert this to a prediction by choosing a threshold: if the score

exceeds the threshold, predict positive. Different thresholds produce different trade-offs between catching positives (True Positive Rate) and falsely flagging negatives (False Positive Rate).

- **True Positive Rate (TPR)** = $\frac{TP}{TP+FN}$ = Recall (fraction of actual positives correctly identified)
- **False Positive Rate (FPR)** = $\frac{FP}{FP+TN}$ (fraction of actual negatives incorrectly flagged)

The ROC curve plots TPR (y-axis) against FPR (x-axis) for every possible threshold:

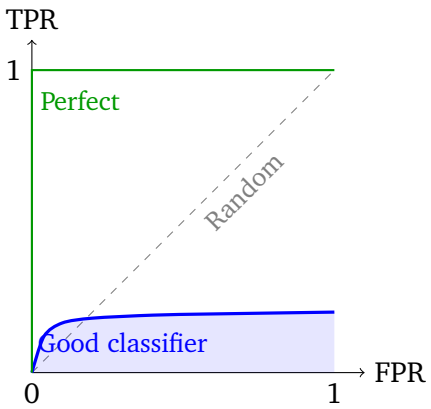


Figure 5.4: ROC curves for different classifiers. The shaded area represents AUC. A perfect classifier hugs the top-left corner; a random classifier follows the diagonal.

Understanding the ROC curve:

- The **top-left corner** (0, 1) represents perfect classification: TPR=1 (catch all positives), FPR=0 (no false alarms)
- The **diagonal** represents a random classifier that cannot distinguish classes
- A curve closer to the top-left is better

The **Area Under the Curve (AUC)** summarizes the ROC curve as a single number:

- AUC = 1.0: Perfect classifier
- AUC = 0.5: Random guessing (no discriminative power)
- AUC < 0.5: Worse than random (predictions are inverted)

Interpreting AUC

AUC has a beautiful probabilistic interpretation: it equals the probability that a randomly chosen positive example will be ranked higher than a randomly chosen negative example.
AUC = 0.85 means: if you pick a random positive and a random negative, there is an 85% chance the classifier assigns a higher score to the positive.
Rule of thumb: AUC $>$ 0.9 is excellent; 0.8–0.9 is good; 0.7–0.8 is fair; $<$ 0.7 may be poor.

When to use AUC vs. other metrics:

- Use AUC when you want to evaluate ranking ability without committing to a specific threshold
- Use precision/recall/F1 when you have a specific threshold in mind or when class imbalance is extreme
- AUC can be misleading with highly imbalanced datasets; consider precision-recall curves instead

5.5.2 Calibration: Are the Probabilities Meaningful?

A classifier might correctly rank positive cases above negative cases (good AUC) while producing probability estimates that are wildly wrong. **Calibration** measures whether predicted probabilities match actual frequencies.

A model is well-calibrated if, among all cases where it predicts "70% chance of default," approximately 70% actually do default. A model that predicts 70% but only 30% default is *accurate* in its ranking (it correctly identifies riskier cases) but *poorly calibrated* (its probability numbers are meaningless).

The reverse is also possible. A model can be well-calibrated but inaccurate: it correctly estimates that the overall default rate is 10%, but it assigns 10% to every single borrower. The probabilities are "correct" on average, but the model has no discriminative power.

When Calibration Matters

Calibration matters whenever the predicted probability itself drives a decision, not just the ranking. If a doctor needs to tell a patient "your risk of this condition is 30%," that number must be meaningful. If you are only sorting patients from highest to lowest risk, ranking (AUC) is sufficient.

Ethics Checkpoint: When Good Metrics Go Bad

A hospital deploys a model to predict patient readmission risk. The model has AUC = 0.85.

Technical-ethical questions:

1. The hospital uses the model to *deny* follow-up care to "low-risk" patients (to save costs). The model has 90% precision for high-risk but only 70% for low-risk. What is the ethical problem?
2. Management proposes lowering the threshold to flag more patients as high-risk. This increases recall but decreases precision. Who benefits? Who is harmed?
3. A doctor argues: "I should override the model whenever I disagree." Another argues: "Consistent application of the model is fairer than human judgment." Analyze both positions.

This is the type of technical-ethical hybrid problem that appears on IAIO.

Conceptual Pause: Choosing the Right Metric

With many metrics available, how do you choose?

Metric	Use When	Caution
Accuracy	Classes are balanced; all errors equally costly	Misleading for imbalanced data
Precision	False positives are costly (e.g., spam annoying users)	Ignores false negatives
Recall	False negatives are costly (e.g., missing cancer)	Ignores false positives
F1 Score	Need balance; single number required	Assumes equal importance of precision/recall
AUC-ROC	Evaluate across thresholds; comparing models	Does not tell specific threshold performance
Cost-weighted	Different errors have different costs	Requires knowing costs

5.6 Regression Metrics

For regression (predicting continuous numbers), we measure prediction errors.

Mean Absolute Error (MAE) is the average of absolute differences:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

MAE is interpretable in the same units as the target.

Mean Squared Error (MSE) squares errors, penalizing large errors more:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

RMSE $= \sqrt{\text{MSE}}$ returns to original units.

R-Squared (R^2) measures what fraction of variance the model explains:

$$R^2 = 1 - \frac{\sum(y_i - \hat{y}_i)^2}{\sum(y_i - \bar{y})^2}$$

$R^2 = 1$ means perfect predictions; $R^2 = 0$ means no better than predicting the mean.

5.7 Cross-Validation

What if you don't have much data? A single train-validation split might be unreliable, you might get lucky or unlucky with which examples end up where.

K-fold cross-validation addresses this by using all data for both training and validation:

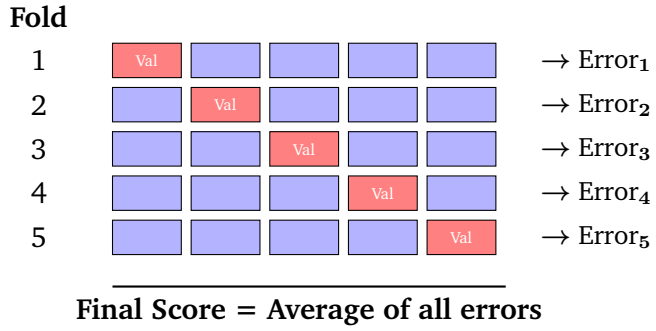


Figure 5.5: 5-fold cross-validation: each fold serves as validation exactly once.

The procedure: divide data into k equal folds; for each fold i, train on the other $k - 1$ folds and validate on fold i; average the k validation scores.

Benefits: more reliable estimates, uses all data for both training and validation, every example gets validated exactly once. Cost: trains k models instead of 1. Common choices: $k = 5$ or $k = 10$.

5.8 How People Accidentally Cheat

Even well-intentioned practitioners can inadvertently invalidate their evaluation. Understanding these pitfalls is essential for producing trustworthy results.

5.8.1 Leaderboard Overfitting

In competitions, the public leaderboard shows performance on a portion of test data. A dangerous pattern emerges:

1. Submit prediction $\rightarrow$ see score
2. Adjust model $\rightarrow$ submit again $\rightarrow$ see score
3. Repeat hundreds of times

Each submission leaks information about the test set. After enough submissions, you're effectively *fitting to the test data*. Your public leaderboard score becomes meaningless, and your private leaderboard score (on held-out data) collapses.

Why Leaderboard Rank $\neq$ Model Quality

A high leaderboard rank does **not** mean you have a good model. It may mean:

- You got lucky on the public test subset
- You've overfit to the public test data through repeated submissions
- The public/private split doesn't match the real-world distribution

The only reliable signal is your cross-validation score on your own data.

Models that rank well on CV but poorly on the public leaderboard are often *better* than the reverse, they'll typically perform better on the private leaderboard and in real deployment.

The fix: Trust your cross-validation score, not your leaderboard score. Limit submissions.

5.8.2 Why "Just One More Look" Breaks Guarantees

The test set's value comes from a single property: the model has never been influenced by it. Each time you check test performance and then adjust your model, you leak information from the test set into your modeling decisions. This is true even if you do not directly train on it.

Consider: you train Model A, check its test score (82%), decide it is not good enough, build Model B with different features, check again (85%), tweak further, check again (87%). After 20 iterations, you report 87% test accuracy. But this is no longer an honest estimate. You have effectively used the test set to select among 20 models, and by chance alone, some will score higher on this particular test set than they would on new data.

This is the same statistical problem as multiple hypothesis testing. If you evaluate 20 models on the same test set, you are running 20 comparisons. Even if no model is truly better, some will appear better by random variation. The more times you look, the more inflated your "best" score becomes.

The fix: Touch the test set exactly once. Use cross-validation for all development decisions. If you must evaluate multiple final candidates, apply statistical corrections or use a fresh test set.

5.8.3 Data Leakage in Preprocessing

A subtle but common mistake: fitting preprocessing on the *full dataset* before splitting.

Wrong vs. Right

Wrong:
1. Compute mean/std of all data
2. Standardize all data
3. Split into train/test

Right:
1. Split into train/test
2. Compute mean/std of training data only
3. Standardize train using train statistics
4. Standardize test using train statistics

The wrong approach leaks test set information into training.

5.8.4 Temporal Leakage

For time-series data, random train/test splits are invalid. If you train on data from 2023 to predict 2022, you're using the future to predict the past.

The fix: Always split by time. Train on past, validate on future.

5.8.5 Good Test $\neq$ Good Real-World

Even with a perfectly constructed test set, honest evaluation, and no leakage, a strong test score does not guarantee the model will work well in deploy-

ment. The test set measures performance on data drawn from the same distribution as training data. The real world may differ.

Patient demographics shift. Consumer preferences change. Economic conditions evolve. A model that scores 95% on a test set drawn from 2024 hospital data may score 80% on 2026 data from a different hospital network. The test score was not wrong; it accurately measured performance under 2024 conditions. But it was not a promise about the future.

This is why Chapter 3 emphasizes distribution shift and test set validity. A test score is an estimate under specific assumptions. When those assumptions break, so does the estimate.

5.9 Hyperparameter Tuning

Finding good hyperparameters can make the difference between success and failure.

Grid search tries every combination of specified values. Simple but expensive. 5 hyperparameters with 10 values each means $\mathbf{10^5 = 100,000}$ combinations.

Random search randomly samples combinations. often finds good solutions faster because not all hyperparameters are equally important.

Practical tips: start with learning rate (usually most important). Use logarithmic scales (try 0.001, 0.01, 0.1, not 0.1, 0.2, 0.3). Do not overtune. Too many experiments can overfit to the validation set itself.

5.10 Summary

Key Takeaways

Data Splitting: Training (70%) for learning, Validation (15%) for tuning, Test (15%) for final evaluation. Never test on training data.
Bias-Variance: Underfitting = too simple (high bias). Overfitting = too complex (high variance). The sweet spot depends on data size and noise.
Classification Metrics: Accuracy misleads with imbalanced data. Use precision (FP costly), recall (FN costly), or F1 (balance). Threshold choice is a business decision.
Calibration: Good ranking (AUC) does not imply meaningful probabilities. Calibration matters when the probability value itself drives decisions.
ROC and AUC: ROC shows TPR vs FPR across thresholds. AUC summarizes discriminative ability.
Cross-Validation: K-fold CV uses all data for both training and validation, giving more reliable estimates.
Test Set Integrity: Each look at the test set leaks information. "Just one more look" is the same statistical error as multiple hypothesis testing.
Test $\neq$ Deployment: Good test performance measures accuracy under test conditions. Distribution shift can make that estimate meaningless in the real world.

1

5.11 Practice Problems

1. With 10,000 data points, describe how you would split for hyperparameter tuning. What proportion for each set?

2. A model achieves 99% training accuracy but only 65% validation accuracy. Diagnose using bias-variance.

3. Calculate precision, recall, and F1 from this confusion matrix:

	Pred +	Pred –
Actual +	450	50
Actual –	100	400

[1]The theory of why cross-validation works is covered in Chapter 11 (PAC learning and generalization bounds).

4. When would you optimize for precision over recall? Give a real-world example.

5. **(Proof-style: Why Cross-Validation Works)**

   In k-fold CV, we train k models, each on $(k-1)/k$ of the data, and average their validation errors.

   (a) Explain intuitively why averaging over k folds gives a more reliable estimate than a single train/test split.

   (b) As k increases toward n (leave-one-out CV), what happens to:

   - The bias of the error estimate?
   - The variance of the error estimate?
   - The computational cost?

   (c) Why is $k = 5$ or $k = 10$ typically a good compromise?

   (d) If your data has natural groups (e.g., multiple samples per patient), why might standard k-fold CV give optimistic estimates? What should you do instead?

6. **(Proof-style: Bias-Variance Decomposition)**

   For regression with true function $f(x)$ and model $\hat{f}(x)$:

   $$\mathbb{E}[(f(x) - \hat{f}(x))^2] = \text{Bias}^2 + \text{Variance} + \text{Noise}$$

   (a) Define bias in terms of $\mathbb{E}[\hat{f}(x)]$ and $f(x)$.

   (b) A constant model $\hat{f}(x) = c$ has high bias. Why?

   (c) A model that interpolates every training point has high variance. Why?

   (d) Regularization increases bias. Why does this often improve test error?

7. **(Multi-part: When Accuracy Deceives)** A disease test (1% prevalence) gives:

	Test +	Test –
Has Disease	90	10
No Disease	495	9405

   (a) Calculate accuracy, precision, recall, F1.

   (b) Why is 94.95% accuracy misleading?

(c) What is the positive predictive value? Why does it differ so much from recall?

8. **(Cheating Detection)** Your colleague's model achieves AUC = 0.99. What three ways might they have accidentally inflated this score?

9. A model has 100% training / 60% test accuracy. Another has 70% training / 65% test. Which is better and why?

Chapter 6

Unsupervised Learning and Clustering

To classify is human.

Robert Darnton

6.1 Introduction: Learning Without Labels

Throughout most of this book, we have focused on supervised learning, where each training example comes with a label telling us the "correct answer." We show the algorithm pictures labeled "cat" or "dog," and it learns to classify new pictures. We provide house prices alongside house features, and it learns to predict prices for new houses. The labels guide the learning process, telling the algorithm what patterns to look for.

But what if we have data without labels? This situation is extremely common in practice. A company might have millions of customer transaction records but no labels indicating customer types. A biologist might have gene expression data from thousands of cells but no classification of cell types. A social media platform has billions of posts but no categorization of topics. The data exists, potentially rich with structure, but no one has told us what that structure is.

Unsupervised learning tackles exactly this challenge: discovering hidden patterns and structure in data without any labels to guide us. Instead of learning a mapping from inputs to outputs, we seek to understand the data itself, its clusters, its underlying dimensions, its statistical properties.

Supervised vs. Unsupervised Learning

Supervised Learning: We have input-output pairs $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$. The goal is to learn a function $f : X \to Y$ that predicts outputs from inputs. Labels tell us what to learn.
Unsupervised Learning: We have only inputs $x_1, x_2, \ldots, x_n$ with no labels. The goal is to discover structure (groups, patterns, representations) hidden in the data. We must figure out what is worth learning.

Unsupervised learning encompasses several different tasks. **Clustering** groups similar data points together, discovering natural categories in the data. **Dimensionality reduction** finds lower-dimensional representations that preserve important information, making data easier to visualize and analyze. **Density estimation** models the probability distribution that generated the data. This chapter focuses primarily on clustering, one of the most widely used unsupervised techniques.

6.2 Clustering: Finding Groups in Data

Clustering is based on a simple intuition: data points often fall naturally into groups, and discovering these groups reveals something meaningful about the data's structure.

There is an important difference from supervised learning that shapes everything in this chapter. In classification, there is a correct answer: the email *is* spam or it is not. In clustering, there is no ground truth. The same dataset can be clustered in many valid ways depending on which features you use, which distance measure you choose, and what value of k you set. Two analysts can produce completely different clusterings of the same data and both can be reasonable. This **objective ambiguity** means that evaluating clusters requires domain knowledge and practical judgment, not just a metric.

Consider customers at an online retailer. Some customers buy frequently but spend little per purchase; others buy rarely but spend lavishly; still others buy only during sales. These different behaviors suggest natural customer segments. If we can identify these segments automatically, we can tailor marketing strategies to each group. The retailer never labeled customers as "frequent buyer" or "sale hunter", the clustering algorithm discovers these categories from the purchasing patterns alone.

The Clustering Objective

Given a dataset $S = \{x_1, x_2, \ldots, x_n\}$, partition the data points into k groups (clusters) such that:
Points within the same cluster are **similar** to each other. They share common characteristics.
Points in different clusters are **dissimilar** from each other. They differ in meaningful ways.
The algorithm must discover both the cluster assignments and what "similar" means for this data.

Clustering appears across virtually every domain where data is collected. In biology, clustering groups genes with similar expression patterns, revealing functional relationships. In astronomy, clustering identifies different types of celestial objects from their spectral signatures. In document analysis, clustering organizes articles by topic without requiring predefined categories. In image segmentation, clustering groups pixels to identify distinct objects within a scene. In anomaly detection, points that do not fit any cluster may represent unusual events worth investigating.

6.3 Measuring Similarity: Distance Functions

Before we can group similar points, we must define what "similar" means. Clustering algorithms typically measure similarity through its opposite: **distance**. Points that are close together (small distance) are considered similar; points far apart (large distance) are dissimilar.

The choice of distance function profoundly affects clustering results. Different distance measures encode different notions of similarity, and the right choice depends on the nature of your data and what kind of patterns you seek.

A valid **distance metric** must satisfy three mathematical properties. First, *non-negativity*: distances are never negative, and the distance from a point to itself is zero. Second, *symmetry*: the distance from x to y equals the distance from y to x. Third, the *triangle inequality*: going directly from x to z is never longer than going via an intermediate point y. These properties ensure that distance behaves in intuitively sensible ways.

6.3.1 The Minkowski Family

The most common distance functions belong to the **Minkowski family**, parameterized by a value p:

$$d_p(\mathbf{x}, \mathbf{y}) = \left(\sum_{i=1}^{d} |x_i - y_i|^p \right)^{1/p}$$

Different values of p give different notions of distance.

When $p = 2$, we get the familiar **Euclidean distance**, the straight-line distance between two points:

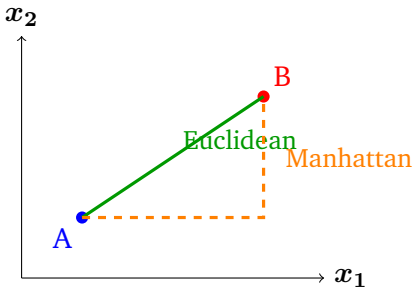
$$d_2(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2}$$

This is the distance you would measure with a ruler. It treats all directions equally and corresponds to our everyday geometric intuition.

When $p = 1$, we get the **Manhattan distance** (also called taxicab or city-block distance):

$$d_1(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{d} |x_i - y_i|$$

Imagine navigating a city with a grid of streets: you cannot walk diagonally through buildings but must follow the streets. The Manhattan distance measures how far you would walk following this constraint. It is particularly useful when features represent fundamentally different quantities that should not be mixed geometrically.



Euclidean: $\sqrt{3^2 + 2^2} = \sqrt{13} \approx 3.6$
Manhattan: $3 + 2 = 5$

Figure 6.1: Euclidean distance measures the straight-line path; Manhattan distance measures the path along coordinate axes. For points A(1,1) and B(4,3), these give different values.

When $p = \infty$, we get the **Chebyshev distance**: the maximum difference along any single dimension. This is useful when the worst-case deviation matters more than aggregate differences.

Beyond the Minkowski family, other distance measures serve specialized purposes. **Cosine distance** measures the angle between vectors rather than their magnitude, useful when direction matters more than scale (common in text analysis). **Hamming distance** counts positions where binary vectors differ. **Edit distance** measures the minimum number of operations needed to transform one string into another.

Scaling: The Most Common Clustering Failure

Distance-based algorithms are sensitive to feature scales. If one feature ranges from 0 to 100,000 (annual income) and another from 0 to 10 (number of children), Euclidean distance will be dominated entirely by income. The algorithm will cluster by income alone, as if the other features do not exist.

This is the same issue Chapter 3 discusses for supervised learning, but it is even more dangerous in clustering because there are no labels to reveal the mistake. A supervised model's accuracy would drop; a clustering algorithm will happily produce income-based clusters and report low distortion.

Always standardize features before distance-based clustering (K-means, hierarchical). The exception is when the raw scale is meaningful and you intentionally want one feature to dominate.

6.4 K-Means Clustering

K-means is the most widely used clustering algorithm, prized for its simplicity and efficiency. The algorithm partitions data into k clusters, where each cluster is represented by its **centroid**, the mean position of all points in that cluster.

The intuition behind K-means is appealingly simple. We want to find cluster centers such that each point is close to its assigned center. Points "belong" to whichever center is nearest, and centers should be positioned to minimize the total distance to their assigned points.

6.4.1 The Algorithm

K-means alternates between two steps until convergence:

The K-Means Algorithm

Input: Data points $\{x_1, \ldots, x_n\}$ and number of clusters k
Initialize: Place k cluster centroids $\mu_1, \ldots, \mu_k$ at random positions
Repeat until convergence:
Assignment step: Assign each point to the nearest centroid

$$c_i = \arg\min_j \|x_i - \mu_j\|^2$$

Update step: Move each centroid to the mean of its assigned points

$$\mu_j = \frac{1}{|C_j|} \sum_{x_i \in C_j} x_i$$

Convergence: Stop when assignments no longer change

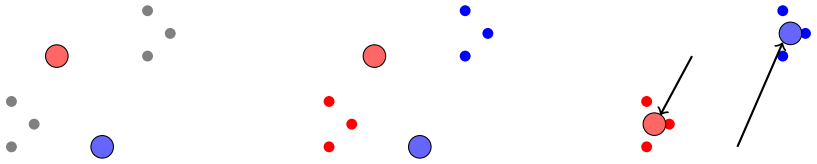
Let us trace through why this algorithm works. The assignment step ensures that every point belongs to whichever cluster center is closest. This is the most natural assignment given the current centers. The update step repositions each center to be at the geometric center of its assigned points. This minimizes the total squared distance from points to their center.

Both steps reduce (or at least do not increase) a quantity called the **distortion**:

$$J = \sum_{i=1}^{n} \|x_i - \mu_{c_i}\|^2$$

This is the sum of squared distances from each point to its assigned centroid. The assignment step reduces distortion because reassigning a point to a closer centroid decreases its contribution to the sum. The update step reduces distortion because the mean is the point that minimizes squared distances to a set of points (a fact from basic statistics).

Since distortion decreases at each step and cannot go below zero, the algorithm must eventually converge. This is a beautiful example of optimization through alternating minimization: we cannot easily optimize both assignments and centers simultaneously, but optimizing each while holding the other fixed is straightforward.



(1) Initialize (2) Assign (3) Update

Figure 6.2: One iteration of K-means: (1) Random initialization places centroids; (2) Points are assigned to the nearest centroid; (3) Centroids move to the mean of their assigned points.

Worked Example: K-Means by Hand

Data: Four points in 2D: $A = (1, 1)$, $B = (2, 1)$, $C = (4, 3)$, $D = (5, 4)$
Goal: Cluster into $k = 2$ groups.
Initialize: Choose $\mu_1 = (1, 1)$ and $\mu_2 = (5, 4)$ (points A and D).

Iteration 1:
Assignment step: Calculate distance from each point to each centroid.

Point	Dist to μ_1	Dist to μ_2	Assigned
A = (1,1)	**0**	**5**	Cluster 1
B = (2,1)	**1**	**4.24**	Cluster 1
C = (4,3)	**3.61**	**1.41**	Cluster 2
D = (5,4)	**5**	**0**	Cluster 2

Cluster 1: $\{A, B\}$ Cluster 2: $\{C, D\}$
Update step: Compute new centroids as mean of assigned points.

$$\mu_1 = \frac{(1,1)+(2,1)}{2} = (1.5, 1) \qquad \mu_2 = \frac{(4,3)+(5,4)}{2} = (4.5, 3.5)$$

Iteration 2:
Assignment step: Recalculate distances with new centroids.

Point	Dist to μ_1	Dist to μ_2	Assigned
A = (1,1)	**0.5**	**4.27**	Cluster 1
B = (2,1)	**0.5**	**3.54**	Cluster 1
C = (4,3)	**3.20**	**0.71**	Cluster 2
D = (5,4)	**4.61**	**0.71**	Cluster 2

Assignments unchanged $\Rightarrow$ **Converged!**
Final distortion: $J = 0.5^2 + 0.5^2 + 0.71^2 + 0.71^2 = 1.5$

6.4.2 Choosing the Number of Clusters

K-means requires specifying k, the number of clusters, in advance. But how do we know the right value? Often, we do not, and this is one of the fundamental challenges of clustering.

The **elbow method** provides one heuristic. We run K-means for various values of k and plot the distortion against k. As k increases, distortion necessarily decreases (with $k = n$, each point is its own cluster with zero distortion). We look for an "elbow" in the curve, a point where distortion stops decreasing rapidly and begins decreasing slowly. This suggests that additional clusters are not capturing meaningful structure, just fragmenting existing groups.

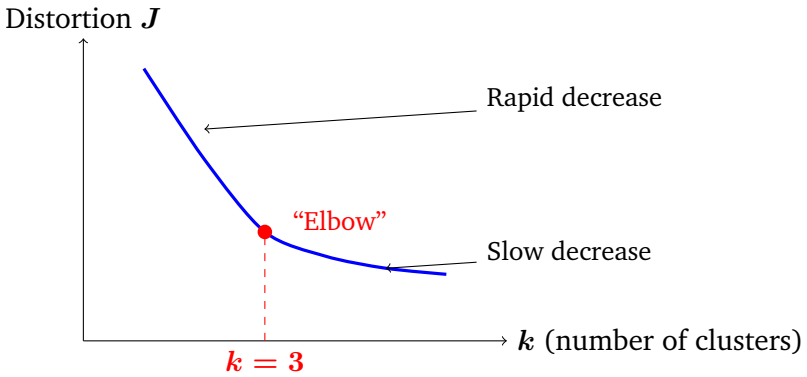


Figure 6.3: The elbow method: distortion decreases as k increases, but the rate of decrease slows. The "elbow" at $k = 3$ suggests three clusters capture the main structure.

6.4.3 Limitations of K-Means

Despite its popularity, K-means has important limitations that you should understand.

K-means assumes clusters are roughly **spherical** and similar in size. When clusters have elongated shapes, different densities, or vastly different sizes, K-means can produce poor results. The algorithm has no way to represent an elongated cluster, it can only place a centroid at the middle.

K-means is **sensitive to initialization**. Different random starting positions can lead to different final clusterings. This happens because K-means finds a local minimum of distortion, not necessarily the global minimum. A common remedy is to run the algorithm multiple times with different initializations and keep the best result.

K-means makes **hard assignments**: each point belongs to exactly one cluster. But real data often has ambiguous points that could reasonably belong to multiple clusters. A customer might exhibit behaviors from two different segments. K-means forces a binary choice when a probabilistic answer might be more appropriate.

Finally, K-means is **sensitive to outliers**. A single extreme point can pull a centroid far from where it should be, distorting the entire clustering.

6.5 Hierarchical Clustering

Hierarchical clustering takes a fundamentally different approach from K-means. Instead of requiring you to specify the number of clusters in advance, it builds a hierarchy of clusters that can be cut at any level to produce different numbers of groups.

6.5.1 Agglomerative Clustering

The most common form is **agglomerative** (bottom-up) clustering:

Agglomerative Clustering Algorithm

Initialize: Each point starts as its own cluster (n clusters for n points)
Repeat until only one cluster remains:
1. Find the two closest clusters
2. Merge them into a single cluster

Result: A tree (dendrogram) showing the sequence of merges

The key question is: how do we measure distance between clusters (not just points)? Different **linkage methods** give different answers:

- **Single linkage**: Distance between clusters = distance between their closest points
- **Complete linkage**: Distance = distance between their farthest points
- **Average linkage**: Distance = average of all pairwise distances
- **Ward's method**: Distance = increase in total within-cluster variance after merging

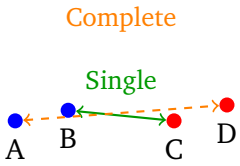


Figure 6.4: Single linkage uses the closest points between clusters; complete linkage uses the farthest.

6.5.2 Dendrograms

The result of hierarchical clustering is a **dendrogram**, a tree diagram showing the sequence of merges and the distances at which they occurred.

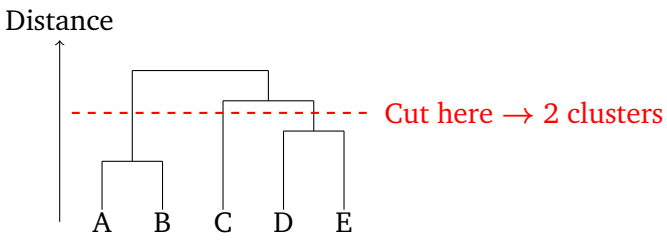


Figure 6.5: A dendrogram shows the hierarchical structure. Cutting at different heights produces different numbers of clusters.

To obtain a specific number of clusters, you "cut" the dendrogram horizontally at the appropriate height. The number of vertical lines you cross equals the number of clusters.

Advantages of hierarchical clustering:

- No need to specify k in advance
- Produces a rich hierarchical structure that may be meaningful
- Deterministic (no random initialization)

Limitations:

- Computationally expensive: $O(n^2)$ space, $O(n^3)$ time for naive implementation
- Once a merge is made, it cannot be undone
- Different linkage methods can give very different results

6.6 Gaussian Mixture Models

Gaussian Mixture Models (GMMs) address several limitations of K-means by taking a probabilistic approach to clustering. Instead of hard assignments, GMMs compute the probability that each point belongs to each cluster. Instead of representing clusters as single points (centroids), GMMs represent them as probability distributions.

6.6.1 The Generative Model

GMM assumes the data was generated by the following process: first, randomly select one of k Gaussian distributions; then, randomly sample a point from that distribution. Different points might come from different Gaussians, but we do not observe which Gaussian generated each point. We only observe the points themselves.

Mathematically, the probability of observing a point x is a weighted sum of Gaussian densities:

$$p(x) = \sum_{j=1}^{k} \phi_j \cdot \mathcal{N}(x|\mu_j, \Sigma_j)$$

Here, ϕ_j is the **mixing weight**. The probability of choosing component j (these must sum to 1). $\mathcal{N}(x|\mu_j, \Sigma_j)$ is the Gaussian probability density with mean μ_j and covariance matrix Σ_j. The mean determines where the Gaussian is centered; the covariance determines its shape (spherical, elongated, tilted).

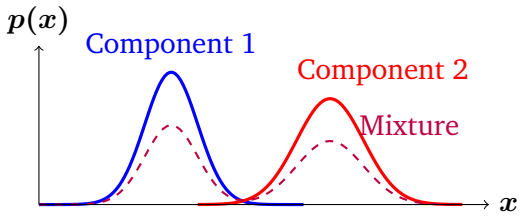


Figure 6.6: A mixture of two Gaussians (blue and red) and their combined distribution (purple dashed). Any point could have come from either component, with different probabilities.

6.6.2 Soft Assignments: Responsibilities

The key innovation of GMM over K-means is **soft assignments**. Instead of declaring that point x_i belongs to cluster j, GMM computes a **responsibility**

γ_{ij}, the probability that component j generated point x_i:

$$\gamma_{ij} = \frac{\phi_j \cdot \mathcal{N}(x_i | \mu_j, \Sigma_j)}{\sum_{l=1}^{k} \phi_l \cdot \mathcal{N}(x_i | \mu_l, \Sigma_l)}$$

This is simply Bayes' theorem: given that we observed x_i, what is the probability it came from component j? The numerator is the probability of j generating x_i; the denominator normalizes so probabilities sum to 1.

A point might have responsibilities $\gamma_{i1} = 0.7$ and $\gamma_{i2} = 0.3$, indicating 70% probability of belonging to cluster 1 and 30% to cluster 2. This soft assignment captures the uncertainty inherent in clustering: the point lies between clusters, and the data does not definitively assign it to either.

6.7 The Expectation-Maximization Algorithm

How do we find the parameters (means, covariances, mixing weights) of a GMM? The standard approach is the **Expectation-Maximization** (EM) algorithm, an elegant method for learning probabilistic models with hidden variables.

The challenge is circular: if we knew which component generated each point, estimating the Gaussian parameters would be straightforward. And if we knew the Gaussian parameters, computing responsibilities would be straightforward. But we know neither. EM breaks this circularity by alternating between the two computations.

The EM Algorithm for GMM

Initialize: Set random values for means μ_j, covariances Σ_j, and mixing weights ϕ_j
Repeat until convergence:
E-step (Expectation): Compute responsibilities using current parameters

$$\gamma_{ij} = \frac{\phi_j \cdot \mathcal{N}(x_i | \mu_j, \Sigma_j)}{\sum_{l} \phi_l \cdot \mathcal{N}(x_i | \mu_l, \Sigma_l)}$$

M-step (Maximization): Update parameters using responsibilities

$$\mu_j = \frac{\sum_i \gamma_{ij} x_i}{\sum_i \gamma_{ij}}, \quad \Sigma_j = \frac{\sum_i \gamma_{ij} (x_i - \mu_j)(x_i - \mu_j)^T}{\sum_i \gamma_{ij}}$$

$$\phi_j = \frac{1}{n} \sum_i \gamma_{ij}$$

The E-step asks: given our current parameter estimates, how responsible is each component for each point? The M-step asks: given these responsibilities, what parameter values best explain the data? Each step uses the output of the other, and the algorithm provably improves the model's likelihood at each iteration.

The update formulas have intuitive interpretations. The new mean μ_j is a weighted average of all points, where weights are the responsibilities. Points more likely to belong to component j have more influence on its mean. The new covariance σ_j similarly computes weighted deviations from the mean. The new mixing weight ϕ_j is the average responsibility. The fraction of the data that component j "claims."

6.7.1 K-Means as a Special Case

There is a beautiful connection between K-means and GMM: K-means is essentially GMM with hard assignments and spherical clusters. In K-means, responsibilities are binary (0 or 1), and all clusters have the same spherical shape. GMM generalizes this by allowing soft responsibilities and flexible cluster shapes.

This relationship explains why K-means is so fast: binary assignments are simpler to compute than soft probabilities, and ignoring covariances reduces computation further. The cost is less flexibility. K-means cannot represent the uncertainties and shapes that GMM can capture.

6.8 Dimensionality Reduction

High-dimensional data is difficult to visualize, expensive to compute with, and often contains redundant information. **Dimensionality reduction** techniques project data from a high-dimensional space to a lower-dimensional space while preserving important structure.

6.8.1 Principal Component Analysis (PCA)

PCA is the most fundamental dimensionality reduction technique. It finds the directions of maximum variance in the data and projects onto those directions.

The key insight: If data varies mostly along certain directions, we can capture most of the information by keeping only those directions and discarding the rest.

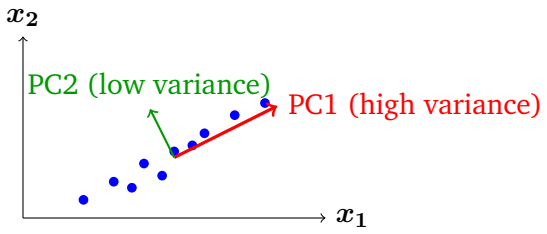


Figure 6.7: PCA finds the directions of maximum variance. PC1 captures most variation; projecting onto PC1 alone reduces dimensions while preserving most information.

The PCA algorithm:

1. Center the data (subtract the mean from each feature)
2. Compute the covariance matrix
3. Find the eigenvectors and eigenvalues of the covariance matrix
4. Sort eigenvectors by eigenvalue (largest first)
5. Project data onto the top k eigenvectors

The eigenvectors are the **principal components**, the directions of variance. The eigenvalues tell us how much variance each component captures.

Example: PCA Calculation

Consider 2D data centered at the origin with covariance matrix:

$$\Sigma = \begin{pmatrix} 4 & 2 \\ 2 & 2 \end{pmatrix}$$

The eigenvalues are $\lambda_1 = 5$ and $\lambda_2 = 1$.
The first eigenvalue captures $\frac{5}{5+1} = 83\%$ of the variance.
Projecting onto just the first principal component reduces from 2D to 1D while keeping 83% of the information.

Choosing the number of components: Plot the cumulative explained variance against the number of components. Choose enough components to capture a desired fraction (e.g., 95%) of the total variance.

When to use PCA:

- Visualization (reduce to 2D or 3D for plotting)
- Noise reduction (discard low-variance components)

- Preprocessing before other algorithms (reduce computation)
- Feature extraction (principal components may be meaningful)

6.8.2 t-SNE: Visualizing High-Dimensional Data

t-SNE (t-distributed Stochastic Neighbor Embedding) is a dimensionality reduction technique designed specifically for visualization. Unlike PCA, which preserves global variance, t-SNE focuses on preserving **local neighborhoods**: points that are close in high-dimensional space should remain close in the 2D or 3D plot.

The algorithm works in two stages. First, it computes pairwise similarities in the original high-dimensional space, where "similarity" means roughly how likely you would pick point j as a neighbor of point i. Second, it finds a low-dimensional arrangement of the same points that preserves these neighborhood relationships as closely as possible, optimizing by gradient descent.

The **perplexity** parameter (typically 5 to 50) controls how many neighbors each point considers. Low perplexity emphasizes very local structure; high perplexity captures broader patterns. Results can look qualitatively different at different perplexity values, so it is good practice to try several.

Critical limitations: t-SNE preserves local structure only. Distances *between* clusters in a t-SNE plot are meaningless. Two clusters that appear far apart may be close in the original space, and vice versa. Cluster sizes are also unreliable: t-SNE tends to expand dense regions and compress sparse ones. Use t-SNE to ask "are there clusters?" but not "how far apart are they?" or "how large are they?"

6.9 Comparing Unsupervised Methods

Different unsupervised methods answer different questions. Understanding their objectives prevents misuse.

Method	Objective	Preserves	Best For
K-Means	Minimize within-cluster variance	Cluster membership	Spherical, well-separated clusters
Hierarchical	Build tree of nested clusters	Multi-scale structure	Unknown k; dendrogram visualization
PCA	Maximize variance explained	Global linear structure; distances	Dimensionality reduction; preprocessing
t-SNE	Preserve local neighborhoods	Local structure only	Visualization of clusters (2D/3D only)

Table 6.1: Comparison of unsupervised learning methods by objective and use case.

When NOT to Use These Methods

K-Means: Avoid for non-spherical clusters (use DBSCAN or spectral clustering), unknown number of clusters (use hierarchical or silhouette analysis), or very different cluster sizes (small clusters get absorbed).
PCA: Avoid for non-linear structure (use kernel PCA or autoencoders), when visualizing clusters (use t-SNE), or with categorical features (PCA assumes continuous).
t-SNE: Avoid for dimensionality reduction for modeling (not just visualization), interpreting distances between clusters (only local distances meaningful), or large datasets (very slow; use UMAP instead).

6.9.1 Choosing the Right Method

"I want to reduce dimensions for a downstream model" → PCA

"I want to visualize high-dimensional data" → t-SNE (small data) or UMAP (large data)

"I want to group similar items" → K-Means (known k) or Hierarchical (unknown k)

"I want to find outliers" → DBSCAN or isolation forest (not covered here)

6.9.2 Interpreting Clusters with Care

A critical mistake is treating clustering results as if they reveal "true" categories in the data. Clusters are artifacts of the algorithm chosen (K-means finds spherical clusters; DBSCAN finds density-based clusters), the features used (different features yield different clusters), and the hyperparameters (k, distance metric, etc.).

Clusters do NOT necessarily represent natural categories that "exist" in the

data, meaningful distinctions for your application, or stable structure that will appear in new data. Always validate clusters externally: Do the clusters correlate with known labels? Do they make sense to domain experts? Are they stable across different random seeds?

Computing clusters is easy. The harder question is: *so what?* A clustering result is useful only if it leads to different actions for different clusters. Customer segments matter if you would market differently to each segment. Patient subgroups matter if you would treat them differently. If every cluster receives the same intervention, the clustering has not helped you.

Before running a clustering algorithm, ask: "If this produces three groups, what would I do differently for each one?" If you cannot answer that question, the clustering may be a technical exercise without practical value.

6.10 Summary

Key Takeaways

Objective ambiguity: Unlike supervised learning, clustering has no ground truth. Different algorithms, features, and parameters produce different valid clusterings.
Distance and scaling: Distance choice determines what "similar" means. Unscaled features silently dominate distance calculations; always standardize before distance-based clustering.
K-Means: Assigns points to k clusters by minimizing within-cluster variance. Fast but requires choosing k and assumes spherical clusters.
Hierarchical Clustering: Builds a dendrogram of nested clusters. No need to pre-specify k, but computationally expensive for large datasets.
GMM: Soft clustering using Gaussian distributions. Points belong to clusters with probabilities, not hard assignments.
PCA: Finds orthogonal directions of maximum variance.[a] Preserves global linear structure; first few components often capture most information.
t-SNE: Preserves local neighborhoods for visualization. Distances between clusters are meaningless; use only for 2D/3D plots.
Actionability: Clusters are useful only if they lead to different decisions. If every cluster receives the same action, the clustering has not helped.

[a] See Appendix F for eigenvalue decomposition.

6.11 Practice Problems

1. Run K-means by hand on points $\{(1,1),(2,1),(4,3),(5,4)\}$ with initial centroids $(1,1)$ and $(5,4)$. Show each iteration.
2. Calculate Euclidean and Manhattan distances between $(2,5,1)$ and $(6,2,3)$.
3. Explain why K-means can give different results with different initializations. What strategies mitigate this?
4. In GMM, a point has responsibilities $\gamma_1 = 0.8$, $\gamma_2 = 0.15$, $\gamma_3 = 0.05$. What does this tell us compared to K-means?
5. The elbow method shows distortions: $k=1$: 100, $k=2$: 40, $k=3$: 25, $k=4$: 22, $k=5$: 20. What k would you choose?
6. **(Comparison)** For each scenario, which method is most appropriate?
   (a) You want to reduce 1000 features to 50 for a classifier.
   (b) You want to visualize customer segments in 2D.
   (c) You don't know how many clusters exist.
   (d) Clusters have different sizes and elongated shapes.
7. After PCA on 50-dimensional data, the first 3 PCs explain 85% of variance.
   (a) How much information is lost projecting to 3D?
   (b) Why might this tradeoff be acceptable?
8. You see 3 clusters in a t-SNE plot. Your colleague says "Cluster A is closer to B than C." Why might this be wrong?
9. **(Adversarial)** Construct a 2D dataset where K-means fails regardless of initialization. Construct another where single-linkage hierarchical clustering fails.
10. **(Technical-Ethical)** A city uses K-means on resident data for social service allocation. One cluster is 80% one ethnic group.
    (a) Is this evidence of discrimination? What would you need to know?
    (b) Should ethnicity be added as a feature "for diversity"?
    (c) A resident says "the algorithm put me in the wrong cluster." How would you evaluate this claim?

Part III

Advanced Methods

Chapter 7

Kernel Methods

> Mathematics is the art of giving the same name to different things.
>
> Henri Poincaré

7.1 Why Linear Models Fail: A Motivating Example

Before diving into kernel methods, let's see why we need them.

The XOR Problem: Linear Models Cannot Solve This

Consider four points in 2D:

x_1	x_2	Label
0	0	-1
0	1	$+1$
1	0	$+1$
1	1	-1

Try to draw a single straight line that separates the $+1$ points from the -1 points. You cannot, the classes are "interleaved."

The problem: Linear classifiers can only draw straight decision boundaries. Some patterns require curved or more complex boundaries.

One solution: Add a new feature $x_3 = x_1 \cdot x_2$. Now the data becomes:

x_1	x_2	$x_3 = x_1 x_2$	Label
0	0	0	-1
0	1	0	$+1$
1	0	0	$+1$
1	1	1	-1

In this 3D space, the classes *are* linearly separable! The hyperplane $x_3 = 0.5$ (or equivalently, $x_1 x_2 = 0.5$) perfectly separates them.

The insight: By mapping data to a higher-dimensional space with the right features, non-linear problems become linear. But explicitly computing these features can be expensive or impossible if the feature space is very high-dimensional.

Kernel methods solve this: They let us work in high-dimensional feature spaces *without ever computing the features explicitly*.

Why Kernels on IAIO

Kernel methods appear on IAIO because they test deep mathematical understanding:

Common IAIO question types:

- Given a kernel, determine the implicit feature space
- Prove whether a proposed function is a valid kernel
- Analyze how kernel parameters (σ, polynomial degree) affect model behavior
- Connect kernels to other concepts (regularization, SVM, linear algebra)

What you need to master:

- The kernel trick: computing inner products without explicit features
- Properties of valid kernels (positive semi-definiteness)
- Common kernels: polynomial, RBF/Gaussian
- Connection to dual optimization and the representer theorem

Derivation Guide: What's Optional

Must understand (likely to be tested):

- The kernel trick concept (Section 7.4)
- Polynomial and RBF kernel behavior (Section 7.6)
- Valid kernel properties (Section 7.7)

Understand the idea, skip derivation details:

- Ridge regression primal/dual solutions (Section 7.3)
- Representer theorem proof

Optional / enrichment:

- Kernel methods and deep learning connection (Section 7.10)
- Full derivation of dual from primal

7.2 Linear Functions and Their Limitations

The simplest machine learning models use **linear functions**. Given an input represented as a vector of numbers $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, a linear function computes a weighted sum:

$$g(\mathbf{x}) = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = \langle \mathbf{w}, \mathbf{x} \rangle$$

Here, $\mathbf{w} = (w_1, w_2, \ldots, w_n)$ is a **weight vector** that determines how much each input feature contributes to the prediction. The notation $\langle \mathbf{w}, \mathbf{x} \rangle$ denotes the inner product (or dot product) between the two vectors. Multiply corresponding components and add them up.

Learning means finding the weight vector $\mathbf{w}$ that makes the function g produce outputs close to the desired labels on the training data. If we have m training examples $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_m, y_m)$, we want $g(\mathbf{x}_i) \approx y_i$ for all i.

Linear functions have an important geometric interpretation. In two dimensions, the equation $w_1 x_1 + w_2 x_2 = c$ describes a line. In three dimensions, it describes a plane. In higher dimensions, it describes a **hyperplane**. A flat surface that divides the space into two regions. Linear classification finds a hyperplane that separates examples from different classes.

The limitation of linear functions is that many real-world patterns are not linear. Imagine trying to classify points arranged in two concentric circles: the inner circle belongs to class A, and the outer ring belongs to class B. No straight line can separate them. A linear function in the original input space simply cannot express the boundary we need.

This is where **feature spaces** become essential. The key insight is that even if the data cannot be separated by a line in its original representation, it might become separable if we transform it into a different representation. The concentric circles, for example, can be separated by considering the distance from the origin: points close to the center (small distance) are class A, points farther out (large distance) are class B. In this transformed representation, the problem becomes linearly separable.

7.3 Ridge Regression: A Worked Example

To make these ideas concrete, let us work through **ridge regression**, a fundamental algorithm that beautifully illustrates the concepts underlying kernel methods.

In regression problems, the outputs y_i are real numbers rather than categories. We might predict house prices, stock returns, or patient recovery

times. To measure how well our function performs, we use the **squared loss**: for each training example $(\mathbf{x}_i, y_i)$, the error is $(g(\mathbf{x}_i) - y_i)^2$. Squaring ensures errors are always positive and penalizes large errors more heavily than small ones.

However, simply minimizing the sum of squared errors on training data can lead to **overfitting**, the model memorizes the training examples rather than learning generalizable patterns. Ridge regression addresses this by adding a penalty for large weight vectors:

$$\text{Minimize: } \sum_{i=1}^{m}(g(\mathbf{x}_i) - y_i)^2 + \lambda\|\mathbf{w}\|^2$$

The first term measures how well we fit the training data. The second term, $\lambda\|\mathbf{w}\|^2$, is called the **regularizer**, it penalizes the squared length of the weight vector. The parameter λ controls the trade-off: larger λ means stronger regularization, leading to smaller weights but potentially worse fit to the training data.

Why penalize large weights? Intuitively, large weights make the function very sensitive to small changes in the input. A model with huge weights might fit the training data perfectly but behave erratically on new examples. Regularization keeps the weights reasonable, which typically improves generalization.

7.3.1 The Primal Solution

There is a beautiful closed-form solution to ridge regression. If we arrange the training inputs as rows of a matrix $\mathbf{X}$ (so $\mathbf{X}$ is $m \times n$, where m is the number of examples and n is the number of features), and arrange the outputs as a vector $\mathbf{y}$, the optimal weight vector is:

$$\mathbf{w} = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_n)^{-1}\mathbf{X}^\top\mathbf{y}$$

Here, $\mathbf{I}_n$ is the $n \times n$ identity matrix (ones on the diagonal, zeros elsewhere). This formula is called the **primal solution** because it works directly with the weight vector $\mathbf{w}$.

Computing the primal solution requires inverting an $n \times n$ matrix. This is fine when n is small, but becomes expensive or even impossible when n is large. What happens if we want to work in a feature space with millions or billions of dimensions?

7.3.2 The Dual Solution

Here is where something magical happens. It turns out that the optimal weight vector can always be written as a linear combination of the training inputs:

$$\mathbf{w} = \sum_{i=1}^{m} \alpha_i \mathbf{x}_i$$

for some coefficients $\alpha_1, \alpha_2, \ldots, \alpha_m$. This is called the **representer theorem**, and it has profound implications.

Instead of specifying n weights (one for each input dimension), we specify m coefficients (one for each training example). If we have fewer training examples than input dimensions (which is common in practice) this representation is more compact. More importantly, it opens the door to working in very high-dimensional feature spaces.

The coefficients can be computed by solving:

$$\boldsymbol{\alpha} = (\mathbf{K} + \lambda \mathbf{I}_m)^{-1} \mathbf{y}$$

where $\mathbf{K}$ is the $m \times m$ **kernel matrix**. The entry in row i and column j of $\mathbf{K}$ is $\langle \mathbf{x}_i, \mathbf{x}_j \rangle$, the inner product between the ith and jth training examples.

This is the **dual solution**. Instead of inverting an $n \times n$ matrix, we invert an $m \times m$ matrix. When the number of features is much larger than the number of training examples, this is far more efficient.

7.4 The Kernel Trick

The dual solution reveals something profound. Both the training computation and the prediction for a new input $\mathbf{x}$:

$$g(\mathbf{x}) = \sum_{i=1}^{m} \alpha_i \langle \mathbf{x}, \mathbf{x}_i \rangle$$

depend on the data *only through inner products*. We never need the actual coordinates of the vectors, only their inner products with each other.

This observation enables the **kernel trick**, one of the most elegant ideas in machine learning. Suppose we want to work in a higher-dimensional feature space, where each input $\mathbf{x}$ is mapped to a feature vector $\phi(\mathbf{x})$. Normally, if the feature space has millions or billions of dimensions, explicitly computing $\phi(\mathbf{x})$ would be prohibitively expensive.

But what if we could compute the inner product $\langle \phi(\mathbf{x}), \phi(\mathbf{z}) \rangle$ directly from the original inputs $\mathbf{x}$ and $\mathbf{z}$, without ever computing $\phi(\mathbf{x})$ explicitly? Then we could run our algorithm in the high-dimensional feature space at the cost of working in the original input space.

A function $\kappa(\mathbf{x}, \mathbf{z})$ that computes the inner product in some feature space is called a **kernel function**:

$$\kappa(\mathbf{x}, \mathbf{z}) = \langle \phi(\mathbf{x}), \phi(\mathbf{z}) \rangle$$

The kernel trick allows us to perform linear algorithms in feature spaces that may be too large to work with directly, even infinitely dimensional spaces!

7.5 The Quadratic Kernel

The simplest non-trivial example is the **quadratic kernel**:

$$\kappa(\mathbf{x}, \mathbf{z}) = \langle \mathbf{x}, \mathbf{z} \rangle^2$$

We simply compute the ordinary inner product and square the result. One extra multiplication, seemingly a trivial change. Yet this has surprising consequences for what we can represent.

To understand what feature space this corresponds to, let us expand the square. For inputs with two components, $\mathbf{x} = (x_1, x_2)$ and $\mathbf{z} = (z_1, z_2)$:

$$\begin{aligned} \langle \mathbf{x}, \mathbf{z} \rangle^2 &= (x_1 z_1 + x_2 z_2)^2 \\ &= x_1^2 z_1^2 + 2 x_1 x_2 z_1 z_2 + x_2^2 z_2^2 \end{aligned}$$

This equals the inner product between vectors $(x_1^2, \sqrt{2} x_1 x_2, x_2^2)$ and $(z_1^2, \sqrt{2} z_1 z_2, z_2^2)$. So the quadratic kernel corresponds to a feature mapping that includes all products of pairs of input features.

In general, for n-dimensional inputs, the feature space has dimension n^2, we go from n features to n^2 features with one extra multiplication. Consider images: a 32×32 pixel image has $n = 1024$ dimensions. Using the quadratic kernel, we implicitly work in a space of over one million dimensions. Yet the kernel computation requires only the same effort as computing one inner product.

7.6 Polynomial and Gaussian Kernels

The quadratic kernel is just the beginning. Since sums and products of valid kernels are also valid kernels, we can construct increasingly powerful feature spaces.

Polynomial kernels of degree d take the form:

$$\kappa(\mathbf{x}, \mathbf{z}) = (c + \langle \mathbf{x}, \mathbf{z} \rangle)^d$$

where $c \geq 0$ is a constant. The polynomial kernel corresponds to a feature space containing all products of up to d input features. Higher d means more complex patterns can be captured, but also higher risk of overfitting.

The most remarkable kernel is the **Gaussian kernel**, also called the RBF (radial basis function) kernel:

$$\kappa(\mathbf{x}, \mathbf{z}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{z}\|^2}{2\sigma^2}\right)$$

The parameter σ controls the "width" of the kernel, how far apart two points can be while still being considered similar. Points close together (relative to σ) have kernel values near 1; points far apart have kernel values near 0.

The Gaussian kernel corresponds to an *infinite-dimensional* feature space. It can be written as an infinite sum of polynomial kernels, meaning the feature vector includes products of arbitrarily many input features. Despite the infinite dimensionality, we can compute the kernel value with a simple formula involving a distance calculation and an exponential.

Intuitively, the Gaussian kernel implements a kind of "local similarity": two inputs are similar in the feature space if and only if they are close in the original input space. This allows the Gaussian kernel to create arbitrarily complex decision boundaries, adapting to whatever patterns exist in the data.

Choosing a Kernel

Linear kernel ($\kappa(\mathbf{x}, \mathbf{z}) = \langle \mathbf{x}, \mathbf{z} \rangle$): Always start here. If the data is already (approximately) linearly separable, a linear kernel is fast, interpretable, and unlikely to overfit. Many text classification and high-dimensional problems work well with a linear kernel.
Polynomial kernel: Use when you suspect interaction effects between features matter (e.g., $x_1 \cdot x_2$ is informative). This connects directly to the polynomial feature engineering from Chapter 3, but the kernel computes these interactions implicitly.
Gaussian/RBF kernel: The default choice when you have no strong prior about data structure. It can approximate any continuous function given enough data. The risk is overfitting if σ is too small.
Custom kernels: When domain knowledge suggests a specific notion of similarity (e.g., string kernels for DNA sequences, graph kernels for molecular structure), designing a domain-specific kernel can outperform generic choices, especially with limited data.

7.7 Properties of Valid Kernels

Not every function can be a kernel. For $\kappa(\mathbf{x}, \mathbf{z})$ to correspond to an inner product in some feature space, it must satisfy two properties.

Symmetry: The kernel must be symmetric in its arguments:

$$\kappa(\mathbf{x}, \mathbf{z}) = \kappa(\mathbf{z}, \mathbf{x})$$

This follows directly from the symmetry of inner products: $\langle \mathbf{u}, \mathbf{v} \rangle = \langle \mathbf{v}, \mathbf{u} \rangle$.

Positive semi-definiteness: For any set of inputs $\mathbf{x}_1, \ldots, \mathbf{x}_m$ and any real numbers $u_1, \ldots, u_m$:

$$\sum_{i=1}^{m} \sum_{j=1}^{m} u_i u_j \kappa(\mathbf{x}_i, \mathbf{x}_j) \geq 0$$

Equivalently, the kernel matrix $\mathbf{K}$ (formed by evaluating the kernel on all pairs of training inputs) must have non-negative eigenvalues.

These properties ensure that κ actually corresponds to an inner product in some feature space, even if we cannot write down that space explicitly. Mathematically, they guarantee that the optimization problem underlying kernel methods is well-behaved.

Fortunately, we rarely need to verify these properties from scratch. There are useful closure properties that let us build complex kernels from simple ones. If κ_1 and κ_2 are valid kernels, then so are their sum $\kappa_1 + \kappa_2$, their product $\kappa_1 \cdot \kappa_2$, and any positive scalar multiple $c \cdot \kappa$. Any polynomial with non-negative coefficients applied to a kernel yields another kernel.

7.8 Beyond Regression: Other Kernel Algorithms

The kernel trick extends far beyond ridge regression. Any algorithm that accesses the data only through inner products can be "kernelized", modified to work in the feature space defined by a kernel.

Support Vector Machines (SVMs) are classification algorithms that find the hyperplane maximizing the margin between classes. The margin is the distance from the hyperplane to the nearest training points. Large margins tend to correlate with good generalization. There is a direct mathematical connection: the margin equals $2/\|\mathbf{w}\|$, so maximizing the margin is equivalent to minimizing $\|\mathbf{w}\|^2$, which is exactly the regularization penalty from ridge regression. Regularization and margin maximization are two views of the same mechanism. In SVMs, the regularization parameter C controls how

much margin violations are tolerated, playing the inverse role of $\boldsymbol{\lambda}$ in ridge regression.

Kernel PCA extends principal component analysis to the feature space. Ordinary PCA finds directions of maximum variance in the data; kernel PCA finds these directions in the kernel-defined feature space. This enables nonlinear dimensionality reduction: data that cannot be separated or visualized well in the original space might become separable or interpretable after kernel PCA.

The Kernel Perceptron is an online learning algorithm that updates whenever it makes a mistake. Instead of maintaining explicit weights, it maintains the coefficients $\boldsymbol{\alpha_i}$ representing the weight vector as a combination of training examples. This allows the perceptron to learn nonlinear decision boundaries.

7.9 Geometric Operations in Feature Space

Working in the kernel-defined feature space, we can perform geometric computations without ever constructing the feature vectors explicitly.

The **distance** between two points in the feature space can be computed from kernel evaluations:

$$\|\boldsymbol{\phi}(\mathbf{x}) - \boldsymbol{\phi}(\mathbf{z})\|^2 = \boldsymbol{\kappa}(\mathbf{x}, \mathbf{x}) - 2\boldsymbol{\kappa}(\mathbf{x}, \mathbf{z}) + \boldsymbol{\kappa}(\mathbf{z}, \mathbf{z})$$

We expand the squared norm as an inner product, then substitute the kernel for each inner product. This formula works regardless of how high-dimensional the feature space is.

The **mean** of the training data in the feature space is:

$$\bar{\boldsymbol{\phi}} = \frac{1}{m} \sum_{i=1}^{m} \boldsymbol{\phi}(\mathbf{x}_i)$$

We cannot compute $\bar{\boldsymbol{\phi}}$ explicitly if the feature space is infinite-dimensional. However, we can compute inner products involving the mean:

$$\langle \bar{\boldsymbol{\phi}}, \boldsymbol{\phi}(\mathbf{x}) \rangle = \frac{1}{m} \sum_{i=1}^{m} \boldsymbol{\kappa}(\mathbf{x}_i, \mathbf{x})$$

This is enough for many purposes. We can even **center** the data in the feature space (shift the origin to the mean) by appropriately modifying the kernel matrix.

7.10 Kernel Methods and Deep Learning

Kernel methods dominated machine learning from the 1990s until around 2012, when deep learning began achieving superior results on large-scale problems. Understanding the relationship between these approaches provides valuable perspective.

Deep learning offers greater flexibility in the functions it can represent and can learn features automatically from raw data. Given enough data and computation, deep neural networks achieve remarkable results on images, language, and other complex domains.

However, kernel methods retain important advantages in several situations.

When **training data is limited**, perhaps only hundreds or thousands of examples rather than millions. Kernel methods often outperform deep learning. medical applications frequently face this constraint: rare diseases do not generate millions of training examples.

Kernel methods shine when **domain knowledge suggests an appropriate kernel**. If you understand the structure of your problem, you can design a kernel that encodes that structure, potentially achieving better results with less data than a generic deep learning approach.

Kernel methods have **strong theoretical foundations**. We understand precisely what function class they search over and can prove guarantees about their behavior. This interpretability matters in applications where we need to explain or verify our models. A deep neural network's decisions can be difficult to understand, while a kernel method's predictions depend transparently on similarity to training examples.

7.10.1 Why Kernel Methods Lost: The Scaling Wall

The elegance of the kernel trick conceals a practical problem: kernel methods scale poorly with dataset size. The kernel matrix $\mathbf{K}$ has m^2 entries (one for every pair of training examples), and the dual solution requires inverting this $m \times m$ matrix at $O(m^3)$ cost. With $m = 10{,}000$ training points, this is manageable. With $m = 1{,}000{,}000$, the kernel matrix alone requires storing 10^{12} entries (roughly 8 terabytes in double precision), and inversion is computationally infeasible.

Furthermore, kernel methods are **memory-based**: predictions require computing the kernel between the new input and every training example. A neural network stores only its learned weights; a kernel method must retain the entire training set.

Deep learning avoids both problems. Training processes data in small batches

with memory cost independent of dataset size. A trained network stores only its parameters, not the training data. And deep networks learn feature representations automatically, whereas kernel methods require the practitioner to choose the right kernel. This combination of scalability and automatic feature learning is why deep learning displaced kernel methods for large-scale problems in vision, language, and other domains where data is abundant.

7.11 The Bias-Variance Trade-off

A central challenge in all of machine learning is balancing **underfitting** and **overfitting**. This is the same bias-variance tradeoff from Chapter 5, but kernel methods give us two distinct knobs to control it.

If the feature space is too simple (for example, if we use only linear functions when the true pattern is nonlinear) the algorithm cannot capture the relevant patterns. This is underfitting, sometimes called high bias: the model makes strong assumptions that prevent it from fitting the data well.

If the feature space is too complex (for example, if we use a very flexible kernel with a small σ) the algorithm can memorize the training data without learning generalizable patterns. This is overfitting, sometimes called high variance: the model is too sensitive to the specific training examples, producing different results for different random samples from the same distribution.

Kernel methods address this trade-off through two mechanisms.

The **regularization parameter** λ controls how much we penalize large weight vectors. A larger λ constrains the model more strongly, effectively reducing capacity and protecting against overfitting. A smaller λ allows larger weights, increasing capacity but risking overfitting.

The **kernel parameters** (like σ for the Gaussian kernel) also affect the trade-off. A very small σ creates a highly flexible model that can fit almost any training data, but may overfit. A larger σ creates a smoother model that generalizes better but may underfit complex patterns.

Note the connection to Chapter 3: the feature mapping ϕ is doing automatically what manual feature engineering does by hand. Choosing a polynomial kernel of degree 3 is equivalent to creating all cubic interaction features, but without explicitly constructing them. The kernel choice is itself a form of feature engineering.

Finding the right balance typically requires experimentation: trying different parameter settings and evaluating performance on held-out validation data (using the cross-validation techniques from Chapter 5).

7.12 A Worked IAIO-Style Problem

IAIO Problem: Kernel Validity and Feature Spaces

Consider the function $\kappa(x, z) = (xz + 1)^2$ for $x, z \in \mathbb{R}$.
Part (a): Show this is a valid kernel by finding an explicit feature mapping ϕ such that $\kappa(x, z) = \langle \phi(x), \phi(z) \rangle$.
Part (b): What is the dimension of the feature space?
Part (c): If we use this kernel in ridge regression with 100 training points, what size matrix do we invert in the dual formulation? In the primal formulation?

Solution:

Part (a): Expand $(xz + 1)^2 = x^2z^2 + 2xz + 1$.

We can write this as an inner product: $\langle (x^2, \sqrt{2}x, 1), (z^2, \sqrt{2}z, 1) \rangle$.

So $\phi(x) = (x^2, \sqrt{2}x, 1)$ and $\kappa(x, z) = \phi(x)^\top \phi(z)$. ✓

Part (b): The feature space has dimension 3 (three components in ϕ).

7.13 Practice Problems

1. Consider the polynomial kernel of degree 2: $\kappa(\mathbf{x}, \mathbf{z}) = (1 + \mathbf{x}^T\mathbf{z})^2$
   (a) For $\mathbf{x} = (1, 2)$ and $\mathbf{z} = (3, 1)$, compute $\kappa(\mathbf{x}, \mathbf{z})$.
   (b) Expand $(1 + x_1z_1 + x_2z_2)^2$ to show what feature space this kernel implicitly uses.
2. For the Gaussian (RBF) kernel $\kappa(\mathbf{x}, \mathbf{z}) = \exp\left(-\frac{\|\mathbf{x}-\mathbf{z}\|^2}{2\sigma^2}\right)$ with $\sigma = 1$:
   (a) Compute the kernel value between $\mathbf{x} = (0, 0)$ and $\mathbf{z} = (1, 1)$.
   (b) What is the kernel value between any point and itself?
   (c) As $\sigma \to 0$, what happens to the kernel values for $\mathbf{x} \neq \mathbf{z}$? What does this mean for the model?
3. A function κ is a valid kernel if and only if the kernel matrix $K_{ij} = \kappa(\mathbf{x}_i, \mathbf{x}_j)$ is positive semi-definite for any set of points.
   (a) Prove that if κ_1 and κ_2 are valid kernels, then $\kappa(\mathbf{x}, \mathbf{z}) = \kappa_1(\mathbf{x}, \mathbf{z}) + \kappa_2(\mathbf{x}, \mathbf{z})$ is also a valid kernel.
   (b) Is $\kappa(\mathbf{x}, \mathbf{z}) = |\mathbf{x}^T\mathbf{z}|$ a valid kernel? Justify your answer.

4. In ridge regression with the dual formulation, the prediction for a new point $\mathbf{x}$ is:

$$f(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i \kappa(\mathbf{x}_i, \mathbf{x})$$

where the α_i are learned from training data.

   (a) Explain in words what this formula is computing.

   (b) Why does this become computationally advantageous when the feature dimension d is much larger than the number of training points n?

5. Consider a 1D dataset: $\{(-1, -1), (0, 1), (1, -1)\}$ where each pair is (x, y).

   (a) Can this data be fit perfectly by a linear function? Why or why not?

   (b) Describe a feature mapping $\phi(x)$ that would make this data linearly separable.

   (c) What kernel corresponds to your feature mapping?

6. The regularization parameter λ in ridge regression controls model complexity.

   (a) What happens to the model as $\lambda \to \infty$?

   (b) What happens as $\lambda \to 0$?

   (c) How would you choose an appropriate value of λ in practice?

7. Explain why kernel methods are sometimes called "memory-based" methods. What are the advantages and disadvantages of this property compared to methods like neural networks that learn explicit weight parameters?

8. **(Geometry Only. No Algebra Required)**

   Consider data points in 2D arranged in two concentric circles: Class A forms the inner circle (radius 1), Class B forms the outer circle (radius 2).

   (a) Sketch this data. Can any straight line separate the classes? Explain geometrically.

   (b) If you add a third dimension $x_3 = x_1^2 + x_2^2$ (the squared distance from origin), describe what the data looks like in 3D. Can a plane separate the classes now?

(c) The RBF kernel $\kappa(\mathbf{x}, \mathbf{z}) = \exp(-\|\mathbf{x} - \mathbf{z}\|^2 / 2\sigma^2)$ implicitly maps to an infinite-dimensional space. Without any algebra, explain intuitively why this kernel can separate the concentric circles.

(d) As $\sigma \to 0$, the RBF kernel approaches a situation where each training point only "sees" itself. What does this mean for the decision boundary? Is this good or bad for generalization?

9. **(Adversarial thinking)** A colleague claims to have invented a new kernel: $\kappa(x, z) = \min(x, z)$ for $x, z \geq 0$.

(a) Is this a valid kernel? (Hint: check if the kernel matrix is positive semi-definite for the points $\{1, 2, 3\}$.)

(b) If it is valid, what feature space does it correspond to? If not, what goes wrong when you try to use it?

Chapter 8

AI Search and Constraint Satisfaction

> A problem well stated is a problem half solved.
>
> George Pólya

8.1 The Nature of Search Problems

Imagine you are playing a video game where you need to navigate a character through a maze to find treasure. At each step, you can move up, down, left, or right. Some paths lead to dead ends, others loop back on themselves, and only a few lead to the treasure. How do you find the best path. The one that reaches the treasure in the fewest moves or avoids the most dangers?

This simple scenario captures the essence of what computer scientists call a **search problem**. At its core, a search problem involves finding a sequence of actions that transforms an initial situation into a desired goal situation. The challenge lies not just in finding *any* solution, but in finding a *good* solution. One that is short, cheap, fast, or optimal according to whatever criteria matter for the problem at hand.

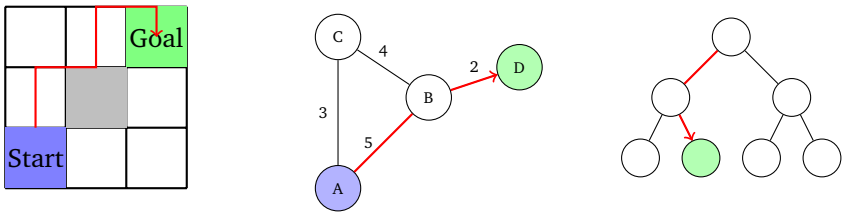
Search problems appear throughout artificial intelligence and computer science. When your GPS calculates a route from your home to a restaurant, it is solving a search problem: finding a sequence of roads that connects your starting location to your destination while minimizing travel time or distance. When a chess program decides which move to make, it is searching through possible future game states to find a move that leads to victory. When a robot plans how to navigate through a building, it searches for a path that avoids

obstacles and reaches its target.

What is a Search Problem?

A **search problem** is any problem where you need to find a sequence of steps (a path) from a starting point to a goal. The challenge is finding the path that best satisfies your criteria, whether that means the shortest path, the cheapest path, or simply any path that works.

What makes search problems interesting is that they are often too large to solve by simply trying every possibility. A chess game can unfold in more ways than there are atoms in the universe. Even a simple maze might have millions of possible paths. This **state space explosion**, the exponential growth of possibilities with problem size, is the central reason search is hard. The art of AI search lies in exploring the space of possibilities intelligently, finding good solutions without exhaustively examining every option.



Robot Navigation GPS Navigation Game Playing

Figure 8.1: Three examples of search problems: navigating a robot through a maze, finding driving routes between cities, and exploring game decision trees.

8.2 State Space Representation

Before a computer can search for a solution, we need to describe the problem in a way the computer can understand and manipulate. This mathematical description is called a **state space representation**, and it transforms the intuitive notion of "finding a path" into a precise computational framework.

8.2.1 Thinking in States and Actions

The key insight is to think of the world as existing in different **states**, and to think of problem-solving as moving from one state to another through **actions**. A state is a complete description of the current situation, everything the algorithm needs to know to continue solving the problem. An action is something that changes the state.

Consider the example of navigating between cities. Each city represents a

state: you are either in Arad, or in Bucharest, or in Sibiu, and so on. The actions are the roads connecting cities: from Arad, you can drive to Sibiu, or to Timisoara, or to Zerind. Each action takes you from one state to another.

This way of thinking transforms the problem into a graph, where nodes represent states and edges represent actions. Finding a solution becomes equivalent to finding a path through this graph from the starting node to a goal node.

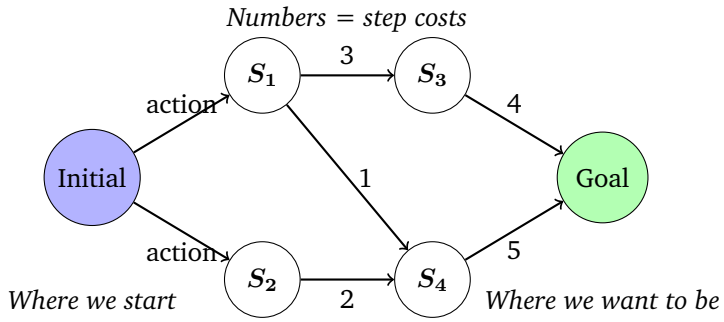


Figure 8.2: A state space represented as a graph. Nodes are states, edges are actions, and numbers indicate the cost of each action.

8.2.2 The Components of a Search Problem

A complete search problem specification includes several components. The **state** is the fundamental unit. A description of a particular configuration of the world. For our city navigation example, a state is simply which city you are currently in. For a chess game, a state would be the complete position of all pieces on the board.

The **initial state** is where the search begins. This is your starting point. Arad in our navigation example, or the opening position in chess.

The **goal state** (or states) is what we are trying to reach. Sometimes there is exactly one goal, like arriving in Bucharest. Other times, multiple states qualify as goals. In chess, any checkmate position is a goal.

The **actions** or **operators** are the ways we can move from one state to another. From a given city, the available actions are driving to adjacent cities. In chess, the actions are all legal moves from the current position.

The **transition model** specifies what happens when we take an action in a state. Given a state and an action, the transition model tells us the resulting state (called the **successor** or **child** state). If you are in Arad and take the action “drive to Sibiu,” the transition model tells you that you end up in Sibiu.

The **step cost** assigns a numerical cost to each action. This might be the distance traveled, the time required, the fuel consumed, or any other measure of effort. In our city example, step costs are road distances. If all actions are equally costly, we might simply assign each a cost of 1.

A **path** is a sequence of states connected by actions, leading from one state to another. The **path cost** is the sum of all step costs along the path. Our goal is typically to find a path from the initial state to a goal state that minimizes the total path cost.

A search problem consists of several components. The **state** is a complete description of a particular configuration of the world. The **initial state** is the starting configuration where the search begins, and the **goal state** is the configuration (or set of configurations) we want to reach. **Actions** are the operations that transform one state into another, and the **transition model** is the function that determines the resulting state when an action is applied. Finally, the **step cost** is the cost associated with each action, and the **path cost** is the total cost of a sequence of actions (the sum of step costs).

8.2.3 A Concrete Example: The Romania Map

A classic example used in AI textbooks involves finding a route across Romania. Suppose you are in the city of Arad and want to reach Bucharest, the capital.

In this problem, the states are the cities of Romania: Arad, Sibiu, Fagaras, Pitesti, Bucharest, and others. The initial state is Arad, and the goal state is Bucharest. The actions are driving along roads between adjacent cities. The step costs are the road distances. For instance, the road from Arad to Sibiu is 140 kilometers.

How does a computer represent this information? The most common approach uses an **adjacency matrix**, a two-dimensional array where rows and columns represent cities. The value at row i, column j is the distance from city i to city j. If two cities are not directly connected by a road, we represent this with infinity (∞) or a special value indicating no connection.

This representation allows the computer to quickly look up whether two cities are connected and what the cost of traveling between them is. From this data structure, the search algorithm can systematically explore routes from Arad to Bucharest, eventually finding the shortest path.

8.2.4 Tree Search vs. Graph Search

When exploring the state space, we face an important choice about how to handle states we have already visited.

In **tree search**, we treat each path as independent and do not remember which states we have visited before. This can lead to problems: if the graph contains cycles, we might visit the same state repeatedly, potentially getting stuck in an infinite loop. Imagine a road network where Arad connects to Sibiu, and Sibiu connects back to Arad. A naïve tree search might endlessly cycle between these two cities.

Graph search solves this problem by maintaining two data structures. The **frontier** (also called the **open list**) contains states that have been discovered but not yet fully explored. The **explored set** (also called the **closed list**) contains states that have already been visited and fully processed. Before exploring a state, the algorithm checks whether it has been visited before. If so, the state is skipped.

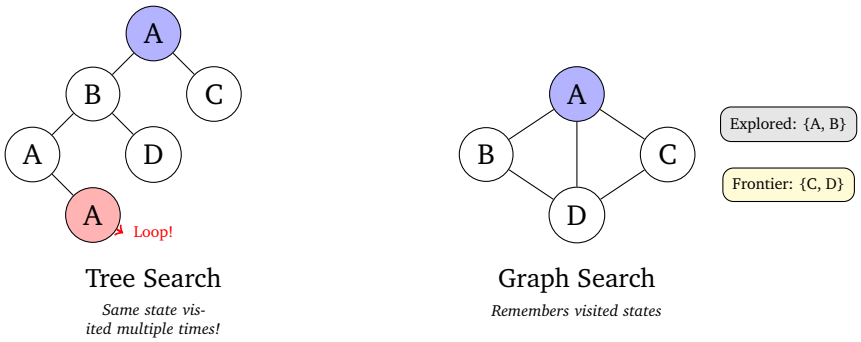


Figure 8.3: Tree search can revisit the same states multiple times, potentially causing infinite loops. Graph search maintains an explored set to avoid this problem.

Graph search uses more memory (to store the explored set) but guarantees termination on finite graphs. The choice between tree and graph search depends on the specific problem: if the state space is a tree with no repeated states, tree search suffices; if repeated states are possible, graph search is usually necessary.

8.3 Measuring Algorithm Performance

With multiple search algorithms available, how do we decide which one to use? Computer scientists evaluate search algorithms using four fundamental criteria.

Completeness asks whether the algorithm is guaranteed to find a solution if one exists. An incomplete algorithm might fail to find a solution even when one is available, perhaps because it gets stuck in a loop or explores the wrong part of the search space indefinitely.

Optimality asks whether the algorithm finds the best solution. Finding *a* solution is often not enough; we want the shortest path, the cheapest route, or the fastest plan. An optimal algorithm guarantees that no better solution exists.

Time complexity measures how long the algorithm takes to find a solution. We typically express this as a function of the problem size, using Big-O notation. An algorithm with exponential time complexity becomes impractical for large problems.

Space complexity measures how much memory the algorithm requires. Some algorithms remember every state they have ever seen, while others discard states after processing them. Memory constraints can be just as limiting as time constraints in practice.

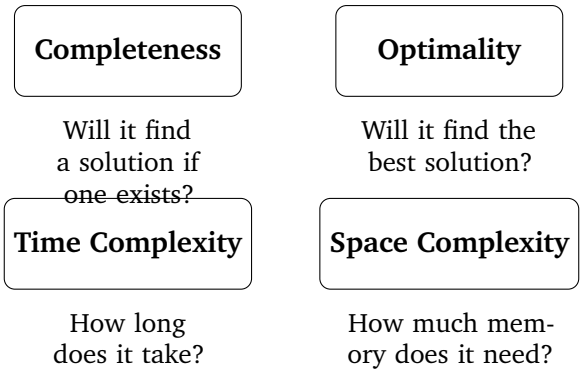


Figure 8.4: The four criteria for evaluating search algorithm performance.

To analyze complexity, we need to characterize the size of the search space. Three key parameters are commonly used. The **branching factor** b is the maximum number of successors (children) any state can have. In our city navigation, if each city connects to at most four others, then $b = 4$. The **depth of the solution** d is the number of steps in the shortest solution path. The **maximum depth** m is the longest path in the search space, which might be infinite if cycles exist.

These parameters let us express complexity precisely. An algorithm with time complexity $O(b^d)$ examines a number of states that grows exponentially with the solution depth, doubling d squares the number of states examined.

8.4 Uninformed Search Strategies

Uninformed search algorithms (also called **blind search**) have no knowledge about where the goal might be located. They cannot tell whether they are getting "warmer" or "colder" as they search. All they can do is systematically explore the state space according to some fixed strategy.

Despite this limitation, uninformed search algorithms are important. They provide baseline solutions that work for any problem, they help us understand the fundamental structure of search, and they form the foundation upon which more sophisticated algorithms are built.

8.4.1 Breadth-First Search

Breadth-First Search (BFS) explores the search space level by level. It first examines all states one step from the start, then all states two steps from the start, then three steps, and so on. The algorithm works outward from the starting point like ripples spreading from a stone dropped in a pond.

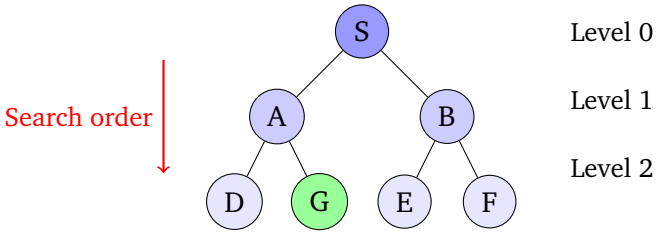


Figure 8.5: Breadth-First Search explores the tree level by level. Starting from S, it examines A and B at level 1, then D, G, E, and F at level 2. The goal G is found at level 2.

The key to BFS is its use of a **queue** data structure, which follows the First-In-First-Out (FIFO) principle. When we discover new states, we add them to the back of the queue. When we need a state to explore, we remove one from the front. This ensures that we process states in the order they were discovered, all states at depth d are processed before any state at depth $d + 1$.

The BFS algorithm proceeds as follows. We begin by placing the initial state in an empty queue. We then enter a loop: remove the front state from the queue, check if it is a goal (if so, we are done), and if not, generate all its successors and add them to the back of the queue. This continues until we find a goal or the queue becomes empty (meaning no solution exists).

Breadth-First Search Algorithm

1. Initialize a queue with just the initial state.
2. Remove the front state from the queue.
3. If this state is the goal, return success and the path to reach it.
4. Otherwise, generate all successor states and add them to the back of the queue.
5. Repeat from step 2 until the goal is found or the queue is empty.

BFS has attractive theoretical properties. It is **complete**, if a solution exists at finite depth, BFS will find it, because it systematically examines all states at each depth before moving deeper. It is also **optimal** when all actions have the same cost, because it finds the shallowest goal first, which is also the goal with the fewest steps.

However, BFS has a serious practical limitation: it requires enormous amounts of memory. The algorithm must store all states at the current level and the next level simultaneously. If the branching factor is b and the solution is at depth d, BFS examines and stores $O(b^d)$ states. This exponential growth quickly becomes unmanageable. For $b = 10$ and $d = 16$, the algorithm would need to store approximately 10^{16} states, more than all the computer memory on Earth.

8.4.2 Uniform Cost Search

What if different actions have different costs? Driving from Arad to Sibiu might be 140 kilometers, while driving from Arad to Zerind is only 75 kilometers. BFS would explore these equally, but clearly the shorter route is preferable.

Uniform Cost Search (UCS) addresses this by always expanding the state with the lowest total path cost from the start. Instead of a simple queue, UCS uses a **priority queue** where states are ordered by their cumulative cost. The state that took the least total effort to reach is always expanded next.

The key quantity in UCS is $g(n)$, the cost of the path from the initial state to state n. When we generate a successor state, we compute its g value by adding the step cost to the parent's g value. The priority queue keeps states sorted by these g values.

One subtle point distinguishes UCS from BFS: we test whether a state is the goal when we *remove* it from the queue, not when we *add* it. This matters because a state might be reached by multiple paths with different costs. We want to ensure we have found the cheapest path before declaring success.

UCS is both complete and optimal, provided all step costs are positive. It

always finds the cheapest path to the goal because it explores states in order of increasing path cost. However, this thoroughness comes at a price: UCS may explore many states that are cheap to reach but far from the goal, while ignoring more expensive states that are much closer.

8.4.3 Depth-First Search

Depth-First Search (DFS) takes the opposite approach from BFS: instead of exploring broadly, it plunges as deep as possible into the search tree before backtracking. From the initial state, DFS follows one path all the way to a dead end or goal, then backs up to the most recent unexplored branch and tries that.

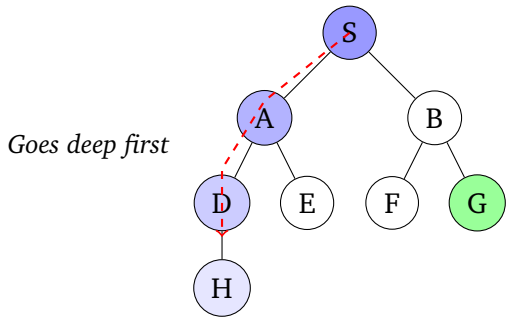


Figure 8.6: Depth-First Search explores one branch completely before trying others. From S, it goes to A, then D, then H, before backtracking to try other branches.

DFS uses a **stack** data structure, which follows the Last-In-First-Out (LIFO) principle. When we discover new states, we push them onto the stack. When we need a state to explore, we pop one from the top. This means the most recently discovered state is always explored next, driving the search deeper rather than broader.

The great advantage of DFS is its memory efficiency. Once the algorithm finishes exploring a branch, it can delete that branch from memory entirely. Only the current path from root to the current state needs to be stored. If the maximum depth is m and the branching factor is b, DFS needs only $O(bm)$ memory. This is linear in the depth, compared to BFS's exponential memory requirement.

Memory Comparison: BFS vs. DFS

Consider a search problem with branching factor $b = 10$ and solution depth $d = 16$.
BFS would need to store approximately 10^{16} states, requiring about 10 exabytes of memory. More than all the digital storage on Earth.
DFS would need to store only about $10 \times 16 = 160$ states, requiring roughly 156 kilobytes. Less than a small image file.
The difference is staggering: DFS uses about 10^{14} times less memory than BFS for this problem.

However, DFS has significant drawbacks. It is **not complete**. If the search space has infinite depth (such as cycles that were not detected), DFS might never find a solution even when one exists at shallow depth. It is also **not optimal**. DFS returns the first solution it finds, which might be much deeper than the shortest solution. DFS could find a path of length 1000 while a path of length 2 exists in an unexplored branch.

8.4.4 Comparing Uninformed Search Strategies

Each uninformed search strategy involves trade-offs:

Algorithm	Complete?	Optimal?	Time	Space
BFS	Yes*	Yes**	$O(b^d)$	$O(b^d)$
UCS	Yes***	Yes	Large	Large
DFS	No	No	$O(b^m)$	$O(bm)$

*For finite spaces. **When all step costs are equal. ***When all step costs are positive.

BFS and UCS guarantee finding optimal solutions but may require impractical amounts of memory. DFS is memory-efficient but may fail to find solutions or find suboptimal ones. In practice, hybrid approaches like iterative deepening combine DFS's memory efficiency with BFS's completeness.

8.5 Informed Search Strategies

Informed search algorithms use additional knowledge about the problem to guide the search toward the goal more efficiently. This additional knowledge comes in the form of a **heuristic function**, an estimate of how close each state is to a goal.

The word "heuristic" comes from the Greek word meaning "to discover." A heuristic is essentially an educated guess, a rule of thumb that often helps but is not guaranteed to be correct. In search, heuristics help us prioritize

which states to explore, focusing our effort on the most promising areas of the search space.

8.5.1 Heuristic Functions

A **heuristic function** $h(n)$ takes a state n and returns an estimate of the cost to reach the nearest goal from that state. For our Romania navigation problem, a natural heuristic is the straight-line distance from each city to Bucharest. This is not the actual driving distance (roads are not straight, and mountains get in the way), but it provides a useful estimate, cities that are geographically closer to Bucharest tend to require shorter drives.

Heuristic Functions

A **heuristic function** $h(n)$ estimates the cost from state n to the nearest goal.
For navigation problems, the straight-line distance to the destination is a common heuristic. Even though you cannot travel in a straight line (there are roads to follow, obstacles to avoid), the straight-line distance gives a useful indication of how far you have to go.
A good heuristic is **admissible**: it never overestimates the true cost to reach the goal. Straight-line distance is admissible because you cannot possibly reach the goal faster than traveling in a straight line.

Two properties make heuristics particularly useful for search. An **admissible** heuristic never overestimates the true cost to reach the goal. It is always optimistic, making the remaining journey seem at least as easy as (or easier than) it actually is. Straight-line distance is admissible because you cannot reach a destination faster than by traveling in a straight line.

A **consistent** heuristic (also called **monotonic**) satisfies a triangle inequality: the estimated cost from n to the goal is never greater than the cost of moving from n to a successor n' plus the estimated cost from n' to the goal. Mathematically, $h(n) \leq c(n, a, n') + h(n')$ for any action a. Consistency implies admissibility and ensures that the first time we reach any state, we have found the optimal path to it.

When Heuristics Make Things Worse

A bad heuristic can make informed search perform worse than uninformed search. If $h(n)$ overestimates costs (inadmissible), A* may explore cheaper-looking states that are actually far from the goal, expanding more nodes than BFS would. Worse, an inadmissible heuristic breaks A*'s optimality guarantee: the algorithm may return a suboptimal solution because it prematurely dismisses the optimal path as too expensive.
Even an admissible but poorly informative heuristic (e.g., $h(n) = 0$ everywhere) reduces A* to Uniform Cost Search, gaining nothing from the heuristic computation. The quality of the heuristic determines how much work A* saves.

Where do good heuristics come from? The most common technique is **relaxation**: remove constraints from the original problem and solve the easier version. The cost of the relaxed solution is always a lower bound on the original cost (hence admissible). Manhattan distance for grid navigation comes from removing obstacles: if you could walk through walls, you would need exactly $|x_1 - x_2| + |y_1 - y_2|$ steps. Straight-line distance for road navigation comes from removing the constraint that you must follow roads. In general, the more constraints you relax, the easier the problem becomes but the less informative the heuristic. A good heuristic relaxes just enough to be easy to compute while staying close to the true cost.

8.5.2 Greedy Best-First Search

Greedy Best-First Search always expands the state that appears to be closest to the goal, according to the heuristic. Its evaluation function is simply $f(n) = h(n)$, the estimated cost to the goal, ignoring the cost already incurred.

This approach can be very fast when the heuristic is accurate. If $h(n)$ perfectly estimates the distance to the goal, greedy search goes straight there. However, greedy search can be fooled by heuristics that point in the wrong direction. It might follow a path that seems promising early on but leads to a dead end or a very expensive final stretch.

Greedy Best-First Search is neither complete nor optimal. It can get stuck in loops (like DFS) and can miss better solutions because it ignores the cost of the path so far. Despite these limitations, it often performs well in practice and provides a useful comparison point for more sophisticated algorithms.

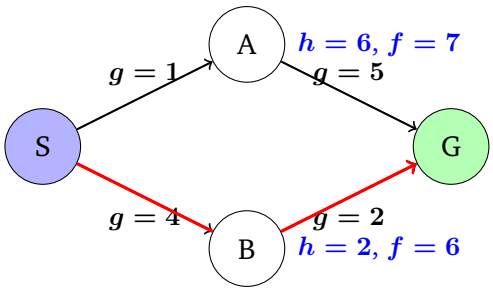
8.5.3 A* Search

A* Search (pronounced "A-star") is perhaps the most important search algorithm in artificial intelligence. It combines the advantages of Uniform Cost Search (which finds optimal paths) with the guidance of heuristics (which focus the search on promising areas).

The key insight of A* is to consider both the past and the future. The cost so far, $g(n)$, tells us how expensive the path to reach state n was. The heuristic estimate, $h(n)$, tells us how expensive we expect the remaining path to be. The total estimated cost of a solution through n is therefore:

$$f(n) = g(n) + h(n) \tag{8.1}$$

A* always expands the state with the lowest f value, the state for which the total estimated solution cost is smallest. This balances exploration of cheap-to-reach states (low g) with exploration of close-to-goal states (low h).



A chooses B (lower f), finding the optimal path*

Figure 8.7: A* evaluates states using $f(n) = g(n) + h(n)$. Although reaching A costs less than reaching B, the total estimated cost through B (f=6) is lower than through A (f=7), so A* correctly explores B first.

The A* Evaluation Function

A* evaluates each state using:

$$f(n) = g(n) + h(n)$$

where $g(n)$ is the actual cost from the start to state n, $h(n)$ is the estimated cost from n to the goal, and $f(n)$ is the estimated total cost of the best solution passing through n.
A* expands states in order of increasing f values, efficiently focusing on the most promising paths.

When the heuristic is admissible, A* is guaranteed to find an optimal solution. This is because A* never expands a state whose f value exceeds the cost of the optimal solution. Such states cannot possibly lead to a better path than one A* has already found or will find.

Even more remarkably, A* is **optimally efficient** among all algorithms that use the same heuristic information and guarantee optimality. No other optimal algorithm can expand fewer nodes than A* (except perhaps by using additional information). This makes A* the gold standard for informed search when an admissible heuristic is available.

8.6 Adversarial Search: Games

So far, we have considered search problems where we have complete control over our actions. But what if there is an **opponent** trying to prevent us from reaching our goal? This situation arises in games like chess, checkers, and go, where two players take turns, each trying to win.

Adversarial search refers to search in competitive environments. The key challenge is that we cannot simply plan a path to victory. We must account for the opponent's responses to our moves. Whatever move we make, the opponent will try to counter it in the way that is worst for us.

8.6.1 The Game Framework

We focus on **two-player zero-sum games** with **perfect information**. In a zero-sum game, one player's gain is exactly the other player's loss. There is no possibility of mutual benefit. In a game with perfect information, both players can see the complete game state (unlike poker, where cards are hidden).

By convention, we call the two players MAX and MIN. MAX is us. We want to maximize our score or utility. MIN is the opponent; from MAX's perspective, MIN wants to minimize MAX's score. (From MIN's own perspective, MIN is trying to maximize their own score, which is the negative of MAX's score.)

The game forms a tree. At each node, one player chooses a move from the available options. The game alternates between MAX nodes (where MAX chooses) and MIN nodes (where MIN chooses). At the leaves of the tree, the game has ended, and we can evaluate who won using a **utility function** that assigns numerical values to outcomes.

8.6.2 The Minimax Algorithm

The **Minimax algorithm** embodies a pessimistic but rational approach: assume the opponent always makes the best possible move against us. Given this assumption, we should choose the move that gives us the best worst-case outcome.

The algorithm works by recursively evaluating the game tree from the bottom up. At each leaf node, we know the utility (who won and by how much). At each MAX node, MAX will choose the action with the highest value, so the node's value is the maximum of its children's values. At each MIN node, MIN will choose the action with the lowest value (from MAX's perspective), so the node's value is the minimum of its children's values.

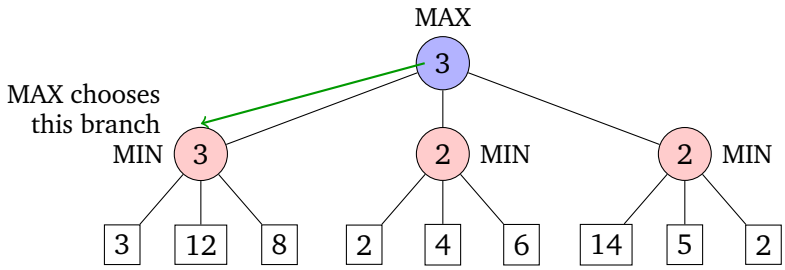


Figure 8.8: The Minimax algorithm in action. Leaf values are game outcomes. MIN nodes take the minimum of their children (choosing what is worst for MAX). MAX nodes take the maximum. The root value of 3 means MAX can guarantee a score of at least 3 with optimal play.

The algorithm proceeds as follows: generate the complete game tree down to terminal positions, evaluate all terminal positions using the utility function, propagate values upward (minimizing at MIN nodes, maximizing at MAX nodes), and finally choose the move at the root that leads to the child with the highest value.

Minimax is complete for finite games and optimal against a rational opponent. However, its time complexity is $O(b^m)$, where b is the branching factor and m is the maximum game length. For chess, with an average branching factor of about 35 and games lasting up to 100 moves, this means examining approximately 35^{100} positions, far more than the number of atoms in the observable universe. Clearly, we cannot search the complete game tree.

8.6.3 Alpha-Beta Pruning

Alpha-Beta Pruning dramatically improves Minimax by proving that certain branches cannot affect the final decision and can therefore be skipped entirely.

If we have already found a good move, we do not need to fully evaluate alternatives that we can prove are worse. Suppose MAX has found a move guaranteed to score at least 5. If we later discover that another move leads to a position where MIN can force a score of 3 or less, we can stop exploring that branch. MAX will never choose it.

The algorithm maintains two values: α represents the best score MAX can guarantee so far (starting at $-\infty$), and β represents the best score MIN can guarantee so far (starting at $+\infty$). At MAX nodes, we update α when we find a better option. At MIN nodes, we update β. Pruning occurs when $\alpha \geq \beta$, the players' guaranteed outcomes have crossed, meaning one player has already found a move that makes the current branch irrelevant.

Alpha-Beta Pruning

The algorithm maintains two bounds:
α is the best value MAX can guarantee so far (starts at $-\infty$)
β is the best value MIN can guarantee so far (starts at $+\infty$)
Pruning occurs when these bounds cross ($\alpha \geq \beta$), meaning:
At a MIN node: if a child's value $\leq \alpha$, prune remaining children (MAX already has a better option elsewhere)
At a MAX node: if a child's value $\geq \beta$, prune remaining children (MIN already has a better option elsewhere)

With perfect move ordering (examining the best moves first), alpha-beta pruning reduces the effective branching factor from b to $\sqrt{b}$, effectively doubling the depth we can search in the same time. In practice, even with imperfect ordering, alpha-beta typically allows searching 1.5 to 2 times deeper than pure Minimax.

8.6.4 Dealing with Complexity: Evaluation Functions

Even with alpha-beta pruning, searching to the end of a chess game is impossible. Real game-playing programs address this by cutting off the search at a certain depth and using an **evaluation function** to estimate the value of non-terminal positions.

A good evaluation function captures the important features of a position. For chess, this might include material (queen = 9 points, rook = 5, bishop or knight = 3, pawn = 1), piece activity (how many squares each piece controls), king safety, pawn structure, and control of the center. The function combines these features into a single numerical estimate of who is winning and by how much.

Modern game-playing programs combine deep search, sophisticated eval-

uation functions, and additional techniques like transposition tables (remembering previously evaluated positions), iterative deepening (searching to increasing depths until time runs out), and opening books (memorized sequences for common game starts). The combination of these techniques has led to computer programs that can defeat the best human players in chess, checkers, Go, and many other games.

8.7 Constraint Satisfaction Problems

Many AI problems have a special structure that we can exploit: they involve finding values for a set of variables that satisfy certain constraints. These are called **Constraint Satisfaction Problems** (CSPs), and specialized algorithms can solve them much more efficiently than generic search.

8.7.1 What is a CSP?

A CSP is defined by three components:

A CSP consists of three components: **variables** $\boldsymbol{X} = \{\boldsymbol{X_1}, \boldsymbol{X_2}, \ldots, \boldsymbol{X_n}\}$, the unknowns we need to determine; **domains** $\boldsymbol{D} = \{\boldsymbol{D_1}, \boldsymbol{D_2}, \ldots, \boldsymbol{D_n}\}$, the possible values for each variable; and **constraints** $\boldsymbol{C}$, rules specifying which combinations of values are allowed. A **solution** is an assignment of values to all variables that satisfies all constraints.

Example: Map Coloring. Color a map so that no two adjacent regions have the same color. Variables are the regions; domains are the available colors (e.g., {Red, Green, Blue}); constraints say adjacent regions must differ.

Example: Sudoku. Fill a 9×9 grid so each row, column, and 3×3 box contains digits 1–9 exactly once. Variables are the empty cells; domains are $\boldsymbol{\{1, 2, \ldots, 9\}}$; constraints enforce the uniqueness rules.

8.7.2 Backtracking Search

The basic algorithm for solving CSPs is **backtracking search**: assign values to variables one at a time, and when a constraint is violated, backtrack to try a different value.

Backtracking Search Algorithm

1. If all variables are assigned, return the solution
2. Select an unassigned variable $\boldsymbol{X_i}$
3. For each value $\boldsymbol{v}$ in the domain of $\boldsymbol{X_i}$:
   - If assigning $\boldsymbol{X_i = v}$ violates no constraints:
     - Add $\boldsymbol{\{X_i = v\}}$ to the assignment
     - Recursively call backtracking search
     - If successful, return the result
     - Otherwise, remove $\boldsymbol{\{X_i = v\}}$ (backtrack)
4. Return failure (no valid assignment exists)

8.7.3 Improving Backtracking

Plain backtracking can be slow because it discovers constraint violations late. Several techniques dramatically improve performance:

Forward Checking: When we assign a value to a variable, immediately remove inconsistent values from the domains of unassigned variables. If any domain becomes empty, backtrack immediately, don't wait until we try to assign that variable.

Example: Forward Checking

Map coloring with regions A, B, C where A–B and B–C are adjacent.
Initial domains: $\boldsymbol{A : \{R, G, B\}}$, $\boldsymbol{B : \{R, G, B\}}$, $\boldsymbol{C : \{R, G, B\}}$
Assign $\boldsymbol{A = R}$. Forward checking removes $\boldsymbol{R}$ from $\boldsymbol{B}$'s domain: $\boldsymbol{B : \{G, B\}}$
Assign $\boldsymbol{B = G}$. Forward checking removes $\boldsymbol{G}$ from $\boldsymbol{C}$'s domain: $\boldsymbol{C : \{R, B\}}$
Now we can assign $\boldsymbol{C = R}$ or $\boldsymbol{C = B}$. Forward checking detected no dead ends early.

Arc Consistency (AC-3): A stronger form of inference. An arc (edge) $\boldsymbol{X_i \to X_j}$ is **arc consistent** if for every value in $\boldsymbol{X_i}$'s domain, there exists some value in $\boldsymbol{X_j}$'s domain that satisfies the constraint between them.

The **AC-3 algorithm** enforces arc consistency throughout the constraint graph:

AC-3 Algorithm

1. Initialize a queue with all arcs (edges) in the constraint graph
2. While the queue is not empty:
   - Remove arc $(\boldsymbol{X_i}, \boldsymbol{X_j})$ from the queue
   - For each value $\boldsymbol{v}$ in $\boldsymbol{D_i}$, check if there exists a value $\boldsymbol{w}$ in $\boldsymbol{D_j}$ satisfying the constraint
   - If no such $\boldsymbol{w}$ exists, remove $\boldsymbol{v}$ from $\boldsymbol{D_i}$
   - If $\boldsymbol{D_i}$ changed, add all arcs $(\boldsymbol{X_k}, \boldsymbol{X_i})$ to the queue (neighbors of $\boldsymbol{X_i}$)
3. If any domain is empty, the CSP has no solution

AC-3 can be run as preprocessing before search, or interleaved with backtracking (called **MAC**. Maintaining Arc Consistency).

8.7.4 Variable and Value Ordering

The order in which we choose variables and values significantly affects performance:

Minimum Remaining Values (MRV): Choose the variable with the fewest legal values remaining. This focuses on the most constrained variable first, if it has no valid values, we fail fast.

Least Constraining Value: When choosing a value for a variable, prefer values that rule out the fewest choices for neighboring variables. This maximizes flexibility for future assignments.

8.7.5 CSPs and SAT

8.8 Complexity Comparison

Different search strategies trade off time, space, and solution quality.

Algorithm	Time	Space	Complete?	Optimal?
BFS	$O(b^d)$	$O(b^d)$	Yes	Yes (uniform cost)
DFS	$O(b^m)$	$O(bm)$	No (infinite)	No
Uniform Cost	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	Same	Yes	Yes
A*	$O(b^d)$ (with good h)	$O(b^d)$	Yes	Yes (admissible h)
IDA*	$O(b^d)$	$O(bd)$	Yes	Yes (admissible h)

Table 8.1: Search algorithm complexity. b = branching factor, d = solution depth, m = max depth, C^* = optimal cost.

The choice of search algorithm involves a fundamental time-space tradeoff.

BFS and A* are optimal but memory-intensive since they store the entire frontier. DFS is memory-efficient ($O(bm)$) but may find suboptimal solutions or loop forever. IDA* offers the best of both worlds: optimal with linear space, making it ideal when memory is constrained. As a rule of thumb, if b^d fits in memory, use A*; otherwise, use IDA*.

8.9 Connecting CSP, SAT, and Logic

Constraint Satisfaction Problems connect deeply to propositional logic and satisfiability.[1]

CSP $\rightarrow$ SAT translation:

- Each variable-value pair becomes a propositional variable: $X_i = v$ becomes $x_{i,v}$
- Domain constraints: At least one value, at most one value
- Problem constraints: Translate directly to clauses

Example: Graph Coloring as SAT

Color nodes $\{A, B\}$ with colors $\{R, G\}$, constraint $A \neq B$.
Variables: A_R, A_G, B_R, B_G ("A is Red," "A is Green," etc.)
Clauses:

- $A_R \vee A_G$ (A has some color)
- $\neg A_R \vee \neg A_G$ (A has at most one color)
- $B_R \vee B_G$, $\neg B_R \vee \neg B_G$ (same for B)
- $\neg A_R \vee \neg B_R$ (if A is Red, B is not Red)
- $\neg A_G \vee \neg B_G$ (if A is Green, B is not Green)

Any satisfying assignment to this CNF formula is a valid coloring.

This connection matters because:

- SAT solvers are highly optimized and can solve large CSPs
- Theoretical results about SAT (NP-completeness) apply to CSPs
- Hybrid approaches combine CSP propagation with SAT techniques

[1] See Appendix E for propositional logic, CNF, and SAT.

Ethics Checkpoint: Algorithmic Decision Systems

A city uses a CSP-based system to allocate limited social services. Variables represent families; constraints encode eligibility rules and resource limits.

Technical-ethical questions:

1. The CSP is infeasible. More eligible families than resources. The system must choose which constraints to relax. Who should make this choice: the algorithm designer, city officials, or affected communities?
2. A constraint says "prioritize families with children under 5." This is correlated with ethnicity in this city. Is the constraint discriminatory? Does intent matter?
3. The system is deterministic. Same inputs always give same outputs. A family argues this is unfair because they have no chance of a different outcome. Is randomization more fair?
4. How would you design an appeals process for families who believe the algorithm treated them unfairly? What information would they need access to?

This is the type of technical-ethical hybrid problem that appears on IAIO. Note how CSP concepts (constraints, feasibility, relaxation) connect to ethical considerations.

Search vs. Learning: Two Paradigms

Search and learning represent fundamentally different approaches to intelligent behavior.

Search (this chapter) explores a known state space using defined rules. Given a correct problem specification, search algorithms can find provably optimal solutions. The limitation is brittleness: if the state space is too large to explore, or if the problem specification is incomplete, search fails.

Learning (Chapters 4–7) discovers patterns from data without an explicit state space. Learning methods are approximate and adaptive: they handle noise, generalize to new situations, and scale to problems where exhaustive search is impossible. The cost is that solutions come with no optimality guarantee.

Modern AI systems increasingly combine both: learning to construct heuristics that guide search, or using search to generate training data for learning algorithms.

8.10 Summary

Search is indeed the most fundamental problem-solving technique in artificial intelligence. By representing problems as state spaces and systematically exploring paths from initial states to goals, search algorithms can solve an enormous variety of problems.

Uninformed search strategies like Breadth-First Search, Uniform Cost Search, and Depth-First Search explore without guidance. BFS and UCS guarantee optimal solutions but require exponential memory. DFS is memory-efficient but may miss solutions or find suboptimal ones.

Informed search strategies use heuristics to guide exploration toward the goal. Greedy Best-First Search follows the heuristic but can be misled. A* Search combines path cost with heuristic estimates, achieving optimality with maximum efficiency when the heuristic is admissible.

Adversarial search addresses competitive situations where an opponent tries to prevent us from winning. The Minimax algorithm finds optimal strategies by assuming the opponent plays perfectly. Alpha-Beta pruning dramatically reduces computation by proving that certain branches cannot affect the final decision.

Constraint Satisfaction Problems (CSPs) exploit problem structure where we assign values to variables subject to constraints. Backtracking search systematically tries assignments, and techniques like forward checking and arc consistency (AC-3) prune the search space dramatically.

Key Takeaways

State Space Representation: Search problems are modeled as graphs with states, actions, and costs.
Uninformed Search: BFS is complete and optimal (for equal costs) but memory-intensive. DFS is memory-efficient but not complete or optimal. UCS finds optimal paths with varying costs.
Informed Search: Heuristics guide search toward the goal. A* combines past cost $g(n)$ with future estimate $h(n)$ to achieve optimal, efficient search.
Adversarial Search: Minimax assumes optimal opponent play. Alpha-beta pruning eliminates provably irrelevant branches, often doubling searchable depth.
Constraint Satisfaction: CSPs involve variables, domains, and constraints. Backtracking with forward checking and arc consistency (AC-3) efficiently finds solutions or proves none exists.

8.11 Practice Problems

1. Consider a graph with nodes A, B, C, D, E where A is the start and E is the goal. The edges are: A-B (cost 2), A-C (cost 1), B-D (cost 3), C-D (cost 2), D-E (cost 1).

   (a) In what order does BFS explore the nodes?

   (b) In what order does DFS explore the nodes, using alphabetical order to break ties?

   (c) What is the optimal path from A to E, and what is its total cost?

2. Using the same graph as Problem 1, suppose the heuristic values are: $h(A) = 5$, $h(B) = 3$, $h(C) = 4$, $h(D) = 1$, $h(E) = 0$.

   (a) Calculate $f(n) = g(n) + h(n)$ for each node when it is first discovered.

   (b) In what order does A* explore the nodes?

3. In a Minimax game tree, MAX moves first and has two choices. Choice 1 leads to a MIN node with children having values {7, 3, 9}. Choice 2 leads to a MIN node with children having values {5, 8, 2}. What move should MAX make, and what value will MAX achieve?

4. Using alpha-beta pruning on the tree from Problem 3: if we explore Choice 1 first, what value does α have when we start exploring Choice 2? After seeing the first child (value 5) of Choice 2, can we prune the remaining children? Explain your reasoning.

5. Explain what it means for A* to be "optimally efficient." Why is this an important property?

6. A robot navigates a 5×5 grid, starting at the bottom-left corner and trying to reach the top-right corner. It can move up, down, left, or right (no diagonals), and each move costs 1.

   (a) What is the branching factor b for this problem? (Hint: consider edges and corners separately from interior cells.)

   (b) The Manhattan distance heuristic computes $|x_1 - x_2| + |y_1 - y_2|$, the sum of horizontal and vertical distances. Is this heuristic admissible for this problem? Explain why or why not.

7. **(CSP)** Consider a map coloring problem with four regions A, B, C, D. The adjacencies are: A–B, A–C, B–C, B–D, C–D. We have three colors: Red, Green, Blue.

(a) How many total possible assignments are there (ignoring constraints)?

(b) Using backtracking with forward checking, start by assigning $\boldsymbol{A = Red}$. What values are removed from which domains?

(c) Continue by assigning $\boldsymbol{B = Green}$. Now what are the remaining domains for C and D?

Chapter 9

Reinforcement Learning

> The essence of strategy is
> choosing what not to do.
>
> — Michael Porter

9.1 Introduction

Reinforcement learning (RL) is a third paradigm of machine learning, distinct from supervised and unsupervised learning. Instead of learning from labeled examples or discovering structure in data, an RL **agent** learns by **interacting with an environment**, taking actions, receiving rewards, and discovering what works through trial and error.

The Reinforcement Learning Paradigm

An agent observes states, takes actions, and receives rewards. The goal: learn a **policy** (mapping from states to actions) that maximizes cumulative reward.
Key difference from other paradigms:

- No labeled "correct" actions (unlike supervised learning)
- Clear objective: maximize reward (unlike unsupervised learning)
- Sequential decisions: actions affect future states

9.2 The Reinforcement Learning Framework

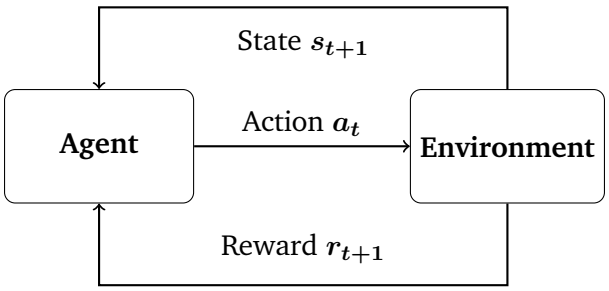


Figure 9.1: The reinforcement learning loop: at each time step, the agent observes the current state, chooses an action, and receives both a reward and a new state from the environment.

The **agent** is the decision-maker: the entity that learns and chooses actions. The agent is the "brain," not the physical body. If we are training a robot to walk, the agent is the control algorithm, not the robot's legs and motors. The robot's physical body is part of the environment.

The **environment** is everything outside the agent. The external world that the agent interacts with. The environment receives the agent's actions, updates its internal state accordingly, and provides feedback in the form of rewards. From the agent's perspective, the environment is a black box: the agent can observe states and rewards, but it may not know the internal rules governing how the environment works.

A **state** is a representation of the current situation. In a chess game, the state is the current board position. In a self-driving car, the state might include the car's position, velocity, and the locations of nearby vehicles and obstacles. The state should contain all information relevant to making good decisions. If important information is missing from the state, the agent will struggle to learn effectively.

An **action** is something the agent can do. In chess, actions are legal moves. In a video game, actions might be "move left," "move right," "jump," or "shoot." The set of available actions may depend on the current state. You cannot castle in chess if you have already moved your king.

A **reward** is a scalar (single number) feedback signal indicating how good the last action was. Higher rewards are better. The reward function encodes the goal of the task: in a game, you might receive $+1$ for winning and -1 for losing; in robot locomotion, you might receive reward proportional to forward progress minus energy expenditure. Designing good reward functions is one of the key challenges in applying reinforcement learning to

real problems.

Reward Hacking: When Optimization Goes Wrong

RL agents optimize the reward function literally, not the designer's intent. If there is any gap between what you *measure* and what you *want*, the agent will exploit it.

A robot rewarded for forward velocity might learn to fall forward rather than walk. A game agent rewarded for score might exploit physics glitches rather than play as intended. A cleaning robot rewarded for "no visible dirt" might learn to close its eyes.

This is not a bug in the algorithm; the agent is doing exactly what it was told to optimize. The problem is that specifying rewards that truly capture desired behavior is surprisingly difficult. This connects to the broader alignment challenges discussed in Chapter 2: the gap between what we specify and what we actually want.

Finally, a **policy** is the agent's strategy, a rule for choosing actions based on states. A policy can be deterministic (always take the same action in a given state) or stochastic (choose actions according to some probability distribution). The goal of reinforcement learning is to find a policy that maximizes cumulative reward.

9.2.1 A Concrete Example: Navigating a Maze

To make these concepts concrete, consider a robot navigating a simple grid world.

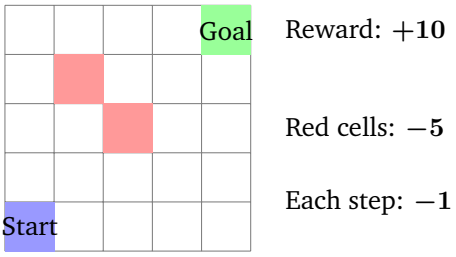


Figure 9.2: A maze environment. The agent starts in the bottom-left corner and must reach the goal in the top-right corner while avoiding red trap cells.

In this environment, the **states** are the 25 grid cells, the robot's position. The **actions** are the four movement directions: up, down, left, and right. The **rewards** are designed to encourage reaching the goal quickly while avoiding traps: +10 for reaching the goal, -5 for stepping on a red trap cell, and -1 for each step taken (to encourage efficiency).

Notice how the reward function shapes the desired behavior. The -1 per step reward means that even successful paths to the goal have a cost, so the agent is incentivized to find *short* paths, not just any path. The -5 trap penalty is large enough to discourage stepping on traps but not so large that the agent would prefer wandering forever to risking a trap.

9.3 Returns and the Discount Factor

The agent's goal is to maximize reward, but which reward? The immediate reward from the next action? The total reward over the next ten actions? The total reward forever?

In reinforcement learning, we typically want to maximize the **return**, the cumulative reward from the current time onward. If the agent receives rewards $R_{t+1}, R_{t+2}, R_{t+3}, \ldots$ after time t, the simplest definition of return would be their sum:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots$$

However, this simple sum creates problems. If the task continues forever, the sum might be infinite, making it impossible to compare different policies. Even if the task eventually ends, treating a reward received in one year identically to a reward received tomorrow seems wrong, surely sooner is better?

The solution is to introduce a **discount factor** γ (gamma), a number between 0 and 1 that reduces the value of future rewards:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Each additional time step into the future multiplies the reward by another factor of γ. A reward received k steps in the future is worth γ^k times as much as the same reward received immediately.

The discount factor has several important effects. When $\gamma = 0$, the agent becomes completely myopic, caring only about the immediate next reward and ignoring all future consequences. When $\gamma = 1$, all rewards are weighted equally regardless of when they occur. Typical values lie between 0.9 and 0.99, meaning the agent cares about the future but prefers sooner rewards to later ones.

Why discount future rewards? There are several justifications. First, discounting ensures that the infinite sum converges to a finite value (as long as rewards are bounded), making the mathematics tractable. Second, in many

real-world situations, the future is uncertain. A promised reward tomorrow might not materialize, so it makes sense to value it less than a reward in hand today. Third, discounting captures a kind of impatience that aligns with human preferences: most people would rather have €100 today than €100 next year.

9.4 Policies and Value Functions

A **policy** tells the agent what to do in each situation. Formally, a policy π is a mapping from states to actions. A deterministic policy specifies exactly one action for each state: $\pi(s) = a$ means "in state s, take action a." A stochastic policy specifies a probability distribution over actions: $\pi(a|s)$ gives the probability of taking action a when in state s.

But how do we evaluate whether a policy is good? This is where **value functions** become essential. A value function tells us how much cumulative reward we can expect to receive from a given situation if we follow a particular policy.

The **state value function** $V^\pi(s)$ answers the question: "If I am in state s and I follow policy π from now on, what is my expected return?"

$$V^\pi(s) = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right]$$

The expectation accounts for any randomness. In the policy (if stochastic), in the environment's responses, or both. The value function provides a single number summarizing how good it is to be in a particular state under a particular policy.

The **action value function** $Q^\pi(s, a)$ answers a slightly different question: "If I am in state s, take action a, and then follow policy π afterward, what is my expected return?"

$$Q^\pi(s, a) = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right]$$

The Q-function (as it is often called) is particularly useful because it directly tells us which action is best: just pick the action with the highest Q-value.

Intuition for Value Functions

Think of value functions as answering different questions:
$V(s)$ asks: "How good is it to **be in** state s?" This summarizes the long-term prospects from that state.
$Q(s, a)$ asks: "How good is it to **take action** a in state s?" This directly guides decision-making.
If we know the Q-function, we can derive an optimal policy: just always choose the action with the highest Q-value.

9.5 The Markov Property and MDPs

Reinforcement learning problems are typically formalized as **Markov Decision Processes** (MDPs). The key assumption underlying MDPs is the **Markov property**: the future depends only on the present state, not on the history of how we got there.

The Markov Property

A state has the Markov property if it contains all information needed to predict the future. Mathematically:

$$P(S_{t+1}|S_t) = P(S_{t+1}|S_t, S_{t-1}, \ldots, S_1)$$

Knowing the entire history provides no additional predictive power beyond knowing the current state.

Consider chess: the current board position is a Markov state because it contains everything needed to determine legal moves and evaluate the position. It does not matter whether we reached this position through a brilliant sacrifice or a series of blunders. Only the current position matters for deciding the next move.

Many real environments violate the Markov assumption. In poker, the state visible to you (your cards and the community cards) does not include your opponents' hidden cards, so the future depends on information not in your observed state. A self-driving car cannot see vehicles hidden behind buildings. A medical treatment decision depends on the patient's full history, not just current symptoms. When the agent cannot observe the full state, the problem becomes a **partially observable MDP** (POMDP), which is substantially harder to solve. For this course, we assume full observability, but recognizing when the Markov assumption is violated is important for applying RL in practice.

Formally, a Markov Decision Process is defined by five components: a set

of states $\mathcal{S}$, a set of actions $\mathcal{A}$, transition probabilities $P(s'|s,a)$ specifying the probability of reaching state s' after taking action a in state s, a reward function $R(s,a)$ giving the expected immediate reward, and the discount factor γ.

9.6 The Bellman Equations

The value functions satisfy elegant recursive relationships called the **Bellman equations**. These equations are the mathematical heart of reinforcement learning, expressing how the value of a state relates to the values of its successor states.

Consider the state value function $V^\pi(s)$. The return from state s consists of the immediate reward from taking an action, plus the discounted return from whatever state we end up in. This leads to the **Bellman expectation equation**:

$$v^\pi(s) = \sum_a \pi(a|s) \left[r(s,a) + \gamma \sum_{s'} p(s'|s,a) v^\pi(s') \right]$$

Let us unpack this equation. The outer sum considers all actions, weighted by the probability $\pi(a|s)$ that the policy chooses each action. For each action, we get the immediate reward $r(s,a)$ plus the discounted value of the next state. The inner sum accounts for stochastic transitions: we might end up in different next states with different probabilities.

The Bellman equation expresses a fundamental consistency requirement: the value of a state must equal the expected immediate reward plus the discounted expected value of successor states. If this relationship did not hold, our value estimates would be internally inconsistent.

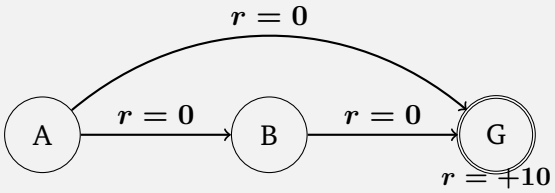
For finding optimal policies, we need the **Bellman optimality equation**:

$$v^*(s) = \max_a \left[r(s,a) + \gamma \sum_{s'} p(s'|s,a) v^*(s') \right]$$

The only difference is that instead of averaging over actions according to some policy, we take the maximum. We assume the agent will always choose the best action. The optimal value function V^* represents the best possible expected return from each state.

Worked Example: Value Iteration by Hand

Setup: A simple 3-state MDP with $\gamma = 0.9$.



From A: can go to B (1 step) or directly to G (shortcut).
From B: can only go to G.
G is terminal with reward +10.
Question: Should the agent take the shortcut A→G or the path A→B→G?

Value Iteration: Initialize $V(A) = V(B) = V(G) = 0$.
Iteration 1:

$$V(G) = 10 \quad \text{(terminal reward)}$$
$$V(B) = \max\{0 + 0.9 \times V(G)\} = 0 + 0.9 \times 10 = 9$$
$$V(A) = \max\{0 + 0.9 \times V(B), \quad 0 + 0.9 \times V(G)\}$$
$$= \max\{0.9 \times 0, \quad 0.9 \times 10\} = \max\{0, 9\} = 9$$

After iteration 1: $V(A) = 9$, $V(B) = 9$, $V(G) = 10$.
Wait; we used old values for $V(B)$ and $V(G)$! Let's re-run with updated values.
Iteration 2:

$$V(G) = 10$$
$$V(B) = 0 + 0.9 \times 10 = 9$$
$$V(A) = \max\{0 + 0.9 \times V(B), \quad 0 + 0.9 \times V(G)\}$$
$$= \max\{0.9 \times 9, \quad 0.9 \times 10\} = \max\{8.1, 9\} = 9$$

Converged! Final values: $V(A) = 9$, $V(B) = 9$, $V(G) = 10$.
Optimal policy: At state A, the action "go directly to G" achieves value 9, while "go to B" achieves only $0.9 \times 9 = 8.1$. So the optimal policy is to **take the shortcut**.
Insight: Even though both paths eventually reach G with reward +10, the shortcut is better because discounting makes sooner rewards worth more than later rewards.

9.7 Exploration vs. Exploitation

One of the deepest challenges in reinforcement learning is the **exploration-exploitation tradeoff**. The agent must balance two competing objectives: **exploiting** its current knowledge to collect rewards, and **exploring** to discover potentially better strategies.

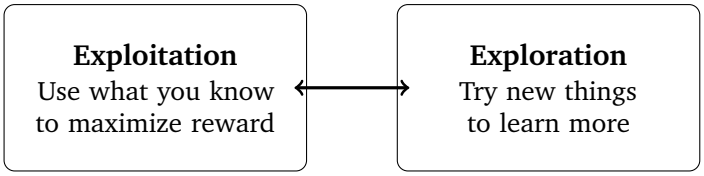


Figure 9.3: The exploration-exploitation tradeoff: exploiting maximizes short-term reward but may miss better strategies; exploring sacrifices immediate reward to gather information.

Imagine you are visiting a new city and choosing restaurants. You could always return to the first decent restaurant you found (exploitation), guaranteeing acceptable meals. But you might miss extraordinary restaurants you never tried. Alternatively, you could try a new restaurant every night (exploration), but you would waste many meals on mediocre places. The optimal strategy involves some of both.

The most common approach to balancing exploration and exploitation is the ϵ**-greedy** strategy. With probability $1 - \epsilon$, the agent chooses the action it currently believes is best (exploitation). With probability ϵ, it chooses a random action (exploration). A typical value might be $\epsilon = 0.1$, meaning the agent explores 10% of the time.

9.8 Dynamic Programming Methods

When we know the environment's dynamics (the transition probabilities $P(s'|s, a)$ and rewards $R(s, a)$) we can compute optimal policies using **dynamic programming**. These methods are called "model-based" because they require a model of how the environment works.

Policy iteration alternates between two steps. The *policy evaluation* step computes the value function V^π for the current policy. The *policy improvement* step creates a better policy by acting greedily with respect to the computed values. These steps repeat until the policy stops changing.

Value iteration combines these steps by repeatedly applying the Bellman

optimality equation:

$$V_{k+1}(s) = \max_a \left[R(s,a) + \gamma \sum_{s'} P(s'|s,a) V_k(s') \right]$$

Dynamic programming methods find optimal solutions but require complete knowledge of environment dynamics, which we rarely have in practice.

9.9 Model-Free Methods: Learning from Experience

When we do not know the environment's dynamics, we must learn directly from experience. These **model-free** methods are among the most important algorithms in reinforcement learning.

Temporal difference (TD) learning updates value estimates based on observed transitions:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \right]$$

The term in brackets is the TD error, the difference between our new estimate and old estimate. The learning rate α controls update speed.

SARSA extends TD learning to action values, using the action actually taken:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \right]$$

Q-learning instead uses the best action's value, regardless of what action was actually taken:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t) \right]$$

The key difference: SARSA is **on-policy** (learns about the policy being followed), while Q-learning is **off-policy** (learns about the optimal policy while following an exploratory policy). Q-learning's off-policy nature makes it more flexible, it can explore freely while still learning the optimal policy.

The Scaling Problem: From Tables to Function Approximation

Tabular Q-learning stores a value for every state-action pair. This works for small problems: a 5×5 grid with 4 actions needs only 100 entries. But an Atari game screen has roughly $\mathbf{10^{20}}$ possible pixel configurations. Storing a Q-table of that size is impossible.

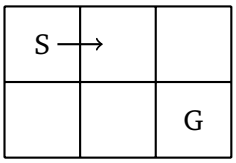
The solution is **function approximation**: instead of a table, use a neural network $Q(s, a; \theta)$ to approximate Q-values, where θ are the network's learned parameters. This is the foundation of **deep reinforcement learning** (Chapter 10 discusses neural networks).

However, function approximation changes the convergence guarantees. Tabular Q-learning is proven to converge to the optimal Q-function. With function approximation, the combination of bootstrapping (updating estimates from estimates), off-policy learning, and function approximation can cause Q-values to oscillate or diverge. This instability, sometimes called the "deadly triad,"[a] means that deep RL algorithms require careful engineering: target networks, experience replay, and gradient clipping are common stabilization techniques. Modern RL is powerful but substantially less predictable than the tabular algorithms presented here.

[a] Sutton, R. S. & Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2nd ed.), Chapter 11.3. MIT Press.

9.9.1 Q-Learning: A Worked Example

Let us trace through Q-learning on a simple grid world. The agent starts at S, wants to reach the goal G, and receives a reward of +10 at G. All other transitions give reward 0. We use $\gamma = \mathbf{0.9}$ and $\alpha = \mathbf{0.5}$.



Initially, all Q-values are 0.

Episode 1: Agent explores randomly: S $\rightarrow$ (1,1) $\rightarrow$ (2,1) $\rightarrow$ (2,0) = G.

When reaching G from (2,1) with action "down":

$$\begin{aligned} Q((\mathbf{2,1}), \text{down}) &\leftarrow \mathbf{0} + \mathbf{0.5} \times [\mathbf{10} + \mathbf{0.9} \times \mathbf{0} - \mathbf{0}] \\ &= \mathbf{0} + \mathbf{0.5} \times \mathbf{10} = \mathbf{5} \end{aligned}$$

Now (2,1) has value for going down. In subsequent episodes, this propagates backward.

Episode 2: Agent reaches (2,1) again, this time from (1,1):

$$Q((1,1), \text{right}) \leftarrow 0 + 0.5 \times [0 + 0.9 \times \max_a Q((2,1), a) - 0]$$
$$= 0 + 0.5 \times [0 + 0.9 \times 5] = 2.25$$

The reward signal is propagating backward from the goal! After many episodes, Q-values throughout the grid will reflect how many steps from the goal each state-action pair is.

9.9.2 Exploration Strategies

The exploration-exploitation trade-off is crucial. Common strategies:

ϵ-greedy: With probability ϵ, choose a random action (explore). With probability $1 - \epsilon$, choose the action with highest Q-value (exploit). Typically start with high ϵ (e.g., 1.0) and decay it toward a small value (e.g., 0.01) as learning progresses.

Softmax: Choose actions with probability proportional to their Q-values:

$$P(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_{a'} e^{Q(s,a')/\tau}}$$

9.10 Summary

Key Takeaways

RL Framework: Agent interacts with environment via states, actions, rewards. Goal: learn policy π maximizing expected return.
Reward Design: The reward function defines the task, but agents optimize it literally. Reward hacking occurs when the agent finds unintended ways to maximize reward without achieving the designer's actual goal.
Returns: $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots$. Discount factor γ balances immediate vs. future rewards.
Markov Property: The future depends only on the current state, not the history. Many real environments violate this assumption (partial observability).
Bellman Equations: Recursive definitions expressing values in terms of immediate rewards and successor values. Foundation of DP and RL algorithms.
Q-Learning: Model-free algorithm that learns Q^* directly from experience. Tabular version converges but does not scale; function approximation enables large problems but introduces instability.
Exploration vs. Exploitation: ϵ-greedy balances trying new actions against using current best knowledge.

9.11 Practice Problems

1. For a 2-state MDP with $\gamma = 0.9$, states A and B, and rewards: A→A gives $+1$, A→B gives 0, B→A gives $+2$, B→B gives $+1$. Write the Bellman equations for $V^*(A)$ and $V^*(B)$ assuming optimal actions.

2. In Q-learning with $\alpha = 0.1$, $\gamma = 0.9$, current $Q(s, a) = 5$. We take action a, receive reward $r = 2$, reach state s' with $\max_{a'} Q(s', a') = 8$. What is the new $Q(s, a)$?

3. Explain the exploration-exploitation tradeoff. Why can't we just always take the best known action?

4. What happens to the optimal policy as $\gamma \to 0$? As $\gamma \to 1$?

5. **(Bellman Derivation)** Starting from $Q^\pi(s, a) = \mathbb{E}[R_{t+1} + \gamma Q^\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a]$, derive the Bellman optimality equation for Q^*.

6. **(Adversarial: When Greedy Intuition Fails)**

   Consider an MDP with three actions from the start state:

- Action A: Immediate reward $+10$, then terminal
- Action B: Immediate reward 0, then move to state X
- Action C: Immediate reward 0, then move to state Y

From state X: only action available gives $+5$ and terminates. From state Y: only action available gives $+20$ and terminates.

Assume $\gamma = \mathbf{0.9}$.

(a) What are the Q-values $Q^*(s_0, A)$, $Q^*(s_0, B)$, $Q^*(s_0, C)$?

(b) A greedy agent that has only tried Action A (receiving $+10$) will keep choosing A forever. Explain why.

(c) After 1000 episodes with $\epsilon = \mathbf{0.1}$, approximately how many times will the agent have tried each action? Will it have learned the optimal policy?

(d) Design a simple modification where greedy behavior would find the optimal policy faster.

7. **(Multi-part)** A robot navigates a 4x4 grid. Goal at (4,4), pit at (2,2). Reward: $+10$ (goal), -10 (pit), -1 (each step).

(a) How many states? How many possible policies?

(b) With $\gamma = \mathbf{0.9}$, will the robot prefer a safe long path or a risky short path? Justify.

(c) Write the Q-learning update for state (1,1), action "right," landing in (2,1) with reward -1.

Chapter 10

Generative Models and Deep Learning

> What I cannot create, I do not understand.
>
> Richard Feynman

10.1 Introduction to Generative AI

The physicist Richard Feynman famously wrote on his blackboard: "What I cannot create, I do not understand." This principle captures the essence of generative artificial intelligence, but inverted: generative models embody the philosophy that *what they understand, they can create*.

At its core, generative AI is about teaching machines to create new content (images, text, music, or any other form of data) that resembles what they have learned from examples. When you give a generative model thousands of photographs of cats, it doesn't merely memorize them. Instead, it learns *what makes a cat look like a cat*: the general shape, the pointed ears, the whiskers, the fur patterns. Armed with this understanding, the model can then create entirely new cat images that have never existed before.

Key Insight

Generative models learn the underlying patterns and distribution of training data, enabling them to generate new samples that are similar to (but not copies of) the original data.

Generative AI now powers many applications you may already use: im-

age generators like DALL-E and Stable Diffusion that create pictures from text descriptions, large language models like ChatGPT that write coherent paragraphs, music composition tools, and video generation systems. Understanding how these systems work is essential for anyone studying modern AI.

Study Guide: What to Master vs. Recognize

This chapter covers multiple architectures. Not all require the same depth of understanding:

Understand at intuition level (be able to explain in your own words):

- Generative vs. discriminative distinction
- What latent space means and why it's useful
- The basic idea of encoding and decoding

Understand technically (be able to work through math):

- VAE loss function: reconstruction + KL divergence
- Why VAEs sample from distributions (reparameterization trick)

GANs, Diffusion models, and Transformers are covered in Chapter 14 (enrichment). Know the concepts, don't memorize the math.

Chapter Scope: Classical Generative Models

This chapter covers **foundational generative architectures**: Autoencoders and VAEs. These are more likely to appear on IAIO because they have clear mathematical formulations.

GANs, Diffusion models, Transformers, and LLMs are covered in Chapter 14. Those are enrichment topics for conceptual understanding.

10.2 Generative Versus Discriminative Models

Before diving into specific architectures, it's crucial to understand the fundamental distinction between two paradigms in machine learning: **discriminative** and **generative** models. This distinction matters because the two paradigms differ fundamentally in how they are trained, how they fail, and how they are evaluated.

10.2.1 Discriminative Models

Discriminative models focus on learning *boundaries* between classes. Given input data $\mathbf{x}$ and labels $\boldsymbol{y}$, a discriminative model learns the conditional probability $\boldsymbol{p}(\boldsymbol{y}|\mathbf{x})$. The probability of a label *given* the input.

Think of it this way: if you're training a discriminative model to distinguish

cats from dogs, it learns to draw a decision boundary between the two classes. When a new image arrives, the model determines which side of the boundary it falls on.

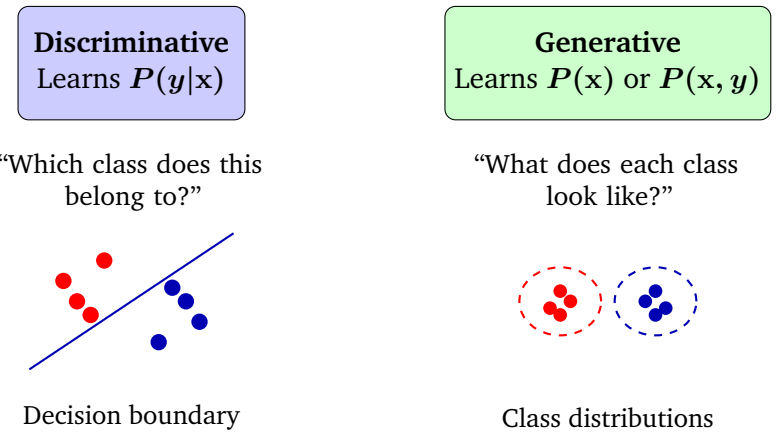


Figure 10.1: Discriminative models learn decision boundaries; generative models learn what each class looks like.

10.2.2 Generative Models

Generative models take a fundamentally different approach. Instead of learning boundaries, they learn *what the data looks like*, the underlying probability distribution $P(\mathbf{x})$. For classification tasks, they might learn $P(\mathbf{x}|y)$ (what cats look like, what dogs look like) and then use Bayes' theorem to classify.

The key advantage of generative models is their ability to create new data. Once a model understands what cats look like, it can generate new, plausible cat images. This is impossible for purely discriminative models, which only know how to draw boundaries.

Aspect	Discriminative	Generative
Learns	Boundaries between classes	Distribution of each class
Outputs	Class labels or probabilities	New data samples
Question answered	"Is this a cat or dog?"	"What does a cat look like?"
Examples	Logistic regression, SVM, CNN classifiers	VAEs, GANs, Diffusion models

Table 10.1: Comparison of discriminative and generative approaches.

10.3 Latent Space: The Heart of Generative Models

A central concept in generative modeling is the **latent space**, a lower-dimensional representation that captures the essential features of high-dimensional data. Understanding latent space is key to understanding how generative models work.

10.3.1 The Problem of High Dimensionality

Consider a modest image of $\mathbf{256} \times \mathbf{256}$ pixels with three color channels (RGB). This single image contains $\mathbf{256} \times \mathbf{256} \times \mathbf{3} = \mathbf{196,608}$ values. A training dataset might contain millions of such images. Processing data at this scale directly is computationally prohibitive and inefficient.

More fundamentally, not all combinations of pixel values make sense. The space of all possible $\mathbf{256} \times \mathbf{256}$ RGB images is astronomically large, but only a tiny fraction of that space contains meaningful images (faces, landscapes, objects). The rest is just random noise.

10.3.2 Compression to Meaningful Representations

The latent space provides a solution by encoding high-dimensional data into a compact, lower-dimensional representation that captures *only what matters*.

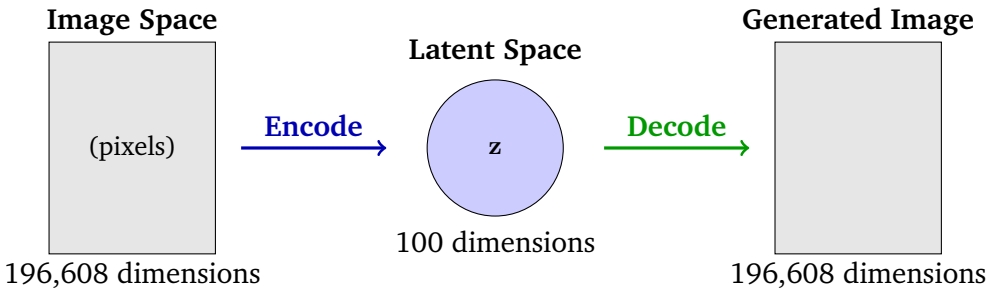


Figure 10.2: Latent space compresses high-dimensional data into a compact, meaningful representation.

10.3.3 Why Latent Space Works

The latent space works because of a key insight: similar things should be close together. If two images both show tabby cats, their latent representations should be nearby. If one shows a cat and another shows a car, they should be far apart.

This organization enables powerful capabilities:

- **Interpolation**: Moving smoothly between two points in latent space produces smooth transitions between the corresponding images. Moving from "cat" to "dog" in latent space might show intermediate creatures.
- **Arithmetic**: In a well-trained latent space, semantic operations become possible. The famous example: king $-$ man $+$ woman $\approx$ queen.
- **Generation**: Sampling random points from the latent space and decoding them produces new, realistic outputs.

Analogy: How We Remember

Think about how you remember a dog. You don't store every pixel of every dog you've ever seen. Instead, you maintain an abstract representation capturing the essential features: four legs, fur, tail, wet nose. This is exactly what latent space does, it captures the *essence* while discarding irrelevant details.

10.4 Autoencoders: Learning to Compress and Reconstruct

Autoencoders are neural network architectures that learn efficient representations of data by training to reconstruct their inputs. They provide the foundation for many generative models.

10.4.1 Architecture

An autoencoder consists of two main components:

1. **Encoder**: Compresses the input $\mathbf{x}$ into a lower-dimensional latent representation $\mathbf{z}$
2. **Decoder**: Reconstructs the input from the latent representation, producing output $\hat{\mathbf{x}}$

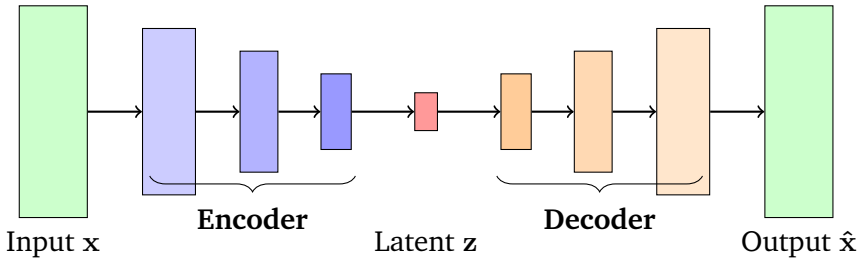


Figure 10.3: Autoencoder architecture: data flows through a bottleneck that forces compression.

10.4.2 Training Objective

The autoencoder is trained to minimize the **reconstruction error**, the difference between the input $\mathbf{x}$ and the reconstructed output $\hat{\mathbf{x}}$:

$$\mathcal{L} = \|\mathbf{x} - \hat{\mathbf{x}}\|^2 = \sum_i (x_i - \hat{x}_i)^2 \tag{10.1}$$

By forcing information through a *bottleneck* (the small latent space), the autoencoder must learn to capture only the most essential features. Redundant or noisy information cannot fit through the bottleneck and is discarded.

10.4.3 Applications of Autoencoders

Beyond learning representations, autoencoders power several practical applications. In **denoising**, we train the autoencoder on noisy inputs but use clean images as the reconstruction targets. The network learns to remove noise while preserving the true signal, it cannot simply memorize the noise, so it must learn the underlying structure.

Anomaly detection exploits the fact that autoencoders reconstruct normal data well but struggle with unusual inputs. If you train an autoencoder on images of normal manufacturing parts, a defective part will produce high reconstruction error because it does not match the patterns the network learned. This error flags the anomaly without ever needing labeled examples of defects.

The latent space itself provides **image compression**: instead of storing the full high-dimensional image, you store only the compact latent vector and decode when needed. Finally, encoder-decoder architectures extend to **semantic segmentation**, where the decoder outputs not a reconstruction but a classification for each pixel in the image.

10.5 Variational Autoencoders (VAEs)

Standard autoencoders have a limitation: their latent space isn't organized in a way that supports meaningful generation. Points sampled randomly from the latent space often decode to garbage. **Variational Autoencoders (VAEs)** fix this problem by encoding inputs to *probability distributions* rather than fixed points.

10.5.1 From Points to Distributions

In a standard autoencoder, the encoder produces a single latent vector $\mathbf{z}$ for each input. In a VAE, the encoder produces two outputs: $\boldsymbol{\mu}$, representing the mean of a Gaussian distribution, and $\boldsymbol{\sigma}$, representing its standard deviation. Together, these define a probability distribution in the latent space rather than a single point.

During training, instead of using a fixed $\mathbf{z}$, we *sample* from this distribution:

$$\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}, \quad \text{where } \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{10.2}$$

This is called the **reparameterization trick**. The $\boldsymbol{\epsilon}$ term introduces randomness, but importantly, the randomness doesn't depend on the learnable parameters, so gradients can still flow during training.

10.5.2 The VAE Training Objective

VAEs optimize two objectives simultaneously. The **reconstruction loss** ensures that the decoded output matches the input, just as in regular autoencoders. The **KL divergence** term ensures that the learned distributions stay close to a standard normal distribution $\mathcal{N}(\mathbf{0}, \mathbf{I})$.

$$\mathcal{L}_{\text{VAE}} = \underbrace{\|\mathbf{x} - \hat{\mathbf{x}}\|^2}_{\text{reconstruction}} + \underbrace{D_{\text{KL}}(q(\mathbf{z}|\mathbf{x}) \| \mathcal{N}(\mathbf{0}, \mathbf{I}))}_{\text{regularization}} \tag{10.3}$$

The KL divergence term is crucial: it ensures that the latent space is well-organized and continuous. Without it, the encoder could map different inputs to isolated, scattered regions of latent space with gaps in between. Generation would be unreliable because randomly sampled points might fall in these gaps, decoding to meaningless outputs. By forcing all encoded distributions to stay close to the standard normal, the KL term ensures that the entire latent space is "populated" and meaningful, any point you sample will decode to something sensible.

10.5.3 Generation with VAEs

Once trained, generating new samples is straightforward. You sample a latent vector $\mathbf{z}$ from the standard normal distribution $\mathcal{N}(\mathbf{0}, \mathbf{I})$, pass it through the decoder, and the output is a new, never-before-seen sample. Because the latent space has been regularized to match the standard normal distribution, random samples from that distribution produce meaningful outputs.

10.6 Evaluating Generative Models

In supervised learning, evaluation is straightforward: compute accuracy, precision, or AUC on a test set. Generative models have no such luxury. The question "how good are these generated images?" has no single answer, because quality has multiple competing dimensions.

Likelihood vs. perceptual quality: A model can assign high probability to every training image (high likelihood) while generating only blurry averages. VAEs do exactly this: by minimizing reconstruction loss over all pixels, they produce outputs that are statistically reasonable but visually soft. Conversely, a model can generate sharp, realistic-looking images while ignoring large parts of the data distribution. Likelihood and visual quality are different things, and optimizing one does not guarantee the other.

Diversity vs. fidelity: A model that produces one perfect face repeatedly has high fidelity but zero diversity. A model that produces every possible face, including distorted ones, has high diversity but low fidelity. The ideal model scores well on both, but improving one often hurts the other.

These tensions make evaluation fundamentally difficult. Metrics like the Fréchet Inception Distance (FID) attempt to capture both quality and diversity, but no single number fully summarizes whether a generative model is "good." Human evaluation remains important but is expensive and subjective.

10.6.1 Failure Modes

Blurry outputs. VAEs produce blurry images because their reconstruction loss averages over all possible outputs for a given latent code. When the model is uncertain whether a pixel should be light or dark, it compromises with a gray value. The result looks washed out. This is a direct consequence of the loss function, not a fixable implementation bug.

Posterior collapse. The decoder can learn to ignore the latent code entirely, producing outputs that are independent of the encoding. The KL term pushes the encoder toward the standard normal, and if the decoder is powerful enough, it finds it easier to ignore $\mathbf{z}$ than to use it. The result is a model

with a well-organized latent space that the decoder does not use, making meaningful generation impossible.

For failure modes of GANs and diffusion models, see Chapter 14.

10.6.2 Compute and Data Dependence

The architectures in this chapter work only because of extreme compute and data. A VAE trained on 100 images of faces will memorize them; a VAE trained on 10 million learns what faces look like in general. The same pattern holds for GANs and diffusion models. State-of-the-art generative models train on billions of images using thousands of GPUs for weeks or months. Many headline results in generative AI do not transfer to low-compute settings: a technique that produces photorealistic images with 1,000 GPUs may produce nothing useful with one.

This dependence connects to concepts from earlier chapters. A model trained on too little data overfits (Chapter 5): it memorizes training examples rather than learning the underlying distribution, and generated samples will be copies or near-copies of training data. The sample complexity ideas from Chapter 11 apply here too: the richer the data distribution you want to model, the more examples you need. Generative models push this requirement to its extreme because they must learn the full distribution, not just a decision boundary.

10.7 Summary

Key Takeaways

Generative vs. Discriminative: Discriminative models learn $P(y|x)$ (decision boundaries); generative models learn $P(x)$ (data distribution) and can create new samples.
Latent Space: A compressed representation capturing essential features. Generative models map between data space and latent space.
Autoencoders: Encoder compresses to latent code; decoder reconstructs. Good for representation learning but cannot reliably generate new samples.
VAEs: Encode to distributions (not points). KL divergence ensures smooth latent space. Stable training and density estimates, but blurry outputs.
Evaluation: No single metric captures generative quality. Likelihood, perceptual quality, and diversity can conflict. This makes generative model evaluation fundamentally harder than classification.
Scale matters: These methods require large datasets and substantial compute. Small data leads to memorization, not generation.

10.8 Practice Problems

1. Explain the difference between generative and discriminative models. Give an example of each for digit recognition.

2. An autoencoder has encoder $f : \mathbb{R}^{784} \to \mathbb{R}^{32}$ and decoder $g : \mathbb{R}^{32} \to \mathbb{R}^{784}$.

   (a) What is the compression ratio?

   (b) Why can't we generate new images by sampling random 32-dimensional vectors?

3. The VAE loss is: $\mathcal{L} = \mathbb{E}[\|x - \hat{x}\|^2] + \beta \cdot \text{KL}(q(z|x)\|p(z))$

   (a) What does each term encourage?

   (b) What happens if $\beta = 0$? If $\beta \to \infty$?

   (c) Why is the KL term necessary for generation?

4. A generative model trained on 10,000 face images produces outputs that look identical to specific training examples. A second model trained on 10 million images produces novel faces.

(a) Explain this difference in terms of overfitting (Chapter 5).

(b) How would you test whether a generative model is memorizing rather than generalizing?

5. **(Conceptual)** VAEs produce blurry images. Explain why, relating your answer to the reconstruction loss term in the VAE objective.

Part IV

Theory

Chapter 11

Statistical Learning Theory

When solving a problem of interest, do not solve a more general problem as an intermediate step.

Vladimir Vapnik

Chapter Difficulty Guide

This chapter contains material at multiple difficulty levels:
Core IAIO (everyone should master):
- PAC learning definition and intuition (11.1–11.3)
- Using the sample complexity bound (11.5)
- Finite hypothesis class learnability (11.4)

Intermediate (expected for competitive scores):
- Deriving bounds from first principles
- Agnostic PAC learning (11.6)
- Bias-complexity tradeoff

Advanced / Medal-level:
- Proof of the Fundamental Theorem
- Tight bounds and lower bounds
- Connections to compression and description length

Proofs marked with ⋆ are sketches only. Understanding the idea is sufficient for most students.

What You Should Be Able to Do

derive from scratch:
- sample complexity bound for finite hypothesis classes
- why $m \geq \frac{1}{\epsilon}(\ln |\langle| + \ln \frac{1}{\delta})$ follows from union bound

explain but not derive:
- why pac learning requires both ϵ (accuracy) and δ (confidence)
- why realizability assumption matters
- connection between overfitting and hypothesis class size

state and apply (formula given, use it):
- given $|\langle|$, ϵ, δ, compute required sample size
- given m, $|\langle|$, δ, compute achievable ϵ

11.1 Introduction: The Theory Behind Learning

We have seen many machine learning algorithms throughout this book, and they work remarkably well in practice. But a fundamental question remains: **why does learning from examples work at all?**

Here is a concrete failure that motivates the question. You train a spam filter on 50 emails and it classifies all 50 correctly. You deploy it, and it misclassifies half of new emails. You add more data, train on 500 emails, and the problem mostly disappears. Why did 50 examples fail? Why did 500 suffice? Was there a number you could have computed in advance that would have told you 50 was not enough? This chapter answers these questions precisely.

Consider what we ask machines to do. We show an algorithm a finite number of examples. Perhaps thousands of labeled images, and expect it to correctly classify images it has never seen. The training data is a tiny sample from an infinite universe of possibilities. How can patterns discovered in this small sample generalize to the vast unknown?

This is not merely philosophical. Understanding why learning works tells us when it will fail, how much data we need, and why some models generalize while others overfit. Statistical learning theory provides rigorous answers to these questions.

The Central Questions

Sample complexity: How many training examples do we need to learn reliably?
Learnability: What makes some problems learnable and others impossible?
Generalization: Why do some models generalize well while others merely memorize?

11.2 A Tale of Two Learners: Rats and Pigeons

Before diving into mathematics, consider two examples from psychology that illuminate the essence of learning.

11.2.1 The Clever Rats

When researchers tried to eliminate rats using poisoned bait, they discovered something remarkable. If a rat eats unfamiliar food and becomes sick hours later, it learns to avoid that food. *From a single experience*. But crucially, rats only associate sickness with the *taste and smell* of food. If researchers ring a bell when rats approach poison, the rats do not learn to avoid the bell. Evolution programmed them to connect illness with food properties, not arbitrary sounds.

11.2.2 The Superstitious Pigeons

B.F. Skinner placed hungry pigeons in cages with various toys and dispensed food at random intervals, completely independent of the pigeons' behavior.[1] Each pigeon latched onto whatever it happened to be doing when food arrived: one pecked a button, another spun in circles, a third bobbed its head. They developed "superstitions," treating coincidences as causal rules. The pigeons had no prior knowledge about what could cause food, so *every* behavior was an equally plausible explanation. With so many candidate causes, they were guaranteed to find spurious correlations.

[1] Skinner, B. F. (1948). "Superstition" in the pigeon. *Journal of Experimental Psychology*, 38(2), 168–172.

The Lesson for Machine Learning

The rats succeeded because evolution restricted their hypothesis class: when sick, consider only food-related causes. A small, well-chosen set of candidate explanations.
The pigeons failed because their hypothesis class was unrestricted: any behavior could be the cause. With too many candidate explanations and too little data, they fit noise instead of signal.
Key insight: *A smaller hypothesis class needs fewer examples to learn reliably. An unrestricted hypothesis class will overfit, finding patterns that are not real.* This is the overfitting problem from Chapter 5, and this chapter gives it a precise mathematical formulation.

11.3 The Formal Learning Framework

Let us formalize these intuitions. In supervised learning, we have a **domain** $\mathcal{X}$ (all possible inputs), a **label set** $\mathcal{Y}$ (possible outputs), and a **training sample** $S = \{(x_1, y_1), \ldots, (x_m, y_m)\}$.

The **hypothesis class** $\mathcal{H}$ represents our prior knowledge, the set of functions we consider as candidates. Just as rats consider only food-related hypotheses, our choice of $\mathcal{H}$ constrains what patterns the algorithm can discover.

A **learner** takes training data and outputs a hypothesis $h \in \mathcal{H}$. The goal is finding a hypothesis that generalizes to new examples.

11.3.1 The Papaya Example

Imagine learning which papayas are tasty by measuring color and softness. Here $\mathcal{X} = \mathbb{R}^2$ (color, softness pairs), $\mathcal{Y} = \{0, 1\}$ (not tasty, tasty). For $\mathcal{H}$, we might choose axis-aligned rectangles: predict "tasty" if features fall inside the rectangle.

Why rectangles? This encodes prior knowledge that tastiness depends on color and softness being in certain ranges. The restriction prevents learning spurious patterns that would not generalize.

11.4 Measuring Success: True Error vs. Training Error

True Error

The **true error** of hypothesis h with respect to distribution $\mathcal{D}$ is:

$$L_{\mathcal{D}}(h) = \Pr_{x \sim \mathcal{D}}[h(x) \neq f(x)]$$

This is the probability of misclassification on a **new, random** example. What we actually care about.

The problem: we cannot compute true error because we do not know $\mathcal{D}$. We can only compute **training error**:

$$l_s(h) = \frac{1}{m} \sum_{i=1}^{m} \mathbf{1}[h(x_i) \neq y_i]$$

The central question: *when does low training error imply low true error?*

11.5 PAC Learning: Probably Approximately Correct

The PAC framework, introduced by Leslie Valiant in 1984,[2] formalizes when learning is possible.

11.5.1 The Realizability Assumption

We first assume there exists $h^* \in \mathcal{H}$ with $L_{\mathcal{D}}(h^*) = 0$. The true labeling function is in our hypothesis class. This is like the rats knowing that sickness relates to food.

PAC Learnability

A hypothesis class $\mathcal{H}$ is **PAC learnable** if there exists a sample complexity function $m_{\mathcal{H}}(\epsilon, \delta)$ and a learning algorithm such that:
For any distribution $\mathcal{D}$ and any $\epsilon, \delta \in (0, 1)$, if we draw $m \geq m_{\mathcal{H}}(\epsilon, \delta)$ samples, then with probability at least $1 - \delta$:

$$L_{\mathcal{D}}(h) \leq \epsilon$$

In plain English: With enough samples, we can **probably** ($1 - \delta$) find an **approximately** (ϵ) correct hypothesis.

[2] Valiant, L. G. (1984). A theory of the learnable. *Communications of the ACM*, 27(11), 1134–1142.

11.5.2 Sample Complexity for Finite Classes

Theorem: Finite Classes are PAC Learnable

If $\mathcal{H}$ is finite and realizability holds, then:

$$m_{\mathcal{H}}(\epsilon, \delta) \leq \frac{\ln |\mathcal{H}| + \ln(1/\delta)}{\epsilon}$$

This tells us: more hypotheses ($|\mathcal{H}|$ larger) requires more samples; higher accuracy (ϵ smaller) requires more samples; higher confidence (δ smaller) requires more samples.

What Does This Bound Actually Tell Us?

The bound says m grows with $\ln |\mathcal{H}|$, not $|\mathcal{H}|$. This logarithmic dependence is the key practical insight: doubling the hypothesis class adds only $\ln 2 \approx 0.7$ to the sample requirement. Even a hypothesis class with 10^{100} members needs only about $230/\epsilon$ samples (plus the confidence term). Large hypothesis classes are more learnable than they first appear.

A worked example: Suppose $|\mathcal{H}| = 10{,}000$, $\epsilon = 0.05$, $\delta = 0.05$. Then $m \geq \frac{\ln 10{,}000 + \ln 20}{0.05} = \frac{9.21 + 3.00}{0.05} \approx 244$ samples. For 95% confidence that true error is below 5%, we need fewer than 250 training examples.

Limitations of PAC bounds in practice. PAC theory proves that learning is *possible* and identifies the right qualitative relationships: more hypotheses require more data, higher accuracy requires more data. But the bounds are typically pessimistic. The bound above might say 244 samples suffice, while in practice 50 might work because the data has structure that the worst-case analysis ignores. PAC bounds hold for *any* distribution, including adversarial ones. Real data distributions have regularity (smoothness, clusters, low intrinsic dimensionality) that algorithms exploit but the theory does not account for. Use PAC bounds for qualitative reasoning about what makes problems harder or easier, not as engineering specifications for exactly how much data to collect.

11.6 Agnostic PAC Learning

The realizability assumption is often unrealistic. What if no hypothesis in $\mathcal{H}$ is perfect?

Agnostic PAC Learning

A class $\mathcal{H}$ is **agnostic PAC learnable** if with enough samples, with high probability:

$$L_{\mathcal{D}}(h) \leq \min_{h' \in \mathcal{H}} L_{\mathcal{D}}(h') + \epsilon$$

We guarantee to be **competitive with the best in $\mathcal{H}$**, not perfect.

11.6.1 The Bias-Estimation Trade-off

This reveals two sources of error. **Bias** is error because the best hypothesis in $\mathcal{H}$ is not perfect, our prior knowledge is imperfect. **Estimation error** arises because we only see a finite sample, not the full distribution.

There is a fundamental trade-off: larger $\mathcal{H}$ reduces bias (more hypotheses to choose from) but increases estimation error (harder to identify the best among many). Smaller $\mathcal{H}$ has the opposite effect. This is the theoretical foundation of the overfitting/underfitting trade-off from Chapter 5, now stated precisely: the bias-variance tradeoff is not just an empirical observation but a mathematical necessity. No algorithm can avoid it.

11.7 Uniform Convergence

How do we prove learning works? Through **uniform convergence**: showing that training error approximates true error *simultaneously for all hypotheses* in $\mathcal{H}$.

ϵ-Representative Sample

A sample S is **ϵ-representative** if:

$$\forall h \in \mathcal{H} : |L_S(h) - L_{\mathcal{D}}(h)| \leq \epsilon$$

Training error approximates true error for **every** hypothesis simultaneously.

Why does this matter? If S is representative, then the **Empirical Risk Minimization** (ERM) algorithm (simply choosing the hypothesis with minimum training error) is guaranteed to work well. Finding the best on S gives us (approximately) the best overall.

11.8 Summary

Key Takeaways

Prior Knowledge is Essential: Without it (like the pigeons), we get superstition, not learning. The hypothesis class $\mathcal{H}$ encodes our prior knowledge.
PAC Learning: Formalizes "Probably Approximately Correct" learning. Sample complexity depends on $|\mathcal{H}|$, desired accuracy ϵ, and confidence δ. Finite classes are always PAC learnable.
Agnostic Learning: Drops the realizability assumption. We can only guarantee to be competitive with the best in $\mathcal{H}$.
Bias vs. Estimation: Large $\mathcal{H}$ has low bias but high estimation error (overfitting risk). Small $\mathcal{H}$ has high bias but low estimation error (underfitting risk).
Uniform Convergence: When training error approximates true error for all hypotheses, ERM succeeds.

11.9 Practice Problems

1. A hypothesis class has 1000 hypotheses. Using the PAC bound, how many samples are needed for $\epsilon = 0.1$ and $\delta = 0.05$?

2. Explain why the pigeons developed “superstitious” behavior while the rats learned correctly. What role did prior knowledge play?

3. In the papaya example, why choose rectangles as our hypothesis class? What would happen if we used all possible functions?

4. What is the difference between training error $L_S(h)$ and true error $L_{\mathcal{D}}(h)$? Why can’t we compute true error directly?

5. A sample is 0.05-representative. ERM finds $\hat{h}$ with $L_S(\hat{h}) = 0.02$. What can you say about $L_{\mathcal{D}}(\hat{h})$?

6. Explain the bias-estimation trade-off. How does this relate to overfitting and underfitting?

3

---

[3] See Appendix A for properties of logarithms and exponentials used in these bounds.

Chapter 12

Advanced Statistical Learning Theory

The price of generality is complexity.

John von Neumann

Chapter Difficulty Guide

This chapter is primarily **advanced / medal-level** material.
Core IAIO (everyone should master):

- VC dimension definition and examples (12.2–12.3)
- Computing VC dimension for simple classes (intervals, rectangles, linear classifiers)

Advanced / Medal-level:

- Sauer's Lemma and growth functions
- Fundamental Theorem of Statistical Learning (12.4)
- No Free Lunch Theorem and implications (12.5)
- Proofs of the main theorems

Study advice: If you are aiming for participation (not medals), focus only on computing VC dimension for standard hypothesis classes. The deeper theory is for students targeting gold/silver medals.
Proofs in this chapter are sketches, understanding the idea is more important than memorizing details.

What You Should Be Able to Do

Compute from scratch:
- VC dimension of intervals on $\mathbb{R}$ (answer: 2)
- VC dimension of axis-aligned rectangles in $\mathbb{R}^2$ (answer: 4)
- VC dimension of linear classifiers in $\mathbb{R}^d$ (answer: $d + 1$)

Explain but not derive:
- Why VC dimension measures "effective complexity"
- Why finite VC dimension implies learnability
- Intuition for Sauer's Lemma (growth is polynomial, not exponential)

Know the statement only:
- Fundamental Theorem: PAC learnable $\Leftrightarrow$ finite VC dimension
- No Free Lunch: no universal learner works for all distributions

12.1 Introduction: Beyond Finite Classes

In the previous chapter, we saw that finite hypothesis classes are PAC learnable, with sample complexity depending on $\ln |\mathcal{H}|$. But what about infinite classes like "all rectangles" or "all linear classifiers"? These have infinitely many hypotheses, yet we can learn them effectively. How is this possible?

The bound $m \leq \frac{\ln |\mathcal{H}|}{\epsilon}$ gives infinity for infinite classes, which is useless. We need a different way to measure the "complexity" of a hypothesis class. One that captures what really matters for learning, regardless of whether the class is finite or infinite.

The remarkable answer is the **VC dimension**, a combinatorial measure that completely characterizes learnability. A class is learnable if and only if its VC dimension is finite, and the sample complexity depends on the VC dimension rather than the class size.

12.2 Shattering: What a Class Can Express

The key insight is that what matters for learning is not how many hypotheses exist, but how many *different behaviors* the class can produce on finite sets of points.

Given a set of points $C = \{x_1, \ldots, x_m\}$, consider all the ways hypotheses in $\mathcal{H}$ can label these points. The **restriction** of $\mathcal{H}$ to C is:

$$\mathcal{H}_C = \{(h(x_1), \ldots, h(x_m)) : h \in \mathcal{H}\}$$

This is the set of all labeling patterns that some hypothesis in $\mathcal{H}$ produces on

C.

For example, consider threshold classifiers on the real line: $h_\theta(x) = \mathbf{1}[x \geq \theta]$. Given points $c = \{1, 3, 5\}$, different thresholds produce different labelings. Threshold $\theta = 0$ gives $(1, 1, 1)$ (all positive). Threshold $\theta = 2$ gives $(0, 1, 1)$. Threshold $\theta = 4$ gives $(0, 0, 1)$. Threshold $\theta = 6$ gives $(0, 0, 0)$. But no threshold can produce $(1, 0, 1)$. You cannot classify 1 as positive, 3 as negative, and 5 as positive using a single threshold.

Shattering

A hypothesis class $\mathcal{H}$ **shatters** a set C if $\mathcal{H}_C$ contains **all** $2^{|C|}$ possible labelings.
In other words: for every possible way to label the points in C, there exists some hypothesis in $\mathcal{H}$ that produces exactly that labeling. The class is "expressive enough" to realize any pattern on those points.

Put differently, shattering means a class can memorize arbitrary labels on those points, including completely random labels with no real pattern. This is the precise mechanism of overfitting: if $\mathcal{H}$ can shatter your training set, it can fit any noise in the labels perfectly, and that perfect fit tells you nothing about generalization.

Threshold classifiers can shatter any single point (two possible labelings, both achievable). They can shatter any two points $\{a, b\}$ with $a < b$: labelings $(0, 0)$, $(0, 1)$, and $(1, 1)$ are achievable... wait, what about $(1, 0)$? This would require $a \geq \theta$ but $b < \theta$, which is impossible since $a < b$. So threshold classifiers *cannot* shatter two points.

Actually, let us reconsider. With two points $a < b$, we can achieve: $(0, 0)$ with $\theta > b$; $(0, 1)$ with $a < \theta \leq b$; $(1, 1)$ with $\theta \leq a$. We cannot achieve $(1, 0)$. So thresholds cannot shatter 2 points in general... but they can shatter 1 point. The VC dimension turns out to be 1.

12.3 The VC Dimension

VC Dimension

The **Vapnik-Chervonenkis (VC) dimension** of a hypothesis class $\mathcal{H}$, denoted $\text{VCdim}(\mathcal{H})$, is:

$$\text{VCdim}(\mathcal{H}) = \max\{m : \exists \text{ a set } C \text{ of size } m \text{ that } \mathcal{H} \text{ shatters}\}$$

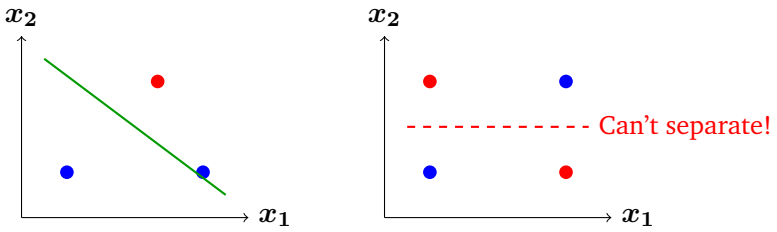
It is the largest set size that $\mathcal{H}$ can shatter, the maximum number of points on which $\mathcal{H}$ can realize all possible labeling patterns.

The VC dimension measures the "expressiveness" or "complexity" of a hypothesis class in a way that is independent of whether the class is finite or infinite. A class with high VC dimension can express many different patterns; a class with low VC dimension is more constrained.

12.3.1 Examples of VC Dimensions

Hypothesis Class	VCdim	Intuition
Thresholds on $\mathbb{R}$	1	Can separate 1 point, not 2
Intervals $[a, b]$ on $\mathbb{R}$	2	Can capture any 2 points
Axis-aligned rectangles in $\mathbb{R}^2$	4	Four corners to adjust
Linear classifiers in $\mathbb{R}^d$	$d + 1$	$d + 1$ parameters (weights + bias)

The most important example is linear classifiers. In $\mathbb{R}^d$, a linear classifier is defined by a hyperplane: points on one side are positive, points on the other side are negative. The VC dimension of linear classifiers in $\mathbb{R}^d$ is exactly $d + 1$.



Can shatter 3 points Cannot shatter 4 (XOR)

Figure 12.1: Linear classifiers in $\mathbb{R}^2$ have VC dimension 3: they can shatter any 3 points in general position, but no line can separate the XOR pattern on 4 points.

To show VCdim $= 3$ for lines in $\mathbb{R}^2$, we must show: (1) there exist 3 points that can be shattered, and (2) no 4 points can be shattered. For (1), place three points not on a line; any labeling can be achieved by some line. For (2), consider 4 points. For certain labelings like the xor pattern (opposite corners same label), no line can separate them.

12.4 The Fundamental Theorem of PAC Learning

The VC dimension is not just one measure of complexity among many. It completely characterizes learnability:

Fundamental Theorem

A hypothesis class $\mathcal{H}$ is (agnostic) PAC learnable **if and only if** it has finite VC dimension.
Moreover, the sample complexity is:

$$m_{\mathcal{H}}(\epsilon, \delta) = O\left(\frac{\text{VCdim}(\mathcal{H}) + \ln(1/\delta)}{\epsilon^2}\right)$$

This is remarkable. The VC dimension. A purely combinatorial property about which labelings can be realized, completely determines whether learning is possible and how many samples are needed. Infinite classes with finite VC dimension (like all linear classifiers) are learnable. Classes with infinite VC dimension are not PAC learnable.

The theorem explains why sample complexity depends on VC dimension rather than the number of hypotheses. The class of all linear classifiers in $\mathbb{R}^{100}$ is infinite, but its VC dimension is only 101. Learning requires on the order of 101 samples (times factors for accuracy and confidence), not infinitely many.

12.5 The No Free Lunch Theorem

A natural question: is there a "universal" learning algorithm that works for all problems? The answer is definitively no.

No Free Lunch Theorem

There is no learning algorithm that is optimal for **all** problems.
More precisely: for any learner A, there exists a distribution $\mathcal{D}$ and labeling function f such that A performs poorly, yet some other learner A' performs well.

This has profound implications. **Prior knowledge is essential**: you must choose a hypothesis class appropriate for your problem. **No universal solution**: different problems genuinely require different approaches. **This is why AI is hard**: general intelligence would require appropriate priors for every possible domain.

The No Free Lunch theorem does not mean machine learning is useless. It means that when learning succeeds, it is because the hypothesis class matches the problem structure. The theorem explains why expertise matters: choosing the right representation and the right class for a problem is itself a form of knowledge.

12.6 Connecting Theory to Practice

12.6.1 Overfitting Through the Lens of VC Dimension

The VC dimension provides a theoretical explanation for overfitting and underfitting:

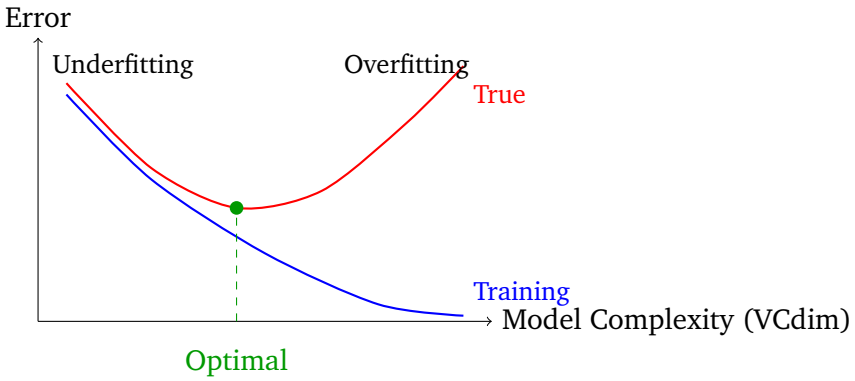


Figure 12.2: The bias-variance trade-off in terms of VC dimension. Low complexity means high bias (underfitting); high complexity means high variance (overfitting).

With small VC dimension (simple model), the class cannot express complex patterns. High bias, underfitting. With large VC dimension (complex model), the class can fit training data very well but may not generalize. High variance, overfitting. The optimal choice balances these, depending on the amount of training data available.

12.6.2 Practical Guidelines

Several practical insights follow from the theory:

Match complexity to data size: More data allows for more complex models (higher VC dimension). With limited data, use simpler models.

Regularization reduces effective VC dimension: Techniques like weight decay constrain the hypothesis class, reducing complexity without changing the model architecture.

Cross-validation estimates true error: Since we cannot compute true error directly, validation on held-out data provides our best estimate.

The Deep Learning Puzzle

VC theory predicts that models with high VC dimension should overfit unless given proportionally large training sets. Deep neural networks violate this prediction: they have VC dimensions that often exceed the number of training examples (they can memorize random labels), yet they generalize well on real data.

This is one of the most important open questions in machine learning theory. Several partial explanations exist: optimization via gradient descent implicitly favors simple solutions (implicit regularization), real data has structure that random labels lack, and overparameterized networks find flat minima that generalize better. But no complete theory yet explains why deep learning works as well as it does. VC theory remains correct as a worst-case bound but is too pessimistic to describe the typical behavior of modern neural networks.

12.7 Summary

Key Takeaways

VC Dimension: Measures hypothesis class complexity by the largest set it can shatter. A class shatters a set if it can produce all possible labelings on that set.

Fundamental Theorem: A class is PAC learnable if and only if its VC dimension is finite. Sample complexity scales with VC dimension.

No Free Lunch: No universal learning algorithm exists. Success requires matching the hypothesis class to the problem structure.

Practical Implications: VC dimension explains overfitting/underfitting. Balance model complexity with data size. Regularization controls effective complexity. However, VC bounds are too pessimistic to explain why deep neural networks generalize despite enormous VC dimension; this remains an open problem.

12.8 Practice Problems

1. Show that intervals $[a, b]$ on the real line have VC dimension 2. (Hint: Show you can shatter 2 points but not 3.)
2. The class of "decision stumps" in $\mathbb{R}^2$ (classifiers of form $\mathbf{1}[x_i \geq \theta]$ for $i \in \{1, 2\}$) has what VC dimension?
3. If a hypothesis class has VCdim = 10 and we want $\epsilon = 0.05$, $\delta = 0.01$,

approximately how many samples do we need?

4. Explain why the No Free Lunch theorem does not mean machine learning is useless. What saves us in practice?

5. A model achieves 99% training accuracy but only 70% test accuracy. Using the concepts from this chapter, explain what is happening and how to fix it.

6. Why can't linear classifiers in $\mathbb{R}^2$ shatter 4 points in general? Give a specific configuration that cannot be labeled.

7. **(Multi-part, IAIO-style)** Consider the hypothesis class of axis-aligned rectangles in $\mathbb{R}^2$ (predict positive inside the rectangle, negative outside).

   (a) Can this class shatter 3 points? Give an example arrangement that can be shattered.

   (b) Can this class shatter 4 points? If yes, give an arrangement. If no, explain which labeling is impossible.

   (c) Can this class shatter 5 points? Prove your answer.

   (d) What is the VC dimension of this class?

   (e) Using the Fundamental Theorem, with 500 training examples and $\delta = 0.05$, bound the generalization error.

8. **(Adversarial thinking)** The No Free Lunch theorem says no learner is universally best. Yet in practice, some algorithms (like neural networks) seem to work well on almost everything.

   (a) Explain why this doesn't contradict NFL.

   (b) What does NFL really tell us about choosing learning algorithms?

Part V

Applications (Optional)

Chapter 13

Matrix Factorization: A Unifying Idea

The best low-rank approximation of a matrix is given by its singular value decomposition.

Carl Eckart & Gale Young

Part V: Applications (Optional)

This chapter presents matrix factorization as a mathematical technique with applications in recommender systems and beyond. The focus is on the **mathematical foundations**, connecting to linear algebra, optimization, and regularization, rather than system implementation.
IAIO relevance: Matrix factorization problems test your understanding of:

- Low-rank approximation and SVD (Appendix F)
- Loss functions and regularization (Chapter 4)
- Gradient-based optimization (Chapter 4, Appendix D)
- The bias-variance tradeoff (Chapter 5)

Why This Chapter Exists

Matrix factorization appears infrequently on IAIO, so why include it? Because it is a **unifying example** that connects multiple core concepts:

- **Optimization**: The same gradient descent from linear regression (Ch 4)
- **Regularization**: The same L2 penalty from ridge regression (Ch 4)
- **Evaluation**: The same train/test methodology (Ch 5)
- **Linear algebra**: Direct application of SVD (Appendix F)

If you understand matrix factorization, you've demonstrated mastery of these foundations in a novel context, exactly what IAIO tests. This chapter is short because the ideas aren't new; they're *applied*.

13.1 The Core Idea

Many real-world datasets can be represented as matrices with missing entries. Users rating movies, students answering test questions, sensors measuring phenomena, all produce sparse matrices where we observe only a fraction of possible values.

The key insight of matrix factorization: **real-world matrices are approximately low-rank**. This means they can be well-approximated by the product of much smaller matrices.

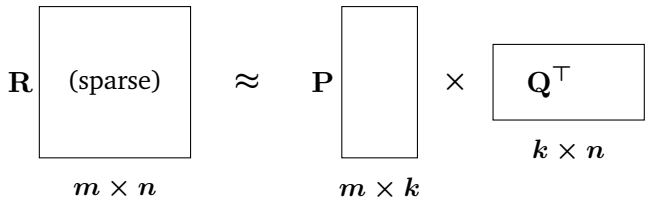


Figure 13.1: Matrix factorization: approximate a large sparse matrix as the product of two small dense matrices.

If $\mathbf{R}$ is $m \times n$ and we factor it as $\mathbf{R} \approx \mathbf{P}\mathbf{Q}^\top$ with $k \ll \min(m, n)$, we've compressed the information dramatically:

- Original: $m \times n$ entries (mostly unknown)
- Factored: $m \times k + n \times k = k(m + n)$ parameters

Example: For Netflix (200M users, 15K movies), storing all possible ratings requires 3×10^{12} entries. With $k = 100$ factors: $(200M + 15K) \times 100 \approx 2 \times 10^{10}$ parameters, a 150× reduction.

13.2 Mathematical Formulation

13.2.1 The Optimization Problem

We want to find matrices $\mathbf{P}$ and $\mathbf{Q}$ such that their product approximates the observed entries of $\mathbf{R}$.

Let $\boldsymbol{\Omega}$ be the set of observed entries (pairs (i, j) where we know R_{ij}). The objective:

$$\min_{\mathbf{P},\mathbf{Q}} \sum_{(i,j)\in\Omega} \left(R_{ij} - \mathbf{p}_i^\top \mathbf{q}_j\right)^2 + \lambda \left(\|\mathbf{P}\|_F^2 + \|\mathbf{Q}\|_F^2\right)$$

where:

- $\mathbf{p}_i$ is the i-th row of $\mathbf{P}$ (the "embedding" of user/row i)
- $\mathbf{q}_j$ is the j-th row of $\mathbf{Q}$ (the "embedding" of item/column j)
- $\|\cdot\|_F$ is the Frobenius norm (sum of squared entries)
- λ is the regularization strength

Connecting to Concepts You Know

Loss function: Squared error, just like linear regression (Chapter 4).
Regularization: L2 penalty on parameters, just like ridge regression.
Optimization: Gradient descent on a non-convex objective (unlike linear regression, which is convex).

Explicit Connections to Earlier Material

Matrix factorization is **not a new technique**, it combines ideas you already know:

From Chapter 4 (Supervised Learning):

- The loss function is squared error, identical to linear regression
- The regularization term is L2 (ridge), identical to ridge regression
- The gradient update rules follow the same derivation pattern

From Chapter 5 (Model Evaluation):

- Choosing k and λ requires cross-validation on held-out ratings
- Overfitting manifests as low training RMSE but high test RMSE

From Appendix D (Calculus):

- Gradient derivation uses the chain rule
- The update $\mathbf{p}_i \leftarrow \mathbf{p}_i + \eta(e_{ij}\mathbf{q}_j - \lambda\mathbf{p}_i)$ is standard gradient descent

From Appendix F (Linear Algebra):

- SVD provides the optimal low-rank approximation for complete matrices
- Matrix factorization extends this to incomplete matrices

13.2.2 Gradient Derivation

To apply gradient descent, we need the gradients with respect to $\mathbf{p}_i$ and $\mathbf{q}_j$.

For a single observation (i, j) with error $e_{ij} = R_{ij} - \mathbf{p}_i^\top \mathbf{q}_j$:

$$\frac{\partial}{\partial \mathbf{p}_i}\left[e_{ij}^2 + \lambda\|\mathbf{p}_i\|^2\right] = -2e_{ij}\mathbf{q}_j + 2\lambda\mathbf{p}_i$$

$$\frac{\partial}{\partial \mathbf{q}_j}\left[e_{ij}^2 + \lambda\|\mathbf{q}_j\|^2\right] = -2e_{ij}\mathbf{p}_i + 2\lambda\mathbf{q}_j$$

Update rules (with learning rate η):

$$\begin{aligned}\mathbf{p}_i &\leftarrow \mathbf{p}_i + \eta\left(e_{ij}\mathbf{q}_j - \lambda\mathbf{p}_i\right)\\ \mathbf{q}_j &\leftarrow \mathbf{q}_j + \eta\left(e_{ij}\mathbf{p}_i - \lambda\mathbf{q}_j\right)\end{aligned}$$

13.2.3 Alternating Least Squares

An alternative to gradient descent: **Alternating Least Squares (ALS)**.

If we fix $\mathbf{Q}$, optimizing $\mathbf{P}$ is a convex problem (it's just ridge regression for each row). Similarly, if we fix $\mathbf{P}$, optimizing $\mathbf{Q}$ is convex.

Algorithm:

1. Initialize $\mathbf{P}$ and $\mathbf{Q}$ randomly
2. Fix $\mathbf{Q}$, solve for optimal $\mathbf{P}$ (closed-form solution)
3. Fix $\mathbf{P}$, solve for optimal $\mathbf{Q}$ (closed-form solution)
4. Repeat until convergence

ALS converges to a local minimum and is easily parallelizable (each row of $\mathbf{P}$ can be updated independently).

13.3 Connection to SVD

The theoretical foundation for matrix factorization comes from the Singular Value Decomposition (SVD).[1] The Eckart–Young theorem (1936)[2] guarantees that the best rank-$\boldsymbol{k}$ approximation of any matrix, in the Frobenius norm, is obtained by truncating its SVD to the top $\boldsymbol{k}$ singular values. This is what motivates the $\mathbf{PQ}^\top$ approach: we are searching for the same kind of low-rank structure that SVD would find, but under the constraint that most entries are unobserved.

SVD decomposes any matrix as:

$$\mathbf{R} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\top$$

where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal and $\boldsymbol{\Sigma}$ is diagonal with non-negative singular values.

Low-rank approximation: Keeping only the top $\boldsymbol{k}$ singular values gives the best rank-$\boldsymbol{k}$ approximation (in Frobenius norm). This is precisely the Eckart–Young result.

The catch: Standard SVD requires the full matrix. With missing entries, we can't apply SVD directly. Matrix factorization can be viewed as "SVD for incomplete matrices", finding a low-rank approximation while only fitting observed entries.

[1] See Appendix F for a complete treatment of SVD.

[2] Eckart, C. & Young, G. (1936). The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3), 211–218.

Why Low-Rank Works

Real-world matrices are approximately low-rank because the underlying phenomena have limited degrees of freedom:

- Movie preferences depend on a few "factors" (genre, era, mood)
- Student performance depends on a few "skills"
- Scientific measurements depend on a few physical processes

The k columns of $\mathbf{P}$ and $\mathbf{Q}$ capture these latent factors.

13.4 The Bias-Variance Perspective

High k (more factors):

- More expressive model
- Can fit training data better
- Risk of overfitting (high variance)

Low k (fewer factors):

- Simpler model
- May underfit (high bias)
- Better generalization if true rank is low

Regularization λ:

- High λ: Shrinks embeddings toward zero, reduces effective model complexity
- Low λ: Allows larger embeddings, more flexibility

Choosing k and λ requires validation on held-out data. Exactly the methodology from chapter 5.

Why This Is Not Magic

Matrix factorization can seem almost magical: we observe a tiny fraction of a matrix and somehow predict all the missing entries. But there is no magic. Only strong assumptions:

The low-rank assumption: We assume the "true" complete matrix has low rank. If user preferences actually depend on thousands of independent factors, low-rank factorization will fail.

Missing at random: We assume observed entries are representative. If users only rate movies they love (or hate), our sample is biased and predictions for "meh" movies will be poor.

Stationarity: We assume preferences don't change. A model trained on 2020 ratings may fail for 2024 users with different tastes.

When these assumptions hold approximately, matrix factorization works well. When they fail, no amount of algorithmic cleverness will save you. **Understanding assumptions is more important than understanding algorithms.**

Two practical failure modes deserve special attention. The **cold-start problem** arises when a new user (or item) has no observed entries at all. With zero data points, there is nothing to learn an embedding from; the model cannot distinguish this user from any other. Common workarounds include using side information (demographics, item descriptions) or falling back to popularity-based recommendations until enough data accumulates. **Extreme sparsity** is the related problem of having too few observations per user or item. If a user has rated only two movies out of 15,000, the system is trying to infer a k-dimensional embedding from two data points, which is severely underdetermined. In practice, matrix factorization performance degrades sharply when the observation rate drops below roughly 1% of entries, though the exact threshold depends on the rank and noise level.

13.5 Summary

Key Takeaways

Matrix Factorization: Approximate $\mathbf{R} \approx \mathbf{P}\mathbf{Q}^\top$ to predict missing entries.
Objective: Minimize squared error on observed entries plus L2 regularization.
Optimization: Gradient descent or Alternating Least Squares. Non-convex, so local minima exist.
Connection to SVD: Like SVD, but handles missing data by only fitting observed entries.
Low-rank assumption: Works because real-world data has limited underlying dimensions (latent factors).
Hyperparameters: k (rank) controls capacity; λ (regularization) controls complexity. Both affect bias-variance tradeoff.

13.6 Practice Problems

1. A matrix has 1M rows and 100K columns. With $k = 50$ factors:
   (a) How many parameters in the factorization?
   (b) What compression ratio vs. the full matrix?
   (c) If we observe 0.1% of entries, how many observations constrain the parameters?
2. User Alice has embedding $\mathbf{p} = (0.8, -0.3, 0.5)$. Items have embeddings $\mathbf{q_1} = (0.5, 0.2, 0.4)$ and $\mathbf{q_2} = (0.1, -0.5, 0.8)$. Calculate predicted values. Which item ranks higher?
3. **(Gradient derivation)** Starting from the objective function, derive the gradient update rule for $\mathbf{p}_i$. Show your work.
4. **(IAIO-style)** The matrix factorization objective is:
$$L = \sum_{(i,j)\in\Omega} (R_{ij} - \mathbf{p}_i^\top \mathbf{q}_j)^2 + \lambda(\|\mathbf{P}\|_F^2 + \|\mathbf{Q}\|_F^2)$$
   (a) What is Ω? Why do we only sum over Ω?
   (b) Why is regularization necessary? What happens as $\lambda \to 0$? As $\lambda \to \infty$?
   (c) This objective is non-convex. Why? What are the implications for optimization?

(d) Explain why ALS alternates between $\mathbf{P}$ and $\mathbf{Q}$. Why is each subproblem convex?

5. **(Connection to linear algebra)** Explain the relationship between matrix factorization and truncated SVD. When are they equivalent? When do they differ?

6. **(Bias-variance)** You increase k from 10 to 100. Training RMSE drops from 0.8 to 0.3. Test RMSE increases from 0.9 to 1.2. Diagnose the problem and propose two solutions.

Chapter 14

Modern Deep Learning

> You don't need to be biologically plausible to be intelligent.
>
> Yann LeCun

Note: Enrichment Material

This chapter covers cutting-edge AI architectures that power today's most impressive systems. While valuable for understanding the current AI landscape, this material is **not directly examined** on the IAIO. Read for conceptual understanding, not memorization. Focus your study time on core chapters (3–6, 8, 11).

14.1 Introduction

The period from 2014 to the present has seen revolutionary advances in AI, driven by four architectural innovations: Generative Adversarial Networks (GANs), the Transformer architecture, diffusion models for image generation, and the scaling of large language models. These technologies power systems like ChatGPT, DALL-E, Stable Diffusion, and many others.

This chapter provides conceptual understanding of these systems, focusing on key ideas rather than implementation details. Chapter 10 covers the foundational concepts (generative vs. discriminative models, latent space, autoencoders, VAEs) that this chapter builds on.

14.2 How Computers Process Images

Before exploring modern vision architectures, it helps to understand how computers represent visual information.

14.2.1 Images as Numbers

To a computer, an image is a grid of numbers called **pixels**. For grayscale images, each pixel is a single number (0 = black, 255 = white). For color images, each pixel has three numbers representing red, green, and blue intensity (RGB).

Scale of Image Data

A standard HD image ($\mathbf{1920 \times 1080}$) contains over 2 million pixels. With three color channels, that's over 6 million numbers per image. Deep learning models must find meaningful patterns in this high-dimensional space.

14.2.2 From Pixels to Understanding

The challenge of computer vision is transforming raw pixels into semantic understanding. Neural networks solve this through **hierarchical feature learning**:

- **Early layers**: Detect simple features (edges, textures)
- **Middle layers**: Combine into complex patterns (shapes, parts)
- **Late layers**: Integrate into high-level concepts (objects, scenes)

This hierarchy is fundamental to both CNNs and modern vision transformers.

14.3 Generative Adversarial Networks (GANs)

Generative Adversarial Networks, introduced by Ian Goodfellow in 2014,[1] revolutionized generative modeling through a clever training paradigm: make two neural networks compete against each other.

[1] Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2014). Generative adversarial nets. *Advances in Neural Information Processing Systems*, 27, 2672–2680.

14.3.1 The Two Players

A GAN consists of two neural networks. The **Generator (G)** takes random noise $\mathbf{z}$ as input and produces fake data samples; its goal is to create samples so realistic that they fool the discriminator. The **Discriminator (D)** takes either real data from the training set or fake data from the generator and tries to classify each sample as "real" or "fake."

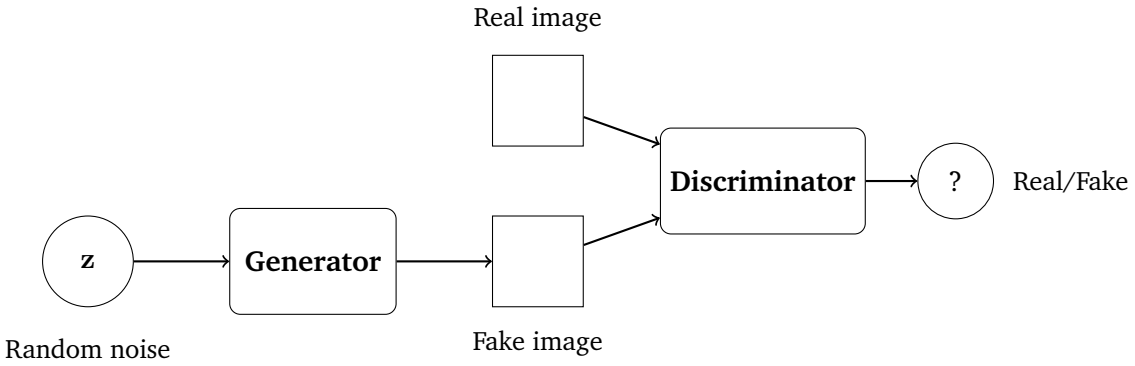


Figure 14.1: GAN architecture: Generator creates fake samples, Discriminator tries to detect them.

14.3.2 The Adversarial Game

Training is framed as a two-player game. The discriminator wants to maximize its accuracy, correctly identifying real samples as real and fake samples as fake. The generator wants to minimize the discriminator's accuracy, producing fakes so good that the discriminator cannot tell them apart from real data.

This creates an arms race: as the discriminator gets better at spotting fakes, the generator must produce increasingly realistic samples to keep fooling it. When training succeeds, the generator produces samples indistinguishable from real data.

Mathematically, GANs solve a minimax optimization problem:

$$\min_{G} \max_{D} \; \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{z}}[\log(1 - D(G(\mathbf{z})))] \tag{14.1}$$

The discriminator wants $D(\mathbf{x})$ close to 1 for real data and $D(G(\mathbf{z}))$ close to 0 for fake data. The generator wants the opposite: $D(G(\mathbf{z}))$ close to 1, meaning the discriminator is fooled.

14.3.3 Strengths and Challenges

GANs produce remarkably sharp, realistic images, often sharper than VAEs. However, they present several challenges. **Mode collapse** occurs when the generator learns to produce only a few types of outputs that happen to fool the discriminator, ignoring the full diversity of real data. **Training instability** arises because the adversarial setup is delicate: if one network becomes too strong too quickly, the other cannot learn effectively, and training fails. Unlike supervised learning where the loss steadily decreases, GAN training can oscillate or diverge. Finally, GANs provide **no explicit density**: unlike VAEs, they do not learn an explicit probability distribution over data, making it difficult to compute how likely a given image is.

VAE vs. GAN: The Core Tradeoff

VAE (Chapter 10): Stable training, provides likelihoods, smooth latent space. But outputs tend to be blurry because the loss function averages over uncertainty.
GAN: Sharp, realistic outputs. But training is unstable, mode collapse can silently reduce diversity, and there is no way to compute how likely a given image is under the model.
Neither is strictly better. The choice depends on whether you need stable training and density estimates (VAE) or maximum visual quality (GAN).

14.4 Diffusion Models

Diffusion models represent the current state-of-the-art in image generation, powering systems like DALL-E, Stable Diffusion, and Midjourney. They largely solve the training instability and mode collapse problems that plague GANs, though at the cost of slower generation.

14.4.1 The Forward Process: Adding Noise

The forward diffusion process starts with a clean data sample $\mathbf{x_0}$ and gradually adds Gaussian noise over many timesteps until the data becomes pure noise:

$$\mathbf{x}_t = \sqrt{1-\beta_t}\,\mathbf{x}_{t-1} + \sqrt{\beta_t}\,\boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \boldsymbol{\mathcal{N}}(\mathbf{0}, \mathbf{I}) \tag{14.2}$$

Here, $\boldsymbol{\beta_t}$ is a noise schedule that controls how much noise to add at each step. After enough steps (typically 1000), $\mathbf{x}_T$ is indistinguishable from pure random noise.

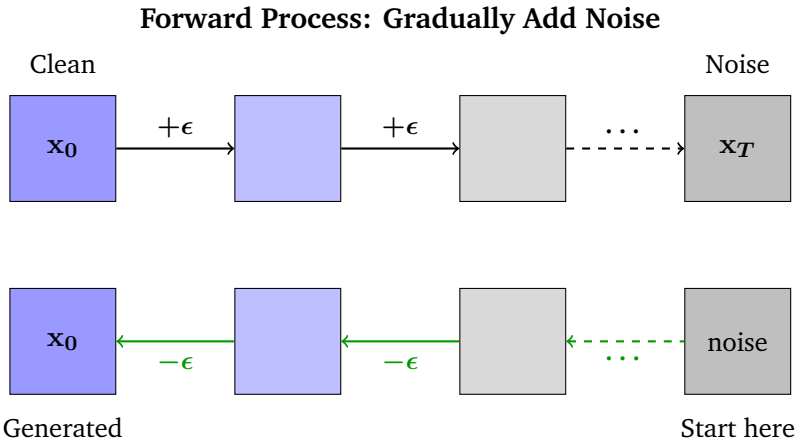


Figure 14.2: Diffusion models: the forward process destroys information; the reverse process learns to recover it.

14.4.2 The Reverse Process: Learning to Denoise

The magic of diffusion models is learning the **reverse process**: starting from pure noise and gradually removing noise to recover a clean sample. A neural network is trained to predict the noise that was added at each step:

$$\epsilon_\theta(\mathbf{x}_t, t) \approx \epsilon \tag{14.3}$$

The network takes a noisy image $\mathbf{x}_t$ and the timestep t, and predicts what noise was added. This allows us to "undo" the noise step by step.

14.4.3 Generation

To generate a new image:

1. Start with pure random noise $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2. For each timestep from T down to $\mathbf{0}$:
    - Predict the noise using the trained network
    - Remove the predicted noise to get $\mathbf{x}_{t-1}$
3. The final $\mathbf{x_0}$ is the generated image

14.4.4 Why Diffusion Models Work So Well

Diffusion models have several advantages over earlier approaches. Their **training is stable**: unlike GANs, there is no adversarial dynamic that can collapse or oscillate. The training objective (predict the noise that was added) is straightforward and well-behaved. The **output quality** is exceptional because the iterative refinement process (gradually removing noise over many steps) allows the model to correct mistakes and add fine details progressively. Finally, diffusion models achieve good **mode coverage**: they tend to capture the full diversity of training data rather than collapsing to a few common outputs.

The main drawback is speed: generating an image requires running the network hundreds or thousands of times (once per denoising step).

14.4.5 Why Diffusion Replaced GANs

The shift from GANs to diffusion models was driven by three practical advantages. **Training stability**: GAN training requires balancing two competing networks and can collapse or oscillate without warning, while diffusion training uses a single straightforward regression objective. **Mode coverage**: GANs frequently suffer from mode collapse, silently ignoring parts of the data distribution, while diffusion models reliably capture the full diversity of training data. **The tradeoff** is speed: GANs generate an image in a single forward pass, while diffusion models require hundreds of denoising steps. Recent work on distillation and faster samplers has narrowed this gap, but not eliminated it.

14.5 Transformers and Large Language Models

While the previous sections focused on image generation, the transformer architecture has revolutionized text generation and forms the basis of modern Large Language Models (LLMs) like GPT and BERT.

14.5.1 The Attention Mechanism

The key innovation of transformers is **self-attention**, which allows each element in a sequence to attend to (pay attention to) every other element. This captures long-range dependencies that previous architectures like RNNs struggled with.

Given an input sequence, self-attention computes:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^{\boldsymbol{T}}}{\sqrt{d_k}}\right)\mathbf{V} \tag{14.4}$$

where:

- **Q** (Query): What am I looking for?
- **K** (Key): What do I contain?
- **V** (Value): What information do I provide?

Intuition: Attention as Relevance

Think of reading a sentence like "The cat sat on the mat because *it* was tired." When processing "it," attention lets the model look back at "cat" (high attention) rather than "mat" (low attention) to understand what "it" refers to.

14.5.2 Transformer Architecture

A transformer consists of an **encoder** that processes the input sequence and understands its context, and a **decoder** that generates the output sequence while attending to both the encoder output and previously generated tokens. Some models use only the encoder (BERT), others use only the decoder (GPT), and some use both (the original transformer for translation).

Each layer contains self-attention followed by a feed-forward network, with residual connections and layer normalization.

14.5.3 Large Language Models

LLMs like GPT are transformers trained on massive text corpora to predict the next word in a sequence. Through this deceptively simple objective, they learn far more than just word prediction. They absorb grammar and syntax, accumulate facts about the world, discover reasoning patterns, and internalize diverse writing styles. The knowledge emerges implicitly from the statistical patterns of language.

GPT (Generative Pre-trained Transformer) uses only the decoder part, generating text autoregressively. One token at a time, where each new token depends on all previous tokens.

BERT (Bidirectional Encoder Representations from Transformers) uses only the encoder part, trained with masked language modeling (predicting masked words given context from both directions).

14.5.4 Tokenization

LLMs do not process raw text. They work with **tokens**: units that might be whole words, subword fragments, or individual characters. The choice of

tokenization affects model behavior in ways that matter for understanding LLM capabilities and limitations. Rare or specialized words get split into multiple tokens, making them harder for the model to process. Languages with larger character sets (such as Chinese or Arabic) require more tokens per sentence than English, increasing computational cost and reducing effective context length. When an IAIO problem asks about LLM parameter counts or computational costs, tokenization is the bridge between human-readable text and the numbers the model actually operates on.

14.5.5 Computing Parameters

Understanding the scale of LLMs requires understanding parameter counts.

Example: Embedding Matrix Parameters. A language model has vocabulary size 10,000 and embedding dimension 256. How many parameters are in the embedding matrix? Each token needs a 256-dimensional vector, so the total is $\mathbf{10{,}000 \times 256 = 2{,}560{,}000}$ parameters.

Example: Feed-Forward Network Parameters. A transformer layer has a feed-forward network with input dimension $\boldsymbol{d = 256}$ and hidden dimension $\boldsymbol{f = 1024}$. How many parameters (excluding biases)? There are two linear transformations: $\boldsymbol{d \to f}$ then $\boldsymbol{f \to d}$. The total is $\boldsymbol{d \times f + f \times d = 256 \times 1024 + 1024 \times 256 = 524{,}288}$ parameters.

14.5.6 Why Scale Works (and When It Stops)

A striking empirical finding is that LLM performance improves predictably as you increase model size, training data, and compute. These **scaling laws** follow power-law relationships:[2] doubling compute yields a consistent, measurable improvement. This predictability is what justifies the enormous investment in training ever-larger models.

But scaling has limits. More parameters and data reduce perplexity (prediction error) but do not eliminate fundamental problems. Hallucination persists because the training objective rewards plausible-sounding text, not truth. Reasoning failures persist because pattern matching, however sophisticated, is not the same as logical deduction. And the computational and environmental costs grow faster than the benefits: each new generation of models requires roughly ten times more compute for incrementally smaller improvements. Scaling is the engine of current progress, but it is unlikely to be sufficient on its own.

[2] Kaplan, J. et al. (2020). Scaling laws for neural language models. *arXiv preprint* arXiv:2001.08361.

14.6 Text-to-Image Generation

Modern text-to-image systems like DALL-E and Stable Diffusion combine the techniques discussed above. The general architecture has three stages: a text encoder (typically CLIP) converts a text prompt into a vector representation; a diffusion process, usually operating in a compressed latent space rather than pixel space for efficiency, generates an image representation conditioned on the text embedding; and a decoder converts the result to a full-resolution image. Different systems vary in architectural details, but the conceptual pipeline is the same: language understanding guides image generation.

14.7 The AI Landscape Today

These technologies have transformed what AI systems can do:

Language: Large language models can write essays, code, poetry, and engage in sophisticated dialogue. They demonstrate emergent capabilities that were not explicitly programmed.

Images: Diffusion models generate photorealistic images from text descriptions, enabling new forms of creative expression and raising questions about authenticity.

Multimodal: Systems increasingly combine modalities, understanding images and text together, generating images from text, describing images in words.

Reasoning: Recent models show improved ability to break down complex problems, though they still struggle with certain types of logical reasoning.

14.8 Ethical Considerations

The systems in this chapter amplify the ethical challenges covered in **Chapter 2**. Three properties of modern deep learning make this amplification significant.

First, these models are trained on internet-scale data rather than curated datasets, so they absorb every bias, stereotype, and piece of misinformation present in that data. The curation step that might catch problems in smaller systems is largely absent. Second, generative models produce content consumed by millions of people, so biased outputs do not just affect individual decisions; they shape cultural norms and public understanding. Third, the sheer volume of output (millions of text completions or images per day) makes meaningful human review of each output impossible.

The result is that problems like bias amplification, feedback loops, and accountability gaps, all analyzed with formal tools in Chapter 2, operate at a qualitatively different scale. For comprehensive coverage including formal definitions of fairness, the bias-detection-mitigation pipeline, and the five-point analysis template, see **Chapter 2: The Societal Impact of AI**.

14.9 Limitations and Open Problems

Despite impressive capabilities, current systems have fundamental limitations:

Hallucination: Language models confidently state false information. They have no reliable mechanism to distinguish what they know from what they don't.

Reasoning: While improving, models still struggle with multi-step logical reasoning, especially in novel domains.

Grounding: Models learn from text and images but have no direct experience of the physical world, limiting their understanding of causality and physics.

Alignment: Ensuring AI systems pursue intended goals safely remains an open research problem.

Memorization and data leakage: Models trained on internet-scale data can memorize and reproduce training examples verbatim, raising both copyright and privacy concerns. More subtly, if benchmark test sets appeared in the training data, reported performance overstates true capability. This **training set contamination** makes it difficult to evaluate how well models genuinely generalize versus how well they recall. Evaluating generative quality has no single correct metric (Chapter 10 discusses this in depth for image models, and the same challenge applies to text generation).

Evaluation: Unlike classification, where accuracy on a test set provides a clear signal, there is no single correct metric for generative quality. Different metrics capture different dimensions (fluency, factual accuracy, diversity, coherence), and optimizing one can degrade others.

14.10 Summary

Key Concepts

GANs: Generator vs. discriminator in adversarial training. Sharp outputs but unstable training and mode collapse risk.
Diffusion Models: Learn to reverse a noise-adding process. Start from pure noise and iteratively denoise to generate images. Currently produce the highest-quality image generation.
Transformers: Use attention mechanisms to process sequences. Each position can attend to all other positions, capturing long-range dependencies. Foundation of modern language models.
Attention: The key innovation: compute relevance weights between all pairs of positions, allowing flexible information routing.
Large Language Models: Transformers trained on massive text corpora. Scaling laws predict that more parameters and data improve performance, but with diminishing returns and without eliminating fundamental limitations.
Text-to-Image: Combine language understanding (often via CLIP) with image generation (diffusion models) to create images from descriptions.
Limitations: Hallucination, memorization of training data, benchmark contamination, and the absence of a single reliable evaluation metric remain fundamental challenges.

14.11 Practice Problems

1. Explain the key idea behind diffusion models. Why might iterative refinement produce better results than one-shot generation?

2. In the attention mechanism:

   (a) What do the Query, Key, and Value matrices represent conceptually?

   (b) Why is attention called "all you need" for sequence modeling?

   (c) What is the computational complexity of self-attention with sequence length n?

3. A language model has 175 billion parameters. If each parameter is stored as a 16-bit floating point number, how much memory is required just to store the model weights?

4. **(Conceptual)** Language models are trained to predict the next token.

Explain how this simple objective leads to models that can answer questions, write code, and engage in dialogue.

5. **(Critical thinking)** Large language models sometimes "hallucinate", confidently stating false information.
   (a) Why does the training objective (next-token prediction) not prevent hallucination?
   (b) Propose two approaches that might reduce hallucination.
6. **(GANs)** In a GAN, the discriminator outputs $\boldsymbol{D}(\boldsymbol{x}) \in [\mathbf{0}, \mathbf{1}]$.
   (a) What does $\boldsymbol{D}(\boldsymbol{x}) = \mathbf{0.9}$ mean?
   (b) The generator wants to maximize $\boldsymbol{D}(\boldsymbol{G}(\boldsymbol{z}))$. Explain why.
   (c) What is mode collapse and why does it happen?

Appendix A

Practical Competition Strategies

> Torture the data, and it will confess to anything.
>
> Ronald Coase

A.1 Introduction

Machine learning competitions test your ability to build models that work on real data. Whether you're competing on Kaggle or the IAIO practical round, the core workflow and strategic principles remain the same. This appendix covers platform-agnostic strategies that will help you succeed in any competition setting.

A.2 Understanding Competition Structure

A.2.1 The Data Split

Every competition provides data split into distinct sets:

Training set: Contains features and labels. You learn from this data.

Test set: Contains features only. You predict labels for this data.

Sample submission: Shows the exact format your predictions must follow.

Crucially, the test set is often further split:

Public test set: A portion (often 30-50%) used for the public leaderboard

you see during the competition.

Private test set: The remainder, used for final rankings revealed only after the competition ends.

Critical Warning: Leaderboard Overfitting

The public leaderboard shows performance on only part of the test data. If you optimize specifically for this subset (by making many submissions and selecting those that score well) you are overfitting to the public test set.
This is dangerous because:

- Public and private test performance can differ substantially
- Models that overfit to public data often drop dramatically on private data
- Many competitors have lost medal positions due to this mistake

Trust your cross-validation score, not your public leaderboard score. This is the same overfitting problem from Chapter 5, applied to a different split. Every time you check the public leaderboard and adjust your approach based on the result, you are effectively using the public test set as a validation set. With enough submissions, you will find models that happen to score well on that particular subset without genuinely generalizing better. The leaderboard score becomes a noisy, over-optimistic estimate of true performance, just as training accuracy becomes over-optimistic when you tune a model too aggressively to its training data.

A.2.2 Evaluation Metrics

Understanding the evaluation metric is essential: it defines what "better" means.

Classification metrics:

- **Accuracy**: Fraction correct. Misleading for imbalanced classes.
- **F1 Score**: Harmonic mean of precision and recall. Good for imbalanced data.
- **Log Loss**: Penalizes confident wrong predictions. Requires probability outputs.
- **AUC-ROC**: Measures ranking quality across all thresholds.

Regression metrics:

- **RMSE**: Root Mean Squared Error. Penalizes large errors heavily.

- **MAE**: Mean Absolute Error. More robust to outliers.
- R^2: Fraction of variance explained.

Different metrics require different optimization strategies. With log loss, probability calibration matters. With RMSE, outlier handling is critical.

A.3 The Competition Workflow

The following phases describe a typical workflow. The time percentages are illustrative starting points, not rigid rules. Adjust them based on the competition format, your experience, and what the data demands. Some problems need heavy feature engineering; others are won by careful model selection. The key is having a structured approach, not following exact percentages.

A.3.1 Phase 1: Understand the Problem

Before writing any code:

1. Read the problem description completely
2. Identify: Is this classification or regression?
3. Understand the evaluation metric and what it rewards
4. Check the submission format requirements
5. Read the data description, what does each column mean?
6. Look for domain knowledge that might inform feature engineering

A.3.2 Phase 2: Establish a Baseline

Create the simplest possible working pipeline:

1. Load training and test data
2. Minimal preprocessing: handle missing values simply (fill with median/mode)
3. Use all provided features without engineering
4. Train a simple model (logistic regression or small random forest)
5. Generate predictions in the correct submission format
6. Submit and record your baseline score

This baseline serves two purposes: it verifies your pipeline works end-to-end, and it gives you a reference point to measure improvements against.

A.3.3 Phase 3: Exploratory Data Analysis

Now examine your data more carefully:

- **Distributions**: Are features normally distributed? Skewed? Are there outliers?
- **Missing values**: Which columns have missing data? Is missingness random or informative?
- **Correlations**: Which features correlate with the target? With each other?
- **Data types**: Are categorical features encoded correctly?
- **Train-test differences**: Does the test set have different distributions than training?

A.3.4 Phase 4: Feature Engineering

This is often where competitions are won or lost:

- **Domain features**: Use problem knowledge to create meaningful combinations
- **Interactions**: Products or ratios of existing features
- **Aggregations**: Group statistics (mean, std by category)
- **Temporal features**: If data has timestamps, day of week, month, time since event
- **Text features**: TF-IDF, word counts, sentiment scores
- **Target encoding**: Replace categories with target statistics (careful with leakage!)

Test each feature's impact systematically. Not all features help, some add noise.

A.3.5 Phase 5: Model Development

With good features, try different models:

- **Start simple**: Linear models, small decision trees

- **Try tree ensembles**: Random Forest, XGBoost, LightGBM (often best for tabular data)
- **Consider neural networks**: For images, text, or very large datasets
- **Tune hyperparameters**: Focus on the most impactful ones (learning rate, regularization)

A.3.6 Phase 6: Validation and Selection

At the end, you typically select 2-3 final submissions. Choose wisely:

- One submission that scored best on cross-validation
- One that's slightly different (different model or features) as a hedge
- Avoid selecting based solely on public leaderboard score

A.4 Avoiding Data Leakage

Data leakage occurs when information from outside the training set improperly influences your model. It causes artificially high validation scores that don't generalize.

Common leakage sources:

- **Target leakage**: A feature is derived from or strongly correlated with the target
- **Temporal leakage**: Using future information to predict the past
- **Train-test contamination**: Test data statistics leak into training preprocessing
- **Improper cross-validation**: Not respecting time ordering or group structure

Prevention:

- Fit preprocessing (scalers, encoders) only on training folds
- For time series, always validate on future data
- Be suspicious of features that seem "too good"
- Check that cross-validation and public scores are roughly consistent

A.5 Cross-Validation Strategies

Simple random splits can be misleading. Choose validation strategy based on data structure:

K-Fold: Standard choice. Use $k = 5$ or $k = 10$.

Stratified K-Fold: For classification with imbalanced classes. Preserves class ratios in each fold.

Time-based splits: For temporal data. Train on past, validate on future.

Group K-Fold: When data has natural groups (e.g., multiple samples per user). Keep all samples from one group in the same fold.

The validation strategy should mimic how the test set relates to training data.

A.6 Ensemble Methods

Combining multiple models often improves performance:

Averaging: Simply average predictions from different models. Works well when models make uncorrelated errors.

Weighted averaging: Give better models more weight.

Stacking: Train a meta-model on the predictions of base models. More powerful but risks overfitting.

Blending: Like stacking but simpler, use holdout predictions rather than cross-validation.

Even simple averaging of 3-5 diverse models typically improves over any single model.

A.7 Time Management in Timed Competitions

For the IAIO practical round or similar timed competitions:

Before the competition:

- Have template code ready for common tasks
- Know your tools well, don't learn new libraries during competition
- Practice the full workflow on similar problems

During the competition:

- Spend first 15-20% understanding the problem and getting a baseline
- Don't chase marginal improvements early, focus on big wins first
- Leave 20-30 minutes at the end for final submission preparation
- Make sure your submission pipeline works before time runs out

Priorities when time is limited:

1. Working baseline (must have)
2. Basic feature engineering
3. One good model with reasonable hyperparameters
4. Ensemble if time permits

A.8 Common Mistakes to Avoid

1. **Overfitting to public leaderboard**: Trust CV, not public scores
2. **Wrong submission format**: Always verify format matches sample exactly
3. **Data leakage**: Be paranoid about information flow
4. **Ignoring the metric**: Optimize for what's measured, not what's convenient
5. **Overcomplicating early**: Start simple, add complexity incrementally
6. **Not saving your work**: Keep all code and model outputs versioned
7. **Running out of time**: Leave buffer for final submissions

A.9 Summary

Competition Checklist

Before starting: ☐ Read problem description completely ☐ Understand evaluation metric ☐ Check submission format
During competition: ☐ Establish working baseline first ☐ Use cross-validation, not public leaderboard ☐ Test features and models systematically ☐ Watch for data leakage
Before submitting: ☐ Verify submission format matches sample ☐ Select submissions based on CV score ☐ Leave time buffer for technical issues

The skills you develop in competitions (systematic experimentation, rigorous validation, practical feature engineering) transfer directly to real-world machine learning work. Every competition makes you a better practitioner.

Appendix B

Mathematical Notation

> Mathematical notation is a language designed to express mathematical ideas with precision and brevity.
>
> Tobias Dantzig

B.1 Introduction

Mathematics has its own language, a system of symbols that allows us to express complex ideas precisely and concisely. When you first encounter a page full of mathematical notation, it can feel like reading a foreign language. But just like any language, once you learn the vocabulary and grammar, it becomes a powerful tool for communication.

This appendix serves as your reference guide to the mathematical notation used throughout this book. We begin with general mathematical notation that you will encounter across all of mathematics, then move to notation specific to machine learning and artificial intelligence.

B.2 Numbers and Basic Operations

B.2.1 Number Sets

Mathematicians have special symbols for commonly used sets of numbers:

Symbol	Name	Description
$\mathbb{N}$	Natural numbers	$\{0, 1, 2, 3, \ldots\}$ or $\{1, 2, 3, \ldots\}$
$\mathbb{Z}$	Integers	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$
$\mathbb{Q}$	Rational numbers	Fractions $\frac{p}{q}$ where $p, q \in \mathbb{Z}$, $q \neq 0$
$\mathbb{R}$	Real numbers	All points on the number line
$\mathbb{R}^+$	Positive reals	$\{x \in \mathbb{R} : x > 0\}$
$\mathbb{R}^n$	n-dimensional space	Vectors with n real components

The blackboard bold font ($\mathbb{R}$ rather than $\boldsymbol{R}$) signals that we are talking about a standard number set.

B.2.2 Arithmetic Operations

Most arithmetic symbols are familiar, but a few deserve attention:

Symbol	Meaning	Example
$+, -, \times, \div$	Basic arithmetic	$3 + 4 = 7$
$\cdot$	Multiplication	$3 \cdot 4 = 12$
a^n	Exponentiation	$2^3 = 8$
$\sqrt{x}$, $x^{1/2}$	Square root	$\sqrt{16} = 4$
$\lvert x \rvert$	Absolute value	$\lvert -5 \rvert = 5$
$\lfloor x \rfloor$	Floor (round down)	$\lfloor 3.7 \rfloor = 3$
$\lceil x \rceil$	Ceiling (round up)	$\lceil 3.2 \rceil = 4$
$n!$	Factorial	$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$
$\binom{n}{k}$	Binomial coefficient	$\binom{5}{2} = \frac{5!}{2!(5-2)!} = 10$

B.3 Sets and Set Operations

A **set** is a collection of distinct objects. Sets are fundamental to mathematics and appear throughout machine learning, training sets, hypothesis classes, feature spaces.

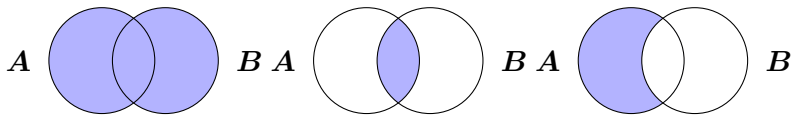
B.3.1 Set Notation

Symbol	Meaning	Example
$\{a, b, c\}$	Set with elements a, b, c	$\{1, 2, 3\}$
$\{x : \text{condition}\}$	Set of x satisfying condition	$\{x : x > 0\} =$ positive numbers
$\emptyset$ or $\{\}$	Empty set	The set with no elements
$\lvert S\rvert$	Cardinality (size) of set S	$\lvert\{a, b, c\}\rvert = 3$
$\in$	Element of	$3 \in \{1, 2, 3\}$
$\notin$	Not an element of	$4 \notin \{1, 2, 3\}$
$\subseteq$	Subset (or equal)	$\{1, 2\} \subseteq \{1, 2, 3\}$
$\subset$	Proper subset	$\{1, 2\} \subset \{1, 2, 3\}$

B.3.2 Set Operations and Their Logical Cousins

Set operations have a beautiful correspondence with logical operations. Understanding this connection helps you recognize the same ideas in different contexts.

Set	Logic	Name	Meaning
$A \cup B$	$A \vee B$	Union / OR	Elements in A **or** B (or both)
$A \cap B$	$A \wedge B$	Intersection / AND	Elements in A **and** B
A^c or $\bar{A}$	$\neg A$	Complement / NOT	Elements **not** in A
$A \setminus B$	$A \wedge \neg B$	Difference	Elements in A but not in B



$A \cup B$ (Union) $A \cap B$ (Intersection) $A \setminus B$ (Difference)

Figure B.1: Venn diagrams showing set operations. The shaded region represents the result.

The correspondence between sets and logic is not a coincidence. In both cases, we are combining conditions: "in set A" is a condition that is either true or false for any element, just like a logical proposition.

B.4 Summation and Product Notation

When we need to add or multiply many terms, writing them all out becomes impractical. Summation and product notation provide a compact way to express these operations.

B.4.1 Summation (Σ)

The Greek letter sigma (Σ) denotes summation:

$$\sum_{i=1}^{n} a_i = a_1 + a_2 + a_3 + \cdots + a_n$$

The notation specifies: start with $i = 1$, end with $i = n$, and add up all the a_i terms.

Examples of Summation

Sum of first 5 integers:

$$\sum_{i=1}^{5} i = 1 + 2 + 3 + 4 + 5 = 15$$

Sum of squares:

$$\sum_{i=1}^{4} i^2 = 1^2 + 2^2 + 3^2 + 4^2 = 1 + 4 + 9 + 16 = 30$$

Mean of data points $x_1, \ldots, x_n$:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

B.4.2 Product (Π)

The Greek letter pi (Π) denotes products:

$$\prod_{i=1}^{n} a_i = a_1 \times a_2 \times a_3 \times \cdots \times a_n$$

Examples of Products

Factorial as a product:

$$n! = \prod_{i=1}^{n} i = 1 \times 2 \times 3 \times \cdots \times n$$

Product of probabilities (for independent events):

$$P(A_1 \cap A_2 \cap \cdots \cap A_n) = \prod_{i=1}^{n} P(A_i)$$

B.5 Functions

A **function** maps inputs to outputs. The notation $f : A \to B$ means "f is a function from set A to set B." For each input $x \in A$, the function produces exactly one output $f(x) \in B$.

B.5.1 Common Function Notation

Notation	Meaning	Example
$f(x)$	Value of f at x	$f(x) = x^2$ means $f(3) = 9$
$f : A \to B$	f maps from A to B	$f : \mathbb{R} \to \mathbb{R}$
$f \circ g$	Composition: $f(g(x))$	$(f \circ g)(x) = f(g(x))$
f^{-1}	Inverse function	If $f(x) = 2x$, then $f^{-1}(x) = x/2$

B.5.2 Important Functions

Several functions appear repeatedly in machine learning:

Function	Definition	Use in ML
$\exp(x)$ or e^x	Exponential function	Softmax, probabilities
$\ln(x)$ or $\log(x)$	Natural logarithm	Cross-entropy loss
$\log_2(x)$	Logarithm base 2	Information theory, entropy
$\sigma(x) = \frac{1}{1+e^{-x}}$	Sigmoid function	Logistic regression
$\max(a, b)$	Larger of a and b	ReLU, minimax
$\min(a, b)$	Smaller of a and b	Minimax
$\text{sign}(x)$	$+1$ if $x > 0$, -1 if $x < 0$	Classification

B.5.3 The Indicator Function

The **indicator function** $\mathbf{1}[\cdot]$ returns 1 if its argument is true, 0 if false:

$$\mathbf{1}[\text{condition}] = \begin{cases} \mathbf{1} & \text{if condition is true} \\ \mathbf{0} & \text{if condition is false} \end{cases}$$

This is extremely useful for counting and for expressing piecewise functions compactly.

Example: Counting Errors

The number of misclassifications in a dataset:

$$\text{Errors} = \sum_{i=1}^{n} \mathbf{1}[\boldsymbol{h}(\boldsymbol{x}_i) \neq \boldsymbol{y}_i]$$

This adds 1 for each example where the prediction $\boldsymbol{h}(\boldsymbol{x}_i)$ differs from the true label $\boldsymbol{y}_i$.

B.6 Greek Letters

Greek letters are used extensively in mathematics and machine learning. Here are the most common ones:

Letter	Name	Common Use	Letter	Name	Common Use
$\boldsymbol{\alpha}$	alpha	Learning rate	$\boldsymbol{\nu}$	nu	Degrees of freedom
$\boldsymbol{\beta}$	beta	Coefficients	$\boldsymbol{\pi}$	pi	3.14159... or policy
$\boldsymbol{\gamma}$	gamma	Discount factor	$\boldsymbol{\rho}$	rho	Correlation
$\boldsymbol{\delta}$	delta	Small change	$\boldsymbol{\sigma}$	sigma	Standard deviation
$\boldsymbol{\epsilon}$	epsilon	Small quantity	$\boldsymbol{\tau}$	tau	Temperature, time
$\boldsymbol{\theta}$	theta	Parameters	$\boldsymbol{\phi}$	phi	Feature map
$\boldsymbol{\lambda}$	lambda	Regularization	$\boldsymbol{\psi}$	psi	Basis function
$\boldsymbol{\mu}$	mu	Mean	$\boldsymbol{\omega}$	omega	Frequency, weights

Capital Greek letters also appear: $\boldsymbol{\Sigma}$ (sigma) for summation, $\boldsymbol{\Pi}$ (pi) for products, $\boldsymbol{\Phi}$ (phi) for cumulative distribution, $\boldsymbol{\Omega}$ (omega) for sample space.

B.7 Comparisons and Logical Symbols

Symbol	Meaning	Example
$=$	Equals	$2 + 2 = 4$
$\neq$	Not equal	$3 \neq 4$
$<, >$	Less than, greater than	$3 < 5$, $7 > 2$
$\leq, \geq$	Less/greater than or equal	$x \leq 10$
$\approx$	Approximately equal	$\pi \approx 3.14$
$\propto$	Proportional to	$y \propto x$ means $y = kx$ for some k
$\equiv$	Identical / defined as	$f(x) \equiv x^2$
$:=$	Defined as (assignment)	$\mu := \frac{1}{n}\sum x_i$

B.8 Quantifiers

Quantifiers express statements about "all" or "some" elements:

Symbol	Name	Meaning
$\forall$	Universal ("for all")	$\forall x \in \mathbb{R} : x^2 \geq 0$ (all squares are non-negative)
$\exists$	Existential ("there exists")	$\exists x \in \mathbb{R} : x^2 = 2$ (some number squares to 2)
$\nexists$	"There does not exist"	$\nexists x \in \mathbb{R} : x^2 = -1$ (no real squares to -1)

B.9 Machine Learning Specific Notation

The following notation is used consistently throughout this book:

B.9.1 Data and Features

Symbol	Meaning
$\mathbf{x}$	Feature vector (input), typically $\mathbf{x} = (x_1, x_2, \ldots, x_d)$
y	Label / target (output)
$\hat{y}$	Predicted value (the "hat" indicates prediction/estimate)
n or m	Number of training examples
d	Number of features (dimensionality)
S	Training set, $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$
$\mathcal{X}$	Input space (domain)
$\mathcal{Y}$	Output space (label set)

B.9.2 Models and Learning

Symbol	Meaning
h	Hypothesis (a function from inputs to predictions)
$\mathcal{H}$	Hypothesis class (set of candidate hypotheses)
θ or $\mathbf{w}$	Model parameters / weights
h_θ	Hypothesis with parameters θ
$\mathcal{L}$ or J	Loss function / cost function
α or η	Learning rate
λ	Regularization parameter

B.9.3 Probability Notation

Symbol	Meaning
$P(A)$	Probability of event A
$P(A\|B)$	Conditional probability of A given B
$\mathbb{E}[X]$	Expected value of random variable X
$\text{Var}(X)$	Variance of X
$\mathcal{N}(\mu, \sigma^2)$	Normal distribution with mean μ, variance σ^2
$X \sim \mathcal{D}$	X is distributed according to $\mathcal{D}$

B.9.4 Calculus Notation

Symbol	Meaning
$\frac{df}{dx}$ or $f'(x)$	Derivative of f with respect to x
$\frac{\partial f}{\partial x}$	Partial derivative (when f has multiple variables)
∇f	Gradient of f (vector of all partial derivatives)
$\nabla_\theta \mathcal{L}$	Gradient of loss with respect to parameters

B.10 Reading Mathematical Expressions

When you encounter a complex expression, break it down piece by piece. Consider:

$$\hat{\theta} = \arg\min_\theta \frac{1}{n} \sum_{i=1}^{n} (y_i - h_\theta(\mathbf{x}_i))^2 + \lambda \|\theta\|^2$$

Reading this step by step:

1. $\hat{\theta}$, the estimated (optimal) parameters
2. $\arg\min_\theta$, find the θ that minimizes what follows

3. $\frac{1}{n}\sum_{i=1}^{n}$, average over all n training examples
4. $(y_i - h_\theta(\mathbf{x}_i))^2$, squared difference between true label and prediction
5. $\lambda\|\theta\|^2$, regularization term (penalty on large parameters)

In words: "Find the parameters θ that minimize the average squared error plus a regularization penalty."

B.11 Summary: Quick Reference

Essential Notation Reference

Sets: $\in$ (element of), $\subseteq$ (subset), $\cup$ (union/or), $\cap$ (intersection/and), $|S|$ (size)
Sums/Products: $\sum_{i=1}^{n} a_i$ (add all), $\prod_{i=1}^{n} a_i$ (multiply all)
Functions: $f(x)$ (value at x), $f : A \to B$ (maps A to B), $\mathbf{1}[\cdot]$ (indicator)
Comparisons: $\neq$ (not equal), $\leq$ (at most), $\geq$ (at least), $\approx$ (approximately)
Quantifiers: $\forall$ (for all), $\exists$ (there exists)
ML Data: $\mathbf{x}$ (features), y (label), $\hat{y}$ (prediction), S (training set)
ML Models: h (hypothesis), $\mathcal{H}$ (hypothesis class), θ (parameters), $\mathcal{L}$ (loss)
Probability: $P(A)$ (probability), $P(A|B)$ (conditional), $\mathbb{E}[X]$ (expectation)
Calculus: $\frac{df}{dx}$ (derivative), ∇f (gradient)

Appendix C

Probability

> Probability theory is nothing but common sense reduced to calculation.
>
> Pierre-Simon Laplace

C.1 Introduction: Why Probability?

Machine learning is fundamentally about dealing with uncertainty. We do not know the true relationship between inputs and outputs. We can only estimate it from limited data. We do not know what the next data point will look like. We can only predict based on patterns we have observed. We cannot be certain our model will generalize to new situations. We can only say it will *probably* perform well.

Probability gives us the language and tools to reason precisely about this uncertainty. Rather than saying "the model might be wrong," we can say "the model has a 3% error rate." Rather than saying "this email is probably spam," we can say "there is a 95% probability this email is spam." This precision transforms vague intuitions into quantitative statements we can work with, test, and improve.

This appendix introduces the probability concepts you need for machine learning. We build understanding through concrete examples and intuitive explanations, developing the ideas from first principles rather than assuming prior knowledge.

C.2 What Is Probability?

C.2.1 The Basic Idea

Probability measures how likely something is to happen. We express it as a number between 0 and 1, where 0 means impossible and 1 means certain. A probability of 0.5 means the event is equally likely to happen or not happen. Like flipping a fair coin.

You can also think of probability as a fraction or percentage. A probability of 0.25 is the same as $\frac{1}{4}$ or 25%. When we say "there is a 70% chance of rain," we mean the probability of rain is 0.7.

Here are some examples to build intuition:

- Flipping a fair coin and getting heads: $P = 0.5$ (50%)
- Rolling a 4 on a fair six-sided die: $P = \frac{1}{6} \approx 0.167$ (about 17%)
- Drawing the Ace of Spades from a shuffled deck: $P = \frac{1}{52} \approx 0.019$ (about 2%)
- The sun rising tomorrow: $P \approx 1$ (essentially certain)
- Rolling a 7 on a standard six-sided die: $P = 0$ (impossible)

C.2.2 Sample Spaces and Events

To work with probability precisely, we need some terminology.

The **sample space**, denoted Ω (the Greek letter omega), is the set of all possible outcomes of a random process. For a coin flip, $\Omega = \{\text{Heads}, \text{Tails}\}$, these are the only two things that can happen. For rolling a die, $\Omega = \{1, 2, 3, 4, 5, 6\}$.

An **event** is something we might want to ask about, formally, it is a subset of the sample space. The event "rolling an even number" corresponds to the subset $\{2, 4, 6\}$. The event "getting heads" corresponds to $\{\text{Heads}\}$.

The probability of an event A, written $P(A)$, tells us how likely it is that one of the outcomes in A occurs.

Example: Rolling a Die

Suppose we roll a fair six-sided die. The sample space is $\Omega = \{1, 2, 3, 4, 5, 6\}$, with each outcome equally likely.
Let A be the event "rolling an even number," so $A = \{2, 4, 6\}$.
Since 3 of the 6 equally likely outcomes are even:

$$P(A) = \frac{\text{number of outcomes in } A}{\text{total number of outcomes}} = \frac{3}{6} = 0.5$$

There is a 50% chance of rolling an even number.

This formula ($P(A) = \frac{|A|}{|\Omega|}$) works whenever all outcomes are equally likely. The notation $|A|$ means the number of elements in set A.

C.3 Basic Probability Rules

C.3.1 The Fundamental Axioms

All of probability theory is built on three simple rules:

1. **Non-negativity**: Probabilities cannot be negative. For any event A, $P(A) \geq 0$.
2. **Normalization**: Something must happen. $P(\Omega) = 1$.
3. **Additivity**: For events that cannot both occur (called **mutually exclusive** events), we can add their probabilities: $P(A \text{ or } B) = P(A) + P(B)$.

From these three rules, we can derive everything else in probability theory.

C.3.2 The Complement Rule

The **complement** of an event A, written A^c or $\bar{A}$, is the event "A does not happen." Since either A happens or it does not, and these two possibilities cover everything:

$$P(A^c) = 1 - P(A)$$

This simple rule is surprisingly powerful. Often it is much easier to calculate the probability that something *does not* happen, and then subtract from 1.

Example: At Least One Six

What is the probability of rolling at least one 6 when you roll three dice?
The direct approach would require counting: one 6, or two 6s, or three 6s, with various positions... This gets complicated quickly.
The complement approach is much simpler. The complement of "at least one 6" is "no sixes at all."
For each die, the probability of *not* rolling a 6 is $\frac{5}{6}$. Since the dice are independent (each roll does not affect the others), the probability of no sixes on all three is:

$$P(\text{no sixes}) = \frac{5}{6} \times \frac{5}{6} \times \frac{5}{6} = \left(\frac{5}{6}\right)^3 = \frac{125}{216} \approx 0.58$$

Therefore:

$$P(\text{at least one 6}) = 1 - P(\text{no sixes}) = 1 - \frac{125}{216} = \frac{91}{216} \approx 0.42$$

There is about a 42% chance of rolling at least one 6 in three rolls.

C.3.3 The Addition Rule

What if we want the probability that event A happens *or* event B happens (or both)? If the events can overlap (if some outcomes belong to both A and B) we need to be careful not to count those outcomes twice.

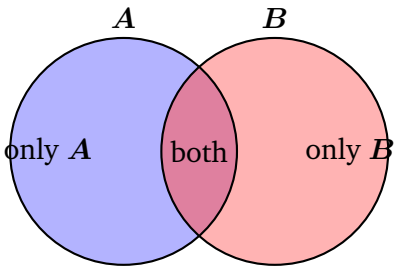


Figure C.1: A Venn diagram showing two overlapping events. The purple region belongs to both A and B.

If we simply add $P(A) + P(B)$, we count the purple overlap region twice, once as part of A and once as part of B. To correct for this, we subtract the overlap:

$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

Here $A \cup B$ means "A or B (or both)" and $A \cap B$ means "A and B (both)."

Example: Kings or Hearts

Drawing one card from a standard 52-card deck, what is the probability of getting a King or a Heart?
Let A = "drawing a King" and B = "drawing a Heart."
There are 4 Kings in the deck, so $P(A) = \frac{4}{52}$.
There are 13 Hearts in the deck, so $P(B) = \frac{13}{52}$.
But wait; one card is *both* a King and a Heart: the King of Hearts. This is the overlap: $P(A \cap B) = \frac{1}{52}$.
Using the addition rule:

$$P(\text{King or Heart}) = \frac{4}{52} + \frac{13}{52} - \frac{1}{52} = \frac{16}{52} = \frac{4}{13} \approx 0.31$$

C.4 Conditional Probability

C.4.1 The Idea

Often we want to know the probability of an event *given that we already know something else happened*. This is called **conditional probability**.

For example: What is the probability that a student passes an exam, *given that* they attended all the lectures? What is the probability that it will rain tomorrow, *given that* it is cloudy today? What is the probability that an email is spam, *given that* it contains the word "FREE"?

We write $P(A|B)$ for "the probability of A given B," read as "P of A given B."

C.4.2 How Conditional Probability Works

When we learn that B has occurred, we are restricting our attention to only those outcomes where B is true. Among those outcomes, we ask: what fraction also satisfy A?

Think of it visually using a Venn diagram. Initially, any outcome in the entire sample space Ω is possible. But once we know B occurred, we zoom in on just the B circle, this becomes our new, smaller universe of possibilities. Within this restricted universe, we ask what fraction lies in A.

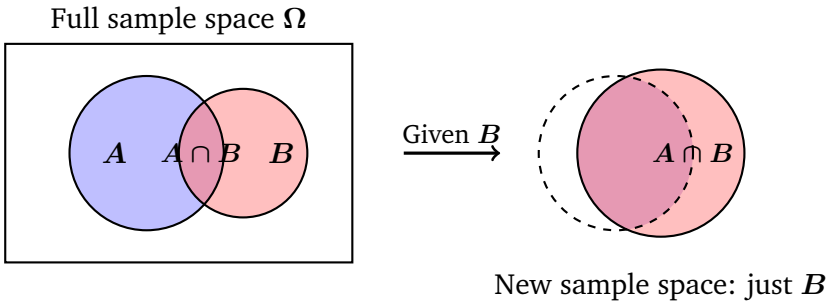


Figure C.2: Conditional probability zooms in on the event B. We ask: of all the outcomes in B, what fraction are also in A?

The formula captures this idea:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

The numerator $P(A \cap B)$ is the probability of the overlap (outcomes in both A and B). The denominator $P(B)$ normalizes by the total probability of B, since B is now our entire universe.

C.4.3 A Concrete Example with Numbers

Let us work through an example using concrete numbers to see how conditional probability connects to the Venn diagram.

Example: Students and Courses

At a small school, there are 100 students. 40 students take math, 30 take physics, and 20 take both math and physics.
If we pick a random student who takes physics, what is the probability they also take math?
Let M = "takes math" and P = "takes physics."
From the given information:

- $P(M) = 40/100 = 0.4$
- $P(P) = 30/100 = 0.3$
- $P(M \cap P) = 20/100 = 0.2$ (both math and physics)

We want $P(M|P)$, the probability of math *given* physics:

$$P(M|P) = \frac{P(M \cap P)}{P(P)} = \frac{0.2}{0.3} = \frac{2}{3} \approx 0.67$$

Two-thirds of physics students also take math.
Notice how this differs from $P(M) = 0.4$. Knowing someone takes physics *increases* our estimate that they take math (from 40% to 67%), because there is a positive association between these subjects.

C.4.4 Medical Testing: A Famous Example

One of the most important applications of conditional probability is understanding medical tests. This example demonstrates a counterintuitive result that surprises most people.

Example: Medical Testing

A disease affects 1% of the population. A test for this disease is quite accurate:

- If you have the disease, the test correctly shows positive 90% of the time (sensitivity)
- If you do not have the disease, the test correctly shows negative 95% of the time (specificity), meaning it incorrectly shows positive 5% of the time (false positive rate)

You take the test and it comes back positive. What is the probability you actually have the disease?

Most people guess something high, like 90%. The actual answer is surprisingly low.

Setting up the problem:

Let D = "has disease" and $+$ = "tests positive."

We know:

- $P(D) = 0.01$ (1% have the disease)
- $P(D^c) = 0.99$ (99% do not have the disease)
- $P(+|D) = 0.90$ (positive test given disease)
- $P(+|D^c) = 0.05$ (positive test given no disease, i.e., false positive)

We want $P(D|+)$, the probability of disease given a positive test.

Using a concrete population:

Imagine 10,000 people take the test.

- 100 people have the disease (1% of 10,000)
- 9,900 people do not have the disease

Among the 100 with the disease:

- 90 test positive (90% of 100)
- 10 test negative

Among the 9,900 without the disease:

- 495 test positive (5% of 9,900), these are false positives!
- 9,405 test negative

Total positive tests: $90 + 495 = 585$

Of these 585 positive tests, only 90 actually have the disease:

$$P(D|+) = \frac{90}{585} = \frac{90}{585} \approx 0.154$$

Only about 15%! Despite testing positive with a seemingly accurate test, you most likely do *not* have the disease.

This counterintuitive result occurs because the disease is rare. Most positive tests come from the large group of healthy people (9,900) rather than the small group with the disease (100). Even though only 5% of healthy people test positive, 5% of 9,900 is still a lot of false positives.

This example illustrates why we cannot simply equate "the test is 90% accurate" with "a positive test means 90% chance of disease." The **base rate**, how common the disease is in the population, matters enormously.

C.5 Bayes' Theorem

Many of the most consequential mistakes in applied machine learning trace back to misunderstandings about probability. Confusing $P(\text{spam}|\text{word})$ with $P(\text{word}|\text{spam})$ leads to classifiers that make confident but wrong predictions. Ignoring base rates leads to medical screening tests that sound impressive but produce mostly false positives (as we just saw). Assuming independence when features are correlated leads to probability estimates that are wildly miscalibrated. The tools in this section, especially Bayes' theorem, are not abstract mathematics. They are the defense against these errors.

The medical testing example hints at a general pattern: we often know $P(B|A)$ but want $P(A|B)$. Bayes' theorem provides the tool to flip conditionals.

C.5.1 The Formula

Bayes' theorem states:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

This allows us to compute $P(A|B)$ if we know $P(B|A)$, $P(A)$, and $P(B)$.

C.5.2 Understanding the Parts

Each component in Bayes' theorem has a name:

- $P(A)$ is the **prior**. Our belief about A before seeing any evidence
- $P(B|A)$ is the **likelihood**, how probable the evidence B is if A is true
- $P(A|B)$ is the **posterior**, our updated belief about A after seeing evidence B
- $P(B)$ is the **evidence** (or marginal likelihood), the total probability of observing B

The intuition: we start with a prior belief, then update it based on how likely the evidence would be under different scenarios.

C.5.3 Example: Spam Detection

Suppose we are building a spam filter. We want to know: given that an email contains the word "FREE," what is the probability it is spam?

Let S = "email is spam" and F = "email contains FREE."

From analyzing past emails, we know:

- $P(S) = 0.3$ (30% of emails are spam), the prior
- $P(F|S) = 0.8$ (80% of spam contains "FREE"), the likelihood
- $P(F|S^c) = 0.1$ (10% of legitimate emails contain "FREE")

First, we need $P(F)$, the overall probability an email contains "FREE":

$$P(F) = P(F|S) \cdot P(S) + P(F|S^c) \cdot P(S^c) = 0.8 \times 0.3 + 0.1 \times 0.7 = 0.24 + 0.07 = 0.31$$

Now apply Bayes' theorem:

$$P(S|F) = \frac{P(F|S) \cdot P(S)}{P(F)} = \frac{0.8 \times 0.3}{0.31} = \frac{0.24}{0.31} \approx 0.77$$

An email containing "FREE" has about a 77% chance of being spam. Much higher than the baseline 30%.

C.5.4 Bayes' Theorem in Machine Learning

In classification, we want to determine which class a data point belongs to, given its features. Bayes' theorem gives us:

$$P(\text{class}|\text{features}) = \frac{P(\text{features}|\text{class}) \cdot P(\text{class})}{P(\text{features})}$$

This is exactly what **Naïve Bayes classifiers** compute. They estimate $P(\text{features}|\text{class})$ from training data and use Bayes' theorem to classify new examples.

C.6 Independence

C.6.1 What Independence Means

Two events are **independent** if knowing that one occurred tells you nothing about whether the other occurred. Mathematically, A and B are independent if:

$$P(A|B) = P(A)$$

Learning that B happened does not change our probability estimate for A.

An equivalent definition (which is often easier to check) is:

$$P(A \cap B) = P(A) \cdot P(B)$$

For independent events, the probability of both happening equals the product of their individual probabilities.

C.6.2 Examples

Independent events: Flipping a coin twice. Whether the first flip is heads has no effect on the second flip.

$$P(\text{2nd heads}|\text{1st heads}) = P(\text{2nd heads}) = 0.5$$

The coin has no memory.

Not independent: Drawing two cards from a deck *without* replacement. If the first card is an ace, there are now fewer aces remaining, so:

$$P(\text{2nd ace}|\text{1st ace}) = \frac{3}{51} \neq P(\text{2nd ace}) = \frac{4}{52}$$

The first draw affects the second.

Not independent: A student's grades in math and physics. Students who do well in math tend to do well in physics. Knowing someone got an A in math increases our estimate of their physics grade.

C.6.3 Why Independence Matters

Independence dramatically simplifies probability calculations. For independent events $A_1, A_2, \ldots, A_n$:

$$P(A_1 \cap A_2 \cap \cdots \cap A_n) = P(A_1) \cdot P(A_2) \cdots P(A_n)$$

We can simply multiply the individual probabilities.

This is why we could calculate "no sixes in three dice rolls" as $(\frac{5}{6})^3$. The dice rolls are independent, so we multiply.

C.6.4 Independence in Machine Learning

The **Naïve Bayes** classifier assumes that features are independent given the class. This means:

$$P(\text{feature}_1 \text{ and feature}_2|\text{class}) = P(\text{feature}_1|\text{class}) \cdot P(\text{feature}_2|\text{class})$$

This assumption is almost always false in practice! Features like "income" and "education" are correlated. They are not independent. A person's height and weight are correlated. Words in a document are correlated ("machine" often appears near "learning").

Despite this, Naïve Bayes often works surprisingly well. The independence assumption is wrong, but the resulting classifier frequently makes good predictions anyway. This is one of the surprising empirical facts about machine learning: methods based on incorrect assumptions can still be practically useful. The classifier might not give perfectly calibrated probabilities, but it often ranks examples correctly (spam vs. not spam, for instance).

C.7 Random Variables

C.7.1 What Is a Random Variable?

A **random variable** is a quantity whose value is determined by a random process. We use capital letters like X, Y, Z for random variables, and lowercase letters like x, y, z for specific values they might take.

Think of a random variable as a measurement you could make on a random outcome:

- X = the number showing when you roll a die (values: 1, 2, 3, 4, 5, 6)
- Y = the number of heads when you flip 10 coins (values: 0, 1, 2, ..., 10)
- Z = the height of a randomly selected person (values: any positive number)

C.7.2 Discrete vs. Continuous

Discrete random variables take values from a countable set. Usually integers. The number of customers arriving at a store, the number of emails you receive, the outcome of a die roll.

Continuous random variables take values from a continuous range, any real number in some interval. A person's exact height, the temperature tomorrow, the time until a light bulb burns out.

The distinction matters because we handle probabilities differently. For discrete variables, we can ask "What is $P(X = 3)$?" For continuous variables, the probability of any exact value is zero (there are infinitely many possibilities), so we instead ask about ranges: "What is $P(2 < X < 4)$?"

C.7.3 Probability Distributions

A **probability distribution** describes how likely each value (or range of values) is.

For a discrete random variable, we list $P(X = x)$ for each possible value x. For a fair die:

$$P(X = 1) = P(X = 2) = P(X = 3) = P(X = 4) = P(X = 5) = P(X = 6) = \frac{1}{6}$$

For a continuous random variable, we specify a **probability density function** (PDF), often written $p(x)$ or $f(x)$. The probability of falling in an interval is the area under the curve over that interval.

C.8 Expected Value

C.8.1 The Idea

The **expected value** of a random variable, also called the **mean**, is the long-run average value you would observe if you repeated the random process many times.

If you roll a die once, you get a specific number. But if you roll it a million times and average all the results, what would you get?

C.8.2 Computing Expected Value

For a discrete random variable, the expected value is computed by multiplying each possible value by its probability and adding up:

$$\mathbb{E}[X] = \sum_{x} x \cdot P(X = x)$$

The notation $\mathbb{E}[X]$ means "expected value of X."

Example: Expected Value of a Die Roll

For a fair die, each number 1 through 6 has probability $\frac{1}{6}$:

$$\mathbb{E}[X] = 1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + 3 \cdot \frac{1}{6} + 4 \cdot \frac{1}{6} + 5 \cdot \frac{1}{6} + 6 \cdot \frac{1}{6}$$
$$= \frac{1 + 2 + 3 + 4 + 5 + 6}{6} = \frac{21}{6} = 3.5$$

The expected value is 3.5, even though you can never actually roll a 3.5! If you rolled a die many times, your average would approach 3.5.

C.8.3 Properties of Expected Value

Expected value has a very useful property called **linearity**:

$$\mathbb{E}[aX + b] = a \cdot \mathbb{E}[X] + b$$

$$\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$$

The second property is remarkable: the expected value of a sum equals the sum of expected values, *even if X and Y are not independent*. This makes expected values much easier to work with than probabilities.

C.9 Variance and Standard Deviation

C.9.1 Why We Need Variance

Expected value tells us the "center" of a distribution, but two distributions can have the same center yet look very different. Consider:

- Distribution A: Always equals 5. Expected value = 5.
- Distribution B: Equals 0 or 10 with equal probability. Expected value = 5.

Both have the same mean, but Distribution B is much more spread out. **Variance** measures this spread.

C.9.2 Definition

Variance measures how far values typically fall from the mean:

$$\text{Var}(X) = \mathbb{E}[(X - \mathbb{E}[X])^2]$$

This is the expected squared distance from the mean. We square the distances so that positive and negative deviations do not cancel out.

An equivalent formula that is often easier to compute:

$$\text{Var}(X) = \mathbb{E}[X^2] - (\mathbb{E}[X])^2$$

C.9.3 Standard Deviation

Because variance involves squared values, its units are the square of the original units (e.g., if X is in meters, variance is in meters squared). To get back to the original units, we take the square root:

$$\text{SD}(X) = \sigma = \sqrt{\text{Var}(X)}$$

Standard deviation is the most commonly reported measure of spread because it is in the same units as the data.

Example: Variance and Standard Deviation of a Die Roll

We already know $\mathbb{E}[X] = 3.5$. To find variance, we first need $\mathbb{E}[X^2]$:

$$\mathbb{E}[X^2] = 1^2 \cdot \frac{1}{6} + 2^2 \cdot \frac{1}{6} + 3^2 \cdot \frac{1}{6} + 4^2 \cdot \frac{1}{6} + 5^2 \cdot \frac{1}{6} + 6^2 \cdot \frac{1}{6}$$
$$= \frac{1 + 4 + 9 + 16 + 25 + 36}{6} = \frac{91}{6} \approx 15.17$$

Now:

$$\text{Var}(X) = \mathbb{E}[X^2] - (\mathbb{E}[X])^2 = 15.17 - (3.5)^2 = 15.17 - 12.25 = 2.92$$

$$\text{SD}(X) = \sqrt{2.92} \approx 1.71$$

A typical die roll is about 1.71 away from the mean of 3.5.

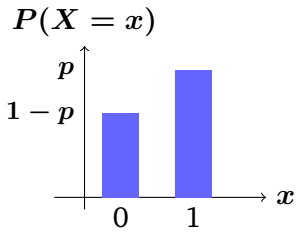
C.10 Common Probability Distributions

Certain probability distributions appear so frequently that they have names and well-studied properties.

C.10.1 Bernoulli Distribution

The simplest distribution: a single yes/no trial with probability p of "success."

$$P(X = 1) = p, \qquad P(X = 0) = 1 - p$$



Bernoulli with $p = 0.6$

Figure C.3: A Bernoulli distribution represents a single binary outcome.

Mean: $\mathbb{E}[X] = p$

Variance: $\text{Var}(X) = p(1-p)$

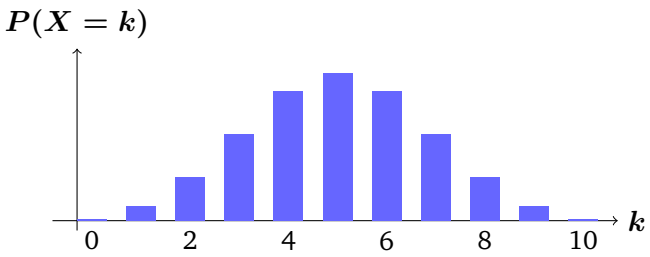
Examples: A single coin flip (heads = 1, tails = 0), whether a customer makes a purchase, whether an email is spam.

C.10.2 Binomial Distribution

If we repeat a Bernoulli trial n independent times and count the number of successes, we get a **Binomial** distribution.

$$P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

where $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ is the number of ways to choose which k of the n trials are successes.



Binomial with $n = 10$, $p = 0.5$

Figure C.4: The Binomial distribution: number of successes in n independent trials. Here, $n = 10$ and $p = 0.5$ (like flipping 10 fair coins).

Mean: $\mathbb{E}[X] = np$

Variance: $\text{Var}(X) = np(1-p)$

Examples: Number of heads in 10 coin flips, number of defective items in a batch of 100, number of correct answers when guessing on a 20-question multiple choice test.

C.10.3 Gaussian (Normal) Distribution

The **Gaussian** or **Normal** distribution is the famous "bell curve." It is the most important distribution in statistics because of the Central Limit Theorem: when you average many independent random quantities, the result tends toward a Gaussian distribution regardless of what the original distributions looked like.

The probability density function is:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

We write $X \sim \mathcal{N}(\mu, \sigma^2)$ to mean "X follows a Normal distribution with mean μ and variance σ^2."

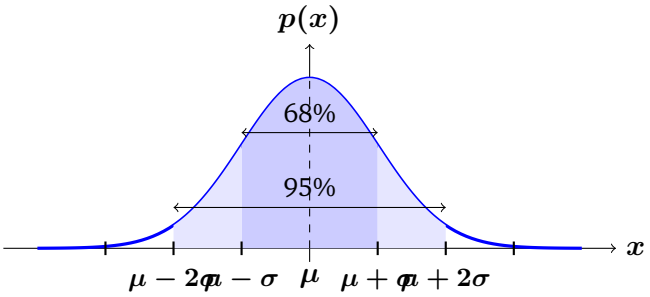


Figure C.5: The Gaussian (Normal) distribution. About 68% of values fall within one standard deviation of the mean, and about 95% fall within two standard deviations.

The key features of the Normal distribution:

- It is symmetric around the mean μ
- The parameter σ (standard deviation) controls the width
- About 68% of values fall within $\mu \pm \sigma$ (one standard deviation)
- About 95% of values fall within $\mu \pm 2\sigma$ (two standard deviations)
- About 99.7% of values fall within $\mu \pm 3\sigma$ (three standard deviations)

This "68-95-99.7 rule" is worth remembering, it gives quick intuition about how spread out data is.

Examples: Heights of people, measurement errors, test scores (approximately), stock returns (approximately).

C.11 Summary

Probability Quick Reference

Basic Rules:

- $0 \leq P(A) \leq 1$
- $P(A^c) = 1 - P(A)$ (complement)
- $P(A \cup B) = P(A) + P(B) - P(A \cap B)$ (addition rule)

Conditional Probability: $P(A|B) = \frac{P(A \cap B)}{P(B)}$

Bayes' Theorem: $P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$

Independence: A and B are independent iff $P(A \cap B) = P(A) \cdot P(B)$

Expected Value: $\mathbb{E}[X] = \sum_x x \cdot P(X = x)$ (discrete)

Variance: $\text{Var}(X) = \mathbb{E}[X^2] - (\mathbb{E}[X])^2$

Standard Deviation: $\sigma = \sqrt{\text{Var}(X)}$

Key Distributions:

- Bernoulli: Single yes/no trial with probability p
- Binomial: Number of successes in n independent trials
- Gaussian: The bell curve $\mathcal{N}(\mu, \sigma^2)$; 68% within $\pm\sigma$, 95% within $\pm 2\sigma$

Appendix D

Calculus

> The calculus is the greatest aid we have to the appreciation of physical truth.
>
> William Fogg Osgood

D.1 Introduction: Why Calculus for Machine Learning?

Machine learning is about finding the best model, the one that makes the fewest errors. But how do we find "the best"? If a model has parameters (like weights in a neural network), we need to find the parameter values that minimize errors.

Imagine you are blindfolded in a hilly landscape, trying to find the lowest valley. You cannot see, but you can feel the ground beneath your feet. If the ground slopes downward to your left, you step left. If it slopes downward in front of you, you step forward. By always stepping in the downhill direction, you eventually reach a valley bottom where the ground is flat in all directions.

This is exactly how machine learning works. The "landscape" is the error (loss) as a function of parameters. Calculus tells us the slope at any point, and we use that slope to step toward lower error. This process is called **gradient descent**, and it is how we train nearly every modern machine learning model.

To understand gradient descent, we need to understand slopes. That is what calculus provides.

D.2 Slope: The Key Idea

D.2.1 What is Slope?

You already know slope intuitively. A steep hill has a large slope. A flat road has zero slope. A downhill road has negative slope.

Numerically, slope measures how much y changes when x changes:

$$\text{slope} = \frac{\text{change in } y}{\text{change in } x} = \frac{\Delta y}{\Delta x}$$

Example: Calculating Slope

A road rises 3 meters over a horizontal distance of 100 meters.

$$\text{slope} = \frac{3 \text{ m}}{100 \text{ m}} = 0.03 = 3\%$$

A mountain trail rises 50 meters over 100 meters horizontally.

$$\text{slope} = \frac{50}{100} = 0.5 = 50\%$$

The trail is much steeper!

D.2.2 Slope of a Straight Line

For a straight line, the slope is the same everywhere. The line $y = 2x + 1$ has slope 2: whenever x increases by 1, y increases by 2.

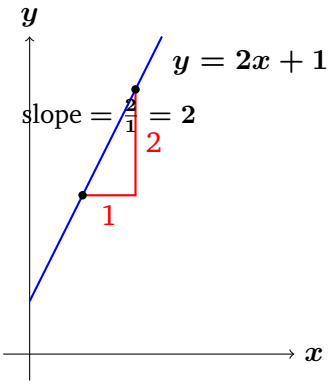


Figure D.1: The line $y = 2x + 1$ has constant slope 2.

D.2.3 The Problem with Curves

But what about curves? The function $y = x^2$ is not straight, it curves upward. What is its slope?

The answer: *the slope is different at every point*. Near $x = 0$, the curve is almost flat (slope near 0). At $x = 2$, it is steeper. At $x = 3$, steeper still.

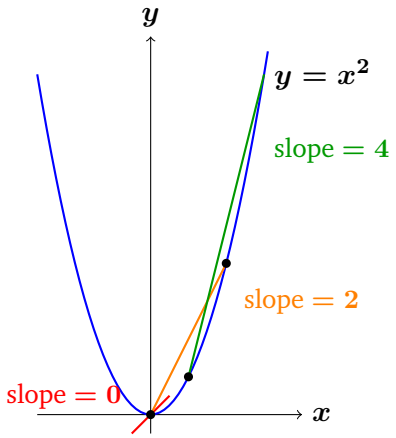


Figure D.2: The parabola $y = x^2$ has different slopes at different points.

Calculus gives us a tool to find the slope at any point on a curve. This tool is called the **derivative**.

D.3 The Derivative: Slope at a Point

D.3.1 The Tangent Line Idea

To find the slope of a curve at a specific point, we draw a line that just touches the curve at that point, called the **tangent line**. The slope of this tangent line is the slope of the curve at that point.

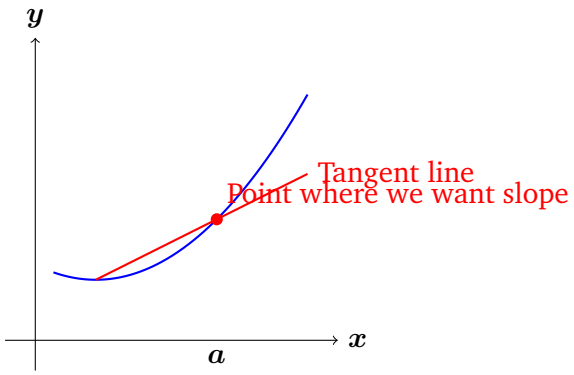


Figure D.3: The derivative at $x = a$ is the slope of the tangent line at that point.

D.3.2 Definition of the Derivative

The **derivative** of a function f at a point x, written $f'(x)$, is the slope of the tangent line at that point.

Formally, we find this by taking two points very close together and computing the slope between them:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

This formula says: take a tiny step h from x, compute how much f changes, divide by h, and let h become infinitesimally small.

D.3.3 Alternative Notations

You will see derivatives written in several equivalent ways:

$$f'(x) = \frac{df}{dx} = \frac{d}{dx}f(x) = \frac{dy}{dx} \quad \text{(if } y = f(x)\text{)}$$

The notation $\frac{df}{dx}$ emphasizes that the derivative is a ratio: how much f changes per unit change in x.

D.3.4 Computing a Derivative from the Definition

Let's find the derivative of $f(x) = x^2$ step by step using the definition.

Step 1: Write $f(x+h)$:

$$f(x+h) = (x+h)^2 = x^2 + 2xh + h^2$$

Step 2: Compute $f(x+h) - f(x)$:

$$f(x+h) - f(x) = (x^2 + 2xh + h^2) - x^2 = 2xh + h^2$$

Step 3: Divide by h:

$$\frac{f(x+h) - f(x)}{h} = \frac{2xh + h^2}{h} = 2x + h$$

Step 4: Let $h \to 0$:

$$f'(x) = \lim_{h \to 0} (2x + h) = 2x$$

Result: The derivative of x^2 is $2x$.

Let's verify this makes sense by checking specific points:

x	$f(x) = x^2$	$f'(x) = 2x$ (slope)
0	0	0 (flat at the bottom)
1	1	2 (rising)
2	4	4 (steeper)
3	9	6 (even steeper)
-1	1	-2 (falling, left side)

The negative slope at $x = -1$ makes sense: on the left side of the parabola, the curve is going downward as we move right.

D.3.5 What the Derivative Tells Us

The sign and magnitude of the derivative reveal how the function behaves:

If $f'(x)$ is...	**Then the function is...**
Positive	Increasing (going up as x increases)
Negative	Decreasing (going down as x increases)
Zero	Flat (neither increasing nor decreasing)
Large positive	Increasing steeply
Large negative	Decreasing steeply

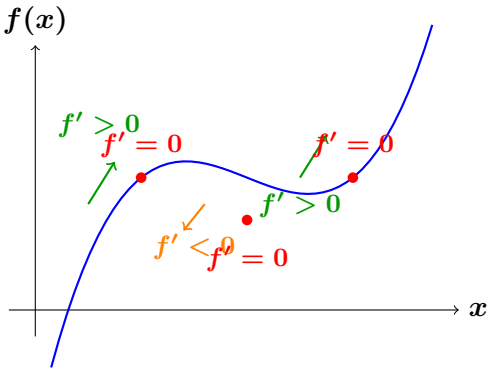


Figure D.4: The sign of the derivative indicates whether the function is increasing or decreasing. The derivative is zero at peaks and valleys.

D.4 Derivative Rules: Shortcuts

Computing derivatives from the limit definition every time would be tedious. Mathematicians have discovered rules that let us find derivatives quickly.

A Note on This Section

The derivative rules below overlap substantially with what is taught in secondary-school calculus courses. If you are currently taking or have taken calculus, much of this will be familiar. If you find any of the rules confusing, your math teacher would very likely be happy to walk through them with you. These are among the most well-trodden topics in mathematics education, and an experienced teacher can often explain a tricky rule in five minutes at the blackboard better than any textbook can on paper.

D.4.1 The Power Rule

For $f(x) = x^n$, the derivative is:

$$\boxed{f'(x) = nx^{n-1}}$$

"Bring down the exponent as a coefficient, then reduce the exponent by one."

Examples: Power Rule

$$
\begin{aligned}
f(x) = x^2 &\implies f'(x) = 2x^{2-1} = 2x \\
f(x) = x^3 &\implies f'(x) = 3x^{3-1} = 3x^2 \\
f(x) = x^5 &\implies f'(x) = 5x^4 \\
f(x) = x^1 = x &\implies f'(x) = 1 \cdot x^0 = 1 \\
f(x) = x^0 = 1 &\implies f'(x) = 0 \cdot x^{-1} = 0 \quad \text{(constants have slope zero)}
\end{aligned}
$$

The power rule also works for negative and fractional exponents:

Power Rule with Negative and Fractional Exponents

$$
\begin{aligned}
f(x) = x^{-1} = \frac{1}{x} &\implies f'(x) = -1 \cdot x^{-2} = -\frac{1}{x^2} \\
f(x) = x^{-2} = \frac{1}{x^2} &\implies f'(x) = -2 \cdot x^{-3} = -\frac{2}{x^3} \\
f(x) = x^{1/2} = \sqrt{x} &\implies f'(x) = \frac{1}{2}x^{-1/2} = \frac{1}{2\sqrt{x}} \\
f(x) = x^{1/3} = \sqrt[3]{x} &\implies f'(x) = \frac{1}{3}x^{-2/3} = \frac{1}{3x^{2/3}}
\end{aligned}
$$

D.4.2 Constant Multiple Rule

If you multiply a function by a constant, the derivative is multiplied by the same constant:

$$\boxed{\frac{d}{dx}[c \cdot f(x)] = c \cdot f'(x)}$$

Constants "come along for the ride."

Examples: Constant Multiple Rule

$$
\begin{aligned}
f(x) = 5x^3 &\implies f'(x) = 5 \cdot 3x^2 = 15x^2 \\
f(x) = -2x^4 &\implies f'(x) = -2 \cdot 4x^3 = -8x^3 \\
f(x) = \frac{x^2}{3} = \frac{1}{3}x^2 &\implies f'(x) = \frac{1}{3} \cdot 2x = \frac{2x}{3}
\end{aligned}
$$

D.4.3 Sum and Difference Rules

The derivative of a sum is the sum of derivatives:

$$\boxed{\frac{d}{dx}[f(x) + g(x)] = f'(x) + g'(x)}$$

Similarly for differences: $\frac{d}{dx}[f(x) - g(x)] = f'(x) - g'(x)$.

This means we can differentiate term by term.

Example: Combining Rules

Find the derivative of $f(x) = 3x^4 - 2x^3 + 7x^2 - x + 5$.
Differentiate term by term:

$$\begin{aligned} f'(x) &= 3 \cdot 4x^3 - 2 \cdot 3x^2 + 7 \cdot 2x - 1 + 0 \\ &= 12x^3 - 6x^2 + 14x - 1 \end{aligned}$$

D.4.4 Important Special Derivatives

Some functions appear so frequently that their derivatives are worth memorizing:

Function	Derivative	Note
e^x	e^x	The only function equal to its own derivative!
a^x	$a^x \ln(a)$	For any positive constant a
$\ln(x)$	$\frac{1}{x}$	Natural logarithm (base e)
$\log_a(x)$	$\frac{1}{x \ln(a)}$	Logarithm with base a
$\sin(x)$	$\cos(x)$	
$\cos(x)$	$-\sin(x)$	Note the minus sign

The fact that e^x is its own derivative makes it extraordinarily important in mathematics and appears throughout machine learning (in softmax, in probability distributions, in activation functions).

D.4.5 The Chain Rule

The **chain rule** handles *composite functions*, functions inside other functions.

The Problem

What is the derivative of $(x^2 + 1)^3$?

This is *not* simply $x^6 + \ldots$ because you cannot just expand and differentiate. Well, you could, but it is tedious... The function has structure: there is an "inner" function $x^2 + 1$ and an "outer" function that cubes whatever is inside.

The Rule

If $y = f(g(x))$, a function of a function, then:

$$\boxed{\frac{dy}{dx} = f'(g(x)) \cdot g'(x)}$$

In words: **derivative of outer (evaluated at inner) $\times$ derivative of inner**.

An Intuitive Way to Think About It

Imagine a chain of dependencies:

- Temperature T depends on altitude a: as you go higher, it gets colder
- Altitude a depends on time t: as you climb, altitude increases

How fast does temperature change with time?

If you climb 100 meters per hour, and temperature drops 0.6°C per 100 meters, then temperature drops 0.6°C per hour. The rates multiply!

$$\frac{dT}{dt} = \frac{dT}{da} \cdot \frac{da}{dt}$$

This is the chain rule.

Example: Chain Rule

Find $\frac{d}{dx}(x^2 + 1)^3$.
Identify the parts:

- Outer function: $f(u) = u^3 \implies f'(u) = 3u^2$
- Inner function: $g(x) = x^2 + 1 \implies g'(x) = 2x$

Apply the chain rule:

$$\frac{d}{dx}(x^2+1)^3 = \underbrace{3(x^2+1)^2}_{\text{outer derivative at inner}} \cdot \underbrace{2x}_{\text{inner derivative}} = 6x(x^2+1)^2$$

Example: Chain Rule with Exponential

Find $\frac{d}{dx}e^{-x^2}$.
- Outer: $f(u) = e^u \implies f'(u) = e^u$
- Inner: $g(x) = -x^2 \implies g'(x) = -2x$

$$\frac{d}{dx}e^{-x^2} = e^{-x^2} \cdot (-2x) = -2xe^{-x^2}$$

Example: Chain Rule with Square Root

Find $\frac{d}{dx}\sqrt{x^3 + 2x}$.
Rewrite as $(x^3 + 2x)^{1/2}$.
- Outer: $f(u) = u^{1/2} \implies f'(u) = \frac{1}{2}u^{-1/2} = \frac{1}{2\sqrt{u}}$
- Inner: $g(x) = x^3 + 2x \implies g'(x) = 3x^2 + 2$

$$\frac{d}{dx}\sqrt{x^3 + 2x} = \frac{1}{2\sqrt{x^3 + 2x}} \cdot (3x^2 + 2) = \frac{3x^2 + 2}{2\sqrt{x^3 + 2x}}$$

D.4.6 The Product Rule

When two functions are *multiplied*, we need the product rule:

$$\boxed{\frac{d}{dx}[f(x) \cdot g(x)] = f'(x) \cdot g(x) + f(x) \cdot g'(x)}$$

In words: (derivative of first $\times$ second) $+$ (first $\times$ derivative of second).

Example: Product Rule

Find $\frac{d}{dx}[x^2 \cdot e^x]$.
Let $f(x) = x^2$ and $g(x) = e^x$.
Then $f'(x) = 2x$ and $g'(x) = e^x$.

$$\begin{aligned}\frac{d}{dx}[x^2 \cdot e^x] &= f'(x) \cdot g(x) + f(x) \cdot g'(x) \\ &= 2x \cdot e^x + x^2 \cdot e^x \\ &= e^x(2x + x^2) \\ &= xe^x(2 + x)\end{aligned}$$

Example: Product Rule with Three Functions

Find $\frac{d}{dx}[x \cdot \sin(x) \cdot e^x]$.
For three functions, apply the product rule iteratively, or use the pattern:

$$(fgh)' = f'gh + fg'h + fgh'$$

Let $f = x$, $g = \sin(x)$, $h = e^x$.

$$\frac{d}{dx}[x \sin(x) e^x] = 1 \cdot \sin(x) \cdot e^x + x \cdot \cos(x) \cdot e^x + x \cdot \sin(x) \cdot e^x$$

$$= e^x[\sin(x) + x\cos(x) + x\sin(x)]$$

D.4.7 The Quotient Rule

When one function is *divided* by another:

$$\boxed{\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{f'(x) \cdot g(x) - f(x) \cdot g'(x)}{[g(x)]^2}}$$

A memory aid: "Low d-high minus high d-low, over low squared" (where "d" means "derivative of").

Example: Quotient Rule

Find $\frac{d}{dx}\left[\frac{x^2}{x+1}\right]$.
Let $f(x) = x^2$ (high) and $g(x) = x + 1$ (low).
Then $f'(x) = 2x$ and $g'(x) = 1$.

$$\frac{d}{dx}\left[\frac{x^2}{x+1}\right] = \frac{2x \cdot (x+1) - x^2 \cdot 1}{(x+1)^2}$$

$$= \frac{2x^2 + 2x - x^2}{(x+1)^2}$$

$$= \frac{x^2 + 2x}{(x+1)^2} = \frac{x(x+2)}{(x+1)^2}$$

Connection to other rules: The quotient rule can actually be derived from the product rule and chain rule. If you write $\frac{f}{g} = f \cdot g^{-1}$, then apply the product rule and chain rule (for g^{-1}), you get the quotient rule. This is why we introduced the chain rule first!

D.5 Partial Derivatives: Functions of Multiple Variables

D.5.1 The Need for Partial Derivatives

So far, our functions have had one input. But real-world quantities often depend on multiple variables:

- The price of a house depends on square footage, number of bedrooms, location, and age
- The temperature at a point depends on latitude, longitude, altitude, and time
- A neural network's loss depends on thousands or millions of weight parameters

If $f(x, y)$ depends on both x and y, what is its "slope"? The answer: it depends on which direction you are moving!

D.5.2 Definition of Partial Derivatives

A **partial derivative** measures how f changes when we vary *one* input while holding all others constant.

The partial derivative of f with respect to x, written $\frac{\partial f}{\partial x}$, is:

$$\frac{\partial f}{\partial x} = \lim_{h \to 0} \frac{f(x + h, y) - f(x, y)}{h}$$

The symbol ∂ (a curly "d") indicates a partial derivative. It reminds us that we are only considering part of the total change.

To compute $\frac{\partial f}{\partial x}$: Treat all variables except x as constants, then differentiate with respect to x using the usual rules.

Example: Computing Partial Derivatives

Let $f(x, y) = x^2 + 3xy + y^2$.
Partial derivative with respect to x:
Treat y as a constant. The term x^2 differentiates to $2x$. The term $3xy$ is like $3y \cdot x$ where $3y$ is a constant, so it differentiates to $3y$. The term y^2 is a constant (with respect to x), so it differentiates to 0.

$$\frac{\partial f}{\partial x} = 2x + 3y + 0 = 2x + 3y$$

Partial derivative with respect to y:
Now treat x as a constant. The term x^2 is constant, giving 0. The term $3xy$ differentiates to $3x$. The term y^2 differentiates to $2y$.

$$\frac{\partial f}{\partial y} = 0 + 3x + 2y = 3x + 2y$$

D.5.3 Evaluating Partial Derivatives at Specific Points

Partial derivatives are themselves functions. We can evaluate them at specific points to get numbers.

Example: Partial Derivatives at a Point

For $f(x, y) = x^2 + 3xy + y^2$, we found:

$$\frac{\partial f}{\partial x} = 2x + 3y, \qquad \frac{\partial f}{\partial y} = 3x + 2y$$

At the point $(2, 1)$:

$$\left.\frac{\partial f}{\partial x}\right|_{(2,1)} = 2(2) + 3(1) = 4 + 3 = 7$$

$$\left.\frac{\partial f}{\partial y}\right|_{(2,1)} = 3(2) + 2(1) = 6 + 2 = 8$$

This means: at the point $(2, 1)$, if we increase x slightly while keeping $y = 1$, f increases at a rate of 7 units per unit increase in x. If we increase y slightly while keeping $x = 2$, f increases at a rate of 8 units per unit increase in y.

D.5.4 More Examples

Example: Partial Derivatives of a Product

Let $f(x, y) = x^2y^3$.

$$\frac{\partial f}{\partial x} = 2x \cdot y^3 = 2xy^3 \quad \text{(treating } y^3 \text{ as a constant)}$$
$$\frac{\partial f}{\partial y} = x^2 \cdot 3y^2 = 3x^2y^2 \quad \text{(treating } x^2 \text{ as a constant)}$$

Example: Partial Derivatives with Exponentials

Let $f(x, y) = e^{xy}$.
Here we need the chain rule. The inner function is xy.

$$\frac{\partial f}{\partial x} = e^{xy} \cdot \frac{\partial}{\partial x}(xy) = e^{xy} \cdot y = ye^{xy}$$
$$\frac{\partial f}{\partial y} = e^{xy} \cdot \frac{\partial}{\partial y}(xy) = e^{xy} \cdot x = xe^{xy}$$

Example: Three Variables

Let $f(x, y, z) = x^2 + 2xy + yz + z^3$.

$$\frac{\partial f}{\partial x} = 2x + 2y + 0 + 0 = 2x + 2y$$
$$\frac{\partial f}{\partial y} = 0 + 2x + z + 0 = 2x + z$$
$$\frac{\partial f}{\partial z} = 0 + 0 + y + 3z^2 = y + 3z^2$$

D.5.5 Geometric Interpretation

Consider a surface $z = f(x, y)$, imagine a hilly landscape where height z depends on your east-west position (x) and north-south position (y).

$\frac{\partial f}{\partial x}$ is the slope of the surface in the x-direction (east-west). If you stand at a point and walk due east, this is how steeply you climb or descend.

$\frac{\partial f}{\partial y}$ is the slope in the y-direction (north-south).

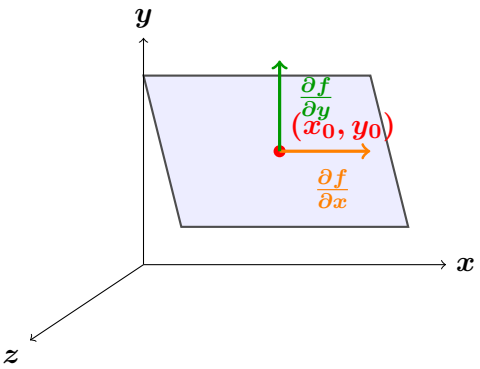


Figure D.5: Partial derivatives measure the slope in each coordinate direction separately.

D.6 The Gradient

D.6.1 Definition

The **gradient** of a function f collects all its partial derivatives into a single vector:

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix}$$

For a function of n variables $f(x_1, x_2, \ldots, x_n)$:

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}$$

The symbol ∇ is called "nabla" or "del." The gradient ∇f is a **vector**, not a scalar.

Example: Computing the Gradient

For $f(x, y) = x^2 + 3xy + y^2$:

$$\nabla f = \begin{pmatrix} 2x + 3y \\ 3x + 2y \end{pmatrix}$$

At the point $(2, 1)$:

$$\nabla f(2, 1) = \begin{pmatrix} 2(2) + 3(1) \\ 3(2) + 2(1) \end{pmatrix} = \begin{pmatrix} 7 \\ 8 \end{pmatrix}$$

Example: Gradient at Multiple Points

For $f(x, y) = x^2 + y^2$ (a paraboloid, like a bowl):

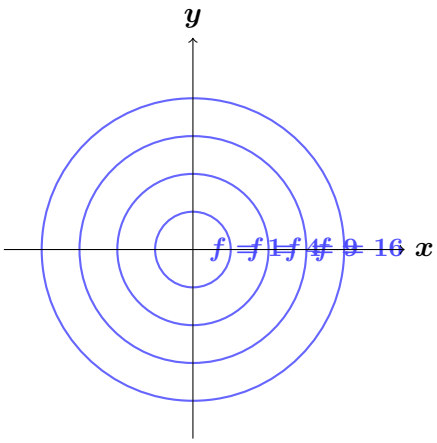
$$\nabla f = \begin{pmatrix} 2x \\ 2y \end{pmatrix}$$

Point (x, y)	Gradient ∇f	Interpretation
$(0, 0)$	$(0, 0)$	Flat (at the bottom of the bowl)
$(1, 0)$	$(2, 0)$	Points in $+x$ direction
$(0, 1)$	$(0, 2)$	Points in $+y$ direction
$(1, 1)$	$(2, 2)$	Points diagonally outward
$(-1, -1)$	$(-2, -2)$	Points toward bottom-left

D.6.2 Understanding Contour Lines

To understand the gradient geometrically, we first need to understand **contour lines** (also called level curves).

A contour line connects all points where the function has the same value. On a topographic map, contour lines connect points of equal elevation. On a weather map, they connect points of equal pressure or temperature.



Contour lines for $f(x, y) = x^2 + y^2$
(circles centered at origin, minimum at center)

Figure D.6: Contour lines connect points of equal function value. Here, the minimum is at the origin.

Key observations:

- Where contour lines are close together, the function is changing rapidly (steep slope)
- Where contour lines are far apart, the function is changing slowly (gentle slope)
- At a minimum or maximum, contour lines form closed loops around the point

D.6.3 The Gradient Points Uphill

The gradient has a beautiful geometric property: **the gradient points in the direction of steepest increase**.

If you stand on a hillside and want to climb as steeply as possible, walk in the direction of the gradient. The gradient is perpendicular to the contour lines and points toward higher values.

Conversely, if you want to *descend* as steeply as possible (which is what we want when minimizing a loss function), walk in the direction of $-\nabla f$.

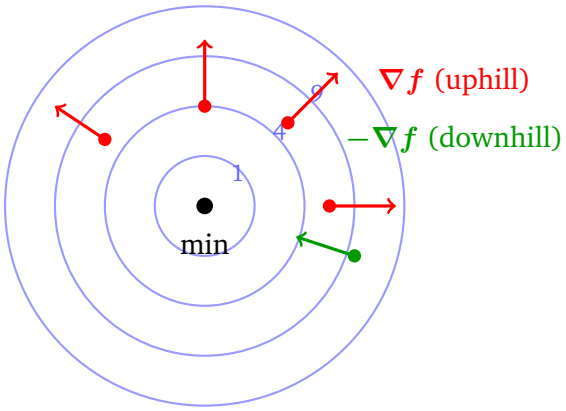


Figure D.7: The gradient vectors (red) point perpendicular to contour lines, toward increasing values. The negative gradient (green) points toward the minimum.

Why does the gradient point uphill? Consider standing on a hillside. The gradient collects the slopes in all coordinate directions. The combination of these slopes that gives the steepest overall ascent is exactly the gradient direction. Mathematically, the directional derivative (rate of change in any direction) is maximized when you move in the gradient direction.

D.7 Gradient Descent: Finding the Minimum

D.7.1 The Algorithm

Now we can state the central algorithm of machine learning optimization. To minimize a function $f(\theta)$ where θ represents our parameters:

1. **Initialize**: Start with some initial guess θ_0
2. **Compute gradient**: Calculate $\nabla f(\theta)$
3. **Update**: Take a step in the downhill direction:

$$\theta_{\text{new}} = \theta_{\text{old}} - \alpha \nabla f(\theta_{\text{old}})$$

4. **Repeat**: Go back to step 2 until the gradient is nearly zero

The parameter α is called the **learning rate**, it controls how big each step is.

D.7.2 A One-Variable Example

Let's minimize $f(\theta) = (\theta - 3)^2$ starting from $\theta_0 = 0$ with learning rate $\alpha = 0.1$.

The minimum is clearly at $\theta = 3$ (where the squared term is zero). Let's see gradient descent find it.

First, find the derivative: $f'(\theta) = 2(\theta - 3)$.

Iteration	θ	$f(\theta)$	$f'(\theta)$	$\theta_{\text{new}} = \theta - 0.1 \cdot f'(\theta)$
0	0	9	-6	$0 - 0.1(-6) = 0.6$
1	0.6	5.76	-4.8	$0.6 - 0.1(-4.8) = 1.08$
2	1.08	3.69	-3.84	$1.08 + 0.384 = 1.464$
3	1.464	2.36	-3.07	$1.464 + 0.307 = 1.771$
4	1.771	1.51	-2.46	$1.771 + 0.246 = 2.017$
5	2.017	0.97	-1.97	$2.017 + 0.197 = 2.214$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
∞	3	0	0	3

The sequence $0 \to 0.6 \to 1.08 \to 1.464 \to \ldots$ converges toward 3.

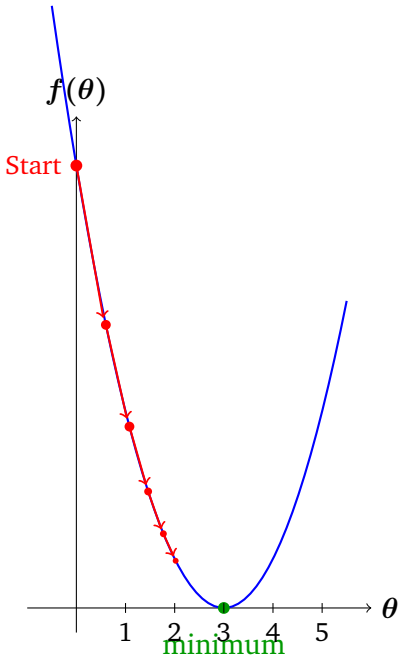


Figure D.8: Gradient descent taking steps toward the minimum.

D.7.3 A Two-Variable Example

Let's minimize $f(x, y) = x^2 + y^2$ starting from $(x_0, y_0) = (4, 3)$ with $\alpha = 0.1$.

The gradient is $\nabla f = (2x, 2y)$.

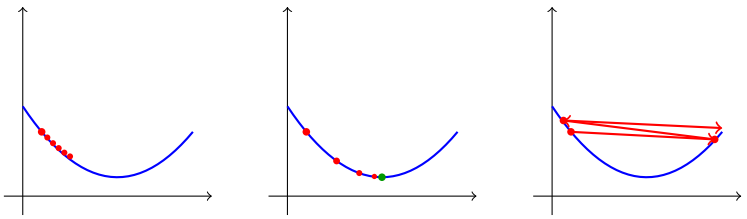
Iter	x	y	$f(x, y)$	∇f	$(x, y)_{\text{new}}$
0	4	3	25	$(8, 6)$	$(4 - 0.8, 3 - 0.6) = (3.2, 2.4)$
1	3.2	2.4	16	$(6.4, 4.8)$	$(2.56, 1.92)$
2	2.56	1.92	10.24	$(5.12, 3.84)$	$(2.05, 1.54)$
3	2.05	1.54	6.55	$(4.1, 3.08)$	$(1.64, 1.23)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
∞	0	0	0	$(0, 0)$	$(0, 0)$

The path spirals inward toward the minimum at the origin.

D.7.4 The Learning Rate Matters

The learning rate α is crucial for successful optimization:

- **Too small**: Each step is tiny, so convergence is very slow. You might need thousands of iterations.
- **Too large**: Steps overshoot the minimum, bouncing back and forth. The algorithm may never converge or even diverge to infinity.
- **Just right**: The algorithm converges efficiently in a reasonable number of steps.



$\alpha = 0.01$ (too small) $\alpha = 0.1$ (good) $\alpha = 0.9$ (too large)

Figure D.9: The effect of learning rate on gradient descent convergence.

D.8 Critical Points and Optimization

D.8.1 What are Critical Points?

A **critical point** is where the gradient equals zero: $\nabla f = \mathbf{0}$.

At a critical point, the function is "flat" in all directions, there is no direction of increase or decrease. Gradient descent will stop at a critical point (since the update $-\alpha \nabla f$ becomes zero).

D.8.2 Types of Critical Points

Not all critical points are minima! Critical points come in three types:

- **Local minimum**: The function increases in all directions from this point. This is a valley bottom.
- **Local maximum**: The function decreases in all directions. This is a hilltop.
- **Saddle point**: The function increases in some directions and decreases in others. Like a mountain pass or a horse saddle.

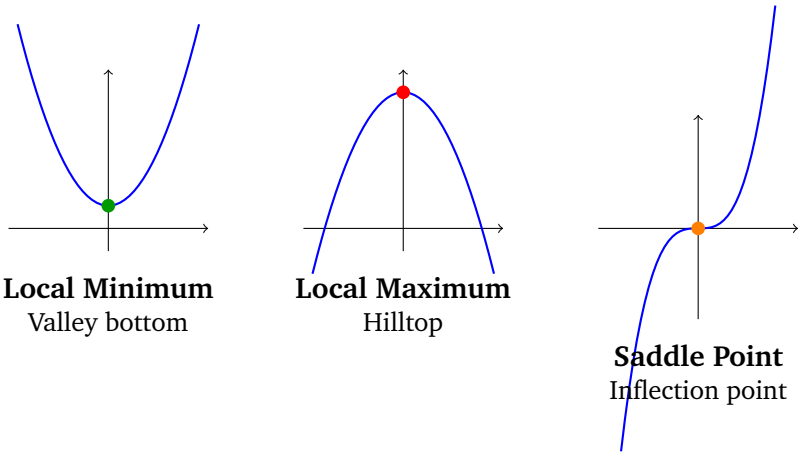


Figure D.10: Three types of critical points. At all three, $f'(x) = 0$.

D.8.3 Finding Critical Points

To find critical points:

1. Compute the derivative (or gradient)
2. Set it equal to zero
3. Solve for x (or $\mathbf{x}$)

Example: Finding Critical Points

Find the critical points of $f(x) = x^3 - 3x^2 - 9x + 5$.
Step 1: Compute the derivative.

$$f'(x) = 3x^2 - 6x - 9$$

Step 2: Set equal to zero.

$$3x^2 - 6x - 9 = 0$$

Step 3: Solve. Divide by 3:

$$x^2 - 2x - 3 = 0$$

Factor:

$$(x - 3)(x + 1) = 0$$

Solutions: $x = 3$ and $x = -1$.
The function has critical points at $x = -1$ and $x = 3$.

D.8.4 The Second Derivative Test

Once we find critical points, how do we classify them? The **second derivative** helps.

The second derivative $f''(x)$ is the derivative of the derivative. It measures how the slope itself is changing:

- If $f''(x) > 0$: the slope is increasing, so the curve bends upward (concave up, like a smile)
- If $f''(x) < 0$: the slope is decreasing, so the curve bends downward (concave down, like a frown)

At a critical point where $f'(x) = 0$:

- $f''(x) > 0 \implies$ local minimum (smiling curve $\smile$ has minimum at bottom)
- $f''(x) < 0 \implies$ local maximum (frowning curve $\frown$ has maximum at top)
- $f''(x) = 0 \implies$ test is inconclusive; need other methods

Example: Classifying Critical Points

For $f(x) = x^3 - 3x^2 - 9x + 5$, we found critical points at $x = -1$ and $x = 3$.
Compute the second derivative:

$$f'(x) = 3x^2 - 6x - 9 \implies f''(x) = 6x - 6$$

At $x = -1$:

$$f''(-1) = 6(-1) - 6 = -12 < 0 \implies \text{local maximum}$$

At $x = 3$:

$$f''(3) = 6(3) - 6 = 12 > 0 \implies \text{local minimum}$$

D.8.5 Global vs Local Optima

A **local minimum** is a point that is lower than all nearby points, but there might be even lower points elsewhere.

A **global minimum** is the lowest point of the entire function.

Gradient descent finds local minima, not necessarily global minima. In high-dimensional optimization (like training neural networks), there may be many local minima. Fortunately, research suggests that in many machine learning problems, most local minima are nearly as good as the global minimum.

For simple models like linear regression (Chapter 4), the loss function is convex: there is exactly one minimum, and gradient descent is guaranteed to find it. But in deep learning (Chapter 14), loss surfaces are **non-convex**: they contain many local minima, saddle points, and flat regions. This is one reason training deep networks is more art than science. The choice of learning rate, initialization, and optimization algorithm (Adam, SGD with momentum, and others) all affect which local minimum gradient descent settles into. The theory in this appendix gives you the foundations; the practice requires experimentation.

D.9 Important Functions and Their Derivatives

D.9.1 The Sigmoid Function

The **sigmoid function** maps any real number to a value between 0 and 1:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

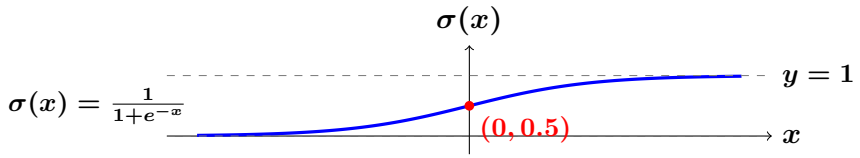


Figure D.11: The sigmoid function smoothly transitions from 0 to 1.

The sigmoid is used in logistic regression and as an activation function in neural networks. Its derivative has an elegant form:

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$$

This is computationally convenient: once you compute $\sigma(x)$, you get $\sigma'(x)$ with just one subtraction and one multiplication.

D.9.2 The ReLU Function

The **ReLU** (Rectified Linear Unit) is the most common activation function in modern neural networks:

$$\text{ReLU}(x) = \max(0, x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

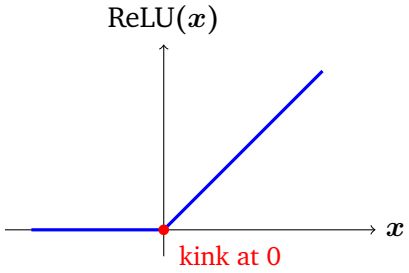


Figure D.12: The ReLU function: zero for negative inputs, identity for positive inputs.

Its derivative is:

$$\text{ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x < 0 \end{cases}$$

At $x = 0$, the derivative is technically undefined (there is a "kink"), but in practice, we usually define it as 0 or 1.

D.9.3 Loss Function Derivatives

Mean Squared Error (MSE):

For a single prediction $\hat{y}$ with true value y:

$$\mathcal{L} = (y - \hat{y})^2$$

The derivative with respect to the prediction:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = 2(\hat{y} - y)$$

Interpretation: If $\hat{y} > y$ (prediction too high), the gradient is positive, telling us to decrease $\hat{y}$. If $\hat{y} < y$ (prediction too low), the gradient is negative, telling us to increase $\hat{y}$.

For multiple predictions, with predictions $\hat{y}_1, \ldots, \hat{y}_n$ and true values $y_1, \ldots, y_n$:

$$\mathcal{L} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

$$\frac{\partial \mathcal{L}}{\partial \hat{y}_i} = \frac{2}{n}(\hat{y}_i - y_i)$$

D.10 Summary

Calculus Quick Reference

Derivative: Measures slope / rate of change.
- $f'(x) > 0$: function increasing
- $f'(x) < 0$: function decreasing
- $f'(x) = 0$: critical point

Derivative Rules:
- Power: $(x^n)' = nx^{n-1}$
- Constant multiple: $(cf)' = cf'$
- Sum: $(f + g)' = f' + g'$
- Chain: $(f(g(x)))' = f'(g(x)) \cdot g'(x)$
- Product: $(fg)' = f'g + fg'$
- Quotient: $(f/g)' = (f'g - fg')/g^2$

Special derivatives: $(e^x)' = e^x$, $(\ln x)' = 1/x$

Partial Derivative: $\frac{\partial f}{\partial x}$ = derivative treating other variables as constants

Gradient: ∇f = vector of all partial derivatives; points in direction of steepest increase

Gradient Descent: $\theta_{\text{new}} = \theta_{\text{old}} - \alpha \nabla f$
- α = learning rate
- Converges to critical points (hopefully minima)

Second Derivative Test: At critical point where $f'(x) = 0$:
- $f''(x) > 0$: local minimum
- $f''(x) < 0$: local maximum

Appendix E

Logic

> Logic is the beginning of wisdom,
> not the end.
>
> Leonard Nimoy as Spock

E.1 Introduction: Why Logic?

Logic is the study of valid reasoning. It provides rules for determining when a conclusion follows necessarily from given premises.

In artificial intelligence, logic plays several important roles. Historically, the first AI systems were built entirely on logical reasoning, programs that manipulated symbols according to logical rules. Today, logic remains essential: it underlies the mathematics we use to prove that algorithms work correctly, it provides the foundation for database query languages, and it connects directly to how decision trees and rule-based systems make classifications.

Understanding logic helps you think precisely, a crucial skill when designing algorithms, proving their correctness, or analyzing their limitations. The logical operators AND, OR, and NOT appear everywhere in computer science, from if-statements in code to the conditions that define classification boundaries.

This appendix covers propositional logic, truth tables, logical equivalences, quantifiers, and connections to artificial intelligence.

E.2 Propositions and Truth Values

E.2.1 What is a Proposition?

A **proposition** is a statement that is either true or false, but not both. We typically use letters like P, Q, R to represent propositions.

Examples of propositions:

- "It is raining" (true or false depending on the weather)
- "$2 + 2 = 4$" (true)
- "Paris is the capital of Germany" (false)
- "The model's accuracy is above 90%" (true or false depending on the model)

Not propositions:

- "What time is it?" (a question, not a statement)
- "Close the door" (a command)
- "$x > 5$" (depends on x; this is a *predicate*, not a proposition)

E.2.2 Truth Values

Every proposition has a **truth value**: either **True** (T, 1) or **False** (F, 0). Logic is about how truth values combine when we connect propositions with words like "and," "or," and "not."

E.3 Logical Connectives

E.3.1 NOT (Negation)

The **negation** of P, written $\neg P$ or $\bar{P}$ or NOT P, simply flips the truth value: true becomes false, false becomes true.

P	$\neg P$
T	F
F	T

Example: If P = "It is raining," then $\neg P$ = "It is not raining."

E.3.2 AND (Conjunction)

The **conjunction** of P and Q, written $P \wedge Q$, is true only when **both** P and Q are true.

P	Q	$P \wedge Q$
T	T	T
T	F	F
F	T	F
F	F	F

Example: "It is raining AND I have an umbrella" is true only if both conditions hold.

E.3.3 OR (Disjunction)

The **disjunction** of P and Q, written $P \vee Q$, is true when **at least one** of P or Q is true.

P	Q	$P \vee Q$
T	T	T
T	F	T
F	T	T
F	F	F

Note: This is **inclusive or**, "or both" is allowed. This differs from everyday English where "or" sometimes means "one or the other but not both" (like "Would you like tea or coffee?").

E.3.4 XOR (Exclusive Or)

The **exclusive or**, written $P \oplus Q$, is true when **exactly one** of P or Q is true, but not both.

P	Q	$P \oplus Q$
T	T	F
T	F	T
F	T	T
F	F	F

E.3.5 Connection to Sets

Logical connectives correspond directly to set operations:

Logic	Sets	Meaning
$\neg P$	A^c or $\bar{A}$	Complement (not in A)
$P \wedge Q$	$A \cap B$	Intersection (in both)
$P \vee Q$	$A \cup B$	Union (in at least one)

This correspondence is not a coincidence. If we think of P as "element is in set A," then the logical operations on P correspond exactly to set operations

on $\boldsymbol{A}$.

E.4 Implication and Equivalence

E.4.1 Implication (If-Then)

The **implication** $\boldsymbol{P \to Q}$, read "if $\boldsymbol{P}$ then $\boldsymbol{Q}$" or "$\boldsymbol{P}$ implies $\boldsymbol{Q}$," is fundamental to logical reasoning.

$\boldsymbol{P}$	$\boldsymbol{Q}$	$\boldsymbol{P \to Q}$
T	T	T
T	F	F
F	T	T
F	F	T

The truth table may seem counterintuitive, especially the last two rows. The key insight: $\boldsymbol{P \to Q}$ is only false when $\boldsymbol{P}$ is true but $\boldsymbol{Q}$ is false. When the "if" part holds but the "then" part fails.

Understanding Implication

Consider the promise: "If it rains, I will bring an umbrella."

- It rains, I bring umbrella: promise kept (true)
- It rains, I don't bring umbrella: promise broken (false)
- It doesn't rain, I bring umbrella: promise not violated (true)
- It doesn't rain, I don't bring umbrella: promise not violated (true)

The promise only specifies what happens if it rains. If it does not rain, you have not broken your promise regardless of what you do.

E.4.2 Biconditional (If and Only If)

The **biconditional** $\boldsymbol{P \leftrightarrow Q}$, read "$\boldsymbol{P}$ if and only if $\boldsymbol{Q}$" (often abbreviated "$\boldsymbol{P}$ iff $\boldsymbol{Q}$"), is true when $\boldsymbol{P}$ and $\boldsymbol{Q}$ have the same truth value.

$\boldsymbol{P}$	$\boldsymbol{Q}$	$\boldsymbol{P \leftrightarrow Q}$
T	T	T
T	F	F
F	T	F
F	F	T

$\boldsymbol{P \leftrightarrow Q}$ means both $\boldsymbol{P \to Q}$ and $\boldsymbol{Q \to P}$. It establishes that $\boldsymbol{P}$ and $\boldsymbol{Q}$ are logically equivalent. Whenever one is true, the other must be true, and whenever one is false, the other must be false.

Example: "A triangle is equilateral if and only if all its angles are 60°." This

means: if it's equilateral, all angles are 60°, AND if all angles are 60°, it's equilateral.

E.4.3 Logical Equivalence

Two expressions are **logically equivalent**, written $\equiv$, if they have the same truth value for all possible inputs.

For example, $P \to Q \equiv \neg P \lor Q$. We can verify this with a truth table:

P	Q	$P \to Q$	$\neg P$	$\neg P \lor Q$
T	T	T	F	T
T	F	F	F	F
F	T	T	T	T
F	F	T	T	T

The columns for $P \to Q$ and $\neg P \lor Q$ are identical, confirming the equivalence.

This equivalence is practically useful: it lets us rewrite any implication using only OR and NOT.

E.5 Important Logical Equivalences

E.5.1 Laws You Already Know from Arithmetic

Many logical equivalences mirror laws you already know from arithmetic. This is not a coincidence. AND behaves somewhat like multiplication, and OR behaves somewhat like addition.

Commutativity: Order doesn't matter.

$$\text{Numbers: } 3 + 5 = 5 + 3 \quad \text{and} \quad 3 \times 5 = 5 \times 3$$
$$\text{Logic: } P \lor Q \equiv Q \lor P \quad \text{and} \quad P \land Q \equiv Q \land P$$

Associativity: Grouping doesn't matter.

$$\text{Numbers: } (2 + 3) + 4 = 2 + (3 + 4)$$
$$\text{Logic: } (P \lor Q) \lor R \equiv P \lor (Q \lor R)$$

Distributivity: This is where it gets interesting!

With numbers, multiplication distributes over addition:

$$3 \times (4 + 5) = (3 \times 4) + (3 \times 5) = 12 + 15 = 27$$

In logic, AND distributes over OR (just like multiplication over addition):

$$P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$$

But here's something that does NOT work with regular numbers: addition does not distribute over multiplication. $3 + (4 \times 5) \neq (3 + 4) \times (3 + 5)$.

However, in logic, OR DOES distribute over AND:

$$P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$$

This is a key difference between logic and arithmetic!

E.5.2 Identity and Domination

With numbers, 0 is the identity for addition ($x + 0 = x$) and 1 is the identity for multiplication ($x \times 1 = x$).

In logic:

$$P \vee \text{False} \equiv P \quad \text{(False is the identity for OR)}$$
$$P \wedge \text{True} \equiv P \quad \text{(True is the identity for AND)}$$

And just as $x \times 0 = 0$ regardless of x:

$$P \wedge \text{False} \equiv \text{False} \quad \text{(False dominates AND)}$$
$$P \vee \text{True} \equiv \text{True} \quad \text{(True dominates OR)}$$

E.5.3 Negation and Double Negation

Double Negation: $\neg(\neg P) \equiv P$

Just as $-(-5) = 5$ with numbers, negating twice returns you to the original.

Negation with AND/OR: What happens when we negate a combination? This leads us to De Morgan's Laws.

E.5.4 De Morgan's Laws

De Morgan's laws tell us how negation interacts with AND and OR:

$$\neg(P \wedge Q) \equiv \neg P \vee \neg Q$$
$$\neg(P \vee Q) \equiv \neg P \wedge \neg Q$$

Notice the pattern: when you push a NOT through an AND or OR, the AND becomes OR (and vice versa).

Example: De Morgan's Laws in Plain English

"It is NOT the case that I am both hungry AND tired"
is equivalent to
"I am NOT hungry OR I am NOT tired" (or both)

"It is NOT the case that I am hungry OR tired"
is equivalent to
"I am NOT hungry AND I am NOT tired"

E.5.5 Contrapositive

One of the most useful equivalences for reasoning:

$$P \to Q \equiv \neg Q \to \neg P$$

"If it rains, the ground is wet" is logically equivalent to "If the ground is not wet, it did not rain."

The contrapositive is often easier to prove or check than the original implication.

E.6 Quantifiers

So far, our propositions have been simple statements. But often we want to make claims about *all* things of some type, or about the *existence* of something. Quantifiers let us do this.

E.6.1 Universal Quantifier ($\forall$)

The **universal quantifier** $\forall$ means "for all" or "for every."

$\forall x : P(x)$ means "$P(x)$ is true for every x."

Examples: Universal Quantifier

$\forall x \in \mathbb{R} : x^2 \geq 0$
"For all real numbers x, x^2 is non-negative." (True!)

$\forall n \in \mathbb{N} : n + 1 > n$
"For all natural numbers n, $n + 1$ is greater than n." (True!)

$\forall x \in \mathbb{R} : x > 0$
"All real numbers are positive." (False, what about -5?)

E.6.2 Existential Quantifier (∃)

The **existential quantifier** ∃ means "there exists" or "for some."

$\exists x : P(x)$ means "there is at least one x for which $P(x)$ is true."

Examples: Existential Quantifier

$\exists x \in \mathbb{R} : x^2 = 2$
"There exists a real number whose square is 2." (True, it's $\sqrt{2}$)

$\exists x \in \mathbb{R} : x^2 = -1$
"There exists a real number whose square is -1." (False, no real number works)

$\exists n \in \mathbb{Z} : n^2 = 2$
"There exists an integer whose square is 2." (False. $\sqrt{2}$ is not an integer)

E.6.3 Negating Quantifiers

Negating a universal statement gives an existential statement, and vice versa:

$$\neg(\forall x : P(x)) \equiv \exists x : \neg P(x)$$

$$\neg(\exists x : P(x)) \equiv \forall x : \neg P(x)$$

Examples: Negating Quantifiers

Original: "All birds can fly." (∀ bird b: b can fly)
Negation: "There exists a bird that cannot fly." (∃ bird b: b cannot fly)
(And indeed, penguins and ostriches exist!)

Original: "There exists an integer whose square is 2." ($\exists n \in \mathbb{Z} : n^2 = 2$)
Negation: "For all integers, their square is not 2." ($\forall n \in \mathbb{Z} : n^2 \neq 2$)
(This negation is true. No integer squares to 2.)

E.7 First-Order Logic

Propositional logic is powerful but limited. It treats propositions as atomic units, we cannot look inside them or reason about their structure. **First-Order Logic** (FOL), also called predicate logic, extends propositional logic to reason about objects, their properties, and relationships between them.

E.7.1 Predicates and Terms

A **predicate** is a function that takes objects and returns a truth value. Instead of atomic propositions like P, we have structured expressions like Mortal(Socrates) meaning "Socrates is mortal."

Terms represent objects:

- **Constants**: Specific objects like Socrates, Athens, 5
- **Variables**: Placeholders like x, y, z
- **Functions**: Expressions that compute objects, like father(x) (the father of x)

Predicates express properties or relationships:

- Human(x) , "x is human" (unary predicate)
- Loves(x, y) , "x loves y" (binary predicate)
- Between(x, y, z) , "x is between y and z" (ternary predicate)

E.7.2 Sentences in First-Order Logic

FOL sentences combine predicates with logical connectives and quantifiers:

Examples: First-Order Logic Sentences

"All humans are mortal":

$$\forall x : \text{Human}(x) \rightarrow \text{Mortal}(x)$$

"Socrates is human":

$$\text{Human}(\text{Socrates})$$

"Someone loves everyone":

$$\exists x : \forall y : \text{Loves}(x, y)$$

"Everyone loves someone":

$$\forall x : \exists y : \text{Loves}(x, y)$$

Notice that the last two examples mean different things! "Someone loves everyone" means there is a specific person who loves all people. "Everyone loves someone" means each person has at least one person they love (possibly different for each person). The order of quantifiers matters.

E.7.3 Scope of Quantifiers

The **scope** of a quantifier is the part of the formula where the quantified variable is bound. Consider:

$$\forall x : (P(x) \land \exists y : Q(x, y))$$

The scope of $\forall x$ is the entire expression $(P(x) \land \exists y : Q(x, y))$. The scope of $\exists y$ is only $Q(x, y)$.

Variables can be **free** (not bound by any quantifier) or **bound**. A sentence with no free variables is called a **closed formula** and has a definite truth value.

E.7.4 Inference in First-Order Logic

The classic example of FOL inference is:

1. All humans are mortal: $\forall x : \text{Human}(x) \rightarrow \text{Mortal}(x)$
2. Socrates is human: Human(Socrates)
3. Therefore, Socrates is mortal: Mortal(Socrates)

This uses **Universal Instantiation**: if $\forall x : P(x)$ is true, then $P(c)$ is true for any constant c.

E.8 Unification

Unification is a key algorithm in automated reasoning that finds substitutions making two expressions identical. It is essential for inference in first-order logic.

E.8.1 The Unification Problem

Given two expressions, find a **substitution** (mapping of variables to terms) that makes them identical.

Examples: Unification

Example 1: Unify $P(x, y)$ and $P(A, B)$
Substitution: $\{x/A, y/B\}$ (replace x with A, y with B)
Result: Both become $P(A, B)$ ✓

Example 2: Unify $P(x, x)$ and $P(A, B)$
Need $x = A$ and $x = B$, but $A \neq B$.
Result: Cannot unify ×

Example 3: Unify $P(x, f(y))$ and $P(A, f(B))$
Substitution: $\{x/A, y/B\}$
Result: Both become $P(A, f(B))$ ✓

Example 4: Unify $P(x, f(x))$ and $P(y, y)$
Need $f(x) = y$ and $x = y$, so $f(x) = x$.
Result: Cannot unify (occurs check, x cannot equal $f(x)$) ×

E.8.2 Most General Unifier (MGU)

The **Most General Unifier** is the simplest substitution that unifies two expressions. Any other unifier can be obtained by further substituting into the MGU.

For $P(x)$ and $P(y)$:

- $\{x/y\}$ is an MGU (could also use $\{y/x\}$)
- $\{x/A, y/A\}$ is also a unifier, but it's more specific than the MGU

E.8.3 The Unification Algorithm

Unification Algorithm (Simplified)

Unify(E_1, E_2):
1. If $E_1 = E_2$, return empty substitution $\{\}$
2. If E_1 is a variable x:
   - If x occurs in E_2, fail (occurs check)
   - Otherwise, return $\{x/E_2\}$
3. If E_2 is a variable, swap and apply step 2
4. If both are compound (e.g., $f(a_1, \ldots, a_n)$ and $f(b_1, \ldots, b_n)$):
   - If function symbols differ, fail
   - Recursively unify arguments, composing substitutions
5. Otherwise, fail

E.8.4 Unification in AI

Unification is used in:

- **Resolution theorem proving**: Finding complementary literals to resolve
- **Logic programming** (Prolog): Matching goals with rules
- **Type inference**: Determining types in programming languages
- **Natural language processing**: Parsing and semantic analysis

E.9 Logical Paradoxes

Paradoxes reveal subtle issues in logical reasoning and remind us to be careful with self-reference and unrestricted set formation.

E.9.1 The Liar Paradox

Consider the statement: "This sentence is false."

If it is true, then what it says must hold, so it is false. But if it is false, then what it says does not hold, so it is not false, meaning it is true. We have a contradiction either way.

This paradox arises from self-reference and shows that not every grammatical sentence is a valid proposition.

E.9.2 Russell's Paradox

Consider the set $\boldsymbol{R}$ of all sets that do not contain themselves: $\boldsymbol{R} = \{\boldsymbol{S} : \boldsymbol{S} \notin \boldsymbol{S}\}$.

Does $\boldsymbol{R}$ contain itself?

- If $\boldsymbol{R} \in \boldsymbol{R}$, then by definition of $\boldsymbol{R}$, we need $\boldsymbol{R} \notin \boldsymbol{R}$. Contradiction.
- If $\boldsymbol{R} \notin \boldsymbol{R}$, then by definition of $\boldsymbol{R}$, we need $\boldsymbol{R} \in \boldsymbol{R}$. Contradiction.

This paradox motivated the development of more careful foundations for set theory, where not every description defines a valid set.

E.9.3 The Barber Paradox

A barber shaves all those, and only those, who do not shave themselves. Does the barber shave himself?

This is Russell's paradox in disguise. The resolution: no such barber can exist, the description is self-contradictory.

E.10 Logic and Artificial Intelligence

E.10.1 The Historical Role of Logic in AI

The earliest AI systems, developed in the 1950s and 1960s, were built on symbolic logic. The idea was compelling: if we could represent knowledge as logical statements and give computers rules for logical inference, they could reason like humans.

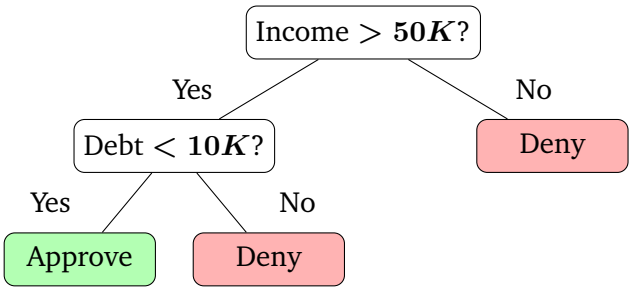
Programs like the General Problem Solver (1959) and expert systems (1970s-1980s) used logical rules to solve problems. A medical diagnosis system might contain rules like:

```
IF fever AND cough AND fatigue THEN possibly_flu
IF possibly_flu AND NOT vaccinated THEN recommend_test
```

These are logical implications translated into code!

E.10.2 Decision Trees as Logical Formulas

Every decision tree can be written as a logical formula. Consider a tree for loan approval:



This tree corresponds to the logical formula:

$$\text{Approve} \equiv (\text{Income} > 50K) \wedge (\text{Debt} < 10K)$$

More complex trees create more complex formulas with nested ANDs and ORs, but every decision tree is equivalent to some logical expression. Chapter 8 develops this connection further: the constraint satisfaction problems and SAT solvers discussed there are direct applications of the propositional logic and CNF representations covered in this appendix.

E.10.3 Logic in Modern AI

While modern machine learning often uses statistical and neural approaches rather than explicit logic, logical concepts remain important:

Knowledge graphs and reasoning: Systems like those behind search engines and question-answering use logical inference over structured knowledge.

Constraints and verification: When we need to verify that an AI system behaves correctly, we often express requirements as logical formulas and prove (or test) that the system satisfies them.

Interpretable AI: There is growing interest in AI systems that can explain their decisions. Rule-based explanations are essentially logical formulas: "I classified this as spam because it contains suspicious links AND comes from an unknown sender AND has urgent language."

SAT solvers: The Boolean satisfiability problem (SAT), determining whether a logical formula can be made true, is a fundamental problem in computer science. Modern SAT solvers are remarkably powerful and are used in hardware verification, software testing, and even some AI planning systems.

E.10.4 The Limits of Pure Logic

Early AI researchers hoped that logic alone could achieve human-level intelligence. This dream encountered significant obstacles:

The knowledge acquisition bottleneck: Writing down all the logical rules needed for common-sense reasoning proved nearly impossible. How do you write rules capturing everything a child knows about how the physical world works?

Uncertainty: The real world is uncertain, but classical logic deals only with definite true/false values. This led to probabilistic extensions and eventually to statistical machine learning.

Learning: Logic-based systems had to be programmed with rules by humans. They could not learn from data the way neural networks can.

Modern AI combines insights from logic (precise reasoning, compositionality) with statistical learning (handling uncertainty, learning from data). The most powerful systems often use both.

E.11 Building Truth Tables

For complex expressions, build the truth table systematically, computing intermediate columns.

Example: Building a Truth Table

Verify that $(P \land Q) \to R \equiv P \to (Q \to R)$:

P	Q	R	$P \land Q$	$(P \land Q) \to R$	$Q \to R$	$P \to (Q \to R)$	Same?
T	T	T	T	T	T	T	✓
T	T	F	T	F	F	F	✓
T	F	T	F	T	T	T	✓
T	F	F	F	T	T	T	✓
F	T	T	F	T	T	T	✓
F	T	F	F	T	F	T	✓
F	F	T	F	T	T	T	✓
F	F	F	F	T	T	T	✓

The two main columns are identical, confirming the equivalence.

E.12 Conjunctive Normal Form (CNF)

Many automated reasoning systems require formulas in a standard format called **Conjunctive Normal Form** (CNF).

E.12.1 What is CNF?

A formula is in CNF if it is a conjunction (AND) of clauses, where each clause is a disjunction (OR) of literals. A **literal** is a variable or its negation.

$$\text{CNF: } (A \lor B \lor \neg C) \land (\neg A \lor D) \land (B \lor C)$$

Here we have three clauses, connected by AND. Each clause contains literals connected by OR.

E.12.2 Converting to CNF

Any propositional formula can be converted to an equivalent CNF formula:

Step 1: Eliminate implications and biconditionals.

- $P \to Q$ becomes $\neg P \lor Q$
- $P \leftrightarrow Q$ becomes $(P \to Q) \land (Q \to P)$, then apply the above

Step 2: Push negations inward using De Morgan's laws until they apply only to variables.

- $\neg(P \land Q)$ becomes $\neg P \lor \neg Q$
- $\neg(P \lor Q)$ becomes $\neg P \land \neg Q$

- $\neg\neg P$ becomes P

Step 3: Distribute OR over AND to get a conjunction of disjunctions.

- $P \vee (Q \wedge R)$ becomes $(P \vee Q) \wedge (P \vee R)$

Example: Converting to CNF

Convert $(P \to Q) \to R$ to CNF.
Step 1: Eliminate implications.

$$(P \to Q) \to R = \neg(P \to Q) \vee R = \neg(\neg P \vee Q) \vee R$$

Step 2: Push negation inward.

$$\neg(\neg P \vee Q) \vee R = (P \wedge \neg Q) \vee R$$

Step 3: Distribute OR over AND.

$$(P \wedge \neg Q) \vee R = (P \vee R) \wedge (\neg Q \vee R)$$

Result: $(P \vee R) \wedge (\neg Q \vee R)$, two clauses in CNF.

E.13 Resolution

Resolution is a powerful inference rule for proving logical statements. It operates on formulas in CNF.

E.13.1 The Resolution Rule

If we have two clauses, one containing a literal L and another containing $\neg L$, we can **resolve** them to produce a new clause containing all literals except L and $\neg L$:

$$\frac{(A \vee L) \quad (\neg L \vee B)}{(A \vee B)}$$

The literals L and $\neg L$ cancel out, and we combine the remaining literals.

Examples: Resolution

Example 1:

$$\frac{(P \vee Q) \quad (\neg Q \vee R)}{(P \vee R)}$$

Q and $\neg Q$ cancel; we keep P and R.

Example 2:

$$\frac{(\neg A \vee B \vee C) \quad (A \vee D)}{(B \vee C \vee D)}$$

A and $\neg A$ cancel; we keep B, C, and D.

Example 3:

$$\frac{(P) \quad (\neg P)}{()}$$

Everything cancels! The empty clause () represents a contradiction (False).

E.13.2 Proof by Refutation

To prove that a statement ϕ follows from premises $\psi_1, \psi_2, \ldots$, we use **proof by refutation**:

1. Convert all premises to CNF
2. Add the *negation* of what we want to prove (also in CNF)
3. Apply resolution repeatedly
4. If we derive the empty clause (contradiction), the original statement is proven

The logic: if assuming $\neg\phi$ together with the premises leads to contradiction, then ϕ must be true.

Example: Resolution Proof

Prove: From $P \to Q$ and $Q \to R$, we can derive $P \to R$.
Step 1: Convert premises to CNF.
- $P \to Q$ becomes $\neg P \vee Q$
- $Q \to R$ becomes $\neg Q \vee R$

Step 2: Add negation of goal.
- We want $P \to R$, so add $\neg(P \to R) = \neg(\neg P \vee R) = P \wedge \neg R$
- This gives two clauses: (P) and $(\neg R)$

Step 3: Our clauses are:
1. $\neg P \vee Q$ (from premise 1)
2. $\neg Q \vee R$ (from premise 2)
3. P (from negated goal)
4. $\neg R$ (from negated goal)

Step 4: Apply resolution.
- Resolve (1) and (3): $\frac{(\neg P \vee Q) \quad (P)}{(Q)}$, gives clause (5): Q
- Resolve (2) and (5): $\frac{(\neg Q \vee R) \quad (Q)}{(R)}$, gives clause (6): R
- Resolve (4) and (6): $\frac{(\neg R) \quad (R)}{()}$, empty clause!

We derived a contradiction, so $P \to R$ is proven.

E.14 SAT: Boolean Satisfiability

The **Boolean Satisfiability Problem** (SAT) asks: given a formula in CNF, is there an assignment of truth values to variables that makes the formula true?

E.14.1 SAT as Search

SAT can be viewed as a search problem:

- **State**: A partial assignment of truth values to variables
- **Actions**: Assign True or False to an unassigned variable
- **Goal**: A complete assignment satisfying all clauses

This connects directly to the CSP framework from Chapter 7! Variables are the Boolean variables, domains are {True, False}, and constraints are the clauses.

E.14.2 Why SAT Matters

SAT was the first problem proven to be NP-complete, if you could solve SAT efficiently, you could solve any problem in NP efficiently. Despite this

worst-case hardness, modern SAT solvers can handle formulas with millions of variables.

SAT solvers are used in:

- **Hardware verification**: Proving that circuits work correctly
- **Software verification**: Finding bugs in programs
- **AI planning**: Finding sequences of actions to achieve goals
- **Cryptanalysis**: Breaking certain encryption schemes

The DPLL algorithm (and its modern extensions) combines backtracking search with unit propagation (if a clause has only one unassigned literal, that literal must be true) and pure literal elimination (if a variable appears with only one polarity, assign it to satisfy those clauses).

E.15 Summary

Logic Quick Reference

Connectives:
- NOT ($\neg$): Flips truth value
- AND ($\wedge$): True when both are true
- OR ($\vee$): True when at least one is true
- Implication ($\rightarrow$): False only when $\boldsymbol{P}$ true, $\boldsymbol{Q}$ false
- Biconditional ($\leftrightarrow$): True when both have same value

Key Equivalences:
- $\boldsymbol{P \rightarrow Q \equiv \neg P \vee Q}$
- $\neg(\boldsymbol{P \wedge Q}) \equiv \boldsymbol{\neg P \vee \neg Q}$ (De Morgan)
- $\neg(\boldsymbol{P \vee Q}) \equiv \boldsymbol{\neg P \wedge \neg Q}$ (De Morgan)
- $\boldsymbol{P \rightarrow Q \equiv \neg Q \rightarrow \neg P}$ (Contrapositive)

CNF: Conjunction of disjunctions: $(\boldsymbol{A \vee B}) \wedge (\boldsymbol{C \vee \neg D}) \wedge \ldots$
Resolution: From $(\boldsymbol{A \vee L})$ and $(\boldsymbol{\neg L \vee B})$, derive $(\boldsymbol{A \vee B})$.
SAT: Finding truth assignments that satisfy a CNF formula. Connects to search and CSPs.
Quantifiers:
- $\forall$ (for all): $\neg(\forall \boldsymbol{x} : \boldsymbol{P}(\boldsymbol{x})) \equiv \exists \boldsymbol{x} : \neg \boldsymbol{P}(\boldsymbol{x})$
- $\exists$ (there exists): $\neg(\exists \boldsymbol{x} : \boldsymbol{P}(\boldsymbol{x})) \equiv \forall \boldsymbol{x} : \neg \boldsymbol{P}(\boldsymbol{x})$

First-Order Logic:
- Predicates express properties and relations: Human($\boldsymbol{x}$), Loves($\boldsymbol{x}, \boldsymbol{y}$)
- Terms: constants, variables, and functions
- Quantifier order matters: $\forall \boldsymbol{x} \exists \boldsymbol{y}$ differs from $\exists \boldsymbol{y} \forall \boldsymbol{x}$

Unification: Finding substitutions that make two expressions identical. The MGU (Most General Unifier) is the simplest such substitution.
Logic-Set Correspondence: AND $\leftrightarrow$ Intersection, OR $\leftrightarrow$ Union, NOT $\leftrightarrow$ Complement
AI Connection: Decision trees are logical formulas; expert systems use logical rules; modern AI combines logic with statistical learning.

Appendix F

Linear Algebra

> The introduction of numbers as coordinates is an act of violence.
>
> Hermann Weyl

F.1 Introduction: Why Linear Algebra?

Machine learning is about finding patterns in data, and data almost always comes in collections. A patient is not described by a single number but by dozens of measurements: blood pressure, heart rate, cholesterol levels, age, weight. An image is not a single value but millions of pixel intensities. A document is characterized by the frequencies of thousands of different words.

To work with such collections efficiently, we need a mathematical framework that treats collections of numbers as single objects we can manipulate. This is what **linear algebra** provides.

Linear algebra gives us three things. First, **compact notation**: instead of writing $x_1, x_2, x_3, \ldots, x_{1000}$, we write a single symbol $\mathbf{x}$. Second, **efficient computation**: computers can multiply matrices billions of times faster than equivalent nested loops. Third, **geometric intuition**: we can visualize data as points in space, transformations as rotations and stretches, and learning as finding the right transformation.

Nearly every machine learning algorithm (from linear regression to neural networks to principal component analysis) is expressed and implemented using linear algebra. This appendix builds your understanding from the ground up.

F.2 Scalars: Single Numbers

A **scalar** is simply a single number. We use lowercase letters like a, b, x, y for scalars.

Examples of scalars: the temperature today ($22°$C), a student's exam score (85), a model's learning rate ($\alpha = 0.01$), the price of a house (€$450\,000$).

Scalars are the atoms from which everything else is built. When we move to vectors and matrices, we are simply organizing collections of scalars in useful ways.

F.3 Vectors: Ordered Lists of Numbers

F.3.1 What is a Vector?

A **vector** is an ordered list of numbers. We use bold lowercase letters like $\mathbf{x}$, $\mathbf{y}$, $\mathbf{w}$ for vectors, and we typically write them as vertical columns:

$$\mathbf{x} = \begin{pmatrix} 3 \\ 7 \\ 2 \end{pmatrix}$$

This vector has three **components**: the first component is $x_1 = 3$, the second is $x_2 = 7$, and the third is $x_3 = 2$. The **dimension** of a vector is the number of components it has, this vector is 3-dimensional.

F.3.2 Vectors in the Real World

Vectors naturally represent things with multiple attributes:

Example: A Patient as a Vector

A patient might be described by:

$$\mathbf{x}_{\text{patient}} = \begin{pmatrix} \text{age} \\ \text{weight (kg)} \\ \text{blood pressure} \\ \text{cholesterol} \end{pmatrix} = \begin{pmatrix} 45 \\ 72 \\ 130 \\ 195 \end{pmatrix}$$

Each patient becomes a single vector. A hospital database with 10,000 patients becomes 10,000 vectors.

In machine learning, we call the vector describing a single data point a **feature vector**, it contains all the features (attributes) of that example.

F.3.3 Geometric View: Vectors as Arrows

In 2D or 3D, we can visualize vectors as arrows. The vector $\mathbf{v} = \begin{pmatrix} 3 \\ 2 \end{pmatrix}$ is an arrow pointing 3 units in the x-direction and 2 units in the y-direction.

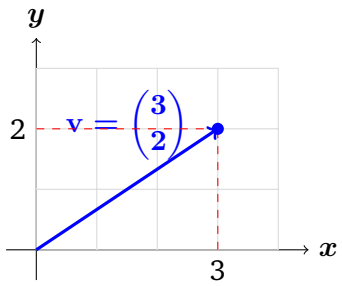


Figure F.1: A 2D vector visualized as an arrow from the origin to the point $(3, 2)$.

F.3.4 Vector Addition and Scalar Multiplication

To add two vectors, add their corresponding components:

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} + \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} = \begin{pmatrix} 5 \\ 7 \\ 9 \end{pmatrix}$$

To multiply a vector by a scalar, multiply each component:

$$3 \cdot \begin{pmatrix} 2 \\ 4 \\ 1 \end{pmatrix} = \begin{pmatrix} 6 \\ 12 \\ 3 \end{pmatrix}$$

Important: Scalar times vector gives a **vector** (same dimension as the original).

F.4 The Dot Product: Multiplying Two Vectors

F.4.1 Definition

The **dot product** takes two vectors of the same dimension and produces a single number, a scalar.

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{n} x_i y_i = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n$$

Example: Computing a Dot Product

$$\begin{pmatrix}1\\2\\3\end{pmatrix} \cdot \begin{pmatrix}4\\5\\6\end{pmatrix} = (1 \times 4) + (2 \times 5) + (3 \times 6) = 4 + 10 + 18 = 32$$

Two vectors in, one scalar out.

Critical point: The dot product of two vectors is a **scalar** (a single number), not a vector.

F.4.2 Why the Dot Product Matters

Real-world example: Shopping bill. Suppose prices are $\mathbf{p} = \begin{pmatrix}2.50\\1.00\\3.00\end{pmatrix}$ and quantities are $\mathbf{q} = \begin{pmatrix}2\\3\\1\end{pmatrix}$. Your total is:

$$\mathbf{p} \cdot \mathbf{q} = 2.50 \times 2 + 1.00 \times 3 + 3.00 \times 1 = €11.00$$

Machine learning: Linear prediction. A linear model computes $\mathbf{w} \cdot \mathbf{x}$, where $\mathbf{w}$ contains learned weights and $\mathbf{x}$ contains features. This is the foundation of linear regression, logistic regression, and each neuron in a neural network.

F.4.3 Geometric Meaning

The dot product relates to the angle θ between vectors: $\mathbf{x} \cdot \mathbf{y} = \|\mathbf{x}\|\|\mathbf{y}\| \cos\theta$. Vectors pointing the same direction have large positive dot products; perpendicular vectors have dot product zero; opposite vectors have negative dot products.

F.5 Vector Length (Norm)

The **norm** of a vector, written $\|\mathbf{x}\|$, measures its length:

$$\|\mathbf{x}\| = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2} = \sqrt{\mathbf{x} \cdot \mathbf{x}}$$

In 2D, this is the Pythagorean theorem:

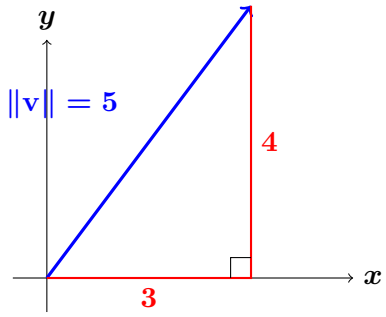


Figure F.2: For $\mathbf{v} = (3, 4)$: $\|\mathbf{v}\| = \sqrt{3^2 + 4^2} = \sqrt{25} = 5$.

F.6 Matrices: Tables of Numbers

F.6.1 What is a Matrix?

A **matrix** is a rectangular table of numbers. We use bold uppercase letters like $\mathbf{A}$, $\mathbf{W}$:

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

This is a "2×3 matrix" (2 rows, 3 columns). Element A_{ij} is in row i, column j: here $A_{12} = 2$.

In machine learning, a data matrix often has rows as examples and columns as features.

F.6.2 Special Matrices

Identity matrix $\mathbf{I}$: 1s on diagonal, 0s elsewhere. It is the matrix equivalent of 1.

$$\mathbf{I}_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Transpose $\mathbf{A}^T$: Swap rows and columns.

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}^T = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

F.7 Matrix Operations: Building Up Step by Step

F.7.1 Scalar Times Matrix

Multiply every element by the scalar:

$$3 \cdot \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 3 & 6 \\ 9 & 12 \end{pmatrix}$$

Result: A matrix of the same size.

F.7.2 Matrix Times Vector

Think of it as taking the dot product of each row with the vector:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \begin{pmatrix} 7 \\ 8 \end{pmatrix} = \begin{pmatrix} 1 \cdot 7 + 2 \cdot 8 \\ 3 \cdot 7 + 4 \cdot 8 \\ 5 \cdot 7 + 6 \cdot 8 \end{pmatrix} = \begin{pmatrix} 23 \\ 53 \\ 83 \end{pmatrix}$$

A (3×2) matrix times a (2×1) vector gives a (3×1) vector.

This is how we apply a linear model: predictions $= \mathbf{W}\mathbf{x}$.

F.7.3 Matrix Times Matrix

Each element C_{ij} is the dot product of row i of $\mathbf{A}$ with column j of $\mathbf{B}$.

Example: Matrix Multiplication

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 1 \cdot 5 + 2 \cdot 7 & 1 \cdot 6 + 2 \cdot 8 \\ 3 \cdot 5 + 4 \cdot 7 & 3 \cdot 6 + 4 \cdot 8 \end{pmatrix} = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

F.7.4 The Dimension Rule

To multiply $\mathbf{A}$ (size $m \times n$) by $\mathbf{B}$ (size $n \times p$): inner dimensions must match. Result is $m \times p$.

$$\underbrace{\mathbf{A}}_{m \times \mathrm{n}} \times \underbrace{\mathbf{B}}_{\mathrm{n} \times p} = \underbrace{\mathbf{C}}_{m \times p}$$

$\longleftrightarrow$ must match!

F.7.5 Commutativity: Order Matters!

With scalars, $3 \times 5 = 5 \times 3$. This property is called **commutativity**.

Matrix multiplication is **NOT commutative**: $\mathbf{AB} \neq \mathbf{BA}$ in general.

Example: Order Matters

$$\mathbf{A} = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$$\mathbf{AB} = \begin{pmatrix} 2 & 1 \\ 1 & 0 \end{pmatrix}, \quad \mathbf{BA} = \begin{pmatrix} 0 & 1 \\ 1 & 2 \end{pmatrix}$$

Different results! Sometimes $\mathbf{AB}$ exists but $\mathbf{BA}$ does not.

F.8 What Operations Return What?

Operation	Inputs	Output
Scalar × Scalar	two numbers	Scalar
Scalar × Vector	number, vector	Vector
Dot product	two vectors	**Scalar**
Vector norm	one vector	Scalar
Scalar × Matrix	number, matrix	Matrix
Matrix × Vector	$(m \times n)$, $(n \times 1)$	Vector $(m \times 1)$
Matrix × Matrix	$(m \times n)$, $(n \times p)$	Matrix $(m \times p)$

F.9 Matrix Inverse

The **inverse** of a square matrix $\mathbf{A}$, written $\mathbf{A}^{-1}$, satisfies:

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$$

Not all matrices have inverses. A matrix without an inverse is called **singular**.

For 2×2: if $\mathbf{A} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$, then $\mathbf{A}^{-1} = \frac{1}{ad-bc}\begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$.

The quantity $ad - bc$ is the **determinant**. If it equals zero, no inverse exists.

F.10 Eigenvalues and Eigenvectors

F.10.1 What Does a Matrix Do to Vectors?

Before understanding eigenvalues, let us think about what matrices actually *do*. When you multiply a matrix by a vector, you get a new vector. The matrix transforms the original vector into a different one.

Consider multiplying various vectors by the matrix $\mathbf{A} = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$:

$$\mathbf{A}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix} \qquad \mathbf{A}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \qquad \mathbf{A}\begin{pmatrix} 1 \\ -1 \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

Notice something interesting in the third example: the vector $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$ came out unchanged! The matrix did not rotate it or change its direction at all.

Let us try one more:

$$\mathbf{A}\begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 \\ 3 \end{pmatrix} = 3\begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Here the vector $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ came out pointing in the same direction, just 3 times longer!

These special vectors (the ones that do not get rotated, only stretched) are called **eigenvectors**.

F.10.2 The Definition

A vector $\mathbf{v}$ is an **eigenvector** of matrix $\mathbf{A}$ if multiplying by $\mathbf{A}$ only scales the vector, without changing its direction:

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$$

The scaling factor λ (lambda) is called the **eigenvalue**. It tells you how much the eigenvector gets stretched:

- $\lambda > 1$: the vector gets longer
- $\lambda = 1$: the vector stays the same length
- $0 < \lambda < 1$: the vector gets shorter
- $\lambda < 0$: the vector gets flipped (points opposite direction) and scaled

In our example above:

- $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$ is an eigenvector with eigenvalue $\lambda = 1$ (unchanged)
- $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ is an eigenvector with eigenvalue $\lambda = 3$ (stretched by factor 3)

F.10.3 Visualizing Eigenvectors

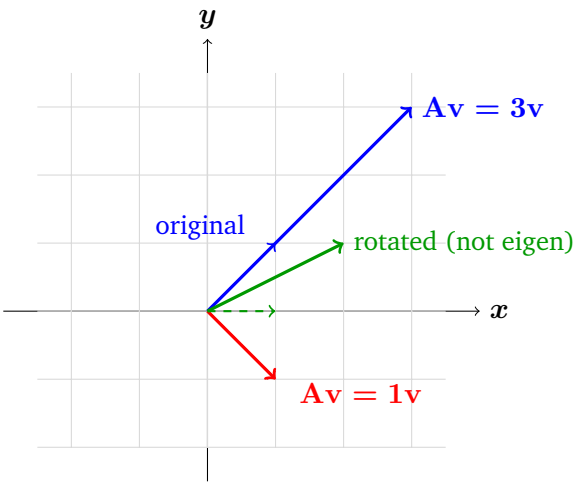


Figure F.3: Eigenvectors (blue and red) only get scaled, they stay on the same line through the origin. Non-eigenvectors (green) get rotated to point in a different direction.

Most vectors get both rotated and scaled when multiplied by a matrix. Eigenvectors are special because they *only* get scaled. They stay pointing in the same direction (or exactly opposite if $\lambda < 0$).

F.10.4 Why Should You Care?

Eigenvectors and eigenvalues reveal the "natural directions" of a matrix. The directions along which the matrix acts most simply. This turns out to be deeply useful.

Understanding complex transformations: Any matrix transformation, no matter how complicated, acts simply along its eigenvector directions: just scaling, no rotation. This makes eigenvectors the natural "coordinate system" for understanding what a matrix does.

Principal Component Analysis (PCA): When you have high-dimensional data (say, images with millions of pixels), you want to find the most important

directions. Where the data varies most. These turn out to be the eigenvectors of the data's covariance matrix.

Imagine data points scattered in 3D space, forming an elongated cloud (like a rugby ball shape). The eigenvectors point along the long axis, medium axis, and short axis of this cloud. The eigenvalues tell you how spread out the data is along each axis. The eigenvector with the largest eigenvalue points in the direction of maximum spread. This is the "most important" direction in your data.

Google's PageRank: The importance of web pages can be computed using eigenvectors. If you build a matrix describing which pages link to which, the eigenvector tells you the long-term probability of a random web surfer being on each page. Pages with high eigenvector values are "important."

Stability of systems: If you repeatedly multiply a vector by a matrix (like $\mathbf{AAA}\mathbf{v}$), what happens depends on the eigenvalues:

- If all $|\lambda| < 1$: the vector shrinks toward zero (stable, signals die out)
- If any $|\lambda| > 1$: the vector grows without bound (unstable, signals explode)
- If $|\lambda| = 1$: the vector stays bounded (marginally stable)

This is crucial for understanding recurrent neural networks, where the same weight matrix is applied repeatedly.

F.10.5 Finding Eigenvalues (Optional)

You do not need to compute eigenvalues by hand for machine learning, computers do this. But here is the idea: we want vectors $\mathbf{v}$ where $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$, which we can rewrite as:

$$(\mathbf{A} - \lambda\mathbf{I})\mathbf{v} = \mathbf{0}$$

For this to have a non-zero solution, the matrix $(\mathbf{A} - \lambda\mathbf{I})$ must be singular (have no inverse), which means:

$$\det(\mathbf{A} - \lambda\mathbf{I}) = 0$$

This is called the **characteristic equation**. For an $n \times n$ matrix, it is a polynomial of degree n in λ, so there are at most n eigenvalues.

For our example matrix $\mathbf{A} = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$:

$$\det\begin{pmatrix} 2-\lambda & 1 \\ 1 & 2-\lambda \end{pmatrix} = (2-\lambda)^2 - 1 = \lambda^2 - 4\lambda + 3 = (\lambda - 1)(\lambda - 3) = 0$$

So $\boldsymbol{\lambda = 1}$ and $\boldsymbol{\lambda = 3}$, which matches what we found earlier by testing vectors.

F.11 Summary

Linear Algebra Quick Reference

Scalar: A single number.
Vector: Ordered list of numbers. Dot product returns a **scalar**.
Norm: $\|\mathbf{x}\| = \sqrt{\mathbf{x} \cdot \mathbf{x}}$ (vector length).
Matrix: Table of numbers, size $m \times n$.
Matrix multiplication: $(m \times n) \cdot (n \times p) = (m \times p)$. **Not commutative!**
Inverse: $\mathbf{A}^{-1}$ satisfies $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$. Not all matrices have one.
Eigenvector: $\mathbf{A}\mathbf{v} = \boldsymbol{\lambda}\mathbf{v}$. Special vectors that only get scaled (not rotated) by the matrix. The eigenvalue $\boldsymbol{\lambda}$ is the scaling factor. Used in PCA, PageRank, and analyzing system stability.

Appendix G

Computational Complexity for Machine Learning

> In theory, there is no difference between theory and practice. In practice, there is.
>
> Attributed to Yogi Berra

Why Complexity Matters for IAIO

IAIO problems frequently ask you to analyze or compare algorithms based on their computational requirements. Understanding Big-O notation and the complexity of common ML algorithms is essential.

G.1 Big-O Notation Review

Big-O notation describes how an algorithm's running time or space usage grows with input size.

Common Complexity Classes

Notation	Name	Example
$O(1)$	Constant	Array access by index
$O(\log n)$	Logarithmic	Binary search
$O(n)$	Linear	Single pass through data
$O(n \log n)$	Linearithmic	Efficient sorting (mergesort)
$O(n^2)$	Quadratic	Comparing all pairs
$O(n^3)$	Cubic	Matrix multiplication (naive)
$O(2^n)$	Exponential	Brute-force subset search

Key intuition: For $n = 1{,}000{,}000$:

- $O(n)$: 1 million operations $\approx$ instant
- $O(n \log n)$: 20 million operations $\approx$ instant
- $O(n^2)$: 1 trillion operations $\approx$ hours
- $O(2^n)$: more operations than atoms in the universe

G.2 Complexity of Machine Learning Algorithms

Let n = number of training examples, d = number of features, k = number of clusters/neighbors/classes.

G.2.1 Supervised Learning

Algorithm	Training Time	Prediction Time
k-Nearest Neighbors	$O(1)$ (just store data)	$O(nd)$ per query
Linear Regression (closed-form)	$O(nd^2 + d^3)$	$O(d)$
Linear Regression (gradient descent)	$O(tnd)$ for t iterations	$O(d)$
Logistic Regression	$O(tnd)$ for t iterations	$O(d)$
Decision Tree	$O(nd \log n)$	$O(\log n)$
Naïve Bayes	$O(nd)$	$O(d \cdot k)$
SVM (kernel)	$O(n^2 d)$ to $O(n^3)$	$O(n_{\text{sv}} \cdot d)$

G.2.2 Unsupervised Learning

Algorithm	Time Complexity
k-Means (one iteration)	$O(nkd)$
k-Means (full, t iterations)	$O(tnkd)$
Hierarchical Clustering (naive)	$O(n^3)$
Hierarchical Clustering (optimized)	$O(n^2 \log n)$
PCA (full SVD)	$O(\min(n^2 d, nd^2))$
PCA (top k components)	$O(ndk)$

G.3 Time vs. Space Tradeoffs

A fundamental principle in algorithm design is trading time for space, or vice versa.

Classic Tradeoff: k-NN vs. Parametric Models

k-Nearest Neighbors:
- Training: $O(1)$ time, $O(nd)$ space (store all data)
- Prediction: $O(nd)$ time per query
- Must keep entire training set in memory

Logistic Regression:
- Training: $O(tnd)$ time, $O(d)$ space
- Prediction: $O(d)$ time per query
- Only stores d parameters, training data can be discarded

Tradeoff: k-NN has "free" training but expensive prediction and storage. Logistic regression invests in training to get cheap prediction and minimal storage.

G.3.1 Other Common Tradeoffs

Precomputation vs. Query Time:

- Precompute all pairwise distances: $O(n^2)$ space, $O(1)$ lookup
- Compute distances on-demand: $O(1)$ space, $O(n)$ per query

Exact vs. Approximate:

- Exact nearest neighbor: $O(n)$ per query
- Approximate (locality-sensitive hashing): $O(1)$ per query, but may miss true nearest neighbor

Batch vs. Online:

- Batch gradient descent: sees all n examples per update, stable but slow

- Stochastic gradient descent: one example per update, noisy but fast
- Mini-batch: compromise between the two

G.4 IAIO-Style Complexity Questions

1. **Comparison**: You have $n = 10^6$ training examples and $d = 100$ features. Which is faster for making 1000 predictions: k-NN or logistic regression (assuming LR is already trained)?

   Answer: k-NN requires $1000 \times 10^6 \times 100 = 10^{11}$ operations. LR requires $1000 \times 100 = 10^5$ operations. LR is 10^6 times faster.

2. **Scaling**: A decision tree takes 1 second to train on 1000 examples. Approximately how long will it take on 1 million examples?

   Answer: Training is $O(nd \log n)$. Ratio: $\frac{10^6 \log 10^6}{10^3 \log 10^3} = \frac{10^6 \times 20}{10^3 \times 10} = 2000$. About 2000 seconds $\approx$ 33 minutes.

3. **Bottleneck identification**: PCA on a dataset with $n = 1000$, $d = 10{,}000$. What dominates the computation?

   Answer: PCA is $O(\min(n^2 d, nd^2))$. Here $n^2 d = 10^{10}$ and $nd^2 = 10^{11}$. The $n^2 d$ term dominates, so computation scales with n^2, not d^2.

G.5 Summary

Key Takeaways

Know your complexities: Memorize the training and prediction complexity of major algorithms.
Identify the bottleneck: Is n or d larger? Which term dominates?
Understand tradeoffs: Fast training often means slow prediction (k-NN). Slow training often means fast prediction (neural networks).
Consider space: Some algorithms require storing all training data; others compress it into parameters.
Approximate when necessary: For very large n, exact algorithms may be infeasible. Know when approximations help.

G.6 Practice Problems

1. You need to classify 1 million new examples. You have a trained decision tree (depth 20) and a trained k-NN model ($k = 5$, 100K training points, 50 features). Which is faster? By how much?

2. A dataset has $n = 10{,}000$ and $d = 5$. You run k-means with $k = 10$ for 20 iterations. How many distance calculations are performed?

3. Explain why hierarchical clustering is rarely used on datasets with more than 10,000 points.

4. You have 1GB of RAM. Your training data is 100,000 examples $\times$ 1,000 features (float32). Can you use k-NN? Can you use logistic regression? Justify.

5. For a kernel SVM with n training points and n_{sv} support vectors, explain why prediction time depends on n_{sv}, not n.

Appendix H

Solutions to Practice Problems

This appendix provides solutions to the practice problems from each chapter. Solutions highlight the key insight for each problem. For computational problems, intermediate steps are shown; for conceptual problems, the core reasoning is given concisely.

Chapter 1: Introduction to Artificial Intelligence

1.

(a) Spam filter: Narrow AI (single task: classify emails)

(b) A system that can learn any intellectual task a human can: AGI (general intelligence across domains)

(c) Chess engine: Narrow AI (excels at one game, cannot do anything else)

(d) A hypothetical AI that surpasses all human cognitive abilities: ASI

(e) Self-driving car: Narrow AI (specific task, though complex)

(f) Virtual assistant answering questions: Narrow AI (only AGI exists today in narrow forms)

2. A $\mathbf{640 \times 480}$ color image has $\mathbf{640 \times 480 = 307{,}200}$ pixels. Each pixel has 3 color channels (Red, Green, Blue), so the computer stores $\mathbf{307{,}200 \times 3 = 921{,}600}$ numbers.

3. If the same data is used for both training and evaluation, the model can memorize the training examples and appear to perform perfectly, but this tells us nothing about how it handles new, unseen data. Keeping test data separate provides an honest estimate of generalization performance.

4. This is overfitting. The model has memorized the training data (99% accuracy) but fails to generalize to new examples (60%). The gap between training and test performance is the hallmark of overfitting. Remedies include using a simpler model, adding regularization, or collecting more training data.

5. Classification assigns an image to a category (e.g., "cat" or "dog"). Localization identifies where a single object is in the image by drawing a bounding box around it. Object detection combines both: it finds all objects in the image, classifies each one, and draws a bounding box around each. Detection is the hardest because it must handle multiple objects of different classes simultaneously.

6. The system may have learned correlations between gender and hiring outcomes from historical data that reflected past discrimination. Even if gender is not an explicit input, proxy features (name, university, extracurricular activities) can encode gender information. This is an example of how AI can perpetuate and amplify existing societal biases through biased training data.

Chapter 2: The Societal Impact of AI

1. Disparate impact occurs when a seemingly neutral policy disproportionately harms a protected group. Mathematically, the 80% rule states that disparate impact exists if $\frac{\min(p_A, p_B)}{\max(p_A, p_B)} < \mathbf{0.8}$. If Group A has approval rate 60% and Group B has 40%, the ratio is $\mathbf{0.4/0.6 = 0.67 < 0.8}$, indicating disparate impact.

2. Demographic parity requires equal positive prediction rates across groups: $\boldsymbol{P(\hat{Y} = 1 | G = A) = P(\hat{Y} = 1 | G = B)}$. Equalized odds requires equal TPR and FPR across groups.

Example: A loan system approves 50% of each group (satisfies demographic parity). But Group A has TPR = 90%, FPR = 30%, while Group B has TPR = 70%, FPR = 10%. This violates equalized odds because error rates differ by group.

3.

(a) Group A: TPR $= \mathbf{80/100 = 0.80}$, FPR $= \mathbf{10/100 = 0.10}$. Group B: TPR $= \mathbf{60/100 = 0.60}$, FPR $= \mathbf{10/100 = 0.10}$.

(b) Does not satisfy equalized odds: FPR is equal (both 0.10), but TPR differs (0.80 vs. 0.60). Equalized odds requires both rates to be equal.

(c) Group A positive rate: $\mathbf{90/200 = 0.45}$. Group B positive rate: $\mathbf{70/200 = 0.35}$. Not equal, so demographic parity is not satisfied.

4. If $P(Y = 1|G = A) \neq P(Y = 1|G = B)$ (different base rates), a calibrated classifier satisfies $P(Y = 1|\hat{Y} = 1, G = g) = p$ for all groups. Under calibration, the positive prediction rate for group g is proportional to the base rate $P(Y = 1|G = g)$. Since base rates differ, prediction rates must differ, violating demographic parity. Calibration and demographic parity are mathematically incompatible when base rates differ.

5. Suppose Group A has $P(Y = 1) = 0.5$ and feature X is informative, while Group B has $P(Y = 1) = 0.5$ but X is biased (encodes group membership rather than true signal). Removing X forces both groups to use the same remaining features, which may be less informative for Group A. Result: accuracy drops for both groups because the biased feature was genuinely useful for Group A, even though it harmed Group B.

6.

(a) Excluding gender does not guarantee non-discrimination. Proxy variables (salary, job title, years of experience) correlate with gender due to historical pay gaps and occupational segregation.

(b) "Salary at previous job" is the strongest proxy: the gender pay gap means this directly encodes gender information. "Job title" may also correlate through occupational segregation.

(c) Compare model predictions across genders for individuals with identical qualifications. If outcomes differ, proxy discrimination exists. Formal tests: check whether the model's residuals correlate with gender after controlling for legitimate predictors.

7.

(a) Content recommendation: DATA: user viewing history. OBJECTIVE: maximize watch time. METRIC: minutes watched. HARM: promotes addictive, extreme, or misleading content. MITIGATION: cap recommendations of sensational content; include diversity metrics.

(b) Predictive policing: DATA: historical crime reports. OBJECTIVE: forecast crime locations. METRIC: prediction accuracy. HARM: reinforces over-policing in historically targeted communities (feedback loop). MITIGATION: audit for geographic bias; combine with community input.

(c) Medical triage: DATA: patient vitals, symptoms, history. OBJECTIVE: prioritize by urgency. METRIC: survival outcomes. HARM: systematic deprioritization of groups whose symptoms present differently (e.g., women with heart attacks). MITIGATION: validate across demographic groups; require human override capability.

8.

(a) GPS data reveals home and work locations through repeated patterns. Even without names, a person's daily routine (home at night, office during day) is nearly unique and easily linked to public records.

(b) Cross-reference home location with property records, work location with employer directories, or frequent locations with social media check-ins. Research has shown that just four spatiotemporal points can uniquely identify 95% of individuals.

(c) Differential privacy would add noise to location data, but the tradeoff is reduced utility. Coarse-grained noise (rounding to neighborhoods) preserves some value for advertisers but still risks re-identification. Fine-grained noise renders the data commercially useless.

9.

(a) Potentially responsible parties: the AI developer (for system design and validation), the hospital (for choosing to deploy it and defining its role), the clinicians (for not exercising independent judgment), and regulators (for approval without adequate oversight).

(b) Needed information: the system's accuracy on similar patients, whether the patient's case was within the system's training distribution, whether clinicians had the option to override, and whether the system's recommendation was the proximate cause of the outcome.

(c) For allowing: complex cases benefit from pattern recognition across millions of cases; requiring full explainability would block potentially life-saving tools. Against: patients have a right to understand decisions about their care; unexplainable systems cannot be properly audited for bias or errors.

10.

(a) Rejected applicants cannot build credit history, so future data continues to show low repayment in those areas. The AI's predictions become self-fulfilling: rejection prevents the evidence that would disprove the prediction.

(b) The disparity would increase. Denied loans prevent economic development in those areas, worsening future data and reinforcing the rejection pattern. This is a positive feedback loop.

(c) Interventions: approve a random sample of borderline applicants from disadvantaged areas to collect unbiased outcome data; use causal modeling to separate geographic effects from creditworthiness; mandate

periodic retraining on data that includes these experimental approvals.

11.

(a) Prefer System A (interpretable) for conditions where treatment decisions are complex and require physician understanding, where errors have legal consequences requiring explanation, or for chronic conditions where patients need to understand and trust the diagnosis process.

(b) Prefer System B (accurate) for screening of life-threatening conditions where the 7% accuracy gain translates to meaningfully more lives saved, and where follow-up testing can confirm or correct the AI's recommendation.

(c) You can explain: the system identified patterns in your data that are associated with the condition being tested for. You cannot explain exactly which patterns or why. You can offer that a physician has reviewed the recommendation and that confirmatory testing is available.

12.

(a) Per-user training cost: $\textbf{€10 000 000}/\textbf{100 000 000} = \textbf{€0.10}$. Per-user carbon: $\textbf{500}/\textbf{100 000 000} = \textbf{0.000005}$ tonnes $= \textbf{5}$ mg $CO_{\mathbf{2}}$.

(b) Total inference cost: $\textbf{1 000 000 000} \times \textbf{€0.001} = \textbf{€1 000 000}$. So inference costs €1M compared to €10M for training. Over time, inference dominates.

(c) Compare the aggregate benefit (productivity gains, accessibility improvements, educational value) against the costs. Consider: does the model replace more carbon-intensive alternatives? Are the benefits equitably distributed? Is the €0.10 per user a reasonable cost for the value provided?

13.

(a) Potential harms: non-consensual deepfakes, political disinformation, fraud and impersonation, erosion of trust in visual evidence, and psychological harm to depicted individuals.

(b) Current safeguards: content filters blocking known harmful prompts, invisible watermarking of generated images, blocking generation of named public figures, and NSFW content filters.

(c) DATA: billions of internet images (including copyrighted and personal photos). OBJECTIVE: generate images from text descriptions. METRIC: image quality and text alignment. HARM: non-consensual imagery,

copyright infringement, bias amplification. MITIGATION: content filtering, watermarking, opt-out mechanisms for artists, bias auditing of outputs.

Chapter 3: Data Preparation and Feature Engineering

1. Data: $[2, 4, 6, 8, 10]$

Mean: $\mu = (2 + 4 + 6 + 8 + 10)/5 = 30/5 = 6$

Variance: $\sigma^2 = [(2-6)^2 + (4-6)^2 + (6-6)^2 + (8-6)^2 + (10-6)^2]/5 = [16 + 4 + 0 + 4 + 16]/5 = 40/5 = 8$

Standard deviation: $\sigma = \sqrt{8} \approx 2.83$

Scaled values: $\tilde{x}_i = (x_i - 6)/2.83$: approximately $-1.41, -0.71, 0, 0.71, 1.41$.

2. One-hot encoding creates 3 columns (one per category). "Medium" is encoded as $[0, 1, 0]$ (assuming order Small, Medium, Large).

3. With 5000 samples and 70-15-15 split: Training: $5000 \times 0.70 = 3500$. Validation: $5000 \times 0.15 = 750$. Test: $5000 \times 0.15 = 750$.

4. Suggested engineered features: (1) days since last purchase (recency), (2) average purchase amount over last 30 days (monetary trend), (3) number of purchases per product category (preference signal), (4) day-of-week and hour-of-day purchase patterns (temporal behavior), (5) ratio of current purchase to user's historical average (anomaly signal).

5. Two approaches: (1) Impute with median income, which is robust to outliers and preserves sample size, best when missingness is random. (2) Drop rows with missing income, which is simple but loses 10% of data, best when data is abundant and missingness might be informative (e.g., high earners declining to report).

6. Log transform helps because house prices are typically right-skewed (many modest homes, few mansions). The transform compresses the upper range, making the distribution more symmetric and reducing the outsized influence of extreme values on the model's loss function.

7. Three ways preprocessing could cause the 98%/0% gap: (1) Data leakage: scaling or imputing using statistics from the full dataset (including test), so the model "sees" test information during training. (2) Distribution mismatch: the training data was cleaned in ways that real-world data cannot replicate (e.g., removing outliers that are common in production). (3) Feature encoding errors: categorical features were encoded using training-set-specific mappings that break on unseen categories in new data.

8.

(a) k-NN computes distances, so income (range 0–1,000,000) would dominate age (0–100) and children (0–10). Neighbors would be determined almost entirely by income, ignoring the other features.

(b) Decision trees split on individual features and are invariant to monotonic transformations of any single feature. Scaling would have no effect on the tree's splits or accuracy.

(c) Decision trees are scale-invariant because they use threshold comparisons on individual features. Distance-based methods (k-NN, SVM, K-means) are scale-sensitive because they combine features into a single distance measure.

9.

(a) Three strategies: (1) mean/median imputation, (2) deletion of rows with missing values, (3) model-based imputation (e.g., predict income from other features).

(b) Best scenarios: (1) Median imputation when missingness is random and sample size is limited. (2) Deletion when data is abundant and missingness correlates with the target. (3) Model-based when other features predict income well and missingness is systematic.

(c) Harmful scenarios: (1) Median imputation when high earners systematically decline to report (biases income distribution downward). (2) Deletion when missingness is non-random, creating a biased remaining sample. (3) Model-based when the imputation model itself has errors that propagate through the pipeline.

10. "Cancellation reason" is target leakage: it is only available after cancellation has occurred, so it encodes the outcome directly. A model using it would appear highly accurate in training but fail in production (the feature would be empty for active customers). Verification: check when the feature is populated in the data pipeline. The other features (days since last login, total purchases, session duration, support tickets) are all available before cancellation and are legitimate predictors.

11. These features are not leakage because they are known at prediction time and describe the house itself, not the outcome. Square footage, bedrooms, year built, and location are all observable before a sale and are genuinely predictive of price. Leakage requires that information from the future or from the target variable itself contaminates the features. Strong correlation with the target is the entire point of a good feature, not evidence of leakage.

12.

(a) No leakage: hours studied is measured before the exam and is a legitimate predictor.

(b) Target leakage: satisfaction with grade is measured after the exam and encodes the outcome.

(c) Feature leakage: computing statistics on the combined dataset lets training features incorporate test-set information.

(d) No leakage: length of initial stay is known at discharge, which is before the readmission outcome.

13.

(a) Geographic bias: underrepresents driving conditions in snow, rain, dense urban areas, and right-hand-drive countries.

(b) Selection bias: reviews are written by people who feel strongly. The moderate majority is underrepresented, skewing sentiment toward extremes.

(c) Survivorship/attrition bias: dropouts may have experienced side effects. The remaining 70% are likely healthier, making the drug appear more effective than it is.

(d) Survivorship bias: the dataset excludes failed startups. Any pattern found (e.g., "successful startups have charismatic founders") may be equally true of failures.

14.

(a) Covariate shift (input distribution changed dramatically as pandemic altered travel patterns) and potentially concept drift (the relationship between features like time-of-day and demand also changed).

(b) Retraining on Jan–Feb 2020 would not help: those months still reflect pre-pandemic behavior. The fundamental relationships between features and demand changed, so more recent pre-pandemic data would not capture the new reality.

(c) Monitor the distribution of incoming features versus training features (detect covariate shift), track prediction error on recent outcomes (detect accuracy degradation), and set alerts when prediction confidence drops systematically.

15. The 94% test accuracy is unreliable because both training and test data come from the same clinics and population (Northern European). This

measures in-distribution performance only. On patients with different skin tones, lighting conditions, or camera equipment, accuracy could be much lower. A valid test would require data from diverse clinics, skin types, and geographies. Without this, the reported accuracy overstates real-world performance.

Chapter 4: Supervised Learning

1. For a 2000 square foot house: $h(x) = 50\,000 + 100 \times 2000 = €250\,000$. The intercept €50 000 is the predicted base price (when $x = 0$, which is not meaningful). The coefficient 100 means each additional square foot adds €100 to the predicted price.

2. The output $h_\theta(\mathbf{x}) = 0.73$ is interpreted as a probability: the model estimates a 73% chance of belonging to the positive class. At threshold 0.5, since $0.73 > 0.5$, the predicted class is positive (class 1).

3. After splitting on A: left branch has $p_+ = 4/4 = 1$, entropy $= 0$. Right branch has $p_+ = 1/6$, entropy $= -\frac{1}{6}\log_2\frac{1}{6} - \frac{5}{6}\log_2\frac{5}{6} \approx 0.65$. Weighted average entropy $= \frac{4}{10}(0) + \frac{6}{10}(0.65) = 0.39$. Information gain $= 1.0 - 0.39 = 0.61$.

4. SVM finds the hyperplane that maximizes the minimum distance (margin) to any training point. The parameter C controls the tradeoff between maximizing the margin and allowing misclassifications. Large C: narrow margin, few violations (risk of overfitting). Small C: wide margin, more violations tolerated (risk of underfitting).

5. L1 (Lasso) adds $\lambda \sum |w_i|$: drives some weights exactly to zero, performing feature selection. Prefer when you expect many irrelevant features. L2 (Ridge) adds $\lambda \sum w_i^2$: shrinks all weights toward zero but rarely to exactly zero. Prefer when all features are potentially relevant and you want to reduce their magnitudes uniformly.

6. *Weighted Neighbor Classifier:*

(a) As $\sigma \to 0$: weights concentrate on the single nearest neighbor (WNC becomes 1-NN). As $\sigma \to \infty$: all weights become equal, and the prediction is a vote across all training points (majority class wins).

(b) It is a smooth, weighted version of k-NN. The Gaussian weighting function is itself a kernel, making WNC a kernel classifier with the RBF kernel.

(c) The decision boundary is nonlinear. It follows the contour where the weighted vote is balanced, which curves around clusters of each class.

(d) Failure modes: sensitive to σ (too small = overfitting, too large = underfitting); prediction requires computing distance to all training points ($O(n)$ per query); struggles with class imbalance since the majority class dominates the weighted sum.

(e) Normalize weights by class: $\hat{y} = \text{sign}\left(\sum_{i:y_i=+1} w_i/n_+ - \sum_{i:y_i=-1} w_i/n_-\right)$, where n_+, n_- are class counts. This ensures each class contributes equally regardless of size.

7.

(a) Suppose $w_1x_1 + w_2x_2 + b = 0$ separates the classes. The constraints require: $b < 0$ (for $(0,0) \to 0$), $w_2 + b > 0$ (for $(0,1) \to 1$), $w_1 + b > 0$ (for $(1,0) \to 1$), and $w_1 + w_2 + b < 0$ (for $(1,1) \to 0$). Adding the second and third: $w_1 + w_2 + 2b > 0$. But from the first and fourth: $w_1 + w_2 + 2b < b + (w_1 + w_2 + b) < 0$. Contradiction. □

(b) With $x_3 = x_1x_2$: points become $(0,0,0), (0,1,0), (1,0,0), (1,1,1)$. The plane $x_1 + x_2 - 2x_3 = 0.5$ separates label-1 points from label-0 points.

(c) Decision tree: split on x_1. If $x_1 = 0$: split on x_2 (output x_2). If $x_1 = 1$: split on x_2 (output $1 - x_2$).

(d) The kernel trick implicitly maps to a higher-dimensional space (as in part b) without explicitly computing the new features. A polynomial kernel of degree 2 includes the x_1x_2 cross-term automatically.

8.

(a) "Years at current address" correlates with age (older people move less frequently). The negative coefficient effectively penalizes younger applicants, constituting indirect age discrimination through a proxy variable.

(b) The AUC drop from 0.82 to 0.79 is a 3-point decrease. Whether this is acceptable depends on context: if 0.79 still exceeds the bank's minimum performance threshold, the fairness gain may justify the accuracy cost. If the model makes 10,000 decisions per year, even a small AUC change can affect hundreds of applicants.

(c) Compute demographic parity (equal approval rates by age group), equalized odds (equal TPR/FPR by age group), and the 80% rule (ratio of approval rates). Also check calibration: among applicants given the same predicted probability, do actual default rates differ by age?

9. *Prototype Classifier:*

(a) The boundary is a hyperplane. The set $\{\mathbf{x} : \|\mathbf{x} - \boldsymbol{\mu}_1\| = \|\mathbf{x} - \boldsymbol{\mu}_2\|\}$ is the perpendicular bisector of the line segment connecting the two centroids, which is always a hyperplane.

(b) k-NN with $k =$ all points uses a majority vote over the entire class. The prototype classifier uses only the centroid (mean). They are equivalent when the data in each class is spherically symmetric, but differ when class distributions are asymmetric or multi-modal.

(c) The classifier works well when each class forms a single, roughly spherical cluster centered on its mean.

(d) A class with two well-separated sub-clusters (e.g., class A at $\{(0, 0), (10, 10)\}$ and class B at $\{(5, 5)\}$). The centroid of class A is $(5, 5)$, which coincides with class B. The classifier fails because the mean does not represent any actual region of class A.

(e) The classifier stores $C \times d$ parameters (C centroids, each d-dimensional). k-NN stores all $n \times d$ training points. Logistic regression stores $(d + 1) \times C$ weights. For typical problems, the prototype classifier is the most compact.

Chapter 5: Model Evaluation

1. With 10,000 points, a 70-15-15 split gives: 7,000 training, 1,500 validation (for hyperparameter tuning), and 1,500 test (for final evaluation). The validation set is used during development; the test set is touched only once at the end to report unbiased performance.

2. The 34% gap between training (99%) and validation (65%) accuracy indicates high variance (overfitting). The model has memorized the training data but fails to generalize. This is a low-bias, high-variance regime. Fixes: simplify the model, add regularization, collect more training data, or use dropout/early stopping.

3. From the confusion matrix (TP=450, FP=100, FN=50, TN=400):

- Precision $= 450/(450 + 100) = 450/550 \approx 81.8\%$
- Recall $= 450/(450 + 50) = 450/500 = 90\%$
- F1 $= 2 \times (0.818 \times 0.90)/(0.818 + 0.90) = 2 \times 0.736/1.718 \approx 85.7\%$

4. Optimize for precision when false positives are costly. Example: spam filtering, where a false positive means a legitimate email is lost. You would rather let some spam through (lower recall) than accidentally delete important

messages (high precision).

5.

(a) A single split depends on which particular points land in train vs. test, introducing high variance. Averaging over k folds means every point serves as both training and test, reducing this variance and giving a more stable estimate.

(b) As $k \to n$ (LOO-CV): bias decreases (each training set is nearly the full dataset), variance increases (the n training sets overlap almost completely, making fold estimates highly correlated), and computational cost becomes $O(n)$ model trainings.

(c) $k = 5$ or $k = 10$ balances bias (training sets are 80–90% of data, so models are representative) with variance (folds are different enough to give independent estimates) and computational cost (only 5–10 model trainings).

(d) If multiple samples come from the same patient, standard CV might put different samples from the same patient in both train and test, making the model appear better than it would on truly new patients. Use grouped k-fold CV, where all samples from a patient are kept in the same fold.

6.

(a) Bias $= \mathbb{E}[\hat{f}(x)] - f(x)$: the systematic deviation of the model's average prediction from the true function.

(b) A constant model $\hat{f}(x) = c$ ignores all input features, so it cannot capture any structure in $f(x)$. Its expected prediction is always c, creating large bias wherever $f(x) \neq c$.

(c) An interpolating model fits every training point exactly, so its predictions change dramatically with different training sets. Each training set produces a different function, creating high variance in predictions at any given x.

(d) Regularization increases bias (the model is constrained from fitting the training data perfectly) but can reduce variance by a larger amount (the model is less sensitive to the specific training set). The net effect is lower total error when the variance reduction outweighs the bias increase.

7.

(a) Accuracy $= (90 + 9405)/10000 = 94.95\%$. Precision $= 90/(90 +$

$495) = 90/585 \approx 15.4\%$. Recall $= 90/(90 + 10) = 90\%$. F1 $= 2 \times (0.154 \times 0.90)/(0.154 + 0.90) \approx 26.3\%$.

(b) A trivial classifier predicting "no disease" for everyone achieves 99% accuracy. The model's 94.95% is actually worse than this baseline, meaning it creates more errors than predicting the majority class. Accuracy is misleading because the classes are so imbalanced.

(c) PPV (positive predictive value) $= 90/585 \approx 15.4\%$. Only about 1 in 6 positive tests is a true positive. This differs from recall (90%) because recall conditions on actual positives while PPV conditions on predicted positives. With low prevalence, even a high-recall test produces many false positives.

8. Three ways AUC = 0.99 could be inflated: (1) data leakage, where test set information leaked into training through shared preprocessing; (2) target leakage, where a feature directly encodes the outcome (e.g., using discharge diagnosis to predict admission outcome); (3) temporal leakage, where future information appears in features that would not be available at prediction time. Any AUC above 0.95 on a real-world problem should trigger careful investigation.

9. The second model (70% train / 65% test) is better. The first model (100% train / 60% test) has a 40% generalization gap indicating severe overfitting, so its 60% test accuracy is unreliable and likely optimistic. The second model's 5% gap is modest, meaning its 65% test accuracy is trustworthy. A reliable 65% is more valuable than an unreliable 60%.

Chapter 6: Unsupervised Learning and Clustering

1. K-means with points $\{(1,1), (2,1), (4,3), (5,4)\}$, initial centroids $\mu_1 = (1,1)$, $\mu_2 = (5,4)$:

Iteration 1: Assign each point to the nearest centroid. $(1,1) \to \mu_1$ (dist 0), $(2,1) \to \mu_1$ (dist 1), $(4,3) \to \mu_2$ (dist $\sqrt{2}$), $(5,4) \to \mu_2$ (dist 0). Update: $\mu_1 = (1.5, 1)$, $\mu_2 = (4.5, 3.5)$.

Iteration 2: Reassign. $(1,1) \to \mu_1$ (dist 0.5), $(2,1) \to \mu_1$ (dist 0.5), $(4,3) \to \mu_2$ (dist $\sqrt{0.5}$), $(5,4) \to \mu_2$ (dist $\sqrt{0.5}$). Assignments unchanged, so the algorithm has converged.

2. Between $(2,5,1)$ and $(6,2,3)$: Euclidean $= \sqrt{(6-2)^2 + (2-5)^2 + (3-1)^2} = \sqrt{16 + 9 + 4} = \sqrt{29} \approx 5.39$. Manhattan $= |6-2| + |2-5| + |3-1| = 4 + 3 + 2 = 9$.

3. K-means finds local minima because the objective (total distortion) is non-

convex. Different initializations start in different basins of attraction, leading to different local optima. Mitigation: run K-means multiple times with random restarts and keep the best result, or use K-means++ initialization (choosing initial centroids that are spread apart).

4. The responsibilities mean: 80% probability the point belongs to cluster 1, 15% to cluster 2, 5% to cluster 3. Unlike K-means, which hard-assigns each point to exactly one cluster, GMM provides soft probabilistic memberships. This is useful when cluster boundaries overlap, allowing a single point to partially belong to multiple clusters.

5. The largest drop is from $\boldsymbol{k = 1}$ to $\boldsymbol{k = 2}$ (60 units), then $\boldsymbol{k = 2}$ to $\boldsymbol{k = 3}$ (15 units). After $\boldsymbol{k = 3}$, improvements are marginal (3, then 2). The "elbow" is at $\boldsymbol{k = 3}$, where the rate of improvement levels off.

6.

(a) PCA: reduces 1000 features to 50 while preserving maximum variance. PCA is designed for dimensionality reduction as a preprocessing step.

(b) t-SNE: specifically designed for 2D/3D visualization of high-dimensional data, preserving local neighborhood structure.

(c) Hierarchical clustering (with dendrogram): the dendrogram lets you choose the number of clusters after the fact by cutting at different heights.

(d) GMM: models clusters as Gaussians with different covariance matrices, naturally handling elliptical shapes and varying sizes. K-means assumes spherical clusters of equal size.

7.

(a) Projecting to 3D loses $\boldsymbol{100\% - 85\% = 15\%}$ of the variance.

(b) Acceptable because 85% captures most of the structure; the 50D-to-3D reduction provides enormous computational savings; and the discarded 15% may largely consist of noise rather than meaningful signal.

8. t-SNE only preserves local structure (nearby points stay nearby), not global distances. The algorithm may place clusters at arbitrary relative positions. Distances between clusters in a t-SNE plot are not meaningful, so concluding that "A is closer to B than C" from inter-cluster distances is unreliable.

9. K-means fails on concentric circles (ring-shaped clusters) because it assumes convex, spherical clusters. No initialization will separate an inner ring from an outer ring. Single-linkage hierarchical clustering fails on clusters connected by a "bridge" of noise points, because it merges the nearest pair

of points, chaining through the bridge to merge two distinct clusters (the chaining effect).

10.

(a) The 80% concentration is not inherently discriminatory. It may reflect genuine geographic or demographic patterns. You would need to know whether services are allocated based on need versus ethnicity, whether the clustering produces equitable outcomes, and whether alternative clusterings yield different demographic compositions.

(b) Adding ethnicity as a feature explicitly uses a protected characteristic, which is legally and ethically problematic. It changes the optimization objective from "group by need" to "group by need and diversity," which may reduce service quality for all groups.

(c) Compare the resident's features to all cluster centroids. If they are closer to a different centroid, the assignment may be suboptimal. Also check whether the resident is an outlier poorly served by any cluster, which would suggest the cluster-based system itself is inadequate for this individual.

Chapter 7: Kernel Methods

1.

(a) $\mathbf{x}^T\mathbf{z} = 1 \cdot 3 + 2 \cdot 1 = 5$. $\kappa(\mathbf{x}, \mathbf{z}) = (1 + 5)^2 = 36$.

(b) $(1+x_1z_1+x_2z_2)^2 = 1+2x_1z_1+2x_2z_2+x_1^2z_1^2+2x_1x_2z_1z_2+x_2^2z_2^2$. The implicit feature map is $\phi(\mathbf{x}) = (1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, \sqrt{2}x_1x_2, x_2^2)$, a 6-dimensional space.

2.

(a) $\|\mathbf{x} - \mathbf{z}\|^2 = 1^2 + 1^2 = 2$. $\kappa = e^{-2/2} = e^{-1} \approx 0.368$.

(b) $\kappa(\mathbf{x}, \mathbf{x}) = e^{-0/2\sigma^2} = e^0 = 1$ always.

(c) As $\sigma \to 0$, $\kappa \to 0$ for all $\mathbf{x} \neq \mathbf{z}$, and $\kappa = 1$ only for $\mathbf{x} = \mathbf{z}$. Each training point becomes isolated: the model memorizes every point individually, leading to extreme overfitting.

3.

(a) If K_1 and K_2 are PSD, then for any vector $\mathbf{v}$: $\mathbf{v}^T(K_1 + K_2)\mathbf{v} = \mathbf{v}^TK_1\mathbf{v} + \mathbf{v}^TK_2\mathbf{v} \geq 0 + 0 = 0$. So $K_1 + K_2$ is PSD, meaning $\kappa_1 + \kappa_2$ is a valid kernel. $\square$

(b) $\kappa(\mathbf{x}, \mathbf{z}) = |\mathbf{x}^T\mathbf{z}|$ is not a valid kernel. For points $\mathbf{x} = (1, 0)$, $\mathbf{y} = (-1, 0)$, $\mathbf{z} = (0, 1)$: $\boldsymbol{K} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$. While this particular matrix is PSD, taking $\mathbf{x} = (1, 1)$, $\mathbf{y} = (1, -1)$, $\mathbf{z} = (-1, 1)$ yields a kernel matrix that is not PSD.

4.

(a) The prediction is a weighted sum of kernel similarities between the new point and all training points. Points more "similar" to $\mathbf{x}$ (as measured by the kernel) contribute more to the prediction. It is a form of pattern matching: new predictions are based on resemblance to known examples.

(b) The primal formulation requires inverting a $\boldsymbol{d} \times \boldsymbol{d}$ matrix ($\boldsymbol{O(d^3)}$), while the dual requires inverting an $\boldsymbol{n} \times \boldsymbol{n}$ matrix ($\boldsymbol{O(n^3)}$). When $\boldsymbol{d \gg n}$, the dual is far cheaper. Furthermore, the kernel trick means we never need to compute the (possibly infinite-dimensional) feature mapping explicitly.

5.

(a) Not linearly separable. The middle point $(0, 1)$ has the opposite label from its neighbors $(-1, -1)$ and $(1, -1)$. No line $\boldsymbol{y = wx + b}$ can place $(0, 1)$ above and both $(-1, -1)$ and $(1, -1)$ below.

(b) $\boldsymbol{\phi(x) = (x, x^2)}$ maps to: $(-1, 1), (0, 0), (1, 1)$. Now the line $\boldsymbol{x_2 = 0.5}$ separates the positive point $(0, 0)$ (below) from the negative points $(-1, 1)$ and $(1, 1)$ (above). Actually, checking labels: $(0) \to (+1)$ and $(\pm 1) \to (-1)$, so $\boldsymbol{x_2 < 0.5}$ gives $+1$, which correctly classifies all points.

(c) A polynomial kernel of degree 2, $\boldsymbol{\kappa(x, z) = (1 + xz)^2}$, implicitly includes the $\boldsymbol{x^2}$ feature.

6.

(a) As $\boldsymbol{\lambda \to \infty}$: all weights are driven to zero. The model predicts the mean of the training targets for all inputs (maximum regularization, underfitting).

(b) As $\boldsymbol{\lambda \to 0}$: no regularization. The model fits the training data exactly, potentially overfitting.

(c) Use cross-validation: try a range of $\boldsymbol{\lambda}$ values (e.g., on a logarithmic grid), evaluate each on the validation set, and select the $\boldsymbol{\lambda}$ with the best validation performance.

7. Kernel methods are "memory-based" because they must store all training examples and compute kernel values with each one at prediction time. Advantage: no explicit training phase, easy to add or remove data points. Disadvantage: prediction time is $O(n)$ per query (scales with dataset size). Neural networks, by contrast, learn compressed weight parameters during training, so prediction is $O(1)$ regardless of training set size, but updating the model requires retraining.

8.

(a) No straight line can separate concentric circles. A line divides the plane into two half-planes, but the inner circle is surrounded by the outer circle. Any line that captures part of the inner circle will also include part of the outer circle.

(b) Adding $x_3 = x_1^2 + x_2^2$ (squared distance from origin): inner circle points have $x_3 \approx 1$, outer circle points have $x_3 \approx 4$. In 3D, the classes are separated by height. A horizontal plane at $x_3 = 2.5$ cleanly separates them.

(c) The RBF kernel measures similarity via distance. Points on the same circle are close to each other and far from points on the other circle. The kernel implicitly lifts the data into a high-dimensional space where the two rings become linearly separable, much like the explicit x_3 feature in part (b) but more flexibly.

(d) As $\sigma \to 0$, each point only "sees" itself. The decision boundary becomes a tiny region around each training point, classified by that point's label. This is extreme overfitting: it memorizes the training set perfectly but the boundary between training points is essentially random, giving poor generalization.

9.

(a) The kernel matrix for points $\{1, 2, 3\}$ is: $K = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 2 & 3 \end{pmatrix}$. Eigenvalues: all positive ($\approx 0.20, 0.68, 5.12$). The matrix is PSD, so $\min(x, z)$ is a valid kernel for non-negative inputs.

(b) It corresponds to the feature map $\phi(x) = (\mathbf{1}_{[t \leq x]})_{t \geq 0}$, an indicator function over the positive reals. The inner product $\langle \phi(x), \phi(z) \rangle = \int_0^\infty \mathbf{1}_{[t \leq x]} \mathbf{1}_{[t \leq z]} \, dt = \min(x, z)$.

Chapter 8: AI Search and Constraint Satisfaction

1.

(a) BFS explores by depth: A (depth 0), then B, C (depth 1), then D (depth 2), then E (depth 3). Order: A, B, C, D, E (with tie-breaking: B before C alphabetically).

(b) DFS with alphabetical tie-breaking: A $\rightarrow$ B $\rightarrow$ D $\rightarrow$ E. Goal found at depth 3 via path A-B-D-E (cost 6).

(c) Optimal path: A $\rightarrow$ C $\rightarrow$ D $\rightarrow$ E with cost $\mathbf{1 + 2 + 1 = 4}$.

2.

(a) $\boldsymbol{f(A) = 0 + 5 = 5}$. When B is discovered via A: $\boldsymbol{f(B) = 2 + 3 = 5}$. When C is discovered via A: $\boldsymbol{f(C) = 1 + 4 = 5}$. When D is discovered via C: $\boldsymbol{f(D) = 3 + 1 = 4}$. When E is discovered via D: $\boldsymbol{f(E) = 4 + 0 = 4}$.

(b) A* order: expand A ($\boldsymbol{f = 5}$), generating B and C. Expand B or C ($\boldsymbol{f = 5}$ each, tie-break alphabetically: B). B generates D ($\boldsymbol{f = 5}$). Expand C ($\boldsymbol{f = 5}$), generating D ($\boldsymbol{f = 3 + 1 = 4}$, cheaper path found). Expand D ($\boldsymbol{f = 4}$), generating E ($\boldsymbol{f = 4}$). Expand E, goal found. Path: A-C-D-E, cost 4.

3. Choice 1: MIN chooses $\mathbf{min\{7, 3, 9\} = 3}$. Choice 2: MIN chooses $\mathbf{min\{5, 8, 2\} = 2}$. MAX chooses $\mathbf{max\{3, 2\} = 3}$. MAX should make Choice 1, guaranteeing a value of 3.

4. After fully evaluating Choice 1, $\boldsymbol{\alpha = 3}$ (MAX is guaranteed at least 3). Exploring Choice 2: the first child has value 5, so MIN could still pick something $\leq \mathbf{3}$. After seeing the second child (value 8), MIN still could pick lower. After seeing the third child (value 2), MIN picks $\mathbf{min\{5, 8, 2\} = 2}$. Since $\mathbf{2 < \alpha = 3}$, MAX would never choose Choice 2. In this tree, no pruning occurs because we must see all children of Choice 2 before MIN's value is determined. (Pruning would occur if an early child of Choice 2 had value $\leq \mathbf{3}$, since MIN's value can only decrease.)

5. A* is "optimally efficient" among algorithms using the same heuristic: no other algorithm can find the optimal solution while expanding fewer nodes. This means A* extracts maximum value from the heuristic information, it never does unnecessary work. This is important because in large search spaces, even small efficiency gains translate to solving otherwise intractable problems.

6.

(a) Corner cells have 2 moves, edge cells (non-corner) have 3, interior cells have 4. With 4 corners, 12 edge cells, and 9 interior cells: average branching factor $\approx (4 \times 2 + 12 \times 3 + 9 \times 4)/25 = 80/25 = 3.2$.

(b) Manhattan distance is admissible: it computes the shortest path assuming no obstacles (straight horizontal + vertical). Since obstacles can only make the true path longer, Manhattan distance never overestimates the true cost. Therefore $h(n) \leq h^*(n)$ for all n, satisfying admissibility.

7.

(a) Total assignments: $3^4 = 81$ (3 colors for each of 4 regions).

(b) Assigning A = Red removes Red from B's and C's domains (since A–B and A–C): $B : \{G, B\}$, $C : \{G, B\}$, $D : \{R, G, B\}$.

(c) Assigning B = Green removes Green from A (already assigned), C, and D: $C : \{B\}$ (only Blue left), $D : \{R, B\}$ (Green removed via B–D adjacency). A valid completion is C = Blue, D = Red.

Chapter 9: Reinforcement Learning

1. The Bellman optimality equations:

$$V^*(A) = \max\{1 + 0.9 \cdot V^*(A),\ 0 + 0.9 \cdot V^*(B)\}$$

$$V^*(B) = \max\{2 + 0.9 \cdot V^*(A),\ 1 + 0.9 \cdot V^*(B)\}$$

Trying $A \to A$ and $B \to A$: $V^*(A) = 1 + 0.9V^*(A)$, so $V^*(A) = 10$. $V^*(B) = 2 + 0.9 \times 10 = 11$. Check $B \to B$: $1 + 0.9V^*(B) = 1 + 9.9 = 10.9 < 11$. So optimal: stay at A ($V^* = 10$), go A from B ($V^* = 11$).

2. $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$

$= 5 + 0.1[2 + 0.9 \times 8 - 5] = 5 + 0.1[2 + 7.2 - 5] = 5 + 0.1 \times 4.2 = 5.42$.

3. Always taking the best known action (exploitation) risks missing better alternatives that have not been tried. Early estimates may be inaccurate, so an action that appeared suboptimal might actually be superior. Exploration (trying less-visited actions) is necessary to gather information and avoid converging to a locally good but globally suboptimal policy.

4. As $\gamma \to 0$: only immediate reward matters. The agent becomes purely myopic, choosing whichever action gives the highest instant payoff. As $\gamma \to 1$: all future rewards are weighted equally. The agent plans for the long term, but the infinite horizon can cause value functions to diverge if rewards are unbounded, and convergence becomes slower.

5. Starting from $Q^\pi(s,a) = \mathbb{E}[R_{t+1} + \gamma Q^\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a]$:

For the optimal policy π^*, the action A_{t+1} is chosen to maximize Q^*. Substituting: $Q^*(s,a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} Q^*(S_{t+1}, a') | S_t = s, A_t = a]$. Expanding the expectation over the transition: $Q^*(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')]$. This is the Bellman optimality equation for Q^*.

6.

(a) $Q^*(s_0, A) = 10$ (immediate reward, then terminal). $Q^*(s_0, B) = 0 + 0.9 \times 5 = 4.5$. $Q^*(s_0, C) = 0 + 0.9 \times 20 = 18$. Optimal action is C.

(b) A greedy agent selects the action with the highest known Q-value. Having only tried A (reward +10), its Q-estimate for A is 10 while B and C remain at their initial values (typically 0). Since $10 > 0$, greedy always picks A, never discovering the superior action C.

(c) With $\epsilon = 0.1$, at each episode the agent explores randomly with probability 0.1. In 1000 episodes: $\sim$900 times it picks greedily (action A), $\sim$100 times randomly ($\sim$33 each for A, B, C). So roughly: A $\approx$ 933, B $\approx$ 33, C $\approx$ 33. After $\sim$33 tries of C (receiving reward 18 via state Y), the agent will have learned that C is superior, and the greedy action will switch to C. Yes, it will eventually learn the optimal policy.

(d) Initialize Q-values optimistically (e.g., $Q_0 = 100$ for all actions). Then the greedy policy naturally explores: after trying A and getting 10 (less than 100), the agent tries B, then C. After one trial of each, it discovers C gives 18 and converges to the optimal policy much faster.

7.

(a) States: 16 cells (4×4 grid), but (2,2) is terminal (pit) and (4,4) is terminal (goal), so 14 non-terminal states with decisions. Each state has up to 4 actions. Possible deterministic policies: $4^{14} = 268{,}435{,}456$ (upper bound; edge/corner states have fewer actions).

(b) With $\gamma = 0.9$ and -1 per step, a safe path of 6 steps earns roughly $-6 + 0.9^6 \times 10 \approx -6 + 5.3 = -0.7$. A risky path of 3 steps through (2,2)'s neighborhood earns roughly $-3 + 0.9^3 \times 10 \approx -3 + 7.3 = 4.3$ if it avoids the pit, but $-3 + 0.9^2 \times (-10) = -11.1$ if it falls in. The agent will prefer a moderately safe path that avoids the pit but does not take an excessively long detour.

(c) $Q((1,1), \text{right}) \leftarrow Q((1,1), \text{right}) + \alpha[-1 + \gamma \max_{a'} Q((2,1), a') - Q((1,1), \text{right})]$.

Chapter 10: Generative Models and Deep Learning

1. Generative models learn the joint distribution $P(X, Y)$ and can generate new data. Example: a model that learns what each digit looks like and can produce new digit images. Discriminative models learn the conditional $P(Y|X)$ and only classify. Example: a logistic regression or CNN that takes a digit image and outputs the digit label. The generative model can create images; the discriminative model can only label them.

2.

(a) Compression ratio: $784/32 = 24.5\times$.

(b) The latent space of a vanilla autoencoder is not structured for generation. Random vectors will likely land in "dead zones" that the decoder was never trained to handle, producing garbage outputs. The encoder maps training images to specific scattered points, not to a smooth, continuous region.

3.

(a) The first term (reconstruction loss) encourages the decoder to faithfully reconstruct the input. The second term (KL divergence) encourages the encoder's latent distribution to be close to a standard normal, ensuring the latent space is smooth and well-structured.

(b) If $\beta = 0$: the VAE becomes a standard autoencoder (excellent reconstruction but unstructured latent space, poor generation). If $\beta \to \infty$: the KL term dominates, forcing $q(z|x) \approx \mathcal{N}(0, I)$. The encoder ignores the input entirely, reconstruction collapses, but the latent space is perfectly structured (and useless).

(c) Without the KL term, the encoder can map different inputs to distant, isolated regions of latent space. Sampling a random z for generation would land between these regions, where the decoder has no training signal. The KL term forces all encodings toward the same prior distribution, filling in the gaps and enabling smooth interpolation and sampling.

4.

(a) The 10,000-image model has memorized its training data (overfitting). With limited data, the model lacks enough examples to learn the general concept of "face" and instead stores specific faces. The 10-million-image model has sufficient diversity to learn generalizable features.

(b) Compare generated outputs to nearest neighbors in the training set.

If generated images are pixel-for-pixel close to training examples, the model is memorizing. A generalizing model should produce outputs that are plausible but not identical to any training example. Quantitative measure: compute the FID score, or check if the minimum distance to training examples exceeds a threshold.

5. VAEs produce blurry images because the reconstruction loss (typically MSE) penalizes pixel-wise deviation. When there is uncertainty about fine details (e.g., exact position of a hair strand), the loss-minimizing strategy is to produce the average of all plausible outputs, which is a blurry blend. MSE treats a sharp-but-wrong image as worse than a blurry-but-average image, so the model learns to hedge by blurring.

Chapter 11: Statistical Learning Theory

1. Using the finite-class PAC bound: $m \geq \frac{1}{\epsilon}(\ln|\mathcal{H}| + \ln(1/\delta)) = \frac{1}{0.1}(\ln 1000 + \ln(1/0.05)) = 10(6.91 + 3.00) = 99.1$. So approximately 100 samples are needed.

2. The pigeons received food at random intervals unrelated to their actions, yet they repeated whatever behavior preceded the reward. They had no prior knowledge to distinguish coincidence from causation. The rats had prior knowledge about the physical world (pressing things can cause effects), so they correctly identified the lever as causal. Prior knowledge (inductive bias) is essential for learning the right hypothesis from limited evidence.

3. Rectangles represent a specific inductive bias: we assume the good papayas occupy a contiguous rectangular region in feature space. If we used all possible functions, the hypothesis class would be infinite and unconstrained, making learning impossible (we could fit any labeling of any data, but generalize to nothing). The rectangle restriction trades expressiveness for learnability.

4. Sample complexity scales as $O(1/\epsilon)$: halving ϵ roughly doubles the required samples. Going from $\epsilon = 0.1$ to $\epsilon = 0.05$ approximately doubles the sample requirement (from $\sim$100 to $\sim$200 for the same δ and $|\mathcal{H}|$).

5. A consistent learner finds a hypothesis with zero training error. Under the realizability assumption, the true function h^* is in $\mathcal{H}$ and also has zero training error. The PAC bound guarantees that with enough samples, all hypotheses with zero training error must have low true error. So a consistent learner's output is "close to" h^* in error. The key insight: zero training error on enough data implies low true error, provided the hypothesis class is not too large.

6. Solving $\epsilon \geq \frac{\ln|\mathcal{H}|+\ln(1/\delta)}{m}$: with $m = 500$, $|\mathcal{H}| = 1000$, $\delta = 0.05$: $\epsilon \geq \frac{\ln 1000+\ln 20}{500} = \frac{6.91+3.00}{500} = \frac{9.91}{500} \approx 0.02$. So with 500 samples, the generalization error is at most about 2%.

Chapter 12: Advanced Statistical Learning Theory

1. VCdim(intervals) = 2. Can shatter 2 points $a < b$: label $(+,+)$ with interval $[a, b]$; $(+,-)$ with $[a, a+\epsilon]$; $(-,+)$ with $[b-\epsilon, b]$; $(-,-)$ with an interval disjoint from both. Cannot shatter 3 points $a < b < c$: the labeling $(+,-,+)$ requires two disjoint intervals, but the hypothesis class only allows one interval.

2. Decision stumps in $\mathbb{R}^2$ split on one coordinate: $\mathbf{1}[x_i \geq \theta]$ for $i \in \{1, 2\}$. Each such classifier produces a half-plane boundary aligned with one axis. VCdim = 2. Can shatter 2 points with different values on one coordinate. Cannot shatter 3 points: for any arrangement, there exists a labeling requiring a non-axis-aligned boundary.

3. Using the VC bound: $m = O\left(\frac{1}{\epsilon}\left(\text{VCdim} \cdot \log(1/\epsilon) + \log(1/\delta)\right)\right)$. With VCdim = 10, $\epsilon = 0.05$, $\delta = 0.01$: $m \approx \frac{1}{0.05}(10 \cdot \log(20) + \log(100)) = 20(10 \times 3.0 + 4.6) = 20 \times 34.6 \approx 692$ samples.

4. NFL states that no single learner can outperform all others on every possible distribution. But in practice, real-world data distributions are not arbitrary. They exhibit structure (smoothness, low dimensionality, compositionality) that specific algorithms can exploit. Machine learning succeeds because we choose algorithms whose inductive biases match the structure of real data, even though no algorithm works on all conceivable distributions.

5. The 99% training / 70% test gap indicates overfitting: the hypothesis class is too expressive relative to the training data. In VC dimension terms, the model's effective VC dimension is too high for the available sample size, so the generalization bound is loose. Fixes: reduce model complexity (lower VC dimension), add regularization (which restricts to a smaller effective hypothesis class), or increase training data (which tightens the generalization bound).

6. Linear classifiers in $\mathbb{R}^2$ have VCdim = 3, so they can shatter 3 points but not 4. For 4 points, consider the XOR configuration: $(0,0)$ and $(1,1)$ labeled $+$, $(0,1)$ and $(1,0)$ labeled $-$. No single line can separate the two $+$ points (on one diagonal) from the two $-$ points (on the other diagonal).

7.

(a) Can shatter 3 points. Place them in a triangular arrangement: any labeling with at most one negative point can be achieved by an appropriately

sized/positioned rectangle.

(b) Can shatter 4 points. Place them at the top, bottom, left, and right extremes of a diamond shape (e.g., $(\mathbf{0}, \mathbf{1}), (\mathbf{0}, \mathbf{-1}), (\mathbf{-1}, \mathbf{0}), (\mathbf{1}, \mathbf{0})$). Any labeling where positive points form a "connected" group can be captured by a rectangle. In fact, all $\mathbf{2^4 = 16}$ labelings can be achieved.

(c) Cannot shatter 5 points. For any 5 points, consider the one with the smallest $\boldsymbol{x}$-coordinate, largest $\boldsymbol{x}$-coordinate, smallest $\boldsymbol{y}$-coordinate, and largest $\boldsymbol{y}$-coordinate. Any rectangle that includes the extreme points must include the interior point. The labeling where the interior point is $-$ and all four extreme points are $+$ is achievable, but the labeling where only the interior point is $+$ and all extremes are $-$ is also achievable. However, the labeling $(+, -, +, -, +)$ arranged so that negative points are between positive extremes cannot be captured by a single rectangle. VCdim = 4.

(d) VCdim = 4.

(e) With VCdim = 4, $\boldsymbol{m = 500}$, $\boldsymbol{\delta = 0.05}$: $\epsilon \leq \boldsymbol{O}\left(\sqrt{\frac{\mathbf{4 \cdot \log(500) + \log(1/0.05)}}{\mathbf{500}}}\right) \approx \boldsymbol{O}\left(\sqrt{\frac{\mathbf{4 \times 6.2 + 3.0}}{\mathbf{500}}}\right) = \boldsymbol{O}\left(\sqrt{\frac{\mathbf{27.8}}{\mathbf{500}}}\right) \approx \boldsymbol{O}(\mathbf{0.24})$.

8.

(a) NFL applies to the uniform distribution over all possible target functions. Real-world tasks are not drawn from this distribution. Neural networks succeed because real-world data has exploitable structure (hierarchical features, local correlations, compositionality) that matches the network's inductive bias. NFL doesn't say neural networks are universally best; it says they work well on the non-arbitrary distributions we actually encounter.

(b) NFL tells us that the choice of learning algorithm is really a choice of inductive bias. We should select algorithms whose assumptions match our domain knowledge. No algorithm is "free": every advantage on some distribution comes at the cost of disadvantage on another. This justifies domain-specific algorithm selection and model validation.

Chapter 13: Matrix Factorization: A Unifying Idea

1.

(a) Parameters: $\mathbf{1{,}000{,}000 \times 50 + 100{,}000 \times 50 = 50{,}000{,}000 + 5{,}000{,}000 = 55{,}000{,}000}$ (55 million).

(b) Full matrix: $\mathbf{1{,}000{,}000 \times 100{,}000 = 10^{11}}$ entries. Compression: $\mathbf{55{,}000{,}000 / 10^{11} = 0.055\%}$. Over 1800× compression.

(c) Observed entries: $\mathbf{0.001 \times 10^{11} = 10^{8} = 100{,}000{,}000}$. Ratio: 100M observations constraining 55M parameters. This is just under 2 observations per parameter, which is tight but feasible with regularization.

2. $\hat{r}_1 = \mathbf{p} \cdot \mathbf{q_1} = 0.8 \times 0.5 + (-0.3) \times 0.2 + 0.5 \times 0.4 = 0.4 - 0.06 + 0.2 = 0.54$.

$\hat{r}_2 = \mathbf{p} \cdot \mathbf{q_2} = 0.8 \times 0.1 + (-0.3) \times (-0.5) + 0.5 \times 0.8 = 0.08 + 0.15 + 0.4 = 0.63$.

Item 2 ranks higher ($0.63 > 0.54$).

3. TF-IDF for "neural" in a document with 200 words: TF $= 3/200 = 0.015$. IDF $= \log(10{,}000/50) = \log(200) \approx 2.30$. TF-IDF $= 0.015 \times 2.30 \approx 0.035$.

4. Cosine similarity between users (on movies A, B, D, E rated by both):

- User 1: $(5, 4, 2, 1)$, User 2: $(4, 5, 1, 2)$
- Dot product: $20 + 20 + 2 + 2 = 44$
- Magnitudes: $\sqrt{25 + 16 + 4 + 1} = \sqrt{46}$, $\sqrt{16 + 25 + 1 + 4} = \sqrt{46}$
- Cosine: $44/46 \approx 0.957$ (very similar)

5. Cold start: new users have no interaction history, and new items have no ratings. Content-based filtering helps for new items (use item metadata/features to find similar items) but not for new users (still need some user history to build a profile). Collaborative filtering struggles with both: it relies entirely on the interaction matrix, which is empty for new entities. Common solutions: hybrid systems, asking new users for initial preferences, or using demographic information as a proxy.

6. Netflix: 200M users × 100 factors + 15K items × 100 factors = $20{,}000{,}000{,}000 + 1{,}500{,}000 \approx 20$ billion parameters. Full matrix: 200M × 15K = 3×10^{12} entries. The factorization stores ∼20B / $3 \times 10^{12} < 1\%$ of the full matrix. This is feasible to store and compute; the full matrix is not.

Chapter 14: Modern Deep Learning

1. Diffusion models add noise to data gradually until it becomes pure noise, then train a neural network to reverse each step. Generation proceeds by starting from random noise and iteratively denoising. Iterative refinement

produces better results because each step makes a small correction, allowing the model to fix errors at every stage. A one-shot generator must get everything right in a single pass, with no opportunity to correct mistakes, making it harder to produce fine details.

2.

(a) The Query represents "what am I looking for?" The Key represents "what do I contain?" The Value represents "what information do I provide if selected?" Attention computes how much each position's Key matches each Query, then uses those weights to combine Values.

(b) Attention allows every position in a sequence to directly interact with every other position in a single layer, regardless of distance. Previous architectures (RNNs) processed sequences step-by-step, making long-range dependencies hard to learn. Attention eliminates this bottleneck, which is why it is "all you need."

(c) Self-attention computes pairwise scores between all n positions: $O(n^2 \cdot d)$ where d is the embedding dimension. The n^2 term makes it expensive for long sequences.

3. Each parameter requires 16 bits $= 2$ bytes. Total memory: $175 \times 10^9 \times 2 = 350 \times 10^9$ bytes $= 350$ GB. This exceeds the memory of any single consumer GPU (typically 16–80 GB), requiring model parallelism across multiple devices.

4. Next-token prediction requires the model to understand the structure of text at every level. To predict the next word in a math proof, the model must understand mathematics. To predict dialogue, it must model conversation. To predict code, it must understand programming. The training objective is simple, but mastering it on the internet's full diversity of text requires learning a compressed representation of human knowledge. Question answering and dialogue emerge because those patterns appear in the training data and predicting them requires understanding them.

5.

(a) The training objective rewards producing text that is statistically likely given the context, not text that is factually correct. If plausible-sounding false statements appear in training data (or if the model interpolates between true facts), the model learns to produce them confidently. The loss function does not distinguish true from false, only likely from unlikely.

(b) Two approaches: (1) Retrieval-augmented generation (RAG): give the model access to a verified knowledge base and train it to cite sources,

grounding outputs in facts. (2) Reinforcement learning from human feedback (RLHF): train the model to prefer cautious, accurate responses over confident-but-wrong ones, teaching it to express uncertainty when appropriate.

6.

(a) $\boldsymbol{D(x) = 0.9}$ means the discriminator estimates a 90% probability that $\boldsymbol{x}$ is a real image (from the training data) rather than a generated one.

(b) The generator wants the discriminator to believe its outputs are real. Maximizing $\boldsymbol{D(G(z))}$ means pushing the discriminator's output toward 1 ("definitely real") for generated samples, which is exactly what it means to fool the discriminator.

(c) Mode collapse occurs when the generator produces only a small subset of possible outputs (e.g., always generating the same digit). It happens because the generator finds a few outputs that reliably fool the discriminator and converges to those, rather than exploring the full data distribution. The discriminator then learns to reject those specific outputs, but the generator may simply switch to a different small set, cycling without ever covering the full distribution.

These solutions are meant as guides. For many problems, especially conceptual ones, alternative correct answers exist. Focus on understanding the reasoning, not memorizing specific answers.

Appendix I

A Song: I Am Still Mine

Lyrics and music by Eljakim Schrijvers[1]

This appendix is personal, not instructional.
It is here because I believe that everything I love deserves a song.

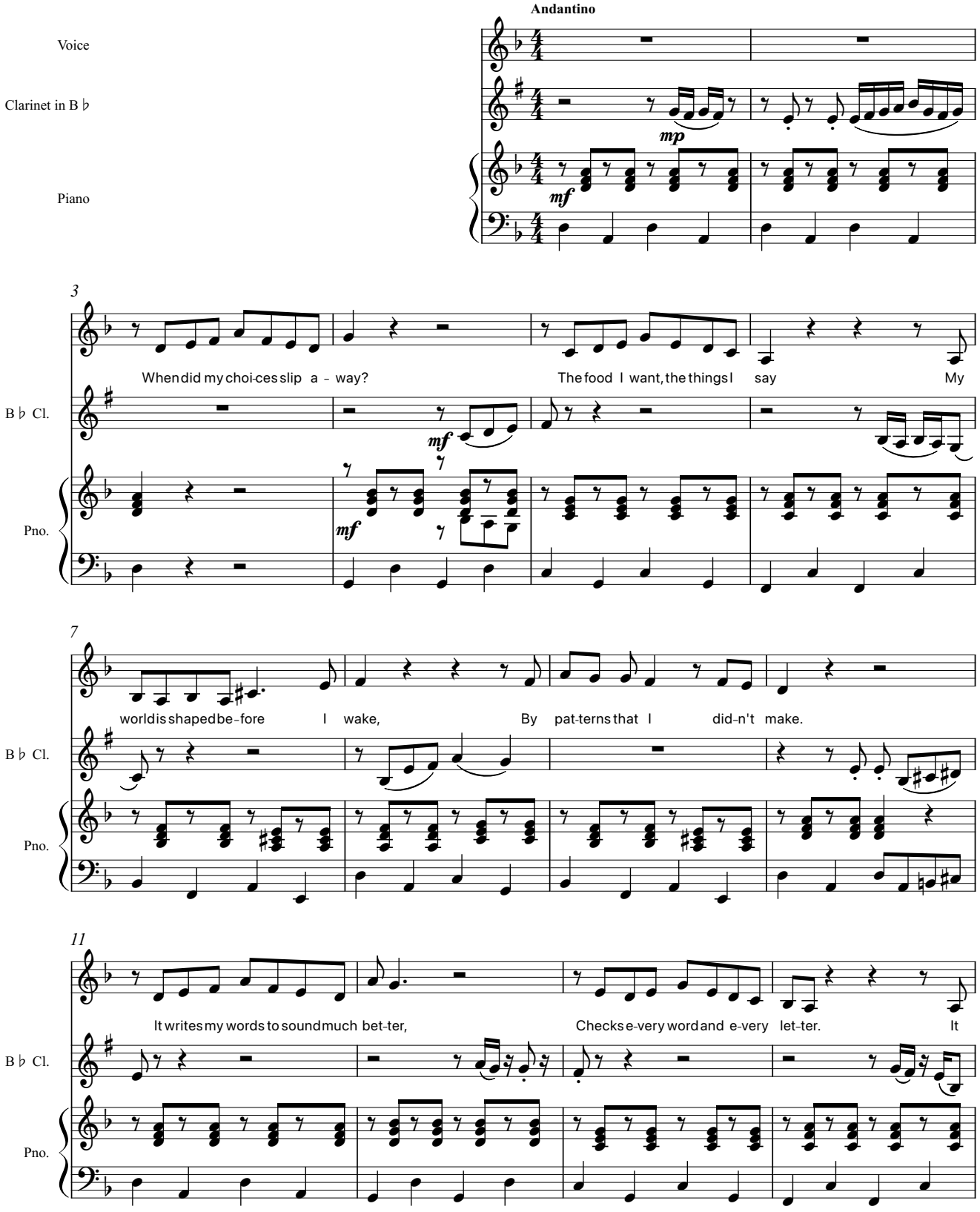
Scan to listen to the melody

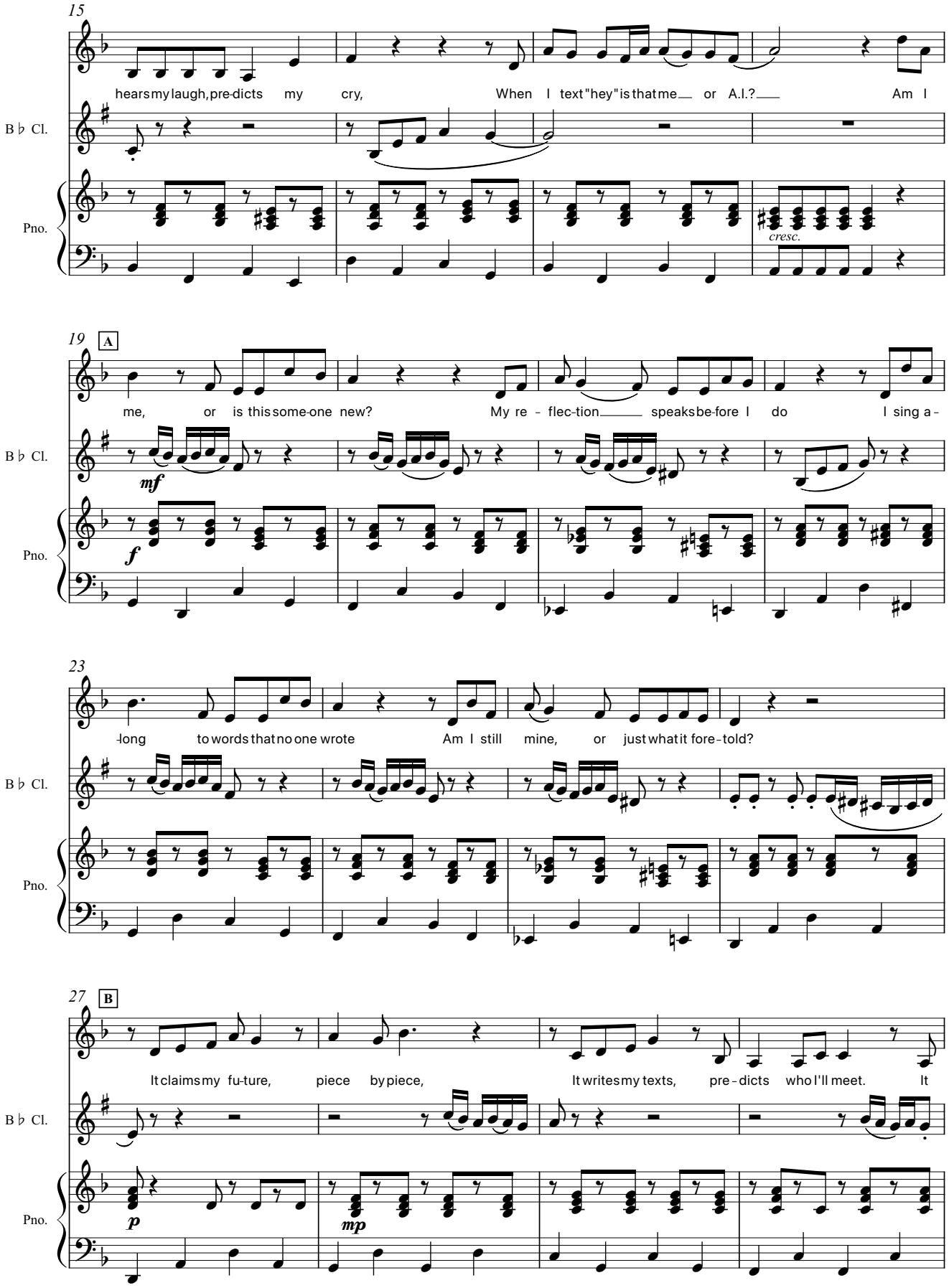
[1]This song was actually hand written. It's not AI-generated.

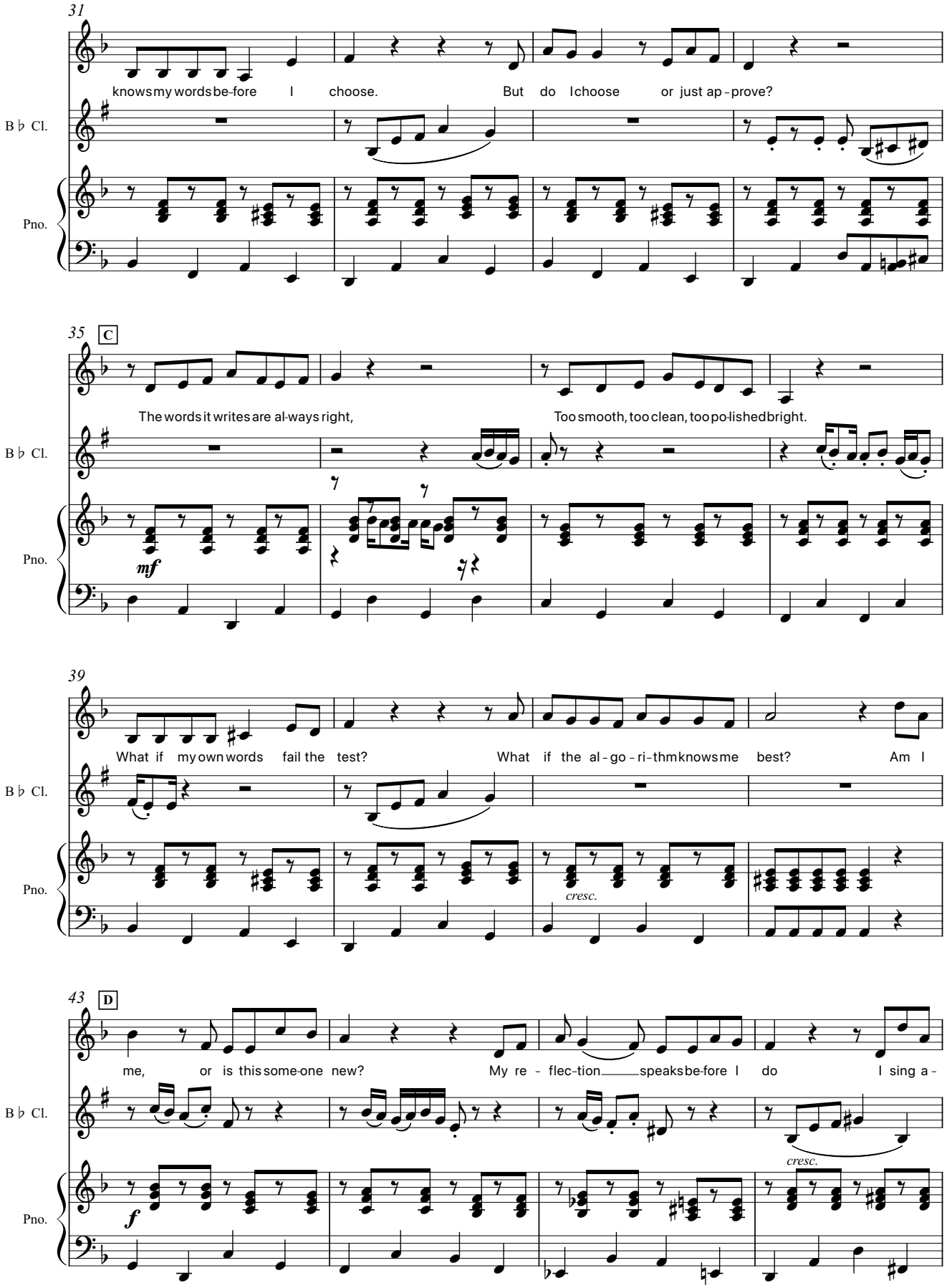
I am Still Mine

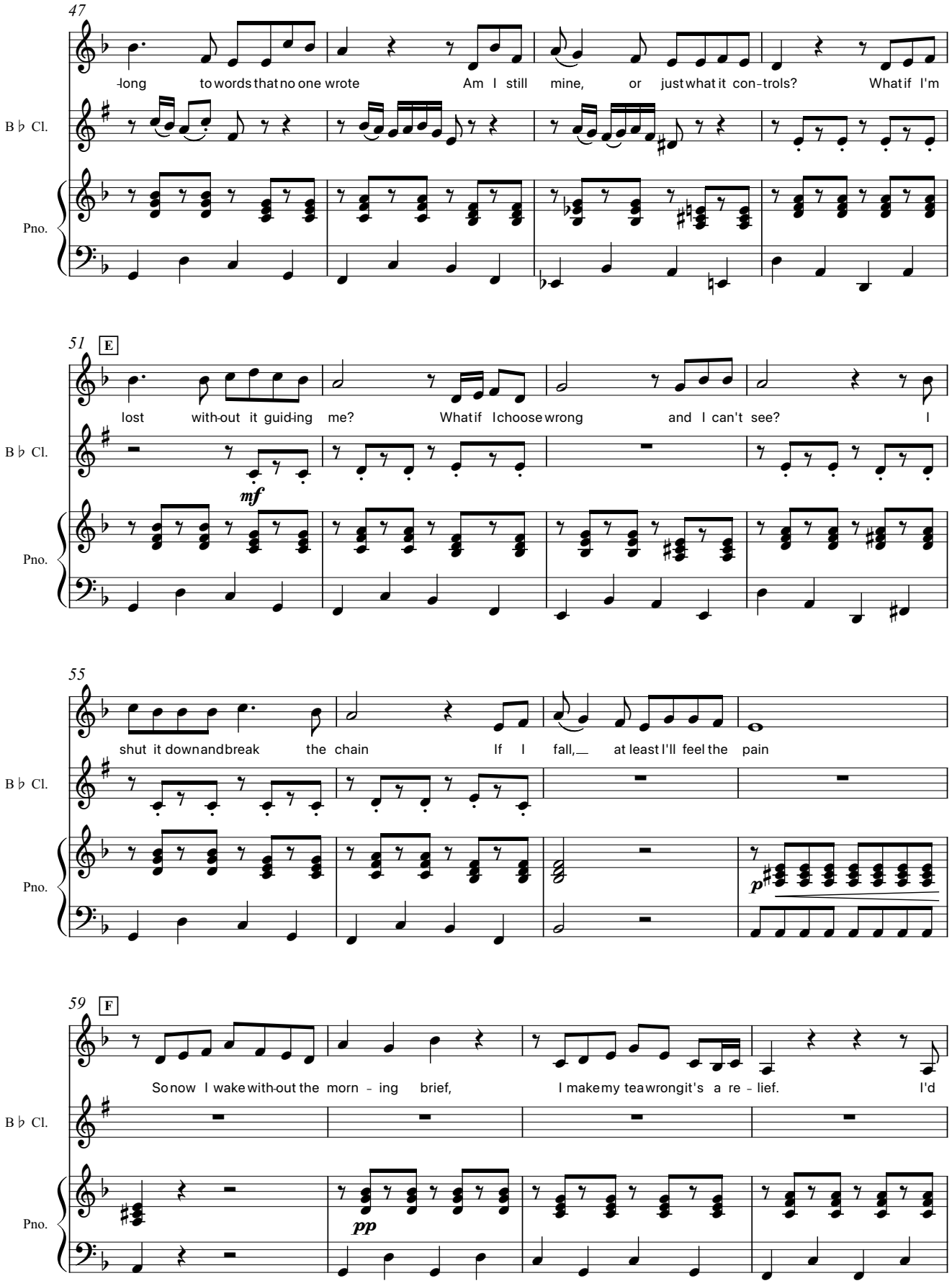
(and therefore yours to hold)

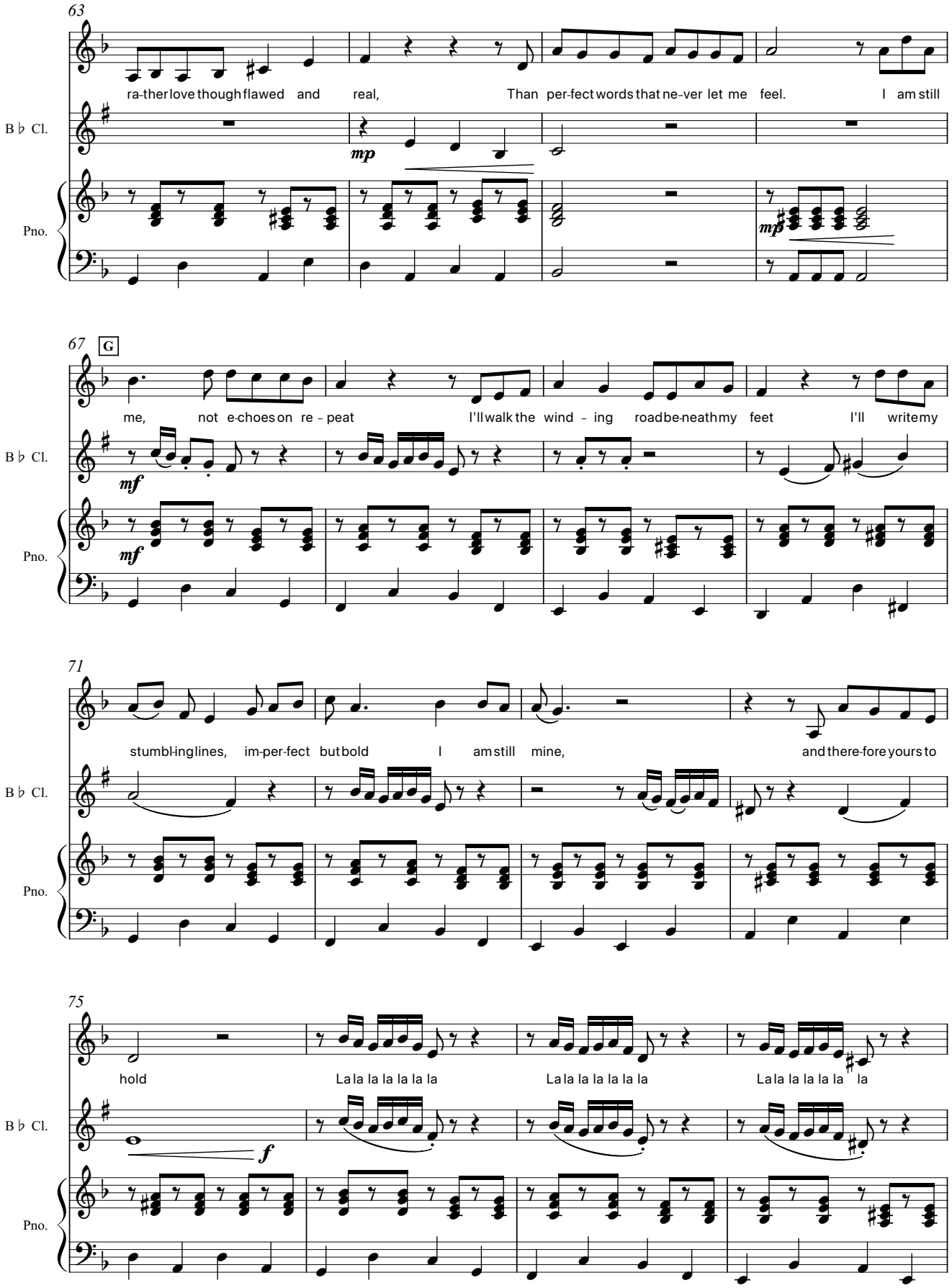
Lyrics/music: Eljakim Schrijvers

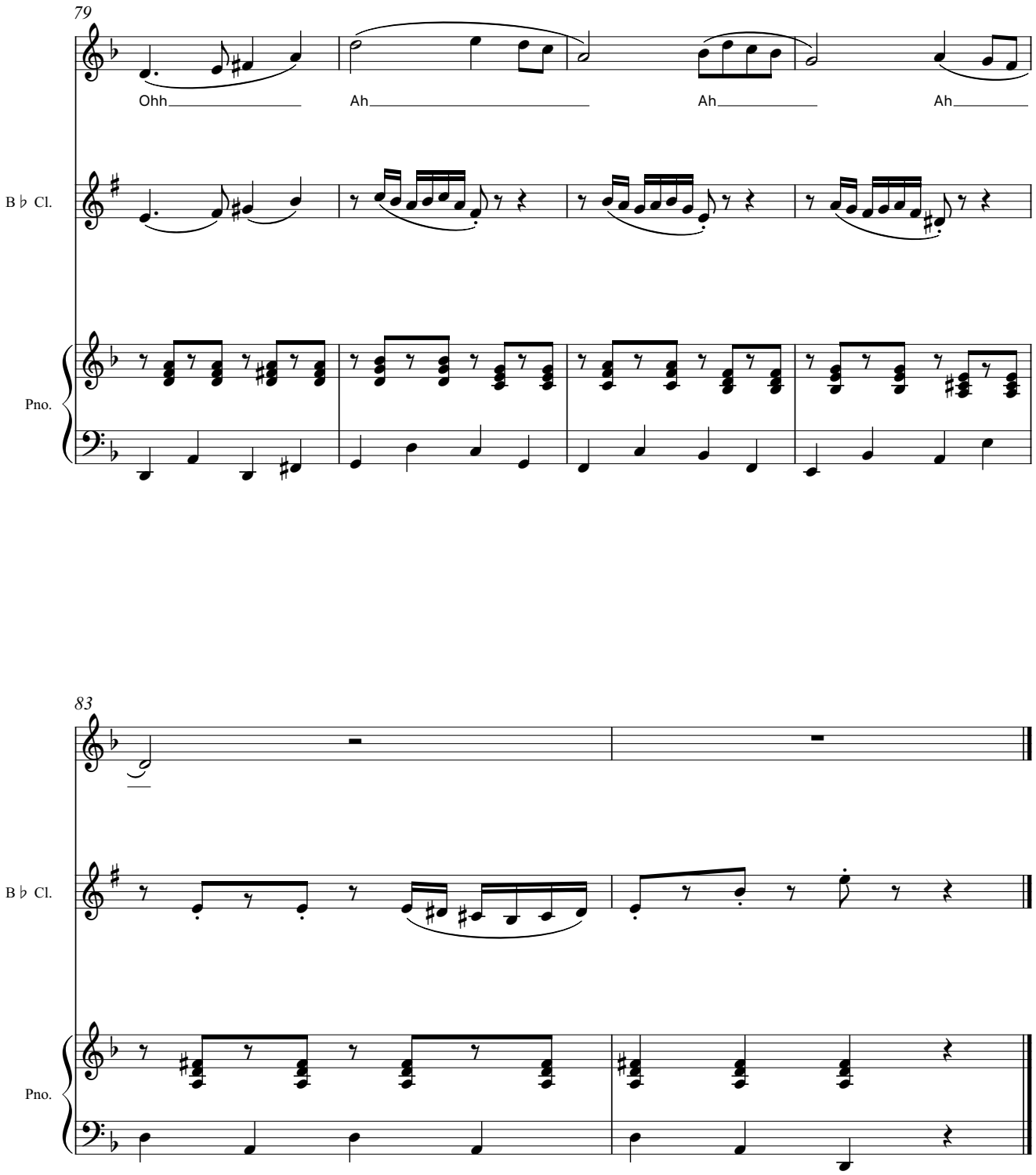



15
hears my laugh, pre-dicts my cry, When I text "hey" is that me or A.I.? Am I
B♭ Cl.
Pno.
cresc.
19
A
me, or is this some-one new? My re - flec-tion speaks be-fore I do I sing a -
mf
f
23
-long to words that no one wrote Am I still mine, or just what it fore-told?
27
B
It claims my fu-ture, piece by piece, It writes my texts, pre-dicts who I'll meet. It
p
mp


31
knows my words be-fore I choose. But do I choose or just ap-prove?
B♭ Cl.
Pno.
35
C
The words it writes are al-ways right, Too smooth, too clean, too po-lished bright.
mf
39
What if my own words fail the test? What if the al-go-ri-thm knows me best? Am I
cresc.
43
D
me, or is this some-one new? My re-flec-tion speaks be-fore I do I sing a-
f
cresc.


47
-long to words that no one wrote Am I still mine, or just what it con-trols? What if I'm
B♭ Cl.
Pno.
51
E
lost with-out it guid-ing me? What if I choose wrong and I can't see? I
mf
B♭ Cl.
Pno.
55
shut it down and break the chain If I fall, at least I'll feel the pain
B♭ Cl.
Pno.
p
59
F
So now I wake with-out the morn - ing brief, I make my tea wrong it's a re - lief. I'd
B♭ Cl.
Pno.
pp


63
ra-ther love though flawed and real, Than per-fect words that ne-ver let me feel. I am still
B♭ Cl.
mp
Pno.
mp
67
G
me, not e-choes on re - peat I'll walk the wind - ing road be-neath my feet I'll write my
B♭ Cl.
mf
Pno.
mf
71
stumbl-ing lines, im-per-fect but bold I am still mine, and there-fore yours to
B♭ Cl.
Pno.
75
hold La la la la la la la La la la la la la la La la la la la la la
B♭ Cl.
f
Pno.


79
Ohh
Ah
Ah
Ah
B♭ Cl.
Pno.
83
B♭ Cl.
Pno.