Machine Learning from Zero

A beginner's book on the fundamentals of machine learning, in plain words and pictures

Nov 26, 2026 · @bagbagig

Welcome

This book teaches the fundamentals of machine learning to someone who has never studied it. You need no maths beyond everyday arithmetic. You need no programming until the last chapter, and even there every line is explained.

Machine learning sounds like magic, but it is not. At heart it is a simple idea: show a computer many examples, and let it find the pattern by itself. Everything else in this book is a way of doing that well, or a way of checking that it was done well.

Who this book is for

- Students who keep hearing about AI and want to know what is really going on.
- Professionals whose work now touches data, models or AI tools.
- Curious adults who want to read the news about AI and judge it for themselves.
- Future programmers and data scientists who want a gentle first step before the heavy books.

If you can read a bar chart and you have ever made a guess based on experience, you already have the two skills this book builds on.

How each chapter works

Every chapter has the same five parts, so you always know where you are.

1. **The big idea** — one or two sentences at the top that say what the chapter is about.
2. **The story** — the explanation in plain words, built around an everyday comparison.
3. **Pictures** — diagrams that show the idea, because one good picture can replace a page of text.
4. **Key ideas** — a short list of what to remember.
5. **Try it yourself** — questions you can answer with a pencil and paper. The answers are in Appendix C.

How to read it

Read Part 1 in order; it builds the vocabulary for everything else. The chapters in Part 2 each teach one kind of model and can be read in any order, though they are easiest in the order given. Part 3 is the most important part of the book. It teaches you how to tell a good model from a bad one, which is the skill most beginners skip and most experts value. Part 4 explains how modern AI, such as image recognition and chatbots, grows out of the same simple ideas.

Do not rush. If a chapter feels hard, read its Key ideas first, then go back to the story.

Our running example: the ice-cream shop

Many chapters return to one small, made-up business so that new ideas attach to something familiar. Maya runs an ice-cream shop by the sea. Every day she writes down a few facts in a notebook: the temperature, whether it rained, whether it was a weekend, and how many ice creams she sold.

After two years she has more than 700 days of notes. Maya wants to know how many ice creams to prepare tomorrow. Too few, and customers leave unhappy. Too many, and ice cream goes to waste. Her notebook, and the questions she asks of it, will teach us most of machine learning.

A word about words

Machine learning has a lot of jargon. Each new term appears in **bold** the first time it is explained. Every bold term is also collected in the glossary in Appendix B, in plain words. When two words mean the same thing, which happens often in this field, the book says so.

What is inside

Part 1 — The Big Picture. What machine learning is, the three main kinds of learning, what data looks like, and how to get data ready.

- Chapter 1: What is machine learning?
- Chapter 2: The three kinds of learning
- Chapter 3: Data, the fuel
- Chapter 4: Getting data ready

Part 2 — Your First Models. Seven families of models, each explained with a picture and an analogy.

- Chapter 5: Drawing a line — linear regression
- Chapter 6: Yes or no — classification and logistic regression

- Chapter 7: Twenty questions — decision trees
- Chapter 8: Wisdom of crowds — random forests and boosting
- Chapter 9: Birds of a feather — k-nearest neighbours
- Chapter 10: Finding groups — clustering with k-means
- Chapter 11: Squeezing data — dimensionality reduction

Part 3 — Is It Any Good? How to test a model honestly, how to keep score, and the traps that fool even experts.

- Chapter 12: The exam analogy — training, validation and test sets
- Chapter 13: Keeping score — metrics
- Chapter 14: Overfitting and underfitting
- Chapter 15: Traps — data leakage, imbalance and baselines

Part 4 — Going Deeper. How machines actually learn, neural networks, modern deep learning, responsibility, and a complete first project.

- Chapter 16: How machines learn — loss and gradient descent
- Chapter 17: Neural networks
- Chapter 18: A tour of deep learning
- Chapter 19: Fairness, ethics and responsibility
- Chapter 20: Your first project, step by step

Appendices. Setting up Python, a glossary, answers to the exercises, and where to go next.

Part 1 — The Big Picture

Chapter 1 — What Is Machine Learning?

The big idea: Machine learning is a way of getting computers to do a task by learning from examples, instead of following rules that a person wrote by hand.

Two ways to teach a computer

Imagine you want a computer to sort your email into “spam” and “not spam”. There are two very different ways to do it.

The first way is the old way, called **traditional programming**. You sit down and write rules. If the email contains the words “free money”, call it spam. If it comes from someone in your contacts, call it safe. If it has more than five exclamation marks, call it spam.

This works for a while. Then problems pile up. You need hundreds of rules, and they start to fight each other. Spammers notice your rules and write “fr3e m0ney” instead. Every week you must write new rules, and you can never keep up.

The second way is **machine learning**. You do not write the rules. Instead you collect thousands of emails that people have already marked as spam or not spam. You give them to the computer and say: find the pattern. The computer studies the examples and builds its own rules. When a new email arrives, it uses those rules to decide.

When spammers change their tricks, you do not rewrite anything. You collect new examples and let the computer learn again.

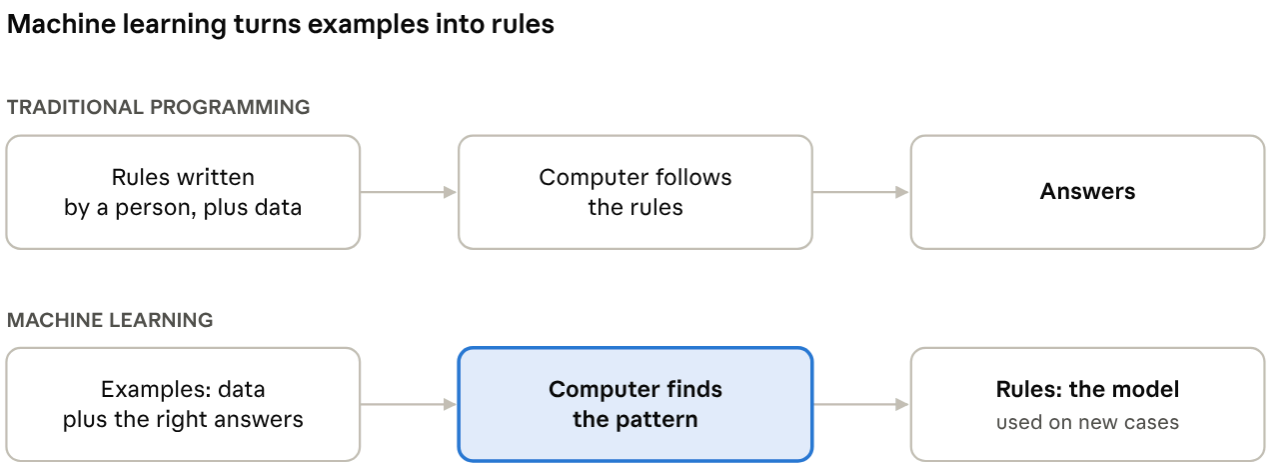


Figure 1.1 · Traditional programming versus machine learning

This is the most important picture in the book. In traditional programming, a person turns knowledge into rules. In machine learning, the computer turns examples into rules. Everything else is detail.

Teaching a child what a dog is

Think about how a small child learns the word “dog”. Nobody gives the child a definition. If you tried, you would struggle: “four legs, fur, a tail” also describes a cat, a horse and a cow.

Instead you point. “Look, a dog!” The child sees a big dog, a small dog, a fluffy dog and a dog in a picture book. Sometimes the child points at a cat and says “dog”, and you gently say “no, that is a cat”. After enough examples and corrections, the child can recognise a dog they have never seen before, of a breed they have never seen before.

That last step is the whole goal of machine learning. Learning the examples by heart is easy. The hard and useful part is doing well on **new** cases. This ability is called **generalisation**, and it will come up again and again in this book.

The words you need from day one

A few words appear on almost every page of machine learning. Learn them now and the rest gets easier.

- **Data** is the collection of examples you learn from. For Maya, it is her notebook of 700 days.
- An **example** (also called a **sample**, a **row** or an **observation**) is one item in the data. One day in Maya's notebook is one example.
- A **model** is what the computer builds from the data. It is a compact summary of the pattern, a little machine that takes in facts and gives out a guess.
- **Training** (also called **fitting** or **learning**) is the process of building the model from the data.
- A **prediction** is the model's guess for a new case. Using a trained model to make predictions is sometimes called **inference**.

Here is the whole cycle in one sentence. We **train** a **model** on **data**, then use the model to make **predictions** about new cases.

Maya's two approaches

Maya could run her shop with hand-written rules. "If it is above 25 degrees, prepare 200 ice creams. Otherwise prepare 100." This is traditional programming done in her head. It is simple, but it ignores weekends, rain, school holidays, and the slow drop in sales near closing season.

Or Maya could let her notebook teach her. A machine learning model can look at all 700 days at once. It can notice that hot weekends sell three times as much as cool weekdays, that rain cuts sales even on warm days, and that the first sunny day after a cold week brings a rush. Nobody had to write those rules down. They were hidden in the data.

Where you already meet machine learning

You probably use machine learning dozens of times a day without noticing.

Where you see it	What it learned from	What it predicts
Email spam filter	Millions of emails marked spam or not spam	Is this new email spam?
Phone face unlock	Many photos of your face from different angles	Is this face the owner?
Streaming recommendations	What millions of people watched and finished	Which show will you enjoy next?

Where you see it	What it learned from	What it predicts
Maps arrival time	Past trips, traffic speeds, time of day	How long will this trip take?
Bank fraud alert	Past card payments, some marked as fraud	Is this payment suspicious?
Keyboard auto-complete	Huge amounts of written text	Which word will you type next?
Photo search	Photos labelled with what they show	Which of your photos contain a beach?
Translation apps	The same texts written in two languages	What is this sentence in Spanish?
Chat assistants	Very large collections of text, plus human feedback	What is a helpful reply to this message?

Each row follows the same pattern. Somebody collected examples, a model learned from them, and now the model makes guesses about new cases.

What machine learning is good at, and what it is not

Machine learning is a powerful tool, but it is a tool, not magic. It shines in some situations and fails in others.

It is good at:

- Tasks where there is lots of past data with a real pattern in it.
- Tasks where the rules are too many or too fuzzy to write by hand, such as recognising faces or voices.
- Decisions that must be made millions of times, quickly and the same way each time.
- Tasks where the world changes and the rules must be updated often, such as spam.

It is poor at:

- Tasks with very few examples. A model that has seen ten cases has learned almost nothing.
- Situations unlike anything in the past. A model trained on summer days knows nothing about a snowstorm.
- Questions of values. A model can predict who will repay a loan; it cannot tell you what is fair.
- Explaining why. Most models find patterns, not causes. Ice-cream sales and sunburn rise together, but ice cream does not cause sunburn. Hot weather causes both.

A good rule of thumb: machine learning can only find patterns that are really in the data. If the data is poor, the model will be poor. People in the field say **garbage in, garbage out**.

AI, machine learning and deep learning

Three terms are often mixed up in the news. They fit inside each other like boxes inside boxes.

Deep learning sits inside machine learning, which sits inside AI

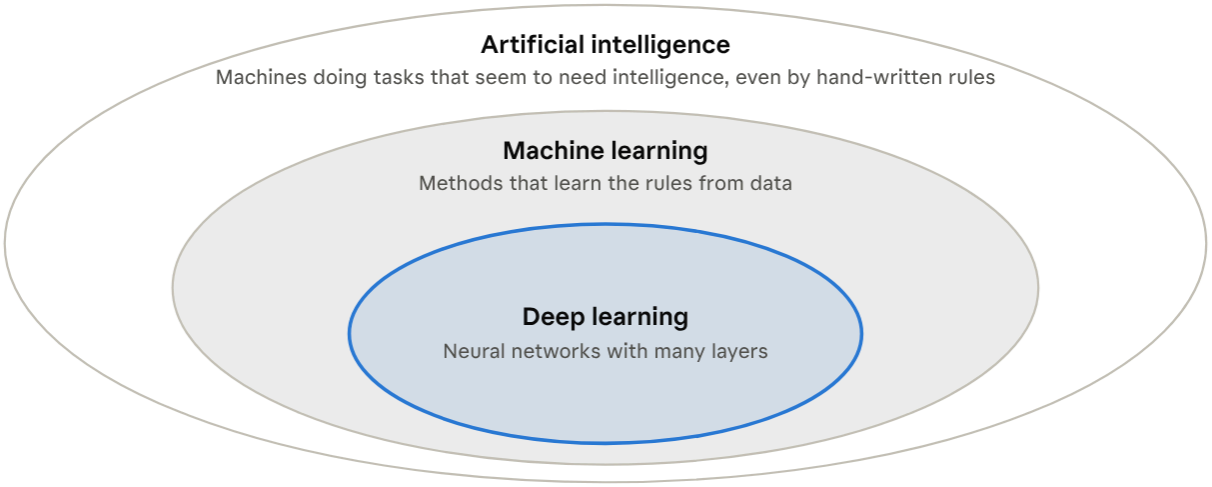


Figure 1.2 · How AI, machine learning and deep learning fit together

Artificial intelligence (AI) is the biggest box. It means any way of making machines do things that seem to need intelligence, such as playing chess, understanding speech or planning a route. Some early AI was built from hand-written rules, with no learning at all.

Machine learning is a large box inside AI. It covers the methods that learn from data. Almost all of today's successful AI uses machine learning.

Deep learning is a box inside machine learning. It uses a particular kind of model, called a **neural network**, with many layers. Deep learning powers image recognition, speech recognition and chatbots. You will meet it in Chapters 17 and 18.

A very short history

Machine learning is older than most people think. The ideas are decades old; what changed recently is the amount of data and computing power.

Year	What happened	Why it matters
1958	Frank Rosenblatt builds the perceptron	One of the first machines that learned from examples; the ancestor of neural networks

Year	What happened	Why it matters
1959	Arthur Samuel uses the phrase “machine learning”	His checkers program improved by playing against itself
1970s–1980s	“AI winters”	Big promises were not met, and funding dried up twice
1986	Backpropagation becomes widely known	A practical way to train neural networks with several layers
1990s	The internet starts producing huge amounts of data	Learning from data becomes practical for real businesses
2001	Leo Breiman publishes random forests	A strong, easy-to-use method that is still popular today
2012	A deep network wins the ImageNet picture contest by a wide margin	Starts the deep learning boom
2017	The transformer design is published	The building block of today’s language models
2022	Chat assistants reach the general public	Hundreds of millions of people use machine learning directly

The machine learning recipe

Almost every machine learning project, big or small, follows the same seven steps. The rest of this book walks through each of them.

1. **Ask a clear question.** What exactly do you want to predict, and how will the prediction be used? Maya wants to predict tomorrow’s sales so she can prepare the right amount.
2. **Collect data.** Gather past examples where you know the answer. Maya has her notebook.
3. **Prepare the data.** Fix mistakes, fill gaps and put it in a form the computer can use (Chapters 3 and 4).
4. **Choose a model.** Pick a kind of model that suits the question (Part 2).
5. **Train the model.** Let the computer learn the pattern from the examples (Chapter 16).
6. **Test the model honestly.** Check it on examples it has never seen (Part 3).
7. **Use it and keep watching.** Put the model to work, and check that it keeps working as the world changes.

In real projects, step 6 often sends you back to step 2 or 3. The model is weak, so you collect better data or prepare it differently. Machine learning is a loop, not a straight line.

Key ideas

- Traditional programming: a person writes the rules. Machine learning: the computer learns the rules from examples.
- A **model** is the learned summary of the pattern. **Training** builds it. A **prediction** uses it.
- The goal is **generalisation**: doing well on new cases, not just remembering old ones.
- Machine learning finds patterns, not causes, and it can only find patterns that are really in the data.
- Deep learning is a part of machine learning, which is a part of AI.

Try it yourself

1. Name three things you did today that probably involved machine learning. For each, say what the examples might have been.
2. A school wants to predict which students will need extra help in maths. Would you use hand-written rules or machine learning? What examples would you need?
3. Explain in your own words why learning examples by heart is not enough.
4. Ice-cream sales and drowning accidents both go up in summer. Does ice cream cause drowning? What is really going on?
5. Give one task where you think machine learning would be a bad idea, and explain why.

Chapter 2 — The Three Kinds of Learning

The big idea: Machine learning comes in three main kinds. In **supervised learning** the examples come with answers. In **unsupervised learning** they do not, and the computer looks for structure. In **reinforcement learning** the computer learns by trying things and getting rewards.

Three ways people learn

Think about three ways you have learned things in your own life.

At school, a teacher gave you practice questions and, later, the correct answers. You compared your answers to the answer key and improved. That is **supervised learning**: learning with a teacher who knows the answers.

As a child, you may have sorted a big box of toys into piles. Nobody told you what the piles should be. You put cars with cars and blocks with blocks because they looked alike. That is **unsupervised learning**: finding groups and patterns with no answer key.

When you learned to ride a bicycle, nobody could give you the answer. You tried, you wobbled, you fell, and slowly you found what worked. Staying up felt good; falling hurt. That is **reinforcement learning**: learning from the results of your own actions.

Computers learn in the same three ways.

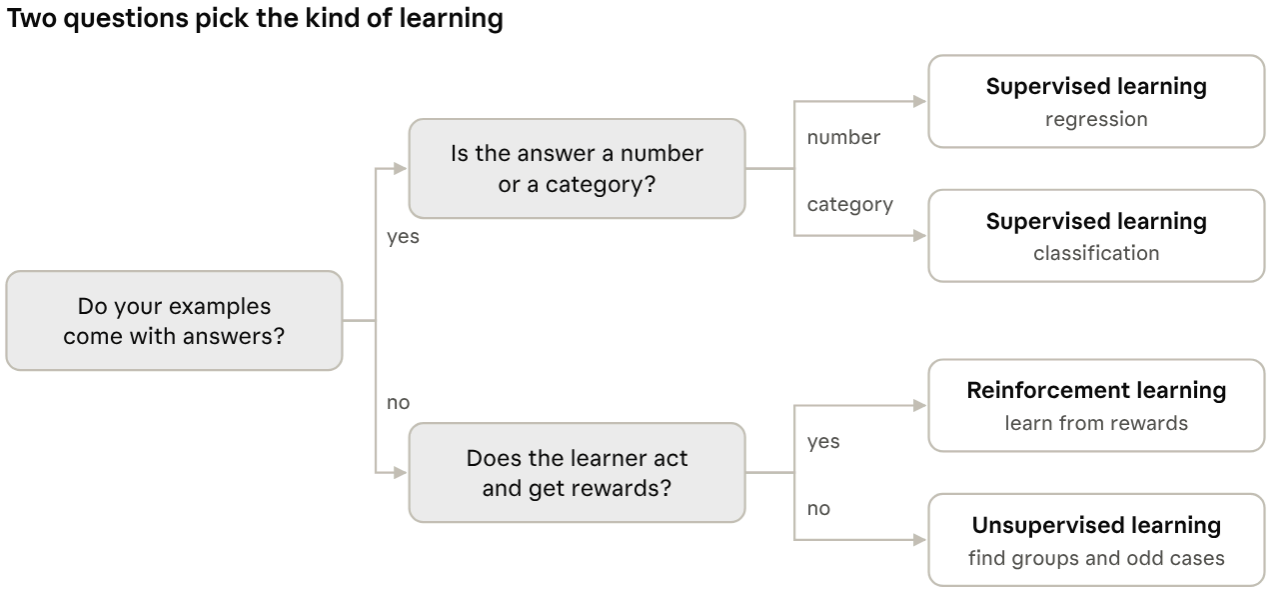


Figure 2.1 · Choosing the kind of learning

Two questions decide which kind you need. First, do your examples come with the answers? Second, if not, is the learner acting in a world and getting rewards?

Supervised learning: learning with an answer key

In supervised learning, every example in the data comes with the answer we want the model to learn. The answer is called the **label** or the **target**. The facts we use to predict it are called the **features** (Chapter 3 explains these words in detail).

For Maya, one example is one day. The features are the temperature, the rain, and whether it was a weekend. The label is the number of ice creams sold that day. The model studies hundreds of days where the label is known. Later, given tomorrow's weather forecast, it predicts tomorrow's sales.

Supervised learning is by far the most common kind in real businesses. It comes in two flavours, depending on what kind of answer you want.

Regression means the answer is a number on a scale. How many ice creams will sell? What will this house sell for? How many minutes will this trip take? The prediction can be 143, or 143.7, or anything in between.

Classification means the answer is a category, chosen from a fixed list. Is this email spam or not spam? Is this photo a cat, a dog or a rabbit? Will this customer come back: yes or no? When there are only two categories, it is called **binary classification**. When there are more, it is called **multi-class classification**.

A simple way to tell them apart: if it makes sense to say the prediction was “a little bit off”, it is regression. Predicting 140 ice creams when 145 sold is a little off. If the answer is either right or wrong, it is classification. Calling a cat a dog is just wrong.

Question	Kind	Why
How many ice creams will sell tomorrow?	Regression	The answer is a number on a scale
Will tomorrow be a busy day?	Binary classification	Two categories: busy or quiet
What price will this flat sell for?	Regression	A number of dollars or euros
Which digit, 0 to 9, is in this picture?	Multi-class classification	Ten categories
What temperature will it be at noon?	Regression	A number of degrees
Is this bank payment fraud?	Binary classification	Fraud or not fraud
Which language is this sentence in?	Multi-class classification	One of many languages

Unsupervised learning: finding structure with no answers

Sometimes there is no answer key. You have plenty of data, but nobody has labelled it. Unsupervised learning looks for structure that is already there.

Imagine a tin of mixed buttons. You might sort them by colour, by size or by number of holes. There is no single right answer. Different ways of grouping are useful for different jobs.

Unsupervised learning has three common jobs.

- **Clustering** finds groups of similar examples. Maya could group her regular customers into types: families who come on weekends, office workers who come at lunch, teenagers who come after school. Nobody told the computer these groups exist; it found them. Chapter 10 covers clustering.

- **Dimensionality reduction** squeezes many features down to a few, keeping as much of the important information as possible. It is like summarising a long book in one page. Chapter 11 covers it.
- **Anomaly detection** finds examples that do not fit any normal pattern. A bank can use it to flag a payment that looks unlike anything a customer has done before, even when there is no labelled fraud to learn from.

The hard part of unsupervised learning is judging the result. With no answer key, there is no simple score. A person usually has to look at the groups and decide whether they make sense and are useful.

Reinforcement learning: learning by trial and reward

Reinforcement learning is how you would train a dog. You do not explain what “sit” means. You wait until the dog sits, and you give it a treat. Over time, the dog learns that sitting when it hears “sit” leads to treats.

In reinforcement learning, the learner is called an **agent**. The world it acts in is called the **environment**. At each step, the agent looks at the situation, chooses an **action**, and receives a **reward**: a number that says how good the result was. The reward can be negative, like a penalty. The agent’s goal is to collect as much reward as possible over time.

The agent acts, the world responds, and the agent learns from the reward

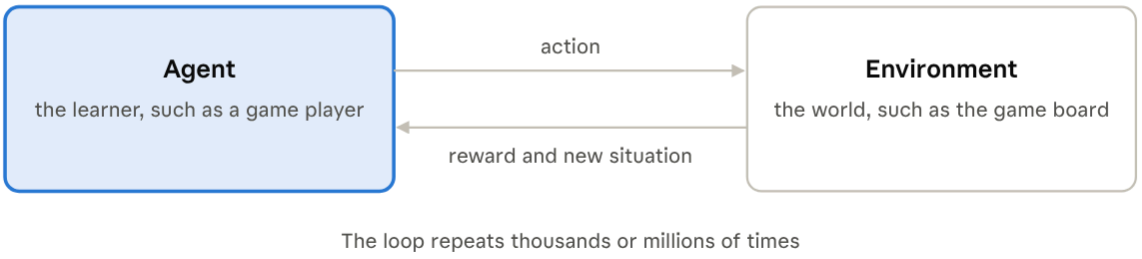


Figure 2.2 · The reinforcement learning loop

The agent repeats this loop thousands or millions of times. Early on, it acts almost at random. Slowly it learns which actions lead to good rewards in which situations.

There is a tricky balance here, called **exploration versus exploitation**. Should the agent keep doing what has worked so far (exploit), or try something new that might work even better (explore)? Think of choosing a restaurant. You can go back to your favourite, or try a new place that might be better or worse. A good learner does a bit of both.

Another difficulty is that rewards can come late. In chess, you only find out at the end whether you won. Which of your forty moves deserves the credit? Working this out is one of the central problems of reinforcement learning.

Reinforcement learning is famous for game-playing. In 2016, a program called AlphaGo beat one of the world's best players at the board game Go, a feat many experts had expected to take another decade. It is also used for controlling robots, managing energy use in buildings, and fine-tuning chat assistants using human feedback on which answers are better.

For most everyday business problems, though, reinforcement learning is not the right tool. It needs a world where the agent can try things safely and many times. Maya cannot try a thousand different prices on real customers just to see what happens.

Kinds in between

The three kinds are the main ones, but the borders are fuzzy. Two in-between kinds are worth knowing by name.

Semi-supervised learning uses a small amount of labelled data together with a large amount of unlabelled data. Labelling is often slow and expensive. A hospital might have a million scans but only a thousand that a doctor has marked. Semi-supervised methods make use of both.

Self-supervised learning creates labels from the data itself. Take a sentence, hide one word, and ask the model to guess the hidden word. The answer was in the sentence all along, so no person had to label anything. This trick lets a model learn from enormous amounts of text. It is how today's large language models learn the basics of language (Chapter 18).

Which kind do you need?

For any new problem, ask these questions in order.

1. **Do I know the right answer for my past examples?** If yes, use supervised learning.
2. **Is the answer a number or a category?** A number means regression. A category means classification.
3. **If I have no answers, what do I want to find?** Groups means clustering. A simpler summary means dimensionality reduction. Odd cases means anomaly detection.
4. **Does the learner make a series of decisions and get rewards?** Then reinforcement learning may fit, if it can practise safely.

Most of this book focuses on supervised learning, because it is the most common and the easiest to check. The ideas you learn there, especially in Part 3, carry over to the other kinds.

Key ideas

- **Supervised learning** learns from examples that come with answers (labels).
- **Regression** predicts a number. **Classification** predicts a category.
- **Unsupervised learning** finds structure in data without answers: groups, summaries and odd cases.
- **Reinforcement learning** learns by acting, getting rewards, and improving over many tries.
- **Self-supervised learning** makes its own labels from the data, such as hiding a word and guessing it.

Try it yourself

For each task, say which kind of learning fits best. If it is supervised, say whether it is regression or classification.

1. Predicting tomorrow's rainfall in millimetres from today's weather.
2. Grouping news articles into topics when nobody has labelled the topics.
3. Teaching a robot arm to stack blocks by rewarding it each time a tower stands.
4. Deciding whether an X-ray shows a broken bone, using X-rays that doctors have already marked.
5. Finding unusual temperature readings in a factory machine that might mean it is about to break.
6. Predicting which of five ice-cream flavours a customer will order, using past orders.

Chapter 3 — Data, the Fuel

The big idea: Machine learning runs on data the way a car runs on fuel. Most data can be pictured as a table: each row is one example, each column is one fact about it, and one special column holds the answer we want to predict.

Maya's notebook as a table

Here are eight days from Maya's notebook, written as a table. A table like this is the most common shape of data in machine learning.

Date	Day	Temperature (°C)	Rain	School holiday	Ice creams sold
3 June	Monday	19	Yes	No	41
4 June	Tuesday	22	No	No	88
5 June	Wednesday	24	No	No	104

Date	Day	Temperature (°C)	Rain	School holiday	Ice creams sold
6 June	Thursday	21	Yes	No	52
7 June	Friday	26	No	No	139
8 June	Saturday	28	No	No	231
9 June	Sunday	27	No	No	219
24 July	Wednesday	29	No	Yes	247

Each **row** is one day: one example. Each **column** is one fact about that day. The last column, ice creams sold, is the thing Maya wants to predict. That is the **label** (or **target**). The other columns are the facts she can know in advance and use to make the prediction. Those are the **features**.

Features go by many names in different books: **inputs**, **variables**, **attributes**, **predictors** or **columns**. The label is also called the **output**, the **response** or the **dependent variable**. They all mean the same two ideas: what you know, and what you want to guess.

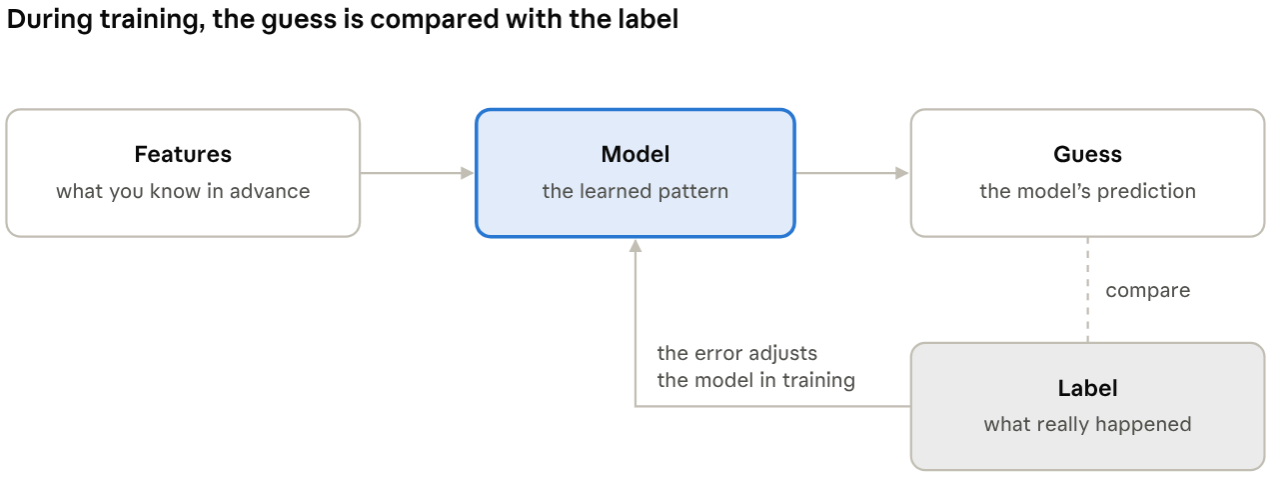


Figure 3.1 · Features go in, a guess comes out, and training compares it with the label

During training, the model sees both the features and the label for each past day. It adjusts itself so its guesses get closer to the labels. After training, it sees only the features of a new day, and its guess is the prediction.

Looking at the data first

Before building any model, look at your data. The simplest way is a picture. Here is temperature plotted against ice creams sold for 24 days.

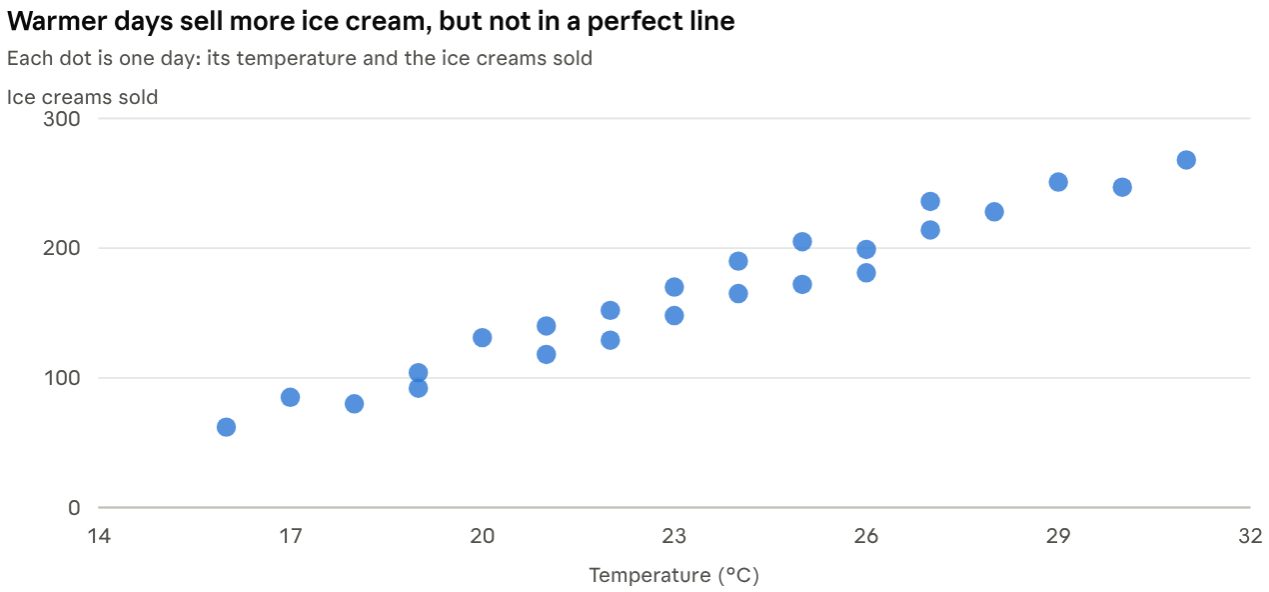


Figure 3.2 · Made-up example data: 24 days from Maya's notebook

Even without a model, you can see the pattern: warmer days sell more. You can also see that the dots do not fall on a perfect line. Two days at the same temperature can sell quite different amounts. Some of that spread comes from other features, such as rain and weekends. Some is just the everyday randomness of life, which no model can predict.

That randomness has a name: **noise**. Every real data set has noise. A model's job is to capture the pattern and ignore the noise. Telling the two apart is one of the central challenges of machine learning (Chapter 14).

Kinds of features

Not all columns are alike. The kind of feature affects how you prepare it and which models can use it.

Numerical features are numbers where the size means something. Temperature is numerical: 28 degrees is warmer than 19. Some numerical features can take any value, such as weight or temperature. Others are counts, such as the number of staff working.

Categorical features are labels from a fixed list. Day of the week is categorical: Monday, Tuesday and so on. Ice-cream flavour is categorical. You cannot add or average categories; "Monday plus Wednesday" means nothing.

Some categories have a natural order, such as small, medium and large cup sizes, or a rating of poor, fair, good and excellent. These are called **ordinal** features. Categories with no order, such as flavours or colours, are called **nominal** features.

Yes/no features (also called **binary** or **boolean** features) have just two values. Rain and school holiday are yes/no features. They are a special case of categorical features, and computers usually store them as 1 for yes and 0 for no.

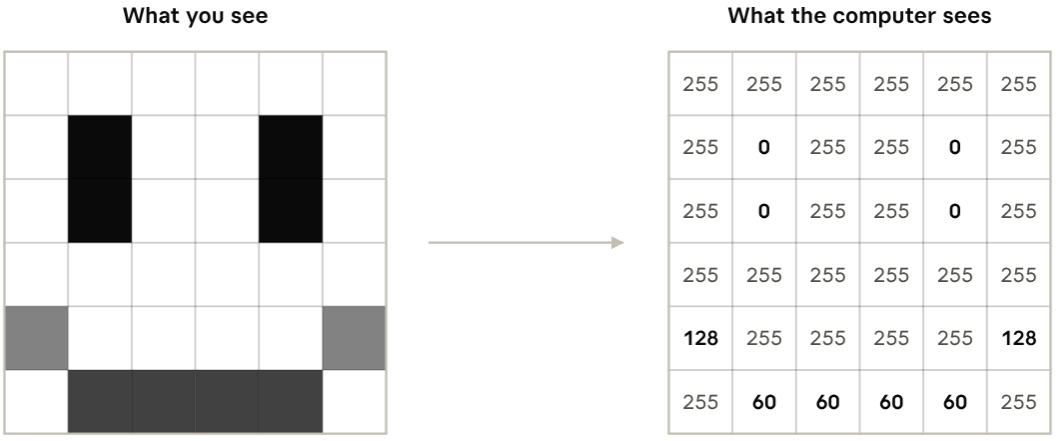
Dates and times look like numbers but are best broken into parts that carry meaning: day of the week, month, hour of the day, whether it is a public holiday. Maya's date column on its own is not very useful. "Saturday in July" is extremely useful.

Data that is not a table

Some of the most exciting machine learning works on data that does not start as a neat table. This is called **unstructured data**, as opposed to the **structured data** of rows and columns.

Images are grids of tiny coloured squares called **pixels**. To a computer, each pixel is just a number for how bright it is. A colour pixel is three numbers: how much red, green and blue it contains.

A computer sees a picture as a grid of brightness numbers



Each square is one pixel: 0 is black, 255 is white, and numbers in between are greys

Figure 3.3 · A tiny 6 by 6 picture and the 36 numbers that store it

A small black-and-white picture of a handwritten digit, 28 pixels wide and 28 pixels tall, is 784 numbers. A phone photo is millions of numbers. Image models learn which patterns of numbers mean "cat" or "stop sign".

Text must also be turned into numbers. The simplest way is to count words: how many times does "free" appear in this email? Modern methods turn each word into a list of numbers that captures its meaning, so that "happy" and "glad" end up with similar lists (Chapter 18).

Sound is a wave, measured thousands of times per second, so it too becomes a long list of numbers. Speech recognition models learn from these lists.

The lesson is simple. Whatever the data looks like at first, the computer always works with numbers in the end. Part of every project is turning the raw data into numbers that keep the important information.

How much data do you need?

This is one of the most common questions, and the honest answer is “it depends”. It depends on how complicated the pattern is, how noisy the data is, and which model you use. Still, a few rules of thumb help.

- **Simple patterns need less data.** Learning that hot days sell more ice cream might need a few dozen days. Learning to recognise faces needs many thousands of pictures.
- **More features need more examples.** Each extra feature is one more thing the model must learn about. A common rough guide for simple models is at least ten examples for every feature, and more is better.
- **Every category needs its own examples.** If Maya wants to predict sales on public holidays, she needs enough holidays in her notebook. Five holidays is too few to learn much.
- **Deep learning is hungry.** Neural networks usually need thousands to millions of examples, which is one reason they took off only when the internet produced huge data sets.

Quality beats quantity

A million bad examples can be worse than a thousand good ones. Data can go wrong in many ways, and each one quietly harms the model.

Problem	Example from Maya’s notebook	What it does to a model
Mistakes	Sales written as 1,040 instead of 104	One wild value can drag predictions badly off
Missing values	No temperature written on a busy day	The model must guess or skip that day
Mixed units	Some temperatures in °F, most in °C	80 °F looks like a heatwave in °C terms
Inconsistent labels	“Rain”, “rain”, “R” and “wet” all used	The computer sees four different values
Out-of-date data	Notes from before the shop moved to a new street	Old patterns may no longer hold
Unrepresentative data	Notes kept only in summer	The model knows nothing about winter

The last problem is the most dangerous because it is invisible. Every single row can be correct, and the data can still mislead. If Maya only took notes on sunny days, a model would never learn what rain does. This is called **sampling bias**: the examples you collected are not a fair picture of the cases you will predict.

Where data comes from

Data can come from many places.

- **Records you already keep:** sales, bookings, website clicks, customer support tickets.
- **Sensors:** thermometers, cameras, phone location, machines in a factory.
- **Surveys and forms:** questions people answer on purpose.
- **Public data sets:** governments, universities and research groups publish data for anyone to use. Well-known collections include the UCI Machine Learning Repository and the data sets shared on Kaggle.
- **Paid labelling:** for supervised learning, people are often paid to add labels, such as marking what is in each photo.

Labels are often the most expensive part. Taking a million photos is easy; having a person mark what is in each one takes a lot of time and money. That is one reason self-supervised learning (Chapter 2) is so valuable.

Making better features

Often the raw columns are not the most useful form of the information. Creating new, more helpful features from the raw data is called **feature engineering**. It is one of the most powerful skills in practical machine learning, and it relies on knowing the subject well.

Maya could add several new features to her notebook without collecting anything new:

- From the date: **day of the week**, **month**, and **is it a weekend**.
- From a holiday calendar: **is it a public holiday**, and **is it a school holiday**.
- From past rows: **yesterday's sales** and **the average sales over the last seven days**.
- From the weather: **is this the first warm day after a cold spell**.

Each of these turns something Maya already knows from experience into a column the computer can use. A simple model with good features usually beats a clever model with poor features.

One warning. A feature must be something you will actually know at the moment you make the prediction. Maya will know tomorrow's forecast tonight, but she will not know tomorrow's actual sales. Using information from the future by accident is a serious trap called **data leakage**, covered in Chapter 15.

Key ideas

- Most data is a table: rows are **examples**, columns are **features**, and one column is the **label** to predict.
- Features can be **numerical**, **categorical** (nominal or ordinal), **yes/no**, or dates and times.
- Images, text and sound are **unstructured data**; they are turned into numbers before a model can use them.
- Data always contains **noise**: random variation no model can predict.
- Quality beats quantity. Watch for mistakes, gaps, mixed units and **sampling bias**.
- **Feature engineering** creates more useful columns from raw data, using knowledge of the subject.

Try it yourself

1. A used-car website wants to predict car prices. List five features it could use and the label. Say whether each feature is numerical or categorical.
2. Is a hotel star rating (one to five stars) nominal or ordinal? Why?
3. A 10 by 10 pixel colour image has how many numbers in it?
4. A gym only surveys members who visit on Monday mornings. What problem might this cause if it wants to understand all its members?
5. Suggest two new features Maya could create from her date column that might help predict sales.

Chapter 4 — Getting Data Ready

The big idea: Real data is messy. Before a model can learn from it, you must check it, fix mistakes, fill in gaps, turn words into numbers, and put numbers on comparable scales. This work is called **data preparation** (or **preprocessing**), and it often takes more time than everything else in a project.

Washing the vegetables first

A good cook does not throw unwashed vegetables straight into the pan. First they wash, peel and chop. Chefs call this preparation **mise en place**, “everything in its place”. It is not glamorous, but a meal is only as good as its ingredients.

Data preparation is the same. A clever model fed with dirty data gives poor results. Many experienced practitioners say that most of their time goes into preparing data, not into choosing models. This chapter walks through the main steps in the order you usually do them.

Step 1: Get to know your data

Before changing anything, look. For each column, ask a few simple questions.

- What kind of column is it: numerical, categorical, yes/no, date?
- For numbers: what are the smallest, largest and typical values?
- For categories: what values appear, and how often?
- How many values are missing?

These checks catch surprising problems. Suppose Maya's temperature column has a smallest value of 8 and a largest of 250. Nobody sells ice cream at 250 degrees. Probably someone wrote 25.0 and lost the dot. Without looking, you would never know.

A picture often helps more than numbers. A simple chart of each column, like the scatter plot in Chapter 3, shows strange values at a glance.

Step 2: Fix mistakes and inconsistencies

Next, clean up what is clearly wrong or untidy.

- **Impossible values.** A temperature of 250 °C, a negative age, a date in the year 2091. Fix them if you can tell what was meant; otherwise treat them as missing.
- **Different spellings of the same thing.** "Yes", "yes", "Y" and "true" should become one value. So should "New York", "NY" and "new york".
- **Mixed units.** Convert all temperatures to one unit, all prices to one currency, all weights to kilograms.
- **Duplicates.** If the same day was entered twice, keep one copy. Duplicates make the model think that day matters twice as much.

Step 3: Deal with missing values

Almost every real data set has gaps. Maya forgot to write the temperature on some days. Most models cannot work with a blank cell, so you must decide what to do. There are several common choices.

Choice	How it works	Good when	Risk
Drop the row	Remove any example with a gap	Only a few rows have gaps	You throw away good information in the other columns
Drop the column	Remove a feature that is mostly empty	The column is missing most of the time	You lose a feature that might matter

Choice	How it works	Good when	Risk
Fill with a typical value	Use the middle value (the **median**) for numbers, the most common value for categories	Gaps are scattered and few	Every filled day looks average, which may be wrong
Fill from neighbours	Use yesterday's or a similar example's value	Values change slowly, like temperature	Wrong if things changed suddenly
Add a "was missing" flag	Fill the gap and add a yes/no column saying it was empty	The gap itself might carry meaning	One more column to handle

The last idea deserves a closer look. Sometimes the fact that a value is missing tells you something. Suppose Maya's thermometer often broke on the hottest days, when it sat in the sun. Then "temperature missing" secretly means "probably very hot". Filling those gaps with an average temperature would hide that signal. A "was missing" flag keeps it.

So always ask: **why** is this value missing? The answer guides the choice.

Step 4: Look hard at unusual values

An **outlier** is a value far from the others. Maya sold about 100 to 250 ice creams on most days, but one day she sold 640. What should she do with that day?

It depends on why it happened.

- If it was a **mistake**, such as 64 written as 640, fix it or remove it.
- If it was **real but special**, such as the day of a big beach festival, you have a choice. If festivals will happen again, keep the day and add a feature such as "festival day". If it was a one-off, you might leave it out so it does not confuse the model about normal days.

Never remove outliers automatically just because they are big. Rare events are sometimes exactly what you most need to predict, such as fraud, floods or machine failures.

Step 5: Turn categories into numbers

Most models only understand numbers. So categorical columns must be converted. The obvious idea is to number the categories: Monday = 1, Tuesday = 2, up to Sunday = 7. But this hides a trap.

Numbers have size and order. To a model, 7 is bigger than 1, and the gap between 1 and 7 is six times the gap between 1 and 2. So the model may think Sunday is “seven times Monday”, or that Saturday is “between” Friday and Sunday in some meaningful amount. For days that is half true by accident; for flavours it is plain nonsense. Chocolate is not “bigger” than vanilla.

The standard fix is **one-hot encoding**. You replace one categorical column with several yes/no columns, one for each category. Each row has a 1 in the column for its category and 0 everywhere else. Only one column is “hot” at a time, hence the name.

One-hot encoding gives each flavour its own yes/no column

BEFORE: ONE CATEGORY COLUMN

Flavour
Vanilla
Chocolate
Mint
Vanilla

AFTER: ONE YES/NO COLUMN PER FLAVOUR

Vanilla	Chocolate	Mint
1	0	0
0	**1**	0
0	0	**1**
1	0	0

Each row has a single 1, in the column for its flavour; no flavour counts as bigger than another

Figure 4.1 · One-hot encoding of a flavour column

Now no category is bigger than another. Each is just present or absent.

One-hot encoding has a cost. A column with 5 categories becomes 5 columns. A column with 10,000 categories, such as postal codes, becomes 10,000 columns, mostly zeros. For columns like that, people group rare categories into “other”, or use more advanced methods.

For **ordinal** features, plain numbering is fine and even better, because the order is real. Small = 1, medium = 2, large = 3 keeps the information that large is bigger than medium.

Step 6: Put numbers on comparable scales

Imagine a model predicting house prices from two features: floor area, from 40 to 300 square metres, and number of bedrooms, from 1 to 5. Floor area changes by hundreds; bedrooms change by one or two. Some models will pay far more attention to floor area simply because its numbers are bigger, not because it matters more.

It is like comparing prices in two currencies without converting them. A price of 5,000 yen looks bigger than 40 dollars, but it is not. You must convert to a common scale before comparing.

Scaling puts all numerical features on a similar range. There are two common ways.

- **Min-max scaling** squeezes each feature into the range 0 to 1. The smallest value becomes 0, the largest becomes 1, and everything else falls in between in proportion. A 40 square metre flat becomes 0, a 300 square metre house becomes 1, and a 170 square metre house becomes 0.5.
- **Standardisation** describes each value by how far it is from the average, measured in "typical distances". A value of 0 means exactly average. A value of 1 means one typical distance above average. A value of −2 means two typical distances below. That typical distance is called the **standard deviation**.

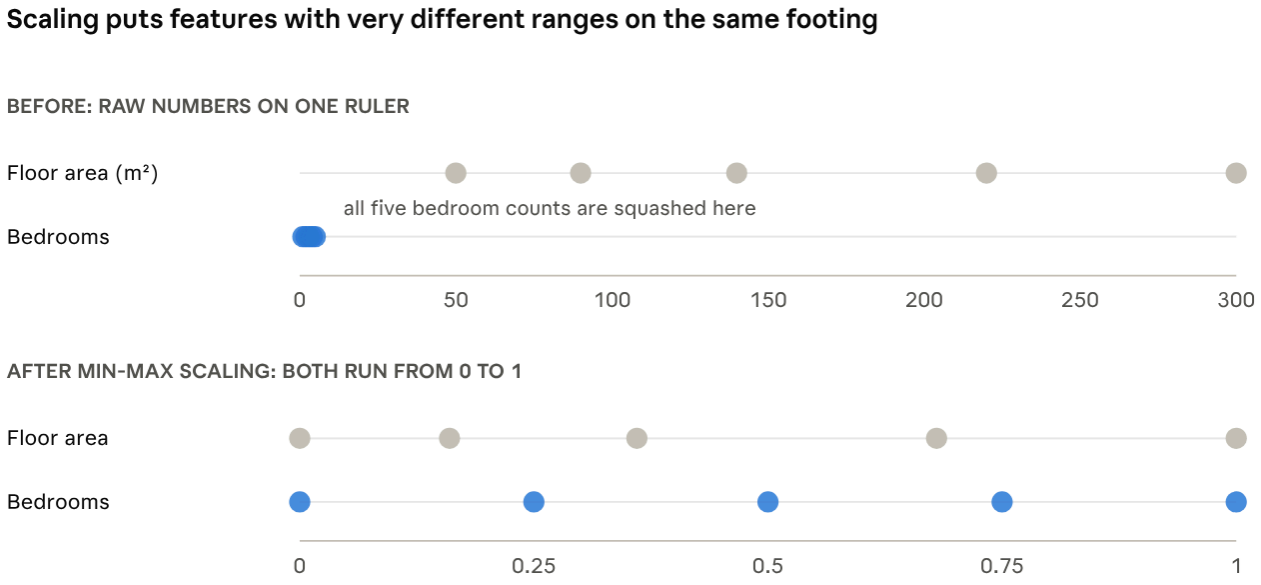


Figure 4.2 · Five example houses before and after min-max scaling

Which models need scaling? Models that measure distances between examples, such as k-nearest neighbours (Chapter 9), and models trained step by step, such as neural networks (Chapter 17), need it badly. Decision trees and the forests built from them (Chapters 7 and 8) do not care, because they only ask "is this value bigger or smaller than some cut-off?". When in doubt, scaling rarely hurts.

The golden rule: learn your preparation from training data only

Here is a subtle point that catches many beginners. Several preparation steps learn something from the data. Filling gaps with the median needs the median. Min-max scaling needs the smallest and largest values. Standardisation needs the average and the typical distance.

In Part 3 you will learn to split data into a part for training and a part for testing. The rule is: work out these numbers from the **training** part only, then apply the same numbers to the test part. If you compute the average from all the data, a little information about the test

examples sneaks into training. Your test score will look better than it should. This is a mild form of the data leakage you will meet in Chapter 15.

Keep a written recipe

Write down every preparation step, in order, as you go. Better still, write it as a program that can be run again. In machine learning software, a chain of preparation steps followed by a model is called a **pipeline**.

A pipeline has three big benefits. You can run exactly the same steps on new data when you make predictions. Someone else can check and repeat your work. And you avoid the golden-rule mistake, because the pipeline learns its numbers from whatever data you train it on.

Key ideas

- Look at your data before changing it: kinds of columns, ranges, counts and gaps.
- Fix impossible values, inconsistent spellings, mixed units and duplicates.
- For missing values, ask **why** they are missing before choosing how to fill them.
- Outliers can be mistakes or real rare events; never remove them automatically.
- Use **one-hot encoding** for categories without order; plain numbers are fine for ordinal categories.
- **Scale** numerical features for distance-based models and neural networks; trees do not need it.
- Learn preparation numbers from the training data only, and keep every step in a **pipeline**.

Try it yourself

1. A column for “country” contains “UK”, “United Kingdom”, “u.k.” and “Britain”. What should you do before training?
2. A survey has an “income” column that is often blank. Why might people with very high or very low incomes be more likely to leave it blank? How would you handle it?
3. One-hot encode a “cup size” column with values small, medium and large. Then explain why plain numbering might be better here.
4. A flat is 85 square metres. The smallest flat in the data is 40 and the largest is 300. What is its min-max scaled value? (Hint: how far along the way from 40 to 300 is 85?)
5. Why is it a mistake to compute the average for filling gaps using all the data, including the test data?

Part 2 — Your First Models

Chapter 5 — Drawing a Line: Linear Regression

The big idea: Linear regression predicts a number by drawing the straight line that best fits the data. It is the simplest useful model in machine learning, and many more advanced ideas are built on top of it.

A ruler on the scatter plot

Go back to the scatter plot of temperature and ice-cream sales in Chapter 3. Imagine laying a ruler across the dots so that it runs through the middle of the cloud. The ruler slopes upward, because warmer days sell more.

Now you can use the ruler to predict. Find tomorrow's forecast temperature on the bottom axis, go straight up to the ruler, and read across to the sales axis. That is linear regression. The rest of this chapter is about doing it precisely and knowing when to trust it.

What a line knows

A straight line is described by just two numbers.

- The **slope** says how steep the line is: how much the prediction rises for each step along the bottom axis. For Maya, it is the number of extra ice creams she sells for each extra degree.
- The **intercept** says where the line starts: its height when the feature is zero. It sets the overall level of the line.

For the 24 days in our example, the best line works out to roughly this rule:

Sales ≈ 14 × temperature − 161

Read it in plain words: "start at minus 161, then add 14 ice creams for every degree". On a 25-degree day the rule predicts 14 × 25 − 161 = 189, which rounds to about 190. The exact line, with all its decimals, predicts 187.

The slope is the interesting number. It says each extra degree brings about 14 more ice creams. That is useful knowledge on its own, even before predicting anything.

In machine learning these numbers are called the model's **parameters**. Training a linear regression simply means finding the best slope and intercept from the data.

What does “best” mean?

You could lay the ruler at many slightly different angles. Which line is best? To answer, we need a way to score a line.

For each day, measure the gap between the real sales and the line’s prediction. This gap is called the **error** or the **residual**. A dot above the line means the line guessed too low. A dot below means it guessed too high.

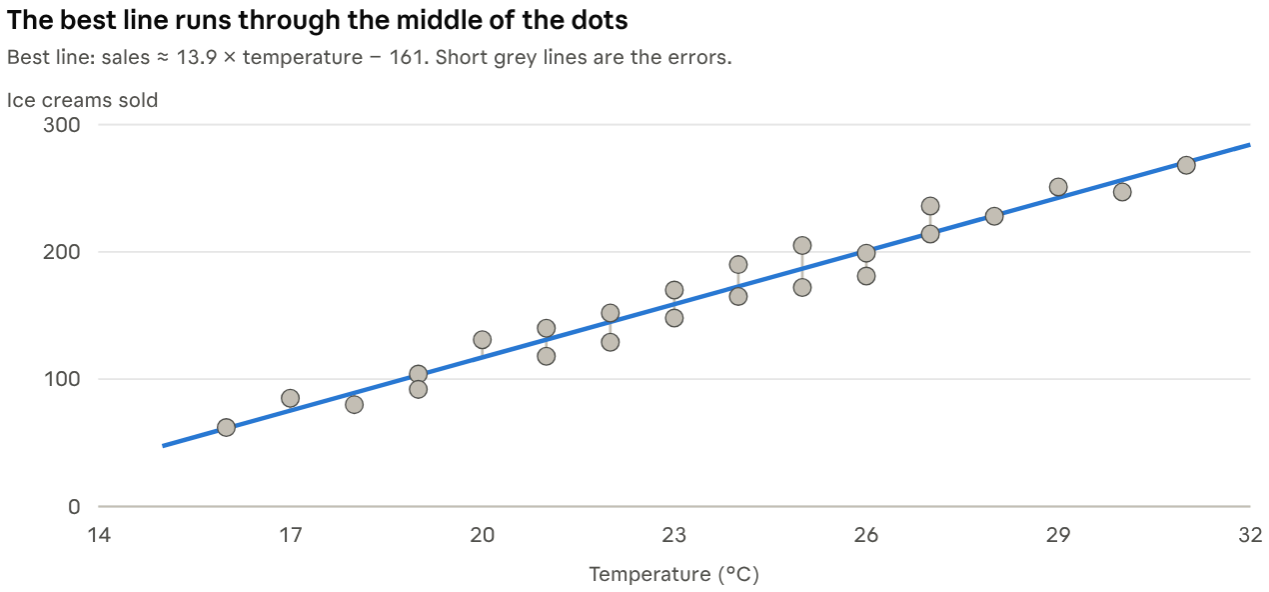


Figure 5.1 · The least-squares line through the 24 example days, with each day’s error

The short grey lines are the errors. A good line keeps them small overall. The standard way to combine them is:

1. Measure each error.
2. Square it, which means multiply it by itself. An error of 3 becomes 9, and an error of −3 also becomes 9.
3. Add up all the squared errors.

The best line is the one with the smallest total. This method is called **least squares**, and it has been used since the early 1800s.

Why square the errors instead of just adding them? There are two reasons. First, errors above and below the line would cancel out if you simply added them, so a terrible line could score zero. Squaring makes every error positive. Second, squaring punishes big misses much more than small ones. One miss of 10 costs 100, while two misses of 5 cost only 50 together. So least squares prefers a line that is a bit off for everyone over one that is badly off for a few.

You do not need to search for the best line by trial and error. For linear regression there is a mathematical shortcut that finds it exactly and instantly. Computers do it for you. For more complicated models, there is no shortcut, and we will need the step-by-step method of Chapter 16.

More than one feature

Temperature is not the only thing that matters. Rain and weekends matter too. **Multiple linear regression** gives each feature its own number, called a **weight** or **coefficient**. The prediction is the intercept plus each feature times its weight.

For Maya, a model with three features might learn something like this (the numbers are made up to show the idea):

- Start at −150.
- Add 13 for every degree of temperature.
- Subtract 55 if it is raining.
- Add 60 if it is a weekend.

On a 25-degree Saturday with no rain, the prediction is $-150 + 13 \times 25 + 60 = 235$. On a 25-degree rainy Tuesday, it is $-150 + 325 - 55 = 120$.

With one feature, the model is a line. With two features, it is a flat sheet tilted in space. With more, it is something we cannot picture, but the idea is the same: each feature pushes the prediction up or down by its own fixed amount.

Reading the weights

One big advantage of linear regression is that you can read it. Each weight tells a simple story: “rain costs about 55 sales”, “a weekend adds about 60”. A shop owner can understand and question that. Models you can read like this are called **interpretable**.

But read weights with care.

- **Weights depend on units.** A weight per degree Celsius is different from a weight per degree Fahrenheit. To compare how much different features matter, scale them first (Chapter 4).
- **Linked features share credit.** If temperature and “hours of sunshine” rise and fall together, the model may split the credit between them in odd ways. One weight might even turn negative. The predictions can still be fine, but the separate weights become hard to read.
- **A weight is not a cause.** A positive weight means the feature goes along with higher sales in this data. It does not prove that the feature causes them.

Straight lines go on forever

A line has no idea where your data ends. It simply keeps going. This leads to one of the most important warnings in machine learning.

Maya's data covers days from 16 to 31 degrees. Ask the line about a 5-degree winter day and it predicts about −90 ice creams. Negative ice creams are impossible. Ask about a 40-degree heatwave and it predicts nearly 400. In reality, on very hot days people may stay indoors, and sales may stop rising.

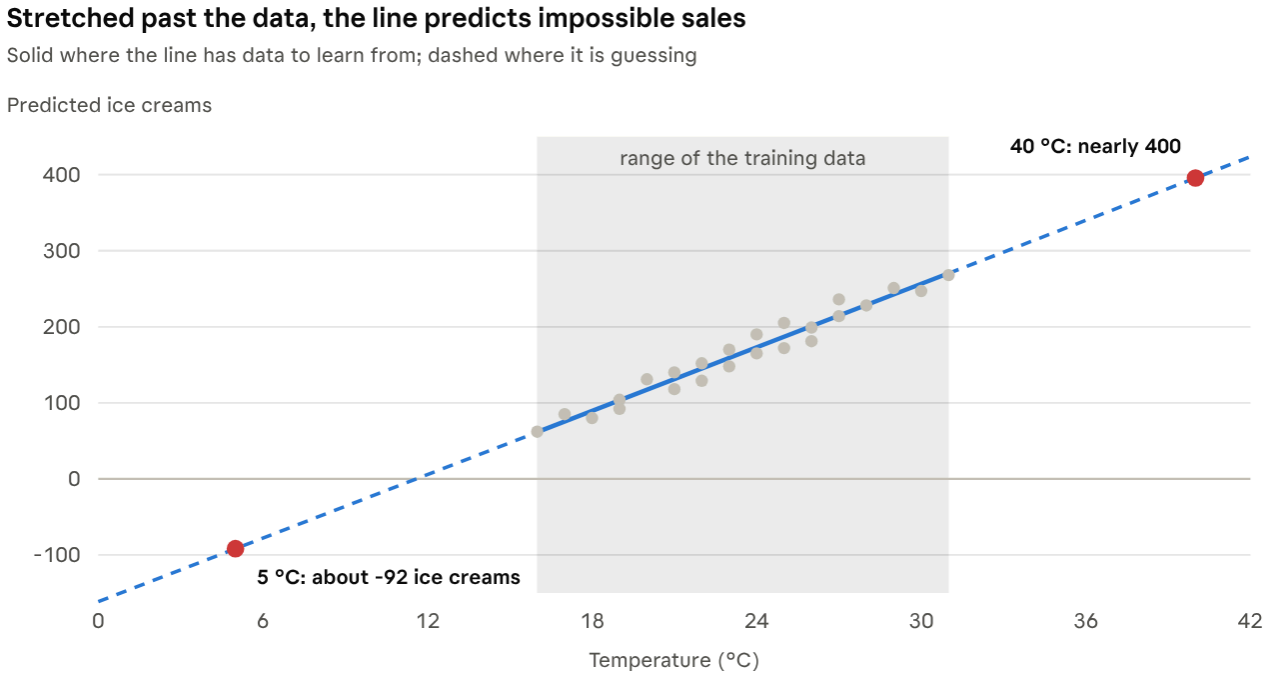


Figure 5.2 · The same line stretched from 0 to 42 °C

Predicting outside the range of the training data is called **extrapolation**. Every model is on shaky ground when it extrapolates, but straight lines are especially bold about it. Predicting inside the range of the data is called **interpolation**, and it is much safer.

The lesson: a model only knows the world it was shown. Always check whether a new case looks like the training data.

When the pattern is curved

Sometimes the real pattern is a curve. Sales might rise quickly from 15 to 25 degrees, then level off. A straight line cannot bend to follow that.

There are two simple fixes that keep the model linear in spirit.

- **Transform a feature.** For example, add a column for temperature multiplied by itself. Now the model can learn a gentle curve, because it can weigh both "temperature" and "temperature squared". This is called **polynomial regression**.

- **Cut a feature into bands.** Add yes/no columns such as “below 18 degrees”, “18 to 25” and “above 25”. Each band gets its own weight.

If the pattern is very bendy or depends on combinations of features, other models handle it more naturally. Decision trees (Chapter 7) and neural networks (Chapter 17) can follow curves without being told the shape in advance.

Strengths and weaknesses

	Linear regression
Strengths	Fast to train, even on huge data; easy to read and explain; works well with little data; a strong first model to compare others against
Weaknesses	Only learns straight-line patterns unless you add curved features; confused by strong outliers, because squaring makes them loud; extrapolates boldly and wrongly
Good first choice when	The relationship looks roughly straight in a scatter plot, or you need a model people can read

Keeping weights calm

When there are many features and not much data, linear regression can learn wild weights that fit the training data closely but predict new data poorly. A family of fixes called **regularisation** adds a small penalty for large weights, so the model prefers calm, modest weights unless the data strongly supports a big one. Two popular versions are called **ridge** and **lasso** regression. Lasso can even push useless weights all the way to zero, which removes those features. You will see why this helps in Chapter 14.

Key ideas

- **Linear regression** fits the straight line (or flat sheet) that best matches the data.
- A line is described by a **slope** and an **intercept**; with many features, each gets a **weight**.
- “Best” means the smallest total of squared errors: **least squares**.
- Weights are easy to read, but they depend on units and do not prove cause.
- Lines go on forever. Be very careful predicting outside the range of the training data (**extrapolation**).
- Add curved or banded features when the pattern bends, or use a more flexible model.

Try it yourself

1. Using the rule “sales ≈ 14 × temperature − 161”, predict sales on a 20-degree day and on a 30-degree day.
2. On one 22-degree day Maya sold 129 ice creams. What did the rule predict, and what was the error?
3. Why does least squares square the errors instead of just adding them up? Give both reasons.
4. A model predicts house prices with a weight of +2,000 per square metre and +15,000 per bedroom. Can you conclude that bedrooms matter more than floor area? Why or why not?
5. Maya opens a second shop in a much colder town where days rarely pass 15 degrees. Should she use her current model there? Explain.

Chapter 6 — Yes or No: Classification and Logistic Regression

The big idea: In **classification**, the model sorts each example into a category. **Logistic regression** is the simplest classifier: it gives a probability between 0 and 1 for a yes/no question, and a cut-off turns that probability into a decision.

Maya’s new question

Maya has a new problem. On busy days she needs a second person behind the counter. She must phone her helper the evening before. So she does not need to know exactly how many ice creams will sell. She only needs to know: **will tomorrow be busy?**

She decides that a busy day means more than 150 ice creams. Now every day in her notebook has a yes/no label: busy or quiet. This is a **binary classification** problem. The model must learn to answer yes or no from the features.

Why a straight line does not fit

We could write busy as 1 and quiet as 0, and try linear regression. But a straight line runs forever in both directions. On a very hot day it would predict 1.8, and on a cold day −0.6. Neither means anything. A yes/no question needs an answer between 0 and 1, ideally read as a probability.

The fix is to bend the line into an **S-shaped curve** that flattens out at 0 at one end and at 1 at the other. This curve is called the **logistic curve** or **sigmoid**. Logistic regression is linear regression passed through this S-curve.

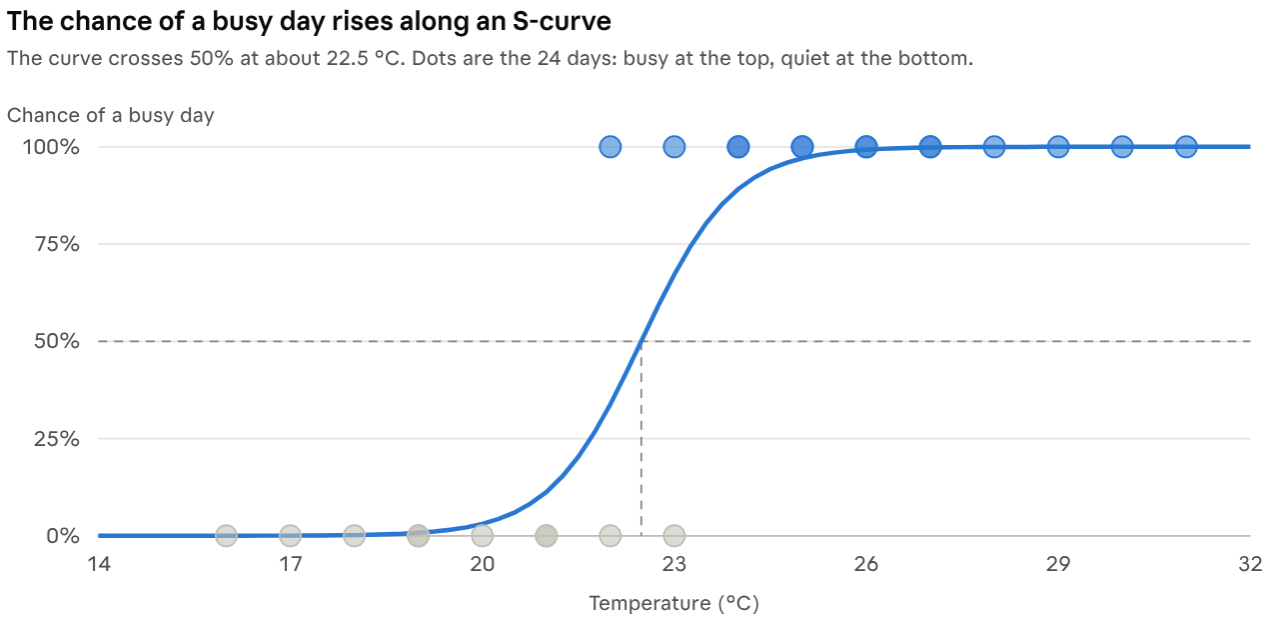


Figure 6.1 · A logistic curve fitted to the 24 example days (busy means more than 150 ice creams)

Read the curve from left to right. On cool days the chance of a busy day is close to zero. Around 21 degrees it starts to rise. At about 22.5 degrees it crosses 50%, the point of greatest doubt. By 25 degrees it is above 95%. On hot days it flattens out near 100%.

The dots show the real days, at the top if they were busy and at the bottom if they were quiet. Notice the overlap near 22 and 23 degrees. On those days, temperature alone cannot settle the question, and the curve honestly says so with probabilities in the middle.

What a probability means

When the model says “67% chance of a busy day”, it means something like: out of many days that look like this one, about two in three were busy. It does not mean the day will be 67% busy. Each day is either busy or not; the probability describes how sure the model is.

This is the same as a weather forecast. “70% chance of rain” does not mean it will rain 70% of the time. It means that on days like this, it rained about seven times out of ten.

A model whose probabilities match reality in this way is called **well calibrated**. If a well-calibrated model says 70% on a hundred different days, about seventy of them turn out busy. Not every model is well calibrated, and checking it is part of judging a classifier.

From probability to decision: the threshold

Maya still has to make a yes/no choice: phone the helper or not. The model gives a probability, so she needs a rule to turn it into a decision. The rule is a cut-off called the **threshold**. If the probability is above the threshold, predict “busy”.

The usual default is 50%. But the best threshold depends on what each kind of mistake costs.

- If Maya calls her helper and the day is quiet, she pays a few hours of wages for nothing. That is a small cost.
- If she does not call and the day is busy, the queue grows long, customers walk away, and she loses many sales. That is a big cost.

Since missing a busy day is much worse, Maya should call her helper even when the chance is only 30% or so. Lowering the threshold means more false alarms, but fewer missed busy days.

The same reasoning matters everywhere. A test for a serious disease uses a low threshold, because missing a sick patient is far worse than a false alarm that leads to a second test. A spam filter uses a high threshold, because throwing a real email into the spam folder is worse than letting a little spam through. The model gives the probability; **people choose the threshold** based on the costs. Chapter 13 shows how to measure these trade-offs.

Two features and the decision boundary

With one feature, the threshold is a single cut-off temperature. With two features, it becomes a line that splits the picture into two regions. That line is called the **decision boundary**. Everything on one side is predicted “yes”, everything on the other side “no”.

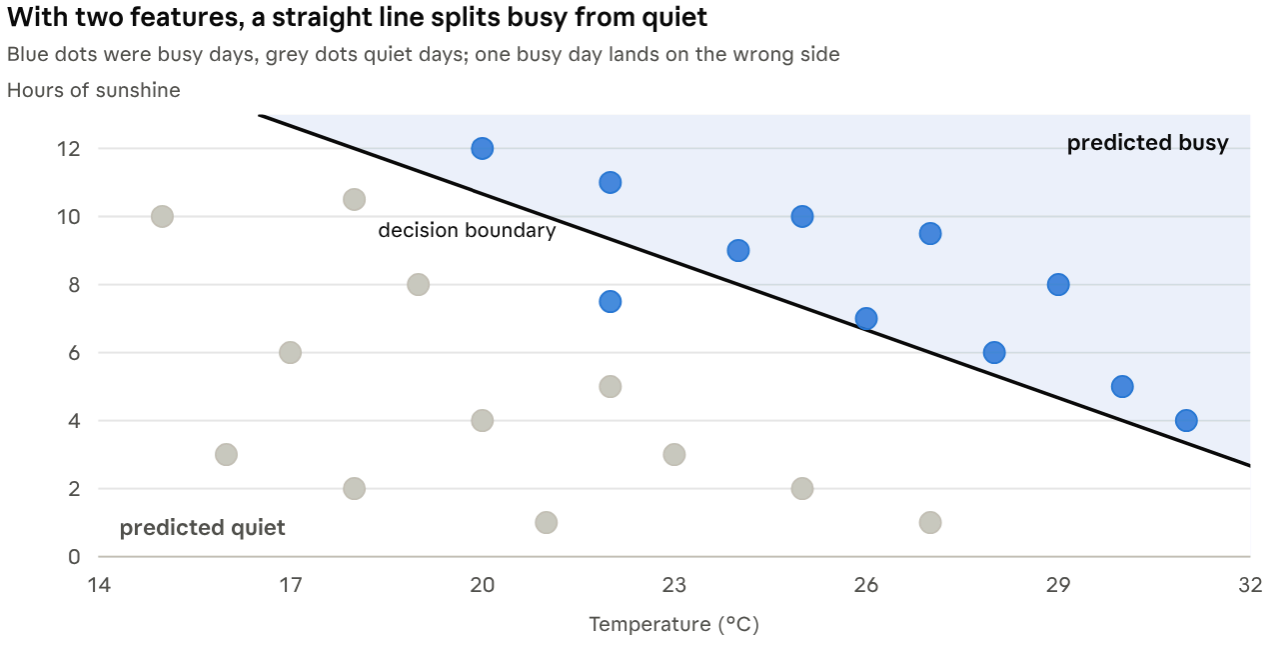


Figure 6.2 · Made-up example: a straight decision boundary on two features

Logistic regression always draws a straight decision boundary (a flat one, with more features). This is its strength and its limit. It is simple and hard to fool with noise, but it cannot separate classes that are tangled in curvy shapes. For those, trees (Chapter 7), nearest neighbours (Chapter 9) or neural networks (Chapter 17) can draw curved boundaries.

How logistic regression learns

Linear regression scores a line with squared errors. Logistic regression uses a different score that fits probabilities better. It is called **log loss** (or **cross-entropy**), and the idea is easy to state in words.

- If the model is confident and right, the penalty is tiny.
- If the model is unsure, the penalty is moderate.
- If the model is confident and **wrong**, the penalty is huge.

Think of a weather forecaster. One who says “60% chance of rain” on a dry day is a little off. One who says “100% certain it will stay dry” and then it pours has lost a lot of trust. Log loss punishes overconfidence hard. Training finds the weights with the smallest total log loss over all the examples, using the step-by-step method of Chapter 16.

More than two categories

What if Maya wants to predict which of four flavours a customer will choose: vanilla, chocolate, strawberry or mint? This is **multi-class classification**. There are two common approaches.

- **One-versus-rest:** train one yes/no model per flavour. “Vanilla or not?”, “Chocolate or not?” and so on. Pick the flavour whose model is most confident.
- **Softmax:** train one model that gives a probability for every flavour at once, with the probabilities adding up to 100%. For example: vanilla 45%, chocolate 30%, strawberry 15%, mint 10%.

Softmax is the standard approach in neural networks. You will meet it again in Chapter 17, where it forms the last step of many image and text models.

A confusing name

Despite its name, logistic regression is used for **classification**, not regression. The name comes from history: inside, it computes a number like linear regression does, then squeezes it through the logistic curve. Do not let the name trip you up.

Strengths and weaknesses

	Logistic regression
Strengths	Fast; easy to read, since each weight says whether a feature pushes towards yes or no; gives probabilities, not just labels; hard to fool with small data; an excellent first model for any yes/no problem
Weaknesses	Only draws straight boundaries unless you add curved features; can be thrown off by features that are strongly linked to each other; less accurate than flexible models when the pattern is complicated
Good first choice when	You need a yes/no answer with a probability, and you want a model you can explain

Key ideas

- **Classification** sorts examples into categories; **binary** means two categories.
- **Logistic regression** bends a straight line into an **S-curve** so its output is a probability between 0 and 1.
- A probability is how sure the model is, like a weather forecast. A **well-calibrated** model's 70% comes true about 70% of the time.
- A **threshold** turns probabilities into decisions. Choose it based on the cost of each kind of mistake, not by habit.
- With several features, the cut is a **decision boundary**; for logistic regression it is straight.
- **Log loss** trains the model by punishing confident wrong answers most.

Try it yourself

1. Name three everyday yes/no questions a classifier could answer, and the features you might use.
2. Using Figure 6.1, roughly what is the chance of a busy day at 22 degrees? At 24 degrees?
3. A fire alarm uses a model to decide whether there is smoke. Should its threshold be high or low? Why?
4. A model predicts "90% chance this email is spam" for 200 emails. If it is well calibrated, about how many of them are really spam?

5. Why would a model that says “100% sure” and is wrong get a much bigger penalty than one that says “55% sure” and is wrong?

Chapter 7 — Twenty Questions: Decision Trees

The big idea: A **decision tree** makes a prediction by asking a short series of yes/no questions about the features. Each answer leads down a branch to the next question, until you reach an end point that gives the prediction.

The game of twenty questions

You may know the game. One player thinks of something, and the other asks yes/no questions to guess it. “Is it alive?” “Is it bigger than a cat?” “Does it live in water?” Good players ask questions that split the possibilities roughly in half each time. Bad players ask “Is it a giraffe?” on their first turn.

A decision tree plays this game with data. It learns which questions to ask, in which order, to reach a good prediction as quickly as possible.

You have also met decision trees in daily life. A doctor’s checklist (“Does the patient have a fever? If yes, is there a rash?”) is a decision tree. So is a phone menu: “Press 1 for billing, press 2 for support.”

A tree for Maya

Here is a small tree that a computer might learn from Maya’s notebook to predict sales.

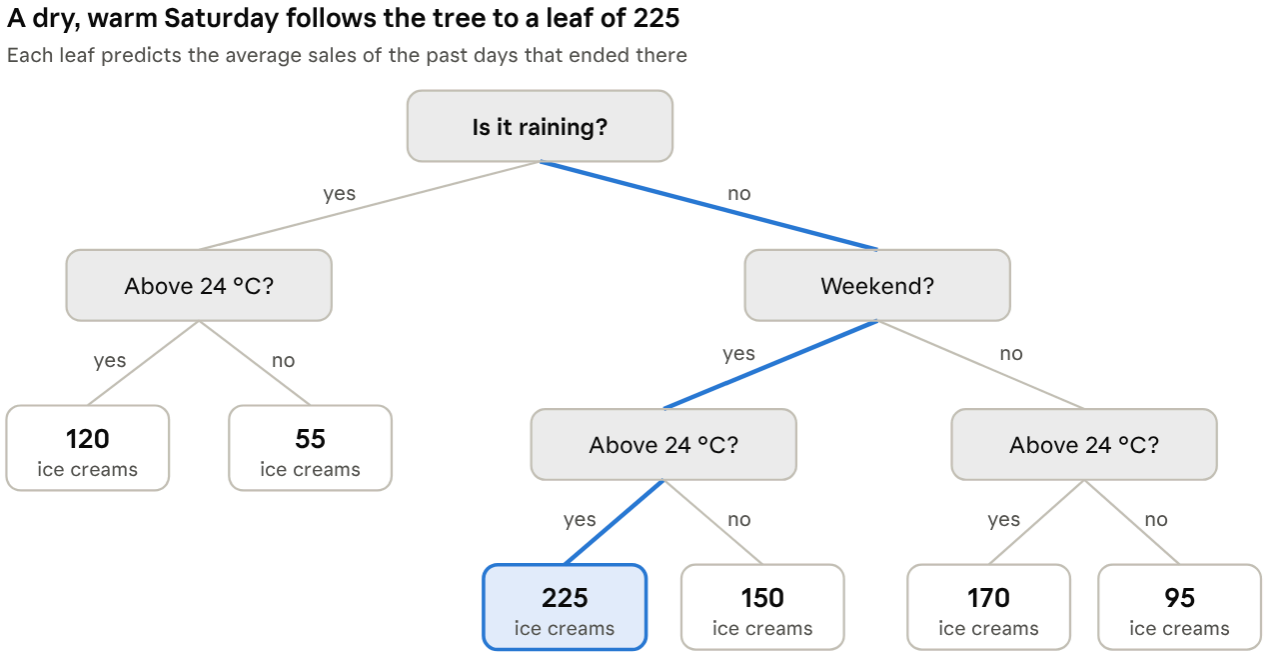


Figure 7.1 · A small decision tree for Maya’s sales, depth 3

To use it, start at the top and answer each question for the day you want to predict. Take a 26-degree Saturday with no rain. Is it raining? No, so go right. Is it a weekend? Yes. Is it above 24 degrees? Yes. The leaf says 225 ice creams.

The number in each leaf is simply the average sales of the past days that ended up there. The tree sorted all of Maya's days into six groups of similar days, and predicts each group's average.

The parts of a tree

Trees have their own vocabulary, borrowed from real trees (although decision trees are drawn upside down, with the root at the top).

- The **root** is the first question, at the top.
- Each question is a **node** (also called a **split**).
- The lines between questions are **branches**.
- The end points, which hold the predictions, are **leaves**.
- The number of questions from the root to the deepest leaf is the tree's **depth**. Maya's tree has depth 3.

How a tree chooses its questions

The computer grows the tree from the top down. At each step it asks itself: of all the questions I could ask right now, which one splits these examples into the most useful groups?

It tries every feature and every sensible cut-off. "Temperature above 18?" "Above 19?" "Above 20?" "Rain?" "Weekend?" For each candidate, it checks how good the two resulting groups are, and keeps the best question. Then it repeats the same search inside each group, and so on down.

What makes a group "good"? For regression, a good group is one where the values are close together, so their average is a good prediction for all of them. For classification, a good group is **pure**: almost all its examples belong to the same class.

Think of sorting laundry. A good first question is "Is it white?", because it splits the pile into two groups that each need very different washing. A bad first question is "Does it have a button?", because both resulting piles are still a mix of colours.

Measuring how mixed a group is

For classification, trees use a score of how mixed a group is. The two common scores are called **Gini impurity** and **entropy**. You do not need their details. Both behave the same way:

- A group where every example has the same class scores **zero**: perfectly pure.
- A group split half and half between two classes scores the **highest**: as mixed as possible.
- Everything else falls in between.

Here is a small example. Maya has 10 days: 5 busy and 5 quiet. She compares two possible first questions.

Question	Group for “yes”	Group for “no”	Verdict
Is it raining?	3 days: 0 busy, 3 quiet	7 days: 5 busy, 2 quiet	Good: one group is pure, the other mostly busy
Is it a Tuesday?	1 day: 1 busy, 0 quiet	9 days: 4 busy, 5 quiet	Poor: the big group is still half and half

The rain question makes both groups much purer than the starting mix. The Tuesday question tidies up just one day and leaves the rest as mixed as before. The tree picks rain.

This way of choosing is called **greedy**: at each step the tree takes the best question available right now, without planning ahead. It does not always find the best possible tree, but it finds a good one very quickly.

When to stop growing

If nobody stops it, a tree keeps asking questions until every leaf holds a single example. Such a tree gets every training day exactly right. It has learned the notebook by heart, including its noise.

That sounds good, but it is a disaster. A leaf that holds one day predicts that day’s exact sales, which included plenty of luck. On new days, the tree’s predictions jump around wildly. This is **overfitting**: fitting the training data so closely that the model fails on new data. It is the subject of Chapter 14.

To prevent it, we limit the tree’s growth with simple settings:

- **Maximum depth:** do not ask more than, say, 4 questions in a row.
- **Minimum leaf size:** every leaf must hold at least, say, 10 examples.
- **Minimum improvement:** only split if the split makes the groups noticeably better.

Another approach is to grow a big tree first and then cut back branches that do not help on fresh data. This is called **pruning**, like a gardener trimming a bush.

Settings like maximum depth are not learned from the data by the training process. You choose them before training. Such settings are called **hyperparameters**, to tell them apart from the parameters that training finds. Chapter 12 explains how to choose them fairly.

What tree predictions look like

A tree cuts the range of each feature into pieces and gives each piece one value. So if you plot a tree's predictions against temperature, you get flat steps, not a smooth line or curve.

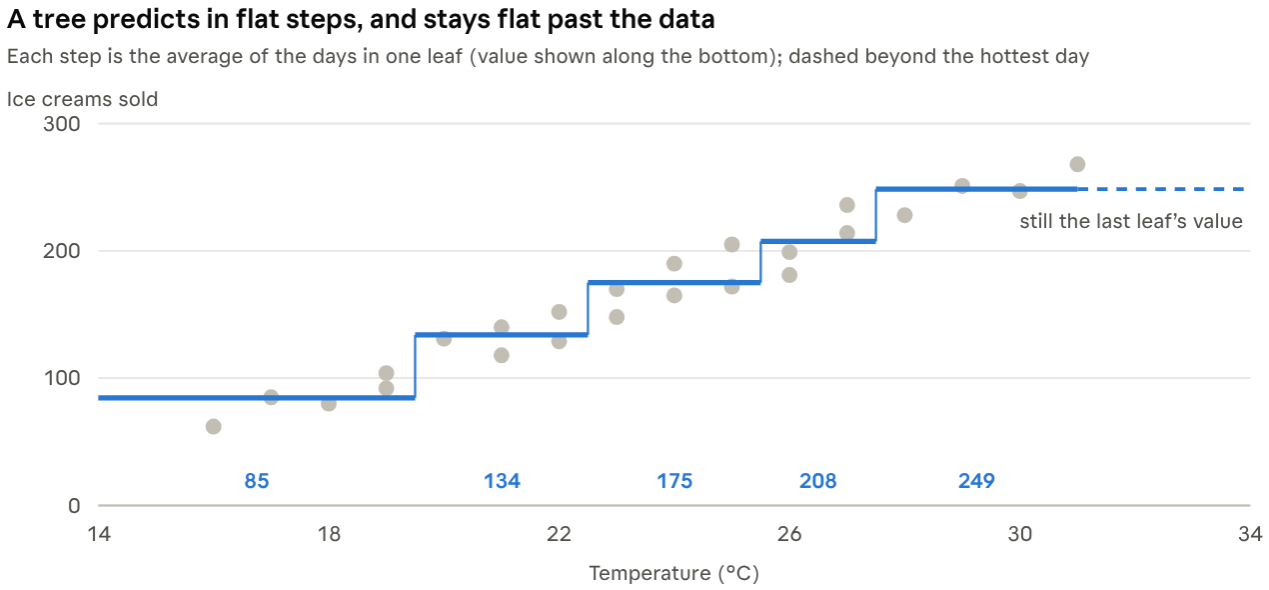


Figure 7.2 · A five-leaf tree on temperature alone, drawn over the 24 example days

The steps show two important habits of trees. First, they can follow a pattern of any shape, bendy or straight, by using enough steps. Second, they **cannot extrapolate**. Beyond the hottest day in the training data, the last step simply stays flat. A tree will never predict more than the highest leaf value it has. Compare this with the straight line of Chapter 5, which happily predicted 400 ice creams at 40 degrees. Neither is right about a heatwave, but they are wrong in opposite ways.

Trees capture interactions

Sometimes the effect of one feature depends on another. Rain might cost Maya only a few sales on a cool day, when few people come anyway, but a great many on a hot day. When one feature changes the effect of another like this, it is called an **interaction**.

Linear regression cannot see interactions unless you build them in by hand. Trees find them naturally. Because the tree asks "Is it raining?" first and then asks about temperature separately on each side, the two sides can treat temperature completely differently.

Which features matter?

A tree can also tell you which features it relied on most. Every time a feature is used for a split, the groups get purer by some amount. Adding up those improvements for each feature gives its **feature importance**. For Maya, temperature might account for most of the improvement, rain for a good share, and weekends for the rest. This is a quick, useful summary, though it should be read with care: two linked features can split the credit between them.

Strengths and weaknesses

	Decision tree
Strengths	Easy to draw and explain; no scaling needed; handles numbers and categories together; finds interactions by itself; fast to train and to use
Weaknesses	Overfits easily if allowed to grow deep; unstable, since a small change in the data can produce a very different tree; step-shaped predictions; cannot extrapolate beyond the training range; a single tree is rarely the most accurate model
Good first choice when	You need a model that non-experts can follow step by step, or as a building block for forests (Chapter 8)

Key ideas

- A **decision tree** predicts by asking a series of yes/no questions about the features.
- It picks each question **greedily**, choosing the split that makes the groups purest or least spread out.
- Leaves predict the average (for regression) or the most common class (for classification) of their training examples.
- Deep trees **overfit**. Limit them with **hyperparameters** such as maximum depth and minimum leaf size, or prune them.
- Tree predictions are flat steps, and trees cannot extrapolate.
- Trees find **interactions** between features on their own.

Try it yourself

1. Use the tree in Figure 7.1 to predict sales on a rainy, 27-degree Wednesday, and on a dry, 20-degree Tuesday.

2. You have 8 animals: 4 birds and 4 fish. Which first question gives purer groups: “Can it fly?” or “Is it bigger than a shoe?” Explain.
3. A tree has grown until every leaf holds exactly one training example. What score will it get on the training data? Why is that not good news?
4. What is the difference between a parameter and a hyperparameter? Give one example of each from this chapter.
5. The highest leaf value in a tree is 260. What will it predict for a record-breaking heatwave day? Is that sensible?

Chapter 8 — Wisdom of Crowds: Random Forests and Boosting

The big idea: Many models together usually beat any single one. A **random forest** grows hundreds of slightly different trees and averages their answers. **Boosting** builds trees one after another, each one fixing the mistakes left by the trees before it. Models that combine other models are called **ensembles**.

Guessing the weight of an ox

In 1906, at a country fair in England, about 800 visitors paid to guess the weight of an ox. The scientist Francis Galton collected their tickets afterwards. Most individual guesses were well off. But the middle guess of the whole crowd came within about 1% of the true weight, closer than almost any single expert.

This effect is called the **wisdom of crowds**. It works because each person’s error goes in a different direction. Some guess too high, some too low, and when you combine them the errors largely cancel out.

There is one condition. The crowd is only wise if its members make **different** mistakes. If everyone copied the loudest person, the crowd would be no smarter than that one person. Keep this in mind: it is the secret of random forests.

Why a single tree needs help

Chapter 7 showed that a single decision tree has a weakness: it is unstable. Remove a few days from Maya’s notebook and the tree might choose a different first question, and everything below it changes. Each tree is like one person at the fair: often right in spirit, but off in its own particular way.

That makes trees perfect members for a crowd. Their mistakes are large, but if we can make them different from each other, averaging will cancel much of the error.

Random forests: many different trees

A random forest makes its trees different in two ways.

Random rows. Each tree is trained on its own random sample of the training examples. The sample is drawn **with replacement**: imagine pulling a day out of a hat, writing it down, and putting it back before pulling the next one. Some days are picked twice or more, and some are not picked at all. Each tree sees a slightly different version of Maya's notebook. This trick is called **bootstrap sampling**, and training many models on bootstrap samples and averaging them is called **bagging** (short for bootstrap aggregating).

Random features. At every split, each tree may only choose from a random handful of the features, not all of them. If temperature is by far the strongest feature, every tree would otherwise put it at the top, and the trees would all look alike. Forcing some trees to try other features first makes the crowd more varied.

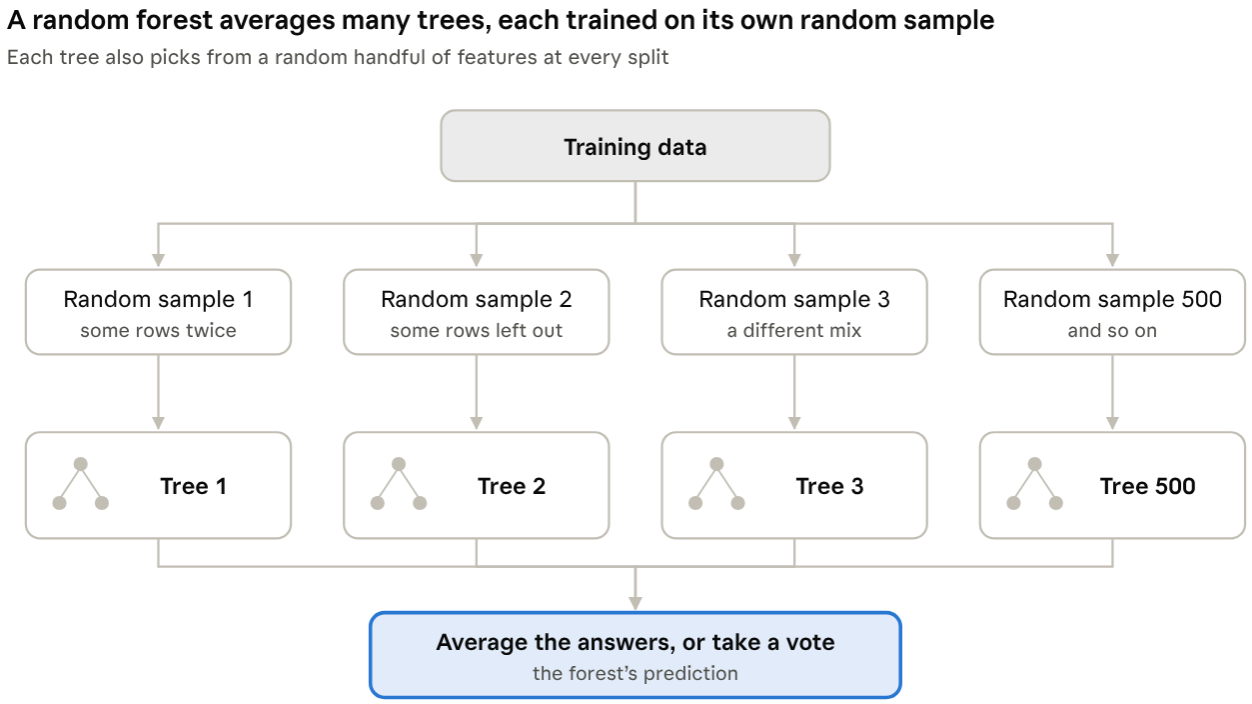


Figure 8.1 · How a random forest is built and how it predicts

To make a prediction, the forest asks every tree. For regression, it averages their answers. For classification, the trees vote, and the class with the most votes wins. The share of votes can also be read as a rough probability.

What the forest buys you

A random forest keeps the good habits of trees and fixes their worst ones.

- It still needs no scaling and handles numbers and categories together.
- It still finds curved patterns and interactions on its own.

- It is far more **stable**: one odd day in the data changes a few trees a little, but barely moves the average.
- It overfits much less, because the noise each tree memorised is different and averages away.

Its predictions are smoother, too. Each tree makes its own staircase with steps in different places, and averaging hundreds of staircases gives something close to a gentle slope.

Adding more trees never makes a random forest overfit; it just makes the average steadier. Beyond a few hundred trees, the gains usually become tiny, and the only cost is computing time.

A free test: out-of-bag examples

Because each tree sees only its own bootstrap sample, roughly a third of the training examples are left out of each tree. They are called **out-of-bag** examples. For any example, you can ask only the trees that never saw it, and compare their average with the real answer. This gives a fair estimate of how well the forest does on new data, without setting any data aside. It is a handy check, though Part 3 explains why a separate test set is still the gold standard.

Boosting: learning from mistakes

Boosting uses the same building blocks, small trees, but combines them in a completely different way.

Think of a student preparing for an exam. They do a practice test and mark it. Then, instead of studying everything again, they focus on the questions they got wrong. After another practice test, they focus on the new mistakes. Each round of study targets what is still weak.

Boosting works the same way.

1. Train a small, simple tree on the data. Its predictions will be rough.
2. For each example, work out the error: how far the prediction is from the truth.
3. Train the next tree to predict those errors, not the original labels.
4. Add a small part of the new tree's answer to the running prediction.
5. Repeat, often hundreds or thousands of times.

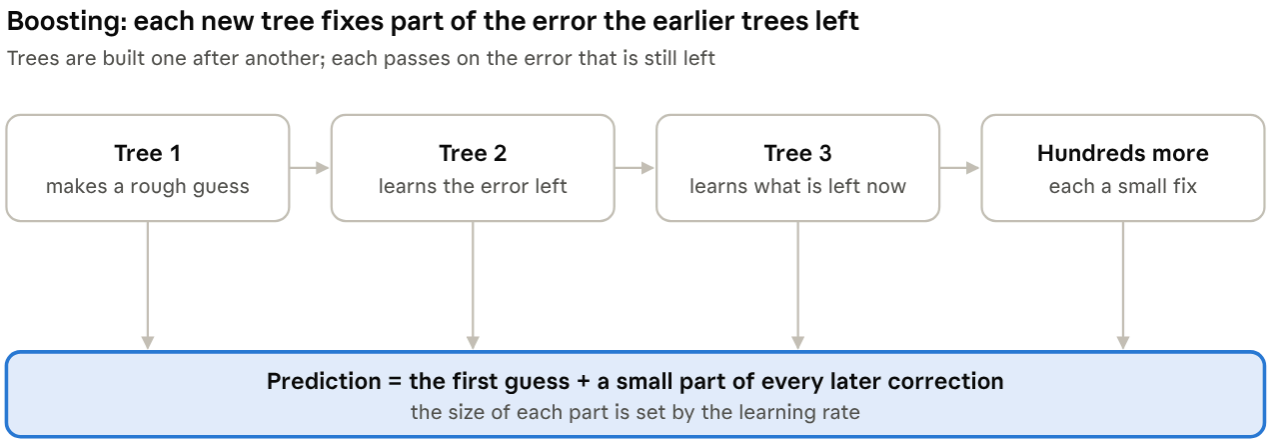


Figure 8.2 · Boosting builds trees in a chain, each fixing what is left

Here is how the running prediction might improve for one day on which Maya really sold 200 ice creams. Each new tree learns the leftover error and adds half of it.

After	Prediction	Leftover error
Tree 1	120	80
Tree 2 (adds half of 80)	160	40
Tree 3 (adds half of 40)	180	20
Tree 4 (adds half of 20)	190	10
Tree 5 (adds half of 10)	195	5

The fraction of each correction that gets added, half in this table, is called the **learning rate**. Real systems usually use a much smaller learning rate, such as a tenth or less, together with many more trees. Taking small steps makes boosting slower but much less likely to chase noise.

The popular modern form is called **gradient boosting**. Several free software libraries implement it very efficiently; the best known are **XGBoost**, **LightGBM** and **CatBoost**. For tables of data, such as sales records, customer data or sensor readings, gradient boosting is often the most accurate method available, and it wins a large share of data science competitions on tabular data.

Boosting can overfit

Unlike a random forest, boosting can overfit if it runs too long. Each new tree chases the remaining errors, and eventually the only errors left are noise. A boosted model that keeps going will start memorising that noise.

The usual protection is **early stopping**: keep a separate set of examples aside, watch the error on them after each new tree, and stop when it stops improving. Chapter 12 explains this kind of check.

Forests and boosting side by side

	Random forest	Gradient boosting
How trees are built	Independently, all at once	One after another, each fixing the last
Typical tree	Deep and detailed	Small and simple
How answers combine	Average or vote	Sum of small corrections
Risk of overfitting	Low; more trees do not hurt	Higher; needs early stopping and a small learning rate
Settings to tune	Few, and defaults work well	Several, and they matter
Accuracy on tables	Very good	Often the best
Best for beginners	Yes, a reliable first strong model	Yes, once you have a fair testing setup

Ensembles beyond trees

The crowd idea works with any models, not only trees. You can average a linear regression, a forest and a neural network. As long as they make different kinds of mistakes, the average tends to beat each one. Training a final model to learn the best way to combine the others is called **stacking**.

The price: harder to explain

A single tree can be drawn on a page. A forest of 500 trees cannot. Ensembles trade readability for accuracy. To see inside them, people use explanation tools. **Feature importance** (Chapter 7) works for forests too, by adding up each feature's contribution across all trees. **Permutation importance** shuffles one feature at a time and measures how much worse the model gets; a feature whose shuffling hurts a lot was important. More advanced tools, such as **SHAP values**, explain how much each feature pushed one particular prediction up or down. Remember that all of these explain the model, not the world: an important feature is useful for prediction, but it is not proven to be a cause.

Key ideas

- **Ensembles** combine many models; their errors partly cancel out, like the wisdom of crowds.
- The crowd only helps if its members make **different** mistakes.
- A **random forest** trains many trees on **bootstrap samples** with random features at each split, then averages or votes.
- **Boosting** builds trees one after another, each learning the errors left by the previous ones; the **learning rate** controls the step size.
- Forests are hard to overfit; boosting needs **early stopping** but is often the most accurate method for tables.
- Ensembles are harder to explain; tools like permutation importance help.

Try it yourself

1. Why would a crowd of people who all copy one expert not be wise? Link your answer to random forests.
2. You draw 5 days with replacement from a hat containing days A, B, C, D and E. Give an example of what your sample might look like. Which days are out-of-bag in your example?
3. In the boosting table, what would the prediction be after Tree 6?
4. A friend says: “My random forest overfits, so I will add more trees.” Will that help? What might help instead?
5. Your boosted model’s error on held-out data falls for 300 trees and then starts to rise. What should you do?

Chapter 9 — Birds of a Feather: k-Nearest Neighbours

The big idea: To make a prediction for a new case, **k-nearest neighbours** (often written **k-NN**) finds the k past examples that are most similar to it, and copies their answer. For a number it takes their average; for a category it takes a vote.

How an estate agent prices a house

Ask an estate agent what your house is worth. They will not build a formula. They will look for houses like yours that sold recently: same street or nearby, similar size, similar number of bedrooms. If the three most similar houses sold for 300,000, 310,000 and 320,000, they will say your house is worth about 310,000.

That is k-nearest neighbours, with k = 3. The idea is as old as common sense: **similar things tend to have similar outcomes**. Birds of a feather flock together.

Maya asks her neighbours

Tomorrow will be 25 degrees, a Saturday, with no rain. Maya opens her notebook and looks for the past days most like that one. Here are six of them, along with a simple measure of how different each is from tomorrow. The measure counts one point for each degree of difference, plus five points if one day is a weekend and the other is not.

Past day	Temperature	Weekend?	Sales	Difference from tomorrow
F	25	Yes	225	0
A	24	Yes	210	1
B	26	Yes	238	1
C	25	No	160	5
D	19	Yes	120	6
E	27	No	185	7

With k = 3, the nearest neighbours are F, A and B. Their average sales are (225 + 210 + 238) ÷ 3, which is about 224. That is the prediction.

Notice what happened to day C. It had exactly the same temperature, but it was a weekday, so it ended up far away and was left out. The rule “five points for a different weekend” decided that. Make a different weekend count for nothing, and C would tie with F as the closest day of all. How much each feature counts in the distance is a real choice, and it matters.

Measuring distance

To find the nearest neighbours, the model needs a way to measure how far apart two examples are. With two features, picture each example as a point on a map, with one feature across and the other up. The distance is just the length of a straight line between two points, measured with a ruler. This is called **Euclidean distance**, and computers can measure it the same way for any number of features.

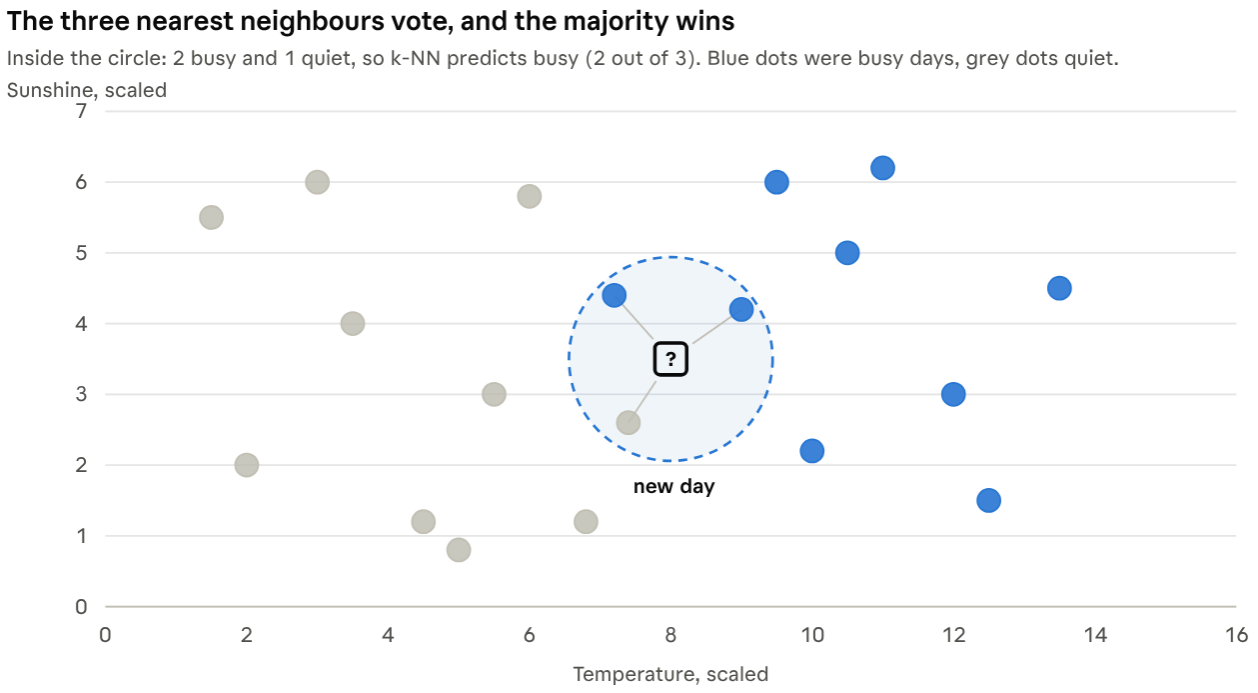


Figure 9.1 · Made-up example: k-nearest neighbours with k = 3

In the picture, the new day sits among busy and quiet days. Its three nearest neighbours are inside the circle: two busy days and one quiet day. The vote is two to one, so k-NN predicts “busy”. It can also report the vote as a rough probability: two out of three, about 67%.

Why scaling matters so much here

Remember the house prices example from Chapter 4. Suppose one feature is floor area, from 40 to 300 square metres, and another is the number of bedrooms, from 1 to 5. A difference of 50 square metres would swamp a difference of 3 bedrooms, simply because the numbers are bigger. The distance would be almost entirely about floor area.

k-NN is the model most hurt by unscaled features, because distance is all it uses. So always **scale your features** before using k-NN. After scaling, each feature starts with an equal say. If you know some features matter more, you can then give them extra weight on purpose, as Maya did with weekends.

Choosing k

The number of neighbours, k, is a hyperparameter: you choose it before using the model. It has a big effect.

Choice of k	What happens	Problem
k = 1	Copy the single nearest example	Very jumpy; one noisy example decides everything (overfitting)
Small k, such as 5	Average a handful of close examples	Usually a good balance
Large k, such as 50	Average many examples, some quite far away	Predictions become blurry and miss local patterns
k = all examples	Every prediction is the overall average	The model ignores the features completely (underfitting)

There is no single right k. Chapter 12 shows how to pick it by testing several values on data the model has not seen. For yes/no problems, people often choose an odd k, such as 3, 5 or 7, so the vote cannot tie.

A useful variation is **weighted k-NN**: closer neighbours get a bigger say than further ones. A neighbour at distance 1 might count twice as much as one at distance 2. This makes the choice of k less critical.

A lazy learner

k-NN has an unusual property: it does no training at all. It simply stores the examples. All the work happens later, when a prediction is needed. For this reason it is called a **lazy learner**. Most other models are **eager learners**: they do the hard work during training and then predict quickly.

Laziness has a cost. To predict one new case, a plain k-NN must measure the distance to every stored example. With a million examples, that is a million distance measurements for every single prediction. There are clever filing systems that speed up the search, but k-NN is still slow for very large data sets.

The curse of dimensionality

k-NN works beautifully with a few features. With many features, it runs into a strange problem with a dramatic name: the **curse of dimensionality**.

Here is the intuition. Finding someone similar to you in two ways, such as age and height, is easy: many people in your town will be close on both. Finding someone similar to you in a hundred ways, including age, height, hometown, favourite food, music taste, job, shoe size and ninety-three more, is almost impossible. With enough features, **everybody is far from everybody**, and the “nearest” neighbour is not really near at all.

When the nearest neighbours are not actually similar, copying their answers stops making sense. Two cures help: use fewer, better features, or squeeze many features into a few with dimensionality reduction (Chapter 11).

Where k-NN is used

- **Recommendations:** "Customers like you also bought…" is nearest-neighbour thinking.
- **Similar-item search:** find photos, songs or documents that are close to one you like.
- **Filling missing values:** guess a missing value from the most similar complete examples.
- **A quick baseline:** k-NN is easy to try and gives a sense of how predictable a problem is.

Strengths and weaknesses

	k-nearest neighbours
Strengths	Very simple to understand and explain ("these past cases were most similar"); no training; follows curvy patterns naturally; new examples can be added instantly
Weaknesses	Needs careful scaling; slow to predict on large data; struggles with many features; sensitive to the choice of k and the distance measure; cannot extrapolate
Good first choice when	The data is small or medium, has few features, and "similar cases" is a natural idea

Key ideas

- **k-NN** predicts by finding the k most similar past examples and averaging (regression) or voting (classification).
- Similarity is measured by **distance**, usually straight-line (**Euclidean**) distance.
- **Scaling** is essential, because distance treats big numbers as more important.
- Small k is jumpy (overfits); large k is blurry (underfits).
- k-NN is a **lazy learner**: no training, but slow predictions.
- With many features, the **curse of dimensionality** makes everything far apart.

Try it yourself

1. Using the table of past days, what would Maya predict with k = 1? With k = 4?
2. Change the rule so that a different weekend adds no points at all. Which two days are now the nearest neighbours, and what is the prediction with k = 2?
3. A k-NN classifier with k = 5 finds 3 spam and 2 safe emails among the neighbours. What does it predict, and with roughly what probability?
4. Why might k-NN give poor results on a data set with 500 features?
5. A friend forgets to scale the features: income in dollars and age in years. Which feature will dominate the distance? Why?

Chapter 10 — Finding Groups: Clustering with k-Means

The big idea: Clustering finds groups of similar examples in data that has no labels. The most popular method, **k-means**, places k centre points, gives each example to its nearest centre, moves each centre to the middle of its group, and repeats until nothing changes.

Parking ice-cream vans on a beach

Maya has bought three ice-cream vans for a long, busy beach. Where should she park them so that every sunbather has a van nearby? She does not know in advance where the crowds will be.

Here is a simple plan that works surprisingly well.

1. Park the three vans anywhere along the beach to start.
2. Every sunbather walks to the van nearest to them.
3. Each van drives to the middle of the group of people who came to it.
4. Now some sunbathers find a different van is nearest. They switch.
5. Repeat steps 3 and 4 until no one switches and no van moves.

At the end, each van sits in the middle of its own crowd, and the crowds have sorted themselves into three groups. That is exactly how k-means works, with k = 3. The vans are called **centroids** (centre points), and the crowds are the **clusters**.

Why cluster at all?

In supervised learning, someone tells the model the right answers. In clustering, nobody does. The model has to discover groups that are already there in the data. This is useful whenever you suspect there are different kinds of things mixed together, but you do not know what the kinds are.

Maya has a loyalty card scheme. For each regular customer she knows three things: the usual time of their visit, how much they spend, and how often they come. She suspects her customers are not all alike, but she has never sorted them. Clustering can do it for her.

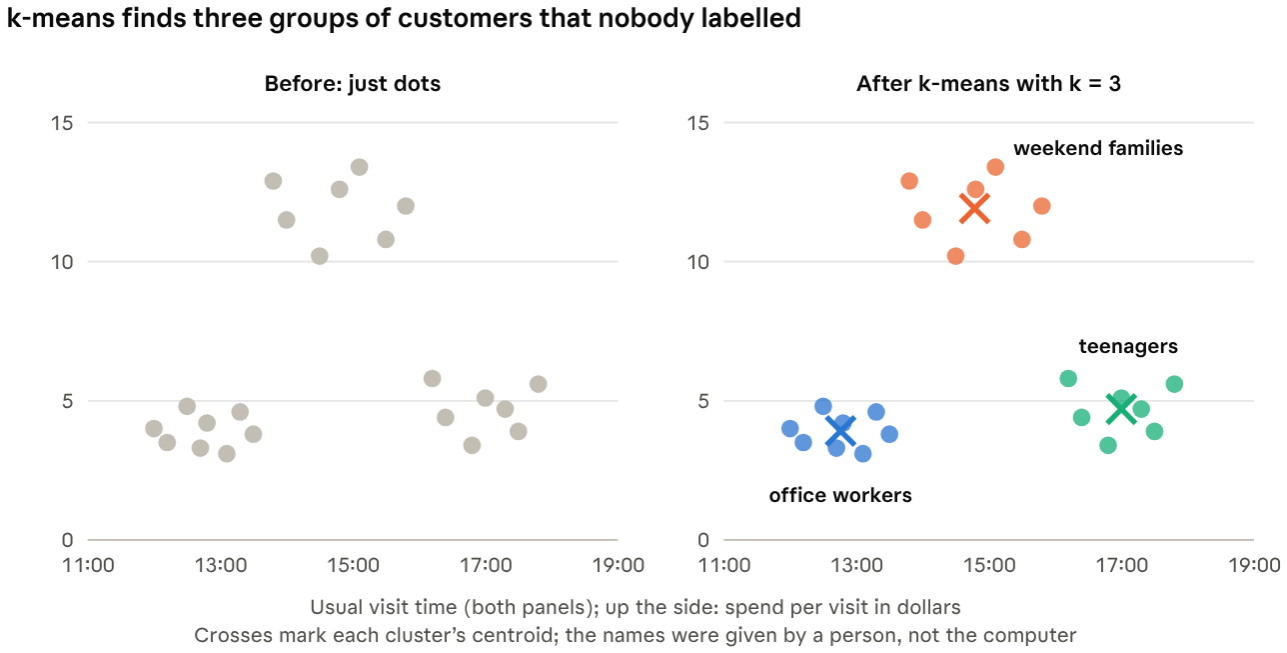


Figure 10.1 · Made-up example: 22 loyalty-card customers before and after clustering

On the left, the customers are just dots. On the right, k-means has found three groups. Looking at what each group has in common, Maya might name them: **lunchtime office workers** who come often and spend a little, **weekend families** who come rarely and spend a lot, and **after-school teenagers** who come in the late afternoon. The computer found the groups; Maya gave them names and meaning.

That last step is always a human job. Clustering produces groups, not explanations. Someone who understands the business must look at each cluster and decide whether it means something useful.

The k-means steps, precisely

Here are the same steps as the beach story, in machine learning words.

1. **Choose k**, the number of clusters you want.
2. **Place k centroids** at random starting points, often on k randomly chosen examples.
3. **Assign** every example to its nearest centroid, using distance as in Chapter 9.
4. **Update** each centroid by moving it to the average position of the examples assigned to it.
5. **Repeat** steps 3 and 4 until the assignments stop changing.

The method always settles down, usually within a few dozen rounds. Each round makes the groups a little tighter: the total distance from examples to their centroids can only go down or stay the same.

How many clusters?

k-means needs you to choose k in advance. But how many groups are there really? Often nobody knows. Two tools help.

The first is the **elbow method**. Run k-means for k = 1, 2, 3 and so on. For each, measure how spread out the examples are inside their clusters. More clusters always means less spread, because each group is smaller. But at some point the improvement suddenly slows. Plot the spread against k and look for the bend, like an elbow in an arm.

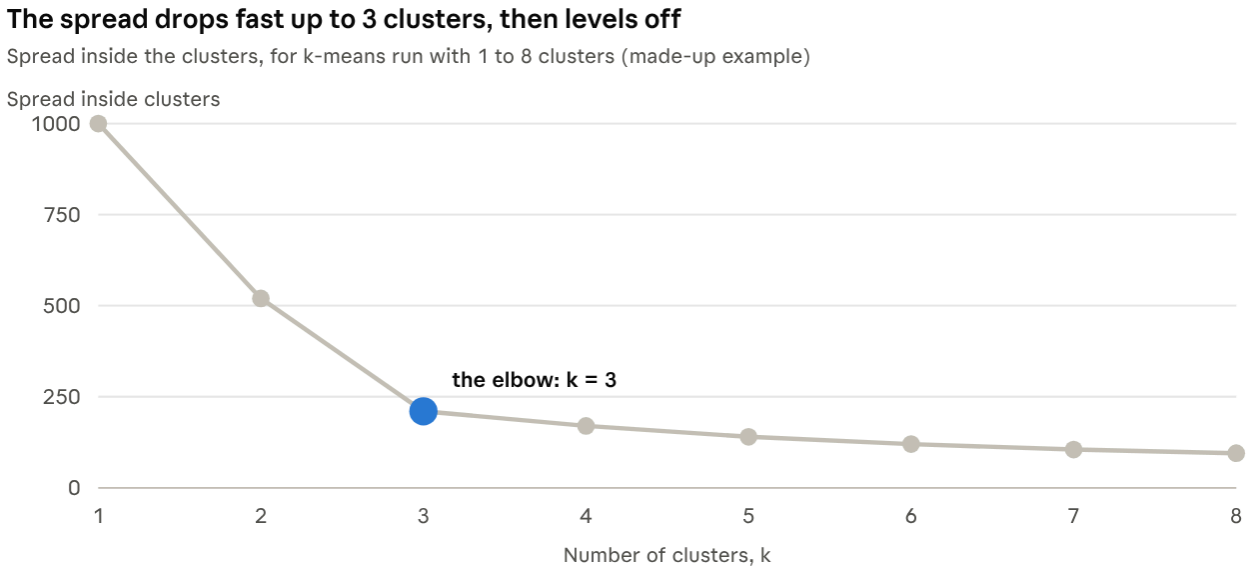


Figure 10.2 · The elbow method for choosing k (made-up numbers)

Going from 1 cluster to 2, and from 2 to 3, cuts the spread a lot. After 3, each extra cluster helps only a little. The elbow at 3 suggests there are three natural groups.

The second tool is **common sense about use**. If Maya can only run three different marketing campaigns, five clusters are no use to her even if the data supports them. The right number of clusters is partly a question about what you will do with them.

There are also scores that measure how well separated the clusters are, such as the **silhouette score**, which compares how close each example is to its own cluster against how close it is to the next nearest one. They are useful guides, but none of them gives a final answer.

Things that go wrong

- **Unlucky starts.** Because the centroids start at random, k-means can settle into a poor answer. Two vans might start in the same crowd and split it, while another crowd gets no van. The fix is to run it several times from different starts and keep the best result. A smart way of choosing spread-out starting points, called **k-means++**, is the default in most software.
- **Unscaled features.** k-means uses distance, so, exactly as with k-NN, a feature with big numbers dominates. Scale first.
- **Odd shapes.** k-means assumes clusters are round blobs of similar size. It fails on groups shaped like long snakes or rings, because it always cuts the space into regions around centres.
- **Outliers.** One very unusual customer can drag a centroid towards them and away from the real middle of the group.

Other ways to cluster

k-means is not the only method. Two others are worth knowing by name.

Hierarchical clustering builds a family tree of examples. It starts with each example alone, then joins the two closest examples, then the next two closest groups, and so on, until everything is one big group. The result is a tree diagram called a **dendrogram**. You can cut it at any height to get as many clusters as you want, which means you do not need to choose k in advance.

DBSCAN finds clusters as dense crowds separated by empty space. It can find groups of any shape, works out the number of clusters by itself, and labels examples that sit alone in empty space as **noise**. It is a good choice when clusters have strange shapes or when you want to spot outliers.

Where clustering is used

- **Customer segments:** grouping shoppers by behaviour to tailor offers, as Maya did.
- **Organising documents:** grouping news articles or support tickets by topic.
- **Shrinking images:** grouping a photo's millions of colours into, say, 16 clusters and painting each pixel with its cluster's colour, which makes the file much smaller.
- **Finding oddities:** examples far from every centroid are unusual, which can flag fraud or faulty sensors.
- **Exploring new data:** clustering is a quick way to see what kinds of things a data set contains before deciding what to predict.

Strengths and weaknesses

	k-means clustering
Strengths	Simple and fast, even on large data; easy to explain with centroids; results are easy to act on, since each cluster has a clear centre
Weaknesses	You must choose k; results depend on the random start; needs scaling; assumes round clusters of similar size; sensitive to outliers
Good first choice when	You want a quick, understandable grouping, and you expect compact groups

Key ideas

- **Clustering** finds groups in unlabelled data; it is **unsupervised learning**.
- **k-means** repeats two steps, assign to the nearest **centroid** and move the centroid to the middle, until nothing changes.
- Choose k with the **elbow method**, separation scores and common sense about how the groups will be used.
- Run k-means several times, scale features, and watch out for odd shapes and outliers.
- The computer finds the groups; **people give them meaning**.

Try it yourself

1. Explain the k-means steps to a friend using the beach van story, without using any jargon.
2. Four customers sit on a number line at positions 1, 2, 9 and 10. Two centroids start at 1 and 2. Do one round of assign and update. Where do the centroids move to?
3. Why might k-means give different results each time you run it? How do people fix this?
4. A shop finds that 7 clusters fit its data best, but it can only run 3 kinds of promotion. How many clusters should it use? Why?
5. Clusters of stars in the night sky are shaped like long curved arms. Would k-means be a good choice? What might work better?

Chapter 11 — Squeezing Data: Dimensionality Reduction

The big idea: When data has many features, many of them often repeat the same information. **Dimensionality reduction** replaces many features with a few new ones that keep most of what matters. The most common method, **principal component analysis (PCA)**, finds the directions in which the data varies the most.

The best angle for a photo

Imagine taking one photo of a teapot to show someone what it looks like. A photo from directly above shows a circle and a lid; you cannot tell it is a teapot. A photo from the side shows the spout, the handle and the shape. Both photos squeeze a three-dimensional object into two dimensions, but the side view keeps far more of the information.

Dimensionality reduction is the search for the best angle. It squeezes data with many features into fewer features, choosing the view that loses as little as possible.

In this field, each feature is called a **dimension**, because each one adds a direction in which examples can differ. A table with 50 features is 50-dimensional data. We cannot picture 50 dimensions, but the maths works the same way as in two or three.

Why squeeze data?

- **To see it.** People can only look at two or three dimensions at once. Squeezing data down to two lets you draw it and spot groups or odd cases.
- **To beat the curse of dimensionality.** As Chapter 9 showed, many features make every example far from every other. Fewer, richer features help distance-based models.
- **To remove repetition.** Features that move together carry the same information several times. Combining them makes models simpler and steadier.
- **To save time and space.** Fewer numbers mean faster training and smaller files.
- **To filter noise.** The small, leftover directions often hold mostly noise. Dropping them can make the pattern clearer.

Maya's weather features

Maya has started recording more about the weather: temperature, hours of sunshine, the UV index, and the "feels like" temperature. But these four features are closely linked. Hot days tend to be sunny, sunny days have a high UV index, and "feels like" closely follows the real temperature. Four columns are really telling roughly one story: **how summery is the day?**

Dimensionality reduction can notice this and build a single new feature, a kind of "summer score", that captures most of what the four columns say. Maya's model then has one strong feature instead of four tangled ones.

How PCA works, in pictures

Take just two of the linked features: temperature and hours of sunshine. Plot each day as a dot. Because they are linked, the dots form a long, tilted cloud.

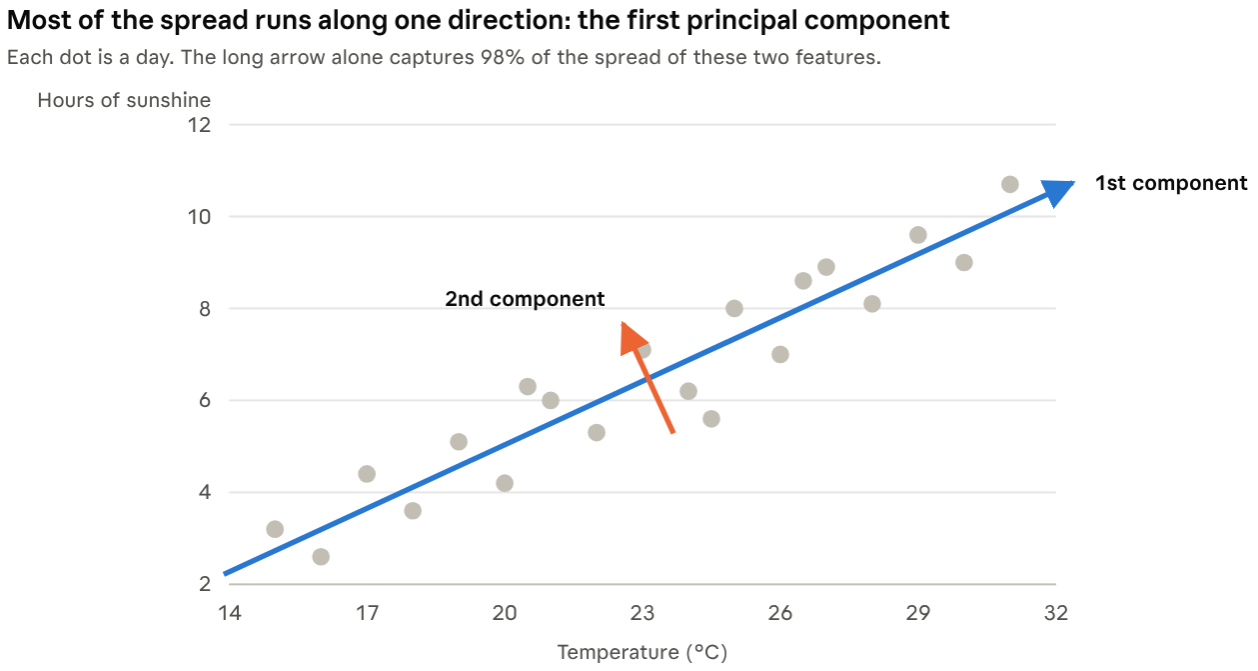


Figure 11.1 · Made-up example: PCA on temperature and hours of sunshine

PCA looks for the direction along which the cloud is most stretched out. That is the long arrow. It is called the **first principal component**. Every day can be described by how far along this arrow it sits: that single number is the summer score.

Next, PCA looks for the direction with the most remaining spread, at a right angle to the first. That is the short arrow, the **second principal component**. It describes how a day differs from the main trend, such as a hot but cloudy day. Here it carries much less information, because the cloud is thin in that direction.

With more features, PCA keeps going: a third direction at right angles to both, then a fourth, and so on, each capturing less spread than the one before. You then keep only the first few components and drop the rest.

How much information is kept?

PCA reports how much of the total spread each component captures. This is often called the **explained variance**. It tells you how many components you need.

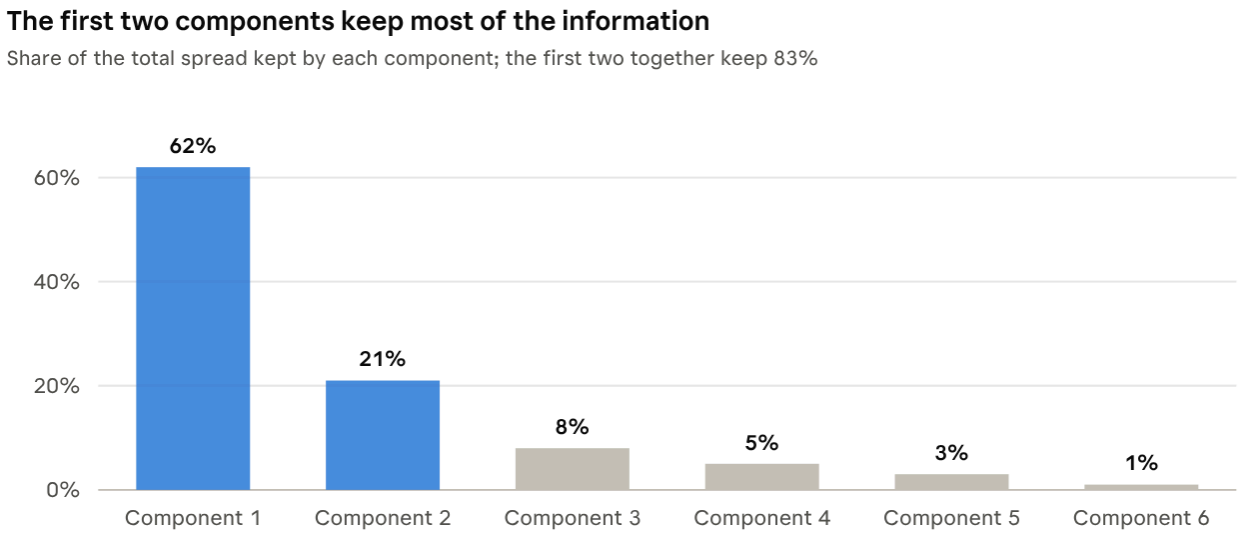


Figure 11.2 · Explained variance for six features (made-up example)

In this example with six features, the first component alone keeps 62% of the information and the first two together keep 83%. The last three keep under 10% between them. Keeping just two or three components would lose little.

A common rule of thumb is to keep enough components to cover around 90% to 95% of the spread. But if your goal is to draw a picture, you keep exactly two, whatever they cover.

The catch: mixtures are hard to name

The new components are mixtures of the original features. The first might be “40% temperature, 30% sunshine, 20% UV index, 10% feels-like”. Sometimes, as with the summer score, the mixture has a clear meaning. Often it does not, and you end up with features called “component 1” and “component 2” that are hard to explain to anyone.

So PCA trades understanding for compactness. If you need to explain your model to a shop owner or a regulator, that trade may not be worth it.

Two practical notes. First, **scale the features before PCA**. Otherwise the feature with the biggest numbers looks like it has the most spread, just because of its units. Second, PCA finds only straight-line directions. If the data lies along a curve, PCA cannot follow it.

Selecting instead of squeezing

There is a simpler way to reduce features: keep some of the original columns and drop the rest. This is called **feature selection**, as opposed to **feature extraction**, which builds new features as PCA does.

Feature selection keeps the original meanings, so the model stays easy to read. Common approaches include:

- Dropping features that hardly ever change, such as a column that is “no” 99.9% of the time.
- Dropping one of each pair of features that move almost perfectly together.
- Keeping the features that a model finds most useful, using the importance tools from Chapter 8.
- Trying the model with and without each feature, and keeping those that improve its score on held-out data.

The last approach is the most reliable, and Part 3 explains how to test fairly. Be careful, though: if you try thousands of combinations, some will look good by luck alone.

Maps of high-dimensional data

Sometimes the goal is just a picture: a two-dimensional map in which similar examples sit near each other. Two popular tools for this are **t-SNE** and **UMAP**. They are very good at showing clusters, for example groups of similar songs, cells or documents.

Read such maps with care. They keep track of which examples are neighbours, but distances between far-apart groups and the sizes of groups on the map can be misleading. Use them to explore, not to measure.

Where dimensionality reduction is used

- **Images:** a photo with millions of pixels can be summarised by a few hundred components for faster searching and comparison.
- **Genetics:** thousands of gene measurements per person squeezed into a handful of components often reveal ancestry and population groups.
- **Finance:** the prices of hundreds of shares often move together; a few components capture “the whole market went up” or “technology shares fell”.
- **Surveys:** forty questions about customer satisfaction may really measure three or four underlying attitudes.

Key ideas

- Each feature is a **dimension**; many features often repeat the same information.
- **Dimensionality reduction** replaces many features with a few that keep most of the information.
- **PCA** finds the directions of greatest spread; the first **principal component** captures the most.
- **Explained variance** shows how much each component keeps, and helps you choose how many to use.
- Components are mixtures and can be hard to name. Scale features before PCA.

- **Feature selection** keeps original columns; **feature extraction** builds new ones.

Try it yourself

1. Give an example of three features about people that probably move together and could be squeezed into one.
2. In Figure 11.2, how much information is kept if you use the first three components?
3. Why should you scale features before PCA? Give an example where forgetting would mislead you.
4. A doctor needs to understand why a model flagged a patient. Would you prefer feature selection or PCA? Why?
5. Describe the teapot photo idea in your own words and link it to the first principal component.

Part 3 — Is It Any Good?

Chapter 12 — The Exam Analogy: Training, Validation and Test Sets

The big idea: A model must be judged on examples it has never seen. So we split the data into a **training set** to learn from, a **validation set** to choose between models and settings, and a **test set** that is used only once, at the very end, as a final exam.

An exam you have already seen

Imagine a teacher hands out the exact exam questions a week early “for practice”. On exam day, every student scores 100%. Does that prove they understand the subject? Of course not. They may have memorised the answers without understanding anything. Give them a single new question and they might be lost.

Testing a model on the same examples it learned from is exactly this mistake. A flexible model can memorise its training data, noise and all. Its score on the training data will look wonderful and mean almost nothing.

The score that matters is how the model does on **new** data, because that is how it will be used. Maya does not need a model that explains last summer perfectly. She needs one that predicts next Saturday well.

Three piles of data

The standard solution is to split the data into three parts before doing anything else.

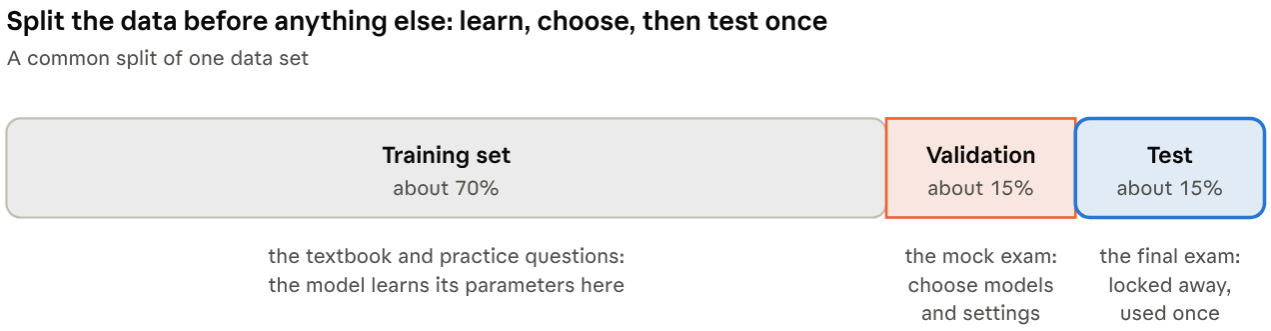


Figure 12.1 · Training, validation and test sets

- The **training set** is the textbook and the practice questions. The model learns its parameters from it.
- The **validation set** is the mock exam. You use it to compare different models and different settings, such as the depth of a tree or the k in k-NN, and pick the best. It is also called the **development set** or **dev set**.
- The **test set** is the real final exam. It stays locked away until all choices are made. You use it once to get an honest estimate of how the finished model will do on new data.

A common split is about 70% for training, 15% for validation and 15% for testing. With a very large data set, the validation and test parts can be smaller percentages, because even 1% of ten million examples is plenty.

Why we need a separate validation set

Why not just two piles, training and test? Because choosing between models is itself a kind of learning.

Suppose Maya tries k-NN with k from 1 to 30 and picks the k with the best test score. She has just used the test set to make a choice. If she tried enough options, one of them was bound to do well partly by luck on those particular test days. The winning score is now a bit too optimistic. The test set has stopped being an unseen final exam and has become a mock exam.

The fix is to make all choices on the validation set, and touch the test set only once, at the end. If the test score is much worse than the validation score, that is a warning that your choices were tuned too closely to the validation data.

Use the test set once

This rule is easy to say and hard to keep. It is tempting to check the test score, make a small change, and check again. Each peek leaks a little information about the test set into your decisions. After enough peeks, you are no longer measuring how good your model is; you are measuring how well you have tuned it to those particular examples.

Teachers call the human version of this **teaching to the test**. Students learn to pass one particular exam instead of learning the subject. In machine learning, it produces models that look great in reports and disappoint in real use.

When time matters: split by date

Maya's data has a special feature: it comes in time order. Her model will always predict the **future** from the **past**. The test should copy that situation.

If she splits her days at random, the test set will contain a Tuesday whose Monday and Wednesday are both in the training set. The model can lean on what happened just before and just after, which it will never be able to do in real life. The score will look better than the model really is.

The honest way is a **time-based split**: train on the earliest days, validate on the next stretch, and test on the most recent days.

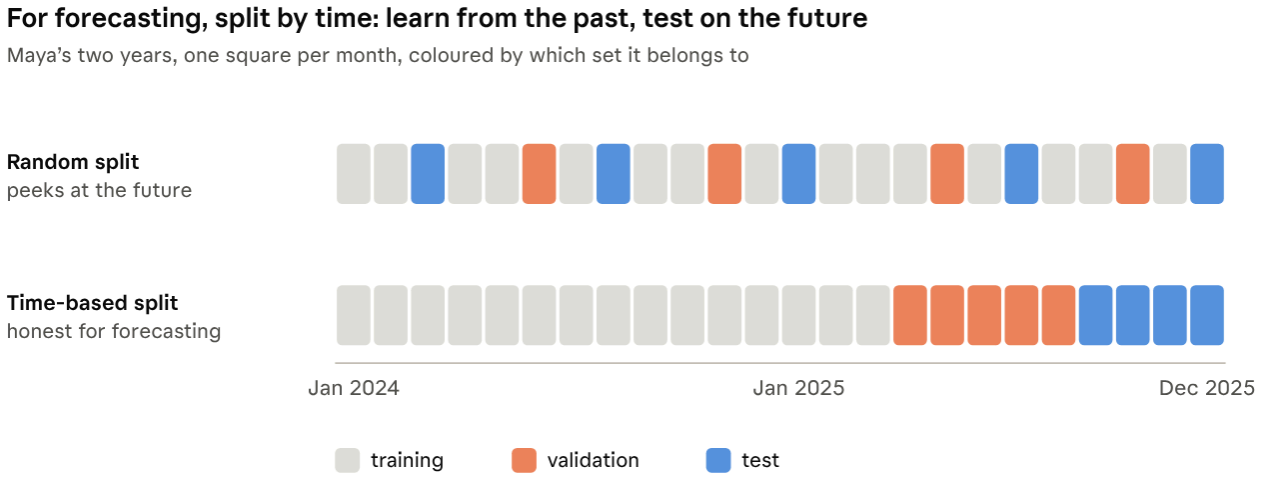


Figure 12.2 · A random split against a time-based split (the time split matches the project in Chapter 20)

This is one of the most common and costly mistakes in practice. Any time the model will be used to predict later events, such as sales, prices, weather or demand, split by time.

Keep related examples together

A similar trap appears when one person, place or object appears many times in the data. Suppose a hospital has ten scans from each patient. If some scans of a patient go into training and others into testing, the model can learn to recognise the patient rather than the illness. It will look brilliant on the test set and fail on new patients.

The fix is a **group split**: all examples from the same patient go into the same pile. The question to ask is always: **in real use, will the model meet things it has already seen during training?** If not, the test should not let it either.

Keep rare classes in every pile

If only 2% of emails are spam, a random split might, by bad luck, put very few spam emails in the test set. Then the test cannot tell you much about how well spam is caught. A **stratified split** fixes this by making sure each pile has the same share of each class as the whole data set.

Cross-validation: making the most of small data

With small data, setting aside 30% hurts. Each pile is small, and the validation score depends a lot on which examples happened to land in it. A different random split might give a quite different score.

Cross-validation solves this by taking turns. In **5-fold cross-validation**, you cut the training data into five equal parts called **folds**. You train on four folds and validate on the fifth. Then you do it again with a different fold held out, and again, until each fold has had one turn as the validation set.

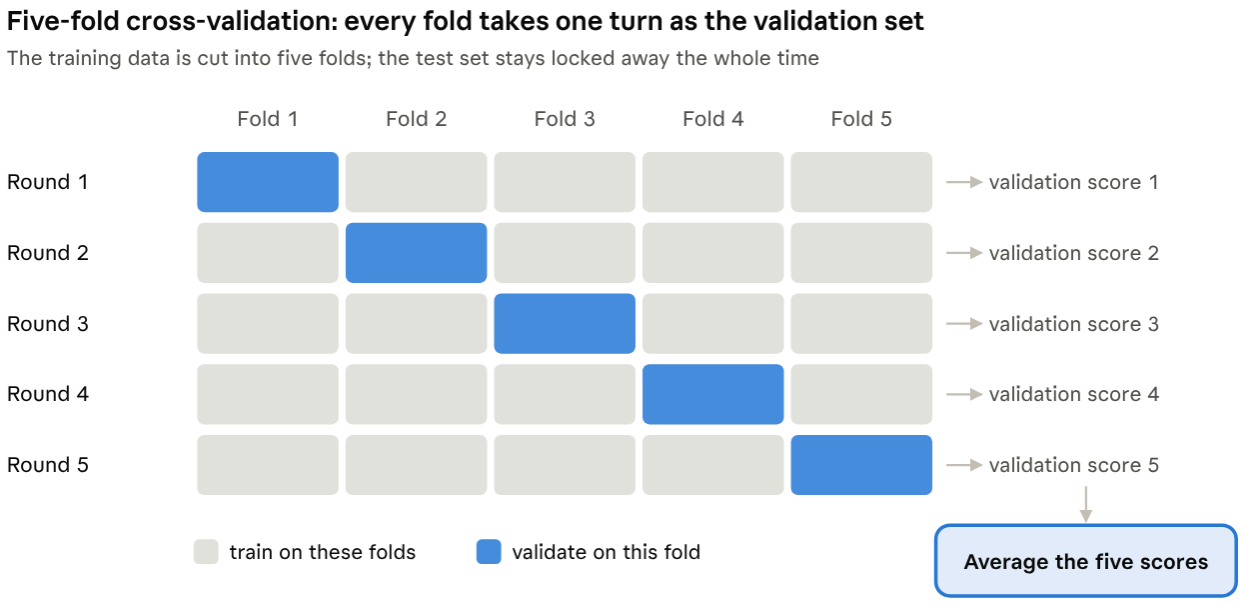


Figure 12.3 · Five-fold cross-validation

You end up with five validation scores. Their average is a steadier estimate than any single split. Their spread is useful too: if the five scores range from 0.60 to 0.90, your model's performance is very sensitive to which data it sees, which is worth knowing.

Cross-validation costs more computing time, since you train five models instead of one. For small and medium data sets, it is usually worth it. Ten folds are also common. The test set still stays locked away outside the folds.

For time-ordered data, ordinary cross-validation breaks the time rule. Use **rolling** or **expanding-window** validation instead: train on months 1 to 6 and validate on month 7; then train on months 1 to 7 and validate on month 8; and so on. The model always predicts

forwards.

After the exam

Once you have chosen your model and settings using the validation data, a common final step is to retrain the chosen model on the training and validation data together, since more data usually helps. Then you measure it once on the test set and report that score.

After the test, resist the urge to go back and tweak. If you must, be honest: say that the test set has been used, and collect fresh data for a new final check.

The test must look like the future

No split can rescue a test set that does not resemble real use. If Maya's test set contains only summer days but she plans to use the model all year, the test score says nothing about winter. Before trusting any score, ask: are these test examples like the cases the model will really face?

Key ideas

- Never judge a model on the data it trained on; that is like an exam with leaked questions.
- **Training set:** learn. **Validation set:** choose models and settings. **Test set:** final exam, used once.
- Every choice made using the test set makes its score too optimistic.
- For data in time order, use a **time-based split**: train on the past, test on the future.
- Use **group splits** when the same person or thing appears many times, and **stratified splits** for rare classes.
- **Cross-validation** takes turns with folds to get steadier scores from small data.

Try it yourself

1. Explain to a friend, using the exam analogy, why a model's score on its own training data can be misleading.
2. Maya has 730 days of data. Describe a time-based split into training, validation and test sets with sensible sizes.
3. A researcher tries 200 different models and reports the best test score. What is wrong with this?
4. A face recognition data set has 50 photos each of 1,000 people. How should it be split, and why?
5. With 5-fold cross-validation, how many models are trained? How many times is each example used for validation?

Chapter 13 — Keeping Score: Metrics

The big idea: A **metric** is a number that scores how well a model predicts. Different metrics reward different things, so choose the one that matches what really matters in your problem, and always compare your model with a simple **baseline**.

Why one score is never enough

A restaurant review that only says “7 out of 10” tells you something, but not much. Was the food great and the service slow? Was it cheap or expensive? A single number hides a lot.

Model scores are the same. Each metric looks at the model’s mistakes from one angle. This chapter introduces the most common metrics, explains in plain words what each one measures, and shows when each one can mislead you.

Scoring number predictions

For regression, every prediction has an error: the gap between the prediction and the truth. Metrics for regression are different ways of summarising all those gaps. Let us use the straight line from Chapter 5 on its 24 days.

Mean absolute error (MAE) is the average size of the misses, ignoring whether they were too high or too low. For Maya’s line it is about 10 ice creams. In plain words: on a typical day, the prediction is off by about 10. MAE is easy to understand because it is in the same units as the thing you predict.

Root mean squared error (RMSE) squares each error, averages the squares, then takes the square root to return to the original units. For Maya’s line it is about 12. Because of the squaring, RMSE punishes big misses more than MAE does. If RMSE is much bigger than MAE, a few days had very large errors.

R-squared (R^2) compares the model with the simplest possible guess: always predicting the average. It says what share of the ups and downs in the data the model explains. An R^2 of 1 means perfect predictions. An R^2 of 0 means the model is no better than always guessing the average. It can even go below 0, which means the model is worse than that simple guess. Maya’s line has an R^2 of about 0.96, so it explains about 96% of the day-to-day variation in these 24 days.

Metric	Plain meaning	Maya’s line	Best for
MAE	Typical size of a miss	About 10 ice creams	Explaining results to anyone
RMSE	Typical miss, with big misses counting extra	About 12 ice creams	When big misses are especially costly

Metric	Plain meaning	Maya's line	Best for
R^2	Share of the variation explained, against guessing the average	About 0.96	Comparing how much of the pattern is captured

Another common metric is **mean absolute percentage error (MAPE)**, the average miss as a percentage of the true value. "Off by 6% on average" is easy to grasp. But be careful when true values can be near zero: missing by 5 when the truth is 1 is a 500% error.

Always compare with a baseline

A metric on its own is hard to judge. Is an MAE of 10 good? It depends. If sales vary between 9 and 11 every day, an error of 10 is terrible. If sales vary between 50 and 300, it is excellent.

So always compare your model with a **baseline**: the score of a very simple method that needs no machine learning at all. For number predictions, good baselines include:

- **Always guess the average.** For Maya's 24 days, always guessing 165 ice creams gives an MAE of about 48.
- **Guess the same as yesterday.** This is often hard to beat for things that change slowly.
- **Guess the same as this day last year,** or the usual value for this time of year, for things with strong seasons.

Maya's line cuts the typical error from about 48 to about 10. That comparison is what makes the number meaningful. A model that cannot beat a simple baseline is not worth using, however clever it is. You will be surprised how often this happens.

One caution about these particular numbers: they come from the same 24 days the line was fitted to. As Chapter 12 explained, an honest score must come from held-out days.

Scoring yes/no predictions: accuracy and its trap

For classification, the most obvious metric is **accuracy**: the share of predictions that are right. It is easy to understand, but it has a famous trap.

Imagine a bank where 1 payment in 100 is fraud. A "model" that always says "not fraud" is right 99 times out of 100. Its accuracy is 99%. It also catches zero fraud, which makes it completely useless.

When one class is rare, accuracy hides failure. That is why we need a closer look at the different kinds of mistakes.

The confusion matrix

For a yes/no problem, every prediction lands in one of four boxes. Here are Maya's results on 100 test days, where the model predicted whether each day would be busy.

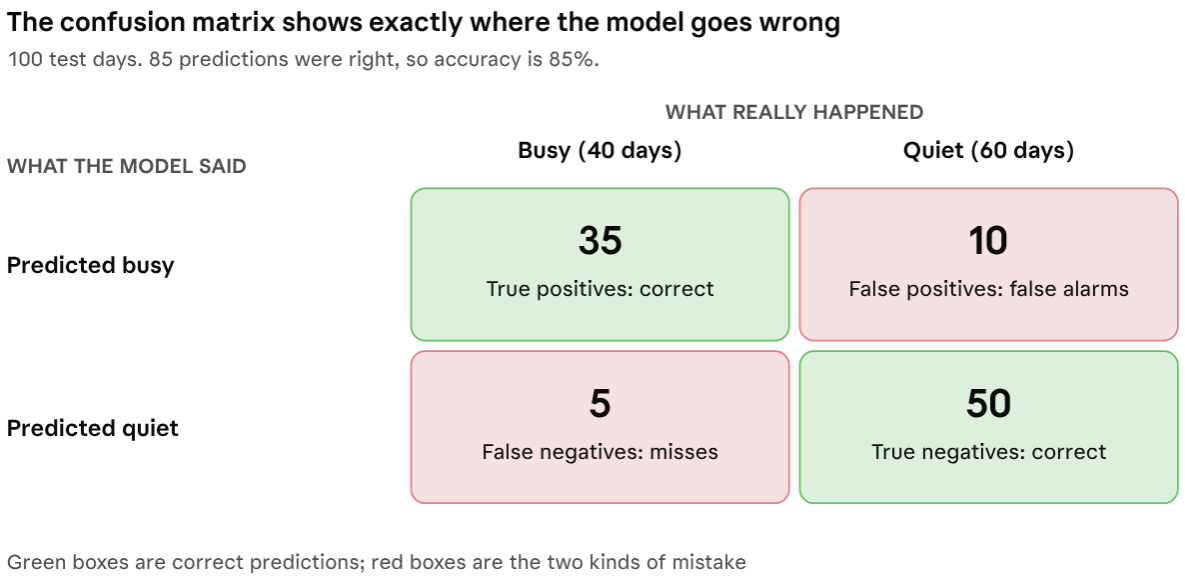


Figure 13.1 · Confusion matrix for 100 test days (made-up example)

The four boxes have standard names.

- **True positives (35):** predicted busy, and it was busy. Correct.
- **True negatives (50):** predicted quiet, and it was quiet. Correct.
- **False positives (10):** predicted busy, but it was quiet. A false alarm. Maya paid her helper for nothing.
- **False negatives (5):** predicted quiet, but it was busy. A miss. The queue was long and customers left.

In this language, "positive" just means the class we are looking for, here "busy". It does not mean good. In medicine, a positive test means the disease was found.

The table is called a **confusion matrix** because it shows exactly where the model gets confused. Accuracy is the two correct boxes divided by everything: (35 + 50) ÷ 100 = 85%.

Precision and recall

Two more metrics come straight from the confusion matrix, and together they tell a much fuller story than accuracy.

Precision asks: when the model says "busy", how often is it right? Of the 45 days predicted busy, 35 were busy. Precision is 35 ÷ 45, about 78%. High precision means few false alarms.

Recall asks: of all the days that really were busy, how many did the model catch? There were 40 busy days, and the model caught 35. Recall is 35 ÷ 40, about 88%. High recall means few misses. Recall is also called **sensitivity** or the **true positive rate**.

An easy way to remember them: precision is about **trusting the alarm**; recall is about **not missing anything**.

The two pull against each other. Recall from Chapter 6 that the threshold turns probabilities into yes/no decisions. Lower the threshold and the model says "busy" more often: it catches more busy days (higher recall) but raises more false alarms (lower precision). Raise the threshold and the opposite happens.

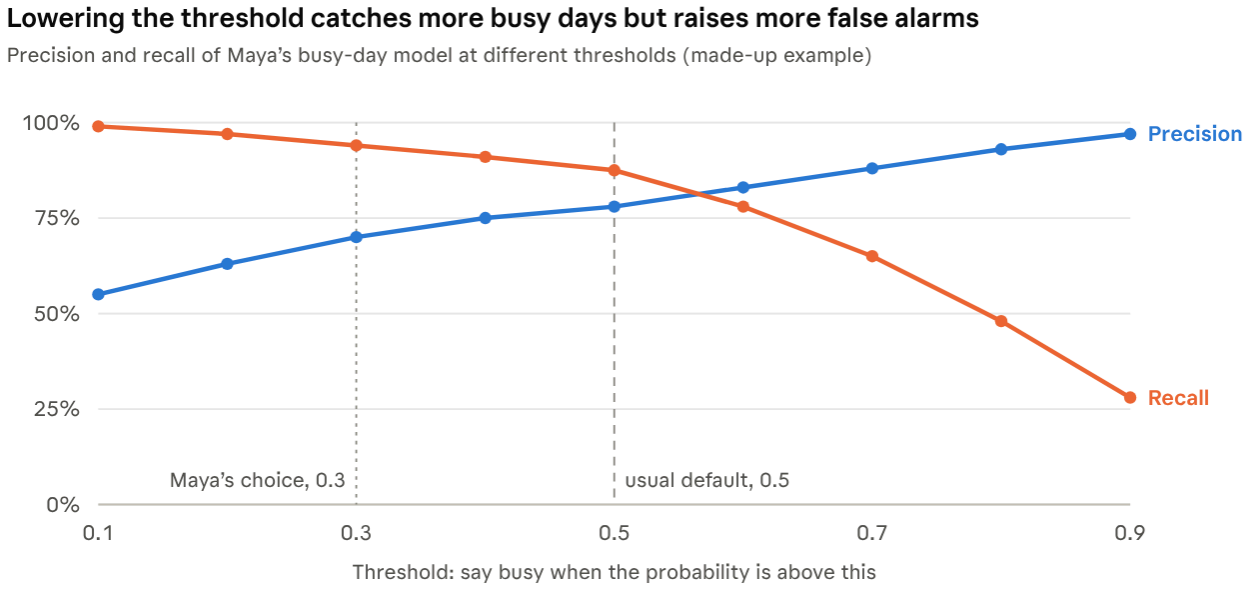


Figure 13.2 · The precision–recall trade-off as the threshold moves (made-up numbers)

Which matters more depends on the costs. For Maya, a miss costs more than a false alarm, so she wants high recall and accepts lower precision. For a spam filter, a false alarm (a real email lost in spam) costs more, so precision matters most. For cancer screening, missing a case is terrible, so recall comes first, and a positive result leads to a second, more careful test.

When you want a single number that balances both, use the **F1 score**. It is a kind of average of precision and recall that is only high when both are high. For Maya's model it is about 0.82. If either precision or recall is poor, F1 is poor too.

Scores that do not need a threshold

Sometimes you want to judge the model's probabilities before anyone has chosen a threshold. Two common tools do this.

The **ROC curve** shows the trade-off between catching positives and raising false alarms for every possible threshold at once. The **area under the curve (AUC)** sums it up in one number between 0.5 and 1. An AUC of 0.5 means the model ranks cases no better than a coin toss; 1 means it always ranks every positive above every negative. AUC is useful for comparing models, but it says nothing about how good any one threshold is, and it can look impressive on problems where the real decisions are still poor. Always look at the confusion matrix at the threshold you will actually use.

Log loss (from Chapter 6) scores the probabilities directly, punishing confident mistakes. It also rewards models whose probabilities are well calibrated.

Baselines for classification

The baseline idea applies here too. The simplest classification baseline is to **always predict the most common class**. In the fraud example, that baseline already scores 99% accuracy. Any real model must beat it on the metrics that matter, such as recall of fraud at an acceptable precision.

Look beyond the average

A single overall score can hide important details. After computing your metric, break it down.

- **By group.** Is the model just as good on weekdays as on weekends? On men and women? On old and new customers? A model can be excellent on average and poor for one group (Chapter 19).
- **By size of error.** What are the ten worst predictions? Do they share something, such as all being holidays?
- **Over time.** Is the error growing in recent months? That can mean the world is changing and the model needs retraining.

Choosing your metric

Situation	Good metric to focus on
Predicting a number, explaining results to non-experts	MAE, compared with a baseline
Predicting a number where big misses are very costly	RMSE
Comparing how much of the pattern different models capture	R^2

Situation	Good metric to focus on
Yes/no with roughly balanced classes	Accuracy, plus the confusion matrix
Yes/no where misses are very costly	Recall, at an acceptable precision
Yes/no where false alarms are very costly	Precision, at an acceptable recall
Yes/no where you need one balanced number	F1 score
Comparing models before a threshold is chosen	AUC or log loss

Pick your main metric before you start building models. Choosing it after you see the results invites you to pick whichever metric makes your favourite model look best.

Key ideas

- A **metric** scores predictions; each one sees mistakes from a different angle.
- Regression: **MAE** is the typical miss; **RMSE** punishes big misses; **R^2** compares against guessing the average.
- Always compare with a **baseline** such as “guess the average” or “same as yesterday”.
- **Accuracy** is misleading when one class is rare.
- The **confusion matrix** shows true and false positives and negatives.
- **Precision**: can you trust the alarm? **Recall**: did you catch them all? The threshold trades one for the other.
- Break scores down by group and over time; choose your main metric before you start.

Try it yourself

1. A model predicts house prices with an MAE of 20,000. What does that mean in plain words? What else would you want to know before calling it good?
2. A disease affects 2 in 1,000 people. What accuracy does the model “always say healthy” get? Why is it useless?
3. A spam filter flags 50 emails. 40 are spam and 10 are real. There were 80 spam emails in total. What are its precision and recall?
4. For a smoke alarm, which matters more, precision or recall? What about a system that automatically blocks bank cards?
5. Model A has an R^2 of 0.3 and Model B has an R^2 of −0.1. What does each number tell you?

Chapter 14 — Overfitting and Underfitting

The big idea: A model can fail in two opposite ways. **Underfitting** means it is too simple to capture the real pattern. **Overfitting** means it is so flexible that it learns the noise along with the pattern, and then fails on new data. Good machine learning means finding the sweet spot between them.

Three students

Three students prepare for a history exam.

The first student skims the textbook for ten minutes. They come away with “there were some wars and some kings”. On the exam, they cannot answer much of anything. This student has **underfitted**: their understanding is too simple to be useful.

The second student memorises every practice question and its answer, word for word, without understanding why. When the exam asks the same facts in a slightly different way, they are stuck. This student has **overfitted**: they learned the exact practice questions, not history.

The third student reads, takes notes, and learns the main causes and effects. They can answer questions they have never seen before. This student has found the right balance. They have **generalised**.

Every model you train will be somewhere between the first two students. Your job is to push it towards the third.

What it looks like

Here is the same set of dots, fitted by three different models. The dots follow a gentle curve, plus some random noise.

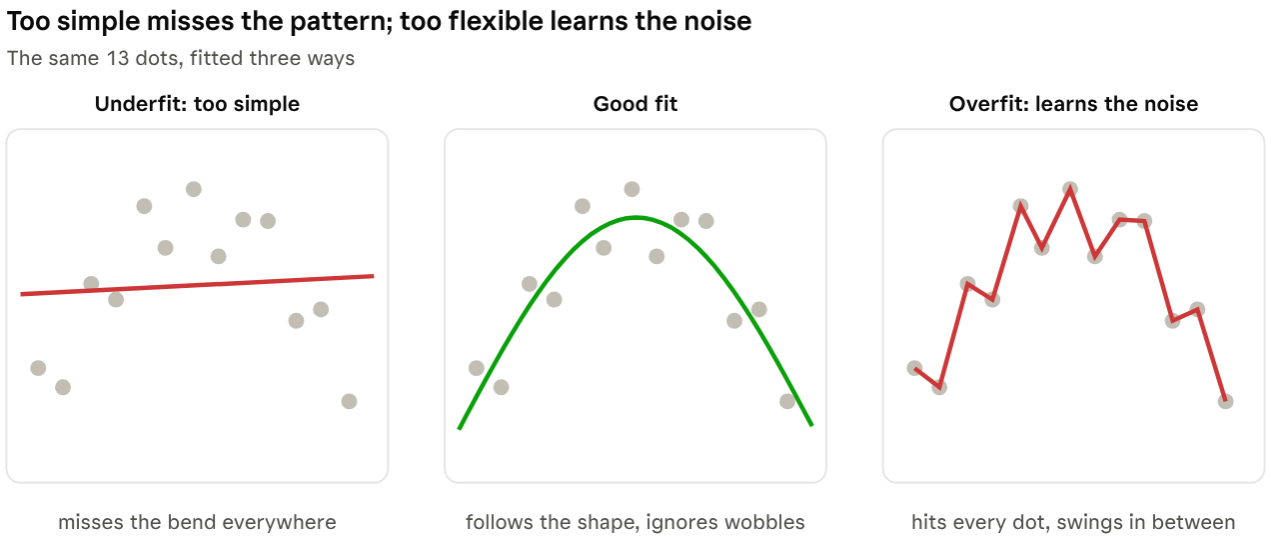


Figure 14.1 · Underfitting, a good fit and overfitting on the same made-up data

- On the left, a straight line **underfits**. It misses the bend that is clearly in the data. It makes large errors on training data and on new data alike.
- In the middle, a smooth curve follows the real shape and ignores the random wobbles. This is a **good fit**.
- On the right, a wiggly line passes exactly through every dot. Its error on the training data is zero. But look at the wild swings between the dots. A new point would almost certainly fall far from that line. This model **overfits**.

The overfitted model is the most dangerous of the three, because its training score looks perfect. Only a test on new data reveals the problem. That is why Chapter 12 matters so much.

Pattern and noise

Remember from Chapter 3 that every data set contains two things: the real **pattern**, which repeats, and **noise**, which is random and will not repeat. Maya's sales rise with temperature; that is pattern. One Tuesday a coach of tourists happened to stop by; that is noise.

A model that overfits treats the noise as if it were pattern. It learns "Tuesdays at 23 degrees sell 40 extra ice creams", because of that one coach. Next Tuesday at 23 degrees there is no coach, and the prediction is wrong.

The more flexible a model is, the more able it is to learn noise. A straight line cannot learn much noise; it is too stiff. A very deep tree or a huge neural network can learn almost anything, including noise.

The tell-tale gap

You can spot overfitting and underfitting by comparing two scores: the error on the training data and the error on the validation data.

Training error	Validation error	Diagnosis
High	High	Underfitting: the model is too simple
Low	Much higher	Overfitting: the model learned the noise
Low	Only a little higher	A good fit

A small gap is normal; models always do a little better on the data they learned from. A large gap is the warning sign.

Watching it happen

Here is what happens to the two errors as a decision tree is allowed to grow deeper, from 1 question up to 12.

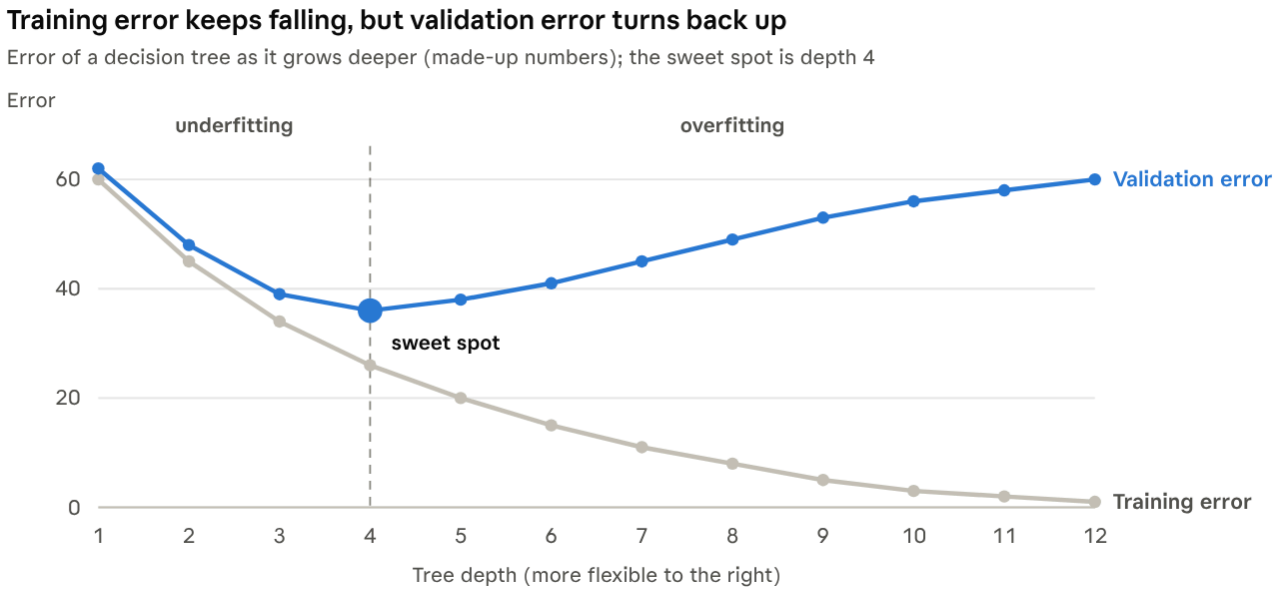


Figure 14.2 · Training and validation error as tree depth grows (made-up numbers)

The training error keeps falling: a deeper tree can always fit its training data better, until each leaf holds one example. The validation error falls at first, as the tree learns the real pattern, and then starts to rise, as the tree starts learning noise. The best depth is at the bottom of the validation curve, here around 4. That is the sweet spot.

This U-shaped validation curve appears almost everywhere: for tree depth, for the k in k-NN (reversed, since small k is the flexible end), for the number of boosting rounds, and for the length of neural network training.

Bias and variance

Experts describe the same idea with two words: **bias** and **variance**. Picture a game of darts.

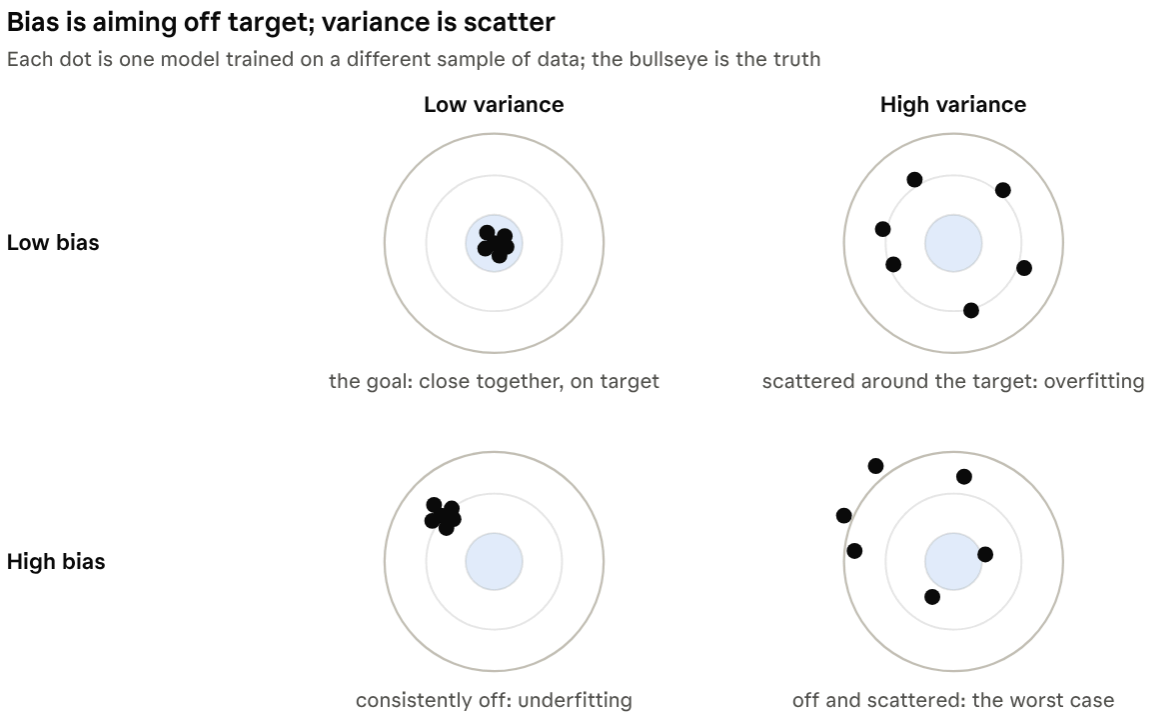


Figure 14.3 · Bias and variance pictured as darts

- **Bias** is being consistently off in the same direction. Your darts land close together, but all to the left of the bullseye. A model with high bias makes the same kind of mistake again and again, because it is too simple. Bias goes with underfitting.
- **Variance** is being scattered. On average your darts are around the centre, but each throw lands somewhere different. A model with high variance would change a lot if trained on a slightly different sample of data. Variance goes with overfitting.

The goal is low bias and low variance: darts close together and on target. Making a model more flexible usually lowers bias but raises variance. Making it simpler does the opposite. This tension is called the **bias–variance trade-off**.

Remember the random forest from Chapter 8? Each deep tree has low bias and high variance. Averaging many of them keeps the low bias and cancels much of the variance. That is why forests work so well.

How to fix overfitting

If your model overfits, there are many remedies. Most of them either give the model more evidence, or make it harder for the model to learn noise.

- **Get more data.** With more examples, the noise in any one example matters less, and the real pattern stands out. This is often the best fix, when it is possible.
- **Use a simpler model.** A shallower tree, a larger k in k-NN, fewer features.
- **Regularise.** Add a penalty for complexity, such as the ridge and lasso penalties on large weights from Chapter 5. The model must "pay" for every wiggle, so it only wiggles when the data really supports it.
- **Stop early.** For models that train step by step, such as boosting and neural networks, watch the validation error and stop when it starts rising.
- **Average many models.** Ensembles cancel out the variance of their members.
- **Remove useless features.** Every feature is another chance to find a coincidence.
- **Augment the data.** For images, flip, rotate or slightly recolour the training pictures. The model sees more variety and learns that a cat is still a cat when tilted.

How to fix underfitting

If your model underfits, it needs more power or better information.

- **Use a more flexible model.** Swap a straight line for a tree, a forest or a boosted model.
- **Add better features.** Often the pattern is there but hidden. Maya's model might underfit until she adds "is it a school holiday?".
- **Reduce regularisation.** If the penalty is too strong, the model cannot learn even the real pattern.
- **Train longer.** A neural network stopped too soon may not have learned enough yet.

Will more data help?

A **learning curve** shows how the training and validation errors change as you give the model more training data. It tells you whether collecting more data is worth the effort.

- If the two errors are far apart and the validation error is still falling as data grows, more data will probably help. The model is overfitting and needs more evidence.
- If the two errors have come together and both are high, more data will not help much. The model is underfitting. It needs better features or more flexibility, not more of the same examples.

Keep it as simple as possible

A principle from philosophy, called **Occam's razor**, says that when two explanations fit the facts equally well, prefer the simpler one. In machine learning: if a simple model and a complex model have similar validation scores, choose the simple one. It is easier to explain, faster to run, and usually more reliable when the world shifts a little.

Key ideas

- **Underfitting:** too simple; high error on training and new data.
- **Overfitting:** too flexible; learns noise; low training error but high error on new data.
- Compare training and validation errors. A large gap means overfitting.
- Validation error usually forms a U shape as complexity grows; aim for the bottom.
- **Bias** is consistent error (underfitting); **variance** is scattered error (overfitting).
- Fix overfitting with more data, simpler models, regularisation, early stopping and ensembles. Fix underfitting with better features and more flexible models.

Try it yourself

1. Describe overfitting using an example from sport, cooking or music.
2. A model has a training error of 2 and a validation error of 30. Another has 25 and 27. Which is overfitting? Which might be underfitting?
3. Why does a decision tree with no depth limit tend to overfit?
4. You collect twice as much data, but the validation error barely changes and stays close to the training error, both high. What should you try next?
5. Explain, using the dartboard picture, why averaging many deep trees helps.

Chapter 15 — Traps: Data Leakage, Imbalance and Baselines

The big idea: Some mistakes make a model look excellent during testing and then fail in real use. The most common are **data leakage** (the model secretly sees the answer), **imbalanced data** (the interesting class is rare), and **missing baselines** (nobody checked whether a simple method does just as well). Learning to spot them is what separates careful practitioners from beginners.

Too good to be true

Here is a rule that experienced people live by: **if a model's score looks too good to be true, it probably is**. Real-world prediction is hard. When a first attempt scores 99%, the right reaction is not celebration but suspicion. Something is usually wrong, and it is usually one of the traps in this chapter.

Trap 1: Data leakage

Data leakage happens when the training data contains information that will not be available when the model is used for real. The model learns to lean on that information, scores brilliantly in testing, and falls apart in practice.

Think of a student who can see the answer sheet reflected in the teacher's glasses during a practice test. Their practice scores are perfect. On the real exam, the teacher wears no glasses.

Leaks from the future

The most common leak is a feature that is only known **after** the moment of prediction.

Suppose Maya adds a column for "number of cones used that day". It predicts sales almost perfectly, because every ice cream needs a cone. But Maya makes her prediction the evening before. She will not know tomorrow's cone count until tomorrow is over. The feature is useless for the real task, and it ruins the model by drowning out the honest features.

Real cases are rarely so obvious. A hospital model predicting pneumonia might use "antibiotics prescribed" as a feature. It works wonderfully in testing, because doctors prescribe antibiotics after diagnosing pneumonia. In real use, the model is needed before the diagnosis, when that column is still empty.

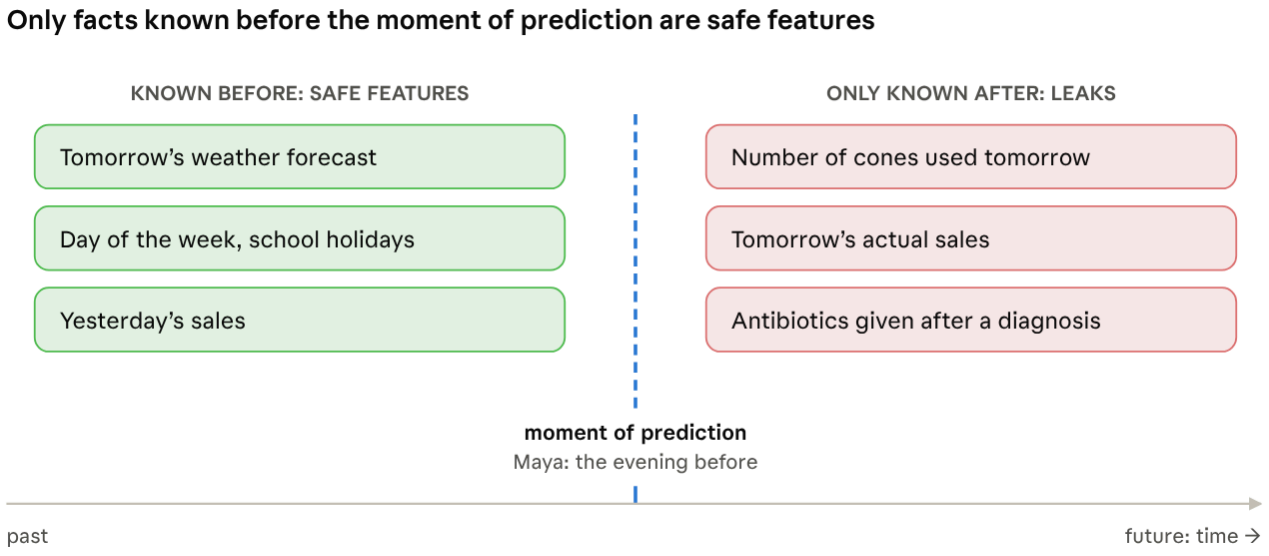


Figure 15.1 · Safe features and leaks around the moment of prediction

The test for every feature is one simple question: **at the moment I make the prediction, will I really know this?** If the answer is no, or "only sometimes", the feature must go.

Leaks through the split

Information can also leak from the test set into training. You met three ways this happens in earlier chapters.

- **Preparing with all the data.** Computing averages for scaling or filling gaps from the whole data set, test part included (Chapter 4).

- **Random splits of time data.** Training on days that come after the test days (Chapter 12).
- **Related examples on both sides.** Scans from the same patient, or duplicate rows, in both training and test (Chapter 12).

Leaks through hidden clues

The sneakiest leaks are clues that are linked to the label for accidental reasons. Researchers have found image models that seemed to detect illnesses from X-rays, but were partly recognising which hospital or machine took the picture. Sicker patients were more often scanned at certain hospitals, and the pictures carried small marks that gave the source away. The model learned the marks, not the medicine.

The fix is to ask what the model is really looking at. Use the explanation tools from Chapter 8. If a feature that should not matter turns out to be the most important one, investigate.

A placebo test for leaks

Here is a powerful trick. Take your training data, randomly shuffle the labels so they no longer match the features, and train again. The shuffled labels contain no real pattern, so an honest setup should score no better than the baseline. If your shuffled model still scores well on validation, information is leaking somewhere in your process. Medicine uses placebo pills to check that a treatment beats chance; this is the same idea.

Trap 2: Imbalanced data

In many important problems, the interesting class is rare. Fraud might be 1 payment in 1,000. A machine failure might happen once in 10,000 hours. Such data is called **imbalanced**.

Imbalance causes two problems.

- **Misleading scores.** As Chapter 13 showed, a model that always predicts “normal” scores 99.9% accuracy and catches nothing.
- **Lazy models.** During training, a model can get its error very low by mostly ignoring the rare class. It sees so few rare examples that learning them barely improves its overall score.

There are several remedies, and they are often combined.

- **Use the right metrics.** Look at precision and recall for the rare class, not accuracy.
- **Weight the classes.** Tell the training process that a mistake on a rare example counts, say, 100 times as much as a mistake on a common one.

- **Resample.** Repeat rare examples so they appear more often in training (**oversampling**), or use only part of the common examples (**undersampling**). Do this to the training data only; the test set must keep its real proportions, or its scores will mislead you.
- **Move the threshold.** A model might give a fraud case only a 10% probability, but if that is far above normal, flag it. Choose the threshold from the costs of each mistake (Chapter 6).
- **Collect more rare examples.** Often the best fix, when possible.

Trap 3: No baseline

It is surprisingly common for a team to build a complex model, report its score proudly, and never check how a simple method would do. Then someone tries “predict the same as last week” and finds it is just as good.

Before believing any model, compare it with baselines such as those in Chapter 13: always the average, always the most common class, same as yesterday, same as last year. A model must beat them clearly to earn its keep. If it does not, the extra complexity is buying nothing except risk.

Baselines are also an early warning system. If a model is **worse** than a simple baseline, something is broken: a bug, a leak in reverse, or a bad split.

Trap 4: The world changes

A model learns the world as it was when the data was collected. When the world changes, the model can quietly become wrong. This is called **distribution shift** or, when the relationship between features and label changes, **concept drift**.

In 2020, the COVID-19 pandemic changed shopping, travel and work almost overnight. Many models that predicted demand, traffic or spending stopped working, because the past no longer looked like the present. On a smaller scale, Maya’s model might drift if a new ice-cream shop opens next door, or if prices rise.

The defence is to **keep watching** after a model goes live. Track its errors over time, compare them with the test results, and retrain when performance slips.

Trap 5: Trying too many things

If you test enough ideas, one will look good by luck. Flip a coin ten times and getting eight heads is unusual. Repeat the experiment with a thousand coins and some will get eight heads easily, though none is special.

In machine learning, this happens when you try hundreds of features, models and settings and keep the best validation score. The winner is partly genuinely good and partly lucky. Its validation score is too optimistic. This is sometimes called the **garden of forking paths**: every choice you make based on the data is a fork, and with enough forks you can reach any result.

The defences are the ones from Chapter 12: decide on your metric in advance, keep a test set locked away, and confirm surprising results on fresh data before believing them.

Trap 6: Predictions that change the world

Sometimes the model's own predictions shape the data it will learn from later. This is a **feedback loop**.

A video platform recommends videos that a model thinks you will watch. You watch some of them, partly because they were recommended. The next model learns from those views and becomes even more sure. Over time, recommendations can narrow, and unrecommended videos never get a chance.

Feedback loops can also be unfair. If a model sends more inspectors to one neighbourhood, more problems will be recorded there, simply because more people are looking. The next model then learns that the neighbourhood has more problems. Chapter 19 returns to this.

A checklist for every project

Trap	Warning sign	Defence
Leak from the future	One feature is almost perfectly predictive	Ask of every feature: will I know this at prediction time?
Leak through the split	Test score far better than real-world results	Prepare data inside a pipeline; split by time and by group
Hidden clues	An odd feature is the most important	Inspect feature importance; run a shuffled-label placebo test
Imbalanced data	High accuracy, few rare cases caught	Use recall and precision; weight classes; move the threshold
No baseline	Nobody can say what "good" means	Compare with average, most common class, yesterday, last year
World changes	Errors grow after launch	Monitor errors over time; retrain regularly

Trap	Warning sign	Defence
Too many tries	Best result found after many attempts	Fix the metric first; lock away the test set; confirm on new data
Feedback loop	The model's choices shape its future data	Keep some random choices; watch for narrowing

Key ideas

- If a score looks too good to be true, it probably is. Investigate.
- **Data leakage** gives the model information it will not have in real use. Ask of every feature: will I know this at the moment of prediction?
- A **shuffled-label placebo test** should score no better than the baseline; if it does, something leaks.
- With **imbalanced data**, use precision and recall, weight the classes, resample the training data only, and tune the threshold.
- Always compare with **baselines**. A model worse than a baseline is broken.
- Models decay when the world changes; keep monitoring. Trying many ideas invites luck; confirm on fresh data.

Try it yourself

1. A model predicts whether a customer will cancel their phone contract next month. One feature is "number of calls to the cancellations department this month". Is this a leak? When might it be fine, and when not?
2. A friend's first model for predicting earthquakes scores 98% accuracy. What two questions would you ask first?
3. Only 1 in 200 loan applicants in a data set defaulted. Name three things you could do to build a useful model.
4. Explain the shuffled-label placebo test in your own words. What result would worry you?
5. Give an example, not from this chapter, of a model whose predictions could change the data it later learns from.

Part 4 — Going Deeper

Chapter 16 — How Machines Learn: Loss and Gradient Descent

The big idea: Training a model means adjusting its numbers, its parameters, so that a score of wrongness called the **loss** gets as small as possible. The most important way to do this is **gradient descent**: like a hiker in fog, feel which way is downhill, take a small step that way, and repeat.

Turning the shower tap

You step into a shower and the water is too cold. You turn the tap a little towards hot. Still cold, so you turn it a bit more. Now it is too hot, so you turn it back slightly. After a few small turns, the temperature is right.

You did not need a formula. You needed three things: a way to tell how wrong things are (too cold or too hot), a knob to adjust, and the habit of making small corrections in the right direction. That is exactly how most machine learning models learn.

The loss: a number for "how wrong"

To learn, a model needs a single number that says how badly it is doing on the training data. This number is called the **loss** (or the **cost**, or the **error function**). Smaller is better. Zero would be perfect.

You have already met two losses.

- For predicting numbers, the usual loss is the average **squared error**, as in least squares (Chapter 5).
- For yes/no predictions, the usual loss is **log loss**, which punishes confident mistakes (Chapter 6).

The model's parameters are its knobs: the slope and intercept of a line, the weights of a logistic regression, or the millions of weights in a neural network. Every setting of the knobs gives some loss. **Training is the search for the knob settings with the lowest loss.**

One knob: a valley

Take Maya's straight line from Chapter 5 and keep just one knob: the slope. For each possible slope, we can work out the loss over the 24 days. Plotting the loss against the slope gives a valley shape.

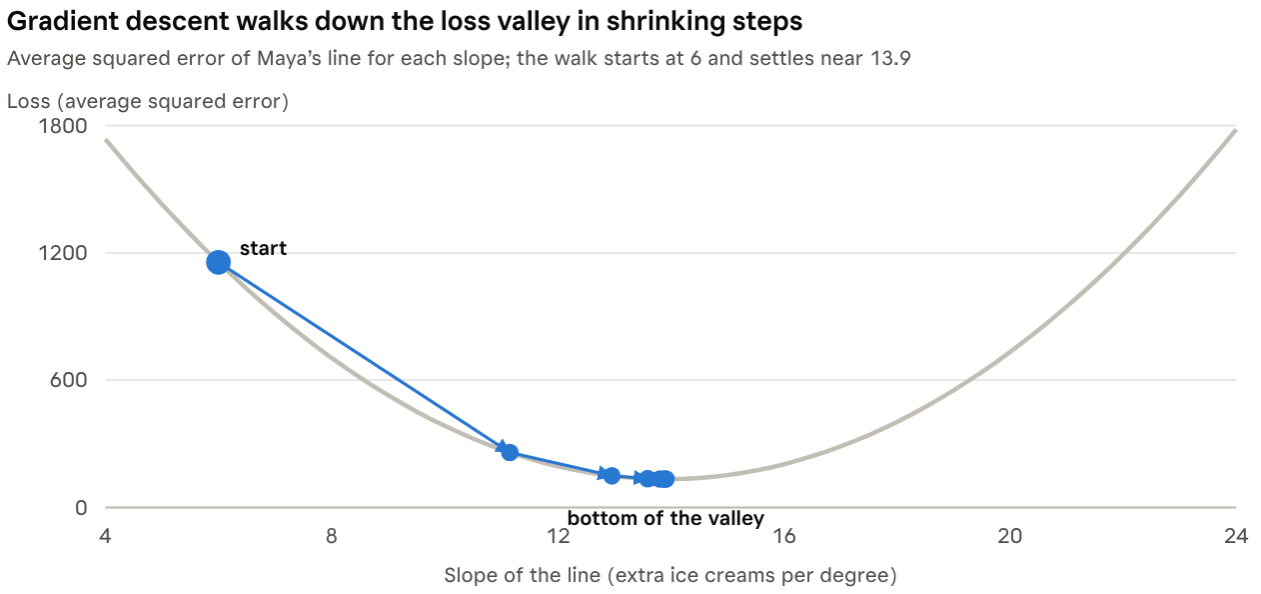


Figure 16.1 · The loss of Maya's line for each slope, computed from the 24 example days, with six gradient descent steps

A slope of 6 is far too shallow, and the loss is high. A slope of 22 is far too steep, and the loss is high again. The bottom of the valley, near a slope of 14, is the best line. That is the same answer least squares gave.

With one knob, you could simply try every slope and pick the lowest point. But real models have thousands or millions of knobs. Trying every combination would take longer than the age of the universe. We need a smarter way to find the bottom.

A hiker in the fog

Imagine you are on a hillside in thick fog. You want to reach the bottom of the valley, but you can only see your own feet. What do you do?

You feel the ground. If it slopes down to your left, you step left. Then you feel the ground again and take another step downhill. Step by step, you descend, without ever seeing the whole valley.

That is **gradient descent**. The **gradient** is the slope of the ground under your feet: which way is uphill, and how steep. The model works out the gradient of the loss for its current knob settings, then nudges every knob a little in the downhill direction. Then it repeats.

In Figure 16.1, the dots show the walk. It starts at a slope of 6, high on the valley wall. The ground is steep, so the first step is big. As the walk nears the bottom, the ground flattens, and the steps naturally get smaller. After a handful of steps, it settles at the bottom.

The training loop

Every model trained by gradient descent repeats the same loop.

1. **Predict.** Use the current knob settings to make predictions for the training examples.
2. **Measure.** Compare the predictions with the true labels and compute the loss.
3. **Find the slope.** Work out the gradient: for each knob, would turning it up make the loss bigger or smaller, and by how much?
4. **Step.** Nudge every knob a small amount in the direction that lowers the loss.
5. **Repeat** until the loss stops going down.

The clever part is step 3. For the models in this book, there are efficient mathematical methods to compute the gradient for every knob at once, even millions of them. For neural networks, this method is called **backpropagation** (Chapter 17). You do not need to do it by hand; machine learning software does it automatically.

The learning rate: how big a step?

How big should each step be? The step size is set by a number called the **learning rate**. It is one of the most important hyperparameters in machine learning, and getting it wrong is very common.

Figure 16.2 · Three learning rates on the same loss valley

- **Too small:** each step is tiny. The walk heads the right way but crawls. After six steps it is still on the valley wall. Training would take far too long.
- **About right:** the walk reaches the bottom quickly and settles there.
- **Too big:** each step overshoots the bottom and lands on the opposite wall. The walk bounces from side to side. If the learning rate is bigger still, each bounce lands higher than the last, and the loss explodes instead of shrinking.

In practice, people try a few learning rates, such as 0.1, 0.01 and 0.001, and keep the one that gives the lowest validation loss. Many training methods also make the learning rate shrink as training goes on: big steps early to cover ground, small steps later to settle precisely. This is called a **learning rate schedule**.

Many knobs: a landscape

With two knobs, the loss is no longer a valley line but a whole landscape, with hills, ridges and basins. With a million knobs, it is a landscape in a million dimensions, which nobody can picture. But the hiker's rule still works: at each point, find the downhill direction for every knob together, and step that way.

Some landscapes have more than one valley. The hiker might walk into a small valley high up on the mountain, a **local minimum**, and stop there, thinking it is the bottom. For simple models like linear and logistic regression, the landscape is a single smooth bowl, so this cannot happen. For neural networks it can, but in practice the huge number of dimensions usually offers a way further down, and the valleys found are good enough.

Small batches: asking a few people for directions

Computing the exact gradient means looking at every training example, which is slow when there are millions. A common shortcut is to estimate the gradient from a small random handful of examples, called a **mini-batch**, often 32 to a few hundred examples.

It is like asking a few people in the street for directions instead of surveying the whole town. Each answer is a bit rough, but you get one quickly and can start walking. Over many steps, the rough directions average out and lead downhill. This method is called **stochastic gradient descent**, or **SGD**. "Stochastic" means involving randomness.

One full pass through all the training data, batch by batch, is called an **epoch**. Training a neural network often takes dozens or hundreds of epochs.

The randomness of mini-batches even helps a little: the slight jiggle in each step can shake the hiker out of shallow local valleys.

Smarter hikers

Researchers have developed improved versions of gradient descent that adjust their step size for each knob automatically. The most widely used is called **Adam**. It keeps a running memory of recent slopes, like a ball rolling downhill that builds up speed in a steady direction and slows when the direction keeps changing. For beginners, Adam with its default settings is a sensible starting point for neural networks.

Watching training happen

During training, people plot the loss after each epoch. This picture is called a **loss curve** or **training curve**.

- If the training loss is falling steadily, learning is working.
- If it is jumping around wildly or rising, the learning rate is probably too big.
- If it falls very slowly, the learning rate may be too small.
- If the training loss keeps falling but the validation loss starts rising, the model has begun to overfit. Stop there. That is **early stopping**, from Chapters 8 and 14.

Which models learn this way?

- **Linear regression** can use gradient descent, though for small problems there is a faster exact shortcut.
- **Logistic regression** has no exact shortcut and is usually trained by gradient descent or a close relative.
- **Neural networks** are trained almost entirely by gradient descent with backpropagation.
- **Gradient boosting** (Chapter 8) uses the same idea in a different form: each new tree is a step downhill on the loss, which is where the name comes from.
- **Decision trees and k-NN** do not use gradient descent. Trees use greedy splitting; k-NN does no training at all.

Key ideas

- The **loss** is a single number for how wrong the model is on the training data.
- Training means finding the parameter settings with the lowest loss.
- **Gradient descent** repeatedly finds the downhill direction (the **gradient**) and takes a small step that way.
- The **learning rate** sets the step size: too small is slow, too big overshoots or explodes.
- **Stochastic gradient descent** estimates the slope from small **mini-batches**; one pass through the data is an **epoch**.
- Watch training and validation loss curves; stop when validation loss starts rising.

Try it yourself

1. Describe gradient descent using an example of your own, such as tuning a radio or adjusting a recipe.
2. In Figure 16.1, why do the steps get smaller near the bottom, even though the learning rate stays the same?
3. Your training loss goes up and down wildly from one epoch to the next. What would you change first?
4. A data set has 10,000 examples and the mini-batch size is 100. How many steps are in one epoch?
5. Why is it impossible to train a model with a million parameters by trying every combination of values?

Chapter 17 — Neural Networks

The big idea: A **neural network** is built from many tiny, simple units called **neurons**, arranged in **layers**. Each neuron weighs its inputs, adds them up, and passes the result through a simple bend. Stacking many layers lets the network learn very complicated patterns. It is trained with the gradient descent of Chapter 16.

Inspired by the brain, but not a brain

Your brain contains billions of nerve cells, also called neurons. Each one receives signals from others, and if the combined signal is strong enough, it fires and sends a signal on. Early researchers, including Frank Rosenblatt with his perceptron in 1958, wondered whether a machine built from simple units like this could learn.

Modern neural networks borrow the name and the rough idea, but they are not models of the brain. They are mathematical tools: large collections of numbers that are adjusted by gradient descent. Think of the brain as the inspiration, the way birds inspired aeroplanes. Aeroplanes do not flap their wings.

One neuron: a tiny decision-maker

Let us start with a single artificial neuron. Imagine it has to decide whether you will go to the beach today. It receives three inputs, each a number:

- Is it sunny? (1 for yes, 0 for no)
- Is it warm? (1 for yes, 0 for no)
- Is your friend going? (1 for yes, 0 for no)

The neuron gives each input a **weight**: how much that input counts. Sunshine might count a lot, with a weight of 3. Warmth counts somewhat, with a weight of 2. Your friend counts a little, with a weight of 1.5. It also has a **bias**, a starting lean before any inputs arrive. A bias of −4 means “by default, I would rather stay home”.

The neuron multiplies each input by its weight, adds them all up, and adds the bias. On a sunny, warm day when your friend is not going, that is 3 + 2 + 0 − 4 = 1. A positive total leans towards yes.

Finally, the neuron passes the total through a simple bend called an **activation function**. If the activation is the S-curve from Chapter 6, a total of 1 becomes a probability of about 73%. You will probably go.

Figure 17.1 · One artificial neuron, worked through with the beach example

If this looks familiar, it should. A single neuron with an S-curve activation is exactly logistic regression. The weights are the same kind of weights; the bias is the intercept. A neural network is what you get when you connect many of these little decision-makers together.

Why the bend matters

The most popular activation function today is wonderfully simple. It is called **ReLU** (short for rectified linear unit), and its rule is: **if the total is negative, output zero; otherwise, output the total unchanged**. That is all.

Why bother with a bend at all? Because without one, stacking neurons is pointless. A weighted sum of weighted sums is just another weighted sum. Ten layers without bends collapse into the same straight-line model as one layer. The bends let each layer add a new kink, and many kinks together can trace almost any shape.

Think of drawing a curve using only straight sticks. One stick gives you a straight line. With many short sticks joined at angles, you can trace a circle, a wave or a coastline. Each ReLU neuron supplies one joint.

Layers

Neurons in a network are arranged in layers.

- The **input layer** holds the features: one unit per feature. It does no calculation; it just passes the numbers in.
- **Hidden layers** sit in the middle. Each neuron takes all the outputs of the layer before, weighs them, adds them, and applies its bend. They are called hidden because we never see their values directly in the data.
- The **output layer** gives the prediction.

Figure 17.2 · A small fully connected network with two hidden layers

In this small network, every unit in one layer connects to every unit in the next. Such layers are called **fully connected** or **dense** layers. A network with many hidden layers is called **deep**, and training deep networks is called **deep learning**. There is no exact cut-off, but networks with dozens or even hundreds of layers are common today.

What the hidden layers learn

Nobody tells the hidden neurons what to look for. They work it out during training, and what they learn tends to build up in stages.

In a network that recognises faces in photos, researchers looking inside often find something like an assembly line. Neurons in the first layers respond to simple edges and spots of colour. Middle layers combine edges into shapes such as curves and corners. Later layers combine shapes into parts such as eyes, noses and ears. The final layers combine parts into whole faces.

Each layer builds new, more useful features out of the features from the layer below. This is the great strength of deep learning: it does its own **feature engineering**. In Chapter 3, Maya had to invent features such as "is it a weekend" by hand. A deep network, given enough data, can discover useful features by itself.

How many knobs?

Every connection has a weight, and every neuron has a bias. Together they are the network's parameters. They add up fast.

A tiny network with 3 inputs, one hidden layer of 4 neurons and 1 output has $3 \times 4 = 12$ weights into the hidden layer, 4 hidden biases, 4 weights to the output and 1 output bias. That is 21 parameters. A network for recognising photos might have tens of millions. The large language models behind modern chat assistants have billions.

More parameters mean more flexibility, but also more risk of overfitting and a greater need for data and computing power.

How a network learns: passing the blame backwards

Training follows the loop from Chapter 16.

1. **Forward pass.** Feed a batch of examples through the network, layer by layer, to get predictions.
2. **Loss.** Compare the predictions with the true labels.
3. **Backward pass.** Work out how much each weight contributed to the error.
4. **Update.** Nudge every weight a little in the direction that lowers the loss.

The backward pass uses **backpropagation**. Think of a restaurant where a customer complains that the soup is too salty. The waiter passes the complaint back to the chef. The chef works out how much was their own seasoning and how much came from the stock, and passes part of the blame back to whoever made the stock. Each person adjusts in proportion to their share of the blame. Backpropagation passes the error backwards through the layers in the same way, so every weight learns how it should change.

Choosing the output

The output layer depends on the question.

Question	Output layer	Loss
Predict a number (regression)	One neuron, no bend	Squared error
Yes or no (binary classification)	One neuron with the S-curve	Log loss
One of several classes	One neuron per class, with **softmax** to turn them into probabilities that add up to 100%	Log loss (cross-entropy)

Keeping networks from overfitting

Because neural networks are so flexible, they overfit easily. Besides the general remedies of Chapter 14, a few tricks are special to networks.

- **Dropout.** During training, switch off a random selection of neurons at each step, often a fifth to a half of them. Picture a sports team that practises with random players missing. No player can become indispensable, so the whole team learns to cope. At prediction time, all neurons work together.
- **Weight decay.** A penalty for large weights, the same idea as ridge regression in Chapter 5.
- **Early stopping.** Watch the validation loss and stop when it starts rising.
- **Data augmentation.** For images and sound, create varied copies of the training examples: flipped, cropped, slightly darker, slightly noisier.

The settings you choose

Hyperparameter	What it controls	A sensible start
Number of hidden layers	How deep the network is	1 to 3 for tables of data
Neurons per layer	How wide each layer is	32 to 256
Activation	The bend in each neuron	ReLU
Learning rate	Step size of gradient descent	0.001 with the Adam method
Batch size	Examples per step	32 to 256
Epochs	Passes through the data	Until validation loss stops improving

Hyperparameter	What it controls	A sensible start
Dropout rate	Share of neurons switched off in training	0.1 to 0.5

Always **scale your inputs** before training a neural network (Chapter 4). Unscaled features make the loss landscape lopsided, and gradient descent struggles.

When to use a neural network

Neural networks are the best tool known for many kinds of unstructured data: images, sound, video and text. They power face unlock, voice assistants, translation and chatbots.

For ordinary tables of data, like Maya's notebook, they are often not the best choice. Gradient-boosted trees (Chapter 8) are usually as accurate or better on tables, need less data, less tuning and less computing power, and are easier to explain. A good habit is to try boosted trees first on tabular data, and reach for neural networks when the data is images, sound, text, or truly huge.

There is a famous result that says a network with enough neurons can approximate almost any pattern. That sounds like magic, but it cuts both ways. A network that can fit any pattern can also fit any noise. Its power is only useful with enough data and honest testing.

Key ideas

- A **neuron** weighs its inputs, adds a **bias**, and applies an **activation function**. One neuron with an S-curve is logistic regression.
- **ReLU** (zero if negative, otherwise unchanged) is the most common activation. Without bends, layers collapse into a straight line.
- Networks have an **input layer**, **hidden layers** and an **output layer**. Many hidden layers make a **deep** network.
- Hidden layers build their own features, from simple to complex.
- Training uses gradient descent; **backpropagation** passes the blame for the error backwards through the layers.
- Fight overfitting with **dropout**, weight decay, early stopping and data augmentation.
- Neural networks shine on images, sound and text; boosted trees often win on tables.

Try it yourself

1. Using the beach neuron (weights 3, 2 and 1.5, bias −4), compute the total for a cloudy, warm day when your friend is going. Does it lean towards yes or no?

2. What does ReLU output for totals of −3, 0 and 2.5?
3. Count the parameters in a network with 2 inputs, one hidden layer of 3 neurons, and 1 output.
4. Explain in your own words why a network without activation functions cannot learn curved patterns.
5. Maya wants to predict daily sales from her notebook of 730 days. Would you start with a neural network or gradient boosting? Why?

Chapter 18 — A Tour of Deep Learning

The big idea: Different kinds of data need different network designs. **Convolutional networks** look at images through small sliding windows. **Transformers** read text by letting every word pay attention to every other word. The biggest modern models are first trained on enormous amounts of data, then adapted to specific jobs.

This chapter is a guided tour, not a manual. The goal is to understand the main ideas behind the AI you read about in the news, and to see that they grow out of everything earlier in this book.

Seeing: convolutional networks

A photo is a grid of numbers (Chapter 3). You could feed every pixel into a plain neural network like the one in Chapter 17. But a phone photo has millions of pixels, so every neuron in the first layer would need millions of weights. Worse, the network would have to learn separately that a cat in the top left corner and a cat in the bottom right corner are both cats.

Convolutional neural networks (**CNNs**) solve both problems with one idea. Instead of looking at the whole picture at once, they slide a small window across it. The window is called a **filter**, and it is often just 3 by 3 pixels. Each filter is a little pattern detector, for example for a vertical edge. Wherever the pattern appears under the window, the filter lights up.

Figure 18.1 · A vertical-edge filter applied to an 8 by 8 picture (the result i…

Think of searching a large page of text for the word “cat” with a magnifying glass. You use the same glass everywhere on the page, and you notice the word wherever it is. A filter is that magnifying glass. Because the same small filter is reused at every position, it has only a handful of weights, and it finds its pattern anywhere in the picture.

A CNN has many filters in each layer, and many layers. Just as Chapter 17 described, early layers find edges and colour spots, middle layers find textures and shapes, and later layers find eyes, wheels and leaves. Between layers, **pooling** steps shrink the picture by summarising small areas, so later layers see a wider view. At the end, ordinary dense layers turn the findings into a prediction such as “cat, 94%”.

In 2012, a CNN won the ImageNet picture-recognition contest by a large margin, sorting photos into a thousand categories far better than earlier methods. That result started the deep learning boom. Today, CNNs and their successors read medical scans, check products on factory lines, recognise faces and help cars see the road.

Standing on the shoulders of giants: transfer learning

Training a big image network from scratch needs millions of labelled photos and a lot of computing power. Most people do not have either. Luckily, they do not need to.

A network trained on millions of everyday photos has already learned general skills: edges, textures, shapes, parts of objects. Those skills are useful for almost any image task. So you can take such a **pretrained** network, keep most of its layers, and retrain only the last few layers on your own small data set. This is called **transfer learning**, and adjusting a pretrained model to a new task is called **fine-tuning**.

It is like hiring an experienced chef to cook a new cuisine. They do not need to learn how to hold a knife or control heat; they only learn the new recipes. A clinic with a few thousand labelled skin photos can build a useful classifier this way, where training from scratch would fail.

Order matters: sequences

Some data comes in sequences where order matters: words in a sentence, notes in a melody, readings from a sensor over time. “Dog bites man” and “man bites dog” contain the same words but mean very different things.

An early answer was the **recurrent neural network** (**RNN**). It reads a sequence one step at a time and carries a memory forward, like a person reading a sentence word by word. Improved versions such as the **LSTM** were better at remembering. But RNNs still struggled with long texts, where something at the start matters at the end, and their one-step-at-a-time reading was slow to train.

Words as numbers: embeddings

Before a network can read text, words must become numbers. Counting words, as in Chapter 3, loses meaning: “glad” and “happy” look as different as “glad” and “toaster”.

The modern answer is the **embedding**: each word is turned into a list of a few hundred numbers, learned from data so that words used in similar ways end up with similar lists. You can picture each list as a position on a map. Words with related meanings sit in the same neighbourhood.

Figure 18.2 · A simplified word-embedding map (made-up positions)

Embeddings capture surprising relationships. In a well-known result from 2013, the direction from “man” to “woman” on the map was roughly the same as the direction from “king” to “queen”. Nobody programmed that; it emerged from how the words are used in text.

In practice, models do not always work with whole words. They split text into **tokens**, which are common words or pieces of words. “Unbelievable” might become “un”, “believ” and “able”. This lets a model handle new or rare words by their parts.

Paying attention: transformers

In 2017, researchers introduced a design called the **transformer**. Its key idea is called **attention**.

Consider this sentence: “The dog did not cross the road because it was too tired.” What does “it” refer to? You know immediately: the dog, not the road, because roads do not get tired. To work that out, your mind connected “it” with “dog” and “tired”. Change the last word to “wide” and “it” now means the road.

Attention lets a model do the same thing. For every token, it asks: which other tokens in the text matter most for understanding this one? It then mixes in information from those tokens, weighted by how much they matter. Picture each word holding a highlighter and marking the other words it needs. A transformer repeats this through many layers, building richer and richer understanding.

Transformers have two big advantages over RNNs. They connect any two words directly, however far apart, so long texts are easier. And they process all the words at once rather than one by one, which suits modern computer chips and makes training on huge amounts of text practical.

Large language models

A **language model** is trained to predict the next token in a text. Show it “The ice cream melted in the hot…”, and it should give a high probability to “sun”. This is the self-supervised learning of Chapter 2: the right answer is just the next word in real text, so no

human labels are needed, and the model can learn from vast collections of books, articles and websites.

Predicting the next word well turns out to require a great deal of knowledge: grammar, facts, common sense, even some reasoning. When transformers were scaled up to billions of parameters and trained on enormous text collections, they became **large language models** (**LLMs**). They can write, summarise, translate, answer questions and write computer code.

A raw language model only continues text. To become a helpful assistant, it goes through further stages.

Figure 18.3 · The three training stages behind a chat assistant

First comes **pretraining** on a huge collection of text, which teaches language and knowledge. Then **fine-tuning** on many examples of good conversations teaches it to follow instructions and answer helpfully. Finally, people compare pairs of answers and say which is better, and the model is trained to produce the preferred kind. This last step is called **reinforcement learning from human feedback** (**RLHF**), linking back to the reinforcement learning of Chapter 2.

What language models can and cannot do

LLMs are remarkable tools, but everything in Part 3 still applies to them.

- **They can be confidently wrong.** A model predicts likely-sounding text, and likely-sounding is not the same as true. Invented facts, quotes or references are often called **hallucinations**. Check anything important.
- **They learn from their data, flaws included.** Biases and errors in the training text can appear in the answers (Chapter 19).
- **Their knowledge has a cut-off.** A model knows little about events after its training data was collected, unless it is connected to search tools or given the information.
- **They can be very good at some tasks and surprisingly poor at others.** Test them on your own task, with honest examples, before relying on them.

Making new things: generative models

Models that create new content, rather than just labelling it, are called **generative models**.

Many of today's image generators use **diffusion**. During training, the model is shown pictures with more and more random noise added, like static on an old television, and it learns to remove a little noise at a time. To create a new picture, it starts from pure noise

and removes noise step by step, guided by a text description, until a clear picture appears. It is a bit like a sculptor revealing a statue from a rough block of stone.

An older approach, the **generative adversarial network** (**GAN**), trains two networks against each other: one makes fake pictures, the other tries to spot the fakes. Each pushes the other to improve, like a forger and a detective.

The same ideas now produce music, speech, video and 3D shapes. Models that handle several kinds of data together, such as text and images, are called **multimodal**.

Which design for which data?

Kind of data	Common design	Example uses
Tables of numbers and categories	Gradient-boosted trees, or a small dense network	Sales, prices, risk scores
Images	Convolutional networks, and newer vision transformers	Medical scans, photo search, quality checks
Text	Transformers	Translation, chat assistants, summaries, search
Sound and speech	Convolutional networks and transformers on sound turned into numbers	Voice assistants, captions, music tagging
Time series	Boosted trees with good features, RNNs or transformers	Demand forecasting, sensor monitoring
Creating new images	Diffusion models	Illustration, design ideas

The cost of scale

The largest models need thousands of specialised computer chips running for weeks, at great expense and with a large energy use. Most people and businesses will never train such a model. Instead, they use pretrained models through software or online services, and fine-tune smaller models for their own tasks. Understanding the fundamentals in this book is what lets you use those tools wisely.

Key ideas

- **Convolutional networks** slide small **filters** over images to find patterns anywhere in the picture.
- **Transfer learning** reuses a **pretrained** network and **fine-tunes** it on a small data set.

- **Embeddings** turn words or **tokens** into lists of numbers where similar meanings sit close together.
- **Transformers** use **attention** so every token can draw on the tokens that matter most to it.
- **Large language models** learn by predicting the next token, then are fine-tuned and trained with human feedback.
- LLMs can **hallucinate**; check important facts. **Diffusion** models create images by removing noise step by step.

Try it yourself

1. Why does reusing the same small filter across the whole image help a CNN? Give two reasons.
2. A bakery has 500 labelled photos of its cakes and wants a model that recognises each type. Why is transfer learning a good idea here?
3. In “The trophy did not fit in the suitcase because it was too big”, what does “it” mean? What if the last word were “small”? How does this relate to attention?
4. Why can a language model learn from text without anyone labelling it?
5. A chat assistant gives you a book reference for your school project. What should you do before using it, and why?

Chapter 19 — Fairness, Ethics and Responsibility

The big idea: Machine learning models make or shape decisions about real people. They can repeat and even amplify unfairness hidden in their data, expose private information, make mistakes nobody can explain, and be put to harmful uses. Building responsibly means asking who could be harmed, measuring it, and keeping people accountable.

A mirror of the past

A model learns from examples of the past. If the past was unfair, the model learns the unfairness too, and then applies it quickly, at scale, with an air of objectivity. A model is a mirror, and sometimes we do not like what it shows.

In 2018, news reports described an experimental hiring tool at a large technology company. It had learned from ten years of past hiring, during which most people hired for technical roles were men. The tool reportedly learned to mark down CVs that mentioned things like a women’s sports club. Nobody programmed it to do that. It found the pattern in the data. The company dropped the tool.

This is the uncomfortable truth behind many problems in this chapter: **the model did exactly what it was trained to do**. The fault lies in what it was trained on, and in the decision to use it without checking.

Where bias comes from

Unfairness can enter at every step of a project, not only in the data.

Figure 19.1 · Where bias can enter a machine learning project

- **Historical bias.** The world itself was unequal, so accurate data about it carries that inequality, as in the hiring example.
- **Representation bias.** Some groups are missing or rare in the data. A 2018 study of commercial face-analysis systems, called Gender Shades, found they were far less accurate at identifying the gender of darker-skinned women than of lighter-skinned men. The training data had too few examples of some groups.
- **Measurement bias.** The label is only a stand-in for what we really care about. A 2019 study in the journal Science looked at a widely used health-care algorithm that used past medical spending as a stand-in for medical need. Because less money had historically been spent on Black patients with the same level of illness, the algorithm underestimated how sick they were.
- **Deployment bias.** A model built for one setting is used in another, such as a model trained on adults applied to children, or on one country applied to another.
- **Feedback loops.** The model's decisions change what data is collected next, locking patterns in (Chapter 15).

Removing the sensitive column is not enough

A natural first idea is: just delete the columns for gender, ethnicity or age, and the model cannot be biased. Unfortunately, it does not work.

Other features often act as stand-ins, called **proxies**. A postal code can reveal a lot about ethnicity and income. A first name can hint at gender. The kinds of shops someone visits, or the hobbies on their CV, can carry the same information. A model that is never shown gender can still learn to treat men and women differently through these proxies.

Worse, deleting the sensitive column makes it harder to check for bias, because you can no longer measure results by group. Fairness work usually needs the sensitive information kept aside for testing, even when it is not used for prediction.

Measuring fairness

The practical first step is to measure. Take the metrics from Chapter 13 and compute them separately for each group.

- Is the **error rate** similar for every group?
- Is the **false positive rate** similar? For example, are innocent people from one group flagged as risky more often?
- Is the **false negative rate** similar? Are qualified people from one group turned away more often?
- Is the **approval rate** similar?

Here is a hard fact: these different ideas of fairness often cannot all be true at once. If two groups have different underlying rates of the outcome, a model generally cannot give both groups equal approval rates and equal error rates at the same time. Researchers have proved this mathematically.

So choosing a fairness measure is not a technical decision. It is a decision about values, and it should involve the people affected, not only the engineers. What the engineer can do is make the trade-offs visible and measured, rather than hidden.

Privacy

Machine learning often runs on personal data: purchases, locations, messages, health records, photos. That brings duties.

- **Consent and purpose.** People should know what their data is used for, and it should not be quietly reused for something else.
- **Collect less.** Only gather the data you really need. Data you do not have cannot leak.
- **Anonymising is harder than it looks.** Removing names is not enough. Researchers have shown again and again that a few details, such as postal code, birth date and gender together, can pick out many individuals in supposedly anonymous data.
- **Models can remember.** A large model can sometimes repeat pieces of its training data, including personal details. Sensitive data needs special care before it is used for training.
- **The law.** Many places have data protection laws. The European Union's General Data Protection Regulation (GDPR), for example, gives people rights over their personal data and limits how it may be used.

Explanation and accountability

When a model refuses someone a loan, a job interview or a benefit, that person deserves to know why, and to be able to challenge the decision.

- For **high-stakes decisions**, prefer models that can be explained, such as linear models or small trees, unless a more complex model is clearly better. Use the explanation tools of Chapter 8 when complex models are used.
- Keep a **human in the loop** for important decisions: someone who reviews the model's recommendation, can overrule it, and is responsible for the outcome.
- Provide a way to **appeal**. Every model makes mistakes; the question is whether those mistakes can be found and fixed.
- **Document** the model. A short **model card** describes what the model is for, what data it was trained on, how it was tested, how well it works for different groups, and where it should not be used. A matching **datasheet** describes the data set itself.

Misuse and wider effects

Some harms come not from mistakes but from what the technology is used for.

- **Deepfakes.** Generative models can make realistic fake videos, voices and images of real people, which can be used for fraud, harassment or misinformation.
- **Surveillance.** Face recognition and tracking can be used to monitor people without their knowledge.
- **Misinformation at scale.** Language models can produce convincing false articles, reviews and messages in huge volumes.
- **Energy and resources.** Training and running very large models uses a lot of electricity and water for cooling. It is worth asking whether a smaller model would do the job.
- **Work.** Automation changes jobs. Some tasks disappear, new ones appear, and the benefits and costs are not shared evenly.

Questions to ask before you deploy

Before putting any model into real use, work through questions like these.

1. **Who is affected** by this model's decisions, and were they consulted?
2. **What happens when it is wrong?** Which kind of mistake is worse, and for whom?
3. **Does it work equally well for different groups?** Have you measured it?
4. **Could any feature be a proxy** for a sensitive characteristic?
5. **Is the training data representative** of the people the model will be used on?
6. **Can its decisions be explained** to the person affected, and can they appeal?
7. **Who is responsible** for the model, and who will monitor it after launch?
8. **Could it be misused,** and what safeguards exist?
9. **Is machine learning needed at all,** or would a simpler, clearer rule be better?

Back to the ice-cream shop

Maya's sales forecasts are low-stakes. If the model is wrong, she has a bit too much or too little ice cream. But suppose she starts using a model to decide which customers get special discounts, or which job applicants get an interview for a summer job. Now the model affects people's money and chances. Every question in the list above becomes important, even for a small shop.

The lesson is general. The same algorithm can be harmless in one use and harmful in another. Responsibility comes from looking at **how a model is used**, not only at how it is built.

Key ideas

- Models learn from the past, including its unfairness; they then apply it at scale.
- Bias enters through history, missing groups, poor stand-in labels, misuse in new settings and feedback loops.
- Deleting sensitive columns does not remove bias; **proxies** carry the information. Keep sensitive data aside for testing.
- Measure performance **by group**. Different fairness definitions can conflict; choosing between them is a values decision.
- Protect **privacy**: consent, collect less, and remember that anonymising is hard.
- Keep humans accountable: explanations, appeals, **model cards**, monitoring.
- Ask who could be harmed **before** deploying, and whether machine learning is needed at all.

Try it yourself

1. A bank trains a loan model on 30 years of past approvals. Why might it be unfair even if it never sees the applicants' ethnicity?
2. Give two features that could act as proxies for age.
3. A face recognition system is 99% accurate overall but much less accurate for one group. Why does the overall number hide a problem?
4. A school wants to use a model to predict which students might drop out, so it can offer help. List three questions from the checklist you would ask first.
5. Why might a company keep data about gender even though its model does not use it for predictions?

Chapter 20 — Your First Project, Step by Step

The big idea: In this chapter you build a complete machine learning project in Python, from raw data to a tested model that predicts tomorrow's ice-cream sales. Every step uses an idea from earlier in the book, and every line of code is explained.

Before you start

You need Python and three free libraries: **pandas** for tables of data, **NumPy** for numbers, and **scikit-learn** for machine learning. Appendix A explains how to install them. If you have never programmed, do not worry. You can follow the steps by copying the code exactly, and the explanations tell you what each part does.

The code in this chapter was run with Python 3.9, pandas 2.3 and scikit-learn 1.6. Newer versions should give the same or very similar results.

Here is the plan. It follows the recipe from Chapter 1.

1. Make a practice data set that behaves like Maya's notebook.
2. Load it and look at it.
3. Choose features and split the data by time.
4. Score a simple baseline.
5. Try three different models.
6. Tune one setting on the validation data.
7. Take the final exam on the test data, once.
8. See which features the model relies on.
9. Predict tomorrow.

Step 1: Make the practice data

Real shop data is private, so we will create a realistic practice version. The program below invents 730 days, from January 2024 to December 2025, for a seaside shop. It sets the temperature by season, adds rain, weekends and school holidays, and computes sales from a hidden pattern plus random noise. Because we wrote the hidden pattern ourselves, we can later check whether the model found something sensible.

Save this as a file called `make_data.py` and run it.

```
import numpy as np
import pandas as pd

rng = np.random.default_rng(42)

dates = pd.date_range("2024-01-01", periods=730, freq="D")
```

```
day_of_year = dates.dayofyear.to_numpy()

# Seaside temperatures: cool in winter, warm in summer, plus day-to-day
wobble
temperature = 18 - 9 * np.cos(2 * np.pi * (day_of_year - 15) / 365) +
rng.normal(0, 2.5, 730)
temperature = temperature.round(1)

# Rain is more likely in winter
rain_chance = 0.35 + 0.15 * np.cos(2 * np.pi * (day_of_year - 15) / 365)
rain = (rng.random(730) < rain_chance).astype(int)

weekend = (dates.dayofweek >= 5).astype(int)
school_holiday = (((dates.month == 7) | (dates.month == 8))
                  | ((dates.month == 12) & (dates.day >= 20))
                  | ((dates.month == 4) & (dates.day <= 14))).astype(int)

# The hidden "true" pattern, plus noise
warmth = np.clip(temperature - 12, 0, None)
sales = (20 + 11 * warmth - 0.12 * warmth ** 2
         + 55 * weekend + 45 * school_holiday
         - rain * (10 + 3.5 * warmth)
         + rng.normal(0, 18, 730))
sales = np.clip(sales, 0, None).round().astype(int)

data = pd.DataFrame({
    "date": dates.strftime("%Y-%m-%d"),
    "temperature": temperature,
    "rain": rain,
    "weekend": weekend,
    "school_holiday": school_holiday,
    "sales": sales,
})
data.to_csv("icecream.csv", index=False)
print("Saved", len(data), "days to icecream.csv")
```

You should see `Saved 730 days to icecream.csv`, and a new file appears in the same folder.

A few things to notice in the hidden pattern. Sales rise with warmth but level off on very hot days (the part with `warmth ** 2`). Weekends and school holidays add customers. Rain costs more on warm days than on cool ones, which is an interaction (Chapter 7). And `rng.normal(0, 18, 730)` adds noise that no model can predict. Starting the random generator with the fixed number 42 means you get exactly the same data every time you run it.

Step 2: Load the data and look

Now start a second file called `project.py`. All the remaining steps go into this file, one after another. First, load the data and look at it, as Chapter 4 advised.

```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor,
HistGradientBoostingRegressor
from sklearn.metrics import mean_absolute_error, r2_score
from sklearn.inspection import permutation_importance

# Step 2: load and look
data = pd.read_csv("icecream.csv", parse_dates=["date"])
print(data.head())
print(data.describe().round(1))
print(data.isna().sum())
```

The first lines bring in the tools we need. `pd.read_csv` loads the file into a table, which pandas calls a **DataFrame**, and `parse_dates` tells it that the date column holds dates. `head()` shows the first five rows:

```
        date  temperature  rain  weekend  school_holiday  sales
0 2024-01-01         10.0     0        0               0     14
1 2024-01-02          6.6     1        0               0     19
2 2024-01-03         11.1     1        0               0      8
3 2024-01-04         11.5     1        0               0      3
4 2024-01-05          4.3     0        0               0     46
```

`describe()` summarises each column. In this data, temperatures run from 3.4 to 32.2 degrees, with an average of 17.9. Sales run from 0 to 305, with an average of about 99. The checks `isna().sum()` count missing values in each column; here every count is 0, because we made the data ourselves. With real data, this is where you would start the cleaning work of Chapter 4.

Step 3: Choose features and split by time

Maya predicts the future from the past, so we split by date, as Chapter 12 insisted. We train on the first 15 months, validate on the next 5, and keep the last 4 months locked away as the test set.

```
# Step 3: split by time
features = ["temperature", "rain", "weekend", "school_holiday"]
train = data[data["date"] < "2025-04-01"]
valid = data[(data["date"] >= "2025-04-01") & (data["date"] < "2025-09-01")]
test = data[data["date"] >= "2025-09-01"]
print(len(train), len(valid), len(test))

X_train, y_train = train[features], train["sales"]
X_valid, y_valid = valid[features], valid["sales"]
X_test, y_test = test[features], test["sales"]
```

The output is `456 153 121`: 456 training days, 153 validation days and 121 test days.

It is a tradition in machine learning code to call the table of features `X` and the labels `y`. So `X_train` holds the four feature columns for the training days, and `y_train` holds their sales. Every feature here is something Maya knows the evening before, from the weather forecast and the calendar. There is no leak from the future (Chapter 15).

Step 4: Score a baseline

Before any model, we need a baseline (Chapter 13). The simplest one is to always predict the average sales of the training days.

```
# Step 4: baseline
average = y_train.mean()
baseline_mae = mean_absolute_error(y_valid, [average] * len(y_valid))
print(f"Training average: {average:.0f} ice creams")
print(f"Baseline MAE on validation: {baseline_mae:.1f}")
```

```
Training average: 87 ice creams
Baseline MAE on validation: 72.5
```

Always guessing 87 ice creams misses by about 72 on a typical validation day. The validation months are spring and summer, when sales are far above the yearly average, so this baseline does badly. Any model worth using must beat 72.5 by a wide margin.

Step 5: Try three models

Now the fun part. We train three models from Part 2 and score each one on the validation data.

```
# Step 5: three models
models = {
    "Linear regression": LinearRegression(),
    "Random forest": RandomForestRegressor(n_estimators=300,
min_samples_leaf=5, random_state=0),
    "Gradient boosting": HistGradientBoostingRegressor(max_depth=3,
learning_rate=0.05,
                                                       max_iter=300,
random_state=0),
}
for name, model in models.items():
    model.fit(X_train, y_train)
    predictions = model.predict(X_valid)
    mae = mean_absolute_error(y_valid, predictions)
    r2 = r2_score(y_valid, predictions)
    print(f"{name:18s}  MAE {mae:5.1f}   R2 {r2:.2f}")
```

In scikit-learn, every model works the same way: `fit` trains it on features and labels, and `predict` makes predictions from features. That makes it easy to swap models in and out. The settings in brackets are hyperparameters: 300 trees in the forest, at least 5 days per leaf, and boosted trees of depth 3 with a learning rate of 0.05. Setting `random_state=0` makes the random parts repeatable, so you get the same numbers each time.

```
Linear regression   MAE  20.4   R2 0.83
Random forest       MAE  17.0   R2 0.88
Gradient boosting   MAE  15.3   R2 0.90
```

All three models crush the baseline of 72.5. Linear regression is the weakest. It can only add up straight-line effects, while the real pattern bends on hot days and has an interaction between rain and warmth. The tree-based models capture those shapes on their own, and gradient boosting does best. This is exactly the pattern Chapter 8 led you to expect for a table of data.

Step 6: Tune one setting

Let us check whether a different tree depth helps gradient boosting. We try several depths and compare validation scores. The test set is still untouched.

```
# Step 6: tune the depth of the boosted trees
for depth in [1, 2, 3, 5, 8]:
    booster = HistGradientBoostingRegressor(max_depth=depth,
learning_rate=0.05,
                                            max_iter=300, random_state=0)
    booster.fit(X_train, y_train)
    mae = mean_absolute_error(y_valid, booster.predict(X_valid))
    print(f"max_depth={depth}  validation MAE {mae:.1f}")
```

```
max_depth=1  validation MAE 19.7
max_depth=2  validation MAE 15.2
max_depth=3  validation MAE 15.3
max_depth=5  validation MAE 15.5
max_depth=8  validation MAE 15.7
```

Depth 1 is clearly worse: trees with a single question cannot capture the interaction between rain and warmth, so the model underfits. From depth 2 upwards, the scores differ by half an ice cream at most. Differences that small are mostly luck in which days landed in the validation set, so do not read much into them. When scores are tied, Occam's razor from Chapter 14 says to prefer the simpler model, so we choose depth 2.

Step 7: The final exam

All choices are made. As Chapter 12 suggested, we retrain the chosen model on the training and validation data together, then score it once on the test set.

```
# Step 7: final exam
best = HistGradientBoostingRegressor(max_depth=2, learning_rate=0.05,
                                     max_iter=300, random_state=0)
train_valid = pd.concat([train, valid])
best.fit(train_valid[features], train_valid["sales"])
test_predictions = best.predict(X_test)
test_mae = mean_absolute_error(y_test, test_predictions)
test_baseline = mean_absolute_error(y_test, [train_valid["sales"].mean()] *
len(y_test))
print(f"Test MAE: {test_mae:.1f}   (baseline: {test_baseline:.1f})")
print(f"Test R2: {r2_score(y_test, test_predictions):.2f}")
```

```
Test MAE: 13.1   (baseline: 47.0)
Test R2: 0.89
```

This is the honest result. On four months the model had never seen, its typical miss is about 13 ice creams, against 47 for always guessing the average. It explains about 89% of the day-to-day ups and downs. The test score is close to the validation score, which

suggests our choices were not over-tuned to the validation days.

This is the number Maya should use to decide whether to trust the model. A miss of 13 ice creams on a typical day is small enough to plan stock with confidence.

Step 8: What did the model learn?

Which features does the model rely on? Permutation importance (Chapter 8) shuffles one feature at a time on the test data and measures how much worse the predictions get.

```
# Step 8: what matters
result = permutation_importance(best, X_test, y_test, n_repeats=20,
random_state=0,
                                scoring="neg_mean_absolute_error")
for name, score in sorted(zip(features, result.importances_mean), key=lambda
p: -p[1]):
    print(f"{name:15s} {score:6.1f}")
```

```
temperature       31.2
weekend           17.2
rain               6.1
school_holiday     4.2
```

Shuffling temperature makes the typical error grow by about 31 ice creams, so temperature matters most. Weekends come next. Rain and school holidays matter less in these autumn test months: autumn has few school holidays, and rain costs less on cool days. All four make sense, and they match the hidden pattern we wrote in Step 1. With real data you would not know the hidden pattern, so this kind of sense check is your best protection against leaks and bugs.

Step 9: Predict tomorrow

Finally, the reason we did all this. Tomorrow's forecast says 26 degrees and no rain, on a Saturday outside school holidays.

```
# Step 9: predict tomorrow
tomorrow = pd.DataFrame([{"temperature": 26, "rain": 0, "weekend": 1,
"school_holiday": 0}])
print(f"Tomorrow: about {best.predict(tomorrow)[0]:.0f} ice creams")
```

```
Tomorrow: about 203 ice creams
```

Maya should prepare about 200 ice creams. Since the typical miss is about 13, a sensible plan is to prepare a little more, perhaps 215, and call her helper, because a 26-degree Saturday is certainly busy.

If something goes wrong

Error message	What it usually means	What to do
`ModuleNotFoundError: No module named 'sklearn'`	scikit-learn is not installed for the Python you are using	Install it as in Appendix A, then run again
`FileNotFoundError: icecream.csv`	Python cannot find the data file	Run `make_data.py` first, and run both files from the same folder
`KeyError: 'temprature'`	A column name is misspelt	Check the spelling against the names printed by `head()`
`IndentationError`	The spaces at the start of a line are wrong	Make lines inside a `for` loop start with the same four spaces
Numbers slightly different from the book	A different version of a library	Small differences are normal and not a problem

Ideas to take it further

You now have a complete, honest machine learning project. Here are ways to extend it, each linked to a chapter.

- **Add features (Chapter 3).** Add the month, or yesterday's sales. For yesterday's sales, make sure each day only uses the day before, never the same day, to avoid a leak.
- **Turn it into classification (Chapters 6 and 13).** Create a `busy` column for days above 150 sales, train a classifier, and look at its confusion matrix, precision and recall.
- **Draw pictures (Chapter 3).** Install the matplotlib library and plot predicted against real sales for the test days. Where are the biggest misses?
- **Try cross-validation (Chapter 12).** scikit-learn's `TimeSeriesSplit` gives rolling time-based folds.
- **Try a placebo (Chapter 15).** Shuffle the sales column in the training data and retrain. The validation score should fall to roughly the baseline. If it does not, something is wrong.
- **Use real data.** Public data sets on bike rentals, house prices or electricity demand let you repeat this whole recipe on real-world problems.

Key ideas

- A real project follows the recipe: look, split by time, baseline, compare models on validation, tune, test once, explain, predict.
- In scikit-learn, every model uses `fit` to train and `predict` to predict, so models are easy to compare.
- Always beat a baseline, and treat tiny validation differences as ties; prefer the simpler model.
- The test score, measured once, is the number to report and to trust.
- Check that what the model relies on makes sense; it is your best guard against leaks and bugs.

Try it yourself

1. Run both programs and check that you get the same numbers as the book.
2. Change the split so the test set is the last 6 months instead of 4. Does the test MAE change much?
3. Remove `school_holiday` from the feature list and rerun everything. How much worse is the validation MAE?
4. Predict sales for a rainy 18-degree Tuesday. Does the answer make sense compared with tomorrow's prediction?
5. Add a `busy` label (sales above 150) and train `RandomForestClassifier` from `sklearn.ensemble`. What accuracy do you get on the validation data, and what does the "always quiet" baseline get?

Appendices

Appendix A — Setting Up Python

You need Python and a few free libraries to run the project in Chapter 20. There are two easy routes. Choose one.

Route 1: Use a notebook in your web browser (no installation)

Several free online services let you write and run Python in a web page, with pandas, NumPy and scikit-learn already installed. Google Colab and Kaggle Notebooks are two popular ones. You need a free account. This is the fastest way to start, and it works on almost any computer, even a school laptop where you cannot install software.

In a notebook, code is typed into boxes called **cells**. Press the run button next to a cell, or Shift and Enter together, to run it. The output appears under the cell. To follow Chapter 20, put the code from `make_data.py` in the first cell and run it, then put each later step in its own cell and run them in order.

Route 2: Install Python on your own computer

1. **Install Python.** Download the installer for your system from the official Python website, python.org. Choose a recent version (3.10 or newer is a good choice). On Windows, tick the box that says “Add Python to PATH” during installation.
2. **Open a terminal.** On Windows, open “Terminal” or “PowerShell” from the Start menu. On a Mac, open the “Terminal” app. On Linux, open your terminal.
3. **Install the libraries.** Type this command and press Enter:

```
python -m pip install numpy pandas scikit-learn matplotlib
```

On Windows, if `python` is not recognised, try `py` instead:

```
py -m pip install numpy pandas scikit-learn matplotlib
```

4. **Check it worked.** Type this command. It should print a version number such as 1.6.1, not an error.

```
python -c "import sklearn; print(sklearn.__version__)"
```

5. **Make a project folder.** Create a folder such as `icecream_project`. Save `make_data.py` and `project.py` from Chapter 20 into it.
6. **Run the programs.** In the terminal, move into the folder and run both files:

```
cd icecream_project
python make_data.py
python project.py
```

Writing code comfortably

Any plain text editor works, but a code editor makes life easier: it colours the code, spots typing mistakes and can run programs with one click. Visual Studio Code and Thonny are free and popular with beginners. Thonny is especially simple and comes with Python built in.

A tidy habit: virtual environments

As you do more projects, different projects may need different library versions. A **virtual environment** is a private copy of Python for one project, so projects do not interfere with each other. You can create one inside your project folder with `python -m venv .venv` and switch it on before installing libraries. You do not need this for the book, but it is worth learning next.

Appendix B — Glossary

Every term printed in bold in the book, in plain words, with the chapter where it is explained in full. Where several names mean the same thing, they are listed together.

Term	Plain meaning	Chapter
Accuracy	The share of predictions that are right; misleading when one class is rare	13
Action	A choice made by a reinforcement learning agent	2
Activation function	The simple bend a neuron applies to its total, such as ReLU or the S-curve	17
Adam	A popular improved version of gradient descent that adjusts step sizes by itself	16
Agent	The learner in reinforcement learning, which acts and receives rewards	2
Anomaly detection	Finding examples that do not fit the normal pattern	2
Artificial intelligence (AI)	Any way of making machines do things that seem to need intelligence	1
Attention	The transformer’s way of letting each token draw on the tokens that matter most to it	18
AUC (area under the ROC curve)	A score from 0.5 (coin toss) to 1 (perfect) for how well a model ranks positives above negatives	13

Term	Plain meaning	Chapter
Backpropagation	The method for working out how much each weight in a network contributed to the error	16, 17
Bagging	Training many models on bootstrap samples and averaging them	8
Baseline	The score of a very simple method, used to judge whether a model is worth having	13
Bias (error)	Being consistently wrong in the same direction; goes with underfitting	14
Bias (neuron)	A neuron's starting lean, added before the bend	17
Bias (unfairness)	Systematic unfairness that enters through data, labels or use	19
Binary classification	Classification with exactly two categories, such as yes and no	2
Boosting	Building models one after another, each fixing the errors of the ones before	8
Bootstrap sampling	Drawing examples at random with replacement, so some repeat and some are left out	8
Categorical feature	A feature whose values are labels from a fixed list, such as flavours	3
Centroid	The centre point of a cluster in k-means	10
Classification	Supervised learning where the answer is a category	2, 6
Clustering	Finding groups of similar examples without labels	2, 10
Concept drift	The relationship between features and label changes over time	15

Term	Plain meaning	Chapter
Confusion matrix	A table of true positives, false positives, false negatives and true negatives	13
Convolutional neural network (CNN)	A network that slides small filters across images to find patterns	18
Cross-validation	Splitting data into folds that each take a turn as the validation set	12
Curse of dimensionality	With many features, every example is far from every other	9
Data	The collection of examples a model learns from	1
Data augmentation	Making varied copies of training examples, such as flipped images	14, 17
Data leakage	Training data containing information that will not be available in real use	15
Data preparation (preprocessing)	Checking, cleaning and converting data before training	4
Decision boundary	The line or surface that separates the regions predicted as each class	6
Decision tree	A model that predicts by asking a series of yes/no questions	7
Deep learning	Machine learning with neural networks that have many layers	1, 17
Dendrogram	The tree diagram produced by hierarchical clustering	10
Dense (fully connected) layer	A layer where every unit connects to every unit in the next layer	17
Diffusion	A way of generating images by removing noise step by step	18
Dimension	One feature, seen as one direction in which examples can differ	11
Dimensionality reduction	Replacing many features with a few that keep most of the information	2, 11

Term	Plain meaning	Chapter
Distribution shift	The data seen in real use no longer looks like the training data	15
Dropout	Switching off random neurons during training so none becomes indispensable	17
Early stopping	Stopping training when the validation error starts to rise	8, 14
Embedding	A list of numbers representing a word or item, where similar meanings are close	18
Ensemble	A model made by combining many models	8
Environment	The world a reinforcement learning agent acts in	2
Epoch	One full pass through all the training data	16
Error (residual)	The gap between a prediction and the true value	5
Euclidean distance	Straight-line distance between two examples	9
Example (sample, row, observation)	One item in the data, such as one day	1, 3
Explained variance	How much of the total spread a principal component captures	11
Exploration versus exploitation	Choosing between trying new actions and repeating ones that worked	2
Extrapolation	Predicting outside the range of the training data; risky	5
F1 score	One number that balances precision and recall	13
False negative	A real positive that the model missed	13
False positive	A false alarm: predicted positive, but really negative	13

Term	Plain meaning	Chapter
Feature (input, variable, attribute, predictor)	A fact used to make the prediction	2, 3
Feature engineering	Creating more useful features from raw data	3
Feature extraction	Building new features from old ones, as PCA does	11
Feature importance	How much a model relies on each feature	7, 8
Feature selection	Keeping some original features and dropping the rest	11
Feedback loop	A model's predictions shape the data it later learns from	15
Filter	A small pattern detector slid across an image in a CNN	18
Fine-tuning	Adjusting a pretrained model to a new task with new data	18
Fold	One of the equal parts data is cut into for cross-validation	12
Garbage in, garbage out	Poor data gives a poor model, however clever the method	1
Garden of forking paths	Trying many choices until one looks good by luck	15
Generalisation	Doing well on new examples, not only the training ones	1
Generative model	A model that creates new content, such as text or images	18
Gini impurity, entropy	Scores of how mixed a group of examples is	7
Gradient	The slope of the loss: which way is uphill for each parameter, and how steep	16
Gradient boosting	Boosting where each new tree is a small step downhill on the loss	8

Term	Plain meaning	Chapter
Gradient descent	Repeatedly stepping downhill on the loss to train a model	16
Greedy	Taking the best choice available at each step without planning ahead	7
Group split	Keeping all examples from the same person or thing on one side of a split	12
Hallucination	A confident but false statement from a language model	18
Hidden layer	A layer of neurons between the input and output layers	17
Hyperparameter	A setting chosen before training, such as tree depth or k	7
Imbalanced data	Data where one class is much rarer than another	15
Inference	Using a trained model to make predictions	1
Interaction	When the effect of one feature depends on another	7
Intercept	Where a line starts: its height when the feature is zero	5
Interpretable	A model whose workings a person can read and understand	5
Interpolation	Predicting inside the range of the training data; safer	5
k-means	Clustering by repeatedly assigning examples to the nearest centroid and moving the centroids	10
k-nearest neighbours (k-NN)	Predicting from the k most similar past examples	9
Label (target, output, response)	The answer the model is trained to predict	2, 3
Language model	A model trained to predict the next token of text	18

Term	Plain meaning	Chapter
Large language model (LLM)	A very large transformer language model, as used in chat assistants	18
Lazy learner	A model that does no training and works only at prediction time, like k-NN	9
Learning curve	How training and validation errors change as training data grows	14
Learning rate	The size of each step in gradient descent or boosting	8, 16
Least squares	Choosing the line with the smallest total of squared errors	5
Linear regression	Predicting a number with the best-fitting straight line	5
Local minimum	A valley in the loss that is not the lowest one	16
Log loss (cross-entropy)	A loss for probabilities that punishes confident mistakes hardest	6
Logistic regression	A classifier that bends a straight line into an S-curve to give probabilities	6
Loss (cost)	One number for how wrong the model is on the training data	16
Machine learning	Getting computers to learn tasks from examples instead of hand-written rules	1
Mean absolute error (MAE)	The typical size of a miss, in the units of the label	13
Median	The middle value when all values are sorted	4
Min-max scaling	Squeezing a feature into the range 0 to 1	4
Mini-batch	A small random handful of examples used for one gradient step	16

Term	Plain meaning	Chapter
Model	What is learned from data: a compact summary of the pattern that makes predictions	1
Model card	A short document describing what a model is for, how it was tested and its limits	19
Multi-class classification	Classification with more than two categories	2
Multimodal	Handling several kinds of data, such as text and images, together	18
Neural network	A model built from many simple neurons arranged in layers	1, 17
Neuron	A unit that weighs its inputs, adds a bias and applies a bend	17
Noise	Random variation in data that no model can predict	3
Nominal feature	A category with no natural order, such as colour	3
Numerical feature	A feature whose values are numbers where size means something	3
One-hot encoding	Turning one category column into several yes/no columns	4
Ordinal feature	A category with a natural order, such as small, medium, large	3
Outlier	A value far from the others; a mistake or a real rare event	4
Out-of-bag examples	The examples a tree in a random forest did not see, usable as a free check	8
Overfitting	Learning the noise along with the pattern, so new data is predicted badly	7, 14
Parameter	A number the model learns during training, such as a weight	5

Term	Plain meaning	Chapter
Permutation importance	How much worse a model gets when one feature is shuffled	8
Pipeline	A fixed chain of preparation steps followed by a model	4
Pixel	One tiny square of an image, stored as numbers	3
Pooling	Shrinking an image inside a CNN by summarising small areas	18
Precision	When the model says yes, how often it is right	13
Prediction	The model's guess for a new case	1
Pretrained	Already trained on a large data set, ready to be reused	18
Principal component	A direction of greatest spread in the data, found by PCA	11
Principal component analysis (PCA)	A method that finds the main directions of spread to reduce features	11
Proxy	A feature that secretly stands in for another, such as postcode for income	19
Pruning	Cutting back branches of a tree that do not help on new data	7
R-squared (R^2)	The share of variation a model explains, compared with always guessing the average	13
Random forest	Many trees trained on bootstrap samples with random features, averaged	8
Recall (sensitivity, true positive rate)	Of all the real positives, how many the model caught	13
Recurrent neural network (RNN)	A network that reads a sequence one step at a time with a memory	18

Term	Plain meaning	Chapter
Regression	Supervised learning where the answer is a number	2, 5
Regularisation (ridge, lasso, weight decay)	A penalty for complexity that keeps weights modest	5, 14
Reinforcement learning	Learning by acting, receiving rewards and improving	2
Reinforcement learning from human feedback (RLHF)	Training a model on people's choices between better and worse answers	18
ReLU	An activation that outputs zero for negative totals and the total otherwise	17
Reward	A number telling a reinforcement learning agent how good a result was	2
Root mean squared error (RMSE)	A typical miss in which big misses count extra	13
Sampling bias	The collected examples are not a fair picture of the cases to be predicted	3
Scaling	Putting numerical features on comparable ranges	4
Self-supervised learning	Learning from labels made out of the data itself, such as hidden words	2
Semi-supervised learning	Learning from a few labelled and many unlabelled examples	2
Sigmoid (logistic curve)	The S-shaped curve that turns any number into a value between 0 and 1	6
Silhouette score	A score of how well separated clusters are	10
Slope	How steep a line is: how much the prediction rises per step of the feature	5
Softmax	Turning several scores into probabilities that add up to 100%	6, 17

Term	Plain meaning	Chapter
Standard deviation	The typical distance of values from their average	4
Standardisation	Describing each value by how many standard deviations it is from the average	4
Stochastic gradient descent (SGD)	Gradient descent that estimates the slope from mini-batches	16
Stratified split	A split that keeps each class's share the same in every part	12
Structured data	Data in rows and columns	3
Supervised learning	Learning from examples that come with the right answers	2
Test set	Data locked away and used once, at the end, as the final exam	12
Threshold	The cut-off that turns a probability into a yes/no decision	6
Time-based split	Training on earlier data and testing on later data	12
Token	A word or piece of a word that a language model works with	18
Training (fitting)	Building a model from data	1
Training set	The data a model learns its parameters from	12
Transfer learning	Reusing a pretrained model for a new task	18
Transformer	A network design built on attention, used for text and more	18
True negative, true positive	Correct predictions of no and of yes	13
Underfitting	Being too simple to capture the real pattern	14
Unstructured data	Data such as images, text and sound that does not start as a table	3

Term	Plain meaning	Chapter
Unsupervised learning	Finding structure in data without answers	2
Validation set (development set)	Data used to compare models and choose settings	12
Variance	Being scattered: results that change a lot with the training sample; goes with overfitting	14
Virtual environment	A private copy of Python for one project	A
Weight (coefficient)	The number a model multiplies a feature by	5, 17
Well calibrated	A model whose stated probabilities come true at the stated rate	6

Appendix C — Answers to the Exercises

Many questions have more than one good answer. Where a question asks for your own example, a sample answer is given. If your answer differs but follows the same reasoning, it is right.

Chapter 1

1. Sample answers: a phone's face unlock (learned from photos of your face), a music app's suggestions (learned from what you and millions of others played and skipped), and a map's arrival time (learned from past trips and traffic speeds).
2. Machine learning, if the school has records of past students (marks, attendance, homework scores) and knows which of them later needed help. Those past students with known outcomes are the examples. Simple hand-written rules might catch the obvious cases, but they would miss combinations of warning signs.
3. A model that only memorises can answer only the cases it has already seen. The whole point is to handle new cases, which requires learning the general pattern. That is generalisation.
4. No. Hot weather causes both: people buy more ice cream and swim more. Two things moving together does not mean one causes the other.

5. Sample answer: deciding prison sentences, because it is a question of values and justice, mistakes are devastating, and past decisions may carry unfairness. Other good answers: tasks with very few examples, or completely new situations.

Chapter 2

1. Supervised learning, regression (the answer is a number of millimetres).
2. Unsupervised learning, clustering.
3. Reinforcement learning.
4. Supervised learning, binary classification.
5. Unsupervised learning, anomaly detection.
6. Supervised learning, multi-class classification.

Chapter 3

1. Sample answer. Features: age of the car (numerical), mileage (numerical), brand (categorical), fuel type (categorical), engine size (numerical). Label: the sale price.
2. Ordinal, because the stars have a natural order: five stars is better than four.
3. 10 × 10 = 100 pixels, each with 3 numbers for red, green and blue: 300 numbers.
4. Sampling bias. Monday-morning visitors may be mostly retired people or shift workers, who may not be like members who visit in the evenings or at weekends.
5. Sample answers: day of the week, whether it is a weekend, the month, whether it is a public holiday.

Chapter 4

1. Replace all four spellings with one standard value, such as “United Kingdom”, before training.
2. The gaps are probably not random: people at the extremes may feel the question is too personal. Fill the gaps (for example with the median) and add a “was missing” yes/no column, so the model can still use the fact that the value was missing.
3. Small becomes 1, 0, 0; medium becomes 0, 1, 0; large becomes 0, 0, 1 (one column each for small, medium and large). Plain numbering (small = 1, medium = 2, large = 3) is better here because the sizes have a real order, and it uses one column instead of three.
4. 85 is 45 above the smallest value, and the full range is 300 − 40 = 260. So the scaled value is 45 ÷ 260, about 0.17.
5. The test data would influence the numbers used in training. A little information about the test examples leaks in, and the test score becomes too optimistic.

Chapter 5

1. At 20 degrees: 14 × 20 − 161 = 119 ice creams. At 30 degrees: 14 × 30 − 161 = 259 ice creams.
2. The rule predicts 14 × 22 − 161 = 147. The real sales were 129, so the error is 129 − 147 = −18: the rule guessed 18 too many.
3. First, errors above and below the line would cancel out if simply added. Second, squaring makes big misses count much more than small ones.
4. No. The weights are in different units: one is per square metre and the other per bedroom. A bedroom adds about as much as 7.5 square metres. To compare importance fairly, scale the features first; and remember that linked features share credit.
5. No. Her model was trained on days from 16 to 31 degrees, and the new town's days are mostly below that. Using the model there would be extrapolation; it might even predict negative sales. She should collect data from the new shop.

Chapter 6

1. Sample answers: "Will it rain tomorrow?" (humidity, air pressure, cloud cover), "Is this card payment fraud?" (amount, place, time, usual spending), "Will this student pass the exam?" (attendance, homework marks, mock exam score).
2. At 22 degrees, about 34%, a bit more than one in three. At 24 degrees, about 89%.
3. Low. Missing a real fire is catastrophic, while an occasional false alarm is only annoying.
4. About 180 of them (90% of 200).
5. Log loss punishes confident mistakes hardest. Saying "100% sure" and being wrong means the model was certain of something false, which is the worst kind of error for decisions based on its probabilities.

Chapter 7

1. Rainy 27-degree Wednesday: raining, then above 24 degrees, so 120 ice creams. Dry 20-degree Tuesday: not raining, not a weekend, not above 24 degrees, so 95 ice creams.
2. "Can it fly?" gives two pure groups: 4 birds that fly and 4 fish that do not (setting aside flying fish!). Size would likely mix birds and fish in both groups.
3. A perfect score: zero error on the training data. That is not good news, because it means the tree memorised every example, including the noise, and will predict new data badly.

4. A parameter is learned from the data during training, such as the cut-off temperature in a split or the value in a leaf. A hyperparameter is chosen before training, such as the maximum depth or the minimum leaf size.
5. 260, never more. That is not sensible for a record-breaking heatwave, since sales might be higher or lower; the tree simply has no information beyond its training range.

Chapter 8

1. People who copy one expert all make the same mistakes, so nothing cancels out when you combine them. Random forests make their trees different on purpose, with random rows and random features, so that their errors are different and partly cancel.
2. Sample answer: A, C, C, D, A. Here B and E were never drawn, so they are out-of-bag.
3. The leftover error is 5, and the tree adds half of it: 195 + 2.5 = 197.5.
4. No. Adding trees to a random forest steadies the average but does not cure overfitting. Instead, try limiting the trees (larger minimum leaf size or smaller maximum depth), using fewer features per split, or collecting more data.
5. Stop at around 300 trees, the point where the held-out error was lowest. This is early stopping.

Chapter 9

1. With k = 1, the nearest day is F, so the prediction is 225. With k = 4, the nearest days are F, A, B and C, so the prediction is (225 + 210 + 238 + 160) ÷ 4, about 208.
2. With no points for weekends, days F and C both have a difference of 0, so they are the two nearest. The prediction with k = 2 is (225 + 160) ÷ 2, about 193.
3. Spam, with roughly 3 out of 5, or 60%, probability.
4. Because of the curse of dimensionality: with 500 features, every example is far from every other, so the “nearest” neighbours are not really similar. Many of the features are probably also noise that drowns out the useful ones.
5. Income. Its numbers are in the tens of thousands, while age differs by tens of years at most, so income differences swamp age differences in the distance.

Chapter 10

1. Sample answer: put the vans anywhere, let everyone walk to the nearest van, move each van to the middle of its crowd, let people switch to a closer van, and keep going until nobody switches.
2. Assign: 1 goes to the centroid at 1; 2, 9 and 10 all go to the centroid at 2, which is nearer to them. Update: the first centroid stays at 1, and the second moves to (2 + 9 + 10) ÷ 3 = 7. In the next round, 2 switches to the first centroid, and the centroids end at 1.5 and

9.5.

3. The centroids start in random places, and different starts can settle into different answers. The fix is to run k-means several times and keep the best result, using spread-out starting points (k-means++).
4. Three, because that is how many groups the shop can act on. It could merge similar clusters, or run k-means with k = 3.
5. No. k-means assumes round blobs and would chop the curved arms into pieces. A density-based method such as DBSCAN, or hierarchical clustering, would work better.

Chapter 11

1. Sample answer: height, weight and shoe size tend to move together and could be squeezed into one “body size” feature.
2. 62% + 21% + 8% = 91%.
3. Without scaling, the feature with the biggest numbers looks as if it has the most spread just because of its units. For example, income in dollars would swamp age in years, and the first component would be almost pure income.
4. Feature selection, because it keeps the original features, which the doctor can understand. PCA components are mixtures that are hard to explain.
5. Sample answer: a photo squeezes three dimensions into two, and the best angle keeps the most information. The first principal component is the “best angle” for data: the single direction that keeps the most of its spread.

Chapter 12

1. Sample answer: if students see the exam questions in advance, a perfect score proves only that they memorised them. A model tested on its own training data is the same: it may have memorised the answers.
2. Sample answer: the first 18 months for training, the next 3 months for validation and the last 3 months for testing, always keeping the time order.
3. Choosing the best of 200 test scores uses the test set to make a choice. Some of the winner’s score is luck, so it is too optimistic. The choice should be made on validation data, with the test set used once at the end.
4. With a group split: all 50 photos of each person go into the same pile. Otherwise the model can learn to recognise particular people rather than learning to recognise faces in general.
5. 5 models are trained. Each example is used for validation exactly once.

Chapter 13

1. On a typical house, the predicted price is about 20,000 away from the real price. You would also want to know the typical house price (20,000 is small for a 1,000,000 house but huge for a 50,000 one) and the MAE of a simple baseline.
2. 998 out of 1,000, which is 99.8%. It is useless because it never finds a single sick person, which is the whole purpose of the test.
3. Precision is 40 ÷ 50 = 80%. Recall is 40 ÷ 80 = 50%.
4. For a smoke alarm, recall: missing a fire is far worse than a false alarm. For automatically blocking bank cards, precision matters a lot, because wrongly blocking an honest customer's card is very disruptive; many banks add a quick check with the customer instead of blocking outright.
5. Model A explains about 30% of the variation compared with always guessing the average: weak, but better than nothing. Model B is worse than simply guessing the average every time, so something is wrong with it.

Chapter 14

1. Sample answer: a footballer who practises only one free kick from one spot until it is perfect, but cannot score from anywhere else, has overfitted.
2. The first model (2 against 30) is overfitting: a huge gap between training and validation error. The second (25 against 27) has almost no gap, but if 25 is high compared with what better models achieve, it is underfitting.
3. Because it keeps splitting until each leaf holds a single example, memorising every quirk and bit of noise in the training data.
4. The model is underfitting, so more data of the same kind will not help. Try better features, or a more flexible model.
5. Each deep tree has low bias but high variance: its darts land all around the bullseye, scattered. Averaging many scattered throws gives a point close to the centre, so the average has low bias and much lower variance.

Chapter 15

1. It depends on timing. If the prediction is made at the start of the month, the calls during that same month are not yet known, so it is a leak. If the feature uses only calls made before the moment of prediction, such as last month's calls, it is fine and probably very useful.
2. First: what share of the examples are earthquakes? If 98% are "no earthquake", then always saying "no" scores 98% too. Second: how was the data split, and could any feature leak information from after the event?

3. Sample answers: judge the model by precision and recall for defaults, not accuracy; weight the default class more heavily in training; oversample defaults in the training data only; choose the threshold from the costs of each mistake; collect more examples of defaults.
4. Shuffle the labels in the training data so they no longer match the features, then train and validate as usual. With no real pattern left, the score should drop to about the baseline. If the shuffled model still clearly beats the baseline, information is leaking somewhere.
5. Sample answer: a shop orders stock based on predicted demand. If it orders too little, it sells out, and the recorded sales are lower than the real demand. The next model learns from those lower sales and orders even less.

Chapter 16

1. Sample answer: tuning a radio by turning the dial a little, listening whether the station got clearer or fuzzier, and turning further in the direction that helped.
2. Each step is the learning rate times the slope of the loss. Near the bottom of the valley the ground is flatter, so the slope is smaller and the steps shrink automatically.
3. Lower the learning rate. Wild jumps usually mean the steps are too big and keep overshooting.
4. 10,000 ÷ 100 = 100 steps.
5. Even if each parameter could take only 10 values, a million parameters would give 10 multiplied by itself a million times as combinations: a number with a million digits. No computer could ever try them all.

Chapter 17

1. 0 × 3 + 1 × 2 + 1 × 1.5 − 4 = −0.5. The total is negative, so the neuron leans towards no.
2. 0, 0 and 2.5.
3. 2 × 3 = 6 weights into the hidden layer, 3 hidden biases, 3 weights to the output and 1 output bias: 13 parameters.
4. Without bends, each layer only computes weighted sums, and a weighted sum of weighted sums is still a weighted sum. However many layers you stack, the whole network behaves like one straight-line model.
5. Gradient boosting. The data is a small table with a few features, where boosted trees are usually as accurate or better, need less tuning and are easier to explain.

Chapter 18

1. It needs far fewer weights, because one small filter is reused everywhere; and it finds its pattern wherever it appears in the picture.
2. 500 photos are far too few to train an image network from scratch. A pretrained network already knows edges, textures and shapes, so the bakery only needs to fine-tune the last layers to tell its cakes apart.
3. With “big”, “it” means the trophy; with “small”, it means the suitcase. Attention lets the model link “it” to the right word by using the rest of the sentence, just as you do.
4. Because the label is simply the next token in real text, which is already there. No person needs to label anything; this is self-supervised learning.
5. Check that the book really exists and says what is claimed, for example in a library catalogue or a trusted search. Language models can produce convincing references that do not exist (hallucinations).

Chapter 19

1. The past approvals may reflect historical unfairness, and other features, such as postal code, can act as proxies for ethnicity. The model can learn the old pattern without ever seeing ethnicity directly.
2. Sample answers: the year someone finished school, years of work experience, and the email service or social media they use.
3. The overall figure is dominated by the largest groups. A smaller group can have a much higher error rate while the overall number still looks excellent.
4. Sample answers: Who is affected, and were students and parents consulted? What happens when the model is wrong: could a wrong “at risk” label harm a student? Does it work equally well for different groups of students?
5. To measure whether the model treats men and women differently. Without the gender information kept aside, the company could not check for unfairness.

Chapter 20

The numbers below come from running the book’s code with the versions named in Chapter 20. Small differences on your computer are normal.

1. You should see a validation MAE of about 15 for gradient boosting and a test MAE of about 13.1.
2. With training before February 2025, validation from February to June 2025 and testing from July to December 2025, the test MAE is about 13.9 against a baseline of about 64.9. The model’s error barely changes. The baseline is much worse, because the summer test months are far from the yearly average.

3. Without the school holiday feature, the validation MAE rises from about 15.2 to about 25.7. The validation period includes the long summer holiday, when this feature matters a lot.
4. The model predicts about 44 ice creams, far fewer than the 203 for a warm Saturday. That makes sense: it is cooler, wet and a weekday.
5. The random forest classifier is right on about 93.5% of the validation days. Always saying “quiet” is right only about 49% of the time, because about half of the spring and summer validation days were busy.

Appendix D — Where to Go Next

You now know the fundamentals: what machine learning is, the main families of models, how to test them honestly, and how modern deep learning grows out of the same ideas. The best next step is to **practise**, and the second best is to learn a little more maths and code alongside it. The titles below are widely recommended; check for their latest editions.

A suggested path

1. **Redo Chapter 20 with real data.** Pick a public data set on something you care about, such as bike rentals, house prices or football results, and follow the same nine steps.
2. **Get comfortable with pandas.** Most real work is loading, cleaning and reshaping tables. Short tutorials on pandas pay off quickly.
3. **Take one structured course** from the list below, and do its exercises, not just its videos.
4. **Enter a beginner competition** on Kaggle. Read other people’s shared notebooks afterwards; it is one of the fastest ways to learn practical tricks.
5. **Read a deeper book** to understand why the methods work, including the basic statistics behind them.
6. **Try deep learning** once you are confident with the fundamentals, starting with transfer learning on images.
7. **Build a small portfolio.** Two or three well-explained projects, honest about their test results and limitations, show more skill than many unfinished ones.

Books

Title	Authors	Why read it
An Introduction to Statistical Learning	Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani	The classic next step: clear explanations of the models in this book with a gentle dose of maths; free from the authors, with an edition using Python
Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow	Aurélien Géron	A practical, code-first tour from first models to deep learning
The Hundred-Page Machine Learning Book	Andriy Burkov	A compact overview of the whole field, good for revision
Deep Learning with Python	François Chollet	An approachable introduction to neural networks by the creator of the Keras library
Weapons of Math Destruction	Cathy O'Neil	A readable account of how models can harm people, a companion to Chapter 19

Courses

Course	Offered by	Good for
Machine Learning Specialization	DeepLearning.AI and Stanford, taught by Andrew Ng, on Coursera	A structured, friendly path through the fundamentals with exercises
Practical Deep Learning for Coders	fast.ai	Learning deep learning by building working models from the first lesson
Kaggle Learn	Kaggle	Short, free, hands-on lessons in Python, pandas and machine learning
Machine Learning Crash Course	Google	A quick, free overview with interactive exercises

Tools and documentation

- **scikit-learn user guide.** Clear explanations of every model in Part 2, with examples. When you want to know what a setting does, look here first.
- **pandas documentation**, especially its getting-started tutorials.
- **Jupyter notebooks**, the most common way data scientists write and share experiments.
- **PyTorch** and **TensorFlow** with **Keras**, the main libraries for deep learning, when you are ready for Part 4 in code.

Places to find practice data

- **Kaggle Datasets**, with thousands of data sets and beginner competitions such as predicting house prices.
- **The UCI Machine Learning Repository**, a long-standing collection of classic data sets used in teaching.
- **Government open data portals**, which publish data on transport, weather, health, education and more.
- **Your own life.** Track something for a few months, such as your sleep, your spending or your running times, and see what you can predict. Small personal data teaches every lesson in Part 3.

Good habits to keep

- **Start with a baseline,** and make every model earn its place against it.
- **Split honestly,** by time and by group when it matters, and use the test set once.
- **Be suspicious of great results,** and look for leaks before celebrating.
- **Look at your data and your errors,** not only at summary scores.
- **Ask who is affected,** and whether the model works for everyone it touches.
- **Write down what you did,** so you and others can repeat it.

Final words

Machine learning is not magic. It is careful pattern-finding from examples, plus honest checking of whether the patterns are real. Every system you read about in the news, from spam filters to chat assistants, rests on the ideas in this book: data, features, models, loss, gradient descent, and above all testing on data the model has never seen.

Maya started with a notebook and a simple question: how much ice cream should I make tomorrow? Every good project starts the same way. Find a question you care about, collect some examples, and begin.