ADDALY · THE AI CLASSROOM

Machine Learning, Foundations

The classical ground under modern AI, for someone who can read code.

9 modules · 92 lessons · 30 figures · 9 case studies · 71,137 words · about 13 hours

Dr Mustafa Mun
Author and editor

ABOUT THIS BOOK

Machine Learning, Foundations, published by Addaly, 2026. Author and editor: **Dr Mustafa Mun**.

The manuscript was drafted with the assistance of a large language model and was edited and published under his name. That is stated plainly here rather than left to be discovered, because a reader is entitled to know how a book they are trusting was made.

This book is the printed form of the course of the same name at addaly.com/learn/machine-learning-foundations. The website is the living version and is corrected first; where the two differ, the website is right.

Free to read, free to download, free to print, and free to teach from. There is no paid edition and there never will be. If you paid for this, somebody has sold you something that was given away.

Every diagram in it is drawn from data by code, not generated as a picture, so the numbers on the page are the numbers in the argument. Answers to every exercise, test and examination question are at the back, with a note on why each wrong option attracts people — those notes are the teaching, not the marking.

Contents

1. What learning is, and what it is not

Take a vague business question, state it as a dataset with a defined row, a defined target and a defined moment of prediction, name the loss and the baseline it must beat, and justify in one paragraph whether machine learning is the right tool at all

2. Linear models and the machinery of fitting

Fit and diagnose a linear or logistic model end to end: compute its loss by hand on a handful of rows, tune the learning rate from the shape of the loss curve, state what a coefficient does and does not mean in the units of the data, and explain what ridge and lasso penalties change about the fitted parameters

1. Fitting a line to five points, by hand
2. There is an exact answer, and we usually refuse it
3. Gradient descent, or walking downhill in fog
4. Batches, epochs, and the two knobs that decide everything
5. Diagnosing a model from the shape of its loss curve
6. Logistic regression, built from odds rather than assumed
7. What a coefficient says, and the four ways it lies
8. Regularisation: paying for large coefficients
9. Making a straight-line model bend
10. More than two classes, and what changes
11. Case study: The coefficient that changed sign in Muzaffarpur
12. Module test

3. The data is the job

Take a raw table with mixed types, missing values, high-cardinality categories, dates and free text, and produce a fitted preprocessing pipeline that leaks nothing from the held-out data, handles categories unseen at training time, and whose every transformation you can justify by what the downstream model requires

1. Features beat models
2. Scaling, and the transforms that change what a model can see
3. Turning categories into numbers without lying about them
4. When a column has thousands of levels
5. Missing values, and the information in the gap
6. Outliers: error, extreme, or the thing you are looking for
7. Dates, durations, and the trouble with midnight
8. Text as columns: counting words before you embed them
9. Leakage: a catalogue of the ways the answer gets in
10. Fit on train, transform everywhere: the contract that keeps you honest
11. Where the rows came from, and who is missing
12. When one class is 0.3% of the data
13. Case study: The 0.94 that had already been promised, in Tiruppur
14. Module test

4. Generalisation, and the discipline of held-out data

Design a validation scheme that matches the structure of the data — grouped, stratified or time-ordered as required — run a hyperparameter search inside it without contaminating the estimate, and state from a learning curve whether the next improvement should come from more data, better features or a different model

1. Train, validation, test, and the discipline of not looking
2. Overfitting, underfitting, and how to tell
3. Cross-validation: using every row twice, legally
4. Splits that respect the structure of the data
5. Bias and variance, with numbers attached
6. Would more data help? The curve that answers it
7. Searching hyperparameters without fooling yourself
8. A validation set you have used two hundred times
9. How much data do I need?
10. Double descent, and the picture that stopped being complete
11. Case study: The score that fell from 0.88 to 0.62 by being measured properly
12. Module test

5. Measuring a model honestly

Given a fitted classifier and a stated cost of each kind of error, compute the confusion matrix at a chosen threshold, defend that threshold by expected cost, say whether the model's probabilities are calibrated, decide with an interval whether one model genuinely beats another, and report per-group performance with a defensible fairness criterion and its known trade-offs

1. When accuracy lies
2. The four numbers, worked through
3. The threshold is a business decision, not a default
4. ROC and precision-recall: every threshold at once
5. Does 0.7 mean seventy per cent?
6. Metrics for a number, and what each one hides
7. Is that improvement real?
8. From a probability to a decision worth making
9. Fairness as arithmetic: the definitions and what each one demands
10. Why you cannot have all three
11. The average is hiding the failure
12. Case study: Six analysts, one threshold, and a second bank's portfolio
13. Module test

6. Trees, ensembles, and the algorithms that win on tables

Choose between a linear model, a random forest and a gradient boosting library for a given tabular problem and defend the choice, tune the four hyperparameters that actually matter in each, and explain why impurity-based feature importance is biased and what permutation importance and SHAP replace it with

1. The classic algorithms, and when each is right
2. How a tree chooses where to cut
3. Bagging: many unstable models, averaged
4. Boosting: each model fixes the last one's mistakes
5. XGBoost, LightGBM, CatBoost: what actually differs
6. k-nearest neighbours, and why distance stops meaning anything
7. Naive Bayes: wrong assumptions, useful results
8. Support vector machines and the kernel trick
9. Feature importance, and the ways it misleads
10. Explaining one prediction, and the shape of one feature
11. Case study: The bar chart that nearly rebuilt three warehouses
12. Module test

7. Learning without labels, and learning a representation

Cluster a dataset and defend the number of clusters with more than an elbow plot, reduce dimensionality with PCA and state how much information the reduction discarded, explain what an embedding is as a geometric object, and build a working nearest-neighbour search over pretrained embeddings without a GPU

1. k-means, and what it assumes about your data
2. Choosing k, and why the elbow rarely helps
3. DBSCAN and hierarchical clustering: when shape and structure matter
4. PCA: fewer columns, and what you gave up
5. t-SNE and UMAP, and how they mislead
6. What an embedding actually is
7. Building a working semantic search with no GPU
8. Recommendation: learning a vector per user and per item
9. Anomaly detection, and the base rate that ruins it
10. Self-supervised learning: making labels out of the data itself
11. Case study: The segmentation that was not there
12. Module test

8. Neural networks, built from what you already know

Explain why a multi-layer network with no non-linearity collapses to a single linear model, trace one backpropagation step through a two-layer network by hand, diagnose a training failure from the loss curve and the gradients, and fine-tune a pretrained model on a few hundred labelled examples using free compute

1. Neural networks as the natural next step
2. XOR, and why the bend is the whole idea
3. The bends: ReLU, and why sigmoid stopped being used inside networks
4. Backpropagation, traced through one tiny network
5. Starting well, and staying well-behaved
6. Momentum, Adam, and what the optimiser is doing for you
7. Keeping a large network from memorising
8. Convolution: the same detector, everywhere in the image
9. Sequences: from recurrence to attention
10. Standing on somebody else's compute
11. Case study: Three hundred and forty leaves
12. Module test

9. Into the world, and keeping it honest

Take a fitted model to a running service with a versioned artefact, a reproducible training run, monitoring that distinguishes covariate shift from concept drift, a stated retraining trigger, and a written record of what the model was trained on and who it should not be used for

1. The model is about 5% of what you have to build
2. Batch, online, and the latency budget
3. When training and serving quietly disagree
4. Monitoring: what to watch, and in what order
5. When to retrain, and what breaks when you do
6. Being able to build the same model twice
7. AutoML: what it automates, and what it cannot
8. Making a model small enough to be affordable
9. When the model changes the data it will be trained on
10. What you owe the people in the data
11. Case study: A week of retraining that fixed nothing
12. Module test

Machine Learning, Foundations — final examination

The paper that opens the next course. Answers to everything follow it.

MODULE 1

What learning is, and what it is not

Before any algorithm, the vocabulary and the honest framing. What a dataset actually is, what a loss function commits you to, what number your model has to beat before anyone should care, and the large class of problems where the correct machine learning solution is not to use machine learning.

> BY THE END OF THIS MODULE YOU CAN
>
> Take a vague business question, state it as a dataset with a defined row, a defined target and a defined moment of prediction, name the loss and the baseline it must beat, and justify in one paragraph whether machine learning is the right tool at all

---

1 · 8 min

What a machine actually learns

> THE ONE THING TO KEEP
>
> Learning means adjusting the parameters of a shape you chose until a chosen error number is as small as it can be.

Fitting, not understanding

Here are four flats in Lagos, with monthly rent in naira.

Floor area (m²)	Rent (₦/month)
40	180,000
55	240,000
70	310,000
90	380,000

You want a program that guesses the rent of a flat it has never seen. You could write rules by hand. Instead you write down a shape with blanks in it:

```
rent = w * area + b
```

`w` and `b` are the blanks. They are called parameters. Every choice of `w` and `b` is a different program. Learning is the search for the pair that works best.

"Best" has to be a number

Guess `w = 4000`, `b = 0`. Run the four flats through it:

Area	Predicted	Actual	Off by
40	160,000	180,000	20,000
55	220,000	240,000	20,000
70	280,000	310,000	30,000
90	360,000	380,000	20,000

Average error: ₦22,500. That single number is called the **loss**. It is the only thing the machine can see about how it is doing.

Now try `w = 4000`, `b = 20000`:

Area	Predicted	Actual	Off by
40	180,000	180,000	0
55	240,000	240,000	0
70	300,000	310,000	10,000
90	380,000	380,000	0

Average error: ₦2,500.

The loss fell from 22,500 to 2,500. That drop is the entire content of the word "learning". No insight was gained. Two numbers moved and a third number got smaller.

The three ingredients

Every supervised model, from a two-parameter line to a language model with a trillion parameters, is made of the same three things.

1. **A shape with parameters.** A line. A tree of if-statements. A stack of matrix multiplications. You choose the shape.
2. **A loss.** One number saying how wrong the current parameters are, averaged over your data. You choose the loss, and the choice matters more than people expect.
3. **A procedure for changing the parameters to lower the loss.** For a straight line you can solve it with algebra. For anything larger you walk downhill, which is Lesson 8.

That is the whole mechanism. Everything else in this course is about doing it honestly.

What the model does not have

The fitted model has no concept of a building. Hand it `area = 500` and it will report ₦2,020,000 without hesitation, because it has never been told that flats of that size are rare, or that the market above 200 m² behaves differently. It has never been told anything. It has four rows and two knobs.

This also means the shape you picked is a hard ceiling. Suppose rents flatten out above 120 m², because very large flats sell rather than rent. A straight line cannot bend. No quantity of extra data will make `w * area + b` produce a curve — more data will only pin down the best possible straight line more precisely, and the best possible straight line is still wrong. If you want a curve, you have to pick a shape that can curve.

This is why "which model should I use" is a real question and not a detail. You are not asking which program is smarter. You are choosing which family of functions the search is allowed to look inside.

Parameters and hyperparameters

`w` and `b` are fitted from the data. But other numbers get chosen before fitting starts: how many terms the polynomial has, how deep a tree may go, how large each downhill step is. These are **hyperparameters**. They are picked by you, by trial, and where you are allowed to run those trials is the subject of Lesson 4.

Keep the distinction clear from the start. Parameters come from the data. Hyperparameters come from you.

BEFORE YOU MOVE ON

You fit `rent = w * area + b` to 50,000 flats. The true relationship curves: rent rises steeply up to 120 m² and then almost flattens. Training finishes. What has it accomplished?

A. It has found the best straight line through curved data, and the curve is permanently out of reach for this model.

B. With 50,000 rows it has enough data to discover the curve on its own.

C. Training will drive the loss to zero if it is allowed to run long enough.

D. The curve is a sign the data is noisy and should be cleaned before fitting.

2 · 7 min

With answers, and without

THE ONE THING TO KEEP

A supervised model learns your labels, not the reality behind them, so who made the labels is part of the model.

Two different things to have

A bank in Nairobi has ten million past transactions. For 4,000 of them, an investigator looked and wrote down `fraud` or `not fraud`. Those 4,000 rows have **labels**: recorded answers.

With labels, you can do **supervised learning**. You show the model an input and the answer, it adjusts, and eventually it maps new inputs to answers. Almost everything with commercial value is supervised: spam filters, credit scoring, delivery-time estimates, medical triage, machine translation.

Without labels you can do **unsupervised learning**. You give the model ten million transactions and no answers, and ask what structure is in there. Clustering groups similar rows. Dimensionality reduction compresses 200 columns into 2 you can plot. Anomaly detection finds rows unlike the rest.

The distinction is not about the algorithm. It is about what you have.

Unsupervised results are questions, not answers

Run k-means on those transactions asking for six clusters, and you will get six clusters. You will get six clusters if the data is pure random noise, too. The algorithm has no way to tell you that the structure it found is not there.

And the clusters arrive unnamed. Cluster 4 is "rows 88,201, 88,540, 91,003 and 402,884 others". Someone has to look at them and say "these are salary deposits on the 25th" or "these are top-ups from a single agent network". That naming is human work, and it is where the mistakes live.

So: unsupervised methods are good at generating hypotheses and terrible at confirming them. If a cluster looks like a fraud ring, that is a lead for an investigator, not a finding.

The label defines the task

Here is the part that gets skipped, and it is the most important idea in this lesson.

A supervised model does not learn the truth. It learns your label. If your label is imperfect, the model learns the imperfection with the same enthusiasm it learns everything else.

The bank's label is not "fraud". It is "transactions an investigator looked at and confirmed as fraud". Those are different things:

- Fraud nobody noticed is labelled `not fraud`.
- Fraud noticed only because it matched the old rule-based system is over-represented.
- Fraud in a customer segment nobody audits is invisible.

Train on that and you get a model that reproduces the investigators' existing reach. It may be genuinely useful — it can rank a queue and save time — but it is a model of the detection process, not of fraud.

The same trap in other clothes:

- A hospital labels "patients with condition X" using billing codes. The model learns which patients get coded, which depends on the clinic, the insurer and the country.
- A recruiter labels "good hire" as "still employed after two years". The model learns retention, which is partly about managers.
- A logistics firm labels "late delivery" using customer complaints. The model learns which customers complain.

Before you look at a single metric, write one sentence: *this label was created by _____, and it misses _____*. If you cannot fill in the blanks, you do not yet know what you are building.

The middle ground, and where modern AI sits

Two shapes sit between the extremes and are worth knowing by name.

Semi-supervised: a small labelled set plus a large unlabelled one. Common in medicine, where labelling means a specialist's time.

Self-supervised: labels manufactured from the data's own structure. Hide a word in a sentence and predict it. Hide a patch of an image and predict it. No human labelled anything, yet the task is supervised in mechanism, and there is an unlimited supply of it.

Self-supervision is how large language models are trained, and it is the main reason they exist at all. The bottleneck on supervised learning was always labels. Self-supervision removed the bottleneck by inventing a task where the answer is already sitting in the data.

BEFORE YOU MOVE ON

A payments team trains a fraud classifier on a label built from cases their investigators confirmed as fraud. On held-out data it performs well. What has the model most likely learned?

A. Which transactions resemble the ones this team's investigators already catch.

B. What fraud looks like in general, since every confirmed case genuinely was fraud.

C. Nothing worth deploying, because the labels are biased.

D. It would avoid this problem if they used unsupervised clustering instead.

3 · 9 min

What a row is, and why that decision is the whole project

THE ONE THING TO KEEP

Decide what one row is and when the target becomes knowable before you write any code, because every split, metric and deployment decision inherits that choice silently.

The two objects

Almost all of classical machine learning operates on two objects. A matrix `X` with one row per thing and one column per measurement, and a vector `y` with one entry per row: the answer you want the model to produce.

```
import numpy as np
X = np.array([[35, 42000, 3],
              [51, 88000, 0],
              [24, 19000, 7]])   # age, annual income, late
payments
y = np.array([0, 0, 1])          # defaulted within 12 months?
X.shape, y.shape   # ((3, 3), (3,))
```

That is it. Three rows, three columns, three labels. Everything later in this course — trees, boosting, neural networks — is a different way of turning the left object into a guess at the right one.

The interesting part is not the code. It is the sentence you had to commit to before you could type it.

"One row is one ____"

Finish that sentence out loud before you write anything. It is called the **unit of analysis**, or the *grain*, and getting it wrong is the most expensive mistake

in this course because nothing downstream complains.

A bank asks: *can we predict default?* There are at least four honest datasets hiding in that question.

- One row is **one customer**. Then a customer with eleven loans appears once, and you must decide what "defaulted" means across eleven loans.
- One row is **one loan**. Then the same customer appears eleven times, and eleven rows share almost every column. Your model will learn that customer, and when you split the data randomly, that customer appears on both sides of the split. The score comes out beautifully and means nothing.
- One row is **one customer-month**. Now you can predict *default in the next 30 days*, which is a different and often more useful product.
- One row is **one application**, including the ones you rejected. This is the only grain that can answer *should we have approved it*, and it is the one you almost never have labels for, because rejected applicants never got the chance to repay.

Those four datasets have different sizes, different targets, different splits and different deployments. Choosing between them is a product decision wearing a data costume.

The target needs a clock

`y` is not a column you find. It is a definition you write, and it has to contain a moment.

"Churned" is not a label. "Had no transaction in the 60 days following the observation date" is a label. "Fraud" is not a label; "was charged back within 90 days" is. The moment matters because the label has to be *knowable later than the features and not before* — and because a shorter window gives you more labelled rows but a noisier target, while a longer one gives cleaner labels and less recent data. There is no right answer, only a stated one.

Write it as one line at the top of the notebook:

> ***Row:*** *one loan application.* ***Features:*** *everything known at the moment of submission.* ***Target:*** *1 if 90 days past due within 12 months of disbur-*

> *sal, else 0.* ***Excluded:*** *anything recorded after disbursal.*

That last clause is the one that saves you. We will return to it in the leakage lesson, but the guard is written here, at the point where the dataset is defined.

Columns have types, and the type changes the maths

Three kinds of column behave differently and need different treatment later:

- **Numerical** — age, income, distance. Order and distance both mean something. `income - 1000` is meaningful.
- **Categorical** — city, product code, device. Values are names. `Mumbai + 1` is nonsense, and the moment you store cities as 0, 1, 2 you have told a linear model that Chennai sits between Mumbai and Delhi.
- **Ordinal** — education level, star rating, T-shirt size. Order means something, distance does not. The gap between 1 star and 2 is not the gap between 4 and 5.

An ID column belongs to none of these and should almost never be a feature. A model that gets `customer_id` will happily memorise it.

Shape errors are the beginner's rite of passage

Most first-week errors are shape errors, and the message is usually accurate but blunt. Two habits fix nearly all of them: print `X.shape` and `y.shape` after every transformation, and keep the convention that rows are samples and columns are features — scikit-learn assumes it everywhere, and a transposed matrix trains without complaint and predicts nonsense.

A single sample has to be reshaped to a matrix of one row before you can predict on it:

```
model.predict(np.array([[35, 42000, 3]]))   # note the double
brackets
```

The honest limitation

This rectangle is a strong assumption, and it is where a lot of real data does not fit. Text, images, audio, graphs and event streams are not naturally rectangular. What we do — and this is worth seeing clearly rather than absorbing by osmosis — is *force* them into the rectangle: counts of words, pixels flattened into columns, an event log aggregated into "number of logins in the last 7 days". Every one of those is a lossy choice made by a person, and most of the accuracy in a classical project comes from making those choices well rather than from the algorithm that eats the result.

Deep learning's claim, which we will reach in module 8, is that it can learn the flattening itself. It is a real claim. It is also why it needs so much more data.

BEFORE YOU MOVE ON

A team building a hospital readmission model has one row per hospital *visit*, and patients often visit many times. They split the rows randomly into train and test, and get 0.94 AUC. What is the mechanism that most likely inflated that score?

A. Visits are not independent — the same patient's rows land in both train and test, so the model can recognise the patient rather than the medical pattern.

B. AUC is the wrong metric for medical data, and accuracy would have shown the true, lower performance.

C. The dataset has too many columns relative to the number of visits, so the model overfits the feature space.

D. Readmission is a rare event, so the majority class dominates and the score is meaningless.

4 · 9 min

The loss is where you state what a mistake costs

THE ONE THING TO KEEP

Squared error fits the mean and fears outliers, absolute error fits the median and ignores them, log loss punishes confident mistakes without limit — and none of these is the metric you report.

Fitting means minimising something you chose

A model has parameters. Training searches for parameter values that make one number small. That number is the **loss**, and the choice of loss is not a technicality — it is the sentence "this is what being wrong costs me", written in maths so an optimiser can act on it.

Two people with the same data and the same algorithm can get different models because they encoded different costs. Neither is wrong. Only one matches the situation.

Squared error punishes the outlier

For a numeric target, the default is mean squared error:

```
mse = ((y_true - y_pred) ** 2).mean()
```

Squaring does something specific. An error of 10 costs 100; an error of 100 costs 10,000. One badly wrong row can outweigh a hundred nearly right ones. Fit a line to house prices with MSE and a single mansion drags the whole line upward, worsening every ordinary prediction, because moving 5 units towards the mansion buys more loss reduction than 5 units of error added across many small houses.

Mean absolute error does not do that:

```
mae = np.abs(y_true - y_pred).mean()
```

An error of 100 costs 100. Outliers get no special weight.

There is a clean way to remember which you are asking for. **Minimising squared error fits the mean; minimising absolute error fits the median.** If your data is `[1, 2, 3, 4, 100]` and you must predict one constant, MSE gives you 22, MAE gives you 3. Ask yourself which of those two answers you actually want a delivery-time estimate to be.

Huber loss is the compromise: squared near zero, linear far away, with a threshold you set. It is a good default for regression on messy real data and it is one line in scikit-learn.

For classification, the loss is about probabilities

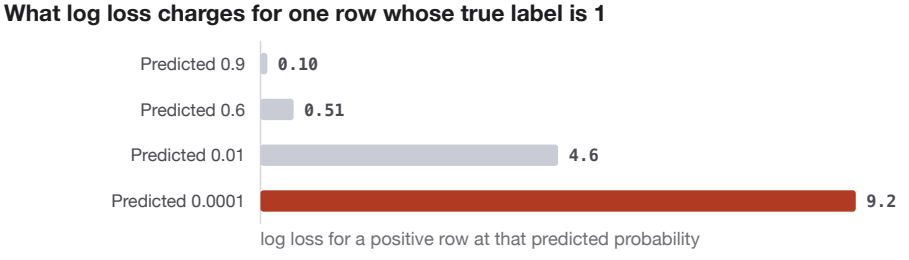
You might think classification should minimise the number of wrong labels. It does not, and the reason is mechanical: the count of errors is a step function. Nudge a parameter slightly and the count either does not move at all or jumps by one. There is no slope to follow, so gradient descent has nothing to work with.

So classification minimises **log loss**, also called cross-entropy, over predicted probabilities:

```
logloss = -(y * np.log(p) + (1 - y) * np.log(1 - p)).mean()
```

Read what this does at two rows. The true label is 1. You predict 0.9, and the loss is `-log(0.9) = 0.105`. You predict 0.6, and it is `0.511`. You predict 0.01, and it is `4.6`. Now the ugly one: you predict 0.0001 and the truth is 1, and the loss is `9.2`. Log loss is asymmetric in a way that matters — being confidently wrong is punished savagely, and as your prediction approaches 0 for a positive case the loss goes to infinity. That is the property that makes models

trained on log loss produce probabilities you can partly trust, and it is why a single confident mistake can dominate a validation score.



The scale is not linear. Moving a prediction from 0.9 down to 0.6 costs about half a unit; moving from 0.01 down to 0.0001 costs another 4.6. One confidently wrong row can outweigh dozens of nearly right ones, which is why a model trained on log loss learns to hedge.

The loss is not the metric

This trips up almost everyone once. The loss is what training optimises; the metric is what you report and decide with. They are often different, deliberately.

You cannot train on "precision at a threshold of 0.7" — it is not differentiable and the threshold is not part of the model. So you train on log loss to get honest probabilities, and then you choose the threshold afterwards using precision and recall, which is exactly what module 5 does. The gap between loss and metric is normal. What is not normal is failing to notice that closing the gap in loss did nothing for the metric, which happens more often than the textbooks admit.

Weighting is how you say some rows matter more

Every major library accepts `sample_weight`. Multiply each row's loss contribution by a number and the optimiser will trade accuracy on cheap rows for accuracy on expensive ones.

This is the cleanest tool available for imbalance and for unequal costs. If missing a fraudulent transaction costs 200 times what a false alarm costs, that is not a threshold trick — you can say it directly, in the loss, and the model will spend its capacity accordingly. `class_weight='balanced'` in scikit-learn is the same mechanism with the weights set to the inverse class frequency.

The limitation nobody puts on the slide

A loss function is a scalar. Real decisions have costs that do not collapse into one number, and when you force them to, you have made an ethical claim with a division sign. A model that decides who gets a loan is trading false rejections against false approvals, and those two harms land on different people. Writing `sample_weight=200` does not resolve that; it records your answer to it in a place where nobody will read it again.

So state the loss in words next to the code. "A missed fraud costs us roughly 200 times a false alarm, based on average chargeback of ₹18,000 against about ₹90 of review time." Someone can then argue with the 200. They cannot argue with a number they never saw.

BEFORE YOU MOVE ON

A delivery firm predicts arrival times. Most trips take 20 to 40 minutes, but a handful take three hours because of accidents. Trained with MSE, the model consistently over-predicts ordinary trips. What is happening?

A. The three-hour trips are rare enough to be treated as noise, so the model is ignoring them and drifting toward the minimum trip time.

B. The model has too few features to distinguish accident days, so it underfits and predicts the overall average for everything.

C. Squared error makes the three-hour errors enormously costly, so the fitted line moves upward to reduce them, adding a little error to every normal trip.

D. Arrival time is not normally distributed, and MSE requires a normal target, so the optimiser fails to converge.

5 · 8 min

The number your model has to beat before anyone should care

THE ONE THING TO KEEP

Compute the constant, persistence and existing-rule baselines before training, and report the gap between baseline and model rather than the model's score alone.

94% accuracy is often a failure

A fraud model reports 94% accuracy and the room applauds. Then someone asks what happens if you predict "not fraud" for every single transaction. The answer is 96%, because 4% of transactions are fraudulent. The model is worse than a single line of code that ignores the data.

This is not a rare embarrassment. It is the default outcome when nobody computes a baseline, and it is the cheapest quality gate in machine learning: one function, five minutes, run before you train anything.

The four baselines worth having

1. The constant. For classification, always predict the most common class. For regression, always predict the mean — or, if the target is skewed, the median, which minimises absolute error.

```
from sklearn.dummy import DummyClassifier, DummyRegressor
DummyClassifier(strategy="most_frequent").fit(X_tr,
y_tr).score(X_te, y_te)
DummyRegressor(strategy="median").fit(X_tr, y_tr)
```

2. Yesterday. For anything with a time axis, predict that tomorrow equals today. This is the *persistence* baseline and it is brutal. In demand forecasting,

sales prediction and traffic estimation it regularly beats models that took three weeks to build, because most series are dominated by their own recent value.

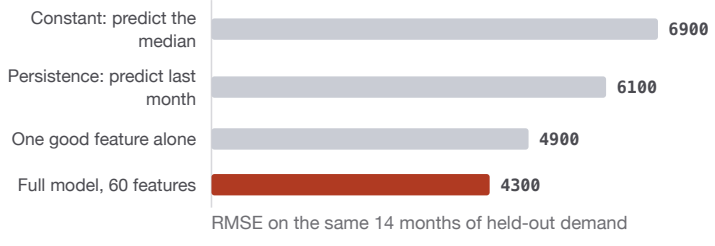
3. The rule the business already uses. Someone in operations has a rule of thumb: flag any transaction over ₹50,000 from a new account. Implement it, score it on your test set, and you now know what your model is replacing. This is the only baseline your stakeholders truly care about, and it is the one most often skipped.

4. One good feature. Fit a model on the single most obvious predictor and nothing else. If "days since last purchase" alone gets you to 0.78 AUC and your 60-feature gradient boosting model gets 0.80, that is worth knowing before you build a feature pipeline that has to be maintained for three years.

Report the gap, not the score

A score alone is uninterpretable. "RMSE 4,300" means nothing until you know the median baseline scores 6,100 — then your model explains about 30% of the gap you had a chance to close, and that is a sentence a non-technical person can act on.

RMSE 4,300 of what



The model beats the persistence baseline by 30 per cent, measured on 14 months of held-out data. Reported alone, 4,300 is a number. Reported beside the baselines it is a claim someone can act on — and the single-feature model shows that most of the gap was closed by one column.

Most honest one-line summary in ML reporting:

> *Baseline (predict last month) RMSE 6,100. Model RMSE 4,300. Improvement 30%, measured on 14 months of held-out data.*

Compare that with "our model achieves an RMSE of 4,300". Same run, wildly different amount of information.

Baselines catch bugs, not just weak models

The second use of a baseline is diagnostic, and it is the reason experienced people compute one before every project rather than for the report at the end.

- **The model does much worse than the constant baseline.** Something is broken. Labels shuffled, features and targets misaligned by one row, a transformation fitted on the wrong data.
- **The model does *far* better than any baseline, immediately, with no tuning.** Almost always leakage. A 0.99 AUC on a problem humans find hard is a bug report, not a result. Look for a column recorded after the outcome — the classic is `account_closed_reason` in a churn dataset.
- **The model exactly matches the baseline.** It has learned to predict the constant. Check whether the features reach the model at all; a pipeline that drops every column silently is a real and common failure.

Ceilings matter as much as floors

The baseline is the floor. There is also a ceiling, and it is rarely 100%.

Give two experienced radiologists the same 200 scans and they will disagree on some fraction of them. That disagreement rate is roughly the point past which a model's extra accuracy stops being measurable, because your labels themselves contain that much noise. The same applies far outside medicine: if two support agents label the same ticket differently one time in six, a model scoring 85% on that task may already be at the ceiling of what the labels can express.

So when you can, measure human agreement on a sample of your own data. Two people, a hundred rows, an afternoon. Between the baseline and the human agreement rate is the entire range in which your work can make a difference, and it is often much narrower than anyone assumed.

What this costs you

About twenty minutes. There is a version of this course that skips this lesson and gets to gradient boosting sooner, and the learner who takes that version

will one day present a number to a room that has no idea whether it is good. The baseline is what turns a score into a claim.

BEFORE YOU MOVE ON

A team builds a model to predict next week's electricity demand and reports RMSE of 210 MW, calling it a strong result. What single check would most change how that number should be read?

A. Compute a confidence interval on the RMSE using the bootstrap.

B. Retrain with cross-validation instead of a single split.

C. Check whether demand is normally distributed so that RMSE is an appropriate metric.

D. Score a persistence baseline that predicts next week equals this week, on exactly the same held-out period.

---

6 · 9 min

Why a model that predicts well cannot tell you what to change

THE ONE THING TO KEEP

A predictive model reports associations that hold when you leave the world alone; changing something requires an experiment or an explicit causal assumption, and no accuracy score can substitute for either.

Two questions that look like one

Who will churn next month? and *what should we do to stop them churning?* sound like the same question with an extra step. They are not, and a model built for the first cannot answer the second — not with better features, not with SHAP, not with more data.

The difference is that prediction only requires a correlation that holds. Intervention requires knowing what happens when you reach in and change something, which is a claim about a world that has not happened yet.

The example everyone should carry

A telecom company finds that customers who call support are far more likely to churn. It is a strong signal, stable across two years and every region. As a predictor it works.

Now read it as an instruction: *reduce churn by making it harder to call support*. Obviously absurd, and the absurdity comes from the same place every time — the call did not cause the churn, the underlying problem caused both. Remove the phone line and you remove the signal while leaving the cause untouched. Churn goes up, because now the frustrated customers cannot even complain.

The model was never wrong. It answered the question it was asked. Somebody then asked it a different question and read the old answer.

Why the coefficient does not mean what you want

A linear model gives you a number per feature, and the temptation to read it causally is nearly irresistible. Here is the mechanism that breaks it.

Suppose hospital patients on a certain drug die more often. Fit a model, and the drug's coefficient is positive. But the drug is given to the sickest patients. The coefficient is measuring "was sick enough to be prescribed this", and it will do so faithfully.

You might say: add severity as a feature and the drug's coefficient will clean up. Sometimes. This is called controlling for a confounder and it is the standard move — but it only works if you measured the confounder, measured it well, and there are no others. In practice severity is a doctor's judgement compressed into a code, and the part of it that is not in your data stays attached to the drug coefficient.

Worse, controlling for the wrong thing actively creates bias. If you control for a variable that sits *downstream* of the treatment — a collider — you can manufacture an association out of nothing. This is not an edge case; it is the single most common statistical error in observational analysis, and it is invisible in any accuracy score.

The one thing that settles it

A randomised experiment. Assign the change at random, and the randomisation guarantees that the treated and untreated groups differ only by chance in *everything*, measured or not — including the confounders you never thought of. That is what randomisation buys, and it is why it remains the gold standard after a century of attempts to replace it.

When you cannot randomise — you cannot randomly assign smoking, or randomly deny a loan to test the model — there is a serious toolkit: difference-in-differences, instrumental variables, regression discontinuity, propensity matching, and the do-calculus that Judea Pearl's group formalised. Every one of these buys a causal claim by making an assumption you cannot test with your data. They are not tricks; they are honest bargains, and you should be able to state the assumption out loud before using one.

This course does not teach them. It teaches you to notice when you need them, which is the part people miss.

What to do on Monday

When someone hands you a predictive model and asks what to change, three moves are available and all of them are legitimate.

1. **Use the model for targeting, not for policy.** "Predict who will churn, then have a human call them" is fine. The model chose who to call; it did not claim the call works. Whether the call works is a separate experiment.
2. **Run the experiment on the intervention.** Take the top 5,000 predicted churners, randomly give half of them the retention offer, measure the difference. Now you have a causal number about the offer, and the

model's only job was to make the experiment cheaper by concentrating the population.

3. **Say the honest sentence.** "This tells us where the risk is. It does not tell us what to do about it, and if we act on the coefficients we may make things worse." This sentence has saved more money than most model improvements.

The limitation, stated plainly

Explanation tools — SHAP values, permutation importance, partial dependence plots — are explanations *of the model*, not of the world. A SHAP value tells you how much this model's output moved because of this feature. That is genuinely useful for debugging, for spotting leakage, and for satisfying a regulator who asks why an application was declined. It is not an estimate of what would happen if you changed the feature.

The distinction sounds pedantic until the day a team acts on a SHAP plot, spends a quarter on the intervention it suggested, and measures nothing. Most people meet this once. The ones who read this lesson meet it in a lesson instead.

BEFORE YOU MOVE ON

A retention model shows that customers who use the mobile app churn much less. The team proposes pushing every customer to install the app. What is the strongest reason to doubt the plan before running it?

A. The relationship may run the other way or come from a common cause — engaged customers both install the app and stay — so installing it for the disengaged need not change anything.

B. The app-usage feature is probably leaking, since usage is recorded after the churn window closes.

C. The model has not been tested for fairness across customer segments, so the policy could be discriminatory.

D. The feature's coefficient has not been checked for statistical significance, so the association may be noise.

7 · 8 min

The problems where the right answer is a rule

THE ONE THING TO KEEP

Machine learning earns its cost only when the decision rule is real but unwritable; when the rule is known, exact, or the data is too thin to tell a good model from a lucky one, code the rule.

A confession the field does not make often

A large share of shipped machine learning could be replaced by a few hundred lines of ordinary code with no loss of accuracy and a large gain in everything else. Knowing when you are in that situation is a skill, and it is one that makes you more employable rather than less, because the people paying for this work have usually been burned once.

Five situations where a rule wins

1. The rule is already known and exact. Tax brackets. Shipping bands. Eligibility criteria written in a regulation. These are not patterns to be discovered; they are specifications. A model that learns them will get 99.4% of them right, which is strictly worse than the `if` statement that gets 100%.

2. You have almost no labelled data. Under a few hundred labelled examples with any real complexity, a model's variance swamps its signal. You will not be able to tell a good model from a lucky one, because your validation set is fifty rows and the confidence interval on any score is enormous. Write the rule, ship it, and let it collect labelled data for the model you build next year.

3. Every decision must be explainable to the person affected, exactly. Some jurisdictions require a specific reason for an adverse decision. A five-condition rule gives you that for free. A gradient boosting model with 400

trees gives you a SHAP plot, which is an approximation of what the model did, and there are contexts where an approximation is not acceptable.

4. The cost of one wrong answer is catastrophic and unbounded. Not "expensive" — catastrophic. A model is a statistical device with a failure rate. If the failure rate must be provably zero on a class of inputs, that class needs a rule, possibly wrapped around a model that handles the rest.

5. The pattern changes faster than you can retrain. If the rules of the game change weekly, a model trained on last month's data is describing a world that no longer exists, and your monitoring will tell you so a week late.

Five situations where a model wins

The symmetric list, because the point is judgement rather than scepticism.

- **The rules exist but nobody can write them down.** Handwriting, speech, whether a photo contains a cat. Humans do it effortlessly and cannot articulate how. This is the original case for machine learning and it is still the strongest.
- **The rule list has grown past maintenance.** When the spam filter has 1,400 conditions and every new one breaks two old ones, you have hand-fitted a model badly. Let the optimiser do it.
- **You need a probability, not a decision.** Rules give you yes and no. Many businesses need "73% likely", so they can rank, prioritise or set a threshold that moves with capacity.
- **The relationship is genuinely multivariate.** Twelve weak signals that only mean something in combination. No human writes that rule; a model finds it in an afternoon.
- **Personalisation at a scale no human can staff.** A different ranking for each of two million users is not a rule problem.

The hybrid is usually the right shape

Most good production systems are not one or the other. They are a rule layer and a model layer with a stated division of labour.

```
if amount > 500000 and account_age_days < 7:
    route_to_human()        # hard rule, never overridden
elif model.predict_proba(x)[1] > 0.82:
    route_to_review()       # model decides in the grey zone
else:
    approve()
```

The rules cover the cases you already understand and the ones you cannot afford to get wrong. The model covers the enormous middle where nobody can write the condition. This shape is easier to reason about, easier to explain, and it degrades sensibly — if the model service goes down, the rules still run.

Addaly's own content moderation is built this way, and the reason is in the failure mode: a rule layer that catches the unambiguous cases means an AI outage does not become a safety outage.

The question to ask first

Before any project, ask: *if I gave a careful, experienced person the same information, could they write down how they decide?*

If yes, and the list is short, write the list. If yes, but the list would run to hundreds of interacting conditions, build a model. If no — if the person says "I just know" — build a model, and expect to need more data than you think.

The answer is not always flattering to the project. Saying so early is the cheapest thing you will ever do for a team.

BEFORE YOU MOVE ON

A logistics firm wants to predict which parcels need customs paperwork. The requirement is published as a document listing exactly which goods, values and destinations trigger it. Why is a classifier the wrong tool here?

A. Customs data is too sensitive to use for model training under most data-protection regimes.
B. There would not be enough historical parcels to train on, so the model would overfit.
C. The decision is a published specification, so a rule reproduces it exactly while any model will have a nonzero error rate on cases the specification already settles.
D. Classifiers cannot handle categorical inputs like destination country without extensive encoding work.

8 · 10 min

The whole loop, once, before we slow down

THE ONE THING TO KEEP

The whole discipline is nine steps with one irreversible arrow: the test set is touched once, at the end, after every decision has been made without it.

A map you can hold in your head

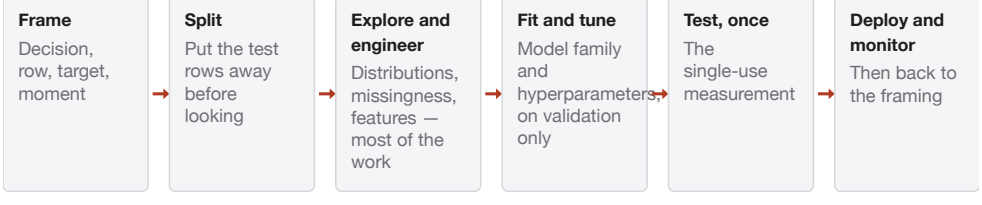
Everything in the next eight modules is one of nine steps. Seeing them together now means each later lesson lands somewhere rather than floating.

1. **Frame.** What decision, what row, what target, what moment. From the previous lessons.
2. **Collect and split.** Get the data, and put a portion away before you look at any of it.

3. **Explore.** Distributions, missingness, obvious errors, the target's base rate.
4. **Engineer features.** The bulk of the work. Module 3.
5. **Choose a model family and fit.** Modules 2, 6, 8.
6. **Tune against validation.** Module 4.
7. **Evaluate on test, once.** Module 5.
8. **Deploy.** Module 9.
9. **Monitor, and go back to step 1.** Also module 9.

The arrows go both ways between 3 and 6 many times. They go one way into 7, and that is the discipline the whole field is built on.

The loop, with its one irreversible arrow



Exploring, engineering and tuning cycle many times, all against validation data. The arrow into the test set goes one way and is followed once, after every decision has been made without it.

The same loop in thirty lines

This runs on a free Colab or Kaggle notebook, on a laptop with 4 GB of RAM, and it contains a version of every step above.

```
import pandas as pd
from sklearn.model_selection import train_test_split, cross_-
val_score
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.dummy import DummyClassifier
from sklearn.metrics import roc_auc_score

df = pd.read_csv("applications.csv")
y = df.pop("defaulted_90d")

# 2. split first, look second
X_tr, X_te, y_tr, y_te = train_test_split(
    df, y, test_size=0.2, stratify=y, random_state=0)

num = X_tr.select_dtypes("number").columns
cat = X_tr.select_dtypes("object").columns

prep = ColumnTransformer([
    ("num", Pipeline([("imp",
SimpleImputer(strategy="median")),
                      ("sc", StandardScaler())]), num),
    ("cat", OneHotEncoder(handle_unknown="ignore"), cat),
])

pipe = Pipeline([("prep", prep),
                 ("model",
HistGradientBoostingClassifier(random_state=0))])

# 5-6. fit and tune using cross-validation on TRAIN only
print("cv auc", cross_val_score(pipe, X_tr, y_tr, cv=5,
scoring="roc_auc").mean())

# baseline for comparison
base = DummyClassifier(strategy="prior").fit(X_tr, y_tr)
print("baseline auc", roc_auc_score(y_te,
base.predict_proba(X_te)[:, 1]))

# 7. test set, once, at the end
pipe.fit(X_tr, y_tr)
```

```
print("test auc", roc_auc_score(y_te, pipe.predict_proba(X_te)
[:, 1]))
```

Read it for the shape, not the details. Four things in there are load-bearing and worth naming now.

The split happens on line 12, before any exploration. Every transformation after it is fitted inside a `Pipeline`, which means the median used for imputation and the mean used for scaling are computed on training rows only. Do that by hand instead and you will one day compute the median on the whole dataset, leak test information into training, and get a score you cannot reproduce in production.

`handle_unknown="ignore"`. A category appears in production that was not in training. Without this line, the pipeline raises and your service returns a 500. With it, that row gets all-zeros for the category block and a prediction still comes out. Neither is obviously right; the point is that you chose.

Cross-validation runs on training data only. The test set is not involved in any decision.

The test line appears once, at the end, after everything else is settled.

Where the time actually goes

If you have read that machine learning is 80% data cleaning, the number is folklore but the direction is right. A more useful breakdown, from projects that shipped:

- Deciding the question, the row and the target: more than people expect, and it is the highest-leverage time in the project.
- Getting the data and making it trustworthy: the largest single block, and the least visible.
- Feature engineering: the second largest, and where most accuracy is won on tabular problems.
- Model selection and tuning: usually far smaller than the excitement about it suggests. `HistGradientBoostingClassifier` with default settings is a

strong model; tuning typically buys single-digit percentage improvements over a sensible default.

- Evaluation, deployment and monitoring: the part that decides whether the work survives a year.

The first three of those five are covered in modules 1, 3 and 5. That is deliberate.

What to do with this lesson

Run the code on any dataset you can find — the Titanic set, a CSV of your own expenses, anything with a column you could try to predict. It will fail in some interesting way. Getting it to run once, end to end, on your own machine or a free notebook, is worth more than the next three lessons read passively, and it gives every later lesson something concrete to attach to.

BEFORE YOU MOVE ON

In the pipeline above, why does putting the imputer and scaler inside `Pipeline` rather than applying them to the full dataframe first actually change the resulting numbers?

A. It ensures the transformations run in the correct order, which manual application can get wrong.

B. It makes the code shorter and reusable, so there is less chance of a typo between the training and prediction paths.

C. It allows the same object to be serialised for deployment, so training and serving stay in sync.

D. Inside a pipeline, the median and the mean are computed from training rows only, so no information from the held-out rows reaches the model.

9 · 8 min

A working setup that costs nothing

THE ONE THING TO KEEP

The standard toolchain for classical machine learning is free and open source; the ceiling on free compute constrains pretraining large models, not the tabular work that most jobs consist of.

What you actually need

Everything in this course runs on free software. Nothing here requires a paid licence, a GPU, or a machine you do not already have. Job adverts will name paid tools, and it is worth knowing what they are, but the free path is not a compromised version of this subject — in classical machine learning it is the industry standard.

The stack, all free and open source:

- **Python** with **NumPy** for arrays, **pandas** for tables, **scikit-learn** for models, **matplotlib** for plots. This is what most working data scientists use, including at large companies. There is no paid tier.
- **XGBoost**, **LightGBM** and **CatBoost** — the gradient boosting libraries that win most tabular competitions. Free, and they run on a CPU.
- **PyTorch** for neural networks. Free. Its main commercial alternative is TensorFlow, which is also free.
- **JupyterLab** for notebooks, or **VS Code** with the Python extension, or **VSCodium** if you prefer a build with no telemetry.

The paid tools you will see in job adverts are **MATLAB** (use **GNU Octave**, which runs most MATLAB code unchanged), **SAS** and **SPSS** (use Python or **R**, both free), **Tableau** (use **Metabase** or matplotlib), and **DataRobot** or **Vertex AI** for AutoML (see the AutoML lesson in module 9 for free equivalents). Learn to say "I have done this in scikit-learn" in an interview; nobody serious will hold it against you.

If you do not have a laptop

This is the more common case than the industry admits, and it is workable.

- **Google Colab** gives you a free notebook in a browser with about 12 GB of RAM and, in the free tier, intermittent GPU access. It runs on a phone browser in a pinch, though typing code on a phone is miserable and you should not plan a career around it.
- **Kaggle Notebooks** give you a free environment with roughly 30 hours a week of GPU time, which is considerably more than Colab's free tier, plus thousands of hosted datasets you can attach with one click and no download.
- Both save to your account. Neither requires a card.

For this course, Kaggle is the better default because the datasets are already there. Open a notebook, attach a dataset, and the code in the previous lesson runs.

Installing locally, if you can

```
python -m venv .venv
source .venv/bin/activate        # Windows: .venv\Scripts\acti-
vate
pip install numpy pandas scikit-learn matplotlib jupyterlab
jupyter lab
```

Use a virtual environment every time. The reason is not tidiness: two projects that need different versions of the same library will fight, and the loser is whichever one you touched second. A `.venv` per project makes that impossible.

If `pip install` fights you over compiled dependencies — common on Windows and on older Macs — use **Miniconda** instead, which ships pre-built binaries:

```
conda create -n ml python=3.11 numpy pandas scikit-learn mat-
plotlib jupyterlab
conda activate ml
```

Datasets to practise on, legally

Use these rather than scraping something. Every one is free and licensed for learning.

- **scikit-learn's built-in sets**: `load_diabetes`, `load_wine`, `fetch_california_housing`. They load in one line and need no download.
- **The UCI Machine Learning Repository** — hundreds of small, well-documented, classic datasets.
- **Kaggle Datasets** — larger and messier, which is realistic.
- **data.gov.in**, **data.gov.uk** and equivalents — real public data with real problems in it, which is the best practice you can get.

A note on the famous ones. The Boston Housing dataset was removed from scikit-learn in version 1.2 because one of its features encodes the proportion of Black residents in a way that was used as a predictor of house price. It is still in old tutorials. Do not use it; `fetch_california_housing` is the direct replacement and the reason for the swap is worth knowing, because a dataset's history is part of what you are teaching a model.

Two habits that save you

Set the random seed and record it. `random_state=0` everywhere. Results that change every run cannot be debugged and cannot be reported.

Version the environment. `pip freeze > requirements.txt` at the start of anything you might revisit. Six months later, an unpinned library upgrade will change your numbers, and you will spend two days finding out why.

The honest limitation

Free compute has ceilings. Colab and Kaggle will disconnect long-running jobs, free GPU quotas run out, and neither is suitable for training a large model from scratch. Everything in this course fits comfortably inside those ceilings — classical machine learning on tabular data is not compute-hungry, and a gradient boosting model on 100,000 rows trains in under a minute on a CPU.

What you cannot do on free hardware is pretrain a large language model. What you can do is use one that somebody else pretrained, fine-tune a small one, and do the entire body of work that most machine learning jobs actually consist of.

BEFORE YOU MOVE ON

Why does creating a virtual environment per project matter more than it appears to for someone learning alone on one machine?

A. Virtual environments isolate your data files, so an experiment cannot accidentally overwrite another project's dataset.

B. Two projects needing different library versions will otherwise share one installation, so upgrading for the newer project silently changes the older project's results.

C. Virtual environments make imports faster because Python searches a smaller set of packages.

D. Without one, pip needs administrator rights and will fail on most systems.

CASE STUDY · MODULE 1

Two definitions of a dropout, in Kota

A coaching centre in Kota with 2,300 students, two counsellors, and an analyst borrowed from the accounts department.

Vidya Path enrols 2,300 students across one-year and two-year programmes for the engineering entrance exam. Fees arrive in three instalments. A student who stops coming after the second instalment costs the centre the third, and costs the family a great deal more than that.

The management asked for a model that would "predict dropouts early enough to do something about it". They had five years of attendance registers, weekly test scores, fee receipts and hostel allocations, and they had Reshma, moved across from accounts, who had worked through a free machine learning course on her phone over four months.

Her first week produced no code. It produced one sentence, and an argument about it.

The row was easy. One student-month. A student appears once for every month they were enrolled.

The target was not. Two definitions were available, and they were not variations on a theme.

Definition A: a student is a dropout if they do not re-enrol for the following term. Unambiguous, already recorded in the fee system, and exactly what the director meant by the word. It also produces one label per student per year — about 2,300 rows a year and roughly 180 positives.

Definition B: a student is a dropout if they attend no class for twenty-one consecutive days. One label per student-month, which over five years is about 27,000 rows and roughly 1,900 positives. It is also wrong sometimes. A student with dengue, or one who went home for a cousin's wedding and came back, gets the same label as one who has left for good.

The director wanted A. "That is what a dropout is." Reshma wanted B, for a reason that had nothing to do with elegance. With 180 positives and forty can-

didate features, the events-per-feature arithmetic said nothing she fitted would be stable, and a validation set holding thirty-six positives could not tell a good model from a lucky one.

The cost fell on both sides and neither was small. Definition A would produce a model whose per-student predictions could not be trusted, wearing a validation score with an interval wide enough to cover both success and failure. Definition B would produce a model that fired on students who were about to walk back through the gate, and every false alarm was a counsellor telephoning a family that had done nothing wrong. In Kota, where the pressure on these students is a matter of public concern, an unnecessary call from the institute is not a neutral event.

The moment of prediction settled less than either of them expected, and it settled it the same way for both definitions. Features had to be everything known as of the last day of the month and nothing else. That excluded `refund_processed`, `hostel_vacated_date` and `final_instalment_status`, all three of which were in the export the office had handed over, and all three of which are written after the outcome they would have been used to predict. Reshma put the exclusion list at the top of the notebook before she opened a single CSV.

The baseline took twenty minutes and changed the conversation. Predicting "nobody drops out" scored ninety-three per cent under definition A. More usefully, the rule the counselling team already used — flag any student whose attendance over the last fortnight fell below sixty per cent — caught forty-one per cent of eventual dropouts and produced about nine flags a day, which two counsellors could just about work through.

Nine flags a day was the real constraint, and nobody had written it down before. The question stopped being "can we predict dropouts" and became "can we beat forty-one per cent recall at nine flags a day, without asking the counsellors to make calls they do not have time to make".

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

They trained on definition B and evaluated on definition A. The noisy monthly label gave the optimiser enough events to learn from; the clean annual label, available for the two cohorts that had completed, was what they reported. At nine flags a day the model reached sixty-three per cent recall against the rule's forty-one.

The first twenty calls found three students on recorded medical leave. Rather than retrain, the centre added a suppression rule outside the model — never flag a student with an approved leave record — which is the rule-and-model hybrid from lesson seven of the module, arrived at by being embarrassed rather than by design.

The director's requirement, added late and kept since, was one line in the handover note: the model predicts a twenty-one-day absence, not a departure, and the counsellor is told which of those they are calling about.

Definition B gave ten times as many labelled events. Why did that not simply make the model better?

Because the extra events are noisier. A twenty-one-day absence includes students who return, so a share of the positive labels do not correspond to the outcome anyone cares about. The model learns the label it is given, so it learns 'who will disappear for three weeks', which overlaps with dropping out without being it. The gain in estimation stability was real and the cost in label fidelity was real; the resolution was to train on the plentiful label and measure against the clean one.

The refund column is strongly associated with dropping out. Why exclude it?

Because at the moment the prediction is made — the last day of a month, for a student still enrolled — the refund has not been processed. It exists in the training table only for students whose outcome is already known, so it is a consequence of the

target rather than a predictor of it. Including it would raise every validation score and produce a model that has nothing to say about a student who is still attending, which is the only kind of student the counsellors can act on.

Why was 'nine flags a day' a better target than 'maximise AUC'?

Because the counsellors are the constraint, and the decision the model supports is which nine students get a call today. AUC averages the model's behaviour over thresholds that will never be used. Precision and recall at the operating point of nine flags describe what will actually happen, they are comparable against the rule already in place, and they make the trade-off legible to a director who does not know what an ROC curve is.

MODULE 1 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 A delivery-time model is trained on data where most trips take 20 to 40 minutes and a handful take three hours because of a road closure. The team switches the training loss from squared error to absolute error and changes nothing else. What changes about the fitted model?

A. It fits closer to the median trip and the rare long trips pull it much less.

B. It becomes more accurate on the long trips, because every minute of error now counts equally.

C. Its predictions stop being biased, because absolute error is an unbiased estimator of the mean.

D. Nothing changes until the learning rate is also lowered to match the new scale of the loss.

2 A churn model reaches 0.99 AUC on its first run, with no tuning, on a problem the operations team finds genuinely difficult. What is the most likely explanation?

A. The features are unusually strong, and churn is simply easier to predict than the team believes.

B. The test set is too small, so the estimate is noisy and this run happened to come out high.

C. A column recorded after the outcome has reached the feature set.

D. The classes are imbalanced, and AUC is inflated whenever one class dominates the data.

3 A bank builds a default model where one row is one loan, and customers hold up to eleven loans each. The rows are split at random. What does the validation score actually measure?

A. How well the model predicts for customers it has never seen, because the individual loans differ.

B. Nothing at all, because a loan-level row cannot be used to predict a customer-level outcome.

C. The right thing, but pessimistically, because one customer's loans are correlated with each other.

D. How well the model predicts for customers whose other loans it trained on.

4 A telecom model finds that customers who call support are far more likely to churn. Management proposes making the support line harder to reach. What is wrong with that reasoning?

A. The coefficient is unstandardised, so its size cannot be compared with the other features.

B. The call and the churn share a cause, so removing the call removes the signal.

C. The model was trained on historical data and would need retraining before it is acted upon.

D. Support calls are rare, so the association is unlikely to be statistically significant.

5 A government office applies an eligibility rule written into a regulation: five conditions, each exactly specified. A team proposes training a classifier on past decisions instead. Why is that a worse design?

A. A classifier needs at least ten thousand examples and the office has fewer decisions than that.

B. The classifier would learn the regulation correctly and then drift as the input data changes.

C. The rule is a specification, not a pattern, so a model can only approximate it.

D. The model's probabilities would be uncalibrated, so the office could not state a confidence level.

6 A team evaluates on the test set, sees a disappointing number, changes two features, and evaluates again. What is the state of their test estimate now?

A. It is optimistic, because a decision was made using it.

B. It is unchanged and still valid, because they never trained on the test rows themselves.

C. It is pessimistic, because the second evaluation used a model tuned for a different feature set.

D. It is valid as long as they report the first number rather than the second one.

MODULE 2

Linear models and the machinery of fitting

The simplest useful model, taken seriously. Fit a line by hand, watch an optimiser find the same answer, then extend it to probabilities, to many classes, and to curves — and learn to read the coefficients without believing more than they say.

> BY THE END OF THIS MODULE YOU CAN
>
> Fit and diagnose a linear or logistic model end to end: compute its loss by hand on a handful of rows, tune the learning rate from the shape of the loss curve, state what a coefficient does and does not mean in the units of the data, and explain what ridge and lasso penalties change about the fitted parameters

10 · 9 min

Fitting a line to five points, by hand

> THE ONE THING TO KEEP
>
> Fitting means choosing parameter values that make a chosen error number small; the coefficient carries real units, and a linear model will extrapolate far outside its data without any signal that it is guessing.

Five houses

Size in square metres, price in lakhs.

```
size   price
 50     32
 70     44
 90     51
110     67
130     74
```

We want `price = w * size + b`. Two numbers. Everything about model fitting is visible in finding them.

Guess, and measure how wrong you are

Start with `w = 0.5`, `b = 0`. Predictions are 25, 35, 45, 55, 65. The errors — actual minus predicted — are 7, 9, 6, 12, 9.

Mean squared error:

```
(49 + 81 + 36 + 144 + 81) / 5 = 391 / 5 = 78.2
```

Now try `w = 0.55`, `b = 3`. Predictions: 30.5, 41.5, 52.5, 63.5, 74.5. Errors: 1.5, 2.5, -1.5, 3.5, -0.5. MSE = `(2.25 + 6.25 + 2.25 + 12.25 + 0.25) / 5 = 4.65`.

The second model is better and we know it by a number, not by looking. That number is the entire mechanism. Everything else in this module is about finding good values of `w` and `b` faster than by guessing.

Which direction is downhill?

A useful thing about MSE is that you can compute, for the parameters you are currently holding, which way to move them. For one row the loss is `(y - (w*x + b))^2`, and the amount the loss changes when you nudge `w` is

```
d/dw = −2 * x * (y − prediction)
```

Read that without the calculus. It says: the push on `w` is proportional to the error, and scaled by `x`. A row where you are badly wrong pushes hard. A row with a large `x` pushes harder than one with a small `x` for the same error — which is exactly why unscaled features cause trouble later.

For the first house at `w = 0.5, b = 0`: error is 7, `x` is 50, so the derivative is `−2 * 50 * 7 = −700`. Negative, meaning increasing `w` reduces the loss. Averaged over all five rows, and moved a small step at a time, this is gradient descent, which you will meet properly in two lessons.

Doing it in one line

```
import numpy as np
from sklearn.linear_model import LinearRegression

X = np.array([[50], [70], [90], [110], [130]])
y = np.array([32, 44, 51, 67, 74])

m = LinearRegression().fit(X, y)
m.coef_, m.intercept_        # array([0.525]), 6.15
m.predict([[100]])           # array([58.65])
```

The best possible line for these five points is `price = 0.525 * size + 6.15`, MSE 3.585. Our hand guess of `0.55` and `3` got to 4.65 — respectable, and it shows that near the optimum the loss surface is fairly flat, which is a property we will come back to.

What the two numbers mean

`w = 0.525` means: **within this data**, each additional square metre is associated with 0.525 lakh more in price. Units are lakhs per square metre. A coefficient always has units, and stating them is the fastest way to catch a nonsense model.

`b = 6.15` is the predicted price of a house of zero square metres, which does not exist. The intercept is usually not interpretable on its own; it is the number that positions the line vertically. If you centre your features first — sub-

tract the mean from each — the intercept becomes the prediction at the average house, which is interpretable.

Two limitations you can see with five points

Extrapolation is a guess dressed as arithmetic. Ask this model for a 400 square metre house and it answers 216. It has never seen a house above 130 and has no way to know the relationship flattens, or that a house that large is a different market. The model will not warn you. Nothing in a linear model warns you when you leave the range of the training data, and this is the failure that most often reaches production quietly.

Five points cannot tell you the relationship is linear. They are consistent with a line. They are also consistent with a gentle curve, and with the true relationship being a step at 100 square metres. With five points you cannot distinguish these; with five hundred you often can. The straight line is an assumption you imposed, not a finding.

Try changing one number

Change the last price from 74 to 140 — say a penthouse crept into the data — and refit. The coefficient jumps from 0.525 to about 0.855 and the whole line tilts. One row in five moved the answer by 63%.

That sensitivity is the squared-error property from the previous module, made visible. With 5,000 rows one outlier moves little. With 50 rows it moves a lot. It is worth knowing which regime you are in before you defend a coefficient in a meeting.

Why this simple thing survives

Linear regression is 220 years old and still the first model to fit on a new problem. It trains in milliseconds, gives a number per feature, extrapolates predictably rather than catastrophically, and needs no tuning. As the baseline lesson argued, you need something to beat. Half the time on tabular data, the fancy model beats this by less than anyone expected, and that fact is worth discovering in the first hour rather than the last.

BEFORE YOU MOVE ON

You fit a line to 40 houses between 50 and 130 square metres and get 0.525 lakh per square metre. A colleague uses it to price a 400 square metre villa. What is the specific defect in that use?

A. The model was fitted with MSE, which is sensitive to outliers, and a 400 square metre house is an outlier.
B. The coefficient is only valid for the range of sizes observed; outside it the model applies a slope it has no evidence for, and it produces the extrapolation with no indication of doing so.
C. The intercept of 6.15 becomes meaningless at large sizes, so the prediction is offset by that amount.
D. Linear regression requires at least 100 rows to be valid, and 40 is too few for any prediction.

11 · 8 min

There is an exact answer, and we usually refuse it

THE ONE THING TO KEEP

Linear regression with squared error has an exact one-step solution; gradient descent is taught because it is the only method that still works when the loss or the model has no such formula.

The formula that ends the search

For linear regression with squared error, you do not need to search at all. There is an exact solution, written in one line of linear algebra:

```
w = (X'X)^−1 X'y
```

where `X'` is the transpose and `^-1` is the matrix inverse. This is the **normal equation**, and it gives the parameters that minimise MSE exactly, in one step, with no learning rate, no iterations and no convergence to worry about.

```
import numpy as np
X = np.array([[50], [70], [90], [110], [130]])
X1 = np.hstack([np.ones((5, 1)), X])          # add the inter-
cept column
y = np.array([32, 44, 51, 67, 74])
np.linalg.solve(X1.T @ X1, X1.T @ y)          # array([6.15,
0.525])
```

Exactly the numbers scikit-learn returned. In fact `LinearRegression` does something very close to this internally — it uses a numerically safer routine called least squares via SVD, but the spirit is the same. There is no gradient descent inside it.

So why does every course teach gradient descent?

Three reasons the exact answer runs out

1. Cost. `X'X` is a `p` by `p` matrix where `p` is the number of features, and inverting it costs roughly `p^3` operations. At 10 features that is nothing. At 1,000 features it is a billion operations — still fine. At 100,000 features, which a text model with one column per word reaches easily, it is 10^15 operations and a matrix of 10 billion entries. The formula has not become wrong; it has become unusable.

2. Memory. The closed form needs all of `X` at once. Gradient descent needs one batch at a time, which is why models train on datasets larger than the machine's RAM. This is not a minor engineering convenience — it is the reason large models are trainable at all.

3. Most losses have no closed form. This is the deep reason. The normal equation exists because squared error against a linear function is a quadratic bowl with one bottom that calculus can locate. Change the loss to log loss, or the model to anything with a nonlinearity in it, and no such formula exists. Logistic regression — next lesson but one — has no closed form. Neural net-

works certainly do not. Gradient descent is the general method; the normal equation is a lucky special case.

The singular matrix, and what it is telling you

Sometimes `X'X` cannot be inverted at all. The mathematical term is singular; the practical meaning is that two or more of your columns carry the same information.

The classic way to cause it: one-hot encode a categorical column into all of its levels *and* keep the intercept. If `is_mumbai + is_delhi + is_chennai = 1` for every row, and the intercept column is also all 1s, then one column is an exact combination of others. There are infinitely many parameter sets that fit equally well, so there is no unique answer, and the inverse does not exist.

Numerically it usually does not crash. It produces enormous coefficients of opposite sign that cancel — `+184,000` on one city and `−183,000` on another — which is the signature you should learn to recognise. When you see coefficients that large on ordinary data, look for collinear columns before you look for anything else.

The fixes: drop one level of each categorical (`drop='first'` in scikit-learn's `OneHotEncoder`), or add a ridge penalty, which we cover later in this module and which makes the matrix invertible by construction.

Which one runs when you call `.fit()`

A useful piece of practical knowledge, because the library names are opaque:

- `LinearRegression` — least squares via SVD. Exact, no iterations, no learning rate. Good up to tens of thousands of features.
- `Ridge` with `solver='auto'` — closed form with the penalty added, usually.
- `SGDRegressor` and `SGDClassifier` — genuinely iterative. Use these when the data does not fit in memory or when you want to fit incrementally with `partial_fit`.

- `LogisticRegression` — always iterative. There is no alternative; the `max_iter` warning you have probably seen is this optimiser saying it stopped before converging.

That last point explains a common confusion. "Increase `max_iter`" is the standard advice for the logistic regression convergence warning, and it works, but the more effective fix is usually to scale your features — an optimiser on badly scaled data takes many more steps to reach the same place, for reasons the next two lessons make concrete.

The honest summary

Exact solutions are better when they exist and are affordable. They exist for one loss and one model family. The field standardised on iterative optimisation not because it is more elegant but because it is the only method that survives contact with a nonlinear model, and once you have built the machinery you use it everywhere.

BEFORE YOU MOVE ON

A model fitted on one-hot encoded city data returns coefficients around +184,000 and -183,000 on two city columns, though prices range from 20 to 200. What has most likely happened?

A. The learning rate was too high, so the optimiser overshot and the parameters diverged.

B. The city column has too many levels for the number of rows, so each city coefficient is estimated from a handful of examples.

C. The one-hot columns plus the intercept are linearly dependent, so infinitely many coefficient sets fit equally well and the solver returned one with huge cancelling values.

D. The target was not scaled, so the coefficients absorbed the target's magnitude.

12 · 8 min

Gradient descent, or walking downhill in fog

THE ONE THING TO KEEP

Gradient descent only knows the slope where it is standing; the step size decides whether that knowledge helps or destroys it.

The picture

You are on a hillside in thick fog. You want the bottom of the valley. You cannot see it. But you can feel which way the ground slopes under your feet, so you take a step in the downhill direction, feel again, and step again.

That is gradient descent, complete. The hill is the loss as a function of the parameters. Your position is the current parameter values. The slope is the gradient. The step size is the learning rate.

An actual walk

One parameter, and a loss with a known bottom:

```
L(w) = (w − 3)²
```

The slope at any `w` is `2(w − 3)`. Start at `w = 0` with learning rate 0.1. The rule is: new `w` = old `w` − learning rate × slope.

Step	w	Slope	New w
1	0.000	−6.00	0.600
2	0.600	−4.80	1.080
3	1.080	−3.84	1.464
4	1.464	−3.07	1.771
…			
20	2.965	−0.07	2.972

It closes in on 3, taking smaller steps as the slope flattens, because near the bottom there is less slope to push it. Nobody told it the answer was 3. It only ever felt the ground where it stood.

Now set the learning rate to 1.1 and start again:

Step	w	Slope	New w
1	0.00	−6.00	6.60
2	6.60	7.20	−1.32
3	−1.32	−8.64	8.18
4	8.18	10.37	−3.22

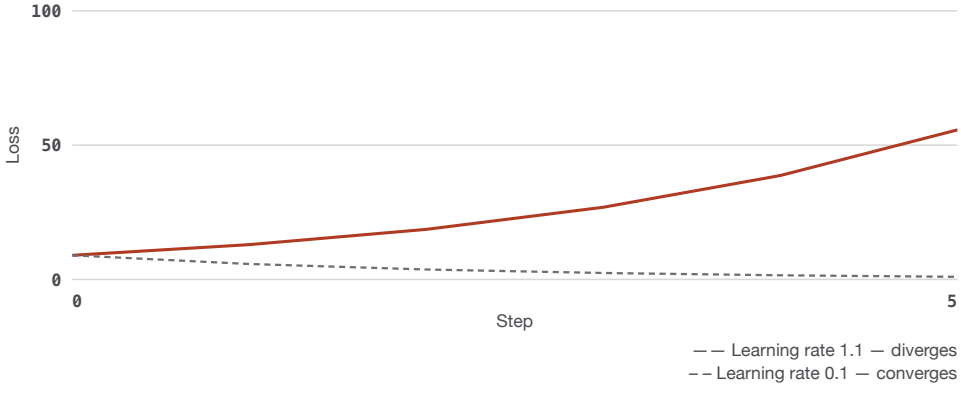
Each step overshoots the bottom, lands further up the far side, and the next slope is steeper still. Within thirty steps the numbers exceed what a float can hold and your loss prints as `nan`. This is not an exotic failure. It is the single most common way training dies, and the fix is to lower the learning rate — usually by a factor of ten and then look again.

Too large diverges. Too small crawls: at a learning rate of 0.0001 the walk above needs tens of thousands of steps. There is no formula for the right value. You try 0.1, 0.01, 0.001 on validation and watch the loss curve.

Real models have many parameters

With two parameters the loss is a surface and the gradient is a pair of numbers, one slope per parameter, each saying how much the loss changes when that one parameter moves. With ten million parameters it is a vector of ten million slopes, and the update rule is the same line of arithmetic applied to each. The mental picture does not need to scale; the arithmetic does, and it does.

Loss after each step, two learning rates, L(w) = (w – 3)²



From w = 0, a rate of 0.1 brings the loss from 9 to under 1 in five steps, each step smaller as the slope flattens. A rate of 1.1 overshoots the bottom every time and lands higher on the far side: 9, 13, 19, 27, 39, 56. Within thirty steps this prints as NaN.

How much data per step

Computing the exact slope means running every training row through the model. With eight million rows that is one step per several minutes, which is unusable.

Mini-batch gradient descent estimates the slope from a random sample — 32, 128, 512 rows — and steps on that estimate. The estimate is noisy, so the path wanders rather than descending smoothly. That turns out to help: the noise shakes the walk out of shallow dips it would otherwise settle in, and it lets you take thousands of steps in the time one exact step would cost. Nearly all modern training is mini-batch.

Getting stuck, honestly

The standard worry is local minima: a small dip that is not the true bottom, where the slope is zero in every direction and the walk stops.

For the classic models this cannot happen. Linear regression, logistic regression and linear SVMs have convex losses — a single bowl, one bottom, and gradient descent finds it from anywhere.

For neural networks there are many minima, and for a long time this was assumed to be the central difficulty. It largely was not. In high dimensions the common flat spots are **saddle points**, where the surface rises in some directions and falls in others, and a walk can be slowed near one without being trapped. And when a large network does settle in a minimum, the minima it reaches tend to be about equally good, so which one you land in matters less than expected.

What helps in practice is momentum — keep some of your previous direction, so you roll through flat regions instead of stalling — and adaptive methods like Adam, which give each parameter its own effective step size. Both are refinements of the same walk. Feel the slope, take a step, repeat.

BEFORE YOU MOVE ON

You start training a network. Loss goes 4.2, then 91.4, then 3.1e6, then `nan` by step 30. What is the most likely cause?

A. The learning rate is too large, so each step overshoots and lands somewhere steeper than before.

B. The model is too small to fit the data.

C. The optimiser is stuck in a local minimum.

D. There is not enough training data.

13 · 9 min

Batches, epochs, and the two knobs that decide everything

THE ONE THING TO KEEP

Mini-batch size trades gradient noise against step count, and the learning rate must match the scale of the features — which is why standardising inputs fixes more convergence problems than raising the iteration limit.

Three ways to take one step

Gradient descent needs a gradient, and computing it means looking at data. How much data you look at before each step is a real choice with real consequences.

Full batch. Compute the gradient over every row, then take one step. The direction is the true average direction. On a million rows, one step costs a full pass over a million rows, so you might take 200 steps in an hour.

Stochastic (SGD). Compute the gradient from one row and step immediately. On a million rows you take a million steps per pass. Each direction is noisy — one row is a poor estimate of the average — but you take so many steps that the noise partly cancels and you get far more progress per unit of computation.

Mini-batch. Compute the gradient from 32, or 256, or 1,024 rows. This is what everything actually uses, because it gets most of SGD's step count while the averaging across a few dozen rows cuts most of the noise, and because a batch of 256 rows is a matrix multiplication that hardware is built to do fast, whereas 256 separate single-row updates is 256 slow operations.

An **epoch** is one full pass over the training data. With 100,000 rows and a batch size of 250, an epoch is 400 steps.

The noise is doing something useful

The standard framing treats mini-batch noise as a cost you pay for speed. That is incomplete. The noise also helps you escape places you would otherwise be stuck in — a flat region or a poor local minimum can trap full-batch descent, while the jitter of a noisy gradient shakes the parameters out of it.

This is why very large batches sometimes generalise slightly *worse* than moderate ones, even at equal accuracy on the training set. It is an active area, the effect is modest, and the practical advice is unexciting: 32 to 512 covers almost everything, start at 128 if you have no reason to prefer another number.

The learning rate is the knob that matters

If you can tune one thing, tune this. The failure modes are distinctive and you can name them from a plot.

- **Far too high.** Loss becomes `NaN` within a few steps. The parameters overshot so far that the next gradient was larger, then larger again. This is divergence and it is unmistakable.
- **Too high.** Loss falls, then bounces around a floor it never gets below, sometimes rising. The optimiser is stepping across the valley rather than down it.
- **About right.** Loss falls quickly, then more slowly, then flattens.
- **Too low.** Loss falls in a nearly straight line and is still falling when you run out of patience. Slow, but not wrong — and this is the safe direction to err in.

A practical starting point for a scaled linear or logistic model is 0.01. For neural networks with Adam, 0.001 is the near-universal default. Then move in factors of three — 0.003, 0.01, 0.03 — rather than in small increments; the useful range spans orders of magnitude.

Scaling and the learning rate are the same problem

This is the connection most courses leave implicit. Suppose one feature is age (18 to 80) and another is annual income (200,000 to 5,000,000). From the

by-hand lesson, the push on a weight is proportional to the feature value. So the income weight receives gradients tens of thousands of times larger than the age weight.

One learning rate has to serve both. Set it small enough not to blow up the income weight and the age weight barely moves; set it large enough to move age and income diverges. The loss surface is a long narrow ravine and the optimiser zigzags down it.

Standardising both features to mean 0 and standard deviation 1 turns the ravine into something closer to a bowl, and the same optimiser converges in a fraction of the steps. **This is why scaling is required for gradient-based models and irrelevant for trees** — trees split on one feature at a time and never combine them in a weighted sum, so the units cancel out of the whole procedure.

Schedules: start big, end small

A constant learning rate has to be small enough for the endgame, which wastes the beginning. So decay it.

```
from sklearn.linear_model import SGDRegressor
SGDRegressor(learning_rate="invscaling", eta0=0.01,
power_t=0.25)
```

The standard shapes are step decay (multiply by 0.1 every N epochs), cosine decay (a smooth fall to near zero), and warmup-then-decay (rise for the first few hundred steps, then fall), which is standard for transformers because early gradients on freshly initialised weights are unreliable and large early steps do damage.

A decaying rate also explains a shape you will see in loss curves: a visible cliff where the loss drops suddenly. That is usually not the model learning something; it is the learning rate stepping down and letting the optimiser settle into a place it had been bouncing around for an hour.

What to do first

If a model will not converge, the order to check is: scale the features, then lower the learning rate, then increase the iterations. Most people do that order backwards and spend an afternoon on it.

BEFORE YOU MOVE ON

A logistic regression on raw features (age in years, income in rupees, tenure in months) issues a convergence warning. Raising max_iter from 100 to 5,000 makes the warning stop but barely changes the coefficients' quality. Why does standardising the features work better?

A. Standardising removes outliers, which were dominating the log loss and preventing convergence.

B. Standardising reduces the number of features the optimiser must handle, so each step is cheaper.

C. The gradient for each weight scales with its feature's magnitude, so wildly different units create a long narrow loss surface that a single learning rate cannot descend efficiently; equalising the scales makes it round.

D. Logistic regression's sigmoid saturates on large inputs, so scaling is required for the model to produce any gradient at all.

14 · 8 min

Diagnosing a model from the shape of its loss curve

THE ONE THING TO KEEP

Plot training and validation loss on one axis every run: the gap diagnoses fit, the slope at the end diagnoses training length, and NaN or a flat line diagnoses a bug rather than a modelling problem.

The plot you should make every time

Two lines, x-axis is epochs or steps, y-axis is loss. Training loss and validation loss on the same axes. This one plot answers more questions than any other artefact in machine learning, and it takes four lines of code.

```
import matplotlib.pyplot as plt
plt.plot(history["train_loss"], label="train")
plt.plot(history["val_loss"], label="validation")
plt.legend(); plt.xlabel("epoch"); plt.ylabel("loss")
```

For scikit-learn models that do not record a history, `HistGradientBoostingClassifier` exposes `validation_score_` per iteration, and `SGDClassifier` can be driven with `partial_fit` in a loop to build one manually. For anything in PyTorch you write it yourself, and you should.

Seven shapes, and what each one means

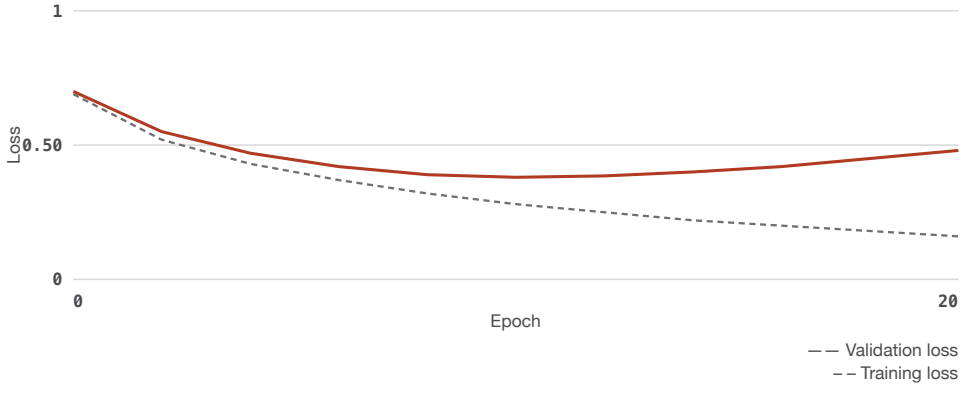
Both lines falling, still falling at the end. Undertrained. Train longer. The single most common mistake in a first project is stopping here and concluding the model is weak.

Both fall, then flatten together with a small gap. This is the healthy result. The remaining gap is normal; a gap of exactly zero is suspicious and usu-

ally means the model is too simple to have fitted anything specific.

Train falls, validation flattens and then rises. Overfitting, in its textbook form. The point where validation turns is where you should have stopped, and `early_stopping` does exactly that automatically.

The textbook overfitting curve



Training loss falls for as long as you let it. Validation loss bottoms out at epoch ten and rises from there, which is the model memorising this particular noise. The epoch where validation turns is where you should have stopped, and early_stopping does exactly that.

Both flat and high from step one. Nothing is learning. Check that the learning rate is not absurdly small, that the features actually reach the model, and that the labels are not shuffled relative to the features. A model that predicts the base rate forever usually has a plumbing fault, not a modelling one.

Loss becomes NaN. Learning rate far too high, or a division by zero somewhere in the features, or a log of zero in a custom loss. Drop the learning rate by a factor of ten first; it is the cause about eight times out of ten.

Validation loss below training loss. Surprising, and it has two innocent causes and one guilty one. Innocent: regularisation such as dropout is active during training and off during evaluation, so training is measured on a handicapped model; and training loss is often averaged over the epoch while validation is measured at the end, so training looks worse by the amount the model improved during that epoch. Guilty: your validation set is easier than your training set, which happens when the split is not random.

A sudden cliff downward. Usually a learning rate schedule stepping down, as described in the previous lesson. Sometimes an unfreezing event in a fine-

tuning run. Rarely a real discovery.

The noise is information too

A validation curve that jumps around wildly from epoch to epoch is telling you the validation set is small. The measurement noise on a set of 200 rows is large; with 5,000 rows the same model gives a smooth line. If your curve looks like static, do not tune against individual points — you will be chasing noise, and you will pick the epoch that got lucky.

This matters because early stopping picks the minimum of a noisy curve, which is biased low. That is fine for choosing when to stop and misleading if you then report that minimum as your score. Report the test set, once, afterwards.

Learning curve is a different plot

Worth separating now because the names get used loosely. The plot above has *epochs* on the x-axis. A **learning curve** has *training set size* on the x-axis: fit on 10% of the data, then 20%, and so on, plotting train and validation score at each size.

That second plot answers a different and very expensive question — *would more data help?* If the validation curve is still climbing at 100% of your data, collecting more will help. If it flattened at 40%, more of the same data will not, and you need better features or a different model. We build it properly in module 4; the distinction is worth having now so the two are never confused.

Save the history

A small habit with a large payoff: store the loss history alongside the model, with the hyperparameters, the seed, and the data version. Three weeks later someone asks why the model changed, and the answer is almost always visible in the two curves side by side. Reconstructing it from memory is not possible, and re-running it is often not either, because the data has moved on.

BEFORE YOU MOVE ON

A neural network's validation loss sits consistently *below* its training loss for every epoch. The split was random and stratified. What is the most likely explanation?

A. The validation set is contaminated with training rows, making it artificially easy.
B. Dropout is active during training and disabled at evaluation, so the training loss is measured on a deliberately handicapped model.
C. The model is underfitting, and underfitted models always show lower validation loss.
D. The validation set is smaller, so its loss is averaged over fewer rows and is therefore lower.

15 · 10 min

Logistic regression, built from odds rather than assumed

THE ONE THING TO KEEP

Logistic regression models the log-odds as a straight line, which is why the sigmoid appears and why a coefficient multiplies the odds by a constant factor rather than shifting the probability by a constant amount.

Why not just fit a line to 0 and 1?

You can. It is called the linear probability model, statisticians use it deliberately in some settings, and for a beginner it fails in two visible ways.

Fit `y = w*x + b` where `y` is 0 or 1, and the line will happily predict 1.4 and -0.3. Those are not probabilities. Worse, the fit is dragged by the squared-error penalty on points that are already unambiguously classified: a customer

the model puts at 0.99 who really is a 1 still contributes error, and the line tilts to reduce it, damaging the boundary where the decision actually happens.

Odds, and then log-odds

Start from something a bookmaker would recognise. If a probability is 0.75, the odds are `0.75 / 0.25 = 3`, said as three to one.

Probability lives in [0, 1]. Odds live in [0, infinity). Take the natural logarithm and log-odds live in (-infinity, +infinity), which is exactly the range a linear function produces.

So model the log-odds as linear:

```
log(p / (1 − p)) = w∗x + b
```

Solve that for `p` and you get the logistic function, which people usually meet as an unexplained curve:

```
p = 1 / (1 + exp(−(w∗x + b)))
```

The sigmoid was not chosen because it looks nice. It is what falls out when you decide to model log-odds with a straight line, and that decision was forced by needing an unbounded quantity for the linear part to produce.

Reading the numbers

The useful properties, worth committing to memory:

- Log-odds 0 means `p = 0.5`. Positive log-odds means over half.
- Log-odds 2.2 means `p = 0.9`. Log-odds -2.2 means `p = 0.1`.
- The curve is steepest at `p = 0.5` and flat at the ends. Near the decision boundary, small changes in the inputs move the probability a lot; far from it, they barely move it at all.

That last property is why logistic regression is well behaved: it concentrates its attention where the decision is difficult, which is what the linear probability

model failed to do.

The coefficient is an odds ratio

A coefficient of 0.7 on "has an overdraft" means: holding everything else fixed, having an overdraft adds 0.7 to the log-odds. Exponentiate and `exp(0.7) = 2.01`, so it roughly **doubles the odds** of the positive class.

Doubling the odds is not doubling the probability. From 0.10, doubled odds gives 0.18. From 0.40, it gives 0.57. From 0.80, it gives 0.89. The same coefficient produces a different change in probability depending on where you start, and this is the single most common misreading of a logistic model in a business meeting. The effect on probability is not constant; only the effect on log-odds is.

Fitting it

There is no closed form. The loss is log loss, from module 1, and it is convex — one bottom, no local minima — so an iterative optimiser reliably finds the global optimum. scikit-learn uses `lbfgs` by default, a quasi-Newton method that converges in far fewer iterations than plain gradient descent.

```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

clf = make_pipeline(StandardScaler(), LogisticRegression(C=1.0,
max_iter=1000))
clf.fit(X_tr, y_tr)
clf.predict_proba(X_te)[:, 1]        # probabilities, not
labels
```

Use `predict_proba`, not `predict`. `predict` silently applies a threshold of 0.5, and 0.5 is almost never the right threshold — module 5 is largely about that.

One trap worth naming: scikit-learn's `LogisticRegression` applies **L2 regularisation by default**, controlled by `C`, where smaller `C` means stronger

penalty. This surprises people coming from R or statsmodels, where the default is unpenalised. If you are comparing coefficients with a statistician's output and they differ, this is usually why. Set `penalty=None` for an unpenalised fit.

Separation, and the coefficient that runs away

If some feature perfectly separates the classes — every row with `is_fraud_flagged = 1` is fraud, no exceptions — then log loss keeps improving as that coefficient grows towards infinity, because pushing the predicted probability from 0.999 to 0.9999 still reduces the loss. Unpenalised, the optimiser chases it until it hits the iteration limit and reports a coefficient of 34.

This is called complete separation. It is a data problem masquerading as a convergence warning, and the default L2 penalty is what usually prevents you from ever noticing it. When you do see a coefficient in the tens, check whether that feature is a giveaway — and then check whether it is leakage, because a perfect separator in training data usually is.

What it is genuinely good at

Fast, calibrated out of the box in a way most models are not, gives one interpretable number per feature, and it is the standard in credit scoring and clinical prediction precisely because a regulator or a clinician can read it. When a gradient boosting model beats it by two points of AUC and cannot be explained to the person being refused a loan, the two points may not be worth it.

BEFORE YOU MOVE ON

A model gives 'previous default' a coefficient of 0.69, so exp(0.69) is about 2. A manager concludes that previous default doubles a customer's chance of defaulting again. What is wrong with that reading?

A. The coefficient is on the log-odds scale, so it doubles the odds; the change in probability depends on the starting probability and is smaller than doubling except at very low base rates.

B. Coefficients from a regularised fit are shrunk toward zero, so the true effect is larger than 2.

C. The coefficient describes the average customer, so it cannot be applied to any individual.

D. Doubling only holds when all other features are zero, and they never are.

16 · 9 min

What a coefficient says, and the four ways it lies

THE ONE THING TO KEEP

A coefficient is a change in the target per unit of one feature, holding the others in the model fixed — so it is only comparable when standardised, only stable when features are uncorrelated, and only meaningful relative to the exact set of columns included.

The sentence a coefficient supports

Here is the only claim a linear coefficient licenses, and it is worth memorising in full:

> *Among rows like the ones in my training data, a one-unit increase in this feature is associated with a change of* `w` *in the target,* ***among rows that are identical in every other feature in the model****.*

Every clause in that sentence is load-bearing, and each one corresponds to a way the reading goes wrong.

Lie one: unstandardised coefficients are not comparable

A coefficient of 0.03 on income and 4,500 on "number of children" does not mean children matter more. Income is measured in rupees and children in whole people. The magnitudes are units, not importance.

Fit on standardised features — mean 0, standard deviation 1 — and the coefficients become comparable: each now says how much the target moves per one standard deviation of that feature. That is a meaningful comparison, and it is the only form in which "which feature matters most" can be read off a linear model at all.

```
from sklearn.preprocessing import StandardScaler
Z = StandardScaler().fit_transform(X)
# now coefficients are in target units per standard deviation
```

Lie two: correlated features split the credit arbitrarily

Put `height_cm` and `height_inches` into the same model. They carry the same information. The optimiser has infinitely many ways to divide the effect between them and will pick one for reasons that have nothing to do with the world — a rounding difference, the order of the columns, a random seed.

Refit on a slightly different sample and the split changes completely, sometimes flipping a sign. The *predictions* remain stable; the *coefficients* do not. This is multicollinearity and it is the reason coefficient-based feature importance is unreliable on real data, where features are correlated by default.

A quick diagnostic is the variance inflation factor, but there is a cheaper one: refit on five bootstrap samples and look at whether any coefficient changes sign. If it does, do not tell anyone what that feature means.

Lie three: the effect depends on what else is in the model

"Holding everything else fixed" is doing enormous work. Add a new column and existing coefficients move — sometimes a lot, sometimes reversing sign.

The cleanest illustration is Simpson's paradox. Across a whole university, the acceptance rate for women may be lower than for men. Split by department and women may have the higher rate in every single department, because they applied disproportionately to the departments that reject most applicants. Both statements are arithmetically true. A model without a department column reports the first; a model with it reports the second. Neither is the coefficient "lying" in a technical sense; both are answering the question you asked, and the questions are different.

Which model is right depends on what you want to know, and that is a subject-matter question no algorithm can settle for you.

Lie four: statistical significance is not importance

With 5 million rows, effects of no practical consequence come out significant. With 300 rows, effects that matter enormously come out insignificant. A p-value is a statement about how surprising the data would be under a null hypothesis; it is not a measure of size and does not become one at any sample size.

Report the effect size and an interval. "Each extra year of tenure is associated with 1.2 fewer support tickets per year, 95% interval 0.4 to 2.0" is a sentence that supports a decision. "Tenure is significant at $p < 0.001$" is not.

Note that scikit-learn deliberately does not give you p-values. If you want them, `statsmodels` does, and its `.summary()` output is the standard table for this. It is free, and reading its output is a genuinely useful skill for anyone who will work near a statistician.

What coefficients are actually good for

Having spent four sections on limits, the positive case. Coefficients are excellent for:

- **Sanity checking.** A negative coefficient on income in a creditworthiness model is a bug alarm. Look at the sign of every coefficient you have a prior about; this catches more errors than any test.
- **Finding leakage.** One coefficient far larger than every other, on a feature you did not expect, is usually a column recorded after the outcome.
- **Required explanations.** Adverse action notices, in several jurisdictions, need a reason. A linear model can produce one per applicant, mechanically and consistently.
- **Communication.** "Every additional late payment adds about 12 points of risk" is a sentence that survives being repeated by someone who did not build the model. That has value that AUC does not.

Just write the sentence with its clauses intact, and refuse to let anyone drop them.

BEFORE YOU MOVE ON

A team adds a 'region' feature to a pricing model and the coefficient on 'floor area' drops by 40%. They ask which version is correct. What is the right response?

A. The second, because more features always means less omitted-variable bias and a more accurate coefficient.

B. The first, because adding a correlated feature has split the credit and understated floor area's true effect.

C. Neither is 'correct' in isolation: the coefficients answer different questions — with and without holding region fixed — and which you want depends on whether region is a confounder or part of the effect you are trying to measure.

D. Whichever model has the better validation score, since predictive accuracy settles which coefficient set is more trustworthy.

17 · 9 min

Regularisation: paying for large coefficients

THE ONE THING TO KEEP

Ridge and lasso add a price on coefficient size to the loss; lasso reaches exactly zero because the gradient of the absolute value stays constant as the coefficient shrinks, while ridge's push fades away near zero.

The idea in one sentence

Add the size of the coefficients to the loss, so the optimiser has to justify every unit of coefficient it spends with a matching reduction in error.

```
ordinary:  minimise  sum of squared errors
ridge:     minimise  sum of squared errors + alpha * sum of w^2
lasso:     minimise  sum of squared errors + alpha * sum of |w|
```

That is the whole mechanism. `alpha` is the price. At `alpha = 0` you are back to ordinary least squares. As `alpha` rises, coefficients shrink towards zero and the model gets simpler whether it likes it or not.

Why shrinking helps

A model overfits by using large, opposing coefficients to thread through individual training points. Two correlated features with `+900` and `−880` can produce wild predictions from small input changes — a tiny shift in either feature moves the output enormously. That is a model with high variance: refit it on a different sample and it makes different predictions.

Making large coefficients expensive removes that option. The model must explain the data with modest weights or not at all. You give up a little training accuracy and buy a large reduction in how much the model changes between samples. That trade is worthwhile whenever you have many features relative to rows, and it is why ridge is a sensible default rather than a rescue.

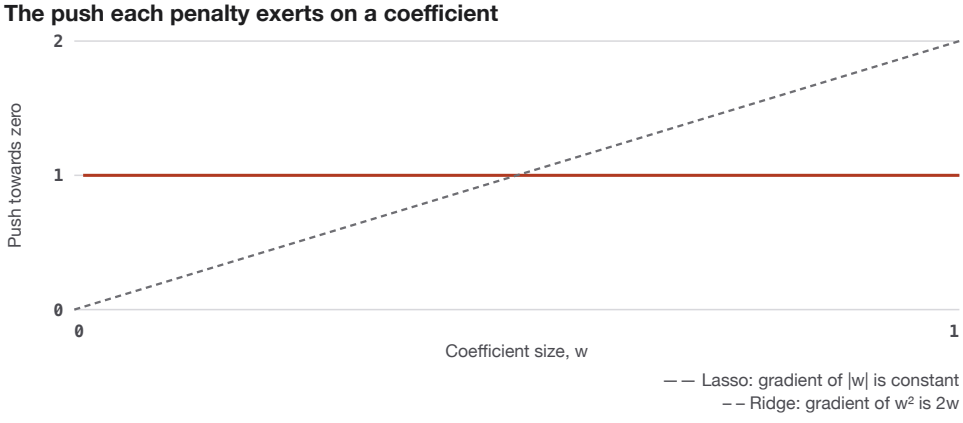
The difference between L2 and L1, mechanically

Everyone learns that lasso sets coefficients to exactly zero and ridge does not. Fewer learn why, and the reason is simple enough to hold.

Consider the penalty's own gradient — the push it exerts on a coefficient.

- Ridge penalises `w^2`. Its gradient is `2w`. As `w` approaches zero, the push approaches zero. The penalty loses interest in small coefficients, so they shrink and shrink and never quite arrive.
- Lasso penalises `|w|`. Its gradient is a constant (the sign of `w`) regardless of size. A coefficient of 0.001 gets exactly the same push toward zero as a coefficient of 50. That constant push is strong enough to drive small coefficients to exactly zero and hold them there.

So lasso performs feature selection as a side effect of its penalty shape, not as a separate step. With `alpha` tuned, a lasso on 200 features may keep 23 and zero the rest.



Ridge's push is 2w, so it fades as the coefficient shrinks and a small coefficient is never quite driven to zero. Lasso's push is the same at 0.001 as at 50 — a constant — which is what carries small coefficients all the way to zero and holds them there.

Which to use

- **Ridge** when features are correlated and you want stability, or when you believe many features each contribute a little. Ridge handles correlated groups gracefully: it shares the coefficient across them rather than picking one.

- **Lasso** when you believe most features are irrelevant and want a short list. Its weakness is exactly ridge's strength — given five correlated features, lasso keeps one essentially at random and zeroes the other four, which makes the selected list unstable across samples.
- **Elastic net** — both penalties, mixed with a ratio parameter. It is the answer when you want lasso's sparsity but have correlated groups, because it tends to keep or drop the whole group together.

```
from sklearn.linear_model import RidgeCV, LassoCV, ElasticNetCV
import numpy as np
ridge = RidgeCV(alphas=np.logspace(-3, 3, 13)).fit(X_tr, y_tr)
ridge.alpha_        # the chosen penalty
```

The `CV` variants choose `alpha` by cross-validation, which is what you want. Searching alpha over a log scale, not a linear one, is essential — the useful range spans several orders of magnitude and a linear grid wastes almost all its points.

Two things that will bite you

Scale the features first, always. The penalty is applied to the coefficient values, and a feature measured in rupees has a small coefficient purely because of its units. Without scaling, ridge penalises features by their unit of measurement rather than their contribution — the income coefficient is naturally tiny so it escapes the penalty, while the coefficient on a 0/1 flag is naturally larger and gets crushed. Put a `StandardScaler` before any penalised model and treat it as non-optional.

Do not penalise the intercept. Every serious library already excludes it, but if you write the loss yourself, remember: penalising the intercept means the model is punished for the target having a nonzero mean, which is not a complexity you want to discourage.

The wider point

Regularisation is not a linear-model technique. It is a general principle — constrain the model's freedom so it cannot fit noise — and every family has its

version. Trees limit depth and require minimum samples per leaf. Neural networks use weight decay (which is exactly L2), dropout, and early stopping. Boosting uses a learning rate and subsampling.

All of them are the same trade in different clothes: accept worse training performance in exchange for a model that changes less when the data changes. Once you see that, `max_depth`, `alpha`, `dropout` and `n_estimators` stop being a list of unrelated knobs and become one knob with several handles.

BEFORE YOU MOVE ON

A dataset has five nearly identical sensor readings, all genuinely predictive. Lasso keeps one and zeroes the other four; ridge keeps all five with smaller coefficients. Which behaviour is a problem, and why?

A. Ridge's, because keeping five collinear features inflates the standard errors and makes the model unstable.

B. Lasso's, if you plan to read the selected features as 'the important ones' — the choice among equally good correlated features is arbitrary and will differ on a resample.

C. Lasso's, because dropping four predictive sensors necessarily reduces predictive accuracy.

D. Neither, since both minimise a convex objective and therefore reach the same unique solution.

18 · 8 min

Making a straight-line model bend

THE ONE THING TO KEEP

A linear model bends by manufacturing columns — squares, interactions, bins, splines — and the interaction terms matter most, because finding those automatically is precisely what tree models do that linear models cannot.

Linear in the parameters, not in the data

The word "linear" in linear regression refers to the parameters, not the features. `y = w1*x + w2*x^2 + b` is still a linear model — it is linear in `w1` and `w2`, which is what the normal equation and the convexity require. The fact that it draws a parabola is irrelevant to the fitting machinery.

This is more useful than it sounds, because it means you can give a linear model any shape you like by manufacturing columns.

Polynomial features

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import Ridge

model = make_pipeline(
    PolynomialFeatures(degree=2, include_bias=False),
    StandardScaler(),
    Ridge(alpha=1.0),
)
```

With three input features and degree 2, `PolynomialFeatures` produces `a, b, c, a^2, ab, ac, b^2, bc, c^2` — nine columns from three. That growth is the catch. At degree 3 with 20 features you get 1,770 columns, and the mod-

el has more freedom than data to constrain it. Degree 2 with a ridge penalty is a reasonable thing to try; degree 5 is almost never one.

And polynomials misbehave at the edges. A degree-4 fit that looks sensible through the middle of your data will shoot off vertically just past the last point. If you take one habit from this lesson: plot the fitted curve over a range wider than your data before trusting it.

Interactions are usually the point

The cross terms matter more than the squares. `ab` lets the effect of `a` depend on the value of `b`, which is how most real relationships work.

A discount raises conversion — but much more for price-sensitive customers than for corporate accounts. A linear model with `discount` and `is_corporate` as separate columns says the discount has one fixed effect for everyone. Add `discount * is_corporate` and it can say something different for each group. That single manufactured column is often worth more than a change of algorithm.

This is also the honest explanation of why gradient boosting beats linear models so often on tabular data. A tree finds interactions automatically by splitting on `b` and then on `a` inside that branch. A linear model finds them only if you thought of them and wrote the column. The gap is not that trees are cleverer; it is that they do this one thing without being asked.

Binning, and what it costs

Cut a continuous feature into ranges and one-hot encode them. Age becomes `18–25, 26–35, 36–50, 51+`, and now the model can fit any step-shaped relationship with no assumption about smoothness.

This is genuinely useful when the relationship has a real threshold — a policy that changes at 60, a tariff band, a legal age. It is wasteful otherwise: you have thrown away all the information inside each bin, so 26 and 35 become indistinguishable, and you have added a hard discontinuity at every boundary, so 35 and 36 differ sharply for no reason in the world.

A rule that holds up: bin when the boundary is real, otherwise use a spline.

Splines: the option most people skip

A spline fits separate low-degree polynomials on segments of the feature's range, joined so the result is smooth. You get flexible curvature without the wild edge behaviour of a high-degree polynomial, because no single polynomial has to cover the whole range.

```
from sklearn.preprocessing import SplineTransformer
model = make_pipeline(
    SplineTransformer(n_knots=5, degree=3),
    Ridge(alpha=1.0),
)
```

Five knots and cubic segments is a good default. This is what a statistician reaches for when a relationship is clearly curved, and it is much better behaved than `PolynomialFeatures(degree=5)`. It is in scikit-learn, it is one line, and hardly anyone uses it.

Transform the target too

If the target is strongly right-skewed — income, house prices, time-to-event, almost any monetary quantity — fitting `log(y)` instead of `y` often does more than any feature transformation.

It makes the errors multiplicative rather than additive, which usually matches reality better: being wrong by ₹50,000 on a ₹2 crore property is a small error, and being wrong by ₹50,000 on a ₹3 lakh property is a disaster. Squared error on the raw target treats them identically. On the log scale it does not.

Two cautions. Use `np.log1p` if the target contains zeros. And remember that back-transforming with `np.exp` gives you the *median* of the predicted distribution, not the mean — so if you need an unbiased total across many rows, the naive exponential will underestimate it. `TransformedTargetRegressor` handles the mechanics; the bias is yours to think about.

When to stop

If you find yourself adding degree-4 polynomials, six interaction terms and manual bins, you have hand-built a worse gradient boosting model. Switch. The point of these techniques is to get a fast, explainable model closer to the truth, not to win a fight with a tree.

BEFORE YOU MOVE ON

A conversion model has 'discount_pct' and 'is_enterprise' as separate features and fits poorly. A colleague suggests raising the polynomial degree to 3. What would more likely fix it, and why?

A. Adding the product discount_pct * is_enterprise, because the model currently forces one fixed discount effect on both customer types.

B. Binning discount_pct into deciles, because the relationship with conversion is probably non-monotonic.

C. Raising the degree to 3 as suggested, since higher degree captures more complex relationships in general.

D. Log-transforming discount_pct, because monetary and percentage features are usually skewed.

19 · 8 min

More than two classes, and what changes

THE ONE THING TO KEEP

Softmax makes classes compete for a fixed probability budget, which is correct when exactly one label is true and wrong for multi-label problems, where independent sigmoids with their own thresholds are needed.

Two strategies

Logistic regression is built for two classes. Real problems have four, or fifty, or three thousand. There are two ways to get there and they behave differently.

One-vs-rest. Train one binary classifier per class: "is it A or not", "is it B or not", and so on. To predict, run all of them and take the highest score. With 50 classes you train 50 models.

Simple and it works with any binary classifier. Two weaknesses: each classifier sees a badly imbalanced problem (one class against 49), and the scores come from independently fitted models, so they are on different scales and do not sum to 1. Comparing them is a bit of a fudge.

Multinomial (softmax). One model, with one weight vector per class, all fitted together against a single loss. For each row, compute a score per class and convert the whole vector to probabilities at once:

```
p_k = exp(s_k) / sum over all j of exp(s_j)
```

That is softmax. It exponentiates to make everything positive, then divides by the total so the probabilities sum to exactly 1. The classes compete: raising one score lowers every other probability, which is the correct behaviour when exactly one class is true.

Modern scikit-learn uses multinomial by default for `LogisticRegression`, and it is the right default. Neural network classifiers always use softmax on the output layer with cross-entropy loss, which is the same construction.

Two numerical details that matter in practice

Softmax overflows if you write it naively. `exp(1000)` is infinity in floating point. Every real implementation subtracts the maximum score before exponentiating, which does not change the result mathematically — the constant cancels between numerator and denominator — but keeps every exponent at or below zero.

```
def softmax(s):
    s = s - s.max()
    e = np.exp(s)
    return e / e.sum()
```

If you ever write this yourself, that one line is the difference between working code and `NaN`.

Softmax and sigmoid are the same thing at two classes. Softmax with two classes reduces exactly to the sigmoid. Nothing new was introduced; the binary case was the special case all along.

Multi-class is not multi-label

A distinction worth being firm about, because mixing them up produces a model that cannot express the truth.

Multi-class: exactly one label is correct. A photo is a cat, or a dog, or a horse. Softmax, and the probabilities sum to 1.

Multi-label: any number of labels can be correct at once. An article is about politics *and* economics *and* India. Here softmax is wrong — it forces the labels to compete for a fixed budget of probability, so a genuinely three-topic article has to divide its confidence three ways. Use independent sigmoids, one per label, each with its own binary cross-entropy and its own threshold. In scikit-learn that is `MultiOutputClassifier` or `OneVsRestClassifier`.

Ask of any labelling task: can two labels be true at once? The answer determines the output layer, and getting it wrong caps your accuracy in a way no amount of tuning recovers.

Metrics need an averaging decision

Precision and recall are defined per class, so reporting one number for a multi-class problem requires choosing how to combine them, and the choices disagree sharply.

- **Macro** — compute the metric per class, then take the plain mean. Every class counts equally, so a rare class with 12 examples influences the score as much as a common one with 40,000.
- **Weighted** — the mean weighted by class frequency. Dominated by the common classes.
- **Micro** — pool all the decisions and compute once. For single-label multi-class this equals accuracy.

These can differ by 30 points on an imbalanced problem, and quoting one without saying which is a common way to mislead without lying. If the rare classes are the point — rare diseases, rare failure modes — use macro and say so.

```
from sklearn.metrics import classification_report
print(classification_report(y_te, pred, digits=3))
```

That report gives per-class precision, recall and support, plus all three averages. Read the per-class rows before the summary line. A model with 0.91 weighted F1 that scores 0.0 on your two rarest classes is a specific, common and completely invisible failure if you only look at the average.

When there are thousands of classes

At very high class counts — product catalogues, word vocabularies — softmax gets expensive, because every prediction computes a score for every class. The workarounds are hierarchical softmax and sampled softmax, and in retrieval

settings the problem is usually reframed as nearest-neighbour search over embeddings, which module 7 covers. Worth knowing the shape of the problem exists; you will not meet it on tabular data.

BEFORE YOU MOVE ON

A news classifier tags articles with any of eight topics, and articles frequently belong to two or three. The team trains a softmax model over the eight topics and finds it never confidently assigns more than one topic. What is the cause?

A. The training data is imbalanced across topics, so the model has collapsed onto the most common one.

B. Eight classes is too many for one softmax layer, and the model needs a hierarchical output.

C. Softmax forces the eight probabilities to sum to 1, so confidence in one topic mechanically reduces confidence in the others; multi-label needs independent sigmoids.

D. The decision threshold is fixed at the argmax, and lowering it to 0.3 would allow several topics through.

CASE STUDY · MODULE 2

The coefficient that changed sign in Muzaffarpur

A microfinance lender in Muzaffarpur with 41,000 active borrowers, a two-person analytics team, and forty field officers who have used the same paper scorecard for nine years.

Saathi Sahyog lends between twelve and sixty thousand rupees to women's self-help groups across four districts of north Bihar. Repayment is weekly and collected in person. The decision they wanted to support is the one a field officer makes on a doorstep: recommend this group's loan for approval, or refer it for a second visit.

The existing instrument is a paper scorecard with eleven questions and a total out of one hundred. It was built in 2017 by a consultant nobody can now reach, and everyone in the field trusts it — partly because they can see it work and partly because a number they can compute themselves is a number they can argue with.

The analytics team, Farhan and Kavya, fitted a logistic regression on 38,000 completed loans with fourteen features. The target was ninety days past due within the loan term. They fitted it in an afternoon and spent three weeks on what it said.

Two features were the problem: `monthly_household_income`, collected by asking, and `weekly_instalment_as_share_of_income`, computed from that same declared income and the loan amount. They are close to mechanically related. In the fitted model, income carried a coefficient of +1.9 on the log-odds of default and the instalment share carried −2.4. Read literally, that says richer households default more and larger instalments relative to income make default less likely. Both readings are absurd, and both are what the numbers said.

Kavya ran the diagnostic from the coefficients lesson: refit on five bootstrap samples. Income's coefficient changed sign in two of them. The predictions barely moved across the five refits; the coefficients moved a great deal. That is

multicollinearity behaving exactly as documented, and it is invisible in any accuracy score.

Now the decision, and it had a genuine cost on each side.

Option one: drop declared income. The instalment share carries most of the information, the remaining coefficients stabilise, and the model becomes explainable. The cost is that income is question three on the paper scorecard. Removing it from the model means the field officers' instrument and the office's instrument now disagree about what matters, and forty people who have collected that number every week for nine years are told it is no longer used. Farhan's estimate, from two conversations in the field, was that a proportion of officers would conclude the office did not understand their job and would stop trusting the output entirely.

Option two: keep both columns and add a ridge penalty. Ridge shares the coefficient across correlated features rather than picking one arbitrarily, and the signs settle. The cost is that the individual coefficients no longer support the sentence the lender needs. A rejection notice under Reserve Bank of India expectations has to carry a reason, and "your score is low because of the combined weighting of two related fields" is not a reason a borrower can act on.

Option three: ship the boosted model instead. They had one. It scored 0.79 AUC against the linear model's 0.74 and the paper scorecard's 0.68. It also has no coefficient to read, and the field officer cannot compute it on a doorstep with no signal, which happens on roughly a fifth of visits.

Five points of AUC against forty people's willingness to use the thing. Nobody in the room thought that was an obvious trade.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

They kept both columns, standardised the features, and used ridge with alpha chosen by cross-validation on a log scale. Then they did the part that mattered: they reported effects for grouped features rather than single ones. Income and instalment share are presented together as one "affordability" block with a combined contribution, and the rejection reason names the block.

The paper scorecard was left alone. Question three still asks about income and the field officers still compute their total, which the model now takes as one of its fourteen inputs. The office score and the doorstep score are two instruments that agree about what is being measured, and the officers were told in a district meeting how the two relate.

The boosted model was fitted, scored, written down and not deployed. The note recording that decision gives the reason as offline availability and the reason requirement, not accuracy, so that whoever revisits it in three years knows what would have to change for the answer to change.

The predictions were stable across the five bootstrap refits while the coefficients were not. Why is that not a contradiction?

Because the two correlated columns carry almost the same information, so many different splits of the weight between them produce nearly the same weighted sum. The optimiser is free to choose any of those splits and picks one for reasons that have nothing to do with the world. The prediction depends on the sum; the interpretation depends on the split. A model can therefore be reliable to predict with and unreliable to read.

Why was the ridge penalty applied only after standardising the features?

Because the penalty is applied to the coefficient values, and a coefficient's size depends on the unit of its feature. Income in rupees has a naturally tiny coefficient and would escape the penalty almost entirely, while a nought-or-one flag has a naturally larger one and would be crushed. Standardising puts every feature on the same scale so the penalty falls on contribution rather than on unit of measurement.

The boosted model was five points of AUC better and was not deployed. Was that the wrong call?

It is defensible, and the case for it is not accuracy. The instrument has to run on a doorstep with no connectivity and has to produce a stated reason for a refusal, and a five-point gain that the people carrying the device will not use is not a five-point gain. What makes the decision sound is that the reasoning was recorded — the constraint, not the score, is what would have to change.

MODULE 2 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 Training a linear model, the loss prints as NaN within a dozen steps. What is the first thing to change?

A. Increase the number of iterations so the optimiser has more time to settle down.

B. Switch from mini-batch to full-batch gradient descent to remove the gradient noise.

C. Lower the learning rate by a factor of ten.

D. Add a ridge penalty so that the coefficients cannot grow without bound.

2 Linear regression with squared error has an exact one-step solution. Why do courses teach gradient descent anyway?

A. Gradient descent reaches a better minimum than the normal equation does on real data.

B. The normal equation is only correct when the features are completely uncorrelated.

C. The normal equation cannot handle a model that includes an intercept term.

D. Most losses and models have no such formula, and the walk still works.

3 Lasso drives some coefficients to exactly zero and ridge does not. What is the mechanism?

A. Ridge is fitted iteratively while lasso is solved exactly, so lasso can reach the boundary.

B. The absolute-value penalty pushes just as hard however small the coefficient is.

C. Lasso removes correlated features first, and a removed feature has a coefficient of zero.

D. Ridge also penalises the intercept, which stops any coefficient from reaching zero.

4 In a logistic model a coefficient of 0.7 on 'has an overdraft' exponentiates to 2.0. A manager reads this as doubling the chance of default. What is the correction?

A. It doubles the odds, so 0.10 becomes 0.18 and 0.40 becomes 0.57.

B. It doubles the log-odds, so the predicted probability rises by seventy percentage points.

C. It doubles the probability, but only for rows where every other feature sits at its mean.

D. It doubles the probability, provided the model's probabilities have been calibrated first.

5 A logistic regression on age, from 18 to 80, and annual income, from 200,000 to 5,000,000, hits its iteration limit without converging. Which change addresses the cause rather than the symptom?

A. Raise the iteration limit until the convergence warning stops appearing in the output.

B. Drop income, because its scale is dominating the fit and destabilising the optimiser.

C. Standardise both features before fitting.

D. Switch to a solver that reports no convergence warning for this kind of problem.

6 A discount raises conversion much more for price-sensitive customers than for corporate accounts. A linear model has 'discount' and 'is_corporate' as separate columns. What has to be added for the model to express this?

A. A squared term on discount, so that the effect is allowed to curve rather than stay flat.

B. The product of discount and is_corporate, as a manufactured column.

C. A higher-degree polynomial on both columns, together with a ridge penalty to control it.

D. Separate models per segment, because a linear model cannot express this relationship at all.

MODULE 3

The data is the job

The part of machine learning that takes most of the time and wins most of the accuracy. Numbers, categories, missing values, dates and text turned into columns a model can use — and the two failures that quietly destroy more projects than any modelling mistake: leakage, and a sample that does not represent what you will predict on.

> BY THE END OF THIS MODULE YOU CAN
>
> Take a raw table with mixed types, missing values, high-cardinality categories, dates and free text, and produce a fitted preprocessing pipeline that leaks nothing from the held-out data, handles categories unseen at training time, and whose every transformation you can justify by what the downstream model requires

---

20 · 9 min

Features beat models

> THE ONE THING TO KEEP
>
> Ask of every feature: at the moment I need the prediction, does this value already exist with this value?

A feature is a number you decided to hand over

The model sees only what you give it. Not the customer, not the flat, not the sentence — a row of numbers you chose to compute. That choice is called fea-

ture engineering, and on ordinary table-shaped data it moves results more than the algorithm does.

A concrete case. You want to predict whether a student finishes a free online course. Your database has:

```
user_id, signup_timestamp, country, device, lesson_events[]
```

Handed over raw, `signup_timestamp` is a number like 1772841600. The model can only ask questions like "is it above 1772000000", which means "did they sign up after a particular Tuesday". Almost useless.

Now derive features from the same column:

- `hours_between_signup_and_first_lesson` — hesitation predicts dropout.
- `sessions_in_first_week` — the strongest single predictor in most course data.
- `signed_up_on_weekend` — different intent.
- `local_hour_of_signup` — converted to the student's timezone, not the server's.

Same raw column. One version carries almost no signal; the other carries most of it. No change of algorithm does that.

One column, two feature sets

Nothing about the algorithm changed. The raw timestamp lets a model ask only whether the number is above some cut-off; the derived columns say what a person would actually look at.

Make the number mean something

Models do not know context, so put the context in the number.

- Predicting whether a flat will rent quickly? `price` is weak. `price ÷ median price of flats within 2 km` is strong, because the model no longer has to learn every neighbourhood's price level from scratch.
- Predicting delivery delay? Raw latitude and longitude force the model to carve the map into boxes. `distance_to_depot_km` and `is_inside_ring_road` say it directly.
- Predicting churn? `payments_made` is confounded by tenure. `payments_made ÷ months_subscribed` is not.

Each of these is arithmetic a nine-year-old could do. Each is usually worth more than swapping in a fancier model.

The honest limit of this claim

This is true for tabular data: rows and columns, the sort of thing that starts life in a spreadsheet or a SQL table. It is not true for images, audio and text.

For thirty years, computer vision was largely the craft of hand-designing features — edge detectors, corner detectors, gradient histograms. Deep learning won that field by learning the features from raw pixels instead. Nobody hand-engineers features for image classification now, and you should not start.

So the rule with its boundary: **on tabular data, spend your time on features; on perceptual data, let the network learn them.** Lesson 9 shows why those are the same statement viewed from two sides.

Leakage: the feature that already knows

One failure mode deserves its own section, because it produces spectacular results and is invisible in every standard check.

Leakage is a feature that contains information which will not exist at prediction time. The model uses it, the score is wonderful, and the system fails on the first real request.

Examples that have all happened:

- A hospital mortality model with `discharge_destination` among its features. One value is "mortuary".
- A loan-default model with `collections_agency_assigned`. Agencies are assigned after default.
- A support-escalation model with `number_of_agent_replies`. Replies accumulate because the ticket is escalating.
- A conversion model where the price column was written back after checkout, so unconverted sessions had a null price.

A correct train/test split does not catch any of these. The split protects you against memorising rows. It does nothing about a column that arrives from the future, because the leak is present on both sides of the split.

The test is a question, and you have to ask it about every column:

> *At the exact moment I need this prediction, would this value already exist, with the value it has here?*

Both halves matter. `number_of_agent_replies` exists at prediction time — it is 0 or 1 — but not with the value in your training table. That is still leakage.

A working order

1. Write down when the prediction has to be made. An actual moment: ticket created, checkout loaded, application submitted.
2. List only the fields that exist at that moment.
3. Derive features that encode what you would look at as a human.
4. Fit the simplest model that can use them, and read the result as a measurement of your features.
5. Only then start comparing algorithms.

Step 4 is the one people skip. A weak score from a simple model on good features tells you something real. A strong score from a complicated model on unexamined features tells you almost nothing.

BEFORE YOU MOVE ON

A team predicts which support tickets will need escalation to a senior engineer. Their strongest feature by importance is `number_of_agent_replies`. They split the data properly and validation performance is excellent. What is the most likely explanation?

A. Replies accumulate because the ticket is already escalating, so that value will not exist when the prediction is needed.
B. The feature is genuinely predictive and should be kept, since the held-out split confirms it.
C. The model is overfitting to one dominant feature and needs regularization.
D. Feature importance scores are unreliable, so the ranking tells them nothing.

21 · 9 min

Scaling, and the transforms that change what a model can see

THE ONE THING TO KEEP

Scaling is required by distance-based, penalised and gradient-fitted models and irrelevant to trees; the log transform is not cosmetic but a claim that ratios matter more than differences.

Which models need scaling, and why

Scaling is not a universal good. It is required by a specific set of models for a specific reason, and applying it blindly wastes time while not applying it where needed silently ruins results.

Needs scaling:

- Anything with a **distance** in it — k-nearest neighbours, k-means, SVMs with RBF kernels. If income runs to millions and age to tens, the distance

between two people is essentially the difference in their incomes; age contributes nothing.

- Anything with a **penalty on coefficients** — ridge, lasso, regularised logistic regression. As module 2 showed, the penalty applies to coefficient size, so it lands on features by unit of measurement rather than by importance.
- Anything fitted by **gradient descent** — neural networks, `SGDClassifier`. Mismatched scales create a long, narrow loss surface that one learning rate cannot navigate.
- **PCA**, because it maximises variance, and a feature in rupees has more variance than one in kilometres for no meaningful reason.

Does not need scaling: decision trees, random forests, and every gradient boosting library. A tree asks "is `income > 450000`", and the answer does not change if you divide the whole column by a thousand. Scaling before XGBoost is harmless and pointless.

The three scalers, and when each is right

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler,
RobustScaler
```

StandardScaler subtracts the mean and divides by the standard deviation. Result has mean 0 and standard deviation 1. The default; use it unless you have a reason not to.

MinMaxScaler maps the observed range to [0, 1]. Useful when a bounded range is required — some neural network inputs, image pixels. Fragile against outliers: one value of 10 million squashes everything else into the bottom 0.1% of the range.

RobustScaler subtracts the median and divides by the interquartile range. Because the median and IQR ignore the tails, extreme values do not move the scaling of the bulk. This is the right default when you know the data has heavy tails and you do not want to remove them.

Log, and what it actually does

Many real quantities are multiplicative rather than additive: income, city population, page views, transaction sizes, time between events. They are strongly right-skewed, with most values small and a long tail of very large ones.

```
import numpy as np
X["log_income"] = np.log1p(X["income"])
```

Use `log1p`, which computes `log(1 + x)` and therefore handles zeros. On negative values the log is undefined and you need a different tool.

What the log changes conceptually is which differences count as equal. On the raw scale, 10,000 to 20,000 and 1,000,000 to 1,010,000 are the same difference. On the log scale, 10,000 to 20,000 and 1,000,000 to 2,000,000 are the same difference. The second framing is usually closer to how the world works: a doubling of income means something similar wherever it starts, while an extra ₹10,000 does not.

That is not a trick to make a histogram prettier. It is a claim about the domain, and it is often the single highest-value transformation you can apply to a linear model.

When you cannot guess the right transform

QuantileTransformer maps a feature to a uniform or normal distribution by rank. It handles any shape, ignores outliers by construction, and is a reasonable choice when you have no theory about the feature.

Its cost is real: it destroys the original units, so coefficients become uninterpretable, and it can turn a genuinely meaningful gap into an ordinary one. A feature where 99% of rows are 0 and 1% are enormous will be flattened into something a model can use but nobody can explain.

PowerTransformer with the Yeo-Johnson method finds a power transformation that makes the distribution as symmetric as possible, and it works with zeros and negatives. It is a good middle option: milder than quantile, more flexible than a fixed log.

Ratios and differences beat raw columns

The transformation that most often wins is not on the list above, because it uses two columns.

- `debt` and `income` separately are weaker than `debt / income`. The ratio is what a credit officer actually looks at.
- `total_spend` and `n_orders` separately are weaker than `average_order_value`.
- `date_of_payment` and `date_due` are weaker than `days_late`.

A tree can approximate a ratio with a staircase of splits, but it takes many splits and a lot of data to do badly what one division does exactly. A linear model cannot do it at all without the column. Sit with the domain for twenty minutes and write down the four or five quantities a knowledgeable person computes in their head; those are usually your best features and they are essentially free.

Do it inside the pipeline

The scaler learns parameters — a mean, a standard deviation, quantile boundaries. Those must be learned from training rows only and then applied unchanged to validation, test, and production data. Fit a `StandardScaler` on the full dataset and you have moved information from the held-out rows into the training process, which inflates your score by a small amount you cannot measure. Two lessons on, we will make that mechanism explicit.

BEFORE YOU MOVE ON

A team standardises every numeric column before training an XGBoost model and reports no change in accuracy. What does this tell them?

A. The scaler was fitted on the wrong subset, so the transformation had no effect.

B. The features were already close to standard normal, so the transformation was near-identity.

C. Boosted trees split on thresholds within a single feature, so any monotonic rescaling produces the same splits and the same predictions.

D. XGBoost internally standardises its inputs, making external scaling redundant.

22 · 9 min

Turning categories into numbers without lying about them

THE ONE THING TO KEEP

One-hot is the safe default because it asserts nothing about order or distance; target encoding is powerful and leaks unless computed out-of-fold and smoothed towards the base rate.

The mistake that gets made first

Given a `city` column, the obvious move is to map cities to integers: Mumbai 0, Delhi 1, Chennai 2, Kolkata 3.

A linear model now believes Chennai is twice Delhi and that Mumbai and Kolkata are three units apart. Those statements are meaningless, but the model cannot know that; it will fit a coefficient as though the ordering were real. This is called label encoding, it is the default in a lot of tutorial code, and it is wrong for any unordered category fed to a linear or distance-based model.

It is *not* wrong for trees, which is the source of much of the confusion. A tree can split "city in {0, 3}" versus "city in {1, 2}" and recover an arbitrary grouping given enough splits. Label encoding for LightGBM is common practice; label encoding for logistic regression is a bug.

One-hot: the safe default

One column per category, 1 in the column for this row's value and 0 elsewhere.

```
from sklearn.preprocessing import OneHotEncoder
enc = OneHotEncoder(handle_unknown="ignore",
sparse_output=True,
                    min_frequency=20)
```

Three arguments carry real weight.

`handle_unknown="ignore"` decides what happens when production sends a category that was not in the training data. Without it, the transform raises and your service returns an error. With it, that row gets all zeros across the block and a prediction still comes out. This is a genuine choice with a downside — silently encoding an unknown city as "none of the above" may be exactly wrong for your problem — but a crash at 2am is worse, and the alternative is at least deliberate.

`min_frequency=20` folds every category with fewer than 20 rows into a single "infrequent" column. This is doing real work: a category with three examples cannot support a reliable coefficient, and giving it one invites the model to memorise those three rows.

`sparse_output=True` matters at scale. One-hot on 5,000 product codes gives 5,000 columns of which one is nonzero per row. Dense, a million rows would be 40 GB. Sparse, it is a few hundred megabytes. Linear models and SGD handle sparse matrices natively; some other estimators do not, and that is often the real reason to prefer another encoding.

Ordinal, where order is real

When the categories genuinely have an order — "never, rarely, sometimes, often, always", or T-shirt sizes, or education level — integers are appropriate, but state the order explicitly rather than letting the encoder guess alphabetically:

```
from sklearn.preprocessing import OrdinalEncoder
OrdinalEncoder(categories=[["never", "rarely", "sometimes",
"often", "always"]])
```

Remember what you have asserted: not just the order, but that the gaps are equal. "Never" to "rarely" gets the same numeric distance as "often" to "always". Sometimes that is fine. When it is not, one-hot instead and let the model learn each level's effect separately.

Target encoding, and the trap inside it

Replace each category with the mean of the target for that category. Mumbai becomes 0.043 because 4.3% of Mumbai rows defaulted.

This is powerful — one column instead of 5,000, and it carries exactly the information the model wants. It also leaks, in a way that produces spectacular validation scores and a model that fails in production.

The mechanism: a category with one row gets encoded as that row's own target. The feature now *contains the answer* for that row. The model learns to trust it completely, and on unseen data where the encoding comes from other rows, that trust is unjustified.

The fixes are standard and non-optional:

1. **Compute the encoding out-of-fold.** For each training row, use the target mean computed from the *other* folds only. scikit-learn's `TargetEncoder` does this internally with cross-fitting.
2. **Smooth towards the global mean.** A category with 4 rows should be pulled most of the way to the overall base rate; one with 40,000 rows should barely move. `smooth="auto"` handles it.

```
from sklearn.preprocessing import TargetEncoder
TargetEncoder(target_type="binary", smooth="auto", cv=5)
```

Even with both, treat a target-encoded model's validation score with a little suspicion, and compare it against the same model with one-hot encoding before believing a large gain.

Counting, which nobody suspects

Replace each category with how often it appears. It uses no target information at all, so it cannot leak, it produces one column, and it is often surprisingly effective — for fraud, a rare merchant is a different kind of thing from a common one, and the count says so directly.

Cheap, safe, and underused. Try it before target encoding.

What to reach for

Under about 15 categories, one-hot and stop thinking about it. Between 15 and a few hundred, one-hot with `min_frequency` set, or count encoding. Above that, count encoding or a cross-fitted target encoder, and for gradient boosting specifically, CatBoost's built-in categorical handling — which is an ordered target encoding done carefully — is frequently the best available option with no work at all.

BEFORE YOU MOVE ON

A team target-encodes a 12,000-level merchant column using the full training set, and cross-validated AUC jumps from 0.81 to 0.97. Production performance matches the 0.81 model. What happened?

A. Merchant identity does not generalise, so the feature was pure noise and the model overfitted it.

B. The encoding was computed before splitting, so each row's encoded value partly contains its own target; within cross-validation the folds inherit that contamination.

C. Cross-validation with 12,000 levels is unstable, so 0.97 is within the noise of the estimate.

D. AUC is inflated by class imbalance, and the target encoding made the imbalance worse.

23 · 8 min

When a column has thousands of levels

THE ONE THING TO KEEP

High-cardinality columns need a fixed-size representation — grouping, counting, hashing or a learned embedding — and any identifier that will not exist for an unseen row is not a feature at all.

The shape of the problem

User ID. Product SKU. Postcode. Merchant name. IP address. URL path. These columns behave differently from `city`, and the difference is not just size.

They follow a long tail: a small number of levels cover most rows, and a very large number of levels appear once or twice. In a typical e-commerce dataset,

the top 200 products might be 60% of orders and the remaining 80,000 products the other 40%, most of them with under five rows each.

That shape breaks the standard advice. One-hot gives you 80,000 columns of which most carry four examples — enough for the model to memorise, not enough for it to learn anything transferable.

Four workable approaches

1. Keep the head, group the tail. Take the categories covering 95% of rows and one-hot them; everything else becomes `other`. Crude, fast, and it captures most of what is there. `min_frequency` in `OneHotEncoder` does exactly this.

2. Count or frequency encoding. One column, no leakage, and for many of these fields the frequency is genuinely the useful signal. A postcode that appears in 40,000 orders and one that appears in three are different in a way that matters.

3. The hashing trick. Apply a hash function to the category string and take it modulo a fixed number of columns, say 2^18.

```
from sklearn.feature_extraction import FeatureHasher
h = FeatureHasher(n_features=2**18, input_type="string")
```

This is fixed-size regardless of how many distinct values arrive, needs no vocabulary stored, handles unseen categories automatically, and streams — you never need the whole dataset in memory. Collisions happen: two categories land in the same column and their effects get mixed. With 2^18 columns and 80,000 categories, collisions are rare enough that the loss is usually small.

The real cost is interpretability. Column 149,203 corresponds to nothing you can name, so you can never explain what the model learned. For ad-click prediction at web scale this has been standard for over a decade. For a credit decision that must be explained, it is unusable.

4. Learned embeddings. Give each category a small vector — 8 to 64 numbers — and learn those numbers as part of training. This is what neural net-

works do with categorical inputs, and it is the technique behind word vectors and recommender systems.

The advantage over one-hot is that similar categories can end up with similar vectors, so the model transfers what it learned about one to another. Two postcodes with similar customers get similar embeddings and share statistical strength. One-hot cannot express similarity at all; every category is equidistant from every other.

The cost is that you need enough data per category to learn the vector, and enough infrastructure to train a network. Module 7 treats embeddings properly.

A rule for the dimension

The widely used heuristic for embedding size is the fourth root of the number of categories, times something between 1 and 3. For 80,000 categories that gives roughly 17 to 50. It is not theory, it is a rule of thumb that people found works, and you should tune around it rather than trust it.

The identifier question

Should `user_id` be a feature at all?

If you will predict for the same users you trained on — a recommender for an existing customer base — then yes, and an embedding per user is the whole design of a recommender system.

If you will predict for users you have never seen — new applicants, new visitors — then no. The model will learn per-user quirks that do not exist for anybody new, your random split will hide this because the same users appear on both sides, and performance will be far worse in production than in testing. This is the single most common cause of the "it worked in the notebook" failure.

The test is simple and worth writing on a sticky note: **will this exact value exist at prediction time for a row I have never seen?** If not, it is not a feature; it is an identifier, and its place is in the index.

What to do with unseen levels

Whatever encoding you choose, production will send you a value that was not in training. Decide now, in code, what happens:

- Hashing handles it for free — the new value hashes somewhere.
- Count encoding can map it to zero or to the minimum observed count.
- Target encoding maps it to the global mean, which is the sensible default.
- One-hot with `handle_unknown="ignore"` gives an all-zero block.

And log it. A rising rate of unseen categories is one of the earliest and clearest signals that your data has drifted, and it is far easier to monitor than the drift in a continuous feature. Module 9 comes back to this.

BEFORE YOU MOVE ON

A churn model includes customer_id as a categorical feature and scores 0.93 AUC on a random split. Deployed on new signups, it scores 0.61. What is the mechanism?

A. New signups have a different churn base rate, so the model's calibration is off on the new population.

B. The random split placed each customer's rows on both sides, so the model learned per-customer behaviour that does not exist for anyone new.

C. customer_id has too many levels for the one-hot encoder, causing collisions that degrade production predictions.

D. The production feature pipeline differs from the training pipeline, so customer_id is encoded inconsistently.

24 · 9 min

Missing values, and the information in the gap

THE ONE THING TO KEEP

Add a missingness indicator before imputing, because the fact that a value is absent is frequently more predictive than the value would have been — and let boosted trees learn their own direction for NaN rather than imputing at all.

Why it is missing changes what to do

Three situations, with consequences that differ enough to be worth the vocabulary.

Missing completely at random. A sensor dropped packets. Nothing about the row determines whether the value is there. Rare in practice, and the only case where dropping rows is genuinely safe.

Missing at random, given what you observed. Income is missing more often for younger applicants, but among applicants of the same age, whether income is recorded has nothing to do with the income itself. Imputation using the other columns is principled here.

Missing not at random. The value is missing *because of what it is*. People with very high or very low incomes decline to state them. Patients who felt fine did not return for the follow-up test. Here no imputation can recover the truth, because the information needed to reconstruct it is exactly what is absent.

The uncomfortable part: you cannot distinguish these from the data. It is a judgement about the process that produced the rows, and it comes from talking to whoever collected them.

Three reasons a value is missing

Missing completely at random	A sensor dropped packets. Nothing about the row decides it. Dropping rows is safe here, and only here. Rare.
Missing at random, given what you saw	Younger applicants skip the income field, but among applicants of one age the gap says nothing about income. Impute from the other columns.
Missing not at random	Very high and very low incomes go unstated. The information needed to fill the gap is exactly what is absent. No imputation recovers it; the indicator carries the signal.

The data cannot tell you which of these you have; whoever collected the rows can. In every case the first move is the same: add the indicator column, then impute — or let boosted trees learn a direction for NaN themselves.

The gap is often the best feature

The practical response to all three cases starts the same way: **add an indicator column** saying whether the value was missing, then impute.

```
from sklearn.impute import SimpleImputer
SimpleImputer(strategy="median", add_indicator=True)
```

That one argument is the highest-value thing in this lesson. In real datasets, missingness is frequently more predictive than the value would have been. A loan applicant who left employer blank, a patient with no recorded blood pressure, a form where the optional field is empty — each carries information about the person and the process, and imputing without the flag throws it away permanently.

Worth being clear-eyed about what that means: your model may be learning from an administrative artefact rather than anything about the world. That can be fine, or it can be a fairness problem if missingness correlates with a protected group, or it can break the moment a form changes. Know which you have.

Choosing an imputation strategy

- **Median** for numeric. Robust to skew, and most real numeric columns are skewed. Mean is fine for symmetric data and dragged around by outliers

otherwise.

- **Most frequent** or an explicit `"missing"` category for categoricals. The explicit category is usually better: it lets the model treat missingness as its own level, which is the categorical version of the indicator column.
- **Constant, out of range** — a value like -999 — works for tree models specifically, because a tree can isolate it with one split. It is nonsense for linear models, which will treat -999 as an enormous negative quantity.
- **KNNImputer** fills from the k most similar rows. More accurate, considerably slower, and it needs scaled features to compute similarity sensibly.
- **IterativeImputer** models each column from the others in turn. Powerful and slow, and easy to over-trust — a well-imputed value is still a guess presented with the same confidence as a measurement.

Let the model handle it, when it can

Some models handle missingness natively, and better than imputation:

```
from sklearn.ensemble import HistGradientBoostingClassifier
HistGradientBoostingClassifier()   # accepts NaN directly
```

At each split, it sends missing values whichever way reduces the loss more, learned separately per split. XGBoost and LightGBM do the same. This is genuinely better than imputing, because the model gets to decide what missing means *in each context* rather than having one substituted value everywhere.

If you are using boosted trees, the correct answer to missing values is often to do nothing.

When to drop

- **A column that is 95% missing** is usually not worth keeping — although keep the indicator, because the 5% who have it may be a specific and interesting group.
- **Rows with a missing target** must be dropped from training. You cannot impute the answer; doing so trains the model on your imputation rule

rather than on reality.

- **Rows missing many features** can be dropped in training if there are few of them, but note that you will still receive such rows in production and must handle them. Dropping in training and crashing in production is a real deployment pattern.

The rule that keeps you honest

The imputer learns something — a median, a mode, a model of the other columns. It must learn it from **training rows only**. Compute the median on the whole dataset and the held-out rows have contributed to the value substituted into training, which is leakage: small, invisible, and enough to make your estimate optimistic.

This is not a hypothetical. It is one of the two or three most common ways a validation score comes out better than production, and the fix is to put the imputer inside the pipeline, which the lesson after next covers in full.

BEFORE YOU MOVE ON

A hospital dataset has 'last cholesterol reading' missing for 40% of patients, mostly those who never had symptoms. The team imputes with the median and drops the indicator. What have they lost?

A. Nothing important, since median imputation preserves the distribution's centre and the model can still use the other features.

B. The signal that the test was never ordered — which encodes a clinician's judgement that the patient was low risk — is now indistinguishable from a genuinely median result.

C. The variance of the cholesterol column, which is now understated because 40% of values are identical.

D. The ability to use the column at all, since 40% missingness exceeds the usual threshold for retaining a feature.

25 · 8 min

Outliers: error, extreme, or the thing you are looking for

THE ONE THING TO KEEP

Decide first whether an extreme value is an error, a genuine rarity, or the target itself, because the same removal step is correct in the first case, harmful in the second, and destroys the problem in the third.

Three different things wearing the same label

Before any technique, a classification, because the right action differs completely.

A data error. Age 214. Temperature -300 Celsius. A price of ₹0 on a completed order. These are impossible values and they came from a bug, a default, or a form. Fix or remove them, and record what you did.

A genuine extreme. A real customer who genuinely spent ₹40 lakh. Real, rare, and it will happen again. Removing it makes your model wrong about a part of the world that exists.

The signal itself. In fraud detection, intrusion detection or equipment failure, the outliers are the positive class. Removing them removes the entire problem. This is not a hypothetical mistake; it is made regularly by people applying an outlier-removal step from a tutorial.

Most of the damage from outlier handling comes from applying the treatment for the first case to data in the second or third.

Finding them

The two standard rules, both crude, both useful:

IQR rule. Anything below `Q1 − 1.5 * IQR` or above `Q3 + 1.5 * IQR`. This is what draws the whiskers on a box plot. It makes no distributional assumption and it is a reasonable first pass.

Z-score. Anything more than 3 standard deviations from the mean. The catch is circular: the standard deviation is itself inflated by the outliers, so a few extreme values raise the threshold and hide themselves. The fix is a modified z-score using the median and the median absolute deviation, which the outliers cannot inflate.

```
import numpy as np
med = np.median(x)
mad = np.median(np.abs(x - med))
modified_z = 0.6745 * (x - med) / mad
```

Both of these are one-dimensional. A row can be perfectly ordinary in every column and still be an impossible combination — a 19-year-old with 30 years of work experience passes every column check. Multivariate detection (isolation forests, Mahalanobis distance) is in module 7; for now, simply knowing that per-column checks miss combinations is enough to stop you trusting them completely.

The four responses

Fix. If you can determine the true value — a decimal point moved, a unit confusion between kg and lbs — fix it. Best outcome, rarely available.

Remove. Only for genuine errors, and only from training. You cannot remove rows from production; whatever you do in training must have a counterpart at serving time.

Clip (winsorise). Cap values at a percentile, typically the 1st and 99th.

```
lo, hi = np.percentile(x_train, [1, 99])
x = np.clip(x, lo, hi)
```

This keeps the row and its information while limiting how much any single value can move the fit. It is the pragmatic default for a numeric feature with a heavy tail feeding a linear model. Compute the percentiles on training data only and store them — this is a learned parameter like any other.

Do nothing, but choose a robust method. Often the best answer. Use MAE or Huber loss instead of MSE. Use `RobustScaler`. Use trees, which are almost entirely insensitive to the magnitude of an extreme value because they only care which side of a threshold it falls on.

Which models care

- **Linear regression with MSE** — very sensitive. One outlier can visibly tilt the line, as we saw with five houses in module 2.
- **Trees and forests** — barely care. An extreme value goes into the right-most leaf and stays there.
- **k-NN and k-means** — sensitive, because both are built on distances and one distant point distorts them.
- **Neural networks** — sensitive, and an extreme input can produce a huge gradient that damages the weights in a single step. Gradient clipping exists partly for this reason.

So "should I remove outliers" has no general answer. It depends on the model you are about to fit, which is why the question belongs after the modelling decision rather than before it.

The professional habit

When you remove or clip anything, count it and write the count down. "Removed 412 rows of 1.2 million where age was above 120 or below 0 — 0.03%." That sentence takes ten seconds and does three things: it lets someone else reproduce your dataset, it makes it obvious if you ever remove 15% of the data by accident, and it stops the quiet process by which an analysis becomes cleaner and cleaner until it no longer describes anything real.

BEFORE YOU MOVE ON

An analyst applies a 3-standard-deviation filter to transaction amounts before training a fraud detector, removing 0.4% of rows. Recall on fraud collapses. Why?

A. Removing 0.4% of rows shrank the training set enough to increase variance in the estimated model.
B. Fraudulent transactions are disproportionately extreme in amount, so the filter preferentially deleted the positive class the model needed to learn.
C. The z-score threshold was computed after scaling, so it removed the wrong rows.
D. Removing rows changed the class balance, so the decision threshold of 0.5 became miscalibrated.

26 · 8 min

Dates, durations, and the trouble with midnight

THE ONE THING TO KEEP

Decompose timestamps into durations and cyclical components, use local time for behavioural features, and shift any rolling window by one period so it cannot include the value you are predicting.

A timestamp is not a feature

Feed a raw timestamp to a model and it becomes a large integer that increases forever. A tree will split it into "before 12 March" and "after", which is memorising your training period rather than learning anything, and every future date falls beyond the largest value it has ever seen.

Timestamps have to be decomposed. The standard set:

```
d = df["created_at"]
df["hour"] = d.dt.hour
df["dayofweek"] = d.dt.dayofweek
df["day"] = d.dt.day
df["month"] = d.dt.month
df["is_weekend"] = d.dt.dayofweek >= 5
df["days_since_signup"] = (d - df["signup_at"]).dt.days
```

The last one is usually the most valuable, and it is the one people forget. **Durations beat dates.** "Days since last purchase", "account age in days", "hours until the deadline" are meaningful for any row at any time. "March 2024" is meaningful only inside your training window.

The midnight problem

Encode hour as 0 to 23 and you have told the model that 23:00 and 00:00 are 23 units apart — the largest possible distance — when they are one hour apart. The same defect appears at December to January, and at Sunday to Monday.

The fix is to place the value on a circle:

```
import numpy as np
df["hour_sin"] = np.sin(2 * np.pi * df["hour"] / 24)
df["hour_cos"] = np.cos(2 * np.pi * df["hour"] / 24)
```

Two columns instead of one, and now 23:00 and 00:00 sit adjacent in the plane. Both columns are needed: sine alone maps 06:00 and 18:00 to the same value, and the cosine is what distinguishes them.

A caveat that is rarely stated. This matters for linear models, distance-based models and neural networks, where continuity is what the model can use. **Trees do not need it** — a tree can split hour at 22 and again at 5 and reconstruct "night" perfectly well. If you are using boosted trees, plain integer hour is fine and the sine-cosine pair is slightly harmful, because it makes a sharp boundary the tree could have found in one split into a curve it must approximate with several.

Time zones, and the bug that survives review

Store everything in UTC and convert once, deliberately, when you need local behaviour.

But note what "9am" means for the model. If your users are spread across time zones and you use UTC hour, then 09:00 UTC is early morning in London, mid-afternoon in Delhi and the middle of the night in California. The behavioural pattern you are trying to capture — people shop in the evening — is smeared across the whole day and the feature stops working.

Use **local time for behavioural features** and UTC for ordering, durations and joins. This is a two-line fix that regularly improves a model more than an afternoon of tuning, and the bug is invisible because both versions run without error.

The calendar is not uniform

Real series are shaped by human calendars, and the calendar you need is regional.

- Public holidays, which differ by country and by state. India's differ by state; the UK's differ between England, Scotland and Northern Ireland.
- Paydays. In many markets demand spikes at month end and the first of the month.
- Festivals that move. Diwali, Eid and Easter shift by weeks against the Gregorian calendar, so "October" does not consistently mean "Diwali month". A feature like `days_to_diwali` works where `month` cannot.
- School terms, financial year ends, tax deadlines.

The free `holidays` package covers over 100 countries and takes one line. Building a `days_until_next_holiday` and `days_since_last_holiday` pair is often worth more than every other date feature combined for retail and logistics.

Lags and rolling windows

For anything with a time axis, the past values of the target are usually your strongest features:

```
df["sales_lag_7"] = df.groupby("store")["sales"].shift(7)
df["sales_roll_28"] = (df.groupby("store")["sales"]
                         .shift(1).rolling(28).mean())
```

Notice the `.shift(1)` before `.rolling(28)`. Without it, the window includes today's sales, which you do not know when you are predicting today's sales. This single missing shift is the most common leakage bug in time series work, it produces a spectacular validation score, and it fails completely in production. If you take one line from this lesson, take that one.

And split by time

Any model with date features must be validated on a period *after* the training period, never on a random sample of rows. Module 4 covers the mechanics; the reason belongs here, because it is a property of the data rather than of the validation technique: a random split lets the model see Tuesday and Thursday while predicting Wednesday, which is not a situation that will ever occur again.

BEFORE YOU MOVE ON

A demand model uses a 28-day rolling mean of sales, computed as `rolling(28).mean()` without shifting. Validation MAPE is 4%; in production it is 19%. What is the mechanism?

A. The rolling window is too short to capture seasonality, so production performance degrades as the season changes.

B. Production data has more missing days, so the rolling mean is computed over fewer observations.

C. The unshifted window includes the current day's sales, so during validation the feature partly contains the answer, which is unavailable at prediction time.

D. The model was validated on a random split rather than a time-based one, so it saw future periods during training.

27 · 9 min

Text as columns: counting words before you embed them

THE ONE THING TO KEEP

TF-IDF over word and bigram counts with a linear model is a strong, fast, fully interpretable text baseline; its ceiling is that it cannot represent word order or word similarity, which is precisely what embeddings add.

The simplest thing that works

Before embeddings and transformers, there is a technique from the 1960s that is still a strong baseline on many text tasks, trains in seconds on a laptop, and is fully interpretable.

Count the words.

```
from sklearn.feature_extraction.text import CountVectorizer
vec = CountVectorizer(min_df=5, max_df=0.8, ngram_range=(1, 2))
X = vec.fit_transform(docs)     # sparse: docs x vocabulary
```

One column per word in the vocabulary; each cell is how many times that word appears in that document. This is the **bag of words**, so called because it discards order entirely: "the dog bit the man" and "the man bit the dog" produce identical rows.

That sounds fatal and mostly is not, for classification. Whether a review is negative, whether an email is spam, whether a ticket is about billing — these are largely determined by which words are present, not their arrangement. Where order matters, `ngram_range=(1, 2)` adds two-word phrases, which recovers "not good" as distinct from "good" and is often the single most useful setting in this vectoriser.

TF-IDF: weighting by how unusual a word is

Raw counts overweight common words. "The" appears everywhere and separates nothing.

TF-IDF multiplies each count by the inverse document frequency — a term that grows as a word appears in fewer documents. Rare words get amplified, ubiquitous ones get crushed towards zero.

```
from sklearn.feature_extraction.text import TfidfVectorizer
vec = TfidfVectorizer(min_df=5, sublinear_tf=True, ngram_range=
(1, 2))
```

`sublinear_tf=True` replaces the raw count with `1 + log(count)`, on the reasoning that a word appearing 40 times is not 40 times as indicative as one appearing once. It usually helps a little, costs nothing, and hardly anyone turns it on.

TF-IDF plus `LogisticRegression` is the classic text pipeline. On a two-class problem with a few thousand labelled documents it is frequently within a couple of points of a fine-tuned transformer, trains in under a second on a CPU,

and gives you a coefficient per word so you can read exactly what it learned. Start here, always. Then measure whether the heavier option is worth its cost.

Character n-grams, which people forget

```
TfidfVectorizer(analyzer="char_wb", ngram_range=(3, 5))
```

This counts sequences of 3 to 5 characters instead of words, and it is the right tool for several situations word-level features handle badly: misspellings, product codes, URLs, names, languages without spaces between words, and code-mixed text like Hinglish where the same word appears in two scripts. It is also robust to the deliberate character substitutions used to evade spam filters, because "v1agra" and "viagra" share most of their character trigrams while sharing no words at all.

For Indian-language text on a Latin keyboard, where spelling is unstandardised and the same word appears five ways, character n-grams frequently beat word features outright.

What preprocessing actually helps

The traditional list is lowercasing, removing punctuation, removing stop words, and stemming. Some of it helps and some is folklore.

- **Lowercasing** — usually helps, halves the vocabulary. Hurts when case carries meaning, as in named entities or angry text.
- **Stop word removal** — much less useful than its reputation. TF-IDF already downweights common words, and `max_df=0.8` removes anything appearing in over 80% of documents. Removing a fixed English stop list can also delete genuine signal: "not" is on most stop lists and is the difference between a positive and negative review.
- **Stemming and lemmatisation** — merges word forms so "running" and "ran" become one. Helps on small datasets where each form is rare; contributes little on large ones. Stemming is crude and fast, lemmatisation is slower and correct.

- **Removing rare words** with `min_df=5` — genuinely important. A word appearing in three documents cannot support a reliable coefficient and inflates the vocabulary enormously.

Test, do not assume. Each of these is one argument, and comparing four pipelines takes ten minutes.

Where this stops working

Be clear about the ceiling. Bag of words cannot tell "the drug caused the reaction" from "the reaction caused the drug". It has no notion that "doctor" and "physician" are related — they are two independent columns as different from each other as either is from "aubergine". And the vocabulary is fixed at training time, so a word that appears only in production is invisible.

Embeddings solve exactly these three problems, and module 7 explains how. But the order matters: build the count-based baseline first, get its number, and then you can say what the embedding bought you. Teams that skip the baseline routinely deploy an expensive model that beat nothing in particular.

BEFORE YOU MOVE ON

A sentiment classifier on product reviews uses TF-IDF with a standard English stop-word list and performs poorly on reviews containing complaints. What is the most likely cause?

A. TF-IDF underweights rare complaint vocabulary, so negative reviews have weak feature values.

B. Negation words like 'not' and 'never' are on standard stop-word lists, so 'not good' and 'good' become indistinguishable.

C. Complaints are longer, and TF-IDF does not normalise for document length.

D. The vectoriser lowercases text, discarding the emphatic capitalisation common in complaints.

28 · 10 min

Leakage: a catalogue of the ways the answer gets in

THE ONE THING TO KEEP

Leakage is any feature whose training value would not be available in that form at prediction time, and the reliable test is to reconstruct a row using only information timestamped before the moment of prediction.

The definition that covers every case

Leakage is any information in your training features that will not be available, in that form, at the moment you have to make a real prediction.

It is the most expensive error in applied machine learning because it does not look like an error. It looks like success. Your score goes up, your cross-validation agrees, your colleagues are impressed, and the model fails the day it meets production.

The signature is worth memorising: **a result substantially better than you expected, arriving with no effort.** Every experienced practitioner has learned to distrust that feeling, and the check takes fifteen minutes.

Target leakage: the column that knows

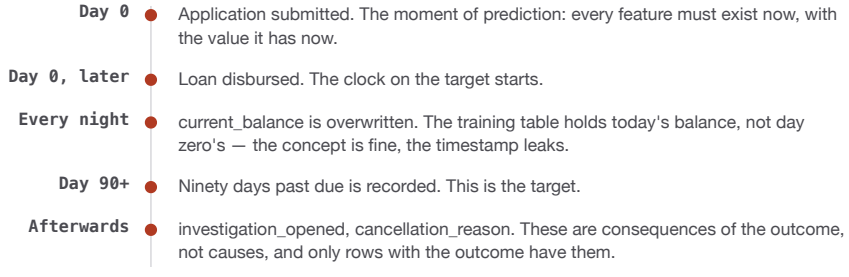
A feature that is a consequence of the target rather than a cause.

- Predicting churn with `cancellation_reason`. Only cancelled customers have one.
- Predicting hospital readmission with `discharge_destination`, when "transferred to another hospital" is recorded as part of the readmission process.
- Predicting fraud with `investigation_opened`. Investigations open after fraud is suspected.

- Predicting default with `current_balance`, refreshed nightly and therefore reflecting today rather than the application date.

The last one is the dangerous shape, because the column is legitimate — balance at application time would be a fine feature. What leaks is the *timestamp* of the value, not the concept. Any feature stored in a table that gets overwritten is suspect for exactly this reason, and it is why event-sourced data with valid-from timestamps is worth so much to a data science team.

Where each column sits against the moment of prediction



Only values fixed at day zero, as they were on day zero, are features. current_balance is a legitimate concept with a leaking timestamp; investigation_opened is a consequence of the target. The test for every column is the same question: what is its value at the moment of prediction, for a row whose outcome is not yet known?

The test: for each feature, ask *what is its value at the moment of prediction, for a row whose outcome is not yet known?* Say it out loud for the ten most important features. It takes fifteen minutes and it is the highest-return quarter hour in this course.

Train-test contamination: fitting on everything

Any transformation that learns from the data must learn from training rows only. The list of things that learn:

- Scalers (mean, standard deviation)
- Imputers (median, mode)
- Target encoders (per-category means)
- Feature selection (which columns to keep)
- PCA (the components)

- Outlier clipping (the percentiles)
- Resampling such as SMOTE (which synthesises rows from neighbours)

Run any of these on the full dataset before splitting and information from the held-out rows has entered the training process. The effect is usually small — one or two points — which is worse than a large effect, because it survives review and quietly makes every comparison you run slightly wrong.

The cleanest defence is structural: put every one of them inside a `Pipeline`, and let cross-validation refit the whole pipeline on each fold. The next lesson is entirely about that.

Group leakage: the same entity on both sides

One patient with 40 visits. One customer with 11 loans. One user with 300 sessions. One document photographed 6 times.

A random row split scatters these across train and test, and the model recognises the entity rather than learning the pattern. The score is inflated by an amount that depends on how much of the signal is entity-specific, which can be most of it.

Split by group, never by row, whenever rows share an entity:

```
from sklearn.model_selection import GroupKFold
cv = GroupKFold(n_splits=5)
cv.split(X, y, groups=df["patient_id"])
```

Temporal leakage: training on the future

A random split lets the model train on November and December while being tested on the intervening period. In production it will only ever have the past.

This matters even without explicit date features, because data has structure in time that a model absorbs implicitly — a product launch, a policy change, a season. Use a time-ordered split for anything with a time axis, which is nearly everything transactional.

Duplicate rows: the quietest one

Exact or near-duplicate rows split across train and test give the model the answer directly. Scraped datasets, merged exports and image sets are full of them, often at 5 to 30% of rows.

Check before splitting:

```
df.duplicated().sum()
df.duplicated(subset=key_columns).sum()
```

For images and text, near-duplicates need fuzzy matching — perceptual hashes for images, shingling or MinHash for text. Several famous published benchmarks have had their results revised downward after duplicate overlap between splits was found, so this is not a beginner's problem.

How to find leakage you have not thought of

Three practical moves, in the order they cost you time:

1. **Look at the top features.** If one feature dominates a model of a hard problem, investigate it specifically. This finds most cases.
2. **Ablate.** Drop the suspect feature and refit. If the score falls from 0.97 to 0.79, you have learned that essentially all the performance came from it, which is worth knowing either way.
3. **Rebuild the row as of the prediction moment.** For one or two rows, reconstruct every feature value using only data with a timestamp before the prediction time. Compare against the values in your training table. Any difference is leakage, and this is the only method that finds the ones you did not think to suspect.

The third is tedious and is what separates a model that works from one that demos.

BEFORE YOU MOVE ON

A loan default model uses `current_account_balance`, pulled from a table that is overwritten nightly. The feature is legitimate in principle. Why is it leaking?

A. Account balance is correlated with the target, and highly correlated features always indicate leakage.

B. The table is overwritten, so the stored value reflects the balance today — after the default — rather than the balance at application time, which is what production would supply.

C. Balance is a continuous feature and needs scaling before it can be used with a penalised model.

D. Balances vary seasonally, so training on them introduces temporal bias that a random split cannot detect.

29 · 8 min

Fit on train, transform everywhere: the contract that keeps you honest

THE ONE THING TO KEEP

Every transformation that learns a parameter must learn it inside each cross-validation fold, which is why a Pipeline changes the score you get rather than merely tidying the code.

Two verbs, and the rule between them

Every preprocessing object in scikit-learn has two methods, and the whole discipline of this module comes down to when you call which.

- `fit` learns parameters from data. A mean. A median. A vocabulary. A set of PCA components.
- `transform` applies those learned parameters to data.

The rule: **`fit` sees training data only. `transform` is applied to everything.**

```
sc = StandardScaler()
X_train_s = sc.fit_transform(X_train)   # learns AND applies
X_test_s  = sc.transform(X_test)        # applies only
```

Calling `fit_transform` on the test set is the error. It computes a new mean from the test rows, so training and test are now on different scales, and the test rows have influenced their own transformation.

Why doing it by hand fails

It is not that manual preprocessing is impossible. It is that it fails under cross-validation, and cross-validation is where you will spend most of your time.

With 5-fold cross-validation, the training portion changes five times. The scaler's mean must be recomputed five times, from five different subsets. Do the scaling once before the loop and every fold's "held-out" portion contributed to the scaling — quietly, in all five folds, with no error and no warning.

`Pipeline` exists to make this structurally impossible:

```
from sklearn.pipeline import Pipeline
pipe = Pipeline([("sc", StandardScaler()), ("clf",
LogisticRegression())])
cross_val_score(pipe, X, y, cv=5)     # refits the scaler in-
side each fold
```

That is not a style preference. It changes the number you get.

ColumnTransformer, because real tables are mixed

Different columns need different treatment, and `ColumnTransformer` routes them:

```
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer

prep = ColumnTransformer([
    ("num", Pipeline([
        ("imp", SimpleImputer(strategy="median",
add_indicator=True)),
        ("sc", StandardScaler()),
    ]), numeric_cols),
    ("cat", Pipeline([
        ("imp", SimpleImputer(strategy="constant",
fill_value="missing")),
        ("oh", OneHotEncoder(handle_unknown="ignore", min_fre-
quency=20)),
    ]), categorical_cols),
], remainder="drop")
```

`remainder="drop"` is worth setting explicitly. The default drops unlisted columns, which is usually right, but writing it down means the next person can see that the omission was intentional rather than an oversight.

One caution: selecting columns by dtype (`selector(dtype_include="number")`) is convenient and fragile. A column that is numeric in training and arrives as a string in production — because one row had "N/A" in it — silently routes down the wrong branch. Explicit column lists are more typing and fewer 3am incidents.

Writing your own step

Anything you can express as a function of the columns can become a pipeline step:

```
from sklearn.preprocessing import FunctionTransformer
log_step = FunctionTransformer(np.log1p,
feature_names_out="one-to-one")
```

`FunctionTransformer` is for stateless transformations — ones that learn nothing. If your step needs to remember something from training (a set of clip boundaries, a category list), write a small class with `fit` and `transform`.

Inherit from `BaseEstimator` and `TransformerMixin` and it will work everywhere in scikit-learn, including inside a grid search.

The payoff at deployment

A fitted pipeline is one object containing every learned parameter — the medians, the category lists, the scaler's means, the model's coefficients.

```
import joblib
joblib.dump(pipe, "model.joblib")
# in the service
pipe = joblib.load("model.joblib")
pipe.predict_proba(new_rows_dataframe)
```

One artefact, one `predict` call, and the serving code cannot apply a different median from the training code because there is only one copy of it. This eliminates an entire category of production bug — training-serving skew — by construction rather than by discipline, which is the only way it stays eliminated after you leave the team.

Two cautions on saved model files. Version pinning matters: a model saved under scikit-learn 1.3 may fail to load under 1.6, so record the version alongside the file. And never load a model file from a source you do not trust — restoring one of these files can execute arbitrary code, so a model downloaded from the internet should be treated exactly like a downloaded executable.

Inspecting what came out

After one-hot encoding you have no idea which column is which. Ask:

```
pipe[:-1].get_feature_names_out()[:20]
```

This is how you connect a coefficient or an importance back to a human-readable name, and it is the difference between "feature 43 is important" and "applications from unverified accounts are important".

The same call is your best defence against a silent column-order change. If an upstream table gains a column, a pipeline built from explicit lists will simply ignore it, while one built from positional indexing will shift every downstream feature by one and keep running. Print the first twenty names into your training log on every run; a diff between two runs then makes the change visible the day it happens rather than the week the metrics drift.

BEFORE YOU MOVE ON

A team runs SMOTE oversampling on the full training set, then performs 5-fold cross-validation on the resampled data. Cross-validated recall is excellent; production recall is poor. What is the structural error?

A. SMOTE should be applied to the test set as well, so that train and test have matching class distributions.

B. SMOTE synthesises rows before splitting, so synthetic points derived from a fold's training rows appear in that fold's validation portion — the model is evaluated on near-copies of what it trained on.

C. SMOTE only works with continuous features, and the categorical columns were interpolated meaninglessly.

D. Recall is the wrong metric for imbalanced data, and precision would have shown the true performance.

30 · 9 min

Where the rows came from, and who is missing

THE ONE THING TO KEEP

A held-out set cannot detect a sample that misrepresents the population, because it is drawn from the same sample — so the composition of the data must be reasoned about from how it was collected, not measured from within it.

The assumption underneath every model

Every method in this course assumes your training rows are drawn from the same distribution as the rows you will predict on. When that fails, no amount of modelling repairs it, and the failure does not appear in any validation score — because the validation set is drawn from the same flawed sample as the training set.

This is the one class of problem that a held-out set cannot detect. It is worth sitting with for a moment, because everything else in this course is a variation on "hold data out and measure".

Four ways the sample goes wrong

Selection bias. Your data records only the cases that reached you. A loan model trained on approved applicants has no rows for anyone rejected, so it learns to predict default *among people the previous process approved* — and it cannot tell you anything about the people it was designed to help you evaluate. The formal name is the rejection inference problem, it is well known in credit, and it has no clean solution. The partial ones are expensive: approve a random small fraction regardless of score, and accept the losses as the price of knowing.

Survivorship bias. Only the survivors are in the table. Analysing customer behaviour using currently active customers excludes everyone who left, which

is precisely the group a churn model needs. Abraham Wald's wartime example remains the clearest: armour the parts of returning aircraft with *no* bullet holes, because planes hit there did not return.

Measurement differences between groups. The same quantity measured differently across subsets. A hospital that records blood pressure in a triage queue and another that records it in a quiet room produce different numbers for the same patient. A model trained mostly on one will misread the other.

Feedback loops. The model's own predictions change the data it is later trained on. Predict high crime in a district, send more officers, record more incidents, retrain, predict higher crime. Nothing about the district changed. This is covered again in module 9 because it is a production problem, but it starts here, in how the next dataset is generated.

Two questions, asked out loud

Before you build anything:

> ***Who is in this data, and who is not?***

> ***What had to happen for a row to exist?***

The second is the sharper one. For a customer support dataset: someone had a problem, knew the support channel existed, had the time and language to use it, and completed the form. Everyone who gave up before completing it is absent, and they may be the group you most need to understand.

For Addaly's own data, the same question has an uncomfortable answer: rows exist for people with a working internet connection, a device, and enough literacy to use the site. That is a real limit on what the data can tell us about learners in general, and stating it is better than discovering it later.

What you can actually do

Reweight, when you know the target population's composition. If your sample is 70% urban and the population is 35%, weight rural rows up. This cor-

rects the marginal distribution and does nothing about groups with zero rows.

Collect deliberately. Oversample small groups so you have enough rows to measure per-group performance — which is different from making the model treat them equally, and is a prerequisite for the fairness work in module 5.

Randomise a slice. In selection-bias settings, the only reliable fix is a small random exploration policy: approve 2% at random, show a random result to 1% of searches. It costs money and it is the only way to get unbiased data about the region your policy currently excludes.

Document the limits. Write down who the model was trained on and who it should not be used for. This is what a model card is, and module 9 builds one.

Distribution shift after deployment

The sample can also be right on day one and wrong by month six. Three named forms, because the distinction guides the fix:

- **Covariate shift** — the input distribution moves, the relationship holds. Your users get younger; age still predicts what it did. Reweighting can help.
- **Label shift** — the base rate moves. Fraud rises from 1% to 4%. Recalibration helps; the ranking may still be fine.
- **Concept drift** — the relationship itself changes. Fraudsters adapt to your model. Nothing helps except new labelled data and retraining.

They are detected differently, they are fixed differently, and calling all three "drift" is how teams end up applying the wrong remedy for a quarter.

The honest position

There is no statistical test that tells you your sample is unrepresentative of a population you did not sample. You can compare your data against a census, against another source, against what a domain expert expects, and those comparisons are worth doing. But the definitive check does not exist, and the appropriate response is to write down what you know about how the rows came

to be, and to say the limits out loud in whatever document accompanies the model.

BEFORE YOU MOVE ON

A bank's default model is trained only on approved applicants and validates well. Why does that validation not tell you how the model will perform on the full applicant pool?

A. The validation set is smaller than the applicant pool, so its confidence interval is too wide to support a conclusion.

B. Approved applicants have a lower default rate, so the model's threshold is calibrated to the wrong base rate.

C. The validation set is drawn from the same approved population, so it shares the selection bias and cannot reveal how the model behaves on applicants the old policy rejected.

D. The approval decision itself is a leaked feature, since it was made using information correlated with default.

---

31 · 9 min

When one class is 0.3% of the data

THE ONE THING TO KEEP

Imbalance usually breaks the decision threshold rather than the model, so move the threshold and change the metric before resampling — and resample only inside the cross-validation fold, never before the split.

What imbalance actually breaks

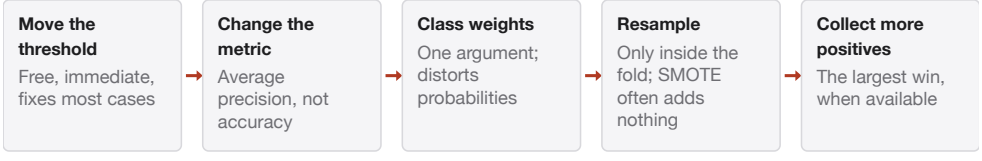
Fraud at 0.3%. Rare disease at 0.1%. Equipment failure at 2%. The intuition is that the model will "ignore" the minority class, and that is roughly right, but it

is worth being precise about the mechanism, because the precision determines the fix.

A model minimising log loss on a 0.3% positive rate can achieve a very low loss by predicting approximately 0.003 for everything. It is not broken; it is correct about the base rate and has found that the features do not move it much for most rows. The problem is that the *decision rule* you then apply — predict positive if p > 0.5 — never fires.

That distinction matters, because it means **the model is often fine and the threshold is the problem.** Try that before anything else.

The order to try things at 0.3 per cent positives



The model is usually fine; the 0.5 decision rule is what never fires. Try the cheap fixes in this order and stop when the number moves. Resampling comes last because it usually adds little over weights and a moved threshold, and it leaks if it happens before the split.

The order to try things

1. Change the threshold. Free, immediate, and it fixes the majority of cases. Get probabilities with `predict_proba`, then choose the cut point using precision and recall on validation data. Module 5 does this properly.

2. Change the metric. Accuracy is useless here — as the baseline lesson showed, predicting "never" scores 99.7%. Use precision, recall, F1, or better, average precision (the area under the precision-recall curve), which is far more informative than ROC AUC at extreme imbalance because it does not reward performance on the vast negative class.

3. Class weights. One argument, no new rows, and it changes the loss so minority errors cost more:

```
LogisticRegression(class_weight="balanced")
RandomForestClassifier(class_weight="balanced_subsample")
```

`"balanced"` sets each class's weight inversely proportional to its frequency. For XGBoost the equivalent is `scale_pos_weight`, set to the ratio of negatives to positives.

Be aware of the side effect: weighting distorts the predicted probabilities. A weighted model's 0.6 is not a 60% chance of the event. If you need calibrated probabilities — for expected-cost decisions, for pricing — either leave the weights out and move the threshold instead, or recalibrate afterwards.

4. Resampling. Only after the above.

Resampling, and why it disappoints

Random undersampling discards majority rows until the classes balance. Fast, and it throws away most of your data, which is a real loss when the majority class contains the variety the model needs to learn the boundary.

Random oversampling duplicates minority rows. No new information, and the model can memorise the duplicated rows, which shows up as an inflated validation score if the duplication happened before the split.

SMOTE creates synthetic minority rows by interpolating between a minority point and one of its k nearest minority neighbours. It is the most cited method in this area and its reputation exceeds its results.

The honest summary of the evidence: on tabular problems with a strong model such as gradient boosting, SMOTE often gives little or no improvement over simply adjusting class weights and the threshold, and several careful comparisons have found it makes calibration worse. It also assumes that the space between two minority points is itself minority territory, which need not hold — in fraud, the region between two different fraud patterns may be entirely legitimate behaviour. And it interpolates numerically, so it does not know what to do with categorical columns; SMOTE-NC exists for that and is more delicate than it looks.

If you use it: `imbalanced-learn` is the free library, and its `Pipeline` — not scikit-learn's — is the one that resamples inside each fold. Resampling before cross-validation is the leakage from the previous lesson, and it is the most common way SMOTE produces a wonderful number that means nothing.

Collect more of the rare class, if you can

Unglamorous and usually the largest win available. Two hundred more genuine fraud examples are worth more than any resampling scheme, because they contain information that no interpolation can invent. If labelling is expensive, spend it on the minority class: label all the positives you can find and a random sample of negatives, then weight the negatives up to correct for the sampling. This is standard practice in survey statistics and underused here.

When it is not a classification problem at all

At extreme imbalance — under about 0.1%, or when the positive class is heterogeneous and the negatives are uniform — reframe.

Anomaly detection models what normal looks like and flags departures, using only the majority class. Useful when the positives are too few and too varied to learn from, and when tomorrow's positives will not resemble today's. Isolation Forest and One-Class SVM are the standard tools; module 7 covers them.

Ranking drops the classification framing entirely. A fraud team can investigate 200 cases a day, so the real question is not "which are fraud" but "which 200 are most worth looking at". Optimise precision at 200, and the imbalance stops being a problem to solve and becomes a property of the situation.

That reframing is often the most valuable thing you can bring to an imbalanced problem, and it usually comes from asking what the person receiving the output is actually going to do with it.

BEFORE YOU MOVE ON

A team applies SMOTE to a 1% fraud dataset and reports cross-validated F1 rising from 0.31 to 0.68. Which check would most likely explain the gain?

A. Whether the model's probabilities are calibrated after resampling, since SMOTE distorts the predicted base rate.

B. Whether F1 was computed with macro rather than binary averaging, which changes the score substantially at 1% prevalence.

C. Whether SMOTE was applied before the cross-validation split, so synthetic points interpolated from a fold's training rows also appear in its validation portion.

D. Whether the k-neighbours parameter of SMOTE was tuned, since the default of 5 is often too small for sparse minority classes.

CASE STUDY · MODULE 3

The 0.94 that had already been promised, in Tiruppur

A garment exporter in Tiruppur, 180 staff, shipping knitwear to three European buyers, with a two-month contract for a shipment-delay prediction model.

Aruna Knits ships about 900 export consignments a year. A consignment that misses its committed date triggers a penalty clause with two of the three buyers and, more expensively, moves the exporter down that buyer's allocation list for the next season. The managing director wanted to know, at the moment a purchase order is confirmed, which consignments were at risk, so that the production plan could be arranged around them.

The work went to a two-person consultancy in Coimbatore. They were given a warehouse extract of 4,100 historical consignments with sixty-one columns, and four weeks.

Week two produced a gradient boosting model at 0.94 AUC on a held-out sample. The consultants sent a slide. The managing director sent it to his buyers' merchandising contacts. The number was, by that point, a commitment.

Week three, one of the consultants ran the check from the leakage lesson: sort the features by importance and interrogate anything that dominates. One column did. `courier_partner_assigned` was carrying more than a third of the model's gain.

The reconstruction took an afternoon. Consignments that were running on time were handed to the regular sea-freight forwarder and had that value from day one. Consignments that had slipped were reassigned to an air-freight partner, some days before the committed date, precisely because they were late. The column existed at prediction time — it was populated — but not with the value it holds in the training table. That is the second half of the leakage test, and it is the half people skip.

Two more columns were in the same family, found by the same route: `overtime_hours_sanctioned`, approved when a line is behind, and `fabric_reis-`

`sue_count`, which is incremented when a cut is rejected and the cutting has to be redone, an event that happens after the consignment is already in trouble.

Removing all three took the model from 0.94 to 0.71.

The decision was now uncomfortable in a specific way. 0.71 is not a bad model. Against the exporter's existing practice — the production manager's judgement, which had been scored on the same 4,100 consignments at roughly 0.63 — it is a real improvement worth having. But 0.94 had been on a slide, and the slide had left the building.

Report the 0.71 and explain. The cost is a conversation the consultancy did not want to have, with an exporter who had already told his buyers something, and a real chance of losing the remaining fortnight of the contract and the referral behind it.

Keep the three columns and add a caveat. They could have argued, and it is the sort of argument that wins: the columns are in the operational system, a prediction can be produced from them, the model will run. It would run and it would be useless, because at the moment the managing director needs the answer — purchase order confirmed, twelve weeks out — every consignment has the regular forwarder, no overtime and no reissues, and the model would return the same low risk for all of them. It would fail quietly and it would fail in a way that looks like the world being unpredictable rather than the model being wrong.

Split the difference: build two models. One at order confirmation with the honest features, and one at week eight with everything then known. This is more work than the contract covers, and it is the only version where the reassignment column is a legitimate input, because by week eight it means what it appears to mean.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

They reported the 0.71, in a meeting, with the reconstruction of a single consignment on a sheet of paper: here is what the system held about consignment 3,318 on the day the order was confirmed, and here is what our training table held about it. The difference was the whole argument and it did not need any statistics.

The contract was extended rather than cancelled, and the extension paid for the two-model version. The order-confirmation model at 0.71 drives the production plan. A week-eight model at 0.86, which is allowed to know about overtime and reassignment, drives the decision about whether to book air freight early — a different decision, made by a different person, with a different cost of being wrong.

The buyers were told the corrected number by the managing director, who reported afterwards that the correction had cost him less than he expected and that being told in week three rather than month six was the part that mattered.

The courier column existed at the moment of prediction. Why is it still leakage?

Because the leakage test has two halves: would the value exist, and would it exist with the value it has here. At order confirmation every consignment carries the regular forwarder, so the column exists and is uninformative. In the training table it carries the air-freight partner for exactly the consignments that later slipped. The model learns the second distribution and meets the first, so the column contributes nothing at serving time while contributing a third of the training gain.

Why did the correct train-test split fail to catch any of this?

Because a split protects against memorising particular rows, and the leaked column is present on both sides of it. Held-out consignments carry the same after-the-fact courier assignment as training consignments, so the held-out score is inflated by the same mechanism. No amount of cross-validation detects a column that arrives from the future, because the future has arrived for every row in the table.

What made the single reconstructed consignment more persuasive than the AUC comparison?

Because it turned a statistical claim into a factual one that the client could check himself. The reconstruction shows what the operational system actually held on a specific date for a specific order, against what the training table claimed. It requires no trust in the consultants' methodology, and it is the only leakage check that finds the cases nobody thought to suspect.

MODULE 3 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 Target encoding replaces each category with the mean of the target for that category. Where does the leak come from?

A. The encoding uses the test set's target values when it transforms the test rows.

B. The encoding produces a single column, so the model comes to over-rely on it.

C. A category with one row is encoded as that row's own target.

D. Category means are unstable, so the encoded feature carries a great deal of variance.

2 A loan dataset has 'employer name' blank for 12 per cent of applicants. Why add a missingness indicator before imputing?

A. Because the blank itself often predicts the target better than the value would.

B. Because imputation is only valid when a value is missing completely at random.

C. Because the imputer has to know which rows it filled in order to fit correctly.

D. Because the model would otherwise treat the imputed median as an unusually common value.

3 A sales model builds a 28-day rolling mean of each store's sales as a feature, calling .rolling(28) without a preceding .shift(1). What goes wrong?

A. The feature is undefined for the first 27 days of each store's history, so those rows are lost.

B. The window crosses store boundaries, mixing one store's sales into another store's feature.

C. The mean is computed on unscaled values, so a gradient-fitted model weights it too heavily.

D. The window includes the day being predicted.

4 A team scales the features once and then runs five-fold cross-validation on the scaled matrix. Why does moving the scaler into a Pipeline change the score rather than only tidying the code?

A. The Pipeline also scales the target variable, which the manual version had not been doing.

B. The scaler's mean is refitted per fold, so held-out rows stop contributing to it.

C. The Pipeline uses a numerically different scaling formula that is more stable in five folds.

D. Cross-validation shuffles the rows, so an externally fitted scaler is applied in the wrong order.

5 Fraud is 0.3 per cent of rows. A model trained on log loss never predicts positive at a threshold of 0.5. What should be tried first?

A. Move the threshold and change the metric.

B. Apply SMOTE to the training set so the two classes are balanced before fitting the model.

C. Undersample the majority class until the positive rate is somewhere near fifty per cent.

D. Train for longer, on the grounds that the model has not yet learned the minority class.

6 A model that will score new applicants includes 'applicant_id' as a feature, and a random row split shows excellent validation performance. What is the defect?

A. The column has too many levels for one-hot encoding to remain memory-efficient at this size.

B. Identifiers are stored as integers, so the model treats them as ordered when they are not.

C. The identifier correlates with application date, which introduces temporal leakage into the fit.

D. A new applicant's id was never in training, so it is not a feature.

MODULE 4

Generalisation, and the discipline of held-out data

Why a model that fits your data is not yet a model that works, and the machinery for finding out: cross-validation, splits that respect the structure of the data, learning curves that tell you whether more data would help, and an honest account of what happens to a validation set you have used two hundred times.

> BY THE END OF THIS MODULE YOU CAN
>
> Design a validation scheme that matches the structure of the data — grouped, stratified or time-ordered as required — run a hyperparameter search inside it without contaminating the estimate, and state from a learning curve whether the next improvement should come from more data, better features or a different model

32 · 9 min

Train, validation, test, and the discipline of not looking

> THE ONE THING TO KEEP
>
> The test set is a single-use measuring device; every peek turns it into another validation set.

Three piles, three jobs

Split your data into three piles before you do anything else.

- **Train** — the model's parameters are fitted here. Typically 60–80%.
- **Validation** — everything *you* choose is chosen here: which model, how deep, which features, what threshold. Typically 10–20%.
- **Test** — used once, at the end, to estimate how the thing will perform. Typically 10–20%.

Most people understand train and test. The pile that gets misused is validation, and the misuse has a specific mechanism.

Why a second pile exists

Suppose you try 40 configurations and score each on the same held-out data. You pick the best. That number is now optimistic, and not by a little.

Here is the arithmetic. Take 1,000 held-out rows and 40 models that are all genuinely 80% accurate. Each measured score is 80% give or take about 1.3 points of sampling noise. Take the maximum of 40 such draws and you land around 82.5% on average. Two and a half points of pure luck, and it looks exactly like progress.

So the moment you use a set to *choose*, you have spent it. It can no longer *measure*. Validation is the pile you are allowed to spend. Test is the pile you are not.

Treat the test set as a single-use instrument. Every extra look converts it into another validation set. If you evaluate on test, adjust something, and evaluate again, you no longer have a test set — you have a second validation set and no honest estimate.

Splitting wrongly is the common failure

A random split assumes every row is an independent draw. Very often it is not.

Rows that share a person. A no-show model has one row per appointment, and patients appear five or ten times. Split rows at random and the same patient sits on both sides. The model learns "patient 40218 misses appointments", which is easy and useless — patient 40218 is not who you will be

predicting for. Split by patient instead: every appointment of a given patient goes wholly into train or wholly into validation.

Rows that share time. With anything that evolves — prices, demand, fraud tactics — a random split lets the model see next month while predicting this month. Split by date: train on January to September, validate on October, test on November and December.

Rows that are near-duplicates. Scraped datasets are full of the same item twice. Deduplicate before splitting, or the split is a formality.

Ask what you are actually going to predict for. A new patient? Split by patient. A future week? Split by time. A new city? Hold out a city.

Preprocessing belongs inside the split

A quiet leak: scaling a column by the mean and standard deviation of the whole dataset, then splitting. Validation rows contributed to that mean, so information crossed the line. The same applies to imputing missing values, fitting an encoder for categories, or selecting the top 20 features by correlation.

Compute all of it on train only, then apply the stored values to validation and test. In scikit-learn that is what a `Pipeline` is for, and it is why fitting a scaler outside one is a recurring bug.

When data is scarce

With 800 rows, a 15% validation slice is 120 rows, which is too noisy to choose between models. Use **k-fold cross-validation**: split the training data into 5 parts, train five times, each time holding out a different part, and average the five scores. Every row gets used for validation exactly once and the estimate is far more stable.

Two cautions. Cross-validation replaces the validation pile, not the test pile — keep the test set out of the folds entirely. And it inherits whatever splitting mistake you made: if the folds are random rows and you needed groups, you have now made the same error five times and averaged it into something that looks solid.

Write the number down

Before the final test-set evaluation, write down what you expect and what result would make you ship. Then run it once.

If the number disappoints and you go back and tune, that is allowed — but say so out loud, and understand that your test estimate is now optimistic. This is not bureaucracy. Roughly every over-promised model comes from a team that quietly iterated against their test set and then reported the best number as if it were the first.

BEFORE YOU MOVE ON

A clinic predicts whether a patient will miss an appointment. The data has one row per appointment, and most patients appear several times. They split rows at random, 80/20. Validation accuracy is 84%. After launch it behaves like 71%. What went wrong?

A. The same patients sit on both sides of the split, so the model was rewarded for recognising individuals rather than generalising to new ones.

B. The validation set was too small to give a stable estimate.

C. Patient behaviour changed after launch, so the data drifted.

D. They should have used cross-validation instead of a single split.

33 · 8 min

Overfitting, underfitting, and how to tell

THE ONE THING TO KEEP

Compare training error with validation error; the gap between them, not the score itself, names the problem.

Two ways to fail

A student who understands the subject does well on a new exam. A student who memorised last year's paper does well on last year's paper. A student who skimmed the syllabus does badly on both.

Models fail in exactly these two directions.

Overfitting — the model has captured accidents of the training data: this particular noise, these particular rows. Training error is low, new-data error is much higher.

Underfitting — the model is not expressive enough, or the features do not carry the signal. Both errors are high.

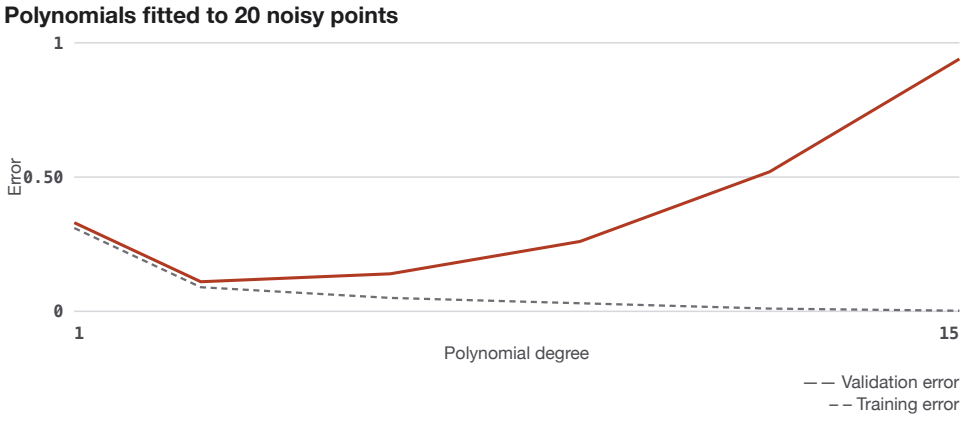
Capacity

The dial between them is **capacity**: how many distinguishable functions the model family can express.

Fit polynomials to 20 noisy points.

- Degree 1, a straight line. Training error 0.31, validation error 0.33. It cannot follow the shape.
- Degree 3. Training error 0.09, validation 0.11. About right.
- Degree 15. Training error 0.002 — the curve threads every point exactly. Validation error 0.94, worse than the straight line, because between the points it swings wildly.

Training error falls monotonically with capacity. Validation error falls, bottoms out, and then rises. That U is the whole story, and every capacity knob you meet is a way of moving along it: tree depth, number of trees, number of layers, number of features, how long you train.



Training error falls without limit as the degree rises. Validation error bottoms out near degree three and climbs to 0.94 at degree fifteen — worse than the straight line — because between the points the curve swings wildly. Every capacity knob you meet moves you along this U.

Diagnosis takes two numbers

You cannot diagnose from one score. You need training error and validation error side by side.

Train	Validation	Diagnosis	What to do
High	High, similar	Underfitting	More capacity, better features, train longer
Low	Much higher	Overfitting	More data, less capacity, regularization, fewer features
Low	Low	Working	Stop touching it
High	Lower than train	Something is wrong	Check the split, check for duplicates

That last row is not a joke. Validation beating training usually means leakage, a broken split, or dropout being left on during evaluation.

The common mistake is reading row one as row two. "Overfitting" has drifted in casual use to mean "the model is bad", so a disappointing score attracts the word regardless of the gap. If training accuracy is 71% and validation is 70%, nothing is being memorised. The model is not seeing enough, and cutting capacity will make it worse.

The cures, in order of how well they work

More data. The most reliable fix and usually the least available. Doubling the rows moves overfitting more than any hyperparameter will.

Less capacity. Shallower trees, fewer components, a smaller network.

Regularization — a penalty added to the loss that discourages complexity:

- *L2 (ridge)*: add the sum of squared weights to the loss. Weights shrink toward zero and no single feature dominates.
- *L1 (lasso)*: add the sum of absolute weights. Weights are driven exactly to zero, which selects features as a side effect.
- *Early stopping*: watch validation error each epoch and stop when it turns upward. Free, and it works.
- *Dropout*: during training, randomly zero out some fraction of a network's units so no unit can rely on any other.
- *Augmentation*: manufacture more training data by transforming what you have — flipping, cropping, adding noise. For images this is the strongest tool on the list.

All of these have a knob, and every knob is chosen on validation. Not on test.

An honest complication

The U-curve is the classical picture and it holds for the tabular models most people build. Very large neural networks break it. Push capacity far past the point of fitting the training data perfectly and validation error sometimes falls *again* — the phenomenon is called double descent, and a network with more parameters than training examples can still generalise well.

Why that happens is still argued over. It does not mean overfitting was a myth; it means capacity alone is a cruder predictor of generalisation than the textbook curve suggests. For a gradient-boosted model on 50,000 rows, keep using the U. For a 70-billion-parameter network, the intuitions in this lesson stop being reliable.

BEFORE YOU MOVE ON

A gradient-boosted model reaches 71% accuracy on training data and 70% on validation. A colleague says it is overfitting and wants to cut the tree depth from 6 to 3. What does the evidence show?

A. It is underfitting — both numbers are low and the gap is negligible, so it needs more capacity or better features.

B. It is overfitting, since 70% is a weak result and overfitting is what makes results weak.

C. It is fitted well; a one-point gap is what a healthy model looks like.

D. Nothing can be concluded until they check the test set.

---

34 · 9 min

Cross-validation: using every row twice, legally

THE ONE THING TO KEEP

Cross-validation averages several held-out estimates so a score depends less on where the split fell, but the best score from a hyperparameter search is biased upward and needs an outer loop or an untouched test set to correct.

The problem with one split

Hold out 20% and score on it. Now do it again with a different random seed. The two scores differ — sometimes by half a point, sometimes by four, de-

pending on how much data you have.

With 500 rows, a 20% test set is 100 rows. If the model gets 83 right you report 83%, and the 95% interval around that runs from roughly 74% to 90%. That range is wide enough to cover "ship it" and "abandon it". A single split is a measurement with a large and usually unreported error bar.

Cross-validation reduces that error by measuring more than once.

The mechanism

Split the data into k parts. Train on k-1 of them, score on the one left out. Repeat until every part has served as the held-out fold once. You get k scores; report their mean and their standard deviation.

```
from sklearn.model_selection import cross_val_score
import numpy as np
s = cross_val_score(pipe, X_train, y_train, cv=5,
scoring="roc_auc")
print(f"{s.mean():.3f} +/- {s.std():.3f}")
```

Every row is used for training in k-1 of the runs and for evaluation in exactly one. Nothing is wasted, and the mean of five scores has less variance than any one of them.

The cost is that you fit the model k times. For a linear model on 50,000 rows that is seconds. For a large network it is a real decision, which is why deep learning usually uses a single large validation set instead.

Choosing k

k = 5 is the default and the right answer most of the time. Each fold trains on 80% of the data, which is close enough to the full set that the estimate is not badly pessimistic.

k = 10 gives a slightly less biased estimate at twice the cost. Worth it on small datasets where 80% is meaningfully less than 100%.

Leave-one-out (k = n) trains on everything but a single row, n times. The estimate has very low bias and famously high variance, and it costs n fits. It is occasionally right for very small datasets and is usually a sign that somebody has more compute than data.

There is a trade-off in k that is worth stating rather than absorbing: larger k means each model trains on more data, so the estimate is less pessimistic, but the k training sets overlap almost entirely, so the k scores are highly correlated and averaging them removes less variance than it appears to.

Stratify, always, for classification

`StratifiedKFold` makes each fold contain the same proportion of each class as the whole set. This is the default for classifiers in scikit-learn's `cross_val_score`, and you should keep it.

Without it, on a 2% positive rate with 5 folds and 1,000 rows, a fold might get 4 positives and another 12. The scores swing wildly for reasons that have nothing to do with the model, and at extreme imbalance a fold can end up with zero positives, at which point some metrics are undefined and others silently return zero.

Repeated cross-validation

If you have the compute and the dataset is small, run the whole k-fold procedure several times with different random partitions:

```
from sklearn.model_selection import RepeatedStratifiedKFold
cv = RepeatedStratifiedKFold(n_splits=5, n_repeats=3,
random_state=0)
```

Fifteen fits, and the variability across repeats tells you how much of your score depends on where the fold boundaries happened to fall. If that variability is comparable to the difference between two models you are comparing, you do not have evidence to prefer one.

What the standard deviation is telling you

Report it. "0.847 +/- 0.004" and "0.847 +/- 0.061" are completely different situations, and quoting only the mean hides which one you are in.

One warning about that number: the standard deviation across folds is not a proper confidence interval on the score. The folds share most of their training data, so the scores are correlated, and the usual formula for a standard error understates the true uncertainty — sometimes badly. Treat it as a rough indication of stability rather than a statistic you can put in a hypothesis test. Module 5 covers comparing two models properly.

The nested problem

Here is the trap that catches people who have understood everything so far.

Use cross-validation to choose hyperparameters, then report the best cross-validated score as your estimate of performance. That number is optimistic, because you selected the maximum of many noisy estimates, and the maximum of noisy things is biased upward. Try 200 configurations and the best one is partly good and partly lucky.

The fix is nested cross-validation: an inner loop that chooses hyperparameters and an outer loop that measures the whole procedure including the choosing.

```
from sklearn.model_selection import GridSearchCV, cross_val_s-
core
inner = GridSearchCV(pipe, grid, cv=5)
scores = cross_val_score(inner, X, y, cv=5)   # 5 x 5 = 25 mod-
el fits
```

Expensive, and honest. In practice most teams use a simpler equivalent: tune with cross-validation on the training set, and keep a genuinely untouched test set for the final number. That is cheaper and gives you the same protection, as long as the test set really is touched once.

BEFORE YOU MOVE ON

A team runs GridSearchCV over 300 configurations with cv=5 and reports the best mean cross-validated AUC of 0.882 as their expected production performance. Why is that number optimistic?

A. Selecting the maximum over 300 noisy estimates favours configurations that happened to suit these particular folds, so the winner's score includes some luck that will not repeat.

B. Five folds is too few to estimate AUC reliably, and ten folds would have given the true value.

C. GridSearchCV refits on the full training set at the end, and the refitted model differs from the ones that were scored.

D. Cross-validated AUC always overstates production AUC because production data is drawn from a later period.

35 · 9 min

Splits that respect the structure of the data

THE ONE THING TO KEEP

Match the split to the structure — group by entity, order by time with a gap equal to the label delay, stratify small or imbalanced sets — and verify with a label-shuffle test that a correct pipeline collapses to chance.

Random is a choice, not a default

`train_test_split` shuffles rows and cuts. That is correct only when rows are independent, and in real data they usually are not. Three structures break it, and each has a matching splitter.

Grouped data: split the entity, not the row

Whenever several rows belong to the same underlying thing — patient, customer, device, document, household — those rows share information. A random split puts some of a patient's visits in training and the rest in test, and the model can identify the patient rather than the condition.

```
from sklearn.model_selection import GroupKFold,
StratifiedGroupKFold
cv = StratifiedGroupKFold(n_splits=5)
for tr, te in cv.split(X, y, groups=df["patient_id"]):
    ...
```

`StratifiedGroupKFold` keeps whole groups together *and* balances the classes, which is what you almost always want and which did not exist in scikit-learn until relatively recently.

The question to ask: *at prediction time, will I have other rows from this same entity in my training data?* For a hospital predicting readmission for an existing patient, yes — and a random split may be defensible. For a model that will run on new patients, no, and grouped splitting is mandatory.

Time series: the future does not leak backwards

For anything with a time axis, train on the past and test on the future. Always. Even when there are no explicit date features, because data carries temporal structure implicitly.

```
from sklearn.model_selection import TimeSeriesSplit
cv = TimeSeriesSplit(n_splits=5, test_size=30)
```

This produces expanding windows: train on days 1-100 test on 101-130, train on 1-130 test on 131-160, and so on. Each evaluation only ever uses the past.

Two refinements matter in practice.

A gap. If your label takes 30 days to become known — did this customer default within 30 days — then at the moment of prediction you cannot know the

labels for the last 30 days of "past" data. Leave a gap between the training window and the test window equal to the label delay, or you are training on labels that would not have existed yet. `TimeSeriesSplit(gap=30)` does it.

Rolling versus expanding. Expanding uses all history, which is right when old patterns still hold. Rolling uses a fixed recent window, which is right when they do not. Which one is better is an empirical question about your data, and comparing them tells you something real about how fast your problem changes.

Stratification: keeping the folds comparable

For classification, stratify on the label. For regression with a skewed target, you can stratify on binned quantiles of the target — not built in, but three lines:

```
import pandas as pd
bins = pd.qcut(y, q=10, labels=False, duplicates="drop")
train_test_split(X, y, stratify=bins, test_size=0.2,
random_state=0)
```

This matters most on small datasets, where one fold can otherwise get all the expensive houses.

When several structures apply at once

Real problems stack them. Predicting equipment failure: rows are grouped by machine *and* ordered in time *and* the failure class is rare. You need all three constraints, and no single scikit-learn splitter does it.

The usual answer is to write the split yourself: choose a cut date, assign whole machines to train or test, and check the class balance on both sides afterwards. Twenty lines, and it is the honest version.

When you cannot satisfy every constraint — you have 40 machines and need both a time split and a group split — say which one you relaxed and why. A stated compromise is engineering; an unstated one is a result nobody can interpret.

Two sanity checks worth running

The adversarial validation trick. Label your training rows 0 and your test rows 1, and try to build a classifier that tells them apart. If it can — AUC well above 0.5 — then the two sets differ systematically, and your test score is measuring performance on a different population. This finds split bugs, hidden time ordering, and genuine distribution shift, and it takes ten lines.

Shuffle the labels. Randomly permute `y` and rerun the whole pipeline. Every score should collapse to chance. If it does not, something in your pipeline is reaching the answer by a route you did not intend, and you have found leakage that no amount of reading the code would have revealed.

That second check is one of the highest-value five-minute tests in this course. Run it on any result that surprises you.

BEFORE YOU MOVE ON

A model predicts whether a customer will default within 30 days. The team uses TimeSeriesSplit without a gap, training up to day 200 and testing from day 201. What is wrong?

A. The test window should be randomly sampled from the whole period, so it represents the full range of conditions.

B. The training set should be a rolling window rather than an expanding one, since old customer behaviour is no longer relevant.

C. Test rows from day 201 onward have labels that are only known 30 days later, so the test set includes outcomes that cannot yet be observed.

D. Labels for training rows near day 200 are only known by day 230, so a model built at day 200 could not have had them; the split trains on information that would not have existed.

36 · 9 min

Bias and variance, with numbers attached

THE ONE THING TO KEEP

High bias is being wrong the same way every time and high variance is being wrong differently every time; the training-validation gap tells you which, and the classical U-shaped curve stops describing very large models.

The decomposition

Imagine fitting your model on many different samples of the same size from the same population, and asking about the expected squared error on one test point. It splits into three parts:

```
expected error = bias^2 + variance + irreducible noise
```

Bias is how far the *average* prediction across all those fits is from the truth. High bias means the model family cannot represent the pattern — a straight line trying to fit a curve. It is wrong the same way every time.

Variance is how much the prediction moves between fits. High variance means the model is sensitive to which rows it happened to see. It is wrong differently every time.

Irreducible noise is the part of the target no model can predict, because it is not determined by your features. This is the ceiling from module 1, and no method removes it.

Watching it happen

Fit polynomials of increasing degree to 20 noisy points from a curve, and refit each on 50 different samples of 20 points.

- **Degree 1.** All 50 lines are nearly identical — low variance — and all of them miss the curve in the same places. High bias.
- **Degree 3.** The 50 curves are close to the truth on average and reasonably close to each other. This is the sweet spot.
- **Degree 15.** Each of the 50 curves passes almost exactly through its own 20 points, and they are wildly different from each other. Average them and you get close to the truth — low bias — but any single one is far off. High variance.

The useful thing about this picture is that it names the fix. **High bias: more capacity, better features, less regularisation. High variance: more data, more regularisation, less capacity, or averaging several models.**

How to tell which you have

From the two numbers you already have.

- Training error high, validation error about the same and also high: **bias**. The model cannot even fit what it has seen.
- Training error low, validation error much higher: **variance**. It fitted what it saw and did not generalise.
- Both low, close together: you are done.
- Training error high, validation error *lower*: check your pipeline, this is usually a bug or an artefact of regularisation being active only during training.

One caution about "high" and "low". They only mean anything relative to the irreducible noise. A training error of 12% on a task where two experts disagree 11% of the time is not high bias; it is a model at the ceiling. This is why estimating human agreement, from module 1, is worth the afternoon it costs.

Bagging and boosting, in one sentence each

The decomposition explains the two great ensemble families, which module 6 covers in full.

Bagging — random forests — fits many high-variance models on different bootstrap samples and averages them. Averaging reduces variance without changing bias, so it takes a family that is individually unstable and makes it stable. That is why the individual trees in a random forest are grown deep and unpruned: you *want* them high-variance, because averaging handles it.

Boosting fits many high-bias models in sequence, each correcting the previous one's errors. It attacks bias, which is why the individual trees in a boosted model are shallow — depth 3 to 6, models that on their own are barely better than a coin.

Same decomposition, opposite ends of it.

Where the classical picture breaks

The textbook draws a U: error falls as complexity rises, reaches a minimum, then climbs as variance takes over. Choose the bottom of the U.

That picture is accurate for the models in this module and misleading for large neural networks, which routinely have far more parameters than training rows and should therefore be at the far-right catastrophic end of the U. They are not; they generalise well.

The empirical finding, named **double descent**, is that test error follows the U up to roughly the point where the model can exactly fit the training data, and then *falls again* as capacity increases further. This is not folklore; it has been demonstrated repeatedly since 2018 across model families, and it is genuinely not fully explained. The leading accounts involve implicit regularisation by the optimiser and the geometry of the solutions gradient descent happens to find, and they remain accounts rather than a settled theory.

What that means for you: the bias-variance frame is a good instrument for the models in this course and should not be treated as a law of nature. Anyone who tells you a model with more parameters than data must overfit is describing a picture that stopped being complete about eight years ago.

BEFORE YOU MOVE ON

A random forest gets 0.02 training error and 0.19 validation error. The team proposes increasing max_depth. What will happen and why?

A. It will help, because deeper trees capture more of the interaction structure the model is currently missing.
B. It will make things worse, because the gap indicates variance and deeper trees increase variance rather than reducing it.
C. It will have no effect, since random forests average many trees and are insensitive to depth.
D. It will help, because random forests underfit by default and depth is their main capacity control.

37 · 8 min

Would more data help? The curve that answers it

THE ONE THING TO KEEP

Plot validation score against training-set size: a curve still climbing means collect more data, a converged pair at a poor score means the features or the model family are the limit, and no learning curve can anticipate a feature you have not built.

An expensive question with a cheap answer

"Should we spend three months collecting more data?" is one of the most consequential questions a team asks, and it is usually answered by intuition. It can be answered by a plot that takes twenty minutes.

Fit the model on 10% of your training data, then 20%, and so on, recording both training and validation score at each size.

```
from sklearn.model_selection import learning_curve
import numpy as np
sizes, train_s, val_s = learning_curve(
    pipe, X, y, cv=5,
    train_sizes=np.linspace(0.1, 1.0, 8), scoring="roc_auc")
```

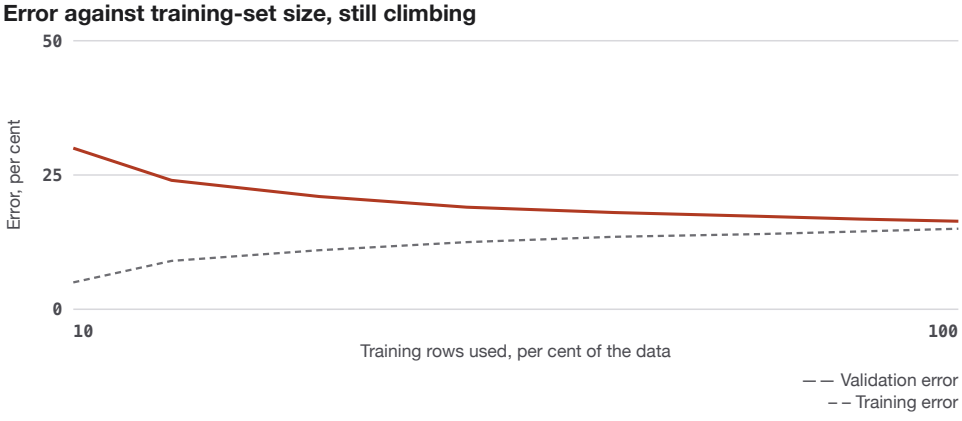
Plot the means of `train_s` and `val_s` against `sizes`.

Three shapes, three decisions

The validation curve is still climbing at 100%. More data will help. The size of the remaining slope tells you roughly how much: if the last doubling bought two points, the next doubling will buy less than two, and you can extrapolate a rough answer to "is 50,000 more rows worth it".

The validation curve flattened at 40% and the training curve is far above it. More of the same data will not help much. You have a variance problem that additional rows are no longer reducing, and the remedy is regularisation, fewer features, or a simpler model.

Both curves flattened and converged at a poor score. Neither more data nor more regularisation will help. The model family cannot express the pattern, or the features do not contain it. Better features, or a different model family. This is the most common shape in a stalled project and the one people most often misdiagnose as needing more data.



Validation error is still falling at 100 per cent of the data: the last doubling took nearly three points off, so more rows will help, and the next doubling will buy less than three. Training error rising is the model meeting reality, not the model getting worse.

Why the training curve falls

A detail worth understanding rather than memorising. The training score usually *decreases* as the training set grows, and beginners often read this as the model getting worse.

It is not. With 50 rows a flexible model can fit them nearly perfectly, so training error is near zero. With 50,000 rows it cannot fit them all, so training error rises to something realistic. The training curve falling and the validation curve rising, converging towards each other, is exactly the picture of a model becoming better calibrated to reality.

What the curve cannot tell you

Three limits, stated because this plot is often over-read.

It assumes more data of the same kind. If your existing rows are biased — the selection problems from module 3 — then more of the same will not fix a thing, and the curve will happily suggest that it might.

It cannot see a new feature. A learning curve that has flattened says nothing about whether adding a `days_since_last_purchase` column would change everything. Feature improvements move the whole curve upward, and the plot has no way to anticipate that.

Extrapolating far beyond your current size is unreliable. Doubling is a defensible extrapolation. Predicting what ten times the data buys is not, because the curve's shape is empirical and the tail is exactly the part you have not measured.

Scaling laws, and their limits

For large models on large corpora, the relationship between data, parameters, compute and loss has been studied enough to produce fitted power laws — the Kaplan and Chinchilla lines of work — that predict loss surprisingly well over several orders of magnitude, and that changed how large models are budgeted.

Two honest caveats before anyone applies that here. Those laws were fitted for a specific setting: next-token prediction with transformers on text at very large scale. They describe *loss*, which is not the same as the accuracy of a downstream task, and the mapping between the two is not a smooth one. And they say nothing about a gradient boosting model on 80,000 rows of tabular data, where the practical returns to more data flatten far sooner.

For the work in this course, the learning curve on your own data is a better instrument than any general law.

The cheaper version

If `learning_curve` is too slow on your dataset, do it manually at three sizes — 25%, 50%, 100% — and look at the direction. Three points is enough to distinguish "still climbing" from "flat", which is the decision you actually need to make. The eight-point version is nicer to look at and rarely changes the answer.

BEFORE YOU MOVE ON

A learning curve shows training and validation scores converging at 0.71 by 40% of the data and staying flat to 100%. What should the team do next?

A. Collect more data, since the flat region indicates the model has not yet seen enough variety.
B. Add regularisation, since the two curves meeting indicates the model is overfitting.
C. Build better features or move to a more expressive model family, since both curves have converged at a low score, indicating the current setup cannot express the pattern.
D. Increase the number of cross-validation folds to get a more reliable estimate before deciding.

38 · 9 min

Searching hyperparameters without fooling yourself

THE ONE THING TO KEEP

Random search over log-scaled distributions beats grid search at equal cost because it tries many distinct values of the parameters that matter, and every configuration evaluated adds selection bias that only an untouched test set can measure.

Parameters and hyperparameters

Parameters are learned from the data — the coefficients, the split thresholds, the network weights. Hyperparameters are set by you before training: `max_depth`, `alpha`, `learning_rate`, `n_estimators`.

The distinction matters because parameters are fitted on the training set and hyperparameters must be chosen on the validation set. Choosing them on the

training set means picking whatever fits it hardest, which is always the most complex option available.

Grid search, and why it wastes most of its budget

```
from sklearn.model_selection import GridSearchCV
grid = {"model__max_depth": [3, 5, 7],
        "model__learning_rate": [0.01, 0.05, 0.1],
        "model__min_samples_leaf": [10, 20, 50]}
search = GridSearchCV(pipe, grid, cv=5, scoring="roc_auc",
n_jobs=-1)
```

Every combination: 3 x 3 x 3 = 27 configurations, times 5 folds, is 135 fits. Adding one more value to one parameter takes it to 180.

The deeper problem is not cost. It is that in most problems only two or three hyperparameters matter, and grid search spends the same budget on the ones that do not. With 27 configurations, the important parameter was tried at exactly three values.

Random search, and why it wins

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import loguniform, randint
dist = {"model__max_depth": randint(2, 12),
        "model__learning_rate": loguniform(1e-3, 3e-1),
        "model__min_samples_leaf": randint(5, 100)}
search = RandomizedSearchCV(pipe, dist, n_iter=60, cv=5,
n_jobs=-1,
                            random_state=0)
```

Sixty random draws from those distributions try sixty *distinct values* of each parameter, rather than three. When one parameter dominates — and one usually does — random search explores it sixty times more finely for the same budget. Bergstra and Bengio's 2012 result made this argument, and it has held up: for the same number of fits, random search generally finds a better configuration than grid search.

Use `loguniform` for anything whose useful range spans orders of magnitude — learning rates, regularisation strengths, `C`. Sampling `alpha` uniformly between 0.001 and 100 puts 99.9% of your draws above 0.1, which is almost certainly the wrong end.

Bayesian and successive halving

Bayesian optimisation builds a model of the score as a function of the hyperparameters and picks the next point to try where the expected improvement is greatest. `optuna` is the free standard here and it is genuinely good. It pays off when each fit is expensive; when a fit takes two seconds, the overhead of being clever exceeds the savings.

Successive halving starts many configurations on a small subset of the data, kills the worst half, doubles the resources for the survivors, and repeats. `HalvingRandomSearchCV` is in scikit-learn. It is a large speedup and it carries an assumption worth knowing: that a configuration which looks poor on 10% of the data would also be poor on 100%. That assumption is usually but not always right, and it fails most often for configurations that need more data to show their value.

Which hyperparameters actually matter

A short list, because the space is large and most of it is irrelevant.

- **Gradient boosting**: `learning_rate` and `n_estimators` together (halve one, double the other), then `max_depth` or `num_leaves`, then `min_child_weight` and the subsampling fractions. Everything else is fine-tuning.
- **Random forest**: `max_features` first, then `min_samples_leaf`. `n_estimators` is not really a hyperparameter — more is never worse, only slower, so set it as high as you can afford and stop tuning it.
- **Linear models**: `C` or `alpha`, on a log scale. That is essentially the whole search.
- **Neural networks**: learning rate, by a wide margin, then batch size and weight decay.

Tune three parameters well rather than nine badly.

The cost you are accumulating

Every configuration you evaluate is another look at the validation data, and the more you look, the more the best score reflects the particular noise in that set. Two hundred configurations is two hundred chances for something to score well by luck.

This is the subject of the next lesson, and it is the reason for the discipline that follows: tune on cross-validation over the training data, and keep a test set that no search has ever seen. When you finally score on it and get a number a point or two below your best cross-validated score, that is not a disappointment — it is the search bias becoming visible, which is exactly what the test set is for.

BEFORE YOU MOVE ON

A team searches ridge alpha over a uniform grid of 20 values between 0.01 and 200 and finds the best at 0.01, the smallest value tried. What does this suggest about the search design?

A. The penalty is unnecessary and they should refit with no regularisation at all.

B. Twenty values is too few, and a denser grid over the same range would find a better optimum.

C. The optimum is at the lower boundary, and a uniform grid puts almost all its points above 10, so the informative region below 1 was barely sampled; use a log scale extending lower.

D. Cross-validation noise made the smallest alpha appear best by chance, and repeating with a different seed would give a different winner.

39 · 8 min

A validation set you have used two hundred times

THE ONE THING TO KEEP

The best of many noisy validation estimates is biased upward by roughly the standard error times the square root of twice the log of the number of comparisons, so the count of experiments belongs in the report alongside the score.

Overfitting the validation set

You did everything right. You split off a validation set, you never trained on it, you used it only to compare models.

And then you compared 340 models against it over six weeks.

Each comparison used the validation set to make a decision. The final model was chosen because it scored highest, and part of why it scored highest is that its particular errors happened not to land on those particular rows. The validation set has been slowly incorporated into the model through your decisions, and its score is no longer an unbiased estimate of anything.

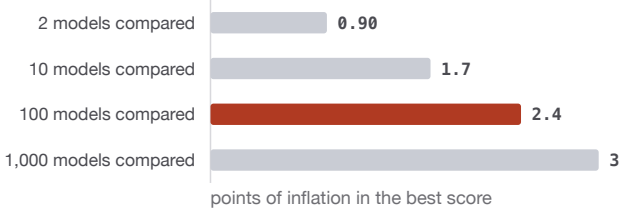
This is sometimes called the garden of forking paths, and it has a mathematical shape: **the maximum of many noisy estimates is biased upward**, and the bias grows with the number of estimates and shrinks with the size of the set.

Roughly how much

A rule of thumb that is worth carrying. If the standard error of your validation score is `s` and you compare `k` genuinely equivalent models, the best of them will beat the true value by roughly `s` times the square root of `2 ln k`.

With a 2,000-row validation set, accuracy around 85%, the standard error is about 0.8 percentage points. Compare 100 models and the winner is inflated by roughly 2.4 points — enough to turn "no real improvement" into "a clear win". Compare 1,000 and it is about 3 points.

How far the winning score overstates the truth



A 2,000-row validation set, accuracy near 85 per cent, standard error about 0.8 points, models genuinely equivalent. The best of a hundred beats its true value by roughly 2.4 points — enough to turn no improvement into a clear win. Nothing was faked; noise was selected for.

That calculation explains a phenomenon everybody has seen: a long tuning session that produces a two-point gain which then fails to appear in production. Nothing was faked. The gain was measurement noise, selected for.

Four defences

1. Keep a test set that is genuinely untouched. The test set is your protection against exactly this. It works only if you use it once, at the end. A test set consulted three times has become a slow validation set.

2. Make the validation set large enough that its noise is small. If the standard error is 0.2 points, a hundred comparisons cost you 0.6 points instead of 2.4. Data spent on validation is not wasted; it buys resolution.

3. Rotate. Refresh the validation set periodically from newly collected data, so that decisions made against the old one do not accumulate against the new one. This is normal practice on long-running production systems and it is worth setting up early.

4. Compare fewer things. The bias grows with the number of comparisons, so a smaller, better-reasoned search is not just cheaper — it is more honest. This is the practical case against searching 5,000 configurations because the compute was available.

The competition evidence

Kaggle provides an unusually clean demonstration. Competitors tune against a public leaderboard computed on part of the test data, and the final ranking uses a separate private portion. Teams that tuned heavily against the public score routinely drop dozens of places when the private score is revealed, and the effect is largest in competitions with small test sets.

A useful piece of context on the other side, though. Several careful studies have rebuilt fresh test sets for well-known benchmarks — a new CIFAR-10 test set, a new ImageNet test set — and found that while every model scored lower on the new data, the *ranking* of models was largely preserved. So years of community-wide tuning against a fixed benchmark inflated absolute numbers considerably and distorted relative comparisons much less than many expected. Both halves of that finding are worth carrying: the absolute number decays fast, the ordering is sturdier.

Writing it down

The practice that actually helps is unglamorous. Keep a log with one line per experiment: what you changed, the validation score, the date. At the end, count the lines.

If you ran 400 experiments and your final model beats your first by 1.5 points, you should say so in exactly those terms, because that sentence lets a reader apply the correction themselves. "We improved AUC from 0.831 to 0.846 over 400 experiments on a 3,000-row validation set" is an honest claim. "We achieved 0.846 AUC" is technically true and tells the reader nothing they need.

The number of times you looked is part of the result.

BEFORE YOU MOVE ON

Two teams each report 0.846 validation AUC on the same 3,000-row set. Team A tried 6 configurations; Team B tried 900. What should you conclude?

A. Team B's model is more likely to be genuinely better, since it explored the space far more thoroughly.

B. Team A's number is more likely to hold up, because the same score selected from 900 candidates carries substantially more upward selection bias than one selected from 6.

C. The two are equivalent, since both used the same validation set and the same metric.

D. Neither can be assessed without knowing the training set sizes, which determine the variance of the estimates.

40 · 8 min

How much data do I need?

THE ONE THING TO KEEP

Size the validation set from the precision you need — the standard error of a proportion at n=1,000 is about 1.1 points — and size the training set by events per feature rather than by total rows.

The honest answer, and then the useful ones

The honest answer is that it depends on how strong the signal is, and you cannot know that before you have some data. What follows are the approximations practitioners actually use, with their reasoning, so you can adapt them rather than recite them.

Rules of thumb, and where they come from

Ten to twenty rows per feature, for a linear model. Below that, the coefficients are estimated so noisily that they are close to meaningless. It comes from the statistics of parameter estimation, and 30 features with 200 rows is genuinely a bad situation regardless of what the validation score says.

Ten to fifteen *events* per feature, for classification. This is the version that matters, and the distinction is the one people miss. With 100,000 rows and 200 fraud cases, your effective sample size for learning what fraud looks like is 200, not 100,000. This is the epidemiologists' events-per-variable rule and it transfers directly.

A few hundred labelled examples per class, minimum, for anything non-trivial. Under 50 per class you cannot reliably distinguish a good model from a lucky one, because your validation set for that class is a dozen rows.

For text classification with TF-IDF and a linear model, a few thousand labelled documents usually gets you most of the achievable performance. For fine-tuning a pretrained transformer, a few hundred to a few thousand is often enough, because the model already knows the language and you are only teaching it your task.

The number that actually constrains you

Work backwards from the validation set instead of forwards from the training set. Ask: *how precisely do I need to know my score?*

For a proportion, the standard error is roughly `sqrt(p * (1 - p) / n)`. At p = 0.85:

- n = 100: standard error 3.6 points. The 95% interval spans about 14 points.
- n = 1,000: 1.1 points. Interval about 4.5 points.
- n = 10,000: 0.36 points. Interval about 1.4 points.

So if you need to detect a 1-point improvement, you need on the order of 10,000 validation rows and no fewer. If you have 300, you can detect large

differences and nothing else, and running a 200-configuration search against those 300 rows is theatre.

This calculation takes one minute, it is the single most useful thing in this lesson, and almost nobody does it before starting.

When you have less than you need

All of these are real options, and none of them manufactures information.

- **Use a simpler model.** Fewer parameters to estimate, less data required. On 400 rows, logistic regression frequently beats gradient boosting.
- **Regularise harder.** Strong regularisation is how you spend a small dataset without overfitting it.
- **Use cross-validation rather than a single split.** Every row does both jobs.
- **Transfer learning.** Start from a model trained on a large related dataset and adapt. This is why a few hundred images can be enough for image classification today and were nowhere near enough in 2012.
- **Augment.** For images and audio, generate variations — rotations, crops, noise. Genuinely effective, and it works because the transformations preserve the label. For tabular data there is no equivalent that reliably works; SMOTE is the usual attempt and module 3 explained its limits.
- **Use rules for the part you understand** and a model for the rest, as module 1 argued.

The counter-argument to "we need more data"

Before anybody commissions a collection effort, run the learning curve from two lessons ago. Teams routinely ask for more data when their curve has been flat for the last three doublings, and what they actually need is a feature nobody has built.

The order that saves the most time: check the learning curve, check the baseline, check for leakage, look at 50 misclassified rows by hand. Then decide

about data collection. Every one of those steps is cheaper than a quarter of collection, and any of them can change the answer.

And when collection genuinely is the answer, be specific about what to collect. "More data" usually means one of three quite different things: more rows of the same kind, which helps only if the learning curve is still climbing; more rows of a specific underrepresented kind, which helps when a slice is failing; or more *columns* about the rows you already have, which is not a collection project at all but a data-access one. Teams that ask for the first and needed the third spend six months and land back where they started.

BEFORE YOU MOVE ON

A team has 40,000 rows with a 0.5% positive rate and 60 features. They are surprised that a gradient boosting model performs poorly. What does the events-per-variable view say?

A. 40,000 rows is ample for 60 features, so the problem must be feature quality rather than sample size.

B. They have about 200 positive cases for 60 features — roughly 3 events per variable — so there is far too little information about the positive class to estimate 60 effects.

C. The class imbalance means accuracy is misleading, and switching to average precision would reveal that the model is fine.

D. Gradient boosting requires balanced classes, so undersampling the majority to 200 rows would fix it.

41 · 8 min

Double descent, and the picture that stopped being complete

THE ONE THING TO KEEP

Test error can fall again past the point where a model fits the training data exactly, because extra capacity gives the optimiser many perfect fits to choose among and gradient descent prefers smooth ones — an observation that is well replicated and not yet fully explained.

What everybody was taught

The classical result says: as you add capacity, test error falls, bottoms out, and rises. Past the interpolation point — where the model can fit the training data exactly — it should be fitting noise and generalising terribly. Choose the bottom of the U.

That picture is correct for the models in this course. It is correct for polynomial regression, for decision trees, for k-NN. It is why `max_depth` and `alpha` exist.

And it does not describe large neural networks, which is a problem, because large neural networks are the reason most people are reading this.

The observation

A network with 100 million parameters trained on 50,000 images has 2,000 parameters per example. By the classical account it should memorise the training set perfectly and be useless on anything else.

It is not. It generalises well, often better than a smaller network that sits at the classical optimum.

When people plotted test error against model size carefully across the whole range, the curve turned out to have a second descent. Error falls, rises to a

peak exactly where the model becomes just barely able to fit the training data, and then falls again — often below the first minimum — as capacity keeps increasing. Belkin and colleagues named it **double descent** in 2019, and Nakkiran and colleagues showed the same shape appears as a function of training time and of dataset size, not only model size.

Why the peak is where it is

The most useful intuition concerns the region right at the interpolation threshold, and it is worth having because it explains why the peak is a peak rather than a plateau.

A model with *just barely* enough capacity to fit the training data has exactly one way to do it. That solution is forced, it is pinned to every noisy point, and it is wildly unstable — this is the classical high-variance disaster.

A model with *far more* capacity than needed has infinitely many ways to fit the data perfectly. Which one it lands on is decided by the optimiser, and gradient descent from a small random initialisation happens to prefer solutions with small weights and smooth behaviour between the training points. That preference is not written down anywhere in the loss function; it is a property of the search procedure. The field calls it **implicit regularisation**.

So the extra capacity is not being used to fit more noise. It is being used to give the optimiser room to choose a well-behaved solution among the many that fit.

What is genuinely unsettled

This is an area to describe carefully, because it is easy to overstate in either direction.

The empirical phenomenon is solid and reproduced widely. The explanation is not settled. There are rigorous results for specific simplified settings — linear regression in high dimensions, random feature models — and there is no complete theory covering deep networks trained with the optimisers people actually use. "Implicit regularisation by gradient descent" is a description with growing formal support behind it, not a finished proof.

It is also not a licence. Double descent does not mean regularisation is obsolete or that bigger is always better. Weight decay, dropout and early stopping still improve real models, and the second descent requires enough data and enough training; a large model on 300 rows overfits exactly as the classical picture predicts.

What to actually do

For the models in this course — linear, trees, boosting — use the classical picture. It is accurate for them, and gradient boosting genuinely does get worse with too many trees at too high a learning rate. Tune capacity and trust the U.

For neural networks, the practical advice that follows from all of this is: **do not choose a small model out of fear of overfitting.** Take a large one, regularise it, use early stopping against a validation set, and let the measurement decide. The measurement is the authority in both worlds; what changed is that the prior belief "more parameters than rows must be bad" is no longer a safe default.

Why this belongs in a foundations course

Because the most common way to be wrong in this field is to hold a rule whose conditions you never learned. "More parameters than data means overfitting" was taught as a law for decades and is now known to be a description of one regime. Learning where a rule's edge is, rather than only the rule, is what separates someone who can apply a method from someone who can tell when it has stopped applying.

BEFORE YOU MOVE ON

A colleague argues that since a network has more parameters than training examples, it must be memorising and should be shrunk. What is the strongest correction?

A. Parameter count is irrelevant to generalisation, so the observation carries no information at all.

B. Networks are regularised by dropout, so overfitting is impossible regardless of size.

C. The parameter-to-data ratio predicts overfitting only near the interpolation threshold; far beyond it, many perfect fits exist and gradient descent tends to select smooth ones, so measured validation error should decide.

D. The classical result was disproved, so the bias-variance trade-off no longer applies to any model.

CASE STUDY · MODULE 4

The score that fell from 0.88 to 0.62 by being measured properly

An agricultural advisory startup in Pune, eleven people, building pest-outbreak warnings for cotton growers in Vidarbha, four weeks before a funding round.

Kisan Drishti sends a weekly SMS to about 14,000 cotton farmers. The product they were building would upgrade that to a warning: pink bollworm pressure is rising on your plot, scout it this week. The data was three seasons of scouting records from 1,900 plots, each visited fortnightly by a field agronomist who counted larvae in ten randomly chosen bolls, plus weather, sowing date, variety and previous-season history.

The row was one plot-fortnight. The target was whether the next visit's count crossed the economic threshold. About 78,000 rows and 6,300 positives.

The first model, a gradient boosting classifier with a random eighty-twenty split, scored 0.88 AUC. The deck was written. The number went on slide nine.

Their advisor, an agricultural statistician at a state university, asked one question: how many rows do you have per plot?

Forty-one, on average. Three seasons of fortnightly visits.

A random row split scatters a plot's forty-one visits across training and validation. The model does not need to learn anything about pest pressure to score well; it needs to learn that plot 1,183 in Yavatmal tends to cross the threshold, which it can do from the other forty visits sitting in the training set. And every plot in the validation set is a plot the model has seen thirty-odd times.

There was a second problem underneath the first. Pest pressure moves regionally within a season. A random split lets the model train on the third week of August while predicting the second, in the same district, in the same year. Neither of those situations occurs when the product runs, because the product runs forward in time on plots whose future is unknown.

They rebuilt the evaluation properly: group by plot, and split by season, training on seasons one and two and validating on season three.

0.62.

Before accepting it they ran one more check, and it is the cheap one. They permuted the target column, retrained the whole pipeline, and looked at the score. Under the grouped, season-ordered split it collapsed to 0.50, which is what a correct pipeline does. Under the original random split it came back at 0.57 — well above chance on shuffled labels, which is only possible if something in the pipeline is reaching the answer by a route nobody intended. The route was a per-plot aggregate feature, computed across the whole dataset before splitting, which carried a fingerprint of each plot's own outcomes into every one of its rows.

0.62 was not a disaster. The baseline — warn everybody every fortnight during the known risk window — is what the extension service effectively does, and against a like-for-like measurement the model was still adding something. But 0.88 was on slide nine, four weeks out.

The decision the founders faced had two costs and no clean side.

Present 0.62 with the explanation. It is defensible, it is the number the product will deliver, and it invites a conversation about validation design with investors who may or may not want to have it. It also invites a comparison against competitors quoting 0.9-something on random splits, which several of them were.

Present 0.88 with a footnote. Nobody would have caught it in the room. The cost arrives later, when the pilot in a fourth district produces field results nowhere near the slide, at which point the gap has to be explained to people who have already given money.

Delay the round by a season and collect a fourth year on new plots. Cleanest evidence, and the runway did not obviously cover it.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

They presented 0.62, next to the like-for-like baseline of 0.51, and next to the 0.88 with an explanation of why it was wrong. The slide was titled "the number we would have shown you".

Two investors' technical advisors said afterwards that the slide was the reason they recommended proceeding. The round closed smaller than the founders had hoped and at terms they described as fair.

The engineering that followed came out of the same afternoon. They kept a fixed reference set — one full season, one district, never touched — and promoted the label-shuffle check from something they had run once to a standing test in the training pipeline: permute the target, retrain, fail the build if anything scores above chance. It earned its place eight months later, on a change by a new joiner who added a rainfall aggregate computed over the whole dataset instead of inside the fold. Nobody reading that pull request would have caught it, and the build did.

Both changes were made at once. How would you tell how much of the fall came from grouping and how much from time?

Run the two splits separately. Group-by-plot with a random season allocation isolates the entity effect; a season-ordered split that still allows a plot to appear on both sides isolates the temporal effect. Doing so is worth an hour, because the two have different remedies: an entity effect says the model is memorising plots and needs features that generalise across them, while a temporal effect says the relationship moves between seasons and may need recent data or a rolling training window.

Was 0.62 the honest number, or is it now pessimistic in some way?

It is the honest number for the product as described — new plots, a future season — and it may be pessimistic in one respect: it is measured on a single held-out sea-

son, so it carries that season's particular weather. Reporting it with the spread across several forward-chaining splits, rather than one number, would describe the uncertainty better. What it is not is pessimistic because of the grouping; that constraint matches how the product runs.

Why does a label-shuffle test belong in the training pipeline rather than in a checklist?

Because the failure it catches is added by future changes, not by the original author. Permuting the target should collapse every score to chance, and anything above chance means some path in the pipeline reaches the answer. A checklist is run by whoever remembers it; a test in the build runs on the change that a new joiner makes eight months later, which is exactly when this class of bug is introduced.

MODULE 4 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 Training accuracy is 71 per cent and validation accuracy is 70 per cent. A colleague proposes reducing the tree depth to fight overfitting. What is wrong with that?

A. The gap is tiny, so the limit is capacity or features, not memorisation.

B. The gap is tiny, so the model is already at the label ceiling and nothing at all will help.

C. Accuracy is the wrong metric here, so no diagnosis can be made from these two numbers.

D. Depth should only be reduced once the learning rate has been tuned on validation data.

2 A team tunes 200 configurations with five-fold cross-validation and reports the best cross-validated score as its estimate of performance. Why is that number optimistic?

A. Cross-validation always overestimates, because each fold trains on only eighty per cent of the data.

B. The maximum of many noisy estimates is biased upward.

C. The folds share training data, so the scores are correlated and their mean comes out too high.

D. Five folds is too few for 200 configurations, so the estimate is unstable rather than biased.

3 A default label becomes known 30 days after the application. A team uses a time-ordered split with no gap between training and test windows. What does the training window contain that it should not?

A. Rows from after the test period, because expanding windows always overlap at the boundary.

B. Duplicate applications, because an expanding window reuses rows it has already trained on.

C. Applications whose labels were not yet known at the prediction moment.

D. Too little data, because the earliest fold trains on a very short slice of the history.

4 A learning curve shows validation score flat from 40 per cent of the data onward, with the training score far above it. What does that say about collecting more rows?

A. More rows will help, because the training score is still far above the validation score.

B. More rows will help only if they are drawn from a different population than the current ones.

C. The curve cannot answer this question until the model's hyperparameters have been tuned.

D. More rows will not help much; regularise or use fewer features.

5 A 2,000-row validation set gives a standard error of about 0.8 points. After comparing 100 models that are genuinely equivalent, roughly how inflated is the winner's score?

A. About 0.8 points, since that is the standard error of the measurement.

B. About 2.4 points.

C. Not at all, provided none of the hundred models was trained on the validation rows.

D. About 8 points, since the bias grows in proportion to the number of comparisons.

6 A colleague cites double descent to argue that a gradient boosting model with 5,000 trees at a learning rate of 0.3 cannot overfit. What is wrong with the argument?

A. Double descent has never been replicated convincingly, so it should not be cited in the first place.

B. Double descent applies only when a model has fewer parameters than it has training rows.

C. Double descent describes large networks; boosted trees still follow the U.

D. Boosting cannot overfit in any case, so the conclusion happens to be right for the wrong reason.

MODULE 5

Measuring a model honestly

Everything that turns a fitted model into a decision: the confusion matrix worked by hand, the threshold that nobody should leave at 0.5, curves that summarise every threshold at once, whether the probabilities mean anything, whether an improvement is real, and what fairness can and cannot be made to mean mathematically.

> BY THE END OF THIS MODULE YOU CAN
>
> Given a fitted classifier and a stated cost of each kind of error, compute the confusion matrix at a chosen threshold, defend that threshold by expected cost, say whether the model's probabilities are calibrated, decide with an interval whether one model genuinely beats another, and report per-group performance with a defensible fairness criterion and its known trade-offs

---

42 · 9 min

When accuracy lies

> THE ONE THING TO KEEP
>
> Choose the metric that matches what each kind of mistake costs, then set the threshold yourself.

The 99.8% model that does nothing

A payments company in Manila processes 100,000 transactions a day. About 200 are fraudulent.

Here is a model:

```
def predict(transaction):
    return "not fraud"
```

Accuracy: 99.8%. It catches nothing, has no parameters, and beats most first attempts on the headline number. Accuracy is the fraction of rows you got right, and when one class is 99.8% of the rows, that fraction is almost entirely a measurement of the class balance.

Rare events are where the money and the harm are. Fraud, disease, equipment failure, account takeover. Accuracy is close to useless for all of them.

Four cells, not one

A real model flags 400 transactions and catches 120 of the 200 frauds.

	Actually fraud	**Actually fine**
Flagged	120 (true positives)	280 (false positives)
Not flagged	80 (false negatives)	99,520 (true negatives)

Everything you need is in that table.

Precision = 120 / 400 = **30%**. Of what you flagged, this fraction was real. It is the analyst's experience: seven wasted reviews in every ten.

Recall = 120 / 200 = **60%**. Of what was really there, this fraction was caught. It is the victim's experience: 80 frauds went through.

They answer different questions and they trade off. There is no way to read either from accuracy, which here is 99.6% and tells you nothing.

The threshold is yours, not the model's

Most classifiers do not output a class. They output a score between 0 and 1, and something turns that into a decision:

```
flag = model.predict_proba(x)[1] > 0.5
```

The 0.5 is a default someone typed. It is not a property of the model and it is rarely right for an imbalanced problem.

Drop it to 0.2 on the fraud model and you might flag 1,100 transactions, catching 170 of the 200. Recall rises from 60% to 85%. Precision falls from 30% to 15%, so analysts now review 1,100 cases instead of 400.

Nothing was retrained. One number changed. Before you conclude that a model needs to be bigger, move the threshold and look at what the trade actually costs.

Which error costs more

This is a business question with a numeric answer, and someone has to answer it.

- **Cancer screening.** A missed tumour can be fatal; a false alarm means an uncomfortable follow-up scan. Favour recall, heavily.
- **Spam filtering.** A spam message in the inbox is a mild annoyance; a job offer in the spam folder is a serious loss. Favour precision.
- **Card blocking.** A missed fraud costs the issuer the disputed amount. A wrongly blocked card strands a customer abroad. Both are real, so put currency on each and pick the threshold that minimises expected cost.

Write the costs down before you look at a curve. Otherwise the threshold gets chosen by whichever number looked best in the notebook.

The summary metrics, honestly

F1 is the harmonic mean of precision and recall. It is popular, and it is usually the wrong summary, because it declares precision and recall equally important — which is exactly the judgement you should be making deliberately rather than inheriting from a formula.

ROC AUC is the probability that a random positive scores above a random negative. It is threshold-free and useful, but on heavily imbalanced data it flatters: the false-positive rate has 99,800 negatives in its denominator, so 280

false alarms barely register. A model can hold a ROC AUC of 0.95 and still be unusable in practice.

Precision-recall AUC uses precision instead, whose denominator is the flagged set. It shows the pain. On rare-event problems, prefer it.

Calibration is separate from all of these and often what you actually need. A calibrated model saying 0.7 is correct about 70% of the time. Ranking metrics ignore calibration entirely, so a model can rank perfectly and still be badly wrong about probabilities. If a downstream decision multiplies the score by an amount — expected loss, expected revenue — calibration is the property that matters, and you check it by bucketing predictions and comparing the predicted rate with the observed one.

BEFORE YOU MOVE ON

A hospital triage model has 60% recall at its default threshold of 0.5. The clinical team needs to catch more cases. An engineer proposes retraining with a larger model on more data. What should be tried first?

A. Lower the threshold and measure what the extra recall costs in precision.

B. Retrain on more data, since recall is limited by how many cases the model has seen.

C. Oversample the positive class until the two classes are balanced.

D. Switch the reported metric to F1 so recall is properly weighted.

43 · 9 min

The four numbers, worked through

THE ONE THING TO KEEP

Precision divides by what the model said and recall by what was true, and precision alone is not a property of the model — it moves with the base rate of the population you run on.

One thousand transactions

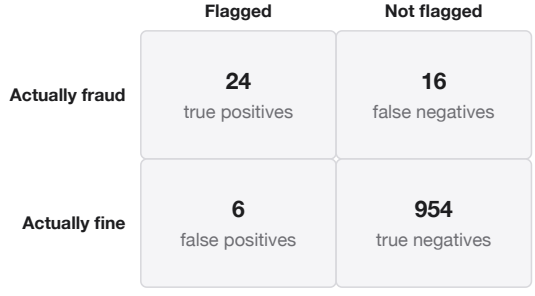
A fraud model runs on 1,000 transactions. 40 are genuinely fraudulent. At a threshold of 0.5 the model flags 30 transactions, and 24 of them are really fraud.

Four numbers describe everything:

```
True positives  (TP) = 24    flagged and fraudulent
False positives (FP) =  6    flagged, actually fine
False negatives (FN) = 16    not flagged, actually fraud
True negatives  (TN) = 954   not flagged, fine
```

Check the arithmetic: 24 + 6 + 16 + 954 = 1,000, and 24 + 16 = 40 actual fraud cases. Everything below is derived from these four.

1,000 transactions, 40 fraudulent, threshold 0.5



Precision 24 / 30 = 80 per cent divides by what the model said — the Flagged column. Recall 24 / 40 = 60 per cent divides by what was true — the top row. Accuracy is 97.8 per cent, against 96 per cent for a model that flags nothing at all.

The metrics, computed

Accuracy = (TP + TN) / all = 978 / 1000 = **97.8%**. And the model that flags nothing scores 96%. Two points of improvement for all that work — this is the accuracy trap from the reuse lesson, in numbers.

Precision = TP / (TP + FP) = 24 / 30 = **80%**. Of what we flagged, 80% was really fraud. This is the number the investigation team cares about, because it determines how much of their day is wasted.

Recall = TP / (TP + FN) = 24 / 40 = **60%**. Of all real fraud, we caught 60%. This is the number the finance team cares about, because 40% of the losses walked past.

F1 = harmonic mean of precision and recall = 2 x (0.8 x 0.6) / (0.8 + 0.6) = **0.686**. The harmonic mean rather than the arithmetic one, because it punishes imbalance: a model with precision 1.0 and recall 0.02 has arithmetic mean 0.51 and F1 of 0.039. F1 refuses to be impressed by one number when the other is terrible.

Specificity = TN / (TN + FP) = 954 / 960 = **99.4%**. Of the innocent, how many we left alone. Standard in medicine, rarer in industry, and the complement of the false positive rate.

The two directions people confuse

Precision and recall are both about the positive class but they divide by different things, and mixing them up is the most common error in this area.

- Precision divides by **what the model said**. Column of the matrix.
- Recall divides by **what was true**. Row of the matrix.

A sentence that fixes it permanently: *precision is how much of my alarm was justified; recall is how much of the danger I noticed.*

Where each one matters

Recall matters most when a miss is catastrophic and a false alarm is merely annoying. Cancer screening: a missed tumour is fatal, a false positive is an

unpleasant follow-up scan. Safety alerts. Content moderation for illegal material.

Precision matters most when acting on a positive is expensive or harmful. Sending a debt collector. Freezing an account. Any automated action against a person. If precision is 20%, four out of every five people you accuse are innocent, and that is a decision about people, not a metric.

Both, at a stated operating point, is the honest report. "At a threshold of 0.62 we get 71% precision and 55% recall" tells someone what will actually happen. "F1 of 0.62" does not, and it hides which side of the trade-off you chose.

The number that is not on this list

Most people asked "the model flagged this transaction — what is the chance it is really fraud?" reach for recall or the model's accuracy. Neither answers it. The answer is precision, and precision depends on the base rate in a way that catches almost everyone.

Run the same model — 60% recall, 99.4% specificity — on a population where fraud is 0.1% rather than 4%. Out of 100,000 transactions, 100 are fraud. The model catches 60. It also flags 0.6% of the 99,900 clean ones, which is 599 false positives. Precision is 60 / 659 = **9%**.

Same model, same sensitivity, same specificity. Precision fell from 80% to 9% because the base rate fell. This is the base rate fallacy, it is why a rare-disease screening test with excellent sensitivity still produces mostly false positives, and it is the single most important thing on this page.

The same model where fraud is 0.1 per cent of 100,000 transactions

	Flagged	Not flagged
Actually fraud	**60** caught	**40** missed
Actually fine	**599** false alarms	**99,301** left alone

Recall is still 60 per cent and specificity still 99.4 per cent. Precision is now 60 / 659 = 9 per cent, because 0.6 per cent of 99,900 clean transactions is 599 false alarms. Precision belongs to the population the model runs on, not to the model.

So whenever someone quotes a precision figure, ask what the prevalence was in the set it was measured on. If your production prevalence differs, the precision you were promised does not transfer.

Getting it out of the library

```
from sklearn.metrics import confusion_matrix, classifica-
tion_report
tn, fp, fn, tp = confusion_matrix(y_true, y_pred).ravel()
print(classification_report(y_true, y_pred, digits=3))
```

Note the order that `ravel()` gives you — `tn, fp, fn, tp` — which is not the order most people expect and is worth checking the first few times rather than assuming.

BEFORE YOU MOVE ON

A screening model with 90% sensitivity and 95% specificity is moved from a high-risk clinic (10% prevalence) to general population screening (0.5% prevalence). What happens to the chance that a positive result is a true case?

A. It falls sharply — from about 67% to about 8% — because the far larger healthy population generates many more false positives even at an unchanged 5% false positive rate.

B. It stays the same, since sensitivity and specificity are properties of the test and do not change with population.

C. It rises, because a lower prevalence means fewer diseased people to miss, so the positives that do appear are more reliable.

D. It cannot be determined without knowing the model's accuracy on the new population.

44 · 9 min

The threshold is a business decision, not a default

THE ONE THING TO KEEP

The model produces a score and the threshold converts it into a decision; choose the threshold from the relative cost of the two errors, the capacity to act, or a stated requirement — never from the library default.

Where 0.5 came from

Nowhere useful. `predict()` applies 0.5 because a library needs a default, not because 0.5 is right for your problem. It is right only in the narrow case where the two error types cost the same and the classes are balanced, which describes almost nothing.

Everything else in this lesson follows from separating two things that beginners treat as one: **the model produces a score; the threshold turns it into a decision.** They are tuned separately, by different people, for different reasons, and the threshold can be changed at any time without retraining.

Moving it, and watching what happens

Take the fraud model from the last lesson and sweep the threshold:

```
import numpy as np
from sklearn.metrics import precision_score, recall_score
p = model.predict_proba(X_val)[:, 1]
for t in np.arange(0.1, 0.95, 0.05):
    pred = (p >= t).astype(int)
    print(f"{t:.2f}  precision {precision_score(y_val,
pred):.3f}"
          f"  recall {recall_score(y_val, pred):.3f}")
```

The pattern is always the same. **Lower threshold: more flags, higher recall, lower precision. Higher threshold: fewer flags, higher precision, lower recall.** There is no setting that improves both; that would require a better model.

A typical run might give you: at 0.20, precision 0.41 and recall 0.88. At 0.50, precision 0.80 and recall 0.60. At 0.75, precision 0.94 and recall 0.31. Those are three different products.

Choosing by expected cost

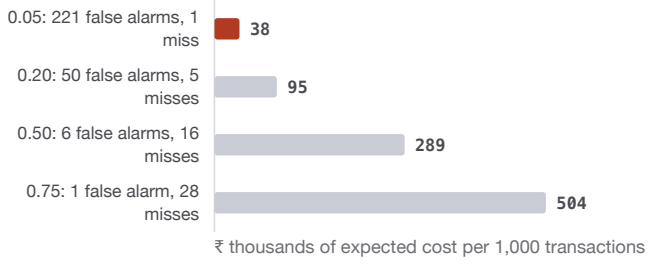
The principled method. Assign a cost to each error type and pick the threshold that minimises total expected cost.

Suppose a missed fraud costs ₹18,000 on average, and a false alarm costs ₹90 of review time. Then:

```
cost = 18000 * fn_count + 90 * fp_count
```

Evaluate that across the threshold sweep and take the minimum. Because a miss costs 200 times a false alarm, the optimal threshold lands far below 0.5 — probably nearer 0.05 — and the model will flag many more transactions than it did at the default.

Total cost of mistakes at four thresholds



A miss costs ₹18,000 and a false alarm ₹90, so the minimum sits far below the library default and the model flags many more transactions than it did at 0.5. Change the 200-to-1 ratio and the winner moves, which is the conversation worth having with whoever owns those costs.

For a well-calibrated model there is a closed form. The optimal threshold is

```
t = cost(FP) / (cost(FP) + cost(FN))
```

With 90 and 18,000 that gives `90 / 18090 = 0.005`. Note the condition: *well-calibrated*. If your probabilities do not mean what they say, this formula gives you a threshold on a scale that does not exist, which is the subject of a later lesson in this module.

Choosing by capacity

Often the constraint is not cost but throughput. The fraud team can review 200 cases a day. Then the threshold is whatever produces 200 flags, and the metric is precision at 200 — how many of that day's investigations were worthwhile.

This reframing is usually closer to how the organisation actually works, and it changes what you optimise. You no longer care about the model's behaviour across the whole score range, only about the ranking at the top. A model that is better at the very top and worse elsewhere is the better model here, and average metrics will not tell you that.

Choosing by a fixed requirement

Sometimes one side is dictated. "We must catch at least 95% of cases" from a regulator, or "no more than 2% of legitimate users may be blocked" from a product owner. Then set the threshold to satisfy the constraint on validation data and report what the other side costs you.

```
from sklearn.metrics import precision_recall_curve
prec, rec, thr = precision_recall_curve(y_val, p)
i = np.argmax(rec >= 0.95)     # first threshold meeting the
requirement
print(thr[i], prec[i])
```

Three failures worth avoiding

Choosing the threshold on the test set. It is a fitted parameter like any other. Choose it on validation data, then measure the chosen threshold's performance on test.

Assuming it transfers. A threshold tuned on last quarter's data assumes this quarter has the same score distribution. When the population shifts, the same threshold produces a different flag rate — often the first symptom of drift anyone notices, and a good thing to monitor.

Reporting only the number at 0.5. If you report "precision 0.80, recall 0.60" without saying which threshold produced it, you have reported one arbitrary point on a curve and hidden the rest.

The output that is not a decision

A final option that is often the best one: do not threshold at all. Show the score.

For a triage queue, a ranked list is more useful than a binary flag, because the human at the other end applies judgement that the threshold would have thrown away. For a risk score shown to an underwriter, a band — low, medium, high — carries more than a yes or no. Forcing a continuous output into two buckets destroys information, and the only reason to do it is that some

downstream system requires a decision. When no such system exists, do not invent one.

BEFORE YOU MOVE ON

A team wants to minimise expected cost where a false negative costs 40 times a false positive. Their model's probabilities are well calibrated. Roughly where should the threshold sit and why?

A. At 0.5, since a calibrated model is already optimal at the default threshold.

B. Around 0.025, because the optimal threshold is cost(FP) divided by the sum of both costs, so a much cheaper false positive pushes the cut far down.

C. Around 0.975, because the expensive error should require a high level of confidence before the model commits.

D. Around 0.4, since the threshold should move modestly from the default in proportion to the log of the cost ratio.

---

45 · 9 min

ROC and precision-recall: every threshold at once

THE ONE THING TO KEEP

AUC is the probability that a random positive outscores a random negative, so it measures ranking only and is insensitive to a large absolute number of false positives when negatives vastly outnumber positives — which is why imbalanced problems should be reported with average precision.

Two curves, two questions

Rather than pick one threshold, plot the model's behaviour at all of them.

The ROC curve plots true positive rate (recall) against false positive rate. It sweeps from the top-right corner — flag everything, catch everything, alarm on everything — to the bottom-left, where you flag nothing. A diagonal line is a model with no information.

The precision-recall curve plots precision against recall over the same sweep. Its baseline is not a diagonal but a horizontal line at the positive class's prevalence, which is what a random model achieves.

```
from sklearn.metrics import roc_auc_score, average_precision_s-
core
roc_auc_score(y_val, p)
average_precision_score(y_val, p)
```

What AUC actually means

The area under the ROC curve has an interpretation that is far more useful than "area under a curve":

> ***AUC is the probability that a randomly chosen positive case gets a higher score than a randomly chosen negative case.***

So 0.5 is a coin flip, 1.0 is perfect ranking, and 0.83 means that if you pick one fraud and one legitimate transaction at random, the fraud scores higher 83% of the time.

That framing tells you what AUC is for and what it is not. It is entirely about **ranking**, so it is unchanged if you apply any monotonic transformation to the scores — a model whose probabilities are all badly wrong can still have an AUC of 0.95, as long as the order is right. AUC says nothing about calibration, and it does not depend on a threshold, which is both its convenience and its limitation.

Why AUC misleads on rare positives

The false positive rate divides by the number of negatives. When negatives outnumber positives 1,000 to 1, a large absolute number of false positives is a small false positive rate, and the ROC curve barely notices.

Concretely: 100,000 transactions, 100 fraudulent. Your model flags 900 legitimate transactions as fraud. That is a false positive rate of 0.9% — the ROC curve looks excellent. It is also 900 investigations for at most 100 real cases, so precision is under 10% and the fraud team's day is ruined.

The precision-recall curve shows this immediately, because precision divides by the number flagged rather than by the number of negatives. **For imbalanced problems, report average precision.** Report AUC too if you like, but do not let it be the headline.

A useful reference point: on a dataset with 1% positives, average precision of 0.30 is a genuinely strong model — it is thirty times the random baseline of 0.01 — even though 0.30 reads like a bad number to anyone used to accuracy.

Reading the shape, not just the area

Two models can have identical AUC and completely different curves. One is excellent at the high-score end and mediocre elsewhere; the other is uniformly average. If you are going to operate at high precision — investigating the top 200 — the first is much better and the summary number cannot tell them apart.

So look at the curve in the region you will actually operate in. A model chosen by AUC alone is a model chosen by its average behaviour across thresholds you will never use.

Partial AUC and top-k metrics

When the operating region is known, measure it directly.

- **Partial AUC** restricts the area to a range of false positive rates, say 0 to 0.1.
- **Precision at k** — of the top 200 by score, how many are positive. This is the metric for any queue with fixed capacity.
- **Recall at k** — of all positives, how many appear in the top 200. This is the same measurement viewed from the other side, and it is the one to quote when the question is coverage.

These are less standard and far more decision-relevant than either curve's area.

The comparison trap

AUC computed on two different datasets is not comparable, and this catches people constantly. A model scoring 0.91 on one population and 0.84 on another has not necessarily got worse; the second population may simply be harder, or have a different mix of easy and hard cases. Comparisons are only meaningful on the same evaluation set.

The same applies across time. A model whose AUC drops from 0.88 to 0.85 between quarters may be degrading, or the quarter may have contained a different mix of customers. Investigate before concluding, and keep a fixed reference set that you rescore each period precisely so that you have one comparison that is genuinely like-for-like.

BEFORE YOU MOVE ON

Two models have identical ROC AUC of 0.88 on a 1% positive dataset, but model A has average precision 0.34 and model B has 0.11. What does this tell you?

A. One of the two metrics has been computed incorrectly, since both summarise the same ranking.

B. Model B is better calibrated, which is why its precision-based metric is lower.

C. Model A ranks the true positives much higher within the top of the list, which average precision rewards and ROC AUC largely averages away across the vast negative class.

D. Model A must have been evaluated at a different threshold, producing the higher score.

46 · 9 min

Does 0.7 mean seventy per cent?

THE ONE THING TO KEEP

Calibration asks whether a predicted 0.7 corresponds to 70% of cases occurring, is separate from ranking quality, and is fixed by a monotonic post-processing step that leaves AUC unchanged.

The question nobody asks

Your model outputs 0.7 for a thousand customers. Do about 700 of them convert?

If yes, the model is **calibrated** and the number is a probability you can compute with. If no — if 400 convert — then 0.7 is a ranking score wearing a probability's clothes, and every expected-value calculation built on it is wrong.

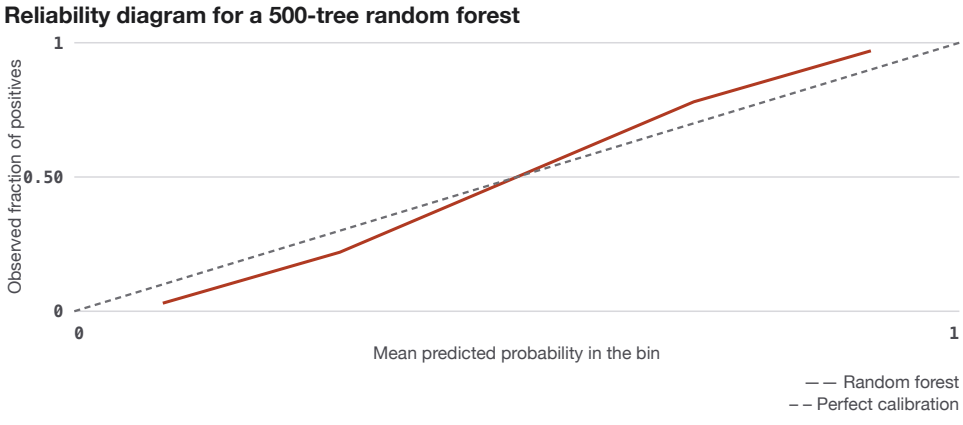
This matters whenever the output feeds arithmetic rather than a simple ordering: expected revenue, expected loss, an insurance premium, a threshold chosen by cost ratio, or any case where a human reads the number and forms a belief.

Measuring it

Bin the predictions and compare the mean predicted probability in each bin with the actual observed rate.

```
from sklearn.calibration import calibration_curve
prob_true, prob_pred = calibration_curve(y_val, p, n_bins=10,
                                         strategy="quantile")
```

Plot `prob_true` against `prob_pred`. Perfect calibration is the diagonal. Below it means over-confident — you predicted 0.8 and observed 0.6. Above it means under-confident.



The forest rarely reaches 0 or 1, because all 500 trees seldom agree, so its curve is steeper than the diagonal: a predicted 0.9 comes true 97 per cent of the time and a predicted 0.1 only 3 per cent. Platt or isotonic scaling straightens the curve without changing a single ranking.

Use `strategy="quantile"` rather than uniform bins. With uniform bins on an imbalanced problem, nine of the ten bins contain almost no data and the plot is mostly noise.

The summary number is the **Brier score**, the mean squared error of the probabilities. It combines calibration and discrimination into one figure, so a poor Brier score does not tell you which is at fault, but it is a reasonable single metric when you need one.

Which models are calibrated, and why

Logistic regression is well calibrated by construction. It minimises log loss, and log loss is what statisticians call a proper scoring rule — it is minimised exactly when the predicted probabilities equal the true ones. The model has no incentive to distort them.

Naive Bayes is badly calibrated. It assumes features are independent, which they are not, so it multiplies correlated evidence repeatedly and ends up pushing predictions towards 0 and 1. Its ranking can be perfectly good while its probabilities are almost all above 0.99 or below 0.01.

Random forests are under-confident at the extremes. A forest's prediction is the fraction of trees voting positive, and for a case that is genuinely

certain, getting all 500 trees to agree is unlikely, so predictions rarely reach 0 or 1. The output tends to bunch between about 0.1 and 0.9.

SVMs produce distances, not probabilities. Whatever `predict_proba` returns for an SVM comes from a calibration step bolted on afterwards.

Boosted trees are usually reasonable but drift with more trees. Training past the point of best log loss tends to push probabilities outward.

Deep networks are typically over-confident, and increasingly so as they get larger — a finding from Guo and colleagues in 2017 that has held up. Modern networks routinely predict 0.99 on cases they get wrong.

Fixing it

Calibration is a post-processing step, fitted on data the model did not train on.

```
from sklearn.calibration import CalibratedClassifierCV
cal = CalibratedClassifierCV(base_model, method="isotonic",
cv=5)
cal.fit(X_train, y_train)
```

Platt scaling (`method="sigmoid"`) fits a one-parameter logistic function mapping scores to probabilities. Two parameters, so it works on small calibration sets, and it can only apply an S-shaped correction.

Isotonic regression (`method="isotonic"`) fits any non-decreasing function. More flexible, needs more data — a few thousand rows at least — and it overfits on small sets, producing a staircase that memorises the calibration data.

Rule of thumb: under about 1,000 calibration rows, use sigmoid; above, isotonic is usually better.

One property worth knowing: both methods are monotonic, so **calibration cannot change the ranking and therefore cannot change AUC.** It changes the numbers and leaves the order alone. If someone reports that calibration improved their AUC, something else changed too.

What breaks calibration in production

Class weighting and resampling. Both deliberately misrepresent the base rate to the model, and the probabilities come out shifted. This is the side effect flagged in module 3, and it is why threshold-moving is preferable when you need meaningful probabilities.

Base rate shift. A model calibrated when fraud was 1% is over-predicting when fraud falls to 0.4%. The ranking may be untouched; the probabilities are all wrong. This is the most common cause in practice and the fix is recalibration on recent data, not retraining.

Time. Calibration decays faster than discrimination. A model can keep its AUC for a year while its probabilities become unusable in three months. Monitor both separately; they fail on different schedules and for different reasons.

BEFORE YOU MOVE ON

A random forest has excellent AUC but its predicted probabilities cluster between 0.15 and 0.85, never reaching the extremes. An analyst multiplies all probabilities by 1.4 to spread them out. What is wrong with that fix?

A. Multiplying is monotonic, so it cannot change the AUC, which means the fix does nothing at all.

B. It pushes values above 1 and applies one linear stretch everywhere, whereas the miscalibration is a non-linear compression that needs a fitted mapping such as isotonic regression on held-out data.

C. It would work, but only if applied to the log-odds rather than the probabilities.

D. Random forest probabilities cannot be calibrated at all, because they are vote fractions rather than probability estimates.

47 · 8 min

Metrics for a number, and what each one hides

THE ONE THING TO KEEP

Report MAE and RMSE together because their gap measures how uneven the errors are, and plot residuals against predictions — the shape of the failure carries information no summary metric retains.

Four metrics, four different claims

When the target is a number rather than a class, the metric choice is quieter but no less consequential.

MAE — mean absolute error. Average size of the error, in the target's units. Interpretable directly: "we are off by 4,200 rupees on average". Treats every rupee of error the same.

RMSE — root mean squared error. Also in the target's units, but errors are squared before averaging, so large errors dominate. RMSE is always at least as large as MAE, and the *gap between them tells you how uneven your errors are*. RMSE 12,000 with MAE 4,200 means a few very large errors; RMSE 4,600 with MAE 4,200 means errors of consistent size. Reporting both is more informative than reporting either.

MAPE — mean absolute percentage error. Scale-free, so it compares across products or regions. Two serious defects: it is undefined when the true value is zero, and it is asymmetric — over-predicting is capped at 100% error while under-predicting is unbounded, so a model tuned on MAPE learns to under-predict systematically. Widely used in demand forecasting and widely criticised there for exactly this.

R^2 — the fraction of the target's variance the model explains. 0 means you match the mean; 1 means perfect; negative means worse than predicting the mean, which happens more often than people expect on held-out data. Its weakness is that it depends on the variance of the evaluation set, so an R^2 of

0.9 on a varied population and 0.3 on a narrow one can come from the same model with the same absolute errors.

Choosing

Ask what an error costs.

- Cost proportional to the size of the error: **MAE**.
- Large errors disproportionately bad: **RMSE**.
- A 10% error means the same at every scale: **MAPE**, or better, fit on the log of the target and use RMSE there, which gives you the same intent without MAPE's asymmetry.
- Explaining to a non-technical audience how much better than nothing you are: **R^2**, alongside the baseline.

And remember from module 1 that the metric and the training loss need not match. Train on Huber for robustness and report MAE; there is nothing wrong with that as long as you say so.

The metric hides the shape

Every one of these is an average, and averages conceal structure. Two diagnostics take a minute each and are worth more than the summary numbers.

Plot residuals against predictions. What you want is a formless cloud centred on zero. What you often get:

- A funnel widening to the right — errors grow with the prediction, which means a log transform on the target is likely to help.
- A curve — the model has systematic bias in a region, which means a missing non-linearity.
- A slope — the classic regression-to-the-mean pattern, where the model over-predicts low values and under-predicts high ones.

Plot the error distribution. A long tail on one side tells you the model fails asymmetrically, which matters enormously if the two directions have different costs. Under-predicting demand means a stockout; over-predicting means

holding stock. Those are not the same problem, and no symmetric metric distinguishes them.

Quantile loss, for when you need a range

Sometimes a single number is the wrong output. "Delivery in 34 minutes" is a promise; "90% of deliveries arrive within 48 minutes" is a promise you can keep.

Quantile regression fits a chosen quantile rather than the mean, by using an asymmetric loss that penalises under-prediction and over-prediction differently.

```
from sklearn.ensemble import GradientBoostingRegressor
GradientBoostingRegressor(loss="quantile", alpha=0.9)
```

Fit three models at 0.1, 0.5 and 0.9 and you have a median with an interval around it. For anything where a customer is given a commitment, this is usually more useful than a point estimate, and it is the honest way to express what the model does and does not know.

Two mistakes to avoid

Comparing RMSE across different target scales. RMSE of 400 on house prices and 400 on delivery minutes are not comparable numbers. Normalise, or use R^2, or report against a baseline.

Averaging percentage errors across items of wildly different size. One tiny item with a large percentage error can dominate a MAPE computed over a thousand products. Weight by volume or value if the business cares about total error rather than average per-item error, and say which you did.

BEFORE YOU MOVE ON

A demand model is tuned to minimise MAPE and consistently under-forecasts, causing stockouts. What property of MAPE explains the direction of the bias?

A. MAPE is undefined at zero, so days with no demand were dropped, biasing the remaining sample downward.
B. MAPE weights every item equally regardless of volume, so small-volume items dominate and pull forecasts down.
C. MAPE divides by the actual value, so it is asymmetric: over-predicting can cost at most 100% while under-predicting is unbounded — so the optimiser learns that forecasting low is the safer error.
D. MAPE is a squared metric, so it over-penalises large errors and the model shrinks all forecasts toward zero to avoid them.

48 · 9 min

Is that improvement real?

THE ONE THING TO KEEP

Bootstrap the difference between two models on the same resampled rows and report the interval; a t-test across cross-validation folds violates independence and declares significance far too often.

The situation

Model A scores 0.847. Model B scores 0.852. Do you ship B?

The wrong answer is "yes, it is higher". Both numbers were measured on a finite sample, both carry uncertainty, and 0.005 may be well inside it.

The bootstrap, which is all you really need

Resample your evaluation set with replacement, recompute the metric, repeat a thousand times. The spread of those thousand values is the sampling distribution of your score.

```
import numpy as np
rng = np.random.default_rng(0)
n = len(y_test)
diffs = []
for _ in range(2000):
    idx = rng.integers(0, n, n)
    diffs.append(roc_auc_score(y_test[idx], pB[idx])
                 - roc_auc_score(y_test[idx], pA[idx]))
lo, hi = np.percentile(diffs, [2.5, 97.5])
print(f"difference {np.mean(diffs):.4f}  95% CI [{lo:.4f},
{hi:.4f}]")
```

One detail carries most of the value: **resample the rows once and score both models on the same resampled rows.** This is a paired comparison, and it removes the variance caused by some rows simply being harder. An unpaired comparison of two intervals is far less sensitive and will tell you "inconclusive" when the paired version would have given you a clear answer.

If the interval on the difference contains zero, you do not have evidence that B is better. That is not the same as evidence they are equal; it means your test set cannot resolve a difference this small.

McNemar's test, for two classifiers on the same data

When comparing predicted labels rather than scores, the relevant information is only in the rows where the two models disagree.

- `b` = rows A got right and B got wrong.
- `c` = rows A got wrong and B got right.

Rows where both were right or both were wrong tell you nothing about which is better. Under the hypothesis that the models are equally good, `b` and `c`

should be about equal, and McNemar's test asks how surprising the observed imbalance is.

```
from statsmodels.stats.contingency_tables import mcnemar
mcnemar([[both_right, b], [c, both_wrong]], exact=True)
```

If B beat A on 60 rows and lost on 45, that is a 60-45 split out of 105 disagreements — nowhere near significant. If it is 60-20, it is. The absolute accuracy difference can look identical in both cases, which is why the disagreement counts are the thing to look at.

Cross-validation is not a free significance test

A tempting move: run 10-fold cross-validation on both models and run a t-test on the ten pairs of scores.

Do not. The folds share training data, so the ten differences are correlated, and the standard t-test assumes independence. Dietterich showed in 1998 that this procedure has a badly inflated false positive rate — it declares differences significant far more often than it should. Repeated cross-validation makes it worse, not better, because the extra repeats are even more correlated.

What to do instead: use a held-out test set and bootstrap it, which is what the first section does. Or report the fold-to-fold spread as a description of stability without attaching a p-value to it.

How large a difference can you even see?

Work it out before running the comparison. On a 2,000-row test set with accuracy near 85%, the standard error is about 0.8 points, and the standard error on a *paired difference* is smaller — often around 0.3 to 0.5 points depending on how correlated the models are.

So a 0.2-point improvement is not measurable on that set, no matter how many times you rerun it. Knowing this in advance saves you from a week of tuning aimed at a difference your evaluation cannot detect.

Practical significance

The last step is the one that gets skipped. Suppose the improvement is real: 0.847 to 0.852, statistically solid on a large test set.

Is it worth shipping? Model B is a stacked ensemble that takes four hours to train, 300ms to serve instead of 12ms, and cannot be explained to a regulator. The gain is 0.5% of AUC, which might translate to a handful of extra fraud cases caught per month.

Statistical significance is a statement about whether an effect exists. Whether it matters is a separate judgement involving latency, cost, maintenance and explainability, and the person who can hold both is considerably more useful than the one who can only compute the first.

BEFORE YOU MOVE ON

Two classifiers are compared on 5,000 test rows. They agree on 4,700. Of the 300 disagreements, model B is right on 170 and model A on 130. What does McNemar's framing say?

A. B is clearly better, since it wins 57% of the contested cases across a large test set.

B. The comparison is invalid because the 4,700 agreements should also be counted, and they show the models are near-identical.

C. A 170-130 split of 300 disagreements is within the range chance produces, so the test set does not provide evidence that B is better despite the 0.8-point accuracy gap.

D. McNemar's test cannot be applied because the models were evaluated on the same rows rather than independent samples.

49 · 8 min

From a probability to a decision worth making

THE ONE THING TO KEEP

The break-even threshold is the cost of a false positive divided by the sum of both error costs, it can and usually should be computed per row from that row's stakes, and the arithmetic is only valid for whoever's costs were put into it.

The last step nobody teaches

A model outputs 0.34. What do you do?

Every lesson so far has been about producing and checking that number. This one is about the arithmetic that turns it into an action, and it is short because the arithmetic is simple. What is not simple is getting the costs.

Expected value

For each available action, compute the expected outcome under the model's probability, and take the best one.

A lender considers approving a ₹200,000 loan. If repaid, profit is ₹24,000. If defaulted, loss is ₹150,000. The model says the default probability is 0.09.

```
Expected value of approving = 0.91 * 24,000 − 0.09 * 150,000
                            = 21,840 − 13,500 = +8,340
Expected value of declining = 0
```

Approve. And the break-even default probability is where the two are equal:

```
p * 150,000 = (1 − p) * 24,000
p = 24,000 / 174,000 = 0.138
```

Approve below 13.8%, decline above. That is the threshold, derived from the business rather than chosen by a library, and it is defensible in a room full of people who do not know what AUC is.

Costs are usually per-row, not global

The fraud example in the threshold lesson used one average cost per miss. Real costs vary by row: missing a ₹500 fraud and a ₹5,00,000 fraud are not the same event.

So compute the threshold per row:

```
threshold = review_cost / (review_cost + transaction_amount)
flag = p_fraud > threshold
```

A ₹90 review against a ₹5,00,000 transaction gives a threshold of 0.00018 — flag on almost any suspicion. Against a ₹500 transaction it gives 0.15 — only flag if reasonably confident. One line of code, and it typically outperforms a globally tuned threshold by a wide margin, because it is using information the global threshold throws away.

This is the most under-used idea in applied classification. It also requires the probabilities to be calibrated, which is why that lesson came first.

More than two actions

Most real systems have three or more: approve, decline, and refer to a human. Then you have two thresholds, and the middle band is where the model defers.

The width of that band is set by review capacity and by the value of a human decision. If a reviewer is right 95% of the time and costs ₹200, you can compute how wide the band should be, and it turns out to depend on how much the model's uncertainty in that region actually costs you.

This "reject option" framing — a classifier allowed to abstain — is standard in medical decision support and in credit, and it is usually a better design than forcing a binary decision. The metric that goes with it is the **coverage-accu-**

racy curve: accuracy on the cases the model chose to decide, plotted against the fraction it decided. A model that is 99% accurate on the 60% of cases it is confident about, and defers the rest, is a genuinely useful product even if its overall accuracy is unremarkable.

Where the costs come from

The hard part is not the arithmetic. It is that nobody has written the costs down.

Ask anyway, and ask badly rather than not at all. "If we wrongly decline a good customer, roughly what does that cost us?" produces an argument, and the argument is the useful output — it surfaces that the marketing team values a customer's lifetime and the risk team values this quarter's losses, and that they have never had to reconcile those numbers before.

When a cost is genuinely unknowable, use a ratio instead of an absolute. "A missed fraud is roughly 200 times a false alarm" is easier to agree on than two rupee figures, and it is all the threshold formula needs.

The costs that do not go in the spreadsheet

Some consequences are not priced, and pretending otherwise is how models cause harm.

A wrongly declined loan may cost the lender ₹24,000 of foregone profit. It may cost the applicant a business that does not open. Those two numbers are not on the same scale and cannot be added.

When the cost falls on a person outside the organisation, expected-value arithmetic optimises the organisation's outcome and is silent about theirs. This is not an argument against doing the arithmetic; it is an argument for stating whose costs are in the sum and whose are not, and for treating some errors as constraints rather than terms. Some things belong in the model as a hard limit — "never decline without a stated reason", "never act automatically above this amount" — precisely because they should not be traded off against anything.

BEFORE YOU MOVE ON

A retailer flags orders for manual review at a fixed probability threshold. Someone proposes computing the threshold per order as review_cost / (review_cost + order_value). Why does this usually outperform the fixed threshold?

A. It removes the need for calibrated probabilities, since the threshold now adapts to each order.

B. It uses each order's own stakes, so cheap orders need strong evidence before a review is worth it while expensive ones justify a review on faint suspicion — information a single global cut discards.

C. It increases the total number of reviews, which raises recall across the board.

D. It makes the classifier cost-sensitive during training, so the model itself learns to weight expensive orders more.

50 · 10 min

Fairness as arithmetic: the definitions and what each one demands

THE ONE THING TO KEEP

Demographic parity, equalised odds and within-group calibration are three precise and different demands, dropping the protected column removes your ability to measure disparity without removing the disparity, and every one of these criteria is silent about bias already inside the labels.

Moving from principle to number

Everybody agrees a model should be fair. The agreement dissolves the moment you write down what that means numerically, and it dissolves in a specific, well-understood way that is worth knowing precisely.

Take a hiring screen with two groups, A and B. Here are three definitions, all reasonable, all in use.

Demographic parity

> *The model selects the same proportion of each group.*

```
P(predicted positive | group A) = P(predicted positive | group
B)
```

This is the criterion behind the four-fifths rule used in US employment discrimination screening: if group B's selection rate is under 80% of group A's, that is treated as evidence of adverse impact worth investigating.

What it demands: equal outcomes at the group level, regardless of anything else.

What it ignores: whether the groups differ in whatever the model is predicting. If they genuinely differ, enforcing parity means using different effective standards, which is exactly the objection its critics raise and exactly the point its defenders make.

Equalised odds

> *The model is equally accurate for each group, in both directions.*

```
P(predicted positive | actually positive, group A)
  = P(predicted positive | actually positive, group B)      #
equal recall
P(predicted positive | actually negative, group A)
  = P(predicted positive | actually negative, group B)      #
equal false positive rate
```

This is the criterion that COMPAS, a US recidivism prediction tool, was found to violate in the 2016 ProPublica analysis. Black defendants who did not reoffend were labelled high-risk at roughly twice the rate of white defendants who did not reoffend — unequal false positive rates.

Equal opportunity is the weaker version requiring only equal recall, on the argument that failing to identify a qualified candidate is the harm that matters most.

Calibration within groups

> *A predicted 0.7 means the same thing whichever group you are in.*

```
P(actually positive | predicted 0.7, group A)
  = P(actually positive | predicted 0.7, group B)
```

This is what Northpointe, the company behind COMPAS, argued in response: their scores were calibrated within groups, so a score of 7 meant the same recidivism rate for everyone. That claim was also true.

Both analyses were arithmetically correct. They measured different things.

Measuring it

```
def group_metrics(y_true, y_pred, group):
    for g in np.unique(group):
        m = group == g
        tn, fp, fn, tp = confusion_matrix(y_true[m],
y_pred[m]).ravel()
        print(f"{g}: rate {(tp+fp)/m.sum():.3f} "
              f"recall {tp/(tp+fn):.3f} fpr {fp/(fp+tn):.3f} "
              f"precision {tp/(tp+fp):.3f}  n={m.sum()}")
```

Print `n` per group, always. A recall of 0.41 computed on 23 people is not a finding; it is a wide interval. The most common error in fairness reporting is treating a small-group number as though it had the precision of a large-group one, and the fix is to bootstrap the per-group metrics and show intervals.

The free library for this work is `fairlearn`, which computes these metrics and implements several mitigation methods.

Removing the protected attribute does not work

The first instinct is to drop the column. It fails, reliably, and the mechanism is worth understanding rather than accepting.

The protected attribute is encoded in other features. Postcode correlates with ethnicity in most countries with residential segregation. First name correlates with gender and often ethnicity. Which school, which sports, which employer, which shopping patterns. A model with 60 features has many routes to reconstruct what you deleted, and it will use them because they are predictive.

Worse, removing the column removes your ability to *measure* the disparity. The model still discriminates and you can no longer see it. This is why several regulators now require collecting protected attributes specifically for auditing, held separately from the modelling data — a distinction between using an attribute for prediction and using it for measurement that is important and frequently misunderstood.

Three places to intervene

Pre-processing — reweight or transform the training data to remove the association. Model-agnostic, and you may lose accuracy you cannot account for.

In-processing — add a fairness constraint to the training objective. Most effective, requires a model you can modify.

Post-processing — use different thresholds per group to equalise a chosen metric. Simple, effective, and legally fraught: in several jurisdictions applying a different decision threshold by race or sex is itself unlawful, regardless of intent. This is a real tension between what the mathematics recommends and what the law permits, it differs by country, and it is genuinely unresolved. It is also not something to resolve from a lesson — the point here is to know that the tension exists before you build something that runs into it.

The limit of all of this

Every definition above compares model outputs against recorded labels. If the labels themselves encode past discrimination — arrests rather than crimes,

promotions granted by biased managers, loans approved by a biased process — then a model that is perfectly fair by every metric on this page faithfully reproduces the injustice in the labels.

No amount of arithmetic detects that. It requires knowing how the labels were made, which is a question for the people who made them.

BEFORE YOU MOVE ON

A lender removes ethnicity from its model and reports that the model 'cannot discriminate because it never sees the attribute'. What is the flaw?

A. The model still needs ethnicity to satisfy demographic parity, so removing it makes the model less fair by construction.

B. Correlated features such as postcode, name and employer reconstruct the attribute, so disparities persist — and without the column the lender can no longer measure them.

C. Removing a feature always reduces accuracy, and a less accurate model produces more errors in every group.

D. Legally, ethnicity must be retained in credit models in most jurisdictions, so the removal itself is non-compliant.

51 · 8 min

Why you cannot have all three

THE ONE THING TO KEEP

When two groups have different base rates, calibration within groups and equal error rates across groups are mathematically incompatible, so fairness requires an explicit and documented choice of which to satisfy.

A theorem, not an opinion

The previous lesson gave three fairness criteria. This one gives the result that makes choosing between them unavoidable.

> *If two groups have different base rates for the outcome, then no classifier can simultaneously satisfy calibration within groups and equal false positive and false negative rates — except a perfect classifier, or one that has no predictive power at all.*

This was proved independently by Kleinberg, Mullainathan and Raghavan, and by Chouldechova, both in 2016. It is a mathematical result. It does not depend on the algorithm, the data quality, the features, or how careful you are.

Seeing why

The intuition takes a paragraph.

Calibration says: among everyone the model scored at 0.7, the same fraction are actually positive, in both groups. Now suppose group A has a 30% base rate and group B has 10%.

To be calibrated for group B, whose members are less likely to be positive on average, the model has to spread B's scores differently — a given score has to be reachable by the same fraction of true positives in each group. Doing that while also holding the false positive rate equal across groups is over-determined. The base rate appears in both requirements, pulling in different direc-

tions, and the only ways out are a model that gets everything right (no errors to distribute) or a model that predicts nothing (nothing to distribute).

You can verify this yourself with a spreadsheet: fix two base rates, fix a false positive rate, and try to satisfy calibration at every score band. It does not close.

COMPAS, again

This is exactly what happened in the recidivism argument. ProPublica showed unequal false positive rates. Northpointe showed calibration within groups. Base rates in the recorded data differed between groups.

Given that difference, both findings had to be true simultaneously. The theorem says so. Neither party was making an error, and neither could have satisfied the other without abandoning their own criterion.

What the debate should have been about — and largely became, afterwards — is which criterion the situation demands, and whether the recorded base rate difference reflects the world or reflects policing patterns. The second question is not a statistical one at all.

What this means for your work

Three consequences, and they are practical.

You must choose, and the choice is a value judgement. There is no technically superior answer. There is a decision about which harm matters more in this context, and it belongs to the people accountable for the decision, not to the person writing the code. Your job is to make the trade-off legible so they can make it knowingly.

Write down the choice and its cost. "We equalised false negative rates across groups, which raised the false positive rate for group B from 8% to 13%. We chose this because a missed diagnosis is worse than an unnecessary follow-up." That sentence is what accountability looks like in this field.

Beware anyone selling a solution. A tool that claims to make your model fair has picked a definition on your behalf. Ask which one. If the answer is not

immediate and specific, the tool is doing something you cannot audit.

The escape that is not a trick

The theorem's condition is *different base rates*. If the base rates are equal, the conflict disappears.

So the question worth asking is why they differ. Sometimes the difference is real. Often it is an artefact of measurement — the outcome you recorded is not the outcome you care about. Recorded rearrest is not reoffending; it is reoffending filtered through where police patrol. Recorded default is not inability to pay; it is inability to pay filtered through who was offered what terms.

When the measured base rate difference comes substantially from the measurement, the right intervention is upstream: fix the label, or change what you are predicting. That is slower and more disruptive than adjusting a threshold, and it is the only move that addresses the cause rather than distributing the consequence.

Why this is in a foundations course

Because it is the clearest case in machine learning where the mathematics tells you something a technique cannot fix. Most of this course teaches methods. This lesson teaches a limit, and the limit is more durable than any method in the syllabus — it will still hold when every algorithm here has been replaced.

BEFORE YOU MOVE ON

A team is told their model must be both calibrated within groups and have equal false positive rates, and the two groups have measurably different base rates. What is the correct response?

A. Collect more data and improve the model until both criteria are met, since the conflict comes from estimation error.

B. Apply post-processing threshold adjustment, which can satisfy both criteria simultaneously.

C. Explain that the two requirements are mathematically incompatible at unequal base rates, ask which harm the organisation considers worse, and document the choice and what it costs the other criterion.

D. Use demographic parity instead, since it avoids the conflict between the other two definitions.

52 · 9 min

The average is hiding the failure

THE ONE THING TO KEEP

A headline metric is a weighted average that hides its worst population, so slice by every dimension you can name, read fifty errors by hand to find the causes, and turn each failure mode into a fixed test set that runs on every change.

One number, many populations

A model reports 0.86 accuracy. Inside that number:

- 0.94 on desktop users, 0.71 on mobile.
- 0.89 on English text, 0.52 on Hindi.
- 0.91 on customers older than a year, 0.63 on those in their first month.

- 0.88 on weekdays, 0.74 on weekends.

The average is the weighted mean of these, dominated by whichever slice is largest. If mobile is 15% of traffic, a catastrophic mobile failure moves the headline number by three points and is invisible.

This is not an exotic scenario. It is the normal condition of a deployed model, and slicing is the routine that finds it.

How to slice

Start with the dimensions you would use to describe a user or an input, then add the ones specific to your system.

- Platform, device, browser, app version
- Language, region, time zone
- Tenure, account age, first-time versus returning
- Volume band — small orders versus large
- Time — hour of day, weekday, month, before and after a product change
- Data quality — rows with missing fields versus complete rows
- Any group with a fairness dimension

```
for col in slice_columns:
    for value, sub in df.groupby(col):
        if len(sub) < 30:
            continue
        acc = (sub["pred"] == sub["y"]).mean()
        print(f"{col}={value}: {acc:.3f}  n={len(sub)}")
```

The `len(sub) < 30` guard matters. Slice enough ways and you will find a group of 11 rows where accuracy is 0.45, and it will be noise. With a hundred slices, several will look bad by chance alone — this is the multiple comparisons problem from module 4 in a new costume. Confirm any finding on a second period or a second sample before acting on it.

Looking at the errors themselves

Automated slicing finds where the model fails. Reading errors tells you why, and there is no substitute for it.

Sort the validation set by loss, take the 50 worst, and read them. Not skim — read. For each, write one line about what went wrong. Then group your lines and count.

What comes out of an hour of this is a list like:

- 18 of 50: the label is wrong. The model was right.
- 12: the input is genuinely ambiguous; two humans would disagree.
- 9: a specific pattern the model has never seen — a new product category.
- 7: an input format the pipeline mangles — a currency symbol becoming a null.
- 4: genuinely hard.

That distribution is a work plan, and it is completely different from what the metrics suggested. Thirty of the fifty are data problems. Tuning the model addresses four of them.

Andrew Ng has made this argument for years and it remains the highest-value hour available to anyone with a model that is not good enough. Almost nobody does it, because it is tedious and there is always a hyperparameter to try instead.

Worst-group performance as a target

Once you have slices, you can optimise for the worst one rather than the average.

This is a real change in objective. A model that scores 0.86 average with a worst slice at 0.71 may be worse, for your purposes, than one scoring 0.84 average with a worst slice at 0.80. Which you prefer depends on whether the slice is a group of people who will be systematically failed.

The methods — group distributionally robust optimisation and its relatives — are beyond this course. The reporting habit is not: **put the worst slice in**

the summary, next to the average. A dashboard that shows only the mean is a dashboard designed to look good.

Turning findings into tests

The last step, and the one that makes this durable. Every failure mode you find becomes a fixed set of examples that runs on every model change.

Twenty Hindi inputs. Twenty mobile-format rows. Twenty rows with missing income. Ten known-hard cases with hand-checked labels. Score each set separately on every candidate model.

Now an improvement that raises the average while breaking Hindi is visible before it ships, rather than after a user reports it. This is the machine learning equivalent of a regression test suite, it takes an afternoon to build, and teams that have one ship far fewer surprises than teams that do not.

BEFORE YOU MOVE ON

An analyst slices a model 80 ways and finds four slices with markedly poor accuracy, each with 15 to 25 rows. What is the right next step?

A. Report all four immediately, since even small slices represent real users being failed.

B. Retrain with those four slices upweighted, so the model attends to them more.

C. Discard them, since any slice under 30 rows is statistically meaningless by convention.

D. Treat them as hypotheses and check the same slices on a separate period or sample, because 80 comparisons produce several poor-looking small slices by chance alone.

CASE STUDY · MODULE 5

Six analysts, one threshold, and a second bank's portfolio

A co-operative bank in Kolhapur with 1.6 million card transactions a month, a fraud desk of six analysts, and a newly acquired portfolio from a smaller bank.

The fraud desk at Janseva Sahakari Bank reviews flagged card transactions and telephones the cardholder. Six analysts, each managing about forty reviews in a shift, which is a hard ceiling of roughly 240 reviews a day including the ones that arrive from other channels.

Their model, built the previous year, scored transactions and flagged anything above 0.5. That produced about 90 flags a day at 31 per cent precision, and caught an estimated 54 per cent of fraud by value.

Three things happened in one quarter.

First, the risk committee asked why the threshold was 0.5. Nobody could answer. It was the library default, applied by a contractor who had left. The analyst who inherited the model, Meghana, worked the arithmetic from the cost-sensitive lesson. A missed fraud on this portfolio costs the bank ₹14,200 on average once the chargeback and the investigation are counted. A review costs about ₹110 of analyst time. The break-even threshold is 110 divided by 14,310, which is 0.0077.

At 0.0077 the model flags roughly 2,400 transactions a day. The desk can review 240.

That is not a modelling failure and it is not a mistake in the arithmetic. It is two constraints that do not both fit, and Meghana wrote both numbers on the whiteboard rather than choosing between them quietly. The cost-optimal threshold says the bank is under-investing in the fraud desk by a factor of ten. The capacity threshold says today's flag list has to be 240 long.

Second, she computed the per-row threshold instead. Review cost divided by review cost plus the transaction amount. A ₹110 review against a ₹4,80,000

transaction gives a threshold near 0.0002 — flag on almost any suspicion. Against a ₹900 transaction it gives 0.109. Applied at a fixed capacity of 240 flags a day, ranked by expected loss rather than by probability, the recovered value rose by 38 per cent on a back-test with no change to the model and no extra analyst.

Third, and this is the part that nearly went wrong, the bank absorbed the card portfolio of a smaller co-operative bank in a neighbouring district. Forty thousand new cards, a different customer profile, and a fraud base rate of 0.09 per cent against Janseva's own 0.31 per cent.

The model was scored on the new portfolio and its AUC held: 0.91 against 0.92. The committee read that as the model transferring cleanly and asked for it to go live on the new cards at the same operating point.

Meghana's objection was the base rate. Precision is not a property of a model. On the acquired portfolio, at a threshold producing the same flag rate, precision would fall from 31 per cent to roughly 9 per cent — three times as many wrongly telephoned customers per genuine case, on a customer base that had just been told its bank had been taken over.

The decision had a cost either way. Running it as-is meant a queue that was 91 per cent wrong, on new customers, with analysts who would learn within a fortnight to distrust the flags. Not running it meant the acquired portfolio was covered by nothing but the old bank's rules for at least a quarter.

> **STOP HERE**
>
> Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

The model went live on the acquired portfolio at a threshold set by expected value per row rather than by probability, and the two portfolios were given separate flag budgets — 200 and 40 — so that the new customers' queue could not be crowded out by the larger book.

The committee also funded two additional analysts, which was the first time the fraud desk's size had been argued for with a number rather than a feeling. The paper Meghana took to them contained the break-even calculation, the capacity constraint, and the value recovered per additional analyst-shift at the current model quality.

Six months later the acquired portfolio's precision had risen to 19 per cent, mostly because a hundred confirmed cases from that book had entered the training data. The AUC was unchanged throughout, which is the point: the thing that improved was not the model's ranking.

AUC was almost identical on the two portfolios. Why was that not evidence that the model would work on the new one?

Because AUC measures ranking only: the probability that a random fraud outscores a random legitimate transaction. It is insensitive to the base rate. Precision divides by what the model flagged, so when fraud falls from 0.31 per cent to 0.09 per cent the same ranking quality produces roughly a third of the precision. The model was equally good at ordering and much worse to work a queue from, and only one of those is what the analysts experience.

The cost-optimal threshold and the capacity threshold differed by a factor of ten. What does that gap actually mean?

It means the bank's own numbers say a reviewed case is worth far more than it costs to review, so the constraint is staffing rather than the model. Writing both thresholds down converts an invisible operational limit into a budget question that somebody can answer. Choosing the capacity threshold quietly would have hidden the fact that the fraud desk is the binding constraint on recovered losses.

Ranking by expected loss recovered 38 per cent more value with the same model. Where did that come from?

From information the global threshold was discarding. A fixed probability cut treats a ₹900 case and a ₹4,80,000 case as equally worth a slot in the queue. Ranking by probability multiplied by amount spends the day's 240 reviews where the losses are. It requires the probabilities to be roughly calibrated, since an uncalibrated score multiplied by an amount is not an expected loss.

MODULE 5 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 A model with 60 per cent recall and 99.4 per cent specificity moves from a population where fraud is 4 per cent to one where it is 0.1 per cent. What happens to precision?

A. It is unchanged, since recall and specificity are properties of the model rather than the population.

B. It falls sharply, because the flagged set is now mostly false positives.

C. It rises, because there are fewer positive cases available for the model to miss.

D. It cannot be computed at all until the model has been retrained on the new population.

2 A missed fraud costs ₹18,000 on average and a review costs ₹90. For a well-calibrated model, what is the cost-minimising threshold?

A. 0.5, because that is the point at which the two outcomes are equally likely for the row.

B. About 0.995, since the model has to be very confident before an analyst is asked to act.

C. About 0.2, because a two-hundred-to-one cost ratio maps onto a fifth of the default threshold.

D. About 0.005, from 90 divided by 18,090.

3 On 100,000 transactions containing 100 frauds, a model flags 900 legitimate ones. Why does the ROC curve barely register that?

A. Its false positive rate divides by the 99,900 negatives.

B. ROC is computed at a single threshold, so it cannot see the size of the flagged set at all.

C. ROC ignores false positives entirely and plots recall against precision as the threshold moves.

D. ROC requires calibrated probabilities, and this model's probabilities have not been calibrated.

4 A team applies isotonic regression to a random forest's outputs and reports that AUC improved. What should you conclude?

A. The calibration worked, because isotonic regression improves calibration and ranking together.

B. The forest was under-confident, and correcting that raises AUC more or less by construction.

C. Something else changed too, since a monotonic map cannot reorder scores.

D. Isotonic regression overfitted the calibration set, which is what inflated the reported AUC.

5 Two models are compared by running ten-fold cross-validation on each and applying a paired t-test to the ten pairs of fold scores. What is the defect?

A. Ten folds is far too few for a t-test to have any useful power at these effect sizes.

B. The folds share training data, so the differences are not independent.

C. A t-test requires the scores to be normally distributed, and AUC is bounded rather than normal.

D. The two models were fitted on different folds, so pairing the scores is invalid in principle.

6 Two groups have different base rates. A vendor claims their tool delivers calibration within groups and equal false positive and false negative rates. What should you conclude?

A. The tool has chosen one criterion without telling you which.

B. The claim is achievable if the model is accurate enough on both of the two groups.

C. The tool must be using per-group thresholds, which are unlawful in several jurisdictions.

D. The claim is achievable provided the two groups are of roughly similar size.

MODULE 6

Trees, ensembles, and the algorithms that win on tables

The family that actually wins most tabular problems, taken apart. How a single split is chosen, why averaging many deep trees works and why adding many shallow ones works for the opposite reason, the handful of hyperparameters that matter in the boosting libraries, and the two older algorithms still worth knowing — plus an honest account of what a feature importance number does and does not tell you.

BY THE END OF THIS MODULE YOU CAN

Choose between a linear model, a random forest and a gradient boosting library for a given tabular problem and defend the choice, tune the four hyperparameters that actually matter in each, and explain why impurity-based feature importance is biased and what permutation importance and SHAP replace it with

53 · 10 min

The classic algorithms, and when each is right

THE ONE THING TO KEEP

On tabular data, use linear models as a floor and gradient boosting as the answer; everything else needs a stated reason.

A short catalogue

Eight families cover almost everything built before deep learning, and most of what is still built on tables today. Each is a different answer to "what shape may the function take".

Linear and logistic regression. Weighted sum of the inputs. Linear regression predicts a number; logistic regression squashes the sum into a probability. Small, fast, and each coefficient is a readable statement: this feature moves the log-odds by this much. Cannot represent interactions or curves unless you build them in as features. Use it as your floor — the score every other model must beat to justify itself — and as your answer whenever you must explain a decision to a regulator or a customer.

k-nearest neighbours. No training. To predict, find the k most similar rows you have seen and take their average or majority vote. Everything hinges on the distance function, so features must be scaled and irrelevant columns actively hurt. Falls apart in high dimensions, where everything is roughly equidistant from everything. Good when you have few examples, a distance that genuinely means something, and you want a baseline in ten minutes.

Decision trees. A learned sequence of yes/no questions. "Is income above ₹40,000? Then is tenure above 8 months?" Handles non-linearity and interactions for free, needs no scaling, and takes mixed data types. A single tree is readable enough to print and put in front of a domain expert, which is sometimes the entire reason to use one. On its own it is unstable — change 5% of the data and the tree can be different — and it overfits unless pruned.

Random forest. Hundreds of trees, each on a bootstrap sample of rows and a random subset of features, averaged. Averaging cancels the instability. Very hard to make it perform badly, robust to defaults, and the honest workhorse when you do not want to think about hyperparameters.

Gradient boosting (XGBoost, LightGBM, CatBoost). Trees again, but built in sequence, each one fitted to the errors the previous ones left behind. Consistently the strongest thing on tabular data and has been for a decade. The cost is hyperparameters that matter — learning rate, depth, number of trees, regularization — and a genuine ability to overfit if you set them carelessly.

Support vector machines. Find the boundary with the widest margin between classes; the kernel trick lets that boundary be curved. Excellent when you have many features and few rows, which is why they still appear in genomics and other wide-data fields. Training scales badly past tens of thousands of rows, and the output is not a probability without extra work.

Naive Bayes. Applies Bayes' rule while pretending every feature is independent, which is false. It is fast, needs very little data, and works better than the assumption deserves for text. Its probabilities are badly calibrated — treat them as a ranking, never as a number.

k-means. Unsupervised. Partition rows into k groups by distance to k moving centres. Cheap and useful for exploration. You must choose k, it assumes roughly round and similarly sized clusters, and it returns clusters whether or not any exist.

The default

For a table of rows and columns:

1. Fit logistic or linear regression. Record the score. That is your floor.
2. Fit gradient boosting with sensible defaults. That is usually your ceiling.
3. If the gap between them is small, your features are the constraint. Go back to Lesson 3.
4. Reach for anything else only for a stated reason: a printable tree because a committee must read it, an SVM because you have 300 rows and 8,000

columns, k-NN because a good distance is the whole domain.

Deep learning does not belong in that list for tabular data. Repeated benchmarks find boosted trees ahead of neural networks on medium-sized tables, and the neural network costs more to tune and run. Use networks where they win: images, audio, text, and anything sequential.

What the gap between two models tells you

This is the useful move, and it costs an afternoon.

Logistic regression can only draw a straight boundary. Gradient boosting can carve arbitrary regions and capture interactions between features. If you fit both and the scores are within a point of each other, that is evidence: there is little non-linear structure left for a flexible model to find, so the ceiling is set by what your columns contain.

The reflex at that point is to reach for something more powerful still. It is the wrong reflex. Two model families with very different expressive power agreeing on a number is a message about the data, and the answer is a better feature, a better label, or a column you do not yet have.

BEFORE YOU MOVE ON

A team predicts crop yield from soil and weather columns. Their logistic-style linear model scores poorly. They switch to a random forest and the score moves by half a point. What does that most likely tell them?

A. The columns they have do not carry much information about yield.
B. Random forests are not strong enough; they should try gradient boosting or a neural network next.
C. The two models are broadly similar, so a similar score is expected and uninformative.
D. They need more training rows.

54 · 9 min

How a tree chooses where to cut

THE ONE THING TO KEEP

A tree scores every candidate threshold by the weighted drop in impurity and takes the best one greedily, which gives it interactions for free but leaves it unable to express a diagonal boundary or extrapolate beyond the training range.

Twenty loans

Twenty applications, six of which defaulted. The tree has to pick one feature and one threshold to split on. How does it decide?

It tries every threshold on every feature and scores each one by how much purer the two resulting groups are than the parent.

Impurity, measured

Gini impurity for a node is `1 − sum of p_k^2` over the classes. It is the probability that two items drawn at random from the node have different labels.

The parent: 6 defaults in 20, so p = 0.3.

```
Gini(parent) = 1 − (0.3^2 + 0.7^2) = 1 − (0.09 + 0.49) = 0.42
```

Now try the split `income < 300000`. It sends 8 rows left, of which 5 defaulted, and 12 right, of which 1 defaulted.

```
Gini(left)  = 1 − (0.625^2 + 0.375^2) = 0.469
Gini(right) = 1 − (0.083^2 + 0.917^2) = 0.153
weighted    = (8/20)*0.469 + (12/20)*0.153 = 0.188 + 0.092 =
0.280
gain        = 0.42 − 0.280 = 0.14
```

Try `age < 30`: 9 left with 3 defaults, 11 right with 3 defaults.

```
Gini(left)  = 1 − (0.333^2 + 0.667^2) = 0.444
Gini(right) = 1 − (0.273^2 + 0.727^2) = 0.397
weighted    = 0.45∗0.444 + 0.55∗0.397 = 0.418
gain        = 0.42 − 0.418 = 0.002
```

Income wins, decisively. The tree takes it, then repeats the whole procedure inside each child.

That is the entire algorithm. Try everything, take the best local gain, recurse. It is greedy — it never reconsiders an earlier split in light of a later one — which means a tree can miss a combination that a two-step lookahead would have found. Nobody does the lookahead, because it is exponentially expensive and the greedy version works well enough.

Entropy, and whether it matters

The alternative impurity measure is entropy, `−sum of p_k ∗ log2(p_k)`, borrowed from information theory. The gain is then called information gain.

In practice the two produce nearly identical trees. Entropy is slightly more expensive because of the logarithm, and its differences from Gini show up mainly in how they treat very unbalanced nodes. Gini is the default in scikit-learn, and the choice is not worth an experiment.

For regression the criterion is different: minimise the variance within each child, which is the same as minimising squared error. Each leaf predicts the mean of its training rows.

What a tree can and cannot represent

Worth being concrete, because it explains most of the algorithm's behaviour.

Can: any axis-aligned region. Interactions of any order, for free — split on income, then on age inside that branch, and you have expressed "young people with low income" without anyone specifying an interaction term. Non-monotonic relationships. Missing values as their own branch.

Cannot, without difficulty: a diagonal boundary. If the true rule is `income > 3 * expenses`, a tree approximates it with a staircase and needs many splits to do badly what one ratio does exactly. This is why the engineered-ratio advice in module 3 helps trees too.

Cannot, at all: extrapolate. Every prediction is a leaf mean computed from training rows, so a tree's output is bounded by the range of the training targets. Ask a tree trained on houses up to 130 square metres about a 400 square metre house and it returns the mean of its rightmost leaf. That is arguably safer than a linear model's confident extrapolation, and it is definitely different — know which failure mode you prefer.

A single tree overfits, spectacularly

Grown without limits, a tree keeps splitting until every leaf is pure, which usually means one training row per leaf. Training accuracy 100%, validation accuracy poor, and the tree is a lookup table.

The controls:

```
DecisionTreeClassifier(
    max_depth=5,              # hard cap on levels
    min_samples_leaf=20,      # the most useful one
    min_samples_split=50,
    max_features="sqrt",      # consider a subset at each split
    ccp_alpha=0.01,           # cost-complexity pruning
)
```

`min_samples_leaf` is the one to reach for first. It directly says "do not make a claim based on fewer than 20 examples", which is the honest version of what you want.

Why a single tree is rarely the answer

The fatal property is instability. Change 5% of the training rows and the top split may change, which changes every split beneath it, and you get a structurally different tree. Two trees fitted on 95% overlapping data can look nothing alike.

That is high variance in the module 4 sense, and it has a direct remedy: fit many trees and average them. Which is the next lesson, and the reason single trees are used today mostly for explanation rather than prediction — a depth-3 tree that a manager can read is worth something even when a forest predicts better.

BEFORE YOU MOVE ON

A regression tree trained on house sizes from 50 to 130 square metres is asked to price a 400 square metre house. What does it return, and why?

A. The mean target of its rightmost leaf, because every prediction is a leaf average computed from training rows and no leaf covers sizes above 130.

B. An extrapolated value following the slope implied by its deepest splits.

C. An error, since the input falls outside the range of every split threshold.

D. The overall training mean, because the input matches no leaf's conditions.

---

55 · 9 min

Bagging: many unstable models, averaged

THE ONE THING TO KEEP

Bagging reduces variance by averaging, and a random forest adds per-split feature sampling because decorrelating the trees removes more variance than making each individual tree better.

Averaging cancels variance

A single deep tree is high variance: fit it on a different sample and you get a different model. But its errors on any given row are somewhat random, and the average of many random errors is smaller than any one of them.

That is the whole idea of bagging — bootstrap aggregating. Fit the same high-variance model on many bootstrap samples of the data and average the pre-

dictions. Bias stays roughly where it was; variance falls.

The bootstrap sample

Draw n rows from n rows, with replacement. Some rows appear two or three times, and about 36.8% appear not at all — the limit of `(1 - 1/n)^n` as n grows, which is `1/e`.

Those left-out rows are called **out-of-bag**, and they are a free validation set. Every tree can be scored on the rows it did not see, and averaging that across trees gives you an estimate of generalisation error with no separate holdout and no cross-validation loop.

```
RandomForestClassifier(n_estimators=500, oob_score=True).fit(X,
y).oob_score_
```

This is genuinely useful on small datasets, where every row you can avoid spending on validation matters.

What makes a forest more than bagged trees

Bagging alone leaves the trees too similar. If one feature is strongly predictive, every tree splits on it first, and the trees end up correlated — and averaging correlated things removes much less variance than averaging independent ones.

Random forests add a second source of randomness: **at each split, only a random subset of features is considered.**

```
RandomForestClassifier(max_features="sqrt")     # classifica-
tion default
RandomForestRegressor(max_features=1.0)         # regression
default
```

With 36 features and `max_features="sqrt"`, each split chooses among 6 randomly selected features. Sometimes the dominant feature is not among them, so the tree is forced to find a different structure. The individual trees get

slightly worse and much more different from each other, and the ensemble gets better. Breiman's insight, in 2001, was that decorrelating the trees matters more than making each one good.

`max_features` is therefore the hyperparameter that actually matters in a forest. Lower means more decorrelation and more randomness; higher means stronger individual trees and more correlation.

What to tune, and what not to

- **`n_estimators`** — not really a hyperparameter. More trees never make a forest worse, only slower; the score converges and then flattens. Set it to 300 to 500 and stop thinking about it. Do not tune it in a grid search.
- **`max_features`** — tune this. Try `"sqrt"`, 0.3, 0.5, 1.0.
- **`min_samples_leaf`** — tune this, especially with noisy data. 1 is the default and is often too aggressive; 5 or 20 frequently helps.
- **`max_depth`** — usually leave unlimited. Deep trees are the point; averaging handles the variance.

That is four parameters of which two matter, which makes a forest one of the easiest models to get near its best performance with almost no tuning. It is an excellent second baseline after a linear model.

Extremely randomised trees

One more step in the same direction. `ExtraTreesClassifier` chooses split thresholds *at random* rather than searching for the best one, then keeps the best of those random candidates. Faster to train, since no threshold search happens, and more decorrelated still. On noisy data it often matches or beats a random forest, and it is worth trying because it costs one line.

Strengths and honest weaknesses

Strengths. Almost no tuning. No scaling needed. Handles mixed types, nonlinearity and interactions. Robust to outliers. Parallel across trees, so it uses every core. Gives out-of-bag error for free. Very hard to make catastrophically wrong.

Weaknesses, in order of how often they matter.

- **Model size.** 500 unlimited-depth trees on a million rows is hundreds of megabytes and slow to load. Gradient boosting with depth-6 trees is far smaller for similar accuracy.
- **Prediction latency.** Traversing 500 trees per prediction is more work than one boosted model of 300 shallow trees.
- **Usually slightly behind well-tuned boosting** on tabular accuracy, typically by a small margin.
- **Regression predictions are conservative.** Averaging pulls extremes toward the middle, so a forest under-predicts the highest values and over-predicts the lowest. If the tails are what you care about, this is a real problem.
- **No extrapolation**, inherited from trees.

When to reach for it

When you want a strong result with almost no effort, when you have limited time to tune, when the data is noisy, or when you need out-of-bag error because the dataset is small. It is the model to fit second, after a linear baseline, before you decide whether tuning a boosted model is worth the afternoon.

BEFORE YOU MOVE ON

A team sets max_features=1.0 in a random forest, so every split considers all features, and finds performance slightly worse than the sqrt default. Why?

A. Using all features overfits each tree, so the individual trees become too complex.

B. Considering all features at every split makes every tree pick the same dominant splits, so the trees become correlated and averaging removes less variance.

C. max_features=1.0 disables bootstrapping, so all trees see identical data.

D. Using all features slows training, so fewer trees complete within the same budget.

56 · 9 min

Boosting: each model fixes the last one's mistakes

THE ONE THING TO KEEP

Boosting fits each shallow tree to the residual left by the previous ones and adds only a fraction of it, so accuracy accumulates slowly enough that noise averages out — which is why a boosted model needs early stopping and a forest does not.

The opposite construction

A forest fits many strong models in parallel and averages away their variance. Boosting fits many weak models in sequence, each one correcting what the previous ones got wrong, and reduces bias.

The trees in a boosted model are deliberately shallow — depth 3 to 6, sometimes a single split. On its own such a tree is barely better than guessing. Several hundred of them, each addressing the remaining error, is usually the strongest thing you can fit to a table.

Watching it work on five rows

Regression, targets `[10, 12, 15, 9, 20]`.

Step 0. Predict the mean, 13.2, for everything. Residuals — what is left to explain — are `[-3.2, -1.2, 1.8, -4.2, 6.8]`.

Step 1. Fit a small tree *to the residuals*, not to the original target. Suppose it learns to predict `-2.9` for rows 1 and 4 and `+2.0` for rows 3 and 5. Add a fraction of that — the learning rate, say 0.1 — to the running prediction:

```
row 1: 13.2 + 0.1 * (-2.9) = 12.91
row 5: 13.2 + 0.1 * ( 2.0) = 13.40
```

Step 2. Recompute the residuals against the new predictions. Fit another small tree to those. Add a tenth of it. Repeat 500 times.

Each tree is fitted to what remains. That is gradient boosting in its original form, and the reason for the name is worth one sentence: for squared error, the residual *is* the negative gradient of the loss with respect to the prediction. So "fit the next model to the residuals" generalises to "fit the next model to the negative gradient", which is what lets the same algorithm work with log loss, quantile loss, or any differentiable objective.

Why the small steps

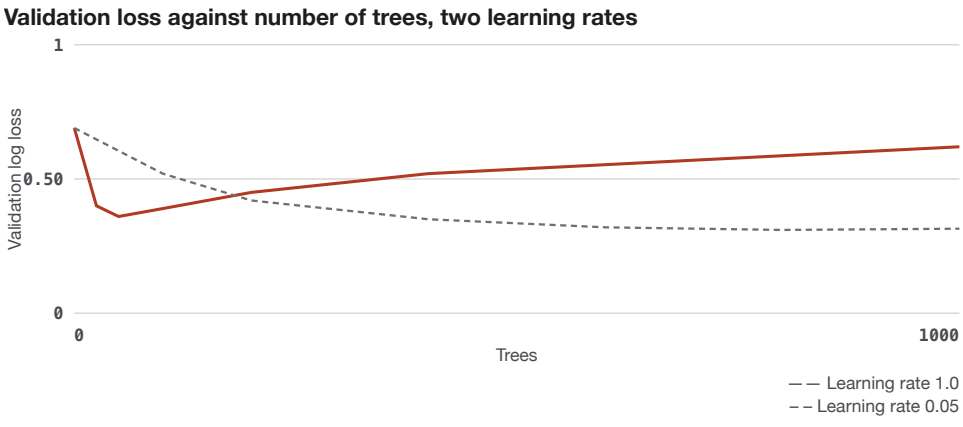
The learning rate — 0.1, or 0.05, or 0.01 — is why boosting works rather than overfitting immediately.

With a learning rate of 1.0 the model takes the full correction each time, drives the training residuals to zero in a few dozen trees, and has fitted the noise. With 0.05 it approaches the answer slowly, and each tree can only make a small claim, so noise gets averaged out across many trees while genuine signal accumulates.

The trade is exact and worth memorising: **halve the learning rate, double the number of trees.** Lower rates generalise slightly better and cost proportionally more time. 0.05 with 1,000 trees is a reasonable default; 0.3 with 100 trees is what you use while iterating quickly.

Early stopping is not optional

Unlike a random forest, **a boosted model gets worse with too many trees.** Each additional tree fits the remaining residuals, and once the signal is exhausted, what remains is noise.



At a rate of 1.0 the model has fitted the residual noise by tree fifty and gets worse from there. At 0.05 it approaches a lower floor slowly and holds it, because each tree can only make a small claim. Halve the rate, double the trees, and let early stopping pick the count.

So you stop when validation loss stops improving:

```
from sklearn.ensemble import HistGradientBoostingClassifier
m = HistGradientBoostingClassifier(
    learning_rate=0.05, max_iter=2000,
    early_stopping=True, validation_fraction=0.1,
n_iter_no_change=50)
m.fit(X_tr, y_tr)
m.n_iter_     # how many trees it actually used
```

`n_iter_no_change=50` means "keep going for 50 more rounds after the best score before giving up", which stops you from halting on a noisy plateau. This one setting converts `n_estimators` from a parameter you must tune into one that tunes itself, and it is the single most useful line in this lesson.

Stochastic boosting

Subsample rows and columns for each tree, exactly as a forest does, and boosting improves — usually a small but consistent gain, plus a speedup.

```
subsample     = 0.8    # fraction of rows per tree
colsample_bytree = 0.8 # fraction of features per tree
```

The reason is the same as in bagging: randomness decorrelates the trees so their individual errors do not compound. In boosting it also acts as regularisation, because each tree sees less data and can therefore make weaker claims.

Boosting versus forests, plainly

Boosting is usually more accurate on tabular data — often by one to three points — and is what wins competitions.

Boosting is more sensitive. Get the learning rate or the depth wrong and it overfits badly, where a forest would have shrugged.

Boosting is sequential, so it cannot parallelise across trees the way a forest can. The libraries parallelise *within* a tree instead, over features and rows, which is why they are still fast.

Forests are more forgiving. If you have one afternoon and no time to tune, a forest with defaults is the safer bet. If you have two days, boosting will beat it.

AdaBoost, the original 1995 boosting algorithm, works slightly differently — it reweights misclassified rows rather than fitting residuals — and is now mostly of historical interest. The gradient formulation replaced it because it extends to any differentiable loss.

BEFORE YOU MOVE ON

A team increases n_estimators from 500 to 5,000 in a gradient boosting model with learning_rate 0.1 and no early stopping. Validation loss gets worse. Why does this happen with boosting but not with a random forest?

A. Boosting trees are shallower, so more of them are needed and 5,000 is still insufficient to converge.

B. Each boosted tree fits the residual left by the previous ones, so once the signal is exhausted the additional trees fit noise; forest trees are fitted independently and averaged, so extra trees only refine the average.

C. Boosting is sequential, so later trees are trained on progressively smaller subsamples and see too little data.

D. The learning rate compounds across trees, so after enough rounds the predictions diverge numerically.

57 · 9 min

XGBoost, LightGBM, CatBoost: what actually differs

THE ONE THING TO KEEP

The three boosting libraries implement the same algorithm with different tree-growth and categorical handling, and six hyperparameters — learning rate, early stopping, depth, minimum leaf evidence, subsampling and L2 — cover essentially all the tuning that matters.

Three libraries, one algorithm

All three implement gradient boosted trees. All three are free and open source. The differences are real but narrower than the marketing suggests, and on a well-tuned problem they usually land within a point of each other.

XGBoost (2014) is the one that made the technique famous. Mature, enormously well documented, level-wise tree growth by default, and available in every language you might need. It is the safe institutional choice.

LightGBM (2017) is usually the fastest and lightest on memory. Two design choices explain most of that: it bins continuous features into a histogram (typically 255 buckets) before searching for splits, which turns a sort into a count; and it grows trees **leaf-wise** rather than level-wise, always splitting the leaf that promises the biggest gain. Leaf-wise growth gets more accuracy per tree and overfits more readily on small data, which is why `num_leaves` matters more there than `max_depth`.

CatBoost (2017) handles categorical features natively with an ordered target statistic — a target encoding computed using only rows that came earlier in a random permutation, which avoids the leakage from module 3 by construction. It also builds symmetric trees, where every node at a level uses the same split, which makes prediction extremely fast. Its defaults are unusually good; it is often the best result you can get without tuning at all.

What to reach for

- Many categorical columns, or you want a strong result with no tuning: **CatBoost**.
- Large data, and training time matters: **LightGBM**.
- You want maximum ecosystem support, or your organisation already uses it: **XGBoost**.
- You do not want another dependency: `HistGradientBoostingClassifier` in scikit-learn, which is a LightGBM-style histogram implementation, handles NaN natively, and is close enough to the others that the difference rarely decides anything.

That last option deserves emphasis. It is in the library you already have, it needs no install, and it is a genuinely competitive model.

The parameters that matter

Every library exposes forty or more. Six of them do the work.

1. `learning_rate` — 0.05 for a final model, 0.3 while iterating. Everything else is easier to tune once this is settled.

2. `n_estimators` **with early stopping** — set it high, let early stopping choose. Do not put it in a grid search.

3. Depth — `max_depth` 4 to 8 for XGBoost and CatBoost; `num_leaves` for LightGBM, where the rough correspondence is `num_leaves` around `2^max_depth` but usually smaller. LightGBM's default of 31 is often too high for datasets under 10,000 rows.

4. `min_child_weight` / `min_data_in_leaf` — the minimum evidence required to make a split. Raise it when overfitting. This is the most under-used control of the six.

5. `subsample` **and** `colsample_bytree` — 0.8 each is a good default and rarely a bad one.

6. `reg_lambda` — L2 on the leaf values. Raise it on noisy data.

A workable starting point:

```
import lightgbm as lgb
m = lgb.LGBMClassifier(
    learning_rate=0.05, n_estimators=3000, num_leaves=31,
    min_child_samples=30, subsample=0.8, subsample_freq=1,
    colsample_bytree=0.8, reg_lambda=1.0, random_state=0)
m.fit(X_tr, y_tr, eval_set=[(X_val, y_val)],
      callbacks=[lgb.early_stopping(100)])
```

Note `subsample_freq=1` — in LightGBM, setting `subsample` alone does nothing without it, which is a long-standing trap that costs people an afternoon.

Where these models are genuinely strong

On tabular data, boosted trees remain the state of the art. This has been tested carefully, most notably by Grinsztajn and colleagues in 2022, who compared tree ensembles against several neural architectures across many tabular

datasets and found trees ahead on most of them — while also finding the gap narrows as datasets grow beyond a few tens of thousands of rows.

The reasons they give are mechanical rather than mysterious: tabular features are often uninformative individually, columns have no meaningful ordering for a network to exploit, and target functions are frequently irregular in a way piecewise-constant models handle naturally. That is a live research area rather than a settled matter, and the honest summary is that on a table with fewer than 100,000 rows, boosted trees are the model to beat and usually are not beaten.

Where they are weak

- **Extrapolation** — inherited from trees, unchanged by ensembling.
- **Very high-dimensional sparse data** — a hundred thousand TF-IDF columns is a linear model's territory, not a tree's.
- **Images, audio, raw text** — no.
- **Latency at scale** — 1,000 trees per prediction adds up, though the fix is well trodden: fewer, deeper trees, or a compiled predictor such as Treelite.
- **Explainability** — 1,000 trees is not readable. Which is the next two lessons.

BEFORE YOU MOVE ON

A team moves from XGBoost to LightGBM on a 6,000-row dataset with identical hyperparameters and finds LightGBM overfits noticeably more. What is the most likely cause?

A. LightGBM bins continuous features into histograms, which loses resolution and forces the model to memorise bin boundaries.
B. LightGBM's categorical handling applies target statistics that leak on small datasets.
C. LightGBM grows trees leaf-wise, always splitting the highest-gain leaf, so at the default num_leaves of 31 it builds deeper, more specialised trees than level-wise growth does on 6,000 rows.
D. LightGBM defaults to a higher learning rate, so the same n_estimators produces a more aggressive fit.

58 · 8 min

k-nearest neighbours, and why distance stops meaning anything

THE ONE THING TO KEEP

k-NN makes the assumption behind most machine learning explicit — nearby points share targets — and the curse of dimensionality breaks it because distances concentrate, so the nearest neighbour in 1,000 raw dimensions is barely nearer than the farthest.

The model with no training

Store the data. To predict, find the k closest training rows and take their average or their majority vote.

```
from sklearn.neighbors import KNeighborsClassifier
KNeighborsClassifier(n_neighbors=15, weights="distance")
```

There is no fitting step — `fit` just stores the array. All the work happens at prediction time, which inverts the usual cost profile: instant training, slow predictions, and the entire training set has to live in memory forever.

k-NN is worth understanding for three reasons beyond its own use. It is the clearest possible illustration of the curse of dimensionality; it is the mechanism underneath every embedding search in module 7; and it makes the assumption behind most machine learning explicit — **that things close together in feature space have similar targets.** Every model assumes some version of that. k-NN says it out loud.

Choosing k

k = 1 memorises: training error is zero and the decision boundary is a jagged shape drawn around individual points. High variance.

Large k smooths: at k = n every prediction is the global average. High bias.

Something between 5 and 50 is usually right, chosen by cross-validation. `weights="distance"` makes nearer neighbours count more, which usually helps and lets you use a larger k without over-smoothing.

Scaling is not optional here

With age in years and income in rupees, the distance between two people is essentially the difference in their incomes. Age contributes nothing measurable. Standardise, always, before any distance-based method — this is the model where forgetting it produces the most obviously wrong answer.

Categorical features are a genuine problem. What is the distance between Mumbai and Chennai? One-hot encoding makes every pair of distinct cities exactly the same distance apart, which at least is not misleading, but it also inflates dimensionality — which brings us to the real issue.

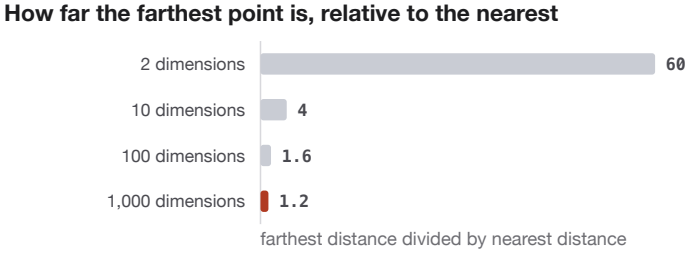
The curse of dimensionality, with numbers

In high dimensions, everything is far away and, worse, everything is roughly *equally* far away.

Generate 1,000 random points uniformly in a unit cube and look at the distance from one point to its nearest and farthest neighbours:

- **2 dimensions**: nearest about 0.02, farthest about 1.2. The ratio is around 60. "Near" and "far" are clearly different.
- **10 dimensions**: nearest about 0.5, farthest about 2.0. Ratio 4.
- **100 dimensions**: nearest about 3.5, farthest about 5.5. Ratio 1.6.
- **1,000 dimensions**: nearest about 12.4, farthest about 14.8. Ratio 1.2.

By 1,000 dimensions, your nearest neighbour is barely nearer than your farthest. The word "neighbour" has stopped picking anything out, and a method built entirely on that word has nothing to work with.



1,000 random points in a unit cube. In two dimensions the farthest neighbour is sixty times further away than the nearest. By 1,000 dimensions it is only 20 per cent further, and the word neighbour has stopped picking anything out. This is geometry, and it affects every method built on distance.

The geometric reason: in d dimensions, distance is a sum of d squared differences. As d grows, that sum concentrates around its expectation — the law of large numbers applied to coordinates — and the spread between the smallest and largest sums shrinks relative to their size.

What this means beyond k-NN

This is not a k-NN problem; it is a geometry problem, and it affects every method that relies on distance or density: k-means, DBSCAN, RBF-kernel SVMs, Gaussian mixtures, and any anomaly detector built on distance.

Two qualifications keep it from being paralysing.

Real data usually has lower intrinsic dimension than its column count. Photographs of faces occupy a tiny, curved region of pixel space; the features are heavily correlated, so the effective dimensionality is far below the nominal one. This is why nearest-neighbour search over 768-dimensional embeddings works well in practice, and it is the reason the manifold idea in module 7 matters.

Learned representations are built to make distance meaningful. An embedding is trained so that similar things land close together. That is precisely what raw high-dimensional coordinates fail to give you, and it is why the pairing of embeddings with nearest-neighbour search is so effective.

When k-NN is the right answer

Small datasets. Low dimensions. Genuinely irregular decision boundaries that a parametric model would smooth over. Recommendation by similarity. As a baseline, because it takes one line and sometimes surprises you.

And when the dataset is large, exact search becomes too slow, at which point you want approximate nearest neighbour indexes — FAISS, HNSW, Annoy, all free — which trade a small amount of recall for enormous speed. Module 7 covers them, because that is where they earn their keep.

BEFORE YOU MOVE ON

A team applies k-NN to 300-dimensional TF-IDF vectors and finds predictions no better than chance, though a linear model on the same features works well. What explains the difference?

A. TF-IDF values are not standardised, and k-NN requires standardisation while linear models do not.
B. In 300 sparse dimensions the distances between documents concentrate, so nearest neighbours are barely nearer than distant ones, while a linear model uses a weighted sum that does not depend on distances at all.
C. k-NN cannot handle sparse matrices, so the vectors were densified incorrectly.
D. 300 dimensions is too few for k-NN, which needs high-dimensional data to find meaningful neighbourhoods.

59 · 8 min

Naive Bayes: wrong assumptions, useful results

THE ONE THING TO KEEP

Naive Bayes assumes features are independent given the class, which is false and inflates its probabilities without usually changing which class ranks highest — so it discriminates well while being badly calibrated.

Turning the problem around

Most classifiers model `P(class | features)` directly. Naive Bayes models `P(features | class)` and flips it with Bayes' rule:

```
P(spam | words) is proportional to P(words | spam) * P(spam)
```

The first term is estimated by counting how often each word appears in spam. The second is just the fraction of messages that are spam. Nothing is fitted iteratively; the whole model is a table of counts.

The naive part

`P(words | spam)` for a 50-word message is a joint probability over 50 variables, and you would need an impossible amount of data to estimate it.

So assume the words are independent given the class, and multiply:

```
P(words | spam) = P(word1 | spam) * P(word2 | spam) * ...
```

This assumption is false. "Free" and "money" co-occur far more than independence predicts. "New" and "York" are not independent in any corpus. The model does not care and works anyway, which is the interesting part.

Why a false assumption still classifies well

The explanation is worth having, because it generalises.

When features are correlated, the model counts the same evidence repeatedly. Seeing "free", "money" and "cash" in one message is treated as three independent pieces of evidence when it is closer to one, so the model's confidence is inflated — often to 0.9999.

But classification only requires the *ranking* of the class scores to be right, not their magnitudes. Over-counting evidence tends to push scores toward the extremes without changing which class comes out on top. The probabilities become nonsense; the decision remains correct.

This is exactly the pattern from the calibration lesson: excellent discrimination, terrible calibration. If you need Naive Bayes probabilities to mean something, calibrate them afterwards. If you only need the label, do not bother.

The three variants

MultinomialNB — for counts. Word counts, TF-IDF. The standard choice for text.

BernoulliNB — for binary presence or absence. Better than Multinomial for short documents, where whether a word appears matters more than how often.

GaussianNB — for continuous features, assuming each is normally distributed within each class. Two assumptions stacked — independence and normality — and it is usually the weakest of the three on real data.

```
from sklearn.naive_bayes import MultinomialNB
MultinomialNB(alpha=1.0).fit(X_counts, y)
```

Smoothing, and why it is mandatory

A word that never appeared in spam during training gets `P(word | spam) = 0`. Multiply by zero and the whole message's spam probability is zero, no matter what the other 49 words say. One unseen word vetoes all the evidence.

Laplace smoothing adds a small count to everything:

```
P(word | spam) = (count + alpha) / (total + alpha * vocabulary_size)
```

`alpha=1.0` is the default and it is the difference between a working model and a broken one. It is worth understanding rather than accepting, because the same zero-probability problem appears in many count-based methods.

A related implementation detail: real implementations sum logs rather than multiplying probabilities, because multiplying fifty numbers below 0.1 underflows to zero in floating point. If you ever implement this, work in log space.

When it earns its place

Very fast. Training is one pass of counting; prediction is a sum of logs. On a million documents this trains in seconds where a boosted model takes an hour.

Works with tiny data. Because it estimates each feature's statistics independently, it needs far fewer examples than a model estimating interactions. With 200 labelled documents it is often the best thing available.

Streams naturally. `partial_fit` lets you update counts as data arrives, with no retraining.

Genuinely competitive on text, especially short text and especially with few labels. It is a legitimate baseline, not a teaching toy — the classic Naive Bayes plus TF-IDF spam filter was the state of the art for years and is still deployed.

Where it fails

When feature interactions carry the signal, it cannot see them, by construction. When you need calibrated probabilities, it is unusable without post-processing. And on tabular data with correlated continuous features it is usually beaten comfortably by anything else in this module.

The reason it stays in the curriculum is that it is the cleanest example of a principle worth internalising: **a model whose assumptions are plainly false can still be the right tool, if the way it is wrong does not affect the decision you are making.** Asking *how* a model is wrong, rather than *whether*, is most of applied judgement.

BEFORE YOU MOVE ON

A Naive Bayes spam filter classifies well but assigns 0.9999 probability to almost every message it calls spam, including ones it gets wrong. What causes the extreme confidence?

A. Laplace smoothing with alpha=1.0 is too weak for a large vocabulary, inflating the estimated word probabilities.
B. Correlated words are treated as independent evidence, so the same signal is multiplied in several times, pushing the score to an extreme without necessarily changing the ranking.
C. The model uses log probabilities, and exponentiating them back introduces numerical error near the boundaries.
D. The class prior is estimated from an imbalanced training set, so the spam prior dominates the word evidence.

60 · 9 min

Support vector machines and the kernel trick

THE ONE THING TO KEEP

An SVM maximises the margin and depends only on the support vectors; the kernel trick works because the optimisation needs only pairwise dot products, so an infinite-dimensional feature space costs one similarity computation instead of infinite coordinates.

Maximising the gap

Many lines separate two classes. A support vector machine picks the one with the widest margin — the greatest distance to the nearest point of either class.

The intuition is that a boundary with room on both sides is more robust: a new point has to move further before it crosses. Only the points on the edge of the margin — the **support vectors** — determine the boundary. Every other

training point could be deleted and the model would be identical, which is a genuinely unusual property and gives SVMs a compact representation.

Soft margins

Real data is not separable, so the SVM allows violations and charges for them. `C` sets the price:

- **Small C** — violations are cheap, so the margin is wide and some points sit inside or on the wrong side. More regularisation.
- **Large C** — violations are expensive, so the boundary contorts to classify every training point. Less regularisation, more overfitting.

`C` is the main hyperparameter and should be searched on a log scale: 0.01, 0.1, 1, 10, 100.

The kernel trick, which is not a trick

Here is the idea that made SVMs famous, and it is worth understanding properly because it recurs elsewhere.

Some data is not linearly separable in its original space but becomes separable in a higher-dimensional one. Points in a circle around the origin cannot be split by a line in two dimensions. Add a third coordinate equal to `x^2 + y^2` and a flat plane separates them perfectly.

The obvious approach is to compute those extra coordinates. For a polynomial of degree 3 over 100 features that is over 170,000 columns, and for some useful transformations the space is infinite-dimensional.

The trick: the SVM's optimisation only ever needs **dot products between pairs of points**, never the coordinates themselves. So if you can compute the dot product in the high-dimensional space directly from the original coordinates, you never need to visit that space.

```
RBF kernel: K(a, b) = exp(−gamma ∗ ||a − b||^2)
```

That is a similarity between two points, computable in the original space, and it corresponds to a dot product in an infinite-dimensional feature space. You get the expressiveness of infinite dimensions for the cost of one exponential per pair.

Tuning gamma

With an RBF kernel, `gamma` controls how far a single point's influence reaches.

- **Small gamma** — wide influence, smooth boundary, close to linear.
- **Large gamma** — each point influences only its immediate surroundings, and the boundary becomes islands around individual training points. This is overfitting, and it looks distinctive if you plot it.

`C` and `gamma` interact, so they must be searched together:

```
from sklearn.svm import SVC
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

model = make_pipeline(StandardScaler(),
                      SVC(kernel="rbf", C=1.0, gamma="scale"))
```

`gamma="scale"` adapts to the data's variance and is a sensible default. Scaling is mandatory — the RBF kernel is a function of Euclidean distance, so unscaled features break it exactly as they break k-NN.

The cost that decided their fate

Training an SVM is roughly quadratic to cubic in the number of rows, because the optimisation works with a matrix of pairwise similarities. At 10,000 rows this is fine. At 100,000 it is slow. At a million it is impractical.

That scaling is the main reason SVMs faded from general use after 2012, not any failure of the idea. Gradient boosting handles a million rows comfortably, and neural networks handle far more.

For a linear kernel there is a way out: `LinearSVC` uses a different solver that scales roughly linearly in rows, and it is genuinely fast on large sparse text data. If you want an SVM on a big dataset, that is the one.

Where they still earn their place

- **Small to medium datasets** with complex boundaries — under about 10,000 rows.
- **High-dimensional data with few samples**, such as gene expression with 20,000 features and 200 patients. SVMs are unusually well behaved here, where most methods struggle.
- **Text classification** with `LinearSVC`, which remains a strong and very fast baseline.
- **When you need a hard decision boundary** rather than a probability.

The honest weaknesses

No native probabilities — `probability=True` runs an internal cross-validated calibration that makes fitting several times slower. Two interacting hyperparameters that need a real search. Poor scaling in rows. And no interpretability at all with a non-linear kernel: the boundary lives in a space you never constructed and cannot inspect.

They are a beautiful piece of mathematics whose practical territory has narrowed considerably. Knowing them is still worthwhile, because the kernel idea — expressing a computation entirely in terms of similarities between pairs — appears again in Gaussian processes, in kernel PCA, and in the attention mechanism that transformers are built on.

BEFORE YOU MOVE ON

An RBF SVM achieves perfect training accuracy and 0.58 validation accuracy. Both C and gamma were set high. Which single change addresses the mechanism?

A. Switch to a polynomial kernel, which is less prone to memorising individual points.
B. Increase the number of training rows, since SVMs need large samples to find a stable margin.
C. Lower gamma, so each training point's influence extends further and the boundary becomes smooth instead of forming islands around individual points.
D. Enable probability=True so the decision boundary is fitted with calibrated probabilities rather than hard margins.

61 · 9 min

Feature importance, and the ways it misleads

THE ONE THING TO KEEP

Impurity importance favours high-cardinality features and is computed on training data; permutation importance on validation data measures what you meant, but both collapse when features are correlated, so permute clustered groups rather than single columns.

The number everybody prints

```
model.feature_importances_
```

One line, a satisfying bar chart, and a plausible story about what drives the outcome. It is also, in its default form, systematically biased in ways that are easy to demonstrate and rarely mentioned.

What impurity importance actually measures

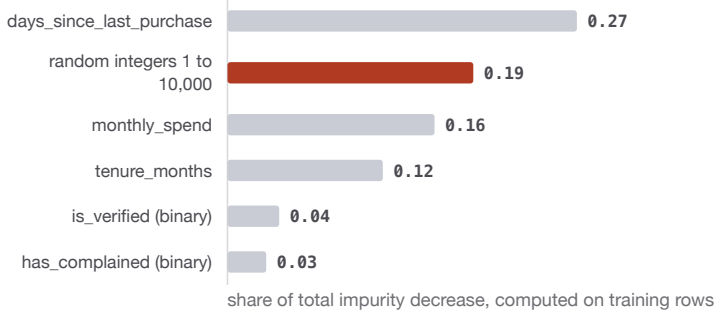
For a tree ensemble, the default importance sums the impurity decrease each feature achieved across all splits, weighted by how many rows passed through those nodes.

That definition contains two biases.

Bias towards high-cardinality features. A feature with many distinct values offers many candidate thresholds, so by chance one of them will reduce impurity. Continuous features and high-cardinality categoricals therefore accumulate importance even when they carry no signal.

The demonstration is easy and worth running once. Add a column of random integers between 1 and 10,000 to your dataset and refit. It will appear in the top five features, above genuinely predictive binary columns. Nothing is broken; the metric is measuring what it says it measures, and what it measures is not what you wanted.

Impurity importance with a column of random noise added



The random column offers 10,000 candidate thresholds, so by chance some split reduces impurity, and it lands second — above two binary columns that genuinely predict the target. Nothing is broken. The metric measures what it says it measures, on training data, and that is not what you wanted.

Computed on training data. Impurity importance reflects how the model used a feature to fit the training set, including the parts it overfitted. A feature that helped the model memorise noise scores highly.

Permutation importance

A better default. Shuffle one feature's values in the *validation* set, breaking its relationship with the target while leaving the rest of the row intact, and measure how much the score drops.

```
from sklearn.inspection import permutation_importance
r = permutation_importance(model, X_val, y_val, n_repeats=10,
                           scoring="roc_auc", random_state=0)
```

This measures what you probably meant: how much this model, on held-out data, depends on this feature. It works for any model, it uses the metric you care about, and it gives a standard deviation across repeats so you can see which differences are real.

It has one serious failure mode, and it is not optional knowledge.

The correlated-feature problem

Suppose `height_cm` and `height_inches` are both in the model. Shuffle one, and the model reads the other and predicts fine. Score barely drops. Shuffle the other, same result. Both features get near-zero importance, and the conclusion "height does not matter" is exactly backwards.

This affects both methods, and it affects them in different directions — impurity importance splits the credit between correlated features, permutation importance zeroes both.

Two practical responses:

1. **Cluster correlated features and permute the whole group at once.** Compute the correlation matrix, cluster it, and shuffle each cluster together. This tells you what the *group* is worth, which is usually the question.
2. **Drop-column importance.** Remove a feature, refit, measure the drop. This is the most trustworthy method and costs one refit per feature, so it is affordable only with fast models and few columns.

The second also has a subtlety: if you drop `height_cm` and refit, the model learns to use `height_inches` instead and the score does not fall. The answer is the same — cluster first.

Importance is not effect, and not causation

Three distinctions that get collapsed constantly.

Importance is not effect size. "Income is the most important feature" does not tell you whether more income raises or lowers the prediction, or by how much. It says the model relies on it.

Importance is a property of the model, not the world. Two models fitted on the same data can rank features differently, both legitimately, because they use different structure.

Importance is not causation. Module 1 made this argument and it applies here with full force: a feature can be important because it is a consequence of the target. A high importance on a suspicious feature is a leakage alarm before it is anything else.

What importance is genuinely good for

Having spent the lesson on limits, the honest positive list.

- **Finding leakage.** One feature far above the others on a hard problem is the most reliable leakage detector there is.
- **Feature selection.** Drop the bottom half, refit, check whether the score moved. Often it does not, and you have halved your pipeline's maintenance cost.
- **Sanity checking.** If your domain expert says the top feature should be irrelevant, one of you has learned something.
- **Debugging a pipeline.** A feature with exactly zero importance across every tree is often not reaching the model at all.

None of those uses requires importance to be a measure of real-world influence. All of them are valuable. The failure is only ever in the last step, when a bar chart becomes a slide titled "what drives customer churn".

BEFORE YOU MOVE ON

A model contains both `total_spend` and `average_order_value`, which are strongly correlated. Permutation importance gives both near zero. What should you conclude?

A. Spending behaviour is genuinely unimportant to this model, and both columns can be dropped.
B. The permutation test was run on training data, which is why the importances collapsed.
C. Shuffling one leaves the other intact, so the model recovers the same information and the score barely moves; permute the correlated group together to see what spending is worth.
D. Near-zero permutation importance always indicates a feature that never reached the model, so the pipeline is dropping both columns.

62 · 9 min

Explaining one prediction, and the shape of one feature

THE ONE THING TO KEEP

SHAP divides one prediction's departure from the average among the features with an exact additive decomposition, partial dependence shows a feature's average learned shape, and both assume feature independence in ways that break when the columns are correlated.

Two different questions

"Which features does this model rely on overall?" was the last lesson. Two more specific questions come up constantly and need different tools.

Why did *this* application get declined? A local explanation, for one row.

How does the prediction change as *this* feature changes? A global shape, for one column.

SHAP: dividing the credit for one prediction

SHAP assigns each feature a number saying how much it pushed this particular prediction away from the average prediction. The numbers add up exactly:

```
base value (average prediction)          0.12
+ income = 240,000                      −0.04
+ late_payments = 3                     +0.19
+ account_age_days = 45                 +0.08
+ everything else                       −0.02
= this prediction                        0.33
```

That is a readable answer to "why 0.33", and it is the format regulators tend to accept for adverse action notices.

The underlying idea comes from cooperative game theory. Shapley values answer: if features are players cooperating to produce a prediction, what is each one's fair share of the result? The fair division is defined by averaging a feature's marginal contribution over every possible order in which features could be added. That is the only allocation satisfying a small set of reasonable axioms, which is why SHAP has a stronger theoretical footing than most explanation methods.

Computing it exactly requires evaluating all `2^n` subsets. `TreeSHAP` does it exactly for tree ensembles in polynomial time, which is why SHAP became standard alongside gradient boosting specifically.

```
import shap
explainer = shap.TreeExplainer(model)
sv = explainer.shap_values(X_val)
shap.summary_plot(sv, X_val)
```

What SHAP does not tell you

Three limits, stated because they are routinely ignored.

It explains the model, not the world. A SHAP value of +0.19 for `late_payments = 3` means this model's output was 0.19 higher because of that value. It does not mean reducing late payments by three would lower the person's real risk. Module 1's distinction, again.

Correlated features share credit in ways that depend on the method. With `height_cm` and `height_inches` in the model, how the credit divides between them is partly an artefact of the algorithm's assumptions rather than a fact about the model.

The standard approximation assumes feature independence. When features are dependent, the method evaluates the model on combinations that never occur in reality — an income of ₹40 lakh with a postcode where nobody earns that. The model's output on impossible inputs then influences the attribution. There are variants that condition on the data distribution instead, and they are slower and less used.

Partial dependence: the shape of one feature

A different question. Take every row, set `income` to 200,000 for all of them, average the predictions. Repeat for 250,000, 300,000, and so on. Plot the curve.

```
from sklearn.inspection import PartialDependenceDisplay
PartialDependenceDisplay.from_estimator(model, X_val,
["income", "age"])
```

The result shows the average relationship the model learned. It is frequently the most informative single plot you can produce about a tree ensemble, and it regularly reveals things nobody expected — a threshold effect at a round number that turns out to be a policy rule, a relationship that reverses direction, a plateau where a linear model assumed a slope.

Its flaw is the same independence assumption. If income and postcode are correlated, setting income to 40 lakh for every row creates impossible people, and the average includes the model's behaviour on them.

ALE plots — accumulated local effects — fix this by computing the effect locally, using only rows where the feature is already near the value being examined. They are more trustworthy on correlated data and less widely used. The `alibi` library implements them, free.

ICE: the curves the average hides

Partial dependence plots an average, and averages hide heterogeneity. An **individual conditional expectation** plot draws one line per row instead.

If all the lines are parallel, the average is a fair summary. If half slope up and half slope down, the average is flat and completely misleading — there is an interaction, and the effect of this feature depends on something else in the row.

```
PartialDependenceDisplay.from_estimator(model, X_val,
["income"],
                                        kind="both")
```

`kind="both"` draws the ICE lines and the average together. It costs nothing and it is the check that stops you reporting a flat average over two opposite populations.

Choosing between them

- **One decision to justify to a person**: SHAP.
- **Understanding the model's overall behaviour on one feature**: partial dependence, with ICE lines.
- **Which features matter at all**: permutation importance, from the previous lesson.
- **Correlated features and you need it right**: ALE, and read the caveats.

And the standing rule underneath all four: these describe the model. If somebody is going to act on the explanation to change the world, that requires an experiment, and no amount of attribution substitutes for one.

BEFORE YOU MOVE ON

A partial dependence plot for 'discount' is flat, suggesting discount has no effect. The ICE plot shows half the lines sloping up steeply and half sloping down. What is happening?

A. The model has not converged, so its predictions are unstable across rows.

B. The partial dependence computation used too few grid points, flattening genuine curvature.

C. Discount raises the prediction for one group of customers and lowers it for another, so the average cancels; there is an interaction the partial dependence plot cannot show.

D. The ICE lines are computed on training data and the partial dependence on validation data, so the two disagree.

CASE STUDY · MODULE 6

The bar chart that nearly rebuilt three warehouses

A grain storage company in Madhya Pradesh with 46 warehouses, and a board paper due in eleven days recommending where to spend a ₹9 crore refurbishment budget.

Annadhan Warehousing stores wheat, gram and soya for traders and for the state procurement agency. Spoilage — grain downgraded on outward inspection — costs the company between two and four crore rupees a year and costs it customers, which is worse.

They had four years of records: 310,000 stack-months, one row per stack per month, with moisture at intake, variety, stack height, days in store, fumigation history, ambient humidity from the nearest weather station, warehouse identifier, and whether the stack was downgraded at outward inspection.

A gradient boosting model reached 0.83 AUC against a baseline of 0.61. That part was uncontroversial and useful — the model was going into the fortnightly inspection schedule, deciding which stacks got opened first.

The controversy was the second slide. Somebody had printed `model.feature_importances_` and the top feature, well clear of everything else, was `warehouse_id`. The operations director read that as the model saying the warehouses themselves were the problem, and the refurbishment paper was drafted around the three warehouses whose identifiers the model split on most often. Nine crore rupees.

The analyst, Devika, had been in the job four months and asked for a week.

Her first test took an hour and is the one from the importance lesson. She added a column of random integers between one and ten thousand, with no relationship to anything, and refitted. The random column came fourth. Nothing was broken. Impurity importance rewards a feature for offering many candidate thresholds, and `warehouse_id` has forty-six levels while `variety` has four and `fumigated_last_30d` has two. A high-cardinality column accu-

mulates gain by having more chances to reduce impurity, whether or not it carries signal.

Permutation importance on held-out data told a different story: moisture at intake first, days in store second, ambient humidity third, warehouse identifier seventh and with an interval that crossed zero.

Then the correlated-feature problem arrived, because these two are not independent. Warehouses in the Hoshangabad belt receive grain from a catchment with systematically higher moisture at intake. Permuting `warehouse_id` alone barely moved the score, because the model simply read moisture instead. Permuting `moisture_at_intake` alone barely moved it either, because the model read the warehouse. Both looked unimportant, which is the failure the lesson warns about, and taken literally it would have said nothing mattered.

Devika clustered the correlated columns and permuted the group. Moisture-and-location together accounted for most of the model's dependence. Which of the two is doing the work is not a question the model can answer, because in this data they never move independently.

The decision, with eleven days left, was genuinely hard.

Recommend the refurbishment as drafted. The three warehouses are in the high-moisture belt and they do have the worst spoilage. Something is wrong there. Spending nine crore on ventilation and flooring is not obviously wasted, and the paper is written.

Recommend intake moisture control instead — testing at the gate, rejection thresholds, a drying arrangement with the procurement agency. Cheaper, and it is what the permutation grouping points at. It is also an intervention, and the model is an association. Nothing in the analysis establishes that lowering intake moisture at those sites lowers spoilage, because intake moisture has never varied at those sites independently of everything else about them.

Ask for the board paper to be delayed by a season and run the only thing that would settle it.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

The paper went to the board with the refurbishment reduced to one warehouse and an explicit statement that the model could not distinguish between the building and the grain arriving at it.

The remainder of the budget funded a trial. Nine warehouses were fitted with intake moisture meters and a rejection threshold at fourteen per cent, with the sites chosen so that the high-moisture belt and the rest of the network were both represented, and the assignment within each group made at random. That is the experiment the module recommends, and it cost less than a fifth of the original proposal.

After two seasons the trial showed a real reduction in downgrading at the treated sites, larger in the Hoshangabad belt, which is the answer neither the bar chart nor the permutation grouping could have given.

The bar chart is still produced. It now carries a printed line underneath it saying what it measures and what it does not, because Devika found that removing it altogether simply meant somebody generated their own.

Why did a column of random integers rank fourth?

Because impurity importance sums how much each feature reduced impurity across all splits, and a feature with ten thousand distinct values offers a very large number of candidate thresholds. Some of them will separate the training rows by chance. The metric is measuring what it claims to measure — contribution to fitting the training data — and that is not the same as carrying signal. High-cardinality columns and continuous columns accumulate importance for structural reasons.

Permuting either the warehouse or the moisture column alone barely moved the score. What would you have concluded if you had stopped there?

That neither mattered, which is exactly backwards. When two features carry nearly the same information, shuffling one leaves the model able to read the other, so the score holds and both appear unimportant. The remedy is to cluster correlated columns and permute the group together, which measures what the group is worth. That is usually the question anyone actually has.

The permutation grouping pointed at intake moisture. Why was that still not enough to spend the budget on moisture control?

Because importance is a statement about the model, and the proposal is an intervention in the world. In this data intake moisture never varies independently of the site, so the association cannot separate 'wet grain causes spoilage' from 'these buildings cause spoilage and also receive wet grain'. Only an experiment that varies intake moisture while holding the site fixed can answer it, and that is what the randomised trial did.

MODULE 6 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 A node holds 20 rows, 6 of them positive. A candidate split sends 8 rows left with 5 positives and 12 right with 1 positive. How does the tree score that split?

A. By the parent's impurity minus the weighted impurity of the two children.

B. By the number of rows in each child that a majority vote inside that child would get right.

C. By the difference between the two children's positive rates, which here is large.

D. By the accuracy of the whole tree once this split has been made and the rest regrown.

2 Why does a random forest consider only a random subset of the features at each split?

A. To reduce training time, since fewer candidate thresholds have to be searched per node.

B. To prevent any single tree from overfitting, because each tree now sees fewer columns.

C. Because correlated features would otherwise be counted twice in the ensemble's vote.

D. Because less similar trees remove more variance when averaged.

3 Adding trees never makes a random forest worse, but it does make a boosted model worse. Why?

A. Boosting uses deeper trees than a forest does, and deeper trees overfit more readily.

B. Each new boosted tree fits the residual, and eventually the residual is noise.

C. Boosting is sequential, so an error made by an early tree accumulates through the rest.

D. Forests average their trees, and averaging is a form of regularisation that boosting lacks entirely.

4 In 1,000 raw dimensions a point's nearest neighbour is only about 20 per cent closer than its farthest. Which methods does that break?

A. Tree splits, because a tree has to search every dimension for a useful threshold.

B. Gradient descent, because the loss surface becomes a narrow ravine in every direction.

C. Anything whose prediction depends on which points are nearby.

D. One-hot encoding, because the resulting columns become too sparse to store efficiently.

5 Naive Bayes classifies documents well while reporting probabilities above 0.99 for most of them. What produces that combination?

A. The smoothing constant is set too small, so rare words come to dominate the product.

B. The model is overfitting, because it memorises the vocabulary of the training documents.

C. Log-space arithmetic underflows on long documents, pushing the outputs towards the extremes.

D. Correlated words are counted as independent evidence, which inflates confidence.

6 Two columns hold the same measurement in different units. Permutation importance reports near-zero importance for both. What is the correct reading?

A. Shuffling one leaves the other readable, so permute the group together.

B. The measurement is genuinely unimportant, so both columns can safely be dropped from the model.

C. Permutation importance is only valid on training data, so it should be recomputed there instead.

D. One column is redundant, so the importance was split evenly between the two of them.

MODULE 7

Learning without labels, and learning a representation

What you can do when nobody has labelled anything: group similar rows, compress many columns into a few, find the rows that do not belong. Then the idea underneath modern search, recommendation and retrieval — an embedding, what it is geometrically, how to get one for free, and how to search a million of them in milliseconds.

> BY THE END OF THIS MODULE YOU CAN
>
> Cluster a dataset and defend the number of clusters with more than an elbow plot, reduce dimensionality with PCA and state how much information the reduction discarded, explain what an embedding is as a geometric object, and build a working nearest-neighbour search over pretrained embeddings without a GPU

---

63 · 9 min

k-means, and what it assumes about your data

> THE ONE THING TO KEEP
>
> k-means minimises within-cluster squared distance, which assumes spherical clusters of similar size with no outliers and a known k, and it will return confident groupings from data that has no group structure at all.

The algorithm in four lines

1. Pick k random points as initial centres.

2. Assign every row to its nearest centre.
3. Move each centre to the mean of the rows assigned to it.
4. Repeat 2 and 3 until nothing moves.

That is the whole of k-means. It converges quickly, it is easy to implement, and it is the most used clustering algorithm by a wide margin.

```
from sklearn.cluster import KMeans
km = KMeans(n_clusters=5, n_init=10, random_state=0).fit(X_s-
caled)
km.labels_, km.cluster_centers_, km.inertia_
```

`inertia_` is the sum of squared distances from each point to its assigned centre. It is what the algorithm minimises, and it will feature in the next lesson.

What it is quietly assuming

Every step above encodes an assumption, and each one fails on some real dataset in a way that produces confident nonsense.

Clusters are spherical. Step 2 assigns by Euclidean distance to a centre, so the region belonging to each cluster is bounded by straight lines equidistant between centres. Two elongated, parallel groups get cut across rather than separated along their length. If your data has stretched or curved groups, k-means will not find them, and it will still return an answer.

Clusters are similar in size and density. The mean is pulled towards wherever the points are, so a large dense cluster next to a small sparse one tends to have its centre annexe part of the small one.

Every point belongs somewhere. There is no "none of these". An outlier 40 standard deviations away is assigned to whichever cluster is nearest and drags that centre towards it.

k is known. The algorithm cannot tell you there are really three groups when you asked for five. It will produce five.

Initialisation matters more than people expect

k-means finds a local minimum, and different starting centres reach different ones. The classic failure is two centres landing inside one real cluster while a genuine second cluster shares a third.

`k-means++` initialisation — the default in scikit-learn — chooses starting centres that are spread out, by picking each new centre with probability proportional to its squared distance from the nearest existing one. It reduces bad outcomes substantially, and `n_init=10` runs the whole thing ten times and keeps the best inertia, which reduces them further.

Do not lower `n_init` to save time unless you have measured that it does not matter for your data.

Scaling, again

k-means is a distance method, so a feature in rupees dominates one in years completely. Standardise before clustering, always.

There is a subtlety worth noticing. Standardising asserts that every feature should count equally in the distance, and that is itself a choice about what similarity means. If age genuinely matters twice as much as income for your purpose, you can say so by scaling age by two after standardising — clustering has no target to tell it what matters, so what matters is entirely encoded in how you scaled.

This is the deepest thing about unsupervised learning: **the answer depends on your definition of similar, and you supply that definition whether or not you notice.**

Variants worth knowing

- **MiniBatchKMeans** — updates centres from small random batches. Much faster on large data, slightly worse clusters. Above a few hundred thousand rows, use it.
- **Gaussian mixture models** — fit ellipse-shaped clusters with orientation, and give each point a probability of belonging to each cluster rather

than a hard assignment. Strictly more flexible; k-means is the special case with spherical, equal-sized components and hard assignment.

- **k-medoids** — uses actual data points as centres rather than means, so the centre is a real example you can look at, and it tolerates outliers better.

What clusters are actually for

Honesty about the use case, because clustering is often run without one.

- **Segmentation** for marketing or product decisions — the most common use, and the one most likely to produce groups that get named and then treated as real.
- **Compression** — replace a row with its cluster id. Genuinely useful as a feature for a supervised model.
- **Exploration** — look at the centres to understand what kinds of rows exist. Frequently the best value.
- **Labelling triage** — cluster unlabelled data, label a few examples per cluster, and you have a rough labelled set for a fraction of the effort.

What clusters are *not*: a discovery of real categories in the world. k-means will partition uniformly random noise into five tidy groups and report a respectable inertia. The groups will be reproducible, describable and entirely artefactual. Before believing any clustering result, run it on shuffled data and see whether the output looks meaningfully different.

BEFORE YOU MOVE ON

A team clusters customer data into 4 groups and finds one enormous cluster and three tiny ones. The features include annual spend, which ranges from 0 to 40 million, alongside standardised behavioural features. What is the most likely cause?

A. Annual spend was not scaled, so distances are dominated by it and the three small clusters are just the extreme spenders split off from everyone else.
B. k=4 is too small, and increasing it would split the large cluster into meaningful groups.
C. k-means++ initialisation placed three centres close together, and n_init should be raised.
D. The data has no cluster structure, so any partition will be arbitrary and unbalanced.

64 · 8 min

Choosing k, and why the elbow rarely helps

THE ONE THING TO KEEP

The elbow rarely produces an unambiguous answer; use silhouette per cluster, the gap statistic — which can say that no clustering is right — and stability across subsamples, and accept that the useful k is often set by what the organisation can act on.

The elbow method, and its problem

Plot inertia against k. Inertia always falls as k rises — at k = n every point is its own cluster and inertia is zero — so you look for the "elbow" where the improvement slows.

On textbook data with well-separated blobs, the elbow is obvious. On real data it usually is not. The curve is a smooth decay with no clear corner, three peo-

ple looking at the same plot pick three different values, and the method provides no way to adjudicate.

It is worth plotting, because it takes one line and occasionally the elbow is real. It is not worth treating as an answer.

Silhouette score

A better measure. For each point, compute:

- `a` = its mean distance to other points in its own cluster.
- `b` = its mean distance to points in the nearest other cluster.
- silhouette = `(b - a) / max(a, b)`.

Runs from -1 to +1. Near +1 means the point is much closer to its own cluster than to any other. Near 0 means it sits on a boundary. Negative means it is closer to a different cluster than to its own — it is probably assigned wrongly.

```
from sklearn.metrics import silhouette_score, silhouette_sam-
ples
for k in range(2, 11):
    labels = KMeans(k, n_init=10,
random_state=0).fit_predict(X)
    print(k, round(silhouette_score(X, labels), 3))
```

The mean silhouette gives a number per k that you can compare. As a rough guide, above 0.5 is a solid structure, 0.25 to 0.5 is weak, and below 0.25 means there is probably no cluster structure worth naming.

Better still, plot `silhouette_samples` per cluster rather than reading only the mean. A configuration with an excellent mean can contain one cluster where most points are near zero, and that cluster is the finding.

Other numerical criteria

Davies-Bouldin — average similarity between each cluster and its most similar neighbour. Lower is better. Fast, and it also prefers spherical clusters.

Calinski-Harabasz — ratio of between-cluster to within-cluster dispersion. Higher is better. Tends to favour larger k.

Gap statistic — compares your inertia against the inertia obtained on uniformly random data with the same bounding box. This is the one with the most useful property: it can tell you that **k = 1 is best**, meaning there is no cluster structure at all. Every other method above assumes at least two clusters exist and never questions it.

Run the gap statistic, or its poor-man's version — cluster a shuffled copy of your data and compare — on any clustering you plan to act on.

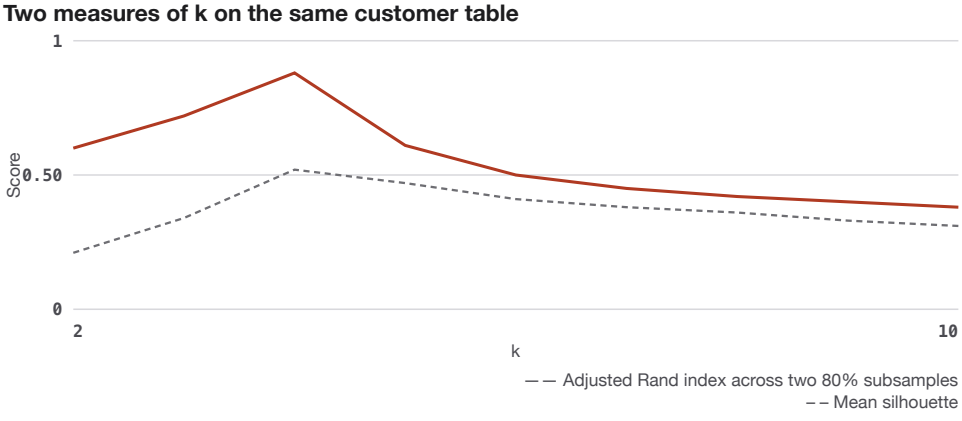
Stability, which is the most practical test

A cluster structure that survives resampling is more likely to be real than one that does not.

Take two random 80% subsets of your data, cluster both, and measure how much the assignments agree on the rows that appear in both, using the adjusted Rand index. Repeat for each candidate k.

```
from sklearn.metrics import adjusted_rand_score
```

The adjusted Rand index corrects for chance agreement, so 0 means random and 1 means identical. A k whose clusters reproduce at 0.85 across subsamples is telling you something. A k that reproduces at 0.3 is telling you the boundaries are arbitrary, whatever its silhouette says.



Both agree on k = 4: the mean silhouette clears 0.5 there and only there, and clusterings of two random 80 per cent subsamples agree at an adjusted Rand index of 0.88. At k = 5 the agreement collapses to 0.61, which says the fifth boundary is arbitrary whatever the silhouette says.

This is slower than the elbow and considerably more informative.

The answer nobody puts in the textbook

Often k is decided by what happens next, not by the data.

If the marketing team can run four campaigns, k is 4. If the app has room for six recommendation shelves, k is 6. If a human has to write a description of each segment, k is under 10 because nobody will read twelve.

This is not a failure of rigour. Clustering is usually a tool for making a decision, and the decision has constraints the data does not know about. State the constraint, pick the k that satisfies it, and use the metrics to check whether the result is stable enough to rely on — which is a different and more answerable question than "what is the true k".

When to stop trying

If silhouette is 0.15 for every k, the gap statistic prefers k = 1, and subsampled clusterings disagree, the answer is that your data has no cluster structure. That is a legitimate finding and it is worth reporting as one. Continuing until a partition emerges will always succeed, and the result will describe your persistence rather than your customers.

BEFORE YOU MOVE ON

A clustering has mean silhouette 0.61 at k=5, which looks strong. Clustering two 80% subsamples and comparing gives an adjusted Rand index of 0.28. What should the team conclude?

A. The silhouette is the more reliable measure since it uses all the data, so the clustering can be trusted.
B. The subsample size is too small at 80%, and repeating with 95% subsamples would raise the agreement.
C. The high silhouette shows the clusters are compact and separated *as fitted*, but the low agreement across subsamples shows the boundaries move when the data changes, so the specific partition should not be treated as a real structure.
D. Adjusted Rand index below 0.5 always indicates a bug in the cluster-matching code rather than genuine instability.

65 · 8 min

DBSCAN and hierarchical clustering: when shape and structure matter

THE ONE THING TO KEEP

DBSCAN defines clusters as dense regions, so it finds arbitrary shapes, decides the cluster count itself and labels outliers as noise — at the cost of assuming one density threshold suits the whole dataset, which HDBSCAN removes.

DBSCAN: clusters as dense regions

k-means asks which centre a point is nearest. DBSCAN asks a different question: is this point in a crowded neighbourhood?

Two parameters define the answer. `eps` is a radius. `min_samples` is how many points must be within that radius for a point to count as a **core point**. Core points that are within `eps` of each other join into one cluster. Points near a cluster but not themselves core are **border points**, and points in neither category are labelled **noise**, with cluster id -1.

```
from sklearn.cluster import DBSCAN
labels = DBSCAN(eps=0.5, min_samples=5).fit_predict(X_scaled)
(labels == -1).sum()      # how many were called noise
```

Three consequences follow directly, and they are exactly the things k-means could not do.

Clusters can be any shape. Two interleaved crescents, a ring around a blob, a long winding band — DBSCAN follows density, so shape is irrelevant. k-means cannot find any of them.

You do not specify the number of clusters. It emerges from the density structure.

Outliers are labelled as outliers rather than being forced into a group. For many purposes this alone justifies the method.

Where DBSCAN struggles

Varying density. One `eps` for the whole dataset means a value that separates a dense region will merge everything in a sparse one, and a value tuned for the sparse region will fragment the dense one. This is the main practical limitation.

HDBSCAN solves it by building a hierarchy across all density levels and extracting the most stable clusters from it. It has one intuitive parameter — the minimum cluster size — and it is generally the better default now. It is a separate free package and it is worth installing.

Choosing eps. The standard method: compute each point's distance to its `min_samples`-th nearest neighbour, sort those distances, and plot them. The curve has a knee, and `eps` at the knee is a good starting value. This is a more

reliable elbow than the k-means one because the quantity being plotted is directly meaningful.

High dimensions. Density is a volume-based notion, and volume behaves badly in high dimensions for the reasons in the k-NN lesson. Reduce dimensions first.

Hierarchical clustering: the whole tree

Instead of one partition, build a nested sequence. Agglomerative clustering starts with every point as its own cluster and repeatedly merges the two closest, recording the order. The result is a dendrogram — a tree you can cut at any height to get any number of clusters.

```
from scipy.cluster.hierarchy import linkage, dendrogram, fclus-
ter
Z = linkage(X_scaled, method="ward")
labels = fcluster(Z, t=5, criterion="maxclust")
```

The **linkage** choice defines what "closest" means between two clusters, and it changes the results substantially:

- **Ward** minimises the increase in within-cluster variance. Produces compact, similar-sized clusters. The usual default.
- **Complete** uses the maximum distance between members. Compact clusters, sensitive to outliers.
- **Average** uses the mean distance. A middle option.
- **Single** uses the minimum distance. Can follow chains of points, which finds elongated shapes and also produces the chaining effect where two distinct clusters merge because of one bridging point.

The dendrogram itself is often the real output. Being able to see that two segments merge at a low height (they are similar) while a third joins only at the top (it is genuinely different) tells you more than any flat partition, and it is the reason hierarchical clustering survives in fields like biology where the nesting is the object of interest.

Its cost is quadratic in memory and roughly cubic in time, so it is impractical above about 20,000 rows without approximation.

Choosing between the three

- Roughly spherical groups, large data, need speed: **k-means**.
- Arbitrary shapes, unknown cluster count, outliers you want identified: **HDBSCAN**.
- Small data, and you want to see the nesting: **hierarchical with a dendrogram**.
- You want soft assignments and elliptical clusters: **Gaussian mixture**.

And run at least two of them. If k-means and HDBSCAN broadly agree, the structure is probably real. If they disagree completely, that disagreement is the most informative result you have, and it usually means the groups are not well separated in the space you built.

BEFORE YOU MOVE ON

A dataset has two dense city clusters and one sparse rural spread. DBSCAN with a single eps either merges the cities or shatters the rural region into noise. What is the underlying limitation?

A. DBSCAN requires standardised features, and the geographic coordinates were left unscaled.

B. min_samples is too high for the rural region, and lowering it would fix both cases.

C. DBSCAN cannot find non-convex clusters, which is what a dispersed rural region is.

D. A single eps sets one density threshold for the whole dataset, and no single value can suit regions whose densities differ by an order of magnitude — which is what HDBSCAN's hierarchy across density levels addresses.

66 · 9 min

PCA: fewer columns, and what you gave up

THE ONE THING TO KEEP

PCA rotates to axes ordered by variance and keeps the first few, which requires standardised features and finds only linear structure — and because it never sees the target, the variance it discards can be the signal you needed.

The idea

Find the direction in which the data varies most. Call it the first principal component. Then find the direction of greatest remaining variation that is perpendicular to the first. Repeat.

The result is a new set of axes, ordered by how much of the data's variance each one carries. Keep the first few and you have compressed many columns into a few while losing as little variance as possible.

```
from sklearn.decomposition import PCA
p = PCA(n_components=0.95).fit(X_scaled)     # keep 95% of
variance
p.n_components_                               # how many that
took
p.explained_variance_ratio_[:5]
```

Passing a float between 0 and 1 lets PCA choose the number of components for you, which is usually what you want. "38 components retain 95% of the variance" is a more useful statement than "we kept 10".

Two prerequisites

Standardise first. PCA maximises variance, and a feature measured in rupees has more variance than one in years for reasons that have nothing to do

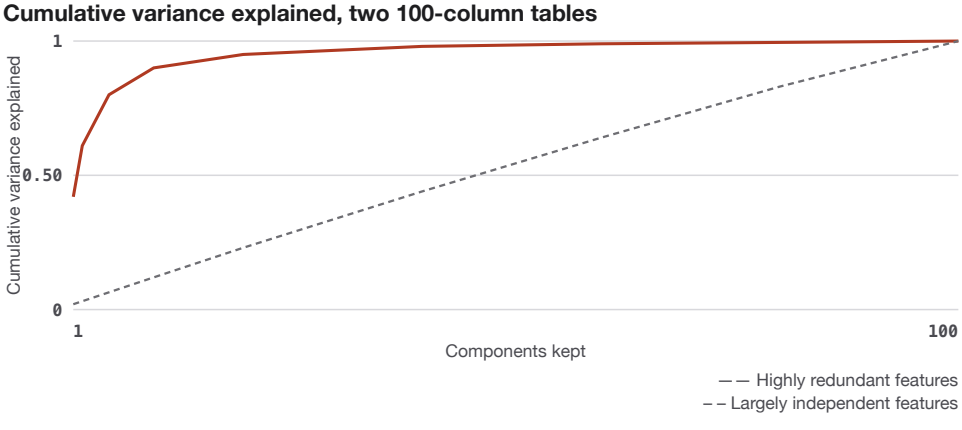
with importance. Without scaling, the first component is whatever feature has the largest units. This is the most common PCA mistake.

PCA finds linear structure only. It fits a rotated set of straight axes. Data on a curved surface — a spiral, a circle — is not compressed well by any linear projection, and PCA will report that it needs many components for something a human can see is one-dimensional. Kernel PCA and the neighbour-embedding methods in the next lesson exist for that case.

Reading the output

Three things are worth looking at every time.

The cumulative explained variance curve. Plot the cumulative sum of `explained_variance_ratio_`. A sharp rise that flattens early means your features are highly redundant. A near-straight diagonal means they are largely independent and PCA will not help you much.



The redundant table reaches 95 per cent of its variance with 20 components; the independent one needs nearly all hundred, and PCA buys almost nothing. The axis is variance, not usefulness. The target was never consulted, and the signal you need can sit in a component this curve tells you to discard.

The loadings. `p.components_[0]` gives the weight of each original feature in the first component. This is how you interpret what a component means. In survey data the first component is often "overall agreement", because someone who answers positively to one item tends to answer positively to all. In financial data it is often "size".

The reconstruction error. `p.inverse_transform(p.transform(X))` maps back to the original space. Comparing that with `X` shows exactly what was lost, per row. Rows with high reconstruction error are rows the components fail to describe, which is a serviceable anomaly detector in itself.

What PCA is genuinely good for

- **Speeding things up.** 500 correlated features to 40 components makes downstream training an order of magnitude faster with little accuracy loss.
- **Removing collinearity.** Components are orthogonal by construction, which fixes the coefficient instability from module 2 completely.
- **Denoising.** Small-variance components often capture measurement noise; dropping them can improve a downstream model.
- **Visualisation.** Two components, one scatter plot, and you can see whether the classes separate at all before building anything.
- **Preprocessing for distance methods.** Reducing to 30 dimensions before k-NN or clustering makes distances meaningful again.

What it costs you

Interpretability. Component 1 is `0.31 * income + 0.28 * tenure − 0.19 * complaints + ...` across every feature. You can no longer say "income matters most", and for a model that must be explained to a regulator this is often disqualifying.

Variance is not the same as usefulness. PCA has never seen your target. It is entirely possible for the signal you need to live in the 12th component, carrying 0.4% of the variance, and for you to have discarded it. This happens, and it is the main reason PCA is not a routine preprocessing step for supervised learning.

If you want a projection that knows about the target, **linear discriminant analysis** finds directions that separate the classes rather than directions of greatest variance. It is the supervised counterpart and is limited to `n_classes − 1` components.

Two practical variants

TruncatedSVD works on sparse matrices without centring them, which PCA cannot do. On TF-IDF matrices this is called latent semantic analysis, and it was the standard way to get dense document vectors for two decades before neural embeddings.

IncrementalPCA processes the data in batches, so you can fit it on more data than fits in memory.

And put it in the pipeline

PCA learns the components from data, so it must learn them from training rows only. Fitting PCA on the full dataset before splitting is on the leakage list in module 3 for exactly this reason, and it is one of the more commonly missed entries on that list.

BEFORE YOU MOVE ON

A team applies PCA to 200 features, keeps components covering 95% of variance, and finds their classifier's AUC drops from 0.86 to 0.79. What is the most likely explanation?

A. 95% is too aggressive a threshold, and keeping 99% would always recover the lost performance.

B. PCA orders components by variance without reference to the target, so a discriminative direction carrying little variance can be among the 5% discarded.

C. PCA requires the target to be included in the matrix so the components align with the outcome.

D. The components are orthogonal, which removes the feature interactions the classifier relied on.

67 · 8 min

t-SNE and UMAP, and how they mislead

THE ONE THING TO KEEP

t-SNE and UMAP preserve local neighbourhoods and deliberately distort cluster sizes and between-cluster distances, so the only reliable information in the plot is which points are neighbours — and they produce convincing clusters from pure noise.

A different goal from PCA

PCA preserves global structure: distances and directions across the whole dataset, as well as a linear projection can. It is often disappointing for visualisation, because two dimensions of a 300-dimensional dataset usually look like a blob.

t-SNE and UMAP have a different objective: preserve **local neighbourhoods**. Points that were close in the original space should be close in the plot. Points that were far can go anywhere.

That trade produces much more revealing pictures, and it is also the source of every way these plots deceive people.

```
from sklearn.manifold import TSNE
emb = TSNE(n_components=2, perplexity=30, init="pca",
           random_state=0).fit_transform(X_reduced)
```

Four things the plot does not tell you

1. Cluster sizes are meaningless. t-SNE expands dense regions and contracts sparse ones, deliberately, so that dense clusters do not collapse into dots. A large blob is not a large cluster.

2. Distances between clusters are meaningless. Two clusters at opposite ends of the plot are not more different than two adjacent ones. Only the fact that points within a cluster are neighbours is reliable.

3. Apparent clusters can be artefacts. Run t-SNE on uniform random noise with a low perplexity and it produces convincing-looking clusters. This is not a rare pathology; it happens readily. Always run the same procedure on shuffled data before believing a structure you see.

4. The result changes with the seed and the parameters. `perplexity` roughly sets how many neighbours each point tries to stay near, and values of 5, 30 and 100 on the same data can give visibly different pictures. Run several and report what is stable across them.

UMAP

UMAP does a similar job with a different mathematical construction and three practical advantages: it is much faster on large data, it preserves more global structure than t-SNE (though still not enough to trust distances), and it can `transform` new points into an existing embedding, which t-SNE cannot do at all.

```
import umap
reducer = umap.UMAP(n_neighbors=15, min_dist=0.1,
random_state=0)
emb = reducer.fit_transform(X_reduced)
```

`n_neighbors` trades local against global structure — low values emphasise fine local detail, high values the broad shape. `min_dist` controls how tightly points may pack, which affects appearance more than substance.

UMAP is a free package and is now the more common choice. Its global structure is better than t-SNE's and still not a metric you should quote.

Run PCA first

Both methods are slow and both suffer in very high dimensions. The standard practice is to reduce to 30 to 50 components with PCA and then run t-SNE or

UMAP on that. It is faster, it removes some noise, and it usually improves the picture.

What these are legitimately for

- **Looking at your data before modelling.** If the classes visibly separate, a simple model will probably work. If they are thoroughly mixed, expect a hard problem — although note the converse does not hold, since a projection that fails to separate them does not prove no model can.
- **Finding data problems.** Duplicate rows appear as unnaturally tight points. A mislabelled group shows up as a coloured island inside another cluster. Batch effects — data collected on different days or instruments — appear as separation by batch rather than by anything you care about, and this is one of the most valuable things these plots reveal.
- **Inspecting embeddings.** Colour points by a known attribute and see whether the representation has organised itself by it.

What they are not for

They are not a preprocessing step for a supervised model. The output coordinates have no stable meaning, t-SNE cannot transform new data at all, and UMAP's transform is not guaranteed stable enough to serve as production features. Use PCA if you need a reduction that a model will consume.

And they are not evidence. A convincing t-SNE plot is a hypothesis. Confirm it with a number — a silhouette score, a classifier that can tell the groups apart, a stability check — before it appears in a document that someone will act on.

The cheapest confirmation is usually a classifier. If you believe the plot shows two real groups, label the points by which group they appear in and train a simple model to predict that label from the *original* features, evaluated on held-out rows. A group that a logistic regression can recover at 0.9 AUC from three interpretable columns is a real and describable structure. One that no model can recover above chance was a shape in the projection and nothing more.

BEFORE YOU MOVE ON

A t-SNE plot shows two tight clusters far apart and a third large diffuse one between them. What can be legitimately concluded?

A. The diffuse cluster is noise, since real clusters appear tight under t-SNE.

B. The third cluster contains more points, since it occupies more area.

C. The two tight clusters are more different from each other than either is from the diffuse one.

D. Points within each visible group were neighbours in the original space; nothing about the groups' sizes or separations can be read from the plot.

68 · 10 min

What an embedding actually is

THE ONE THING TO KEEP

An embedding maps things to vectors so that geometric closeness means similarity in the sense the training task defined, which is why the vectors are compared by angle rather than length and why they carry the training corpus's biases in their geometry.

A definition worth being precise about

An embedding is a mapping from things — words, images, products, users, sentences — to vectors of real numbers, arranged so that **geometric closeness corresponds to similarity in whatever sense the training set defined.**

That last clause carries all the weight. There is no universal notion of similarity. An embedding is similar-in-a-particular-sense, and which sense depends entirely on what it was trained to do.

Why one-hot is not enough

One-hot encoding gives every category its own dimension. "Cat" is `[1, 0, 0, ...]` and "dog" is `[0, 1, 0, ...]`. The distance between cat and dog is exactly the distance between cat and "quantitative easing". Every pair of distinct categories is equidistant, so the representation cannot express any relationship at all.

An embedding gives "cat" something like `[0.24, −0.81, 0.15, ...]` in 300 dimensions, with "dog" nearby and "quantitative easing" far away. It has traded 50,000 sparse dimensions for 300 dense ones and gained the ability to represent degrees of similarity.

Where the numbers come from

Nobody chooses them. They fall out of training a model on a task where similarity is useful.

word2vec (2013) trains a small network to predict a word from its neighbours, or the neighbours from the word. The training signal is only "which words appear near which", and the vectors are a side effect — the hidden layer weights. Words appearing in similar contexts end up with similar vectors, which is the distributional hypothesis from linguistics: a word is characterised by the company it keeps.

Matrix factorisation for recommendations decomposes a users-by-items matrix into user vectors and item vectors whose dot product reconstructs the ratings. The vectors are what makes the reconstruction work.

Modern sentence embeddings train on pairs known to be related — a question and its answer, a sentence and its translation — with a contrastive loss that pulls related pairs together and pushes unrelated ones apart.

In every case the embedding is not the goal of training. It is the internal representation a model built because that representation made the task easier, and it turns out to be useful elsewhere.

The arithmetic, and how much to believe

The famous demonstration: `king − man + woman` lands near `queen`. Directions in the space appear to encode relationships — a gender direction, a plural direction, a capital-city direction.

It is real and it is oversold. Later analyses showed the result depends on excluding the input words from the candidate answers, that many analogies fail, and that some apparent successes come from the query word simply being near the answer already. Treat it as a striking property of these spaces rather than a reliable operation.

What is robust is that **similarity works.** Nearest neighbours in a good embedding space are genuinely related, and that is what everything downstream is built on.

Cosine, not Euclidean

Embeddings are almost always compared by cosine similarity — the cosine of the angle between two vectors — rather than by Euclidean distance.

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

The reason is that vector *length* often encodes something other than meaning: word frequency, document length, how confident the model was. Two documents about the same topic at different lengths point in the same direction with different magnitudes. The angle captures the topic; the length captures something else.

A useful practical note: if you normalise every vector to unit length first, cosine similarity and Euclidean distance give the same ranking, and the dot product becomes the cosine directly. Most vector databases normalise on the way in for exactly this reason.

What an embedding inherits

The geometry reflects the training data, including its biases. Word embeddings trained on web text place occupations near gendered words in patterns that match the corpus rather than any ideal — the Bolukbasi and Caliskan papers documented this in 2016 and 2017, and the finding has been reproduced many times since.

Debiasing methods exist and are contested: some remove the measured association without removing the underlying structure, so the bias remains detectable by other probes. This is unresolved. What is not unresolved is that an embedding is a compressed summary of its training corpus, and anything systematic in that corpus is in the geometry.

Where they are used

- **Semantic search.** Embed the query and the documents, return the nearest. Finds "how do I cancel my subscription" for a document titled "ending your plan", which keyword search cannot.
- **Recommendation.** Items near the ones a user liked.
- **Retrieval-augmented generation.** Fetch relevant passages by embedding similarity, then hand them to a language model.
- **Deduplication and clustering.** Near-duplicates are near neighbours.
- **Features for a classifier.** Embed the text, feed the vector to logistic regression. Often better than TF-IDF, and one line either way, so measure both.

And inside a transformer, the first thing that happens to a token is that it becomes an embedding. The whole architecture operates on vectors of this kind, which is why understanding what one is matters well beyond the classical setting.

BEFORE YOU MOVE ON

A team embeds product descriptions and finds that a 200-word description and a 20-word description of the same product have low cosine similarity, though the words overlap heavily. What is the most likely cause?

A. Cosine similarity is length-sensitive, so the differing document lengths reduce the score directly.

B. The embedding model was applied beyond its maximum input length, so the 200-word description was truncated and the vector represents only its opening.

C. Embeddings require standardised inputs, and the descriptions were not normalised before encoding.

D. Cosine similarity is unsuitable for text and Euclidean distance should be used instead.

69 · 9 min

Building a working semantic search with no GPU

THE ONE THING TO KEEP

Free CPU-sized embedding models plus an approximate nearest-neighbour index give working semantic search, but chunking affects quality more than model choice and pure embedding search fails on identifiers and negation, which is why production systems combine it with keyword search.

Getting embeddings for free

You do not train these yourself. Pretrained models are free, small enough to run on a CPU, and good enough for most applications.

```
from sentence_transformers import SentenceTransformer
m = SentenceTransformer("all-MiniLM-L6-v2")
vecs = m.encode(list_of_texts, normalize_embeddings=True)
vecs.shape        # (n, 384)
```

`all-MiniLM-L6-v2` is about 80 MB, produces 384-dimensional vectors, and encodes a few hundred short texts per second on an ordinary laptop CPU. It is the sensible default.

When you need better quality, `all-mpnet-base-v2` is larger and slower. For non-English or mixed-language text, the multilingual models — `paraphrase-multilingual-MiniLM-L12-v2` and its relatives — cover 50 or more languages in one shared space, which means a Hindi query can retrieve an English document. For Indian languages specifically, `MuRIL` from Google Research is free on Hugging Face and trained on 17 Indian languages including transliterated text.

All of these run locally. No API key, no per-call cost, no data leaving the machine — which matters when the documents are confidential.

Searching a million vectors

Comparing a query against a million vectors by brute force is a million dot products of 384 numbers each. In NumPy that is one matrix multiplication and takes a few hundred milliseconds — genuinely fine for many applications, and you should measure before assuming otherwise.

```
sims = corpus_vecs @ query_vec        # both normalised
top = np.argpartition(-sims, 10)[:10]
```

Above a few million, use an approximate index. These trade a small amount of recall for very large speedups.

- **FAISS** — free, from Meta, the standard. Runs on CPU. `IndexFlatIP` is exact; `IndexHNSWFlat` is the usual approximate choice.
- **hnswlib** — a small, fast standalone HNSW implementation.

- **Annoy** — from Spotify, memory-mapped so an index larger than RAM can be searched.

HNSW builds a layered graph of neighbours and walks it from a coarse layer down to a fine one. It typically achieves 95 to 99% recall of the true nearest neighbours at a hundred times the speed of brute force. That recall figure is a parameter you set, and it is worth measuring on your own data rather than accepting the default, because "the tenth result is sometimes wrong" matters more for some products than others.

Chunking, which decides more than the model

For documents longer than the model's input limit — often 256 or 512 tokens — you must split before embedding. This choice affects retrieval quality more than the choice of model, and it is usually made carelessly.

- **Too large**, and one vector averages several topics, so it matches everything weakly and nothing strongly.
- **Too small**, and the chunk lacks the context needed to be interpretable. A sentence beginning "This is not recommended" is useless without its antecedent.
- **Overlap** of 10 to 20% keeps sentences that straddle a boundary retrievable from both sides.
- **Split on structure where you can** — paragraphs, sections, headings — rather than on a fixed character count. A chunk that respects the document's own boundaries is more coherent than one that ends mid-sentence.

A reasonable starting point for prose is 300 to 500 tokens with 50 tokens of overlap, split at paragraph boundaries, and then measured.

Measuring retrieval

Do not deploy on the impression that results look reasonable. Build a small evaluation set: 50 real queries with the document that should be returned for each, written by hand. Then compute recall at 5 and mean reciprocal rank.

Fifty queries takes an afternoon and turns every subsequent decision — model, chunk size, index type — into a measurement rather than an argument. This is the same discipline as the rest of this course applied to a system that superficially seems not to need it.

Where pure embedding search fails

Three failure modes worth knowing before you commit.

Exact identifiers. Searching for an order number, a part code or an error string. Embeddings represent meaning, and a part number has no meaning to represent. Keyword search wins outright.

Negation and specificity. "Restaurants without parking" embeds close to "restaurants with parking". The vectors capture topic much better than logical structure.

Rare exact terms. A drug name appearing three times in the corpus may be poorly represented, where an inverted index finds it perfectly.

The standard answer is **hybrid search**: run keyword search (BM25) and embedding search in parallel and combine the rankings, usually with reciprocal rank fusion, which needs no score calibration between the two systems. Hybrid reliably beats either alone, it is not much more code, and it is what production systems do.

Why production search runs both

Embedding search wins

- Paraphrase: 'refund not received' finds 'the money has not come back'
- A Hindi query retrieving an English document, in one multilingual space
- Questions phrased nothing like the passage that answers them
- The long tail of wordings for one meaning

Keyword search (BM25) wins

- Exact identifiers: an order number, a part code, an error string
- Negation: 'restaurants without parking' embeds beside 'with parking'
- Rare exact terms: a drug name that appears three times in the corpus
- Anything a user expects to match letter for letter

Hybrid search runs BM25 and embedding search in parallel and fuses the two rankings with reciprocal rank fusion, which needs no score calibration between the systems. It reliably beats either alone and is not much more code.

BEFORE YOU MOVE ON

A support search built on embeddings fails to find the article for the query 'error code E4302' even though the article contains that exact string. What is the mechanism, and what fixes it?

A. The embedding model represents meaning, and an arbitrary code carries none, so the query vector lands nowhere near the article's; adding keyword search and fusing the rankings recovers it.

B. The article chunk containing the code was too long, so the code was averaged away; smaller chunks would fix it.

C. The vectors were not normalised, so cosine similarity was computed incorrectly for short queries.

D. The index is approximate and dropped the correct neighbour; switching to exact search would find it.

70 · 9 min

Recommendation: learning a vector per user and per item

THE ONE THING TO KEEP

Matrix factorisation learns a vector per user and per item by fitting only the observed cells, which makes it an embedding trained on interactions — and its recommendations shape the next dataset, so offline metrics are always measured on data the previous model helped create.

The matrix with holes in it

Users down the rows, items across the columns, ratings in the cells. Almost every cell is empty — a user who has rated 40 films out of 50,000 leaves 99.92% of their row blank. Recommendation is the problem of guessing the missing values.

Two families

Content-based. Describe items by their attributes — genre, price, text — and recommend items similar to what the user liked. Works for a brand-new item with no interaction history, which is its main advantage. Tends to be narrow: someone who watched three thrillers gets only thrillers.

Collaborative filtering. Ignore item attributes entirely and use the pattern of who interacted with what. "People who liked what you liked also liked this." It finds connections no attribute list would suggest, and it fails completely for a new item or a new user, since neither has any pattern yet.

Matrix factorisation, which is an embedding

The central technique. Assume the ratings matrix `R` — users by items — can be approximated by the product of two thin matrices:

```
R (100,000 x 50,000)  is approximately  U (100,000 x 50) x V'
(50 x 50,000)
```

Each user becomes a vector of 50 numbers; each item becomes a vector of 50 numbers; a predicted rating is the dot product of the two.

Those 50 dimensions are learned, not designed. They may turn out to correspond loosely to things a human would name — how much action, how recent, how mainstream — but nobody specified them. This is the same construction as the embeddings in the last two lessons, with the training signal being interaction rather than co-occurrence in text.

The compression is the point. 5 billion possible cells are described by 7.5 million learned numbers, and the reduction forces the model to find structure rather than memorise.

Fitting it on observed cells only

The critical detail. You minimise error over the cells you *have*, not over the whole matrix:

```
minimise  sum over observed (r_ui − u_i . v_u)^2  +  lambda ∗
(||u||^2 + ||v||^2)
```

Treating missing cells as zero would tell the model that every unwatched film was disliked, which is false — most were never seen. The regularisation term is not optional here: with 50 dimensions per user and some users having rated four items, the model will otherwise fit those four perfectly and generalise to nothing.

Alternating least squares is the classic fitting method — hold the item vectors fixed and solve for the user vectors, which is an ordinary least squares problem, then swap. It parallelises well. The free library is `implicit`, and `surprise` is the friendlier one for explicit ratings.

Explicit and implicit feedback are different problems

Explicit — the user gave a rating. Rare, biased towards strong opinions, and you know what a low value means.

Implicit — clicks, views, purchases, watch time. Abundant, and there are **no negatives**. A film you never watched might be one you would love. Treating absence as dislike is wrong, and the standard solution is to treat observed interactions as positives with a confidence weight, and unobserved ones as low-confidence negatives rather than definite ones.

Most real systems are implicit, and getting this framing right matters more than the choice of algorithm.

The cold start problem

The permanent difficulty. A new user has no vector. A new item has no vector. Collaborative filtering has nothing to work with.

The usual answers: fall back to popularity for new users, use content features for new items, ask a few questions during onboarding, and use a hybrid that combines both families. There is no clean solution — this is a structural property of the approach rather than a gap in the technique.

Popularity bias and the feedback loop

A recommender trained on interactions recommends what people interacted with, which causes more interaction with those items, which strengthens the pattern. Popular items become more popular because they were recommended, and the long tail becomes unreachable.

This is the feedback loop from module 3 in its most visible form, and it has two costs: the catalogue effectively shrinks, and the offline evaluation becomes untrustworthy, because your test data was generated by the previous recommender's choices rather than by the users' free preferences.

Mitigations exist and all cost something: inject randomness, penalise popularity in the ranking, optimise for coverage as well as accuracy, and measure diversity as an explicit metric alongside precision. None of them is free, and choosing how much to spend is a product decision.

Evaluating honestly

RMSE on held-out ratings is the classic metric and it is a poor one, because users only rate things they chose to consume — the held-out cells are not a random sample of the matrix.

Better: ranking metrics on held-out interactions, split **by time** rather than randomly, since recommending yesterday's purchase using tomorrow's data is the temporal leakage from module 3. Precision at 10, recall at 10, and NDCG are the standard set. And note that all of them are still measured on a population shaped by the old recommender, which is why the definitive test remains an online experiment.

BEFORE YOU MOVE ON

A team builds a recommender on purchase data and fills unpurchased items with zeros before factorising. Why is this a mistake?

A. Zeros make the matrix dense, so factorisation becomes computationally infeasible at this scale.

B. Zeros should be replaced with the item's mean rating so the model has an unbiased starting estimate.

C. It asserts that every item a user never bought was actively disliked, when almost all of them were simply never seen; the fit should use observed cells only, weighting unobserved ones as weak rather than definite negatives.

D. Zeros break the regularisation term, since the norm of the user vectors is computed over all cells.

71 · 8 min

Anomaly detection, and the base rate that ruins it

THE ONE THING TO KEEP

Anomaly detection models normality instead of learning a boundary, and at a 0.01% base rate even a 1% false positive rate produces a queue that is 99% wrong — so the design work is narrowing the population or ranking, not improving the detector.

A different problem shape

Classification learns the boundary between two classes you have examples of. Anomaly detection learns what normal looks like and flags departures, using mostly or entirely normal data.

That framing is right when the anomalies are too rare to learn from, too varied to have a consistent pattern, or when tomorrow's anomaly will not resemble

today's. Equipment failures, novel intrusions, new fraud schemes — in each case the positive class is not one thing.

Four methods

Isolation Forest. Build random trees by picking a random feature and a random split point. Anomalies get isolated in fewer splits, because they sit in sparse regions where a random cut is likely to separate them. The score is the average path length to isolate a point.

```
from sklearn.ensemble import IsolationForest
iso = IsolationForest(contamination=0.01,
random_state=0).fit(X)
scores = -iso.score_samples(X_new)     # higher means more
anomalous
```

Fast, handles mixed distributions, needs little tuning, scales well. The usual first choice.

Local Outlier Factor. Compares a point's local density with the density of its neighbours. Its advantage is finding *local* anomalies — a point that is unremarkable globally but strange for its immediate neighbourhood. A ₹50,000 transaction is ordinary overall and extraordinary for a customer who has never exceeded ₹2,000.

One-Class SVM. Fits a boundary around the normal region. Sensitive to its parameters and slow above tens of thousands of rows, but useful when you have clean normal-only data.

Reconstruction error. Fit PCA or an autoencoder on normal data, reconstruct a new point, and measure the error. Points the model cannot rebuild are unlike what it learned. Works well with images and sensor data, and PCA reconstruction error is a perfectly serviceable version that needs no neural network.

The contamination parameter

`contamination` tells scikit-learn what fraction of the data to treat as anomalous, and it sets the threshold. Setting it is the same decision as choosing a threshold in module 5, and it is frequently set at the default without thought.

If you have any labelled anomalies at all, use them to pick this by expected cost rather than guessing. If you have none, set it by capacity: how many alerts can somebody actually look at per day?

The arithmetic that sinks most deployments

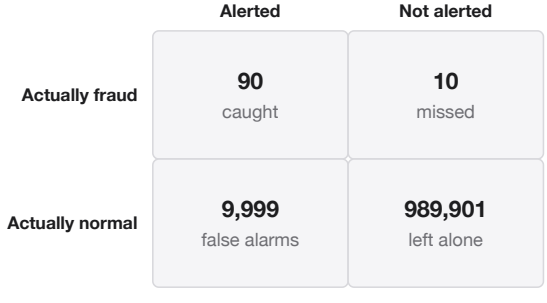
This is the lesson's real content, and it follows from the base rate discussion in module 5.

One million transactions a day. 100 are fraudulent. Your detector catches 90% of them and has a 1% false positive rate, which sounds excellent.

```
True positives  = 90
False positives = 0.01 * 999,900 = 9,999
Precision = 90 / 10,089 = 0.9%
```

Ten thousand alerts a day, of which 99.1% are wrong. No team can work that queue, and within two weeks they will ignore it entirely — which is worse than having no detector, because everyone believes the problem is covered.

A million transactions a day, 100 fraudulent, 90% recall, 1% false positive rate



Precision is 90 / 10,089 = 0.9 per cent: ten thousand alerts a day, 99 of every 100 wrong. Reaching 50 per cent precision needs a false positive rate near 0.01 per cent, a hundred times better — or a smaller population to score, where the base rate is higher and the arithmetic changes.

To get to 50% precision you would need a false positive rate near 0.01%, a hundred times better. That is the actual requirement, and it should be stated before the project starts rather than discovered after.

What to do about it

Narrow the population. Do not score all transactions. Score transactions above ₹10,000 from accounts under 30 days old, where the base rate might be 2% instead of 0.01%. Precision rises by orders of magnitude for free, because the arithmetic changes.

Cascade. A cheap fast filter passes 1% of traffic to an expensive accurate model. Each stage only needs to be good enough not to discard true positives.

Rank, do not alert. Give the team the top 200 by score every morning. Now the metric is precision at 200 and the queue is finite by construction.

Aggregate. One anomalous transaction is weak evidence; five from the same account in an hour is strong. Alerting on entities rather than events raises the base rate of the thing you alert on, which is the same trick as narrowing the population.

And use labels the moment you have any

A supervised model beats an unsupervised one whenever labels exist, and it is not close. Anomaly detection is what you do while you have no labels, or for anomalies unlike anything you have seen. The moment your investigators have confirmed 200 cases, train a classifier on them — and keep the anomaly detector running alongside for the novel cases the classifier has no examples of. The two catch different things, and running both is normal.

BEFORE YOU MOVE ON

A network intrusion detector flags 0.5% of 20 million daily connections. Genuine intrusions number about 50 per day and it catches 80% of them. Why will the security team stop using it?

A. 80% recall is too low for security work, and missing 10 intrusions per day is unacceptable.

B. A 0.5% flag rate on 20 million connections is 100,000 alerts a day against 40 true detections, so precision is about 0.04% and no team can work that queue.

C. Anomaly detectors cannot be evaluated without labels, so the team has no way to know whether it works.

D. The contamination parameter was set too low, causing the model to under-flag genuine intrusions.

72 · 9 min

Self-supervised learning: making labels out of the data itself

THE ONE THING TO KEEP

Self-supervised learning manufactures a supervised task from the data's own structure, and the pretext task chosen decides what the representation encodes — which is why a model pretrained with colour augmentation has learned that colour does not matter.

The trick that changed the field

Labels are expensive. Raw data is abundant. Self-supervised learning bridges the gap by constructing a supervised task out of unlabelled data, using part of the data as the label for the rest.

Hide a word and predict it from its context. Mask a patch of an image and reconstruct it. Take two crops of the same photograph and train the model to

recognise that they came from the same photograph. In every case the supervision comes from the data's own structure, so the training set is as large as the internet.

The representation learned this way transfers. A model that has learned to predict missing words has necessarily learned a great deal about language, and that knowledge is available for any downstream task with only a few hundred labels.

The three main shapes

Masked prediction. Hide part of the input, predict it. BERT masks 15% of tokens; masked autoencoders hide up to 75% of image patches. The task is trivially generated and forces the model to learn context.

Next-item prediction. Predict what comes next from what came before. This is how language models are trained, and it is the same objective as the word2vec idea scaled up enormously.

Contrastive learning. Create two augmented views of the same item — two crops, two colour distortions — and train so those two land close together while views of different items land apart. SimCLR and MoCo established this for images. The subtlety is that the augmentations define what the model is allowed to consider irrelevant: if you augment with colour jitter, the model learns that colour does not matter, which is wrong for a task about ripeness of fruit.

That last point is the general lesson. **The pretext task decides what the representation encodes**, and choosing it is where the design judgement lives.

Semi-supervised learning: a few labels plus a lot of data

A related and more immediately usable idea when you have 500 labels and 50,000 unlabelled rows.

Self-training / pseudo-labelling. Train on the labels you have, predict on the unlabelled data, add the most confident predictions as labels, retrain. It

works when the initial model is decent and confidence is meaningful. Its failure mode is confirmation: a confident early mistake becomes a label and the model learns it thoroughly. Use a high confidence threshold, add data gradually, and check whether performance on your genuine validation set actually improves at each round.

Consistency regularisation. Require the model to give the same output for two perturbed versions of the same unlabelled input. This uses the unlabelled data without ever guessing a label for it, which avoids the confirmation problem, and it is the basis of most modern methods in this area.

Label propagation. Build a graph of similar rows and spread labels along it. Available in scikit-learn as `LabelSpreading`, and effective when your data really does form clusters that align with the classes.

Active learning: choosing what to label

If labelling costs money, choose carefully what to label next.

- **Uncertainty sampling** — label the rows the current model is least sure about, since those are nearest the decision boundary and most informative.
- **Diversity sampling** — label rows unlike anything already labelled, to cover the space.
- **Query by committee** — train several models and label the rows they disagree about most.

A practical loop: label 200 rows, train, score the unlabelled pool, take the 100 most uncertain and the 50 most different, label those, repeat. Compared with random labelling, this routinely reaches the same accuracy for half the labels, and the saving is real money.

The caveat: uncertainty sampling biases your labelled set towards boundary cases, so it is no longer a random sample of the population. That is fine for training and it means you cannot use it as a test set. Keep a separate randomly labelled set for evaluation, always.

What this means for a small team

The practical route for anyone with a real problem and no labels:

1. **Start from a pretrained model** — a sentence embedding model, an image model from `timm`, all free on Hugging Face. Somebody else paid for the self-supervised pretraining.
2. **Label a few hundred examples** using active learning to choose which.
3. **Fine-tune, or just train a linear model on the embeddings.** The second option needs no GPU and is often within a point or two of the first.
4. **Keep a randomly sampled test set** so you can tell whether any of it worked.

That sequence turns "we have no labelled data" from a blocking condition into an afternoon, and it is available to anyone with a laptop and an internet connection.

BEFORE YOU MOVE ON

A team uses uncertainty-based active learning to select 800 rows for labelling, then splits those 800 into train and test. Their test accuracy is far below production performance. Why?

A. 800 rows is too few for a reliable test split, so the estimate is dominated by variance.

B. Active learning selected rows the model was least certain about, so the labelled pool is concentrated near the decision boundary and is systematically harder than the real population.

C. Uncertainty sampling introduces label noise, since uncertain rows are harder for annotators to label correctly.

D. The model was trained on rows it had already scored, which constitutes leakage into the test set.

CASE STUDY · MODULE 7

The segmentation that was not there

A state women's helpline in Kerala with 62,000 call records, a research officer, and a proposal to reorganise counsellor training around caller segments.

The helpline receives calls about domestic violence, workplace harassment, legal questions, mental distress and a long tail of everything else. Counsellors record a structured summary at the end of each call: category, district, caller age band, whether a referral was made, duration, time of day, whether the caller had called before, and a free-text note.

A proposal had been funded to redesign counsellor training around "caller segments" — the idea being that a small number of recognisable caller types exist, that each needs a different opening approach, and that training modules could be built for each. The proposal named six segments as a target, because six is the number of training modules the calendar could absorb.

Nandana, the research officer, was asked to find them. She had 62,000 records, no labels, and a genuine question with money behind it.

She did the work properly. Categorical fields one-hot encoded, duration log-transformed, everything standardised, k-means for k from two to ten, with `n_init` left at ten because lowering it to save time was not something she had measured.

The elbow plot was a smooth decay with no corner. Three colleagues looking at it picked three different values.

Silhouette scores ran between 0.11 and 0.17 across every k she tried. The rough guide in her notes said above 0.5 is solid structure, 0.25 to 0.5 is weak, and below 0.25 means there is probably nothing worth naming.

The gap statistic — which is the only one of these that is allowed to answer "one" — preferred k equal to one. She checked it the cheap way as well, clustering a shuffled copy of the data and comparing, and the shuffled version looked much like the real one.

The stability test was the clearest. Two random eighty per cent subsets, clustered separately, compared on the rows they shared with the adjusted Rand index. At k equal to six the agreement was 0.31. Rerunning with a different seed produced six different groups containing largely different callers, all of which could be described and named, none of which reproduced.

She also ran HDBSCAN, which is allowed to decide the count itself and to label points as noise. It found two small dense groups — repeat callers from two districts with an unusual referral pattern, about 900 records between them — and called everything else noise.

Now the decision.

Deliver six segments. She could. k-means returns six clusters from anything, the centres are describable, and a page of prose about "the hesitant first-time caller" and "the informed repeat caller" would read well and would survive a steering committee. The training modules would be built and would probably do some good, because good counsellor training does some good regardless of whether its organising scheme is real.

Report that there are no segments. True, defensible, and it terminates a funded project. It also tells a room of people who have committed to a plan that the plan's premise does not hold, which is a specific kind of unpopular.

Deliver the two small dense groups and nothing else. Honest, and 900 records out of 62,000 is not a training programme.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

She reported that the data did not support caller segments, and she brought the evidence in an order the committee could follow: the shuffled-data com-

parison first, because it needs no statistics, then the stability numbers, then the silhouette values.

The recommendation that replaced the segments was to organise the training around the two things that did vary and could be measured — call category, which is recorded, and whether the caller had called before, which is also recorded. Both are known at the start of a call, which the cluster assignment would not have been.

The two dense groups from HDBSCAN went to the operations team rather than the training team. Both turned out to be a single referral pathway in two districts that was sending callers back to the helpline because a partner office had changed its intake hours. That was a fixable process fault, found by the method that was permitted to say "most of this is noise".

The proposal document was rewritten rather than abandoned, and the sentence Nandana asked to have kept in it is that a clustering that does not reproduce across subsamples describes the analyst's persistence rather than the callers.

k-means returned six clusters that could be named. Why is that not evidence that six segments exist?

Because k-means partitions whatever it is given. It will return six tidy groups from uniform random noise, with a respectable inertia, and the centres of those groups can always be described in words. Describability is a property of humans, not of the data. The test is whether the same structure appears when the algorithm sees a different sample, and at an adjusted Rand index of 0.31 it did not.

Why does the gap statistic matter more here than the silhouette score?

Because the silhouette is only defined for two or more clusters and never questions whether any clustering is appropriate. It can only tell you which k is least bad. The gap statistic compares the achieved inertia against what would be achieved on random data with the same bounding box, so it is able to conclude that one cluster is best, which is the finding this dataset actually supported.

The recommendation ended up using two recorded fields rather than a learned segmentation. What did that buy beyond honesty?

It buys availability at the moment of use. A cluster assignment has to be computed from the full record, most of which is written after the call. Category and repeat-caller status are known as the call begins, which is when a counsellor needs to choose an approach. A segmentation that only exists after the conversation cannot change the conversation.

MODULE 7 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 k-means is run on two long, parallel, elongated groups of points and returns clusters that cut across both of them. Why?

A. The chosen value of k was too small for the shape that the data actually has.

B. Points go to the nearest centre, so the regions are split by straight lines.

C. The features were standardised first, and standardising destroyed the elongation.

D. The k-means++ initialisation spreads the starting centres out, which splits long groups in half.

2 Silhouette scores hover near 0.14 for every k from 2 to 10. Which measure is able to tell you that no clustering is appropriate at all?

A. Davies-Bouldin, because a value above one indicates that no structure is present in the data.

B. Calinski-Harabasz, since it is the only one of these criteria that is corrected for chance.

C. The gap statistic, which compares your inertia against random data.

D. The adjusted Rand index computed between two random subsamples of the same dataset.

3 A team reduces 200 features to 30 components retaining 95 per cent of the variance, and the downstream supervised model gets worse. What is the most likely explanation?

A. PCA never sees the target, so the discarded variance held signal.

B. The components were not standardised again after the projection into the new space.

C. Thirty components is too few to represent a model that started with 200 original features.

D. PCA requires the original features to be uncorrelated, and here they clearly were not.

4 A t-SNE plot shows two tight clusters at opposite ends of the picture and one large diffuse blob. Which reading is supportable?

A. The two distant clusters are more different from each other than either is from the blob.

B. The blob holds more members than either cluster does, because it occupies more of the plot.

C. The structure must be real, because a projection of random noise would not produce clusters.

D. Points inside each cluster are neighbours in the original space.

5 Sentence embeddings are compared by cosine similarity rather than by Euclidean distance. What does that choice deliberately discard?

A. The direction of the vectors, which is what encodes the topic being written about.

B. The vectors' length, which often encodes frequency or document length.

C. The sign of each coordinate, which carries no meaning in a learned embedding space.

D. The dimensionality, since cosine similarity is defined in any number of dimensions.

6 A detector catches 90 per cent of the 100 daily frauds among a million transactions, with a 1 per cent false positive rate. What does the alert queue look like?

A. About a thousand alerts a day, of which roughly 9 per cent would be genuine cases.

B. About ninety alerts a day, of which about 90 per cent would be genuine cases.

C. About ten thousand alerts a day, roughly 1 per cent of them real.

D. About a hundred alerts a day, matching the number of frauds that occur each day.

MODULE 8

Neural networks, built from what you already know

A network is a stack of linear models with a bend between them, trained by the same gradient descent from module 2. This module builds it that way: why one layer is not enough, how the chain rule assigns credit backwards, what actually stopped networks working before 2012, and the two architectures — convolution and attention — that carry almost everything you have heard of.

BY THE END OF THIS MODULE YOU CAN

Explain why a multi-layer network with no non-linearity collapses to a single linear model, trace one backpropagation step through a two-layer network by hand, diagnose a training failure from the loss curve and the gradients, and fine-tune a pretrained model on a few hundred labelled examples using free compute

73 · 9 min

Neural networks as the natural next step

THE ONE THING TO KEEP

A neural network learns its own features; the loss, the splits, the metrics and the thresholds do not change.

Logistic regression, stacked

Logistic regression takes a weighted sum of the inputs and squashes it into a probability. Draw it and you get inputs on the left, one output on the right, a weight on every arrow. That is a neural network with one layer.

A network adds layers. The outputs of one layer become the inputs of the next:

```
h1 = relu(W1 @ x  + b1)   # 784 inputs -> 128 numbers
h2 = relu(W2 @ h1 + b2)   # 128 -> 64
y  = softmax(W3 @ h2 + b3)  # 64 -> 10 class probabilities
```

Three matrix multiplications, three additions, and a non-linear function between them. That is the whole computation.

Why the non-linearity is not optional

Remove the `relu` calls and the network is `W3 @ (W2 @ (W1 @ x))`. Matrix multiplication is associative, so that equals `(W3 @ W2 @ W1) @ x`, which is a single matrix. Three layers collapse to one. You can stack fifty of them and still have a linear model with a slower forward pass.

So something non-linear has to sit between the layers, and it can be very simple. `relu(z) = max(0, z)` — pass positives through, set negatives to zero. That one bend, repeated, is enough to let a deep stack express essentially any function.

The standard choices: **ReLU** in hidden layers, **sigmoid** on the output for binary classification, **softmax** on the output for multi-class.

What the layers are for

Lesson 3 said the features you hand over decide the ceiling. A network's hidden layers *are* learned features.

Train a network on satellite photos to find rooftop solar panels. Inspect what the layers respond to and you find, roughly: the first layer fires on edges and colour transitions; the middle layers on corners, repeating textures and rectangular grids; the last hidden layer on things that are recognisably panel arrays. Nobody designed those. They fell out of minimising the loss.

That is the real claim of deep learning, and it is narrower and more interesting than "bigger models are better". A network is not a superior curve-fitter. It is a system that does its own feature engineering, which is why it took over the domains where features were hardest to write by hand — vision, speech, language — and why it does not take over a spreadsheet with 18 columns you already understand.

Backpropagation is not a new idea

Gradient descent needs the gradient: how much the loss changes when each weight changes. In a network the weights sit several function-compositions away from the loss, so the derivatives compose too — that is the chain rule from calculus.

Backpropagation is the chain rule arranged efficiently. Run the input forward and keep the intermediate values. Then walk backward from the loss, and at each layer combine the gradient arriving from above with the values you saved, producing both the gradient for that layer's weights and the gradient to pass further back. One backward pass gives you the derivative with respect to every weight, at roughly the cost of one forward pass.

It is not a learning algorithm. It is how you get the number that Lesson 8's walk needs.

What carries over, and what gets harder

Everything in this course still applies, and most of it matters more.

- **Loss, gradient descent, learning rate.** Unchanged. The learning rate is still the hyperparameter that most often decides whether training works at all.
- **Train, validation, test.** Unchanged, and now enforced by people with millions of parameters and every incentive to peek.
- **Overfitting.** Now the default state. A network can memorise a training set outright, so regularization is not optional: dropout, weight decay, early stopping, data augmentation.
- **Metrics and thresholds.** Unchanged. A network's output is a score and the threshold is still yours.
- **Features.** Learned inside the model for perceptual data. Still yours for tabular data — which is why a gradient-boosted model will usually beat a network on 5,000 rows of a spreadsheet, and cost far less to train and run.

Where this goes

A large language model is the same three ingredients at a scale that changes the character of the thing. The shape is a transformer, whose layers pass information between positions in a sequence rather than only upward. The loss is: predict the next token. The optimiser is a refined gradient descent, running for weeks across thousands of accelerators.

The labels are free, because the next token is already in the text — the self-supervision from Lesson 2. That is what removed the bottleneck. Everything else you have learned here is still underneath it.

BEFORE YOU MOVE ON

An engineer stacks five linear layers with no activation function between them, trains carefully, and finds it performs no better than a single linear layer. Why?

A. Composing linear functions yields another linear function, so five layers can express exactly the same set of models as one.
B. The extra layers need far more training time before their behaviour separates from a single layer's.
C. The gradients vanished on the way back through five layers, so the early layers never learned.
D. Five layers is too shallow for depth to show any benefit.

74 · 8 min

XOR, and why the bend is the whole idea

THE ONE THING TO KEEP

A stack of linear layers collapses algebraically into one linear layer, so every capability a network has beyond linear regression comes from the non-linearity applied between the layers.

Four points that broke the field for a decade

Four inputs, one output:

```
(0,0) -> 0
(0,1) -> 1
(1,0) -> 1
(1,1) -> 0
```

This is XOR: output 1 when exactly one input is 1. Try to separate the 1s from the 0s with a straight line. You cannot. The two positive points sit on one diag-

onal and the two negative points on the other, and no line puts them on opposite sides.

A single-layer perceptron is a linear model with a step at the end, so it draws exactly one straight boundary. It cannot learn XOR. Minsky and Papert made this point in 1969, and interest in neural networks collapsed for most of the following decade.

The fix, and its catch

Add a hidden layer. Two units, each drawing its own line, and an output unit combining them. Unit A fires when at least one input is on; unit B fires when both are on; the output is A minus B, which is exactly XOR.

That works — but only because of something easy to miss. Suppose the hidden units are purely linear, with no bend applied to their output. Then:

```
layer 1: h = W1 x + b1
layer 2: y = W2 h + b2
       = W2 (W1 x + b1) + b2
       = (W2 W1) x + (W2 b1 + b2)
```

That is `W x + b` for some other matrix `W`. **A stack of linear layers is a single linear layer.** Fifty layers, ten thousand units each, and it is still one linear model with a lot of wasted arithmetic. Depth buys nothing without a non-linearity between the layers.

So the essential component is not the layers. It is the bend applied after each one — ReLU, or sigmoid, or tanh. Everything a network can do that a linear model cannot comes from those bends.

What a network is, structurally

```
input -> [linear] -> [bend] -> [linear] -> [bend] -> [linear] -
> output
```

Each linear step is a matrix multiplication plus a bias — the same operation as module 2's linear model, applied to many outputs at once. Each bend is a fixed function applied elementwise, with no parameters at all.

```
import numpy as np
def forward(x, W1, b1, W2, b2):
    h = np.maximum(0, x @ W1 + b1)      # linear, then ReLU
    return h @ W2 + b2                  # linear
```

That is a complete two-layer network in three lines. Everything else in this module — better initialisation, better optimisers, normalisation, convolution, attention — is refinement of how those matrices are shaped and found.

What depth buys

The universal approximation theorem says a network with one hidden layer can approximate any continuous function on a bounded region to any accuracy, given enough units.

Read the small print. It says such a network *exists*; it does not say gradient descent will find it, or that the number of units required is affordable. For many functions, a shallow network needs exponentially more units than a deep one to reach the same accuracy.

The intuitive reason is composition. Early layers learn simple pieces, later layers combine them into more complex ones, and a deep network can reuse the same piece in many combinations. A shallow one has to represent every combination separately. In vision this is visible directly: first-layer filters learn edges, later layers learn textures, then parts, then objects — and the edge detectors are shared by everything above.

Where the biological story helps and stops

The vocabulary — neurons, activation, firing — comes from a 1943 model of a biological neuron, and it is worth knowing that the analogy is loose and mostly historical.

Real neurons communicate in spikes over time, have complex dendritic processing, and there is no known biological mechanism that computes backpropagation's exact gradients. Nothing in this module depends on the biology being accurate. A network is a differentiable function with many parameters, fitted by gradient descent, and that description is complete.

The useful mental model is the one from module 2 extended: a linear model whose features are themselves learned by earlier linear models with bends in between. The loss, the splits, the metrics and the thresholds all work exactly as before.

BEFORE YOU MOVE ON

A team builds a 6-layer network for a tabular problem but omits the activation functions between layers. It performs identically to logistic regression. Why?

A. Composing linear maps yields another linear map, so the six layers reduce to a single equivalent linear layer regardless of their width.

B. Six layers is too few to show a benefit; with 20 layers the extra capacity would become visible.

C. Tabular data has no hierarchical structure, so depth cannot help on this kind of problem.

D. The layers were not initialised properly, so the later ones never learned anything beyond the first.

75 · 8 min

The bends: ReLU, and why sigmoid stopped being used inside networks

THE ONE THING TO KEEP

Sigmoid's derivative peaks at 0.25, so backpropagation shrinks the gradient by at least a factor of four per layer and deep networks stopped learning; ReLU's derivative is exactly 1 where it is active, which is why a crude hinge replaced a smooth curve.

Sigmoid, and the problem it caused

Early networks used the sigmoid, `1 / (1 + exp(-x))`, inside hidden layers. It squashes any input into (0, 1) and it is smooth, which felt right.

Its derivative is `s(x) * (1 - s(x))`, which peaks at 0.25 when `x` is 0 and approaches zero as `x` moves away in either direction. That maximum of 0.25 is the whole problem.

Backpropagation multiplies derivatives layer by layer. With sigmoids, each layer multiplies the gradient by at most 0.25:

```
1 layer:   0.25
5 layers:  0.25^5  = 0.00098
10 layers: 0.25^10 = 0.00000095
```

By ten layers the gradient reaching the first layer is a millionth of what arrived at the last. The early layers effectively stop learning. This is the **vanishing gradient problem**, and it is the main reason deep networks did not work before about 2010. It was not a shortage of ideas; it was that the signal did not reach the bottom.

tanh is the same shape centred on zero, with a maximum derivative of 1. Better, and still saturating, so it only postpones the problem.

ReLU

```
ReLU(x) = max(0, x)
```

A hinge. Negative input gives zero, positive input passes through unchanged.

Its derivative is exactly 1 for positive inputs. Multiply by 1 through fifty layers and nothing shrinks. That single property is most of why deep networks became trainable, and it is why a function this crude replaced a smooth one that had a century of mathematical respectability behind it.

Why a hinge replaced a smooth curve

Sigmoid in a hidden layer	ReLU, max(0, x)
• Derivative peaks at 0.25 and falls towards zero on either side	• Derivative is exactly 1 wherever the input is positive
• Ten layers: 0.25 to the tenth, about a millionth of the gradient reaches the bottom	• Fifty layers of multiplying by 1: nothing shrinks
• Early layers stop learning; deep networks did not work before about 2010	• Half the units output zero for any input, so the representation is sparse
• An exponential per unit	• A comparison; costs almost nothing
	• The price: a unit whose input is always negative is dead for good

Backpropagation multiplies one local derivative per layer. A factor of at most 0.25 per layer starves the early layers; a factor of exactly 1 leaves the gradient intact. The sigmoid did not disappear — it moved to the output layer, where one of it is applied and saturation is the point.

Two further benefits. It is cheap — a comparison, not an exponential. And it produces genuine zeros, so a typical layer has half its units inactive for any given input, which makes the representation sparse and, in practice, easier to optimise.

Dying ReLU

The cost. If a unit's input is negative for every training example, its gradient is zero for every example, so its weights never update. The unit is dead permanently and contributes nothing for the rest of training.

This happens most often after a large learning-rate step pushes the bias strongly negative. With a badly chosen learning rate, 40% of units can die in the first epoch and the network never recovers that capacity.

The variants that address it:

- **Leaky ReLU** — `max(0.01 * x, x)`. A small slope on the negative side means the gradient is never exactly zero, so a unit can come back.
- **ELU** and **SELU** — smooth on the negative side, with outputs that push activations towards zero mean.
- **GELU** — a smooth approximation to ReLU weighted by a Gaussian. Standard in transformers, including BERT and the GPT family.
- **Swish / SiLU** — `x * sigmoid(x)`. Found by automated search, marginally better than ReLU on deep vision networks.

The honest summary: on most problems the differences between these are under a point. ReLU is the right default. GELU if you are building a transformer, because that is what everything else in that family uses.

The output layer is a separate decision

Hidden layers and the output layer answer different questions, and the output is dictated by the task rather than chosen by preference.

- **Regression**: no activation. You want an unbounded real number.
- **Binary classification**: sigmoid, giving one probability. Paired with binary cross-entropy.
- **Multi-class, one label**: softmax across the classes, from module 2. Paired with categorical cross-entropy.
- **Multi-label**: independent sigmoids, one per label. Also from module 2, and the distinction matters as much here as it did there.
- **Bounded positive output** such as a count: softplus, or exponentiate, or predict the log and transform back.

So sigmoid did not disappear. It moved from the hidden layers, where its vanishing derivative was fatal, to the output layer, where exactly one of them is applied and the saturation is a feature rather than a defect.

The numerical detail worth knowing

Every serious framework computes the sigmoid or softmax *together with* the loss in one fused operation — `BCEWithLogitsLoss` in PyTorch, `from_logits=True` in Keras — rather than applying the activation and then the loss separately.

The reason is precision. A sigmoid output of 0.9999999 rounds to 1.0 in float32, and `log(1 - 1.0)` is negative infinity. Computing the loss directly from the pre-activation values avoids the rounding entirely. Use the fused version; the separate version will eventually produce a `NaN` you cannot explain.

BEFORE YOU MOVE ON

After a few epochs, 45% of a network's ReLU units output zero for every input in the validation set and their weights have stopped changing. What most likely happened?

A. The network is overfitting, and the dead units are a form of automatic regularisation.

B. ReLU saturates for large positive inputs, so those units have reached their maximum and stopped responding.

C. A large learning-rate step pushed those units' biases strongly negative, so their inputs are always negative, their gradient is exactly zero, and they can never update again.

D. The inputs were not standardised, so half the features are negative and ReLU zeroes them.

76 · 10 min

Backpropagation, traced through one tiny network

THE ONE THING TO KEEP

Backpropagation multiplies the gradient arriving from above by each operation's local derivative, which computes every weight's gradient in one backward pass — and explains why gradients vanish, why dead ReLUs stop learning, and why training needs far more memory than inference.

The question backpropagation answers

A network has thousands of weights. Change one, and the loss changes by some amount. Gradient descent needs that amount for every weight, and computing it by nudging each one and re-running the network would take one forward pass per weight — impossible at any real scale.

Backpropagation computes all of them in one backward pass, at roughly the cost of one forward pass. That efficiency is the reason deep learning exists at all.

The smallest possible network

One input, one hidden unit, one output. Three weights and no biases, to keep the arithmetic visible.

```
x = 2.0
w1 = 0.5      h_pre = w1 * x        = 1.0
              h     = ReLU(1.0)     = 1.0
w2 = -0.3     y_pre = w2 * h        = -0.3
              y     = y_pre         = -0.3
target = 1.0
loss  = (y - target)^2 = (-1.3)^2   = 1.69
```

Backwards, one step at a time

We want how much the loss changes per unit change in `w1` and in `w2`. Work backwards from the loss, carrying one number at each stage.

Step 1 — loss with respect to the output.

```
d(loss)/dy = 2 * (y - target) = 2 * (-1.3) = -2.6
```

So increasing `y` by a small amount reduces the loss at a rate of 2.6 per unit. Good; `y` is below the target.

Step 2 — loss with respect to w2. Since `y = w2 * h`, changing `w2` changes `y` at a rate of `h`:

```
d(loss)/dw2 = d(loss)/dy * dy/dw2 = -2.6 * 1.0 = -2.6
```

Step 3 — loss with respect to h. Changing `h` changes `y` at a rate of `w2`:

```
d(loss)/dh = -2.6 * (-0.3) = 0.78
```

Step 4 — through the ReLU. `h_pre` was 1.0, which is positive, so ReLU's derivative is 1 and the gradient passes through unchanged.

```
d(loss)/d(h_pre) = 0.78 * 1 = 0.78
```

Had `h_pre` been negative, this would be 0.78 * 0 = 0, and nothing below would have learned from this example. That is the dying-ReLU mechanism, visible in one line of arithmetic.

Step 5 — loss with respect to w1. Since `h_pre = w1 * x`:

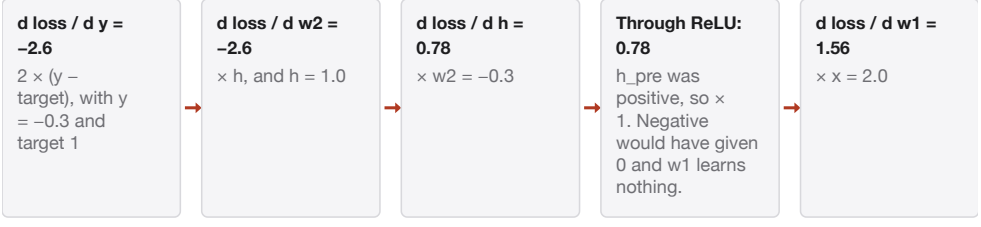
```
d(loss)/dw1 = 0.78 * 2.0 = 1.56
```

Update, learning rate 0.1:

```
w2 = −0.3 − 0.1 ∗ (−2.6) = −0.04
w1 =  0.5 − 0.1 ∗ ( 1.56) = 0.344
```

Rerun the forward pass with those values and the loss falls from 1.69 to about 1.10. One step, and it worked.

One backward pass through the three-weight network



Read right to left in time, left to right on the page: each box is the number arriving from the layer above multiplied by one local derivative. Ten layers would be ten such multiplications, which is exactly where gradients vanish or explode.

The pattern

Every step is the same shape: **the gradient arriving from above, multiplied by the local derivative of this operation.** That is the chain rule, applied mechanically, and it is why the whole thing costs one pass — each node needs only the number handed to it and its own local rule.

Two consequences follow directly.

Multiplication compounds. Ten layers means ten multiplications. If the local derivatives are consistently under 1, the gradient vanishes; if consistently over 1, it explodes into `NaN`. This is the same arithmetic that killed sigmoid networks, seen from the other direction.

You must store the forward values. Step 5 needed `x`, step 2 needed `h`. Every intermediate activation from the forward pass has to be kept until the backward pass uses it — which is why training uses far more memory than inference, and why the standard fix when a model will not fit is to reduce the batch size.

In practice

You will never write this. Every framework builds a graph of operations during the forward pass and walks it backwards automatically.

```
import torch
x = torch.tensor([2.0])
w1 = torch.tensor([0.5], requires_grad=True)
w2 = torch.tensor([-0.3], requires_grad=True)

loss = (torch.relu(w1 * x) * w2 - 1.0) ** 2
loss.backward()
w1.grad, w2.grad        # tensor([1.5600]), tensor([-2.6000])
```

The same numbers. `requires_grad=True` is what tells PyTorch to record the operations, and `.backward()` is the walk.

Why do it by hand once

Because the errors you will actually meet are all visible in the trace above. A gradient that is zero because a ReLU was inactive. A gradient that vanished through many small multiplications. A gradient that exploded through many large ones. Memory exhausted because activations are retained. A `.backward()` that fails because something in the chain was converted to NumPy and broke the graph.

Every one of those is obvious once you have seen the mechanism and opaque if you have not. Twenty minutes with a pen buys you that.

BEFORE YOU MOVE ON

In the traced network, suppose w1 had been -0.5 instead of 0.5, making h_pre negative. What would the gradient on w1 be, and why?

A. Larger in magnitude, because the error would be bigger with a negative hidden pre-activation.

B. The same, because ReLU's derivative does not depend on the sign of its input.

C. Negative rather than positive, because the sign of the pre-activation flips the sign of the gradient.

D. Exactly zero, because ReLU's derivative is 0 for negative inputs, so the gradient arriving from above is multiplied by 0 and nothing below the ReLU updates for this example.

77 · 9 min

Starting well, and staying well-behaved

THE ONE THING TO KEEP

Initialisation scale keeps activation variance stable across layers, normalisation keeps it stable across training, and a residual connection gives the gradient an unmultiplied path backwards — which is what made networks deeper than about twenty layers trainable at all.

Why not zero

Initialise every weight to zero and every unit in a layer computes the same output, receives the same gradient, and updates identically. The layer behaves as one unit forever, no matter how wide it is. This is the symmetry problem, and it is why initialisation must be random.

Why not any random values

Random is necessary and not sufficient. The scale matters enormously, and the reasoning is the same compounding arithmetic as backpropagation.

Consider a 20-layer network with weights drawn from a normal distribution.

- **Standard deviation too small**, say 0.01. Each layer's outputs are smaller than its inputs. After twenty layers the activations are effectively zero, the gradients are effectively zero, and nothing learns.
- **Too large**, say 1.0 with wide layers. Each layer amplifies. Activations grow layer by layer, saturating any bounded activation and overflowing to `NaN` with unbounded ones.

You want the variance of the activations to stay roughly constant as the signal passes through, in both directions.

The two standard schemes

Xavier / Glorot — variance `2 / (fan_in + fan_out)`. Derived assuming a symmetric activation like tanh. Use it with tanh or sigmoid.

He / Kaiming — variance `2 / fan_in`. The extra factor of 2 compensates for ReLU zeroing half the inputs, which otherwise halves the variance at every layer. Use it with ReLU and its relatives.

```
import torch.nn as nn
layer = nn.Linear(256, 128)
nn.init.kaiming_normal_(layer.weight, nonlinearity="relu")
nn.init.zeros_(layer.bias)
```

Frameworks default to something sensible, so you rarely set this by hand. It matters when you write a custom layer, and it matters that you know why a network that will not train at all might have this as its cause.

Biases start at zero, which is fine — there is no symmetry problem, because the weights already differ.

Batch normalisation

Initialisation fixes the start. Normalisation keeps things well-behaved throughout training.

Batch norm standardises each feature across the current mini-batch to mean 0 and variance 1, then applies two learned parameters — a scale and a shift — so the network can undo the normalisation if that is what it needs.

```
nn.Sequential(nn.Linear(256, 128), nn.BatchNorm1d(128),
nn.ReLU())
```

What it buys is substantial: much higher learning rates become safe, training is faster and more stable, and it acts as a mild regulariser because each example's normalisation depends on which other examples happened to be in its batch.

What it costs is worth knowing before you use it. Behaviour differs between training, where batch statistics are used, and inference, where a running average is used — so a model can train well and behave differently at prediction time. It performs badly with small batches, since the statistics are estimated from too few examples. And it creates a dependence between examples in a batch, which is awkward for some architectures and for reproducibility.

The original 2015 explanation was that it reduces "internal covariate shift". Later work argued the real effect is that it smooths the loss surface, making it easier to descend. The mechanism is still debated; the practical benefit is not.

Layer normalisation

Normalise across the *features* of each individual example rather than across the batch.

This removes every problem in the list above: no dependence between examples, identical behaviour in training and inference, works with a batch size of one. It is the standard in transformers and in any sequence model, where sequences have different lengths and batch statistics are ill-defined.

The rough division: batch norm for convolutional vision networks, layer norm for transformers and anything sequential.

Residual connections

The third piece, and arguably the most important of the three.

```
output = x + F(x)
```

Add the layer's input to its output. On the backward pass, the gradient flows through both branches, and the `x` path passes it through **unchanged** — no multiplication, so nothing shrinks. The vanishing gradient problem has a direct route around it.

This is what made networks of 50, 100 and 1,000 layers trainable. Before residual connections, adding layers past about 20 made networks *worse on the training set*, which is not overfitting but a failure to optimise. He and colleagues introduced them in 2015 and depth stopped being the constraint.

A useful second reading: with a residual connection, a layer that learns nothing outputs zero and the block becomes the identity. So adding layers cannot make the architecture less expressive, and the network can effectively choose its own depth.

The order to check

When a network will not train, the checks in order of frequency: learning rate too high, inputs not standardised, initialisation wrong for the activation, no normalisation in a deep stack, no residual connections past about 20 layers. Four of those five are one line each.

BEFORE YOU MOVE ON

A 40-layer plain network without residual connections has higher training error than a 20-layer version of the same architecture. What does this indicate?

A. An optimisation failure rather than a capacity problem: gradients degrade through 40 multiplications, so the deeper network cannot even fit what the shallower one fits — which is the problem residual connections solve.

B. The deeper network needs a larger batch size for its batch normalisation statistics to be reliable.

C. 40 layers exceeds the universal approximation requirement, so the extra layers add noise to the function being represented.

D. Overfitting, since the deeper model has more parameters and memorises the training data.

78 · 8 min

Momentum, Adam, and what the optimiser is doing for you

THE ONE THING TO KEEP

Momentum averages recent gradients so oscillations cancel and consistent directions accumulate; Adam adds a per-parameter step size from the average squared gradient, and its weight decay must be applied outside that scaling, which is what AdamW does.

What plain SGD struggles with

Gradient descent follows the slope where it currently stands. Two situations make that inefficient.

Ravines. A surface that is steep across and shallow along — the long narrow valley from module 2. The gradient points mostly across the valley, so the optimiser zigzags from wall to wall and makes slow progress along the floor.

Uneven scales. One parameter needs large steps and another needs small ones, and a single learning rate has to serve both.

Momentum

Keep a running average of recent gradients and step in that direction.

```
v = beta * v + gradient          # beta typically 0.9
w = w - learning_rate * v
```

The zigzag components across the ravine alternate in sign and cancel in the average. The consistent component along the floor accumulates. The result is faster progress in the useful direction and damped oscillation, and it also carries the optimiser through small bumps and flat regions where the raw gradient is near zero.

With `beta = 0.9`, the effective step is roughly ten times the current gradient once the average has built up, which is why momentum with a given learning rate is considerably more aggressive than plain SGD with the same rate.

Nesterov momentum evaluates the gradient at the position the momentum is about to take you rather than where you are — a lookahead that damps overshoot slightly. A small, real improvement.

Adam

Adam adds a per-parameter learning rate on top of momentum. It keeps two running averages: one of the gradient, and one of the gradient *squared*. Then it divides the step by the square root of the second.

```
m = b1 * m + (1 - b1) * g          # average gradient
v = b2 * v + (1 - b2) * g * g      # average squared gradient
w = w - lr * m / (sqrt(v) + eps)
```

The effect: a parameter that consistently receives large gradients has a large `v`, so its steps are scaled down; one receiving small gradients gets steps scaled up. Every parameter ends up with a step size suited to its own history,

which removes most of the sensitivity to feature scaling and to how the layers happen to be sized.

Defaults `b1 = 0.9`, `b2 = 0.999`, `lr = 0.001` work across an enormous range of problems, which is why Adam is what almost everyone uses by default.

Adam's costs

Memory. Two extra values per parameter. For a model with a billion parameters, the optimiser state is twice the size of the model, and it is a large part of why training needs so much more memory than inference.

Generalisation. On image classification specifically, well-tuned SGD with momentum often generalises slightly better than Adam. The gap is small and the tuning effort is not, and Adam remains the right default when you do not have time to tune.

Weight decay is broken in plain Adam. L2 regularisation added to the loss gets divided by the same `sqrt(v)` term, so parameters with large gradients receive weaker decay — which is the opposite of intended. **AdamW** applies the decay directly to the weights, outside the adaptive scaling, and it is what you should use.

```
torch.optim.AdamW(model.parameters(), lr=3e-4,
weight_decay=0.01)
```

AdamW is the standard for training transformers and should be your default over Adam. The fix is one letter and it matters.

Schedules

The learning rate should not stay constant, for the reasons in module 2.

Warmup — increase from near zero over the first few hundred to few thousand steps. Essential for transformers, where early gradients on freshly initialised weights are unreliable and a large early step does lasting damage.

Cosine decay — fall smoothly to near zero over training. The current standard.

One-cycle — warm up to a high rate, then decay below the starting point. Trains surprisingly fast and is built into fastai and PyTorch.

```
sched = torch.optim.lr_scheduler.OneCycleLR(opt, max_lr=1e-3,

total_steps=n_steps)
```

Gradient clipping

One more safety measure. If the gradient's norm exceeds a threshold, scale it down.

```
torch.nn.utils.clip_grad_norm_(model.parameters(),
max_norm=1.0)
```

This prevents a single unusual batch from taking a catastrophic step and destroying weights that took hours to train. It is standard for recurrent networks and transformers and costs essentially nothing. If your loss occasionally spikes to `NaN` after training normally for a while, this is usually the fix.

What to actually use

AdamW at 3e-4, cosine decay with warmup, gradient clipping at 1.0. That combination trains almost anything without drama. Reach for SGD with momentum only if you are chasing the last fraction of a point on a vision benchmark and have the compute to tune it.

BEFORE YOU MOVE ON

A team uses Adam with weight_decay=0.01 and finds regularisation appears weaker than expected on the parameters that change most. Why does switching to AdamW help?

A. Adam's momentum term cancels the decay over successive steps, while AdamW omits momentum entirely.
B. AdamW uses a larger effective decay coefficient by default, so the same value regularises more strongly.
C. AdamW disables the adaptive per-parameter scaling, so every weight receives the same decay.
D. In Adam the L2 term enters the loss and is then divided by the square root of the average squared gradient, so parameters with large gradients get weaker decay; AdamW applies the decay directly to the weights, outside that scaling.

79 · 8 min

Keeping a large network from memorising

THE ONE THING TO KEEP

Early stopping and more data outrank every other regulariser, dropout must be disabled at inference or predictions become random, and before regularising anything you should confirm the network can deliberately memorise twenty examples — otherwise you are tuning around a bug.

Five tools, in order of how much they matter

1. Early stopping. Watch validation loss and keep the weights from its best epoch. Free, effective, and it is the one to implement first. Everything else is refinement on top of it.

2. More data, or augmentation. The most effective regulariser there is. For images, augmentation is nearly free performance:

```
from torchvision import transforms
transforms.Compose([
    transforms.RandomResizedCrop(224, scale=(0.7, 1.0)),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(0.2, 0.2, 0.2),
])
```

The transformations must preserve the label. Horizontal flip is fine for cats and wrong for handwritten digits, where a flipped 2 is not a 2, and wrong for medical images where laterality is diagnostic. Think before adding one.

For text, augmentation is harder because small edits change meaning; back-translation and synonym replacement are the usual attempts and their gains are modest. For tabular data there is no reliable equivalent, as module 3 said of SMOTE.

3. Weight decay. L2 from module 2, applied through AdamW. 0.01 to 0.1 is the usual range for transformers, 1e-4 for vision networks. Consistently helpful, rarely dramatic.

4. Dropout. During training, randomly zero a fraction of activations in a layer.

```
nn.Sequential(nn.Linear(512, 256), nn.ReLU(), nn.Dropout(0.3))
```

The mechanism: no unit can rely on any particular other unit being present, so the network cannot build fragile chains of co-adapted features. It is also usefully read as training an ensemble of many thinned networks that share weights, and averaging them at inference.

At inference dropout must be switched off and the activations scaled to compensate. In PyTorch that is `model.train(False)` — often written as the evaluation-mode call — and **forgetting it is one of the most common bugs in PyTorch code**: the model returns different predictions on each call and nobody can work out why.

A note on current practice: dropout was essential in the era of large fully connected layers. In convolutional networks batch norm largely replaced it, and

in transformers rates are typically low — 0.1 rather than the 0.5 of older tutorials. It remains useful and it is no longer the first thing to reach for.

5. Label smoothing. Instead of training towards a target of exactly 1.0 for the correct class, train towards 0.9, spreading the remaining 0.1 across the others.

This stops the network from driving its logits towards infinity to squeeze the last fraction out of the loss, which is where over-confidence comes from. It improves calibration meaningfully and often improves accuracy slightly.

```
nn.CrossEntropyLoss(label_smoothing=0.1)
```

Diagnosing which you need

The same reading as module 4, applied to a training curve.

- Training loss falls, validation loss rises: overfitting. Add regularisation, or stop earlier, or get more data.
- Both plateau high: underfitting. **Reduce** regularisation, or increase capacity, or train longer.
- Training loss is unstable or spikes: learning rate or gradient clipping, not regularisation.

The second case is worth flagging because the instinct is always to add regularisation, and adding it to a network that is underfitting makes things worse while looking like diligence.

The unfashionable first move

Before any of the above: **overfit a tiny subset deliberately.**

Take 20 examples and train until the model gets all 20 exactly right. If it cannot, you have a bug — a broken loss, a label misalignment, a learning rate so small nothing moves, a layer not connected. No amount of regularisation fixes a bug, and this test takes two minutes and finds most of them.

Only once the model can memorise 20 examples does it make sense to give it the full dataset and start controlling how much it memorises. Karpathy's recipe puts this step first for a reason: it separates "the model cannot learn" from "the model learns the wrong thing", and those need completely different responses.

Regularisation is not free, and the price is legible

One closing point, because it is easy to treat these five tools as costless improvements. Every one of them deliberately makes the model worse on the training data in exchange for a smaller gap. That is the trade from module 4 stated in a new place, and it means there is always a level at which more regularisation stops helping and starts hurting.

The way to find it is not intuition. Fix everything else, sweep one setting — dropout at 0.0, 0.1, 0.3, 0.5 — and plot validation loss against it. The curve is almost always U-shaped, the bottom is usually lower than the default you started with, and the whole sweep on a small network takes minutes.

Do them one at a time. Dropout, weight decay and augmentation all reduce effective capacity, so turning up all three together often overshoots into underfitting while each individual setting still looks reasonable. When a heavily regularised network's training and validation losses have converged at a poor level, the correct move is to remove regularisation, and almost nobody's first instinct is to try that.

BEFORE YOU MOVE ON

A network's training loss falls to 0.02 while validation loss plateaus at 0.31 and slowly rises. A colleague suggests increasing model width. What is wrong with that suggestion?

A. Wider layers require a lower learning rate, so training would become unstable.

B. The curves show overfitting — the model already fits the training data almost perfectly — so adding capacity increases the gap rather than closing it.

C. Width should only be increased alongside depth, or the network becomes unbalanced.

D. Wider layers need batch normalisation to train, which the network may not have.

80 · 9 min

Convolution: the same detector, everywhere in the image

THE ONE THING TO KEEP

A convolution applies the same small filter at every position, so it learns a pattern once instead of separately per location — which cuts parameters by orders of magnitude and builds a hierarchy from edges to objects as depth increases.

Why a fully connected layer fails on images

A 224 by 224 colour image is 150,528 numbers. A fully connected layer with 1,000 units needs 150 million weights for the first layer alone. That is unaffordable, and worse, it is wasteful in a specific way: a cat detector learned for the top-left corner would have to be learned again, from separate examples, for every other position.

The fully connected layer knows nothing about the fact that neighbouring pixels are related and that a pattern means the same thing wherever it appears.

What a convolution does instead

Slide a small window — typically 3 by 3 — across the image. At each position, multiply the window's values by a small set of learned weights and sum. That set of weights is a **filter** or kernel.

```
nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3,
padding=1)
```

This layer has 64 filters, each 3 by 3 across 3 input channels: `64 * 3 * 3 * 3 + 64 = 1,792` parameters. Compare that with 150 million.

Two ideas make the saving possible.

Local connectivity. Each output depends on a 3 by 3 patch, not the whole image. Pixels far apart are combined only in later layers, and only after the nearby structure has been summarised.

Weight sharing. The *same* filter is applied at every position. Learn an edge detector once and it works everywhere. This is what gives convolutional networks their translation equivariance: move the object in the image and the response moves with it.

What the filters learn

The learned filters are inspectable, and what they contain is consistent across networks and datasets.

- **Layer 1**: edges at various orientations, colour blobs. These look strikingly like Gabor filters, which is what early visual cortex is understood to compute — a convergence nobody designed.
- **Layer 2 to 3**: corners, curves, simple textures.
- **Middle layers**: repeating patterns, object parts — a wheel, an eye, a doorway.

- **Late layers**: whole objects and scene types.

Each layer combines the previous layer's pieces. This is the compositional hierarchy from the depth lesson, made visible, and it is the reason transfer learning works so well in vision: the early layers are learning something general about images rather than something specific to the training task.

The receptive field

One 3 by 3 filter sees 3 pixels. Stack two and the second sees a 5 by 5 region of the original. Stack three and it is 7 by 7. Add pooling or a stride and the region grows much faster.

This is why depth matters in vision. A network needs a receptive field covering enough of the image to recognise a whole object, and stacking small filters is a cheaper way to get there than using one large one — two 3 by 3 layers have 18 weights per channel pair and cover the same region as a single 5 by 5 with 25, while including an extra non-linearity.

Pooling, and its decline

Max pooling takes the maximum over each 2 by 2 block, halving the spatial dimensions. It reduces computation and gives a little tolerance to small shifts.

Modern architectures increasingly use strided convolutions instead — a convolution that moves 2 pixels at a time — which achieves the same downsampling with learned weights rather than a fixed rule. Pooling is not gone, and it is no longer automatic.

Where the field went

Two developments worth knowing so the picture is current.

Depthwise separable convolutions split a convolution into a per-channel spatial step and a 1 by 1 mixing step, cutting the parameter count by roughly a factor of 8 to 9 with little accuracy loss. This is what makes MobileNet and its relatives run on a phone, and it is the reason on-device vision is practical at all.

Vision transformers cut an image into patches and treat them as a sequence, with no convolution at all. They beat convolutional networks at very large data scales and lose to them at small ones, because the convolution's assumptions — locality and translation equivariance — are genuine prior knowledge that a transformer must learn from data instead. Hybrid designs now dominate.

That trade is a good general lesson. An architecture with built-in assumptions needs less data when the assumptions hold, and hits a ceiling the more flexible one does not.

What you will actually do

Almost nobody trains a vision network from scratch. You take a pretrained one and adapt it, which is the last lesson in this module. Free pretrained weights for hundreds of architectures are in `timm` and `torchvision`, and a fine-tuned ResNet on a few hundred images will beat anything you build from nothing on a free GPU.

BEFORE YOU MOVE ON

Why do modern architectures prefer stacking two 3x3 convolutions over a single 5x5 convolution, given both cover the same receptive field?

A. The 3x3 filters can be applied in parallel, whereas a 5x5 must be computed sequentially.

B. Two 3x3 layers use fewer weights per channel pair and insert an extra non-linearity between them, giving more expressive power for less parameter cost.

C. A 5x5 filter cannot be initialised stably, since its fan-in is too large for standard schemes.

D. Smaller filters produce a larger receptive field, which is what deep vision networks need.

81 · 9 min

Sequences: from recurrence to attention

THE ONE THING TO KEEP

Attention lets every position read every other in one step by computing a similarity-weighted average, which removes the multiplication chain that limited recurrent networks and parallelises training — at a cost that grows with the square of the sequence length.

The problem with a sequence

A sentence, a time series, a sequence of user actions. Two properties break the tools so far: the length varies, and order matters.

Recurrent networks, and their ceiling

Process one element at a time, carrying a hidden state forward:

```
h_t = f(W_h ∗ h_{t−1} + W_x ∗ x_t)
```

The state is a running summary of everything seen so far. Elegant, handles any length, and it dominated sequence modelling for two decades.

It has two problems that turned out to be fatal.

Long-range dependencies. "The cat, which had spent the entire afternoon asleep on the warm bricks by the kitchen door, **was** hungry." The verb agrees with "cat", 18 words back. To carry that information the network multiplies its hidden state by `W_h` eighteen times, and repeated multiplication shrinks or explodes exactly as in backpropagation. **LSTMs** and **GRUs** address this with gates that let information pass through unchanged when the gate is open — the same idea as a residual connection — and they extend the usable range considerably without removing the limit.

No parallelism. `h_t` needs `h_{t-1}`, so a sequence of 500 elements requires 500 sequential steps. This cannot be parallelised, and on hardware built for parallel arithmetic it is the difference between training in days and training in weeks.

Attention

Instead of forcing everything through one running state, let every position look directly at every other position and decide what is relevant.

For each position, produce three vectors from its representation: a **query** (what am I looking for), a **key** (what do I offer), and a **value** (what I will contribute). Compare each query against every key by dot product, softmax the results into weights, and take the weighted sum of the values.

```
attention(Q, K, V) = softmax(Q K' / sqrt(d)) V
```

Read it as a soft dictionary lookup. Rather than retrieving one entry by exact match, you retrieve a weighted blend of all of them, weighted by similarity.

The `sqrt(d)` matters and is easy to skip past. Dot products of `d`-dimensional vectors grow with `d`, and large values push softmax into a region where it is nearly one-hot and its gradients nearly vanish. Dividing by the square root of the dimension keeps the scale stable.

What this changed

Any position reaches any other in one step. No multiplication chain, so the 18-word dependency is as easy as an adjacent one. Long-range structure stops being the hard case.

Everything is parallel. Every position's attention is computed simultaneously, as a couple of large matrix multiplications, which is exactly what GPUs are built for. This is the practical reason transformers won: the same idea, trainable on far more data in the same wall-clock time.

Order must be reinserted. Attention treats the input as a set — permute the inputs and the outputs permute with them, unchanged. So position information is added explicitly, as a positional encoding, which is a slightly awkward patch on an otherwise clean idea and is still an area of active design.

The cost

Every position attends to every other, so computation grows with the **square** of the sequence length. Double the context and you quadruple the cost. This is the central engineering constraint of the whole architecture, and the reason context windows were 512 tokens in 2018 and expanded only as sparse patterns, linear approximations and memory-efficient exact implementations such as FlashAttention arrived.

It is worth being clear that this is a genuine limit rather than a temporary one. The workarounds are approximations or better memory management, not a change to the quadratic relationship itself.

Multi-head attention

Run several attention operations in parallel with different learned projections, then concatenate. One head may track syntactic agreement, another coreference, another position. Analyses of what individual heads do have found interpretable roles in some cases and nothing clean in many others, so treat "head 4 does syntax" as an occasional finding rather than a design principle.

Where this leaves you

A transformer block is: multi-head attention, a residual connection, layer normalisation, a small feed-forward network, another residual, another normalisation. Every component in that list appeared earlier in this module. Stack the block 12 or 96 times and you have the architecture behind essentially every modern language model.

That is the point of ending here. The parts are not exotic. They are the linear layer from module 2, the non-linearity from earlier in this module, the residual connection and normalisation from the initialisation lesson, and one new

operation — a weighted average whose weights are computed from similarities. Understanding that composition is a reasonable place for a foundations course to stop.

BEFORE YOU MOVE ON

A team replaces an LSTM with a transformer on 800-element sequences and finds memory use rises sharply. What property explains it?

A. Transformers have more parameters per layer than LSTMs, so the model itself is larger.

B. Layer normalisation stores statistics per position, which grows with sequence length.

C. Attention computes a score for every pair of positions, so an 800-element sequence produces a 640,000-entry matrix per head per layer, and memory grows with the square of the length.

D. Transformers cannot process sequences incrementally, so the whole dataset must be held in memory at once.

---

82 · 9 min

Standing on somebody else's compute

THE ONE THING TO KEEP

Pretrained early layers encode general structure that transfers across tasks, so start by freezing the backbone and fitting a linear model on its features — and escalate to full fine-tuning only when the measurement says your data supports it.

Why you should not train from scratch

Training a vision model from random weights needs hundreds of thousands of labelled images. Training a language model from scratch needs hundreds of

billions of tokens and a compute budget in the millions.

You do not have either. You do not need either, because someone already did it and released the weights.

Transfer learning takes a model trained on a large general dataset and adapts it to your specific task with a few hundred or few thousand examples. It is the single most practically important technique in this module, and it is what makes deep learning available to someone with a laptop.

Why it works

The convolution lesson showed what the early layers learn: edges, textures, shapes. None of that is specific to the thousand ImageNet categories. Edges are edges whether the image is a dog, a chest X-ray, or a defective weld.

So the early layers are reusable and only the final task-specific layers need replacing. The same holds in language: a model that learned to predict the next word has learned grammar, common sense and world knowledge that no task-specific dataset could teach it.

Three levels of adaptation

1. Feature extraction. Freeze the whole pretrained model, use it to produce vectors, train a small classifier on those.

```
import timm, torch
backbone = timm.create_model("resnet50", pretrained=True, num_-
classes=0)
backbone.train(False)                 # evaluation mode: no
dropout, fixed norm stats
with torch.no_grad():
    feats = backbone(images)          # (n, 2048)
# then: LogisticRegression().fit(feats, y)
```

Fastest, needs the least data, runs comfortably on a CPU, and cannot overfit the backbone because the backbone never changes. On a few hundred images

this is often within a couple of points of full fine-tuning, and it is what to try first.

2. Fine-tune the last layers. Freeze the early layers, train the last block plus a new head with a low learning rate. A middle option when your data resembles the pretraining data but not exactly.

3. Fine-tune everything. Train all weights with a small learning rate, typically 10 to 100 times smaller than you would use from scratch. Best results with enough data; risks destroying the pretrained features if the learning rate is too high, which is called catastrophic forgetting and looks like the model getting suddenly and permanently worse in the first epoch.

Discriminative learning rates — a lower rate for early layers and a higher one for later ones — split the difference, on the reasoning that early features need less change than late ones. It is standard practice in fastai and worth knowing.

Parameter-efficient fine-tuning

For large language models, fine-tuning every weight requires holding the model, its gradients and the optimiser state in memory at once — far more than a free GPU has.

LoRA freezes the original weights and trains small low-rank matrices added alongside them. It typically trains under 1% of the parameters, reaches close to full fine-tuning quality, and the resulting adapter is a few megabytes rather than gigabytes. **QLoRA** adds 4-bit quantisation of the frozen base model, which brings fine-tuning a 7-billion-parameter model within reach of a single free-tier GPU.

The free library is `peft` from Hugging Face, and this is genuinely the route by which a person with no budget can adapt a large model.

Where to get models

Hugging Face Hub — hundreds of thousands of models, free. **timm** — vision models. **torchvision** and **Keras Applications** — the classics, bundled.

Check the licence before you deploy. It varies from fully permissive to research-only to "open weights with restrictions on use above a certain scale", and the terms are not always what the word "open" suggests. This matters commercially and it is a genuine area of dispute — several licences described as open are not open by the standard definitions, and the disagreement is unresolved.

When transfer learning does not help

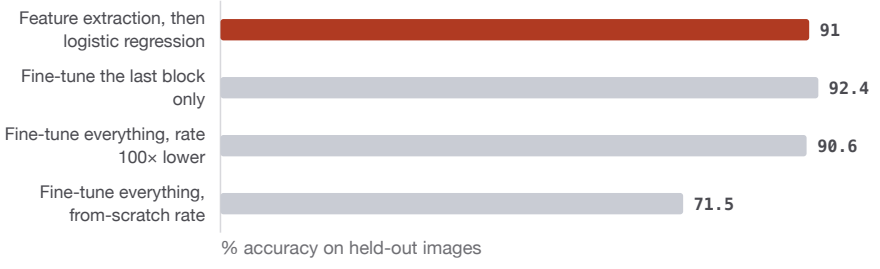
- **Your domain is very unlike the pretraining data.** ImageNet features transfer poorly to spectrograms, satellite imagery or scientific plots. It usually still beats random initialisation, and by less than you expect.
- **Tabular data.** There is no equivalent of ImageNet for tables, because there is no shared structure across datasets to transfer. Foundation models for tabular data are an active research area — TabPFN is the most notable — and gradient boosting remains the practical answer.
- **You have millions of labels already.** Then training from scratch is viable and may fit your domain better.

The honest recipe for someone with a laptop

1. Find a pretrained model close to your domain on Hugging Face.
2. Extract features and fit logistic regression. Record the number.
3. Fine-tune the last block. Record the number.
4. Fine-tune everything with a low rate, if step 3 helped and you have the data.
5. Stop at whichever step stopped improving.

Most people skip to step 4 and are surprised when it does worse than step 2 on 300 images. The steps are in this order because each one needs more data than the last, and the measurement tells you which one your data supports.

Four ways to adapt a pretrained ResNet-50 to 300 labelled images



On this little data the frozen backbone with a logistic regression on its features is within two points of the best, and full fine-tuning at a from-scratch learning rate destroys the pretrained features in the first epoch. The steps are ordered by how much data each needs, and the measurement says which one yours supports.

BEFORE YOU MOVE ON

A team fine-tunes all layers of a pretrained ResNet on 300 medical images at the same learning rate used for training from scratch. Accuracy is worse than a logistic regression on frozen features. Why?

A. Fine-tuning requires batch normalisation to be frozen, and leaving it active corrupted the running statistics.

B. ResNet features do not transfer to medical images, so the pretrained weights were useless from the start.

C. 300 images is below the minimum for any deep learning approach, so both results are unreliable.

D. A learning rate suited to random initialisation is far too large for pretrained weights, so the first epochs overwrite the general features the model had learned before the small dataset can teach it anything better.

CASE STUDY · MODULE 8

Three hundred and forty leaves

A district agriculture office in Belagavi, one extension officer with a laptop and a free Colab account, and 340 labelled photographs of diseased chilli leaves sent in by farmers over one season.

Farmers in the district send photographs to a WhatsApp number when a crop looks wrong. An officer looks at each one and replies. The volume had grown to about ninety photographs a day in the season, against one officer, and the reply time had stretched to two days, by which point spraying advice is worth much less.

The proposal was a classifier that would sort incoming photographs into five classes — leaf curl virus, anthracnose, thrips damage, nutrient deficiency, and healthy or unclear — so that the officer worked the queue in a sensible order rather than in arrival order.

The labelled set was 340 photographs, labelled by the officer himself over one season. Not thousands. Three hundred and forty, unevenly split, with the smallest class holding 31.

Prakash had worked through the transfer learning lesson and knew the recipe: feature extraction first, then last-block fine-tuning, then full fine-tuning, stopping at whichever step stopped improving. He also had a free GPU quota and an understandable wish to use it.

Step one took forty minutes. A pretrained ResNet-50 with the classifier head removed, run in evaluation mode over the 340 images to produce 2,048-dimensional feature vectors, then a logistic regression on top. Grouped five-fold cross-validation, grouped by the farmer who sent the photograph, because eleven farmers had sent more than five images each and several had photographed the same plant repeatedly. Macro F1 of 0.71.

Step two, fine-tuning the last block, took two hours including the mistakes. Macro F1 of 0.74, with a spread across folds wide enough that he could not be confident the improvement was real.

Step three, full fine-tuning, produced 0.62 and a training curve that fell off a cliff in the first epoch. That is catastrophic forgetting with the learning rate set too high, and dropping it by a factor of thirty recovered to 0.73 — no better than step two, at eight times the training time and with a model he could no longer explain the behaviour of.

So the measurement said step one or step two, and the difference between them was inside the noise. That was not the hard decision.

The hard decision was what the grouped cross-validation had exposed on the way. Every photograph in the training set came from a smartphone camera, in daylight, held about thirty centimetres from the leaf, by somebody who had been told by the officer how to take the photograph. The officer had, in effect, curated his own training distribution over a season of replying to people.

The incoming queue is not that. It contains photographs taken at dusk, at two metres, of a whole plant, through a polythene tunnel, and by farmers who have never been told anything about how to hold a phone.

Deploy the 0.74 model on the whole queue. It is a triage aid, not a diagnosis, and a wrong ordering is not a wrong prescription. The cost is that its confidence on out-of-distribution photographs is not lower than on good ones — a network is typically over-confident, and increasingly so as it gets larger — so the queue would be ordered wrongly with no signal that anything was amiss.

Deploy it only on photographs that resemble the training set, using reconstruction error or a simple image-quality check as a gate, and route everything else to the officer in arrival order. Less coverage, and it requires building the gate.

Spend the season collecting a representative labelled set first. Correct, and it means the reply time stays at two days for another year.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

He shipped step one — the frozen backbone plus logistic regression — with a gate in front of it, and he shipped it with an abstention band. Photographs whose predicted probability did not exceed 0.55 for any class were not ordered at all; they joined the arrival-order queue.

The gate was crude and worked: image size, blur estimate, and the fraction of the frame occupied by green pixels. Photographs failing it went straight to arrival order. About a third of the incoming queue failed it in the first month, which is roughly what he had expected from looking at fifty images by hand.

The logistic regression was chosen over the fine-tuned model for a reason that had nothing to do with the scores. It runs on a CPU on the office laptop, it retrains in ninety seconds when the officer adds new labels, and he does add them — he labels the abstentions as he works them, which is uncertainty sampling arrived at by accident, and the labelled set passed 900 by the end of the following season.

Reply time for the photographs the system ordered fell to under four hours. For the third it declined to order, it did not change.

Full fine-tuning did worse than a frozen backbone. What does that tell you about the dataset rather than about the method?

That 340 images across five classes do not support updating twenty-five million parameters. Each level of adaptation needs more data than the last, and full fine-tuning has enough freedom to reshape the pretrained features around a small sample. The first-epoch collapse was the learning rate destroying representations that took a great deal of compute to build, and even with the rate corrected there was not enough data to earn back the freedom.

Why group the cross-validation by farmer?

Because several farmers sent multiple photographs, sometimes of the same plant, and a random image split would put near-duplicates on both sides. The model would be recognising the plant rather than the disease and the score would be inflated by an amount nobody could estimate. The product runs on photographs from farmers whose other photographs are not in the training set, so the split has to match that.

The abstention band lowered coverage. Why was that a better design than ordering the whole queue?

Because the model's confidence carries no information about whether a photograph resembles the data it was trained on, and networks are typically over-confident. Ordering a queue wrongly with no signal is worse than not ordering part of it, since the officer would learn to distrust the whole ranking. Abstaining is the reject option from the cost-sensitive lesson: a system that is right on the cases it chooses to decide and defers the rest is a usable product.

MODULE 8 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 Fifty linear layers are stacked with no activation function between them. What function does the network compute?

A. A piecewise-linear function with as many segments as there are layers in the stack.

B. A polynomial of degree fifty in the inputs, one degree contributed by each layer.

C. A linear function whose gradients vanish somewhere after about twenty layers.

D. A single linear function of the input.

2 Why did the sigmoid stop being used inside hidden layers?

A. It is expensive to compute compared with a single comparison against zero at each unit.

B. Its derivative peaks at 0.25, so gradients shrink fourfold per layer.

C. It produces outputs that are never negative, which destabilises the following layer.

D. It cannot represent a non-linear decision boundary when used on its own in one layer.

3 During a backward pass the gradient arriving at a hidden unit is 0.78 and that unit's pre-activation was negative. With a plain ReLU, what gradient continues below it?

A. 0.78, because ReLU passes the incoming gradient through to the layer beneath unchanged.

B. Minus 0.78, because the sign of the gradient flips on the negative side of the hinge.

C. Zero, so nothing below learns from this example.

D. 0.0078, because a leaky negative slope of 0.01 is applied to the incoming gradient.

4 Why does AdamW apply weight decay outside Adam's adaptive scaling?

A. Because Adam's scaling would otherwise make the decay far too strong on every parameter.

B. Inside the scaling, parameters with large gradients receive weaker decay.

C. Because L2 regularisation is not differentiable and therefore cannot be added to the loss.

D. Because the decay has to be applied before the momentum term is updated each step.

5 A PyTorch model returns a different prediction each time it is called on the same row. What is the most likely cause?

A. The model was left in training mode, so dropout is still active.

B. The random seed was not set before the model file was loaded from disk.

C. The saved model file was written by a different version of the library.

D. The input tensor is not being normalised consistently between one call and the next.

6 With 300 labelled images, full fine-tuning of a ResNet scores worse than a frozen backbone with logistic regression on top. What does that indicate?

A. The pretrained weights are unsuitable for this domain and ought to be discarded entirely.

B. The logistic regression is overfitting, which is what flatters its cross-validated score.

C. The data does not support updating that many parameters.

D. The learning rate happened to be better tuned for the frozen version than for the other.

MODULE 9

Into the world, and keeping it honest

The half of the job that starts when the notebook closes. Serving a model without the training and serving paths drifting apart, monitoring the things that actually fail, deciding when to retrain, making a model small enough to be affordable, and the two obligations that outlast the model: not letting it corrupt its own training data, and documenting who it was built for.

> BY THE END OF THIS MODULE YOU CAN
>
> Take a fitted model to a running service with a versioned artefact, a reproducible training run, monitoring that distinguishes covariate shift from concept drift, a stated retraining trigger, and a written record of what the model was trained on and who it should not be used for

83 · 8 min

The model is about 5% of what you have to build

THE ONE THING TO KEEP

The model is a small fraction of a production system; the rest is feature computation, versioning, serving, monitoring and rollback — and knowing the serving constraint before choosing a model is what stops you building something that cannot be deployed.

What surrounds the model

A well-known 2015 paper from a Google team — "Hidden Technical Debt in Machine Learning Systems" — included a diagram of a production machine learning system with the model as one small box surrounded by much larger ones: data collection, feature extraction, verification, configuration, serving infrastructure, monitoring, resource management, process tools.

The point is not that the model is unimportant. It is that the model is the part you have been taught and the small part of what has to exist.

Here is the same claim as a list of what a working system needs beyond `model.fit`:

- Somewhere the features come from at prediction time, computed identically to training.
- A way to serve a prediction within a latency budget somebody agreed to.
- Version control over the model, the data, and the code that joined them.
- Monitoring that notices when inputs change shape or predictions change distribution.
- A path to roll back to yesterday's model in minutes.
- A retraining process that somebody can run without you.
- A record of what the model is for and what it is not for.

Each of those is a week or two of work. The model was an afternoon.

Four kinds of debt specific to this work

Entanglement. In ordinary software you can change one module and reason about the blast radius. In a model, changing anything changes everything: add a feature and every existing coefficient moves. The paper's name for this is CACE — changing anything changes everything — and it means you cannot make a local change to a model. You can only retrain and re-measure.

Undeclared consumers. Your model writes a score to a table. Six months later, three other teams read that table, and one of them uses it in a way you have never heard of. You change your model and their systems break with no visible connection to your change. Access to model outputs needs to be as deliberate as access to a database.

Data dependencies without tests. Code dependencies have compilers and linters. A data dependency — an upstream table whose meaning quietly changed — has nothing, unless you build it. An upstream team renaming a category from "CANCELLED" to "cancelled" is a production incident that no test suite in the repository will catch.

Pipeline jungles. The classic evolution: a script becomes a script that calls another script becomes an untraceable graph of jobs nobody can reproduce. This one is preventable and the prevention is boring — one pipeline definition, in code, in version control.

Glue code and the 5%

The paper's most quoted observation is that mature systems end up with roughly 5% machine learning code and 95% glue: reading data, reshaping it, handling errors, logging, joining, writing results, retrying.

That ratio is not a failure. It is what taking something from a notebook to a service actually involves, and it is worth knowing in advance so that the first production project does not come as a shock. It also explains why teams value people who can do both halves so highly: the modelling is not the scarce skill.

The minimum viable production system

You do not need a platform. Here is a version that fits in one repository and works:

1. **One pipeline script** that goes from raw data to a saved model artefact, runnable end to end with one command.
2. **A version on everything** — data snapshot, code commit, model file, all recorded together in a small metadata file.
3. **A serving function** that loads the artefact and exposes `predict`, importing the same feature code the training pipeline used.
4. **Logging of every prediction**, with inputs, output, model version and timestamp.
5. **A daily job** that checks input distributions and prediction distributions against the training data and alerts on large moves.
6. **A rollback**, which given item 2 is a matter of pointing at a different model file.

That is a week of work and it covers most of what an expensive platform provides. Add tooling — MLflow for tracking, DVC for data versioning, both free — when the manual version starts hurting, which it will, and not before.

The question to ask on day one

Before any modelling, ask: *what happens to this prediction?*

If the answer is "it appears on a dashboard someone reads weekly", you need almost none of the above and a scheduled notebook is fine. If it is "it decides in 40 milliseconds whether a payment goes through", you need all of it, and you needed to know that before you chose a model with 1,000 trees.

The serving constraint should shape the modelling, and it usually arrives last. Asking early is free.

BEFORE YOU MOVE ON

A team's model writes daily risk scores to a shared table. They retrain with a new feature set, scores shift downward, and two other teams' dashboards break. What is the underlying failure?

A. Undeclared consumers: other systems came to depend on the score's distribution without any contract, so a legitimate model change broke them invisibly.

B. The new features leaked, which is why the score distribution shifted.

C. The model should have been recalibrated after retraining so the scores stayed on the same scale.

D. The retraining pipeline was not versioned, so the change could not be rolled back.

84 · 8 min

Batch, online, and the latency budget

THE ONE THING TO KEEP

Feature retrieval usually dominates the latency budget rather than model inference, so precompute slow-moving features — and run a new model in shadow mode against real traffic before it serves anyone.

Three shapes of serving

Batch. Score everything on a schedule, write the results to a table, and let consumers read them. A nightly job computes tomorrow's churn risk for six million customers.

Simple, cheap, easy to monitor, and easy to roll back — you can rerun yesterday. Its limitation is freshness: a customer who signed up this morning has no score until tonight, and a customer whose circumstances changed at 10am is scored on last night's picture.

Online. A service receives a request and returns a prediction. Fraud scoring, search ranking, anything reacting to a user action.

Fresh, and it drags in everything hard: latency budgets, availability, autoscaling, and the requirement that every feature be computable within the request.

Streaming. Predictions computed continuously as events arrive, results written where consumers can read them. A middle ground for systems built on Kafka or similar.

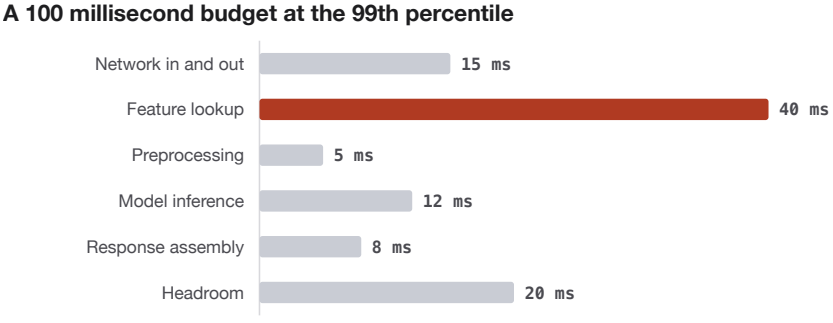
Most teams need less online serving than they assume. Ask what the freshness requirement actually is — often "within an hour" is fine, and an hourly batch job is dramatically cheaper to build and operate than a service.

The latency budget

If you serve online, somebody has a number. Say the API must answer in 100ms at the 99th percentile. That budget covers everything:

```
network in/out          15 ms
feature lookup          40 ms      <- usually the largest
preprocessing            5 ms
model inference         12 ms
response assembly        8 ms
headroom                20 ms
```

Notice where the time goes. It is rarely the model. **Feature retrieval is usually the bottleneck** — a database round trip for the customer's 30-day transaction count costs more than a boosted tree traversal.



The model is 12 of the 100 milliseconds. The database round trip for a customer's 30-day transaction count costs more than a boosted tree traversal, which is why slow-moving features are computed nightly and cached, and only the genuinely real-time ones are fetched inside the request.

That has a design consequence. Precompute what you can. Features that change slowly — account age band, historical averages, segment — can be computed nightly and cached. Only genuinely real-time features are computed in the request. Doing this well is most of what makes an online model affordable.

And measure the tail, not the mean. A mean of 40ms with a p99 of 900ms is a bad service, because 1% of your users are having a terrible time and they are disproportionately your heaviest ones.

Making inference faster

In rough order of effort:

- **Batch requests together.** Ten predictions in one call is far cheaper than ten calls, because the per-call overhead dominates for small models.
- **Use fewer, deeper trees.** 200 trees of depth 8 usually serve faster than 1,000 of depth 4 at similar accuracy.
- **Compile the model.** Treelite and ONNX Runtime convert a model into optimised code, often 2 to 10 times faster with identical predictions.
- **Cache.** If the same inputs recur — the same user, the same query — cache the result with a short expiry.
- **Use a smaller model.** The next lesson but two.

Where a request goes wrong

The failure modes worth handling before launch, because each one has bitten somebody:

A missing feature. The feature store is slow or down. Decide now: fail the request, or predict with an imputed value and flag it. Both are defensible; neither should be discovered at 3am.

An unseen category. Module 3's `handle_unknown="ignore"`. Without it, the transform raises and the request 500s.

An out-of-range value. A negative age, an income of 10^12. Validate inputs at the boundary and reject or clip explicitly rather than letting the model extrapolate silently.

The model file will not load. Version mismatch between the training environment and the serving one. Pin versions, and check that the artefact loads as part of your deploy, before traffic reaches it.

Shadow and canary

Two deployment patterns that cost little and prevent most incidents.

Shadow mode. Run the new model alongside the old one, log both predictions, serve only the old one. After a week you know how the new model behaves on real traffic — including on the inputs your test set never contained — with no risk at all. This is the single most useful practice in this lesson.

Canary. Route 1% of traffic to the new model, watch the metrics, increase gradually. Standard software practice, and it applies unchanged.

Both require that you can run two models at once and route between them, which is worth building early because everything else depends on it.

BEFORE YOU MOVE ON

An online fraud service has a mean latency of 35ms and a p99 of 850ms against a 100ms budget. Where should the team look first?

A. The model's inference time, since the tail suggests some predictions require deeper tree traversal.

B. The mean, since 35ms is already a third of the budget and reducing it lowers the tail proportionally.

C. Feature retrieval, since a database round trip that occasionally misses a cache or waits on a slow query is the usual source of a long tail while the mean stays low.

D. Network time, since 1% of requests may be coming from distant regions.

85 · 9 min

When training and serving quietly disagree

THE ONE THING TO KEEP

Training-serving skew is a systematic difference between how a feature is computed offline and online, and the reliable defences are shipping one serialised pipeline, importing shared feature code rather than copying it, and an automated test asserting both paths produce identical predictions on the same rows.

The most expensive bug in production machine learning

The model scored 0.89 offline and 0.71 in production. Nothing is broken, no errors are logged, and the difference is that the features computed at serving time are not the features the model was trained on.

This is **training-serving skew**, and it is worth its own lesson because it accounts for a large share of failed deployments and is almost never visible from inside the notebook.

Four ways it happens

1. Two implementations of the same feature. Training features are computed in Python by the data scientist. Serving features are computed in Java by the platform team, from a specification in a document. "Average order value over the last 30 days" turns out to mean calendar days in one and rolling 720 hours in the other; one excludes cancelled orders and the other does not.

The difference is small, systematic, and applies to every prediction.

2. Different time semantics. Training computes features from a table containing the whole history, so "transactions in the last 7 days" is computed with hindsight. Serving computes it from data that has arrived so far, which for a system with a two-hour ingestion delay is not the same window. This is the leakage from module 3 reappearing as a production discrepancy.

3. Different preprocessing. The scaler's mean was fitted on training data and hardcoded into the training script. The serving code recomputes it from whatever batch it currently has. Or the category list differs, so one-hot columns are in a different order — which produces no error and completely scrambles the input.

4. Different data sources. Training joins a clean warehouse table; serving reads a live operational database where the same field has nulls, different casing, or a different unit.

The three defences

Ship one implementation. Serialise the whole fitted pipeline — every transformation plus the model, as module 3 described — and call it in both places. The scaler's mean is a value stored in the artefact rather than a computation that happens twice. This eliminates cause 3 entirely and is the highest-value practice in this lesson.

Share the feature code. If features are computed before the pipeline, put that code in a library that both the training job and the serving path import. Not copied. Imported. A copy is a second implementation with a delayed divergence.

Log serving features and compare. For a sample of production requests, write the computed feature values to a table. Then run the training-time computation over the same entities and compare distributions, feature by feature.

That third one is what actually finds skew. Any feature whose serving distribution differs meaningfully from its training distribution is either drift or a bug, and both need investigating. Do it weekly, automatically, and the comparison becomes a report rather than an investigation.

Feature stores

The systematic answer to causes 1 and 2. A feature store computes each feature once and serves it to both training and inference, with two properties that matter:

- **Point-in-time correctness.** When building a training set, it returns each feature's value *as of* the moment of the event, not the current value. This is the leakage guard from module 3 implemented as infrastructure.
- **Consistency.** The same computation feeds both paths, so they cannot diverge.

Feast is the free open-source option; the cloud vendors all sell one. They are genuine infrastructure with real operational cost, and they are worth it when you have many models sharing features and several teams. For one model and one team, shared feature code in a library plus a serialised pipeline gets you most of the benefit for a fraction of the work.

The test that catches it before launch

One procedure, and it takes an afternoon.

Take 1,000 rows from your test set. Score them through the *training* pipeline. Then send the same 1,000 rows through the *serving* path as if they were live requests. Compare the two sets of predictions.

They should be identical to floating-point precision. If they differ at all, find out why before you launch — a discrepancy in the fourth decimal place is usu-

ally a different library version, and a discrepancy in the first is a feature computed differently.

Make this an automated test that runs on every deploy. It is the production equivalent of the label-shuffle check from module 4: cheap, mechanical, and it catches a class of error that reasoning about the code does not.

BEFORE YOU MOVE ON

A model performs well offline and poorly in production with no errors logged. Predictions from the training pipeline and the serving path on the same 1,000 rows differ in the second decimal place. What does that indicate?

A. Ordinary floating-point non-determinism between environments, which is expected at this magnitude.

B. Concept drift, since production data no longer resembles the training distribution.

C. At least one feature is computed differently in the two paths — a different window, unit, category ordering or preprocessing constant — which is a bug rather than a data problem.

D. The serving model is an older version, so the weights differ.

86 · 9 min

Monitoring: what to watch, and in what order

THE ONE THING TO KEEP

Monitor system health, input distributions, prediction distributions and finally actual performance — the first three need no labels and move earlier, and the alert must name which kind of shift it found because covariate, label and concept drift take different remedies.

Four layers, from cheapest to most valuable

1. System health. Request rate, error rate, latency percentiles, memory. Standard software monitoring, and it catches the model container restarting every ten minutes. Necessary and not sufficient — a model returning confident nonsense is perfectly healthy by these measures.

2. Input distributions. For each feature: mean, standard deviation, missing rate, and for categoricals, the set of levels and the rate of previously unseen ones. Compare against the training data.

This is where most problems appear first, because most problems are upstream data problems. A column that becomes 100% null overnight, a currency changing units, a category renamed — all visible here within hours.

3. Prediction distributions. The mean predicted probability, the distribution's shape, the rate of predictions above your decision threshold. You do not need labels for any of this, and it moves before your accuracy metrics can even be computed.

A flag rate that jumps from 2% to 9% overnight is an incident. It might be real fraud, or an upstream change, or a broken feature, and you want to know within the hour rather than at the next monthly review.

4. Actual performance. Accuracy, AUC, calibration, per slice. The thing you truly care about, and the slowest to arrive, because labels lag — some-

times by 90 days for a default, sometimes forever for a loan you declined.

Build all four, and understand that the first three are early warnings for the fourth.

Four layers of monitoring, cheapest first

1. System health	Request rate, errors, latency percentiles, memory. Catches a container restarting; misses a model returning confident nonsense.
2. Input distributions	Means, missing rates, unseen categories, against the training data. Where most problems appear first, within hours.
3. Prediction distributions	Mean score, shape, rate above threshold. A flag rate that goes from 2% to 9% overnight is an incident. Needs no labels.
4. Actual performance	Accuracy, AUC, calibration, per slice. What you care about, and the slowest to arrive — 90 days for a default, never for a loan you declined.

The first three need no labels and move earlier; they are early warnings for the fourth, which is the only one that can see concept drift and the last to arrive, because labels lag by weeks or forever.

Detecting a change in a distribution

Three standard measures, and the practical advice about each.

Population Stability Index. The credit industry's standard. Bucket the feature into deciles based on training data, compare the proportion in each bucket now, and sum `(now - then) * ln(now / then)`. The conventional reading: under 0.1 is no meaningful shift, 0.1 to 0.25 is moderate, above 0.25 is significant. Those thresholds are convention rather than theory, and they are useful precisely because everyone uses the same ones.

Kolmogorov-Smirnov test for continuous features, comparing two distributions. Its problem in production is sample size: with a million rows a week it declares tiny, meaningless differences significant. Look at the test statistic's magnitude and ignore the p-value.

Chi-squared for categorical features, with the same caveat.

My practical recommendation: use PSI for its interpretable thresholds, and always plot the two distributions before acting. A number told you something changed; the plot tells you what.

The three shifts need different responses

From module 3, and this is where the distinction pays off.

Covariate shift — inputs moved, the relationship holds. Your users got younger. Detected by input monitoring. Often the model is still fine; check performance before retraining.

Label shift — the base rate moved. Fraud went from 1% to 4%. Detected by prediction distribution and, later, by calibration. Recalibration usually fixes it and is much cheaper than retraining.

Concept drift — the relationship changed. Fraudsters adapted. Detected only by actual performance, which is why layers 1 to 3 cannot substitute for layer 4. Requires new labelled data and retraining.

An alert saying "drift detected" without saying which of these it is sends the team to the wrong remedy. Name the type in the alert.

Getting labels

The hard part of layer 4, and the options are limited:

- **Wait.** For a 30-day outcome, you know last month's performance today. Acceptable, and it means your monitoring is always a month behind.
- **Sample and label by hand.** 200 predictions a week, reviewed. Expensive and it works, and it gives you a slice-level picture the automatic metrics cannot.
- **Use a proxy.** Did the user click, did the flagged case get confirmed. Fast and biased, because you only observe outcomes for the cases your model acted on.
- **Randomise a small slice.** The exploration policy from module 3. The only way to get unbiased labels in the region your model currently excludes, and it costs money.

Alert on the right things

The common failure is alerting on everything and being ignored within two weeks.

Alert on: prediction rate moving beyond a stated band, a feature's missing rate jumping, unseen categories exceeding a threshold, and performance falling below the number you committed to.

Do not alert on: every PSI above 0.1, every statistically significant difference, or every slice that looks poor this week. Those go on a weekly report that a human reads, not into a pager.

And write down what you would do in response to each alert. An alert with no defined action is not monitoring; it is anxiety with a delivery mechanism.

BEFORE YOU MOVE ON

A model's input distributions are unchanged, but predicted positive rate has climbed from 3% to 8% over two months and calibration has degraded while AUC is stable. What has most likely happened?

A. A data quality problem upstream, since a sudden rise in positive predictions usually means a corrupted feature.

B. Concept drift, since the relationship between features and outcome has changed.

C. Training-serving skew, since the serving features must be computed differently from the training ones.

D. Label shift: the base rate of the positive class has risen, which moves the predicted rate and breaks calibration while leaving the ranking, and therefore AUC, intact.

87 · 8 min

When to retrain, and what breaks when you do

THE ONE THING TO KEEP

Retraining is a deployment and needs the same gates as one, including re-choosing the decision threshold for the new score distribution — and it is the correct remedy only when the relationship has genuinely changed, not for upstream data faults or serving bugs.

Three policies

Scheduled. Retrain every week, or month, or quarter, regardless. Predictable, easy to automate, and it does work nobody asked for and sometimes misses a fast change between runs.

Triggered. Retrain when a monitored metric crosses a line — performance below a threshold, PSI above 0.25, a flag rate outside its band. Responsive, and it requires the monitoring to be trustworthy first.

Continuous. Update the model incrementally as data arrives. Necessary for genuinely fast-moving problems such as ad ranking. It also makes debugging much harder, because "which model made this prediction" has a different answer every hour.

Most teams should start scheduled — monthly is a reasonable default — and add triggers once the monitoring has earned trust. Continuous is a commitment, not an upgrade.

How much history to use

A real decision with no universal answer.

All of it if the relationship is stable. More data, better estimates.

A rolling window — the last 12 months — if the world changes. Old data actively misleads when behaviour has moved on.

Weighted — all of it, with recent rows weighted more. The middle path, and often the best: `sample_weight` decaying with age gives you both the volume and the recency.

Decide it empirically. Train on the last 3, 6, 12 and 24 months, evaluate each on the most recent held-out period, and let the numbers choose. This experiment takes an hour and teams routinely guess instead.

Retraining is a deployment

The most common production mistake in this area is treating a retrained model as automatically safe because "it is the same code with newer data". It is a new model with different behaviour and it needs the same gates as any deploy.

The minimum checks before a retrained model goes live:

1. **Performance on a fixed reference set** — a held-out period that never changes — compared against the current production model. If the new one is worse, stop.
2. **Prediction distribution comparison.** Score the last week of production traffic with both models. A large shift in the distribution needs an explanation before it needs a deployment.
3. **Slice checks.** From module 5. A model that improves overall while collapsing on one group must not ship silently.
4. **The fixed test sets** you built from past failure modes.
5. **Shadow mode** for a period, if the stakes justify it.

Automate all five. A retraining pipeline that trains and deploys without gates will eventually deploy a model trained on a corrupted extract, and it will do so at 4am on a Sunday.

The threshold moves too

Easy to forget: your decision threshold was chosen for the old model's score distribution. A retrained model's scores are distributed differently, so the same threshold produces a different flag rate.

Re-choose the threshold on validation data as part of every retraining, using the same cost or capacity criterion from module 5. Otherwise the model improves and the system behaves worse, which is a confusing failure to debug and a common one.

Keep the old model

Rollback should take minutes, which means the previous model artefact stays available and the serving layer can point at either. This is the same discipline as any deploy and it is the single thing most likely to save you.

Also keep a record: which model version made which prediction. When someone asks in March why a decision was made in January, you need to know which model was live and what it was trained on. In regulated settings this is a requirement rather than a convenience.

The honest counterweight

Retraining is not always the answer, and it is reached for reflexively.

If performance dropped because an upstream table changed, retraining teaches the model the broken data. If it dropped because the population shifted into a region you have no examples of, retraining on data that lacks that region changes nothing. If it dropped because of a serving bug, retraining hides the symptom while the bug remains.

Diagnose before retraining. Look at the errors, check the inputs, compare the two serving paths. Retraining is a remedy for one specific cause — the relationship in the data has changed and you now have labels reflecting that — and applying it to any other cause wastes a week and leaves the fault in place.

BEFORE YOU MOVE ON

After an automated monthly retrain, a fraud model's offline AUC improves slightly but the number of cases sent for review triples overnight. What is the most likely cause?

A. The new model is overfitting, so it flags many more cases while scoring well offline.
B. The retraining window included a period of elevated fraud, so the model learned a higher base rate.
C. The threshold was carried over from the previous model, but the new model's scores are distributed differently, so the same cut point now selects a much larger fraction of traffic.
D. Training-serving skew was introduced by the retrain, so serving features no longer match training features.

88 · 8 min

Being able to build the same model twice

THE ONE THING TO KEEP

Reproducibility requires pinning code, data, environment and seeds together, and because GPU and parallel arithmetic are not bitwise deterministic, the standard to hold is a rerun scoring within noise of the original rather than an identical artefact.

The test

Can you rebuild, byte for byte, the model that is running in production right now?

Most teams cannot. That matters for three practical reasons: you cannot debug a model you cannot reconstruct, you cannot demonstrate to an auditor

what a decision was based on, and you cannot tell whether a change you made improved things or whether something else moved underneath you.

The four things that must be pinned

1. Code. A git commit hash. The easy one, and the one everybody does.

2. Data. The hard one. "The customers table" is not a version; it changes every night. You need either a snapshot, or a query with an explicit as-of timestamp, or a content hash of the exact extract. DVC and lakeFS are free tools for this; a dated Parquet file in object storage with its hash recorded is a perfectly good version-one answer.

3. Environment. Library versions. `pip freeze > requirements.txt` at minimum; a container image digest if you can. scikit-learn changes default parameters between versions, and a model trained under 1.2 and retrained under 1.6 with identical code can differ meaningfully.

4. Randomness. Every seed, everywhere.

```
import random, numpy as np, torch
SEED = 0
random.seed(SEED); np.random.seed(SEED);
torch.manual_seed(SEED)
```

And pass `random_state=SEED` to every splitter, sampler and estimator that accepts it. A model whose score moves by 1.5 points between runs cannot be compared against anything.

Where seeds are not enough

Worth knowing so you do not chase a ghost.

GPU non-determinism. Some CUDA operations accumulate floating-point sums in a non-deterministic order, so results differ in the last decimal places and can amplify over a long training run. `torch.use_deterministic_algorithms(True)` forces deterministic implementations where they exist and is measurably slower.

Parallelism. Multi-threaded reductions sum in whatever order threads finish. Same cause, same tiny differences.

Hardware. Different CPU instruction sets can produce slightly different floating-point results for the same code.

So exact bitwise reproducibility across machines is often not achievable. **Statistical reproducibility is** — the same data, code, environment and seeds should give a score within noise of the original, and if it does not, something is unpinned. That is the standard to hold yourself to.

Tracking runs

Once you are running dozens of experiments, memory and file names stop working.

```
import mlflow
with mlflow.start_run():
    mlflow.log_params({"max_depth": 6, "lr": 0.05, "data":
"2026-08-14"})
    mlflow.log_metrics({"val_auc": 0.847, "val_ap": 0.312})
    mlflow.log_artifact("model.joblib")
```

MLflow is free and runs locally with no server. Weights and Biases has a free tier and a better interface. Or a CSV appended to by your training script, which is genuinely adequate for one person and infinitely better than nothing.

What to log, in every run: the data version, the code commit, the seed, every hyperparameter, every metric on every slice, the training duration, and the model artefact. When someone asks in six months why the model changed, this is the only thing that answers.

The model registry

A place where model artefacts live with metadata: version, training data, metrics, stage (staging, production, archived), and who approved the promotion.

This is what makes "which model is in production and what was it trained on" a lookup rather than an investigation, and it is what makes rollback a pointer

change. MLflow includes one; a directory with a naming convention and a JSON file per model is a legitimate starting point.

The lightweight version that actually gets done

If all of the above sounds like a quarter of work, here is the version that takes an afternoon and gets you 80% of the benefit:

```
{
  "model_version": "2026-09-06-a",
  "git_commit": "4294d8f",
  "data_snapshot": "s3://bucket/train/2026-09-01.parquet",
  "data_sha256": "e3b0c442...",
  "seed": 0,
  "sklearn": "1.6.1",
  "metrics": {"val_auc": 0.847, "test_auc": 0.839},
  "threshold": 0.62,
  "trained_by": "pipeline",
  "trained_at": "2026-09-06T11:04:00Z"
}
```

One JSON file written next to every model artefact by the training script. That is it. It answers almost every question anyone will ask, and the discipline of writing it forces you to actually know the data version — which, more often than anyone admits, is the moment a team discovers they do not.

Reproducibility is what makes comparison possible

One closing argument, because reproducibility is usually presented as a compliance chore.

Every claim in modules 4 and 5 — this model beats that one, this feature helped, this threshold is right — assumes that rerunning the same configuration gives the same answer. When it does not, you cannot tell an improvement from a different random seed, and every comparison you make becomes an opinion supported by a number.

So the discipline is not paperwork. It is the precondition for the measurement being about the model at all. A team with unpinned data is not doing worse

science slowly; it is doing something that only resembles science, and it usually finds out during the first serious disagreement about whether a change helped.

BEFORE YOU MOVE ON

A team pins the git commit and sets every random seed, but a rerun of last month's training gives 0.831 AUC where the original gave 0.847. What is the most likely unpinned element?

A. The metric computation changed, since AUC implementations differ between library versions.
B. GPU floating-point non-determinism, which accumulates over training into a visible score difference.
C. The seed was set after the data was loaded, so the split differed.
D. The data: the training query reads a live table that has changed since last month, so the same code trained on different rows.

89 · 8 min

AutoML: what it automates, and what it cannot

THE ONE THING TO KEEP

AutoML automates model selection and hyperparameter search — genuinely well, and free — but it cannot frame the problem, build features, detect leakage or choose the metric, which is where most of a model's quality is determined.

What it does

AutoML systems search over preprocessing steps, model families, hyperparameters, and often ensembles of the results, using cross-validation to choose. You supply a table and a target column; it returns a fitted pipeline.

The free options are good and worth knowing by name:

- **auto-sklearn** — searches scikit-learn pipelines, uses meta-learning from previous datasets to start the search in a promising region.
- **AutoGluon** — from Amazon, free and open source. Trains several strong models and stacks them. It is frequently the strongest of the free options on tabular data with no tuning at all.
- **FLAML** — from Microsoft, free, optimised for finding a good model quickly on a small compute budget.
- **H2O AutoML** — free, mature, with a usable interface.
- **TPOT** — searches pipelines with genetic programming, and outputs readable Python code for the winner, which is genuinely useful for learning.

The paid ones — Vertex AI, DataRobot, SageMaker Autopilot — add hosting, monitoring and an interface. The modelling is not meaningfully better than AutoGluon.

```
from autogluon.tabular import TabularPredictor
p = TabularPredictor(label="target",
eval_metric="roc_auc").fit(train_df)
p.leaderboard()
```

When it is the right call

As a baseline. Run it for an hour on any new problem. Now you know roughly what is achievable, and your hand-built model has a number to beat. If AutoML gets 0.86 and two weeks of your work gets 0.84, that is worth knowing on day one rather than at the end.

When nobody on the team is a specialist. A small business with a real prediction problem and no data scientist gets far more from AutoGluon than from a poorly tuned model built by someone learning as they go.

When time matters more than the last two points. Many business problems are worth solving to 90% of achievable quality this week.

As a check on your own work. If AutoML substantially beats your model, you have made a mistake somewhere and it is worth finding.

What it does not do

This is the part the marketing does not cover, and it maps exactly onto the modules of this course.

It does not frame the problem. What one row is, what the target means, when the prediction is made. Module 1. AutoML accepts whatever table you hand it, including the wrong one.

It does not build features. This is the largest source of accuracy on tabular problems, it comes from domain knowledge, and no search invents `debt / income` or `days_since_last_purchase` from a schema. AutoML tunes a model over the features you gave it.

It does not detect leakage. In fact it makes leakage worse, because a search that maximises validation score will enthusiastically select whatever configuration best exploits the leaked column. A leaked feature plus AutoML produces a spectacular number and a useless model.

It does not choose the metric or the threshold. It optimises what you specified. Specifying the wrong thing, at scale, is what AutoML is unusually good at.

It does not tell you whether the sample is representative. Module 3's selection bias is invisible to any search.

It does not decide whether the model should be built. Module 1's argument about rules, and module 5's about fairness, are outside its scope entirely.

The honest summary: **AutoML automates the part of this course that is easiest to automate, which is model selection and hyperparameter tuning — and that was never where most of the value was.**

The costs

Compute. A serious search is hundreds of model fits. On a free tier this means hours.

Complexity of the result. A stacked ensemble of 12 models is slow to serve, large to store, and impossible to explain to anyone.

Overfitting the validation set. Hundreds of configurations evaluated against one validation set is exactly module 4's problem in concentrated form. Keep a test set the search has never touched, and expect the final number to be below the search's best.

Skill. A team that only runs AutoML does not develop the judgement to notice when the output is wrong, and the output being wrong is not rare.

How to actually use it

Use it as an instrument rather than a replacement. Run it early to establish what is achievable. Spend your time on framing, features and evaluation — the things it cannot do. Then run it again at the end on your engineered features, and ship whichever is better.

That use is unambiguously good practice, and it is not the use the products are sold for.

BEFORE YOU MOVE ON

A team runs AutoML on a churn dataset and gets 0.97 AUC, far above their hand-built model's 0.82. What should they do first?

A. Deploy it, since the search evaluated hundreds of configurations against proper cross-validation.

B. Investigate for leakage, because a large unexplained jump on a hard problem is the signature of a feature containing the answer — and a search that maximises validation score selects the configuration that exploits it best.

C. Rerun with a larger time budget, since the search may not have converged and the true score could be higher still.

D. Retrain their hand-built model with the hyperparameters AutoML selected, to close the gap.

90 · 8 min

Making a model small enough to be affordable

THE ONE THING TO KEEP

Quantisation, distillation and pruning trade a small amount of accuracy for large reductions in size and latency — and unstructured pruning reduces parameter count without reducing runtime, which is the distinction most often missed.

Why size becomes the problem

Offline, a model's size does not matter. In production it decides three things: what hardware you need, what latency you can promise, and what the monthly bill is.

A model at 10,000 predictions per second costs real money per point of accuracy, and there is a size below which it runs on a phone and above which it

does not. That threshold is often the difference between a product and a demo.

Quantisation

Store weights in fewer bits. A float32 weight becomes int8 — four times smaller, and integer arithmetic is faster on most hardware.

Post-training quantisation converts a trained model directly. No retraining, a few lines, and accuracy typically falls by well under a point for int8. This is the first thing to try and it is often the only thing needed.

Quantisation-aware training simulates the reduced precision during training so the model adapts to it. Better accuracy at low bit widths, requires retraining.

For large language models, 4-bit quantisation via GPTQ or AWQ is what makes a 7-billion-parameter model run on a consumer GPU at all. The quality loss is real and modest, and the alternative is not running the model.

Distillation

Train a small "student" model to reproduce a large "teacher" model's outputs — not the hard labels, but the full probability distribution.

The interesting part is why this works better than training the small model on the labels directly. A teacher that outputs `[cat 0.7, lynx 0.25, car 0.05]` is communicating that cats and lynxes are similar and neither resembles a car. The hard label `cat` contains none of that. Hinton and colleagues called it dark knowledge, and it means the student learns from a much richer signal than the original training data provided.

DistilBERT is 40% smaller and 60% faster than BERT while retaining about 97% of its performance on standard benchmarks. Those are typical numbers for a well-executed distillation.

Pruning

Remove weights that contribute little. Many trained networks have most of their weights near zero, and removing them changes the output hardly at all.

Unstructured pruning zeroes individual weights. Very high sparsity is achievable — 90% is common — and it gives no speedup on standard hardware, because a sparse matrix with scattered zeros is still processed as a dense one.

Structured pruning removes whole channels, filters or attention heads. Lower achievable sparsity and a real speedup, because the resulting model is genuinely smaller.

The distinction matters and is frequently missed: unstructured pruning reduces the number of non-zero parameters, which is a different thing from reducing time or memory.

For tree models

The boosting equivalents, which are simpler and rarely discussed:

- **Fewer trees.** Early stopping already chose a number; check whether 60% of them give nearly the same score. Often they do.
- **Fewer, deeper trees.** 200 at depth 8 rather than 1,000 at depth 4, at similar accuracy and much faster traversal.
- **Compile it.** Treelite converts a tree ensemble to C and typically runs 2 to 10 times faster with identical predictions.
- **Drop features.** From module 6's importance work. Fewer features means less to compute at serving time, which is usually the dominant cost anyway.

Deciding what to spend

The useful framing is cost per unit of accuracy, computed honestly.

A model at 0.847 AUC costs ₹40,000 a month to serve. A model at 0.839 costs ₹6,000. Is 0.008 of AUC worth ₹34,000 a month?

Sometimes obviously yes — in fraud, 0.008 might be crores of losses. Sometimes obviously no. The point is that this is an answerable question that most teams never ask, because the modelling and the infrastructure budget are held by different people.

Build the estimate. Predictions per month, cost per prediction, and the business value of the accuracy difference. It takes twenty minutes, it frequently changes the decision, and it is the kind of analysis that makes a machine learning person unusually valuable in a room of engineers.

Where to start

Measure first. Profile the serving path and find where the time actually goes — and recall from the serving lesson that it is usually feature retrieval rather than the model. Optimising a model that accounts for 12% of your latency caps your possible improvement at 12%, and people do this regularly because the model is the part they know how to change.

BEFORE YOU MOVE ON

A team prunes 90% of their network's weights using unstructured magnitude pruning and finds inference is no faster. Why?

A. The pruned weights were set to zero but not removed, and standard hardware processes a matrix with scattered zeros exactly as it processes a dense one.

B. 90% sparsity is too aggressive, so the model compensates by requiring more forward passes.

C. Pruning must be followed by quantisation to produce any speedup.

D. The speedup appears only after fine-tuning the pruned model to recover accuracy.

91 · 9 min

When the model changes the data it will be trained on

THE ONE THING TO KEEP

A deployed model shapes the data its successor is trained on, so offline metrics measure agreement with the previous model's choices — and only randomised exploration, logged propensities and control groups produce evidence that escapes the loop.

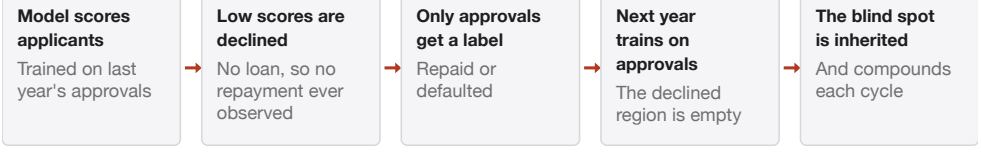
The property that makes this different from other software

A deployed model acts on the world, and the world it acts on generates the data for the next model. That loop has no equivalent in ordinary software, and it produces failures that no amount of offline evaluation detects.

Four shapes it takes

Selective labelling. A loan model declines an applicant. You never learn whether they would have repaid. Next year's training data contains only approved applicants, so the model's blind spot is inherited by every successor. Module 3 named this; here it compounds year on year.

Selective labelling, one cycle



The declined applicant who would have repaid never appears in any dataset, so offline metrics measure agreement with last year's model rather than with the world. Only randomised approvals, logged propensities or a control group produce evidence from outside the loop.

Self-fulfilling prediction. A model predicts a district is high-crime. More officers are sent. More incidents are recorded there — not because more hap-

pened, but because more were observed. The next model sees confirmation. Nothing about the district changed and the prediction has manufactured its own evidence.

Position and exposure bias. A recommender ranks item A first. It gets clicked more because it is first. The click data says A is better. Next model ranks it higher still. The measured preference is largely a measurement of the previous model's choices.

Behavioural adaptation. People learn the rule and change to satisfy it. Credit scores optimised for, spam filters evaded, ranking signals gamed. The feature stops meaning what it meant, which economists call Goodhart's law: a measure that becomes a target ceases to be a good measure.

How to notice

The loop is hard to see from inside, and three signals are worth watching for:

- **Confidence rising while errors stay flat.** The model agrees with itself more and is no more correct.
- **Predictions concentrating.** The distribution narrows over successive retrainings — fewer items recommended, fewer applications approved, a shrinking effective catalogue.
- **Offline metrics improving while the business metric does not.** The clearest signal. Your model is getting better at predicting data it helped create.

What actually helps

Randomised exploration. Approve 2% at random regardless of score. Show a random result to 1% of searches. This is the only method that produces genuinely unbiased data about the region your policy excludes, it costs money, and there is no substitute. Treat the cost as the price of knowing rather than as waste.

Log the propensity. Record, for each action, the probability the system had of taking it. That single logged number enables inverse propensity weighting,

which lets you estimate offline how a *different* policy would have performed. This is the foundation of off-policy evaluation and it is nearly free at logging time and impossible to reconstruct afterwards. If you take one engineering action from this lesson, log the propensity.

Hold out a control group. A slice of users who never receive the model's output. Expensive, uncomfortable, and the only clean measure of what the system is doing overall.

Measure diversity as a target. Not only precision. Catalogue coverage, the share of recommendations going to the top 1% of items, the spread of approvals across segments. What you do not measure will narrow.

Cap the loop. Do not train on outcomes the model itself determined, where you can avoid it. If a case was auto-approved by the model, its outcome is weaker evidence than one reviewed by a human, and weighting accordingly is a defensible choice.

The case that ended the argument

The most cited example is predictive policing, where the loop is visible and the harm is concrete: allocation of officers by predicted crime, arrest data as the label, and arrest rates that reflect where officers were sent. Several jurisdictions have discontinued such systems, and the evaluations that led to those decisions found the models largely reproducing their own deployment history.

The machine learning content of that finding is not exotic. It is module 1's distinction between prediction and intervention, applied to a system whose predictions are interventions.

The obligation

When a model affects the data it is trained on, offline evaluation is not sufficient evidence that it works. That is not an ethical position; it is a statistical one. The training data is no longer a sample of the world, it is a sample of the world as shaped by the previous model, and every metric computed on it inherits that shaping.

The answers — randomisation, propensity logging, control groups, online experiments — all cost something and all exist. Choosing not to pay for them is a decision, and it should be a stated one rather than an omission.

BEFORE YOU MOVE ON

A recommender's offline precision@10 improves with every monthly retrain, but total purchases per user are flat. Catalogue coverage has fallen from 40% to 12%. What is happening?

A. The model is overfitting the training data, which is why offline metrics improve without a business effect.

B. The evaluation metric is wrong, and NDCG would show the true picture.

C. Each model is evaluated on interactions its predecessor generated, so it is measured on agreement with past recommendations; coverage falling to 12% shows the catalogue narrowing as the loop reinforces itself.

D. Purchases are flat because of seasonality, and the model is genuinely improving.

---

92 · 9 min

What you owe the people in the data

THE ONE THING TO KEEP

A model card recording the target definition, training population, per-slice performance and out-of-scope uses costs an hour and outlives everyone involved — and the legal questions around copyright, erasure and automated decisions are genuinely unresolved and differ by country.

The rows are people

Every row in a customer table is a person who did not choose to be a training example. That fact does not stop the work, and it does shape a small number

of obligations that are cheap to meet and awkward to retrofit.

Write the model card

A model card is one page recording what the model is and is not. Mitchell and colleagues proposed the format in 2019 and it has become standard practice, including on Hugging Face where many published models carry one.

What it contains:

- **What it predicts**, in one sentence, with the exact target definition and the moment of prediction.
- **Who it was trained on** — time period, geography, population, and how the rows were selected.
- **Performance overall and per slice**, with sample sizes for each slice.
- **Intended use**, and explicitly **out-of-scope use**. "This was trained on applicants aged 21 to 65 in three states and should not be used outside that population."
- **Known limitations**, including which groups it performs worse on and by how much.
- **The fairness criterion chosen**, from module 5, and what it costs.
- **When it was trained and when it should be reviewed.**

This takes an hour. It is the difference between a model that someone can use responsibly in two years and one whose limits live only in the memory of a person who has left.

A data card does the same for a dataset — how it was collected, what consent was given, known gaps — and is at least as valuable, because datasets outlive models.

Data minimisation

Collect what you need for the prediction, not everything available.

This is a legal requirement under GDPR and in various forms elsewhere, and it is also good engineering: fewer fields means less to maintain, less to secure,

less to leak, and fewer routes for a protected attribute to enter the model by proxy.

The question to ask of each field is not "could this help" but "does this help enough to justify holding it". A model that improves by 0.002 AUC because it knows people's precise location is a trade you should make deliberately.

Deletion, and what it means for a model

A person exercises their right to erasure. You delete their rows. The model trained on those rows still exists and still encodes some information about them.

Whether a trained model constitutes personal data is genuinely unsettled, and it differs by jurisdiction. Regulators have taken different positions, and there is no settled answer to point at. Some organisations retrain on a schedule that bounds how long any individual's influence persists; some treat the model as derived, aggregated output; some do neither and have not been tested.

The research area is called machine unlearning, and it is early — exact unlearning is expensive and approximate methods are hard to verify. What you can do now: retrain regularly enough that deletion propagates within a stated period, and write down what that period is.

And note that legal holds cut the other way. When retention is required by law — a fraud investigation, a regulatory obligation — that obligation can override an erasure request. Both rules exist and they conflict, and which wins is a legal question rather than an engineering one.

Privacy attacks are real and often overstated

Be accurate about this rather than alarming.

Membership inference — determining whether a specific record was in the training set — works better against overfitted models and against those trained on small datasets. It is a genuine risk and a well-regularised model on a large dataset leaks much less.

Model inversion — reconstructing training examples — has been demonstrated, most convincingly against large generative models that can reproduce memorised training text or images. Reconstructing a row of tabular training data from a gradient boosting model is far harder.

Differential privacy provides a formal guarantee by adding calibrated noise during training. It costs accuracy, sometimes a lot, and it is the right answer when the guarantee is required. `Opacus` for PyTorch is the free implementation. It is not a routine default and should not be presented as one.

The law is unsettled, and saying so is the honest position

Three questions with no agreed answer across jurisdictions, and this course will not pretend otherwise:

- **Whether training on copyrighted material is permitted.** Different countries take different views, several major cases are unresolved, and the fair-use and text-and-data-mining arguments have not been settled by courts in a way that transfers internationally.
- **Who owns a model's output**, and whether it can be copyrighted at all.
- **What counts as a decision requiring human review** under provisions like GDPR Article 22, and how far automated support goes before it becomes automated decision-making.

These differ by country and are moving. If your work touches any of them, that is a question for a lawyer in your jurisdiction and not for a course. What a course can do is tell you the questions exist, so that you notice you are inside one.

The last thing

The most common ethical failure in this field is not a bad decision. It is an undocumented one — a threshold set without a reason recorded, a fairness criterion chosen by default, a population excluded without anyone noticing.

Write down what you chose and why. It costs an hour, it survives your departure, and it is what turns a model from a black box somebody inherited into a

decision somebody can be accountable for.

BEFORE YOU MOVE ON

A user exercises their right to erasure and the team deletes their rows from the warehouse. What remains unresolved?

A. Nothing — deleting the source rows satisfies the request completely under every framework.

B. The backup copies of the warehouse, which must also be purged before the request is complete.

C. Whether the model trained on those rows must also be retrained, since whether a trained model constitutes personal data is disputed and differs by jurisdiction.

D. Whether the deletion breaks the model's calibration, which would require recalibration on the remaining data.

CASE STUDY · MODULE 9

A week of retraining that fixed nothing

A pharmacy chain in Hyderabad with 212 branches, a four-person data team, and a demand forecast that started missing badly in the second week of March.

Sanjeevini Pharmacy forecasts daily demand per stock-keeping unit per branch, about 1.4 million forecasts a night, feeding an automatic replenishment order. When the forecast is low, a branch runs out and a customer walks to the shop next door. When it is high, slow stock ages towards expiry.

The model had been stable for fourteen months. Weighted mean absolute percentage error sat between 18 and 21 per cent. In the second week of March it went to 34 and stayed there.

The commercial director asked for a retrain by Friday. The reasoning was not unreasonable: the model is stale, the world has moved, retrain it on recent data. The data team's lead, Anwar, asked for two days to diagnose first, and had to argue for them, because two days is two days of stockouts.

The monitoring told him something before he opened a notebook. System health was clean. Prediction distributions had shifted downward — the model was forecasting less than it had. Input distributions had a specific and loud alarm: for one feature, `pack_size_units`, the missing rate had gone from under one per cent to 23 per cent on the eighth of March and stayed there.

That is a covariate shift in the monitoring's language and a data fault in reality. Twenty-three per cent is not a population moving. It is something upstream breaking on a particular day.

The trace took four hours. The procurement team had migrated their supplier master to a new system on the seventh of March. In the new system pack size for one category of products is recorded in a separate strength field, and the old column is null for anything created after the migration. Nobody told anyone, because from procurement's point of view nothing had broken.

There was a second effect stacked on the first, and it is the one that would have made a retrain actively harmful. The imputer in the pipeline filled the missing pack size with the training median. For the affected category — largely paediatric syrups — the median across the whole catalogue is meaningfully smaller than the true value, so the forecast for those lines was systematically low, which is exactly the error pattern the branches were reporting.

Now the decision, and both options had a cost that landed on somebody.

Retrain on recent data, as asked. It would have produced a better-scoring model, because a model trained on the last twelve weeks learns that pack size is missing for a quarter of rows and adapts to predict without it. The score recovers. The forecasts stay wrong in a new way, the fault stays in the upstream system, and the next person to look at it inherits a model that has quietly encoded a broken join. Retraining teaches the model the broken data.

Fix the upstream mapping. Correct, and it required a change from a procurement team in the middle of a migration, with no incentive and no spare week, and it would take longer than Friday.

Patch the pipeline to read the new strength field when the old column is null, and leave the model alone.

> STOP HERE
>
> Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

They patched the pipeline, in an afternoon, and the error was back under 22 per cent within four days with no retraining at all.

The procurement mapping was fixed six weeks later through the normal change process, and the patch was left in place because a fallback that costs nothing and covers a class of failure is worth keeping.

Three things changed permanently as a result. Unseen-category and missing-rate alerts were promoted from the weekly report to the pager, with a stated action next to each: a jump in the missing rate for any feature is an upstream investigation, not a modelling one. The team wrote down a rule that no retrain ships without a diagnosis naming which of covariate shift, label shift or concept drift it is a remedy for. And the retraining pipeline gained the gate it had been missing — performance on a fixed reference period against the current production model, a prediction distribution comparison, and slice checks by branch and by category.

Anwar's note to the commercial director afterwards made the argument that the two days had been the cheapest part of the incident: a retrain on the Friday would have hidden the fault, restored the metric, and left the paediatric lines under-ordered for as long as anyone cared to look.

A retrain would have restored the error metric. Why would that have been the wrong outcome?

Because the metric would have recovered while the forecasts stayed wrong for a specific category. A model trained on data where pack size is missing for a quarter of rows learns to predict without it and scores well on data with the same fault. The remedy matches the diagnosis only when the relationship in the world has genuinely changed. Here an upstream field had moved, and retraining converts a visible fault into an invisible one.

Why did input monitoring find this before the performance metrics did?

Because input monitoring needs no labels and moves first. Actual demand takes a day or more to arrive, and the error metric is a weighted average over 1.4 million forecasts, so a fault confined to one category moves it slowly. A missing rate jumping from one per cent to twenty-three on a single day is unambiguous, visible within hours, and points at a date that can be matched against a change log.

The imputer turned a missing value into a wrong value. What would have made that failure louder?

Adding a missingness indicator alongside the imputed value, as the missing-values lesson recommends, so the model and the monitoring both see that the value was absent rather than small. Beyond that, letting the gradient boosting model handle

nulls natively would have avoided substituting a catalogue-wide median into a category where it does not apply. Either way, the fix is to make absence visible rather than to replace it with a plausible number.

MODULE 9 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 An online scoring service must answer within 100 milliseconds at the 99th percentile and is over budget. Where does the time most often go?

A. Model inference, which dominates the budget for any large tree ensemble.

B. Response serialisation, which grows with the number of features in the payload.

C. Feature retrieval, usually a database round trip.

D. Network transfer in and out, which is the largest fixed cost in most services.

2 Which test catches a feature that is computed differently in the serving path from in training?

A. Cross-validate the model with more folds and check that the fold scores agree.

B. Score the same 1,000 rows through both paths and compare.

C. Compare the training and test distributions of every feature in the dataset.

D. Run the new model in shadow mode and compare its accuracy against the old model's.

3 The base rate of fraud rises from 1 per cent to 4 per cent while the model's ranking quality is unaffected. What is the cheapest correct remedy?

A. Recalibrate the probabilities and re-choose the threshold.

B. Retrain the model on the most recent twelve months of labelled transaction data.

C. Rebuild the feature pipeline, since the input distributions must also have shifted.

D. Do nothing, because AUC is unchanged and ranking is what the queue depends on.

4 A retrained model scores better on the fixed reference set and ships with the old decision threshold unchanged. What is likely to happen?

A. Nothing, since the threshold is a property of the metric rather than of the model itself.

B. The model becomes miscalibrated, because a threshold and a calibration map are the same thing.

C. Both precision and recall improve, since the new model is better on every measure reported.

D. The flag rate changes, because the scores are distributed differently.

5 A team runs AutoML on a table that contains a leaked column. What is the effect?

A. It detects the leak, because cross-validation exposes a column that is too predictive.

B. It selects whichever configuration exploits the leak best.

C. It down-weights the column, since a feature that predictive is treated as suspicious.

D. It fails to converge, because the classification problem has become trivially separable.

6 A lender wants to estimate offline how a different approval policy would have performed. What must have been logged at the moment each decision was made?

A. The model's version number together with the values of all its input features.

B. The reviewer's stated reason for each approval or decline that was recorded.

C. The probability the system had of taking the action it took.

D. The applicant's eventual outcome, including for the applicants who were declined.

EXAMINATION

Machine Learning, Foundations — final examination

This paper covers the whole course: framing a problem as a dataset with a defined row, target and moment of prediction; fitting and diagnosing linear and logistic models; building a preprocessing pipeline that leaks nothing; designing a validation scheme that matches the structure of the data; measuring a classifier honestly and choosing a threshold; trees, forests and gradient boosting; clustering, dimensionality reduction and embeddings; neural networks and transfer learning; and taking a model to a running service and keeping it honest. You get twenty-five questions, drawn from a larger pool, with every module represented. The pass mark is 70 per cent. There is no timer, so read each question as slowly as you need to. If you do not pass, you can retake after thirty minutes and the questions will be drawn fresh, so a retake is a different paper rather than the same one again. A teacher can open the next course for you at any time regardless of this result.

On the website a sitting is 25 questions drawn from this pool and the pass mark is 70%. There is no time limit, there and here. Passing opens the next course in the track.

1 `1. What learning is, and what it is not`

A team's write-up says the tree's max_depth was learned from the data. What is wrong with that description?

A. Depth is chosen by you on validation data, not fitted from the rows.

B. Depth is fitted from the rows, but only once the split thresholds have been chosen.

C. Depth is fitted on the test set, which is where hyperparameters are supposed to be chosen.

D. Nothing is wrong; depth is a parameter of the model exactly like a coefficient.

2 1. What learning is, and what it is not

A hospital labels 'patients with condition X' using billing codes and trains a model on those labels. What has the model learned?

A. The presence of condition X, since billing codes are audited for accuracy each year.

B. Nothing usable, because billing codes are categorical rather than numeric quantities.

C. Which patients get coded, which varies by clinic and insurer.

D. The severity of condition X, because codes are assigned in bands by severity.

3 1. What learning is, and what it is not

k-means returns six clusters from a bank's transaction table. What does that establish?

A. That six customer types exist, because the algorithm minimises within-cluster distance.

B. Very little, since six clusters come back from random noise too.

C. That the transactions carry at least six dimensions of genuine structure.

D. That labelling can now begin, because each cluster has a natural name.

4 1. What learning is, and what it is not

Which grain is the only one that can answer 'should we have approved this application'?

A. One row per customer, aggregated across every loan that customer has held.

B. One row per loan, because that is the level at which default is actually recorded.

C. One row per customer-month, because it supports a rolling thirty-day prediction.

D. One row per application, including the rejected ones.

5 1. What learning is, and what it is not

A binary model predicts 0.0001 for a row whose true label is 1. Roughly what does log loss charge for that row?

A. About 1.0, because the label is one and the prediction is close to zero.

B. About 9.2, and it rises without limit as the prediction nears zero.

C. About 0.0001, because the loss is simply the predicted probability itself.

D. Exactly 1, since the row is counted as one mistake like any other.

6 1. What learning is, and what it is not

A team sets class_weight='balanced' and then reads the model's output of 0.6 as a sixty per cent chance. What is the error?

A. There is no error; class weights change the loss but never the predicted probabilities.

B. The error is reading 0.6 at all, because a decision threshold must always be 0.5.

C. Weighting distorts the probabilities, so 0.6 is no longer a rate.

D. Balanced weighting is defined only for trees and not for logistic regression.

7 1. What learning is, and what it is not

Two experienced coders disagree on one support ticket in six. A model reaches 85 per cent accuracy on that labelling task. What follows?

A. It may already be at the ceiling the labels can express.

B. It is clearly underfitting, because fifteen per cent of the rows are still wrong.

C. The labels should be re-collected until the two coders agree on every ticket.

D. Accuracy is invalid on this task, so a different metric has to be reported.

8 2. Linear models and the machinery of fitting

A linear model fitted on flats between 40 and 130 square metres is asked about a flat of 400. What happens?

A. It raises an error, because the value lies outside the range it was fitted on.

B. It returns the mean of the training targets, in the way a decision tree would.

C. It returns a confident number with no sign that it is guessing.

D. It returns a wider prediction interval to reflect the extrapolation involved.

9 2. Linear models and the machinery of fitting

A linear model returns a coefficient of +184,000 on one city and -183,000 on another. What should you look for first?

A. Columns that are exact combinations of one another.

B. A learning rate that was set far too high during the iterative fitting process.

C. Outliers in the target, which are dragging the whole fitted line upward.

D. A missing intercept term, which forces the fitted line through the origin.

10 2. Linear models and the machinery of fitting

What does raising the mini-batch size from 32 to 4,096 do to a training run?

A. More steps per epoch, because more rows are processed in total during the pass.

B. Fewer and less noisy steps per epoch, sometimes generalising slightly worse.

C. Nothing at all, provided the learning rate is left exactly where it was.

D. Better generalisation, because each gradient estimate is far more accurate.

11 2. Linear models and the machinery of fitting

Validation loss sits below training loss for a whole run. Which is the innocent explanation?

A. The validation set happens to be easier than the training set, which is the usual cause.
B. The model has converged, and the two curves always cross once that has happened.
C. Early stopping fired before the training loss had a chance to settle down.
D. Dropout is active during training and switched off at evaluation.

12 2. Linear models and the machinery of fitting

An unpenalised logistic regression reports a coefficient of 34 on one binary flag. What is the likely cause?

A. The feature was not standardised, so its coefficient is expressed on a large scale.
B. The optimiser diverged, and the learning rate ought to be reduced by a factor of ten.
C. That flag separates the classes perfectly, and is probably leakage.
D. The flag is correlated with the intercept term, which inflates its fitted coefficient.

13 2. Linear models and the machinery of fitting

An article can be about politics and economics and India at the same time. What output layer is correct?

A. One sigmoid per label, each with its own threshold.
B. Softmax over the labels, so that the predicted probabilities sum to exactly one.
C. Softmax with a lowered temperature, so that several labels can score highly at once.
D. A single sigmoid, with the labels ordered by their frequency in the training data.

14 2. Linear models and the machinery of fitting

A multi-class model reports weighted F1 of 0.91 while scoring zero on its two rarest classes. Which average would have exposed that?

A. Micro, which pools all the decisions together before computing the metric once.

B. Macro, which counts every class equally whatever its size.

C. Weighted, provided the support column is printed alongside the score itself.

D. None of them, because only a full confusion matrix could have revealed it.

15 3. The data is the job

You are about to fit XGBoost on a table where income runs to millions and age to tens. Do you standardise the features?

A. Yes, because gradient boosting is fitted by gradient descent over the feature values.

B. Yes, because the L2 penalty on the leaf values is applied on the feature's own scale.

C. Only income, so that the two features end up with comparable numeric ranges.

D. No, because a split does not change when a column is rescaled.

16 3. The data is the job

Applying log1p to income changes which differences the model treats as equal. What becomes equal?

A. An increase of ten thousand rupees, wherever on the scale it starts from.

B. A doubling of income, wherever on the scale it starts.

C. The gap between the smallest and the largest value present in the column.

D. Nothing changes; the log only makes the histogram look more symmetric.

17 3. The data is the job

Production sends a city that was not present during training, and the encoder was built with handle_unknown='ignore'. What does the model receive?

A. The most frequent city that was seen during the training run.

B. An error, which the serving layer has to catch and translate itself.

C. An all-zero block across that column group.

D. A new column appended to the end of the feature matrix for that row.

18 3. The data is the job

Why is count encoding incapable of leaking the target?

A. It uses no target information at all.

B. It is computed after the split, so the folds never see each other's rows.

C. It is smoothed towards the global mean, which removes the leak entirely.

D. It produces a single column, and one column cannot carry the answer.

19 3. The data is the job

You encode hour of day as a sine and cosine pair and feed it to LightGBM. What is the effect?

A. Strongly helpful, because the model now knows that 23:00 and 00:00 are adjacent.

B. Neutral, because a tree simply ignores any feature it cannot split on cleanly.

C. Harmful, because tree models cannot handle negative feature values at all.

D. Slightly harmful, since a tree could split hour at 22 and again at 5.

20 3. The data is the job

Why can a held-out set not detect that your sample misrepresents the population you will predict on?

A. It is too small to detect a difference of that size with any reliability.

B. It is drawn from the same sample.

C. It is drawn without stratification, which makes the comparison invalid.

D. It can detect it, provided the split is grouped by the sampling entity.

21 3. The data is the job

Why must SMOTE be applied inside each cross-validation fold rather than before the split?

A. It is considerably slower on the full dataset than on each fold separately.
B. It changes the class balance, which makes stratified splitting invalid.
C. It synthesises rows from neighbours, so it learns from the data.
D. It needs each fold's own random seed in order to produce reproducible rows.

22 4. Generalisation, and the discipline of held-out data

At a 2 per cent positive rate with 1,000 rows and five folds, what does dropping stratification risk?

A. A fold with no positives, where some metrics are undefined.
B. Folds of unequal size, which biases the averaged score downward by a small amount.
C. Training sets that overlap, which correlates the fold scores with one another.
D. A fold whose features happen to be scaled differently from the other four.

23 4. Generalisation, and the discipline of held-out data

You label training rows zero and test rows one, and a classifier separates them at 0.82 AUC. What does that tell you?

A. The model is overfitting, because it is able to identify its own training rows.
B. The split was random, and 0.82 is about what a random split normally produces.
C. The two sets differ systematically, so the test score is not comparable.
D. The test set is too small to be reliably distinguished from the training set.

24 4. Generalisation, and the discipline of held-out data

When is leave-one-out cross-validation usually the wrong choice?

A. On small datasets, where a training set of eighty per cent is already too small.

B. Almost always: it costs n fits and its estimate has high variance.

C. When the classes are imbalanced, because each fold contains only one class.

D. When the model is a tree, because a single held-out row cannot be scored.

25 4. Generalisation, and the discipline of held-out data

Repeated cross-validation shows a spread across repeats comparable to the gap between two models. What follows?

A. The higher-scoring model should be chosen, since it won on average across the repeats.

B. More repeats will shrink the spread until the difference between them is clear.

C. The spread can be reported as a confidence interval on the difference in scores.

D. You do not have evidence to prefer either model.

26 4. Generalisation, and the discipline of held-out data

Fifty refits on different samples produce nearly identical predictions that all miss the truth in the same places. Which problem is it?

A. High bias, and the fix is capacity or better features.

B. High variance, and the fix is more data or stronger regularisation of the model.

C. Irreducible noise, which by definition no model or feature can ever reduce.

D. Neither one; identical predictions simply mean that training has converged.

27 4. Generalisation, and the discipline of held-out data

Why sample a regularisation strength with loguniform between 0.001 and 100 rather than uniformly?

A. A uniform draw cannot produce values below one at all in that range.

B. A uniform draw puts almost every point above 0.1.

C. Log sampling is required by the interface of RandomizedSearchCV in scikit-learn.

D. Uniform sampling correlates successive draws, which biases the whole search.

28 4. Generalisation, and the discipline of held-out data

You need to detect a one-point improvement in accuracy near 85 per cent. Roughly how large must the validation set be?

A. On the order of three hundred rows, which is enough for a stable estimate.

B. On the order of one thousand rows, since that is the usual rule of thumb.

C. Size does not matter provided the comparison is repeated enough times.

D. On the order of ten thousand rows.

29 5. Measuring a model honestly

Which sentence separates precision from recall correctly?

A. Precision divides by what was true; recall divides by what the model said.

B. Precision counts the true negatives, and recall counts the false negatives.

C. Precision divides by what the model said; recall by what was true.

D. Precision is measured on validation data and recall is measured on test data.

30 5. Measuring a model honestly

A fraud team can review 200 cases a day. Which metric describes what they actually experience?

A. Precision among the top 200 cases by score.
B. ROC AUC computed across the whole range of possible score thresholds.
C. F1 measured at the library's default decision threshold of 0.5.
D. Accuracy over every transaction that was scored during that day.

31 5. Measuring a model honestly

Why is F1 usually the wrong summary for a business problem?

A. It is undefined whenever recall for one of the classes comes out at zero.
B. It declares precision and recall to be equally important.
C. It cannot be computed at all without well-calibrated predicted probabilities.
D. It is dominated by the majority class whenever the data is heavily imbalanced.

32 5. Measuring a model honestly

Which of these produces well-calibrated probabilities without any post-processing?

A. Naive Bayes, because it applies Bayes' rule to the evidence directly.
B. A random forest, because it averages the votes of five hundred trees.
C. A support vector machine, because it maximises the margin between classes.
D. Logistic regression, by construction.

33 5. Measuring a model honestly

Residuals plotted against predictions fan outward as the prediction grows. What does that suggest?

A. The model is underfitting and needs more capacity or a longer training run.

B. An outlier is present and ought to be removed from the training set.

C. Fitting the log of the target is likely to help.

D. The metric should be changed from RMSE to R squared before reporting.

34 5. Measuring a model honestly

MAE is 4,200 and RMSE is 12,000 on the same set of predictions. What does the gap tell you?

A. A few of the errors are very large.

B. The errors are of fairly consistent size across all of the evaluated rows.

C. The model is biased upward on the majority of the predictions it makes.

D. RMSE must have been computed on a different subset of rows from MAE.

35 5. Measuring a model honestly

Why does slice-based error analysis need a minimum group size?

A. Small groups cannot be stratified, which leaves their metrics mathematically undefined.

B. With enough slices, several will look bad purely by chance.

C. Metrics are only valid on groups larger than the number of features in the model.

D. Small groups are usually duplicates of larger groups elsewhere in the same table.

36 6. Trees, ensembles, and the algorithms that win on tables

The true rule is that income exceeds three times expenses. What does a decision tree do with it?

A. Learns it exactly, because trees capture interactions between features for free.

B. Approximates it with a staircase of axis-aligned splits.

C. Refuses to split, because neither column on its own reduces impurity at all.

D. Learns it exactly given enough depth, because deep trees are universal approximators.

37 6. Trees, ensembles, and the algorithms that win on tables

What makes out-of-bag error possible in a random forest?

A. Each tree is fitted on a different random subset of the available features.

B. The forest holds out a validation fold before it fits any of its trees.

C. Trees are grown to full depth, so every training row ends up in its own leaf.

D. About 37 per cent of rows are left out of each bootstrap sample.

38 6. Trees, ensembles, and the algorithms that win on tables

You halve a boosted model's learning rate. What else should change?

A. The number of trees should roughly double.

B. The maximum depth of each tree should roughly double to compensate.

C. The row subsampling fraction should be halved to match the new rate.

D. Nothing at all; the learning rate is independent of the number of trees.

39 6. Trees, ensembles, and the algorithms that win on tables

You set subsample to 0.8 in LightGBM and nothing changes. Why?

A. LightGBM subsamples features rather than rows, so the parameter is the wrong one.

B. Subsampling only takes effect once early stopping has been triggered by the run.

C. subsample_freq is still at its default of zero.

D. The value has to be supplied as a percentage rather than as a fraction of one.

40 6. Trees, ensembles, and the algorithms that win on tables

Why must features be standardised before fitting an SVM with an RBF kernel?

A. The margin is measured in the units of whichever feature comes first.

B. The kernel is a function of Euclidean distance.

C. The solver requires every input to be bounded between zero and one.

D. C and gamma are only defined for inputs that have been standardised.

41 6. Trees, ensembles, and the algorithms that win on tables

A SHAP value of +0.19 is reported for late_payments equal to three. What does that license you to say?

A. Reducing this person's late payments by three would lower their real risk of default.

B. Late payments are the strongest driver of default risk in this population overall.

C. The model would have approved this application if the value had been lower.

D. This model's output was 0.19 higher because of that value.

42 6. Trees, ensembles, and the algorithms that win on tables

A partial dependence curve for income is flat, and the ICE lines show half the rows sloping up and half sloping down. What have you learned?

A. There is an interaction, and the average is misleading.

B. The feature has no effect, and the ICE lines are simply sampling noise around zero.

C. The model is unstable and should be refitted with a different random seed.

D. The average is correct, and the individual ICE lines should be smoothed first.

43 7. Learning without labels, and learning a representation

You standardise every column before running k-means. What have you thereby asserted?

A. That the clusters in the data are spherical and of roughly similar size.
B. That the data contains at least as many clusters as it has columns.
C. That every feature should count equally in the distance.
D. Nothing at all, because standardising is a neutral preprocessing step.

44 7. Learning without labels, and learning a representation

DBSCAN merges everything in a sparse region while fragmenting a dense one. What is the cause?

A. One eps has to serve the whole dataset.
B. min_samples was set higher than the size of the smallest genuine cluster.
C. The features were standardised, which equalised the densities artificially.
D. DBSCAN requires the number of clusters to be supplied before it can run.

45 7. Learning without labels, and learning a representation

What is the most common mistake made when applying PCA?

A. Keeping too many components, which reintroduces the noise you meant to remove.
B. Not standardising, so the largest-unit feature dominates.
C. Applying it to categorical columns that have already been one-hot encoded.
D. Fitting it after the split rather than on the whole dataset before splitting.

46 7. Learning without labels, and learning a representation

Which of these is true of UMAP and not of t-SNE?

A. It preserves the distances between clusters accurately enough to quote.
B. It produces the same layout every run regardless of the random seed.
C. It can transform new points into an existing embedding.
D. It requires no dimensionality reduction with PCA beforehand on wide data.

47 7. Learning without labels, and learning a representation

In implicit feedback data, what does an unobserved user-item cell mean?

A. A dislike, which is why unobserved cells are conventionally treated as zeros.

B. A neutral rating, sitting halfway between a like and a dislike on the scale.

C. An error in the event log, since every interaction ought to have been recorded.

D. Usually nothing, since the user may never have seen the item.

48 7. Learning without labels, and learning a representation

Why does pure embedding search fail on an order number?

A. An identifier has no meaning for the vectors to represent.

B. Identifiers are too short to meet the embedding model's minimum input length.

C. Embeddings are normalised to unit length, which removes numeric information.

D. The index returns approximate neighbours, so an exact match may be missed.

49 7. Learning without labels, and learning a representation

You label the most uncertain rows each round. Why keep a separate randomly labelled set as well?

A. Uncertainty sampling tends to produce labels of noticeably lower quality.

B. The uncertain rows are not a random sample of the population.

C. The model has already seen the uncertain rows while scoring the unlabelled pool.

D. A random set is needed in order to compute the uncertainty threshold itself.

50 8. Neural networks, built from what you already know

A network with one hidden layer can approximate any continuous function on a bounded region. What does that result not say?

A. That the approximation is exact rather than merely arbitrarily close to it.

B. That the function being approximated must be bounded on the region concerned.

C. That gradient descent will find it, or that it is affordable.

D. That deeper networks are able to approximate the same class of functions.

51 8. Neural networks, built from what you already know

Why does He initialisation use twice the variance of Xavier for a ReLU network?

A. ReLU zeroes about half the inputs, which halves the variance.

B. ReLU doubles the magnitude of every positive activation that passes through it.

C. ReLU is unbounded, so the initial weights must start larger in order to compete.

D. He initialisation is meant for convolutional layers and Xavier for dense ones.

52 8. Neural networks, built from what you already know

What is the main practical cost of batch normalisation?

A. It cannot be combined with a residual connection anywhere in the network.

B. It behaves differently in training and at inference.

C. It requires the learning rate to be lowered substantially to remain stable.

D. It removes the need for an activation function between consecutive layers.

53 8. Neural networks, built from what you already know

What does adding the block's input to its output do on the backward pass?

A. It scales the gradient by the number of layers inside the block.

B. It prevents the gradient from exploding by clipping its norm at one.

C. It removes the need for any normalisation layer inside that block.

D. It passes the gradient back unmultiplied.

54 8. Neural networks, built from what you already know

Why does training a network need far more memory than inference?

A. Every forward activation is kept until the backward pass uses it.

B. The optimiser stores a full copy of the training dataset for each epoch.

C. Gradients are stored at a higher numerical precision than the weights are.

D. Dropout keeps a second copy of the whole network for the averaging step.

55 8. Neural networks, built from what you already know

A Conv2d layer has 64 filters of 3 by 3 over 3 input channels. How many weights, ignoring the biases?

A. 576, which is 64 filters times the 3 by 3 window in a single channel.

B. 150,528, which is one weight for each pixel of a 224 by 224 colour image.

C. 1,728, which is 64 times 3 times 3 times 3.

D. 192, which is 64 filters times the 3 input channels being read.

56 8. Neural networks, built from what you already know

What grows with the square of the sequence length in a transformer?

A. The parameter count, because each position needs its own set of weights.

B. Attention itself, since every position attends to every other.

C. The feed-forward layers, which are applied independently at each position.

D. The positional encoding, which stores one vector for each pair of positions.

57 9. Into the world, and keeping it honest

Your model writes a score to a table, and you change the model. Which technical debt does that expose?

A. Entanglement, because changing one feature moves every coefficient in the model.

B. Pipeline jungles, because the table is written by a long untraceable chain of jobs.

C. Data dependencies, because the upstream schema may have changed without notice.

D. Undeclared consumers reading that table break silently.

58 9. Into the world, and keeping it honest

The stated freshness requirement is 'within an hour'. What follows?

A. An online service is required, because an hour is a latency budget to be met.

B. An hourly batch job is far cheaper to build and to operate.

C. Streaming is required, because batch jobs can only be scheduled overnight.

D. The choice does not matter, since both approaches produce the same predictions.

59 9. Into the world, and keeping it honest

What does point-in-time correctness in a feature store prevent?

A. Two teams computing the same feature in two different programming languages.

B. The serving path reading from a different database than the training job used.

C. Training on a feature's current value instead of its value then.

D. An unseen category arriving at inference and causing the transform to raise.

60 9. Into the world, and keeping it honest

Which monitoring layer moves first when an upstream table breaks?

A. Input distributions, which need no labels at all.

B. Actual performance, as soon as the outcome labels begin to arrive.

C. Prediction distributions, which lag the input distributions by about a day.

D. System health, which reflects the serving container's memory and error rate.

61 9. Into the world, and keeping it honest

What is the status of the population stability index thresholds of 0.1 and 0.25?

A. Derived from the chi-squared distribution at the ninety-five per cent level.

B. Convention, and useful because everyone uses the same numbers.

C. Set by regulation in most jurisdictions that supervise credit scoring.

D. Chosen so that a shift of exactly one standard deviation crosses the upper one.

62 9. Into the world, and keeping it honest

You prune 90 per cent of a network's weights to zero and see no speedup at all. Why?

A. Ninety per cent is below the sparsity threshold at which speedups begin to appear.

B. The pruning was applied after quantisation when it should have come before it.

C. Scattered zeros are still processed as a dense matrix.

D. The framework re-densifies the pruned weights when the model is saved to disk.

63 9. Into the world, and keeping it honest

Which item belongs on a model card and is most often left off?

A. The uses the model is explicitly not for.

B. The exact hyperparameter values chosen by the search that produced it.

C. The training duration and the hardware the training run was executed on.

D. The library versions that were used to fit and serialise the model artefact.

Answers

Each entry gives the correct option, then what each wrong one reveals about how somebody was thinking. The second part is worth more than the first: a wrong answer here is a named misreading, not a mistake.

Lesson checks

1. What a machine actually learns — A

B — Learning sounds like discovery, so it feels as though a large enough dataset should let the model outgrow the shape it was given.

C — Training is described as lowering the loss, so it is natural to assume the loss keeps falling toward zero rather than bottoming out at whatever the best line can manage.

D — Dirty data really is the culprit in many bad fits, so blaming the data is a habit that usually pays off — here the data is fine and the model is the problem.

2. With answers, and without — A

B — The labels contain no false positives, which feels like the definition of a trustworthy label — the damage is in what the label omits, not in what it asserts.

C — Once bias is named it invites a clean rejection, but a model of the detection process can still rank a review queue usefully as long as everyone knows that is what it is.

D — No labels sounds like no label bias, but the same features, the same sampled population and the same human interpretation of the clusters carry the bias straight through.

3. What a row is, and why that decision is the whole project — A

B — Blames the metric for a data-splitting fault; the same contamination would inflate accuracy, F1 and every other score equally.

C — Describes ordinary overfitting, which would show up as a train-test gap — here the test set itself is compromised, so the gap never appears.

D — Class imbalance distorts accuracy, but AUC is comparatively robust to base rate and would not be lifted to 0.94 by imbalance alone.

4. The loss is where you state what a mistake costs — C

A — Inverts the effect: squared error makes rare extreme values more influential, not less, so they cannot be quietly ignored.

B — Would produce a flat prediction, but does not explain the specific upward bias, which comes from how squared error weights the long trips.

D — Treats a distributional assumption as a hard requirement; MSE converges fine on skewed targets, it just fits the mean of that skewed distribution.

5. The number your model has to beat before anyone should care — D

A — Quantifies uncertainty in the number, which is useful, but tells you nothing about whether 210 is good or bad in the first place.

B — Improves the reliability of the estimate but leaves the same interpretive vacuum — a more trustworthy 210 is still meaningless without a reference point.

C — Applies a distributional worry that does not bear on interpretation; RMSE is well defined regardless and the question is what to compare it against.

6. Why a model that predicts well cannot tell you what to change — A

B — Names a real hazard, but leakage would invalidate the prediction itself; here the prediction can be perfectly valid and the intervention still fail.

C — Raises an important separate concern that does not address whether the proposed intervention would work at all.

D — Treats it as a sampling-uncertainty problem; the association can be large, stable and significant and the causal conclusion still be wrong.

7. The problems where the right answer is a rule — C

A — Raises a governance issue that may or may not apply, and would not change the analysis if the data were freely usable.

B — Plausible in some firms, but a large logistics operation has millions of parcels — the objection fails as soon as data volume is adequate, whereas the real problem does not.

D — Confuses an implementation detail with a fit-for-purpose judgement; categorical encoding is routine and covered later in this course.

8. The whole loop, once, before we slow down — D

A — Ordering is a real convenience of pipelines, but applying the same steps manually in the same order would still produce different numbers, so ordering is not the mechanism.

B — Describes an engineering benefit rather than a statistical one; the numbers would differ even in typo-free manual code.

C — True and important for module 9, but this benefit is about production drift rather than about the validation score being honest.

9. A working setup that costs nothing — B

A — Confuses environment isolation with filesystem isolation; a venv separates installed packages, not data.

C — Invents a performance rationale; import time is not a meaningful factor and is not why the practice exists.

D — Describes a permissions annoyance that has workarounds, and would not explain why experienced practitioners use a venv even with full rights.

10. Fitting a line to five points, by hand — B

A — Correctly recalls that MSE is outlier-sensitive, but that concerns rows in the training data, not a prediction requested outside its range.

C — Fixes on the intercept, which contributes the same fixed 6.15 at every size and is not what makes the villa prediction unreliable.

D — Invents a sample-size rule; 40 rows give a usable but uncertain fit, and no row count would make extrapolation to 400 safe.

11. There is an exact answer, and we usually refuse it — C

A — A plausible cause of exploding parameters in an iterative fit, but LinearRegression does not iterate, and the near-exact cancellation points elsewhere.

B — Small per-level samples make coefficients noisy and unstable, but would not produce two huge values that cancel almost exactly.

D — Target scale shifts coefficient magnitude proportionally, which cannot turn a 20 to 200 range into six-figure coefficients of opposite sign.

12. Gradient descent, or walking downhill in fog — A

B — A terrible loss reads as insufficient capacity, but underfitting produces a loss that settles high and stays flat — it never explodes toward infinity.

C — Local minima are the most repeated explanation for training failures in the whole field, which makes them the first thing people reach for — but being stuck makes a loss go flat, not infinite.

D — Too little data explains a model that fails to generalise, which is a different symptom from a loss that diverges within thirty steps of starting.

13. Batches, epochs, and the two knobs that decide everything — C

A — Confuses standardising with clipping; subtracting the mean and dividing by the standard deviation preserves outliers, it does not remove them.

B — Confuses scaling with dimensionality reduction; the same number of columns goes in and comes out.

D — Saturation is real and does slow learning, but the optimiser still receives usable gradients — the dominant effect is the mismatch in gradient magnitudes across weights.

14. Diagnosing a model from the shape of its loss curve — B

A — Leakage would inflate validation performance, but it usually shows as validation tracking training almost exactly rather than sitting systematically below it.

C — Underfitting produces both losses high and close together, with no reason for validation to be the lower of the two.

D — Averaging over fewer rows increases variance but does not bias the mean downward, so it cannot produce a consistent gap in one direction.

15. Logistic regression, built from odds rather than assumed — A

B — Correctly notes that L2 shrinks coefficients, but this concerns the size of the estimate, not the manager's conversion from log-odds to probability.

C — Sounds appropriately cautious but misstates the model: the coefficient does apply per row, just on the log-odds scale rather than the probability scale.

D — Confuses the intercept's role with the coefficient's; the odds ratio holds while other features are held fixed at any values, not only at zero.

16. What a coefficient says, and the four ways it lies — C

A — Treats adding controls as monotonically improving, which is not true — controlling for a variable downstream of the effect can introduce bias rather than remove it.

B — Assumes the smaller coefficient must be the distorted one, which is not determinable from the drop alone.

D — Uses predictive fit to adjudicate an interpretive question; two models can predict equally well while attributing the effect completely differently.

17. Regularisation: paying for large coefficients — B

A — Correctly notes collinearity inflates coefficient uncertainty, but ridge was specifically designed to stabilise exactly this case, which is why it shares the weight.

C — Assumes the dropped features add predictive value, but with near-identical readings the retained one carries essentially the same information, so accuracy barely moves.

D — Both objectives are convex, but they are different objectives with different minima, so convexity does not imply they agree.

18. Making a straight-line model bend — A

B — Binning adds flexibility within one feature, but the deficiency described is a between-feature dependence that binning alone cannot express.

C — Degree 3 does include the interaction, but it also adds many other terms and sharp edge behaviour, so it is a costly way to get the one term that is needed.

D — Transforms the shape of one feature's effect, which does not let the effect differ between enterprise and non-enterprise customers.

19. More than two classes, and what changes — C

A — Imbalance would skew which topic is chosen, but not create a structural inability to assign several topics at once.

B — Hierarchical softmax addresses computational cost at thousands of classes, not the number of labels a model can assign per row.

D — Thresholding softmax outputs can surface secondary topics, but the normalisation still caps total confidence at 1, so a three-topic article cannot be confident about all three.

20. Features beat models — A

B — A clean split feels like proof of generalisation, but a split can only detect memorised rows — it is blind to a column whose value is filled in after the outcome.

C — High importance concentrated in one feature is a classic overfitting signature, but here training and validation agree closely, so variance is not what is happening.

D — Importance scores really are shaky and distrusting them sounds sophisticated, but the timing question about that column stands whether or not it tops the ranking.

21. Scaling, and the transforms that change what a model can see — C

A — Assumes a bug, but a correctly fitted scaler on tree input also produces no accuracy change, so the null result is expected rather than diagnostic.

B — Would explain no change for a scale-sensitive model, but the result holds for trees regardless of the original distribution.

D — Invents an internal preprocessing step; XGBoost does no such thing, and it does not need to because splits are scale-invariant.

22. Turning categories into numbers without lying about them — B

A — Comes close, but noise alone would not lift cross-validated AUC — the fold structure should have caught it, which points to the encoding rather than the feature.

C — A 16-point gap is far outside plausible cross-validation noise on any reasonably sized set.

D — AUC is comparatively robust to base rate, and an encoding does not change the class distribution at all.

23. When a column has thousands of levels — B

A — Base-rate shift affects calibration and can lower some metrics, but AUC is rank-based and would not collapse by 32 points from a base-rate change alone.

C — Names a real high-cardinality hazard, but collisions would degrade the training score too, not only the production one.

D — Training-serving skew is a genuine and common failure, but here the drop is fully explained by the identifier having no meaning for unseen customers even with identical encoding.

24. Missing values, and the information in the gap — B

A — Preserving the centre is a property of median imputation, but the concern is the information carried by which patients are missing, not the shape of the imputed column.

C — Variance shrinkage is a real side effect of single imputation, but it matters far less here than the erasure of a strong risk signal.

D — Applies a threshold rule that does not exist; a 40%-missing column with an informative missingness pattern is often among the most useful in the table.

25. Outliers: error, extreme, or the thing you are looking for — B

A — A 0.4% reduction is far too small to matter for variance, and would not cause a collapse in one specific class's recall.

C — Order of scaling and thresholding changes which rows are flagged only if the scaler is non-monotonic; standard scaling preserves the same set of extreme rows.

D — Class balance did shift slightly, but a 0.4% removal cannot move the base rate enough to explain a collapse, and rebalancing would not restore deleted examples.

26. Dates, durations, and the trouble with midnight — C

A — Window length affects how much seasonal signal is captured, but it would show up equally in validation rather than producing a five-fold gap.

B — Missing data would add noise to the rolling feature, but not create a systematic advantage during validation that disappears afterwards.

D — A real and related error, but the described defect is inside the feature itself and would inflate the score even under a correct time-based split.

27. Text as columns: counting words before you embed them — B

A — Inverts the mechanism: inverse document frequency amplifies rare terms rather than suppressing them.

C — TfidfVectorizer applies L2 normalisation per document by default, so length is largely accounted for already.

D — Case can carry emphasis, but losing it is a mild effect compared with deleting the words that reverse a sentence's meaning.

28. Leakage: a catalogue of the ways the answer gets in — B

A — Correlation with the target is what a good feature has; it is the timing of the value, not its predictiveness, that constitutes leakage.

C — A preprocessing requirement unrelated to whether the value is knowable at prediction time.

D — Seasonality is a genuine modelling concern, but the defect here would remain under any split because the recorded value is from the wrong point in time.

29. Fit on train, transform everywhere: the contract that keeps you honest — B

A — Resampling a test set would destroy its representativeness; the held-out data must reflect the real base rate.

C — A real limitation of SMOTE on mixed data, but it would degrade both the cross-validated and production scores rather than creating a gap between them.

D — Recall alone is incomplete, but the discrepancy between validation and production is caused by the resampling order, not by the metric chosen.

30. Where the rows came from, and who is missing — C

A — Sample size affects precision of the estimate, but the problem persists at any size because the missing population is absent from both training and validation.

B — Base-rate mismatch is a genuine consequence, but recalibration would not repair a model that has never seen the rejected population's behaviour.

D — Approval-related columns can indeed leak, but the defect here is which rows exist at all, and it remains even if every leaked column is removed.

31. When one class is 0.3% of the data — C

A — Calibration does degrade after resampling and is worth checking, but miscalibration would not by itself inflate a cross-validated F1 score.

B — Averaging choice matters a great deal for reported numbers, but it would affect both the before and after figures equally.

D — Tuning k changes how synthetic points are placed, but no setting of k produces a doubling of F1 through legitimate means.

32. Train, validation, test, and the discipline of not looking — A

B — Small samples really do produce unreliable scores, but sampling noise is symmetric — it would sometimes flatter and sometimes penalise, not deliver a consistent thirteen-point gift.

C — Drift is real and common, and it is the first explanation most teams reach for — but this gap was built into the split before a single live prediction was made.

D — Cross-validation is the more rigorous-sounding option, yet folds built from the same random row assignment repeat the identical mistake five times and average it.

33. Overfitting, underfitting, and how to tell — A

B — "Overfitting" is used colloquially as a synonym for "bad model", so any disappointing score pulls the word toward it regardless of what the two numbers say.

C — The tiny gap genuinely is healthy, and it is easy to slide from "not overfitting" to "nothing wrong" without noticing that both numbers are low.

D — Deferring to the test set sounds like the disciplined answer, but the test set answers a different question and spending it on a diagnosis destroys the one honest estimate they have.

34. Cross-validation: using every row twice, legally — A

B — More folds reduce variance slightly but do nothing about the selection bias introduced by taking a maximum over many candidates.

C — Refitting on all training data usually helps rather than hurts, and it is not the source of the upward bias in the reported score.

D — Temporal shift is a real separate hazard, but the bias described here would exist even if production data were drawn from exactly the same distribution.

35. Splits that respect the structure of the data — D

A — Random sampling would reintroduce future information into training, which is the failure TimeSeriesSplit exists to prevent.

B — Rolling versus expanding is a real empirical choice, but it does not address information that would not have been available at prediction time.

C — Correctly notes label delay, but the test set is allowed to use labels resolved after the fact — the problem lies on the training side of the boundary.

36. Bias and variance, with numbers attached — B

A — Deeper trees do capture more interactions, but the near-zero training error shows the model already fits the training data, so capacity is not the constraint.

C — Averaging reduces variance substantially but does not make depth irrelevant; individual tree depth still changes the ensemble's behaviour.

D — Random forests are usually grown deep by design precisely because averaging controls the resulting variance, so 'underfit by default' inverts the situation.

37. Would more data help? The curve that answers it — C

A — Reads a flat curve as unfinished; convergence at a stable level is precisely the signal that additional rows of the same kind will not move the score.

B — Inverts the diagnosis: overfitting shows as a persistent gap between the curves, and the curves here have converged.

D — More folds tighten the estimate slightly but would not change a clearly converged pattern, and the decision does not hinge on that precision.

38. Searching hyperparameters without fooling yourself — C

A — The optimum sitting at the boundary suggests looking further in that direction, but jumping to zero penalty skips the region between 0 and 0.01 where the answer may lie.

B — Density within the same range does not help when the best value is at the edge; the range itself is the problem.

D — Noise can shuffle nearby candidates, but a systematic result at the boundary usually indicates the range is wrong rather than that the estimate is unstable.

39. A validation set you have used two hundred times — B

A — Wider search does raise the chance of finding a genuinely good configuration, but it raises the selection bias in the reported score at the same time, so the identical number does not favour B.

C — Identical set and metric make the numbers comparable in form, but the number of selections made against that set changes what each number means.

D — Training size affects model quality, but the bias described here is governed by the validation set size and the number of comparisons, both of which are given.

40. How much data do I need? — B

A — Counts total rows, which is the framing that makes this situation look comfortable; what constrains learning here is the number of positive cases.

C — Metric choice does matter at this prevalence, but a better metric reveals performance rather than creating the events needed to learn from.

D — Undersampling to match would leave 400 rows total and discard nearly all the information about what normal looks like.

41. Double descent, and the picture that stopped being complete — C

A — Overcorrects: parameter count still matters, particularly at small data sizes where the classical picture holds firmly.

B — Dropout helps but does not make overfitting impossible; a large network on very little data still overfits badly.

D — Overstates the finding: the trade-off remains accurate for trees, linear models and boosting, which is most of what this course covers.

42. When accuracy lies — A

B — More data improves most things, and recall feels like a fixed property the model possesses rather than a dial the team is already choosing every time it runs.

C — Rebalancing is the standard first move against imbalance and it does raise recall — largely by shifting the effective threshold, while quietly ruining the calibration of the output probabilities.

D — Changing the metric feels like changing the priority, but F1 weighs precision and recall equally, which is precisely the wrong weighting for a clinical triage system.

43. The four numbers, worked through — A

B — Correctly notes that sensitivity and specificity are population-independent, but the probability a positive is correct is not — it depends on prevalence.

C — Reverses the arithmetic: fewer true cases means the same false positive rate produces a larger share of the positive results.

D — Treats the calculation as under-determined when sensitivity, specificity and prevalence are sufficient to compute it exactly.

44. The threshold is a business decision, not a default — B

A — Calibration makes the probabilities meaningful but says nothing about where to cut them; 0.5 is optimal only when the two errors cost the same.

C — Inverts the direction: raising the threshold produces fewer positive predictions and therefore more of the expensive false negatives.

D — Invents a logarithmic adjustment; the relationship is a simple ratio and produces a far more extreme threshold than a modest shift.

45. ROC and precision-recall: every threshold at once — C

A — Both do summarise the same ranking, but they weight regions of it differently, so identical AUC with different average precision is entirely consistent.

B — Neither metric measures calibration; both depend only on the ordering of scores.

D — Both metrics are threshold-free summaries over the whole score range, so no threshold is involved in either.

46. Does 0.7 mean seventy per cent? — B

A — Correctly notes the transformation is monotonic, but the goal was never to change AUC — it was to fix the probabilities, and the problem with the fix lies elsewhere.

C — Working on the log-odds scale is a sensible instinct, but a single fixed multiplier is still an unfitted guess at a correction that should be estimated from data.

D — Vote fractions are exactly the kind of score that post-hoc calibration handles well; forests calibrate readily.

47. Metrics for a number, and what each one hides — C

A — The zero problem is real and does distort MAPE, but excluding zero-demand days does not systematically push forecasts below actual demand.

B — Equal weighting is a genuine MAPE weakness, but it distorts which items the model attends to rather than the direction of the error.

D — MAPE uses absolute rather than squared errors, so the described mechanism is not present.

48. Is that improvement real? — C

A — Treats a 57% share as decisive, but a 170-130 split of 300 is close to what chance produces and does not reach conventional significance.

B — The agreements are correctly excluded: rows where both models behave the same carry no information about which is better.

D — Inverts the requirement: McNemar's test exists precisely for paired predictions on the same rows.

49. From a probability to a decision worth making — B

A — Inverts the dependency: a per-row threshold relies more heavily on calibrated probabilities, not less, because the cut point is now computed rather than tuned.

C — The rule raises the flag rate for large orders and lowers it for small ones; the total can move either way and higher volume is not the source of the gain.

D — Confuses a decision rule applied after prediction with a change to the training objective; the model is untouched by this.

50. Fairness as arithmetic: the definitions and what each one demands — B

A — Parity is one criterion among several and is not automatically the goal; the deeper flaw is about what the remaining features encode.

C — Accuracy may fall slightly, but a uniformly less accurate model is not what makes the fairness claim wrong.

D — Requirements vary and often concern collection for auditing rather than use in the model, so this misstates the legal position and misses the mechanism.

51. Why you cannot have all three — C

A — Treats the incompatibility as a data-quantity problem; the result is structural and holds for any imperfect classifier however much data it saw.

B — Post-processing can equalise one chosen criterion, but it redistributes errors rather than escaping the theorem, so the other criterion still breaks.

D — Parity is a third criterion with its own costs, and switching to it is itself the value judgement being avoided rather than a way around the result.

52. The average is hiding the failure — D

A — Small slices do represent real users, but reporting unconfirmed noise as findings wastes the team's credibility and its time.

B — Acts on the findings before establishing they are real, which risks degrading the model to fit twenty rows of noise.

C — Applies a rule mechanically; a genuinely failing small slice matters, and the correct response is confirmation rather than dismissal.

53. The classic algorithms, and when each is right — A

B — There is a real ladder of model power, so a flat result reads naturally as not having climbed high enough yet.

C — Both count as standard ML models, but a forest can express curves and interactions that a linear model structurally cannot — which is exactly why their agreement carries information.

D — More data reliably fixes variance problems, but here two model families with very different capacity land in the same place, which points at the inputs rather than the sample size.

54. How a tree chooses where to cut — A

B — Attributes a slope to a tree; trees produce piecewise-constant outputs and hold no notion of a trend to continue.

C — Out-of-range inputs route down the appropriate branch without difficulty; no error is raised.

D — Every input reaches exactly one leaf by following the splits, so the global mean is never what comes back.

55. Bagging: many unstable models, averaged — B

A — Individual trees do become stronger, but a forest tolerates strong individual trees well; the loss comes from what happens between them, not within one.

C — Bootstrapping is controlled by a separate parameter and remains active; the rows still differ between trees.

D — Training is slower per split, but the number of trees is fixed by n_estimators rather than by a time budget.

56. Boosting: each model fixes the last one's mistakes — B

A — Shallowness is why many trees are needed, but the observed degradation shows the model has passed its optimum rather than not reached it.

C — Subsampling is optional and fixed per tree when used; later trees do not see progressively less data.

D — Contributions are added with a fixed small weight rather than compounding, so predictions do not diverge — they overfit.

57. XGBoost, LightGBM, CatBoost: what actually differs — C

A — Histogram binning is a real difference and it slightly reduces resolution, which tends to regularise rather than to overfit.

B — Describes CatBoost's mechanism rather than LightGBM's, and LightGBM's categorical support is not enabled by identical numeric hyperparameters.

D — The premise states identical hyperparameters, so the learning rate is the same in both.

58. k-nearest neighbours, and why distance stops meaning anything — B

A — Scaling does matter for k-NN, but TF-IDF vectors are already L2-normalised by default, so the usual scale mismatch is absent.

C — scikit-learn's k-NN accepts sparse input directly; a representation issue would raise an error rather than produce chance-level predictions.

D — Inverts the relationship: k-NN works best in low dimensions and degrades as dimensionality rises.

59. Naive Bayes: wrong assumptions, useful results — B

A — Smoothing affects unseen words specifically and would not systematically push confidences toward 1 for messages it does classify.

C — Working in log space is a numerical safeguard rather than a source of error, and it does not create systematic over-confidence.

D — The prior shifts all scores in one direction but is a single multiplicative term, far too weak to produce 0.9999 on its own.

60. Support vector machines and the kernel trick — C

A — Changes the kernel family without addressing the settings that caused each point to dominate its own neighbourhood; a high-degree polynomial overfits similarly.

B — More data usually helps, but the configuration will continue to carve islands around points at any dataset size.

D — Adds a post-hoc calibration step that changes the reported probabilities, not the shape of the fitted boundary.

61. Feature importance, and the ways it misleads — C

A — Reads the number at face value; dropping both would likely cause a large score drop, which is exactly what the individual permutations failed to reveal.

B — Running on training data inflates importances rather than zeroing them, and the described pattern points to redundancy between columns.

D — A plumbing fault is one cause of zero importance, but it would show equally in impurity importance and is ruled out by the columns being correlated in a way the model exploits.

62. Explaining one prediction, and the shape of one feature — C

A — Instability would produce noisy, disordered lines rather than two coherent opposing groups.

B — Coarse grids smooth a curve, but they cannot average two opposite slopes into flatness across individual rows.

D — Both are computed on the same dataset by the same call, so a train-validation mismatch is not the source.

63. k-means, and what it assumes about your data — A

B — More clusters would subdivide the large group, but the lopsided result points at the distance metric rather than the number of centres.

C — Poor initialisation does produce degenerate solutions, but n_init=10 with k-means++ makes this an unlikely explanation for a systematic pattern.

D — Structureless data typically yields roughly balanced arbitrary partitions rather than one giant cluster and three tiny ones.

64. Choosing k, and why the elbow rarely helps — C

A — Uses more data but measures a different property: silhouette describes separation within one fitted partition, not whether that partition reappears on new data.

B — Larger subsamples overlap more and mechanically raise agreement, which is why 80% is the more informative test rather than a flaw to be tuned away.

D — The adjusted index handles label permutation correctly, so a low value reflects genuine disagreement rather than mismatched cluster names.

65. DBSCAN and hierarchical clustering: when shape and structure matter — D

A — Scaling matters for any distance method, but scaling coordinates uniformly would not resolve a genuine difference in density between regions.

B — Lowering min_samples helps the sparse region but simultaneously makes the dense regions merge more readily, so the trade-off remains.

C — Finding non-convex shapes is precisely DBSCAN's strength; the difficulty here is density variation, not shape.

66. PCA: fewer columns, and what you gave up — B

A — Raising the threshold retains more components and may help, but the mechanism — that variance and predictiveness are different things — persists at any threshold.

C — Including the target would leak it into the features; PCA is unsupervised by design.

D — Orthogonality removes correlation between components, but a model can still form interactions among them, so this is not what caused the loss.

67. t-SNE and UMAP, and how they mislead — D

A — Diffuseness reflects local density in the original space rather than a distinction between signal and noise.

B — Area in a t-SNE plot reflects the method's expansion of sparse regions, not the number of points in a group.

C — Between-cluster distances are not preserved, so relative separation in the plot carries no information about the original space.

68. What an embedding actually is — B

A — Inverts the property: cosine normalises out magnitude, which is exactly why it is preferred for documents of different lengths.

C — Text encoders handle raw text directly; there is no scaling step of the kind numeric features need.

D — On normalised vectors the two produce the same ranking, so switching metrics would not resolve a genuine difference in content.

69. Building a working semantic search with no GPU — A

B — Smaller chunks do sharpen matching, but a code with no semantic content remains poorly represented at any chunk size.

C — Missing normalisation distorts scores generally rather than failing specifically on identifier queries.

D — Approximate indexes do miss occasionally, but a systematic failure on identifiers points at the representation rather than the index's recall.

70. Recommendation: learning a vector per user and per item — C

A — Density does raise the cost substantially, but the fundamental problem is what the zeros assert rather than what they cost to process.

B — Mean imputation is a reasonable instinct for missing numeric data, but it still treats absence as an observation rather than as unknown.

D — The penalty applies to the learned vectors rather than to matrix entries, so filling cells does not affect it directly.

71. Anomaly detection, and the base rate that ruins it — B

A — Missing intrusions is a genuine concern, but it is not what makes the tool unusable in practice — the team never gets far enough to notice the misses.

C — The scenario supplies both the intrusion count and the catch rate, so evaluation is possible; the problem is the volume of false alerts.

D — Lowering the flag rate would reduce alerts, but the parameter here is already producing far too many, so the direction of the criticism is wrong.

72. Self-supervised learning: making labels out of the data itself — B

A — A small test set is imprecise, but imprecision produces scatter in both directions rather than a consistent underestimate.

C — Boundary cases can indeed be harder to annotate, but label noise would degrade training and testing alike rather than creating this specific gap.

D — Scoring unlabelled rows to select them does not expose their labels, so no target information leaks through the selection step.

73. Neural networks as the natural next step — A

B — Deeper networks genuinely do train more slowly, so "give it more epochs" is a plausible-sounding remedy for an equivalence that no amount of training can break.

C — Vanishing gradients are a real deep-network failure and the symptom looks similar, but here the result would hold even with a perfect optimiser and infinite precision.

D — Benefits from depth do grow with scale, so it is easy to read this as a demonstration that is simply too small rather than something mathematically impossible.

74. XOR, and why the bend is the whole idea — A

B — Depth cannot help while the composition remains linear, so no number of layers changes the outcome.

C — Depth does tend to help less on tabular data, but that empirical tendency is not why this particular network is exactly equivalent to a linear model.

D — Poor initialisation produces slow or unstable training rather than an exact match to a linear model's performance.

75. The bends: ReLU, and why sigmoid stopped being used inside networks — C

A — Sparse activations do have a mild regularising effect, but permanently dead units are lost capacity rather than a controlled constraint.

B — Inverts ReLU's behaviour: it is unbounded above and saturates only on the negative side, at exactly zero.

D — Negative inputs are ordinary and do not kill units by themselves; a unit dies only when its own pre-activation is negative for every example.

76. Backpropagation, traced through one tiny network — D

A — The loss would indeed differ, but the size of the loss does not determine whether the gradient reaches w1 through a ReLU that is inactive.

B — ReLU's derivative is precisely a function of the input's sign: 1 when positive and 0 when negative.

C — Sign flipping happens when a weight in the chain is negative, but ReLU zeroes the negative side rather than reflecting it.

77. Starting well, and staying well-behaved — A

B — Small-batch statistics do destabilise batch norm, but that would affect both depths rather than degrading specifically with added layers.

C — Misapplies the theorem, which concerns what a network can represent rather than imposing an upper limit on useful depth.

D — Overfitting produces lower training error with higher validation error; here the training error itself is worse.

78. Momentum, Adam, and what the optimiser is doing for you — D

A — AdamW retains momentum; removing it would change the optimiser's behaviour far more than the observed regularisation difference.

B — The coefficient's meaning differs between the two, but the fix is where the decay is applied rather than a change in its magnitude.

C — AdamW keeps the adaptive step sizes for the gradient term; only the decay is moved outside them.

79. Keeping a large network from memorising — B

A — Width does interact with the learning rate through initialisation scaling, but that is a training-stability point rather than the reason the suggestion misreads the curves.

C — Invents an architectural rule; width and depth are tuned independently and neither is required to accompany the other.

D — Normalisation helps wide and deep networks train, but its absence is not what makes added capacity the wrong response here.

80. Convolution: the same detector, everywhere in the image — B

A — Both are parallel operations on modern hardware; stacked layers are in fact sequential with respect to each other.

C — Initialisation schemes account for fan-in directly, so a 5x5 filter initialises perfectly well.

D — The premise states the receptive fields are the same, so a size advantage is not what distinguishes them.

81. Sequences: from recurrence to attention — C

A — Parameter counts do differ, but the model's own size does not scale with sequence length, and the reported problem appeared when sequences got long.

B — Layer norm's statistics are a few numbers per position and are trivial next to the attention matrix.

D — Batching works normally; it is the per-sequence attention matrix rather than the dataset that drives the memory cost.

82. Standing on somebody else's compute — D

A — Batch norm statistics on small batches are a real fine-tuning hazard worth handling, but they are secondary to a learning rate orders of magnitude too high.

B — Domain distance does reduce transfer, but the frozen-feature model outperformed the fine-tuned one using the same weights, so the features clearly carried value.

C — The frozen-feature approach works well at this scale, which is exactly why it is the recommended first step.

83. The model is about 5% of what you have to build — A

B — Leakage would change the scores, but nothing here indicates the new scores are wrong — the problem is that the change propagated without warning.

C — Recalibration would mask this instance, but it does not address the fact that unknown consumers depend on an output with no agreed contract.

D — Versioning enables recovery after the fact and does nothing to prevent the dependency problem that caused the breakage.

84. Batch, online, and the latency budget — C

A — Tree traversal time varies little between inputs, so it cannot produce a twenty-fold spread between the mean and the tail.

B — Tail latency usually has a different cause from the mean, so improving the mean rarely moves the p99 proportionally.

D — Geographic variation does add tail latency, but it is bounded and typically far smaller than the 800ms excess described here.

85. When training and serving quietly disagree — C

A — Floating-point differences between environments appear around the sixth or seventh decimal place, not the second.

B — Drift would change performance without making the two paths disagree on identical input rows.

D — A stale artefact would produce a difference, but it is ruled out cheaply by comparing version identifiers and would typically show a larger and more uniform gap.

86. Monitoring: what to watch, and in what order — D

A — A corrupted feature would normally register in the input distribution monitoring, which the scenario states is unchanged.

B — Concept drift typically degrades ranking quality, and AUC has held steady, which points away from the relationship itself having changed.

C — Skew would show as a difference between the two paths on identical rows, and it would usually disturb the input distributions being monitored.

87. When to retrain, and what breaks when you do — C

A — Overfitting would usually show as a worse offline score on held-out data, and the offline metric improved.

B — A higher learned base rate would shift scores upward, but the review volume is set by the threshold applied to those scores rather than by the model alone.

D — Skew is a real hazard but would normally degrade the offline-to-production relationship rather than appear as a clean tripling of volume with an improved offline score.

88. Being able to build the same model twice — D

A — AUC is a well-specified computation and implementations agree closely; a change of this size would not come from the metric.

B — Non-deterministic accumulation produces differences in the last decimal places, not a 1.6-point gap.

C — Ordering of seed initialisation is a genuine trap, but the scenario states every seed was set, and a split difference would typically produce a smaller and non-directional change.

89. AutoML: what it automates, and what it cannot — B

A — Cross-validation inside the search does not protect against a leaked column, which contaminates every fold equally.

C — Extending the search compounds the problem rather than examining it; the anomaly is the size of the gain, not insufficient search.

D — Copying hyperparameters would transfer the same exploitation of a leaked feature into the hand-built model without ever diagnosing it.

90. Making a model small enough to be affordable — A

B — A network performs one forward pass regardless of sparsity; there is no compensating repetition.

C — Quantisation would help independently, but the absence of a speedup here is explained by the sparsity pattern rather than by precision.

D — Fine-tuning recovers accuracy lost to pruning; it does not change how the hardware processes the weight matrices.

91. When the model changes the data it will be trained on — C

A — Overfitting would degrade performance on held-out data; here the held-out metric improves, because the held-out data comes from the same loop.

B — Changing the ranking metric measures the same biased interaction data differently rather than escaping the loop that produced it.

D — Seasonality could mask a real gain over a few months, but it would not explain a sustained collapse in catalogue coverage.

92. What you owe the people in the data — C

A — Treats the question as settled when regulators and jurisdictions differ on whether a trained model derived from those rows is itself in scope.

B — Backup handling is a real operational requirement, but it is a well-understood one rather than the genuinely unsettled question here.

D — Removing one person's rows has no measurable effect on calibration and is a technical rather than a legal concern.

Module test — 1. What learning is, and what it is not

1. A

Minimising squared error fits the mean and minimising absolute error fits the median. Squaring makes one three-hour trip cost as much as many small errors, so the fitted line is dragged towards it; absolute error charges an error of 100 exactly 100, so the long trips lose their special weight and the model settles on the typical trip instead.

2. C

A result far better than expected, arriving with no effort, is the signature of leakage. On a problem humans find hard, 0.99 is a bug report rather than a result, and the usual cause is a column such as cancellation_reason or account_closed_date that only exists once the outcome has happened. AUC is not inflated by imbalance, and sampling noise does not move a score that far.

3. D

A random row split scatters one customer's eleven loans across both sides. Eleven rows share almost every column, so the model can recognise the customer rather than learn the pattern, and the held-out loans belong to customers it has already seen thirty times. The score is inflated by however much of the signal is customer-specific, and the fix is to split by customer.

4. B

Prediction requires only that a correlation holds when the world is left alone. Intervention requires knowing what happens when you reach in and change something. The underlying problem causes both the call and the churn, so closing the phone line deletes the measurement while leaving the cause in place, and the frustrated customers now cannot even complain.

5. C

Machine learning earns its cost when a decision rule is real but nobody can write it down. Here it is already written down and exact. A model fitted to past decisions will get 99.4 per cent of them right, which is strictly worse than the if-statement that gets 100 per cent, and it adds a failure mode the regulation does not have.

6. A

The test set is a single-use measuring device. The moment a set is used to choose between options, it has been spent: the chosen model is partly good and partly a match for that set's particular noise. Not training on it is not enough. Iterating against it converts it into a second validation set, and the honest report says how many times it was consulted.

Module test — 2. Linear models and the machinery of fitting

1. C

NaN within a few steps is divergence. The step overshot the bottom, landed further up the far side where the slope is steeper, and each step made it worse until the numbers exceeded what a float can hold. More iterations make it worse faster. Dropping the rate by a factor of ten and looking again is the standard first move.

2. D

The normal equation exists because squared error against a linear function is a quadratic bowl that calculus can locate. Change the loss to log loss, or put a non-linearity in the model, and no such formula exists. Gradient descent is the general method and the closed form is a lucky special case, which is why one is taught and the other is used where it applies.

3. B

Ridge penalises the square of the coefficient, so its gradient is proportional to the coefficient and fades to nothing as the coefficient shrinks. Lasso penalises the absolute value, whose gradient is a constant regardless of size, so a coefficient of 0.001 receives the same push as one of 50. That constant push is strong enough to reach exactly zero and hold there.

4. A

Logistic regression models the log-odds as a straight line, so a coefficient adds a constant to the log-odds and multiplies the odds by a constant. The effect on probability is not constant: doubled odds move 0.10 to 0.18 and 0.80 to 0.89. Only the effect on log-odds is the same everywhere, which is why the odds ratio has to be translated before anyone quotes it.

5. C

The push on a weight is proportional to the feature value, so the income weight receives gradients tens of thousands of times larger than the age weight. One learning rate cannot serve both, and the loss surface is a long narrow ravine the optimiser zigzags down. Standardising turns the ravine into something closer to a bowl and the same optimiser converges in a fraction of the steps.

6. B

With the two columns entered separately, the model states one fixed effect of discount for everybody. The cross term lets the effect of discount depend on the value of is_corporate, which is what the situation describes. Finding such interactions without being told is precisely what a tree does and a linear model does not, and it is a large part of why boosting wins on tables.

Module test — 3. The data is the job

1. C

For a rare category the mean is computed from very few rows, and in the limit from the row itself, so the feature contains the answer for that row. The model learns to trust it completely, and on unseen data where the encoding comes from other rows that trust is unjustified. The fixes are cross-fitting the encoding out of fold and smoothing towards the base rate.

2. A

Missingness is rarely an accident. An applicant who left employer blank, a patient with no recorded reading, a form where the optional field is empty — each carries information about the person and the process. Impute without the flag and that information is destroyed permanently, and it is frequently more predictive than the value itself would have been.

3. D

Without the shift, the window ends on the current row, so today's sales are inside the feature used to predict today's sales. It produces a spectacular validation score and fails completely in production, where today's number does not exist yet. This single missing shift is the most common leakage bug in time series work.

4. B

Under five-fold cross-validation the training portion changes five times, so the mean and standard deviation must be recomputed five times from five different subsets. Scaling once beforehand means each fold's held-out rows contributed to the numbers used to transform the training rows. That is leakage, it happens silently in all five folds, and it moves the number you report.

5. A

A model minimising log loss at a 0.3 per cent base rate can reach a very low loss by predicting about 0.003 everywhere. It is not broken; it is correct about the base rate. What is broken is the decision rule that fires above 0.5. Moving the threshold and reporting average precision instead of accuracy is free and fixes most cases before any resampling is considered.

6. D

The test is whether this exact value will exist at prediction time for a row nobody has seen. It will not: every new applicant has an id the model has never met. A random split hides this, because the same applicants appear on both sides, and the model has learned per-applicant quirks that do not exist for anyone new. An identifier belongs in the index, not the feature matrix.

Module test — 4. Generalisation, and the discipline of held-out data

1. A

Overfitting is a gap between training and validation error, not a synonym for a disappointing score. One point of gap means nothing is being memorised. This is underfitting, and cutting capacity will make both numbers worse. The remedy is more capacity, better features, or longer training.

2. B

Every configuration is another draw from a noisy distribution, and taking the maximum of many draws lands above the true value. With a two-point standard error and a hundred comparisons the winner is inflated by roughly two and a half points, which looks exactly like progress. Correlated folds affect the variance of the estimate; they are not what makes the maximum biased.

3. C

At the moment of prediction you cannot know the labels for the last 30 days of what is nominally the past, because those outcomes have not resolved yet. Training on them is training on information that would not have existed. The remedy is a gap between the training and test windows equal to the label delay.

4. D

A validation curve still climbing at 100 per cent means more data helps. One that flattened at 40 per cent with a large gap above it means additional rows are no longer reducing variance, so the money goes to regularisation, fewer features, or a simpler model. The gap itself is not evidence that more rows will close it, and reading it that way is the usual misdiagnosis.

5. B

The bias of the maximum is roughly the standard error times the square root of twice the natural log of the number of comparisons. With 0.8 points and a hundred comparisons that is about 2.4. It grows with the log of the count rather than the count itself, which is why a thousand comparisons costs about three points rather than eighty.

6. C

The second descent is a well-replicated observation about very large models, and it does not transfer. For linear models, trees and boosting the classical U-shaped curve is accurate, and a boosted model at a high learning rate with far too many trees genuinely does get worse. Knowing where a rule's edge is matters as much as knowing the rule.

Module test — 5. Measuring a model honestly

1. B

Recall and specificity are computed within the true classes and do not move with the base rate. Precision divides by what the model flagged. With 100 frauds among 100,000 transactions, the model catches 60 and flags about 600 clean ones, so precision falls from 80 per cent to about 9. Always ask what prevalence a quoted precision was measured at.

2. D

The break-even threshold is the cost of a false positive divided by the sum of both error costs. Ninety divided by 18,090 is about 0.005, far below the library default, so the model should flag many more transactions than it does at 0.5. The formula assumes calibrated probabilities: on an uncalibrated score it gives a cut point on a scale that does not exist.

3. A

Nine hundred false positives against 99,900 negatives is a false positive rate of 0.9 per cent, which looks excellent on a ROC curve. It is also 900 investigations for at most 100 real cases. Precision divides by the flagged set instead, so the precision-recall curve shows the pain immediately, which is why average precision is the headline for imbalanced problems.

4. C

Both Platt scaling and isotonic regression are monotonic, so they change the numbers and leave the order alone. AUC depends only on the order, so it is unchanged by construction. An AUC that moved after calibration means the model, the data or the evaluation set changed as well, and that is worth finding before the result is believed.

5. B

Any two folds share most of their training rows, so the ten differences are correlated and the t-test's independence assumption fails. Dietterich showed the procedure declares differences significant far more often than it should, and repeating the cross-validation makes it worse rather than better. Use a held-out test set and bootstrap the paired difference on it.

6. A

Kleinberg and colleagues, and Chouldechova, proved independently that with different base rates no classifier satisfies calibration within groups and equal error rates in both directions, except a perfect classifier or a useless one. It is a theorem, so no amount of accuracy or data escapes it. A tool claiming all three has picked a definition on your behalf.

Module test — 6. Trees, ensembles, and the algorithms that win on tables

1. A

The tree computes the parent's Gini impurity, computes each child's, weights the children by how many rows they hold, and takes the drop. Here that is 0.42 minus 0.28, a gain of 0.14. It tries every threshold on every feature, takes the largest gain, and recurses. The procedure is greedy and never revisits an earlier split in the light of a later one.

2. D

Bagging alone leaves the trees too alike: if one feature dominates, every tree splits on it first and their errors coincide. Averaging correlated things removes much less variance than averaging independent ones. Sampling features per split makes each tree slightly worse and much more different, and the ensemble improves. That is Breiman's point, and it makes max_features the parameter to tune.

3. B

A forest averages independent estimates, so more of them only reduces variance and the score converges. Boosting adds each tree to the running prediction to explain what is left, so once the signal is exhausted the remaining residual is noise and the model starts fitting it. That is why boosting needs early stopping and a forest does not, and why n_estimators is not a hyperparameter for a forest.

4. C

Distance in d dimensions is a sum of d squared differences, and as d grows that sum concentrates around its expectation, so the spread between nearest and farthest shrinks relative to their size. The word neighbour stops picking anything out, which affects k-NN, k-means, DBSCAN, RBF kernels and distance-based anomaly detection alike. Learned embeddings exist partly to restore it.

5. D

Seeing free, money and cash is close to one piece of evidence and the model counts it as three, so the scores are pushed towards the extremes. Classification needs only the ranking of the class scores to be right, and over-counting rarely changes which class wins. The result is excellent discrimination with unusable probabilities, which is the calibration distinction from module five in its clearest form.

6. A

Permuting one column breaks its relationship with the target while leaving its twin intact, so the model reads the twin and the score barely moves. Do it the other way and the same thing happens, so both look worthless. Cluster the correlated columns and shuffle each cluster as a unit, which measures what the group is worth, and that is usually the question anyone had.

Module test — 7. Learning without labels, and learning a representation

1. B

Assignment is by Euclidean distance to a centre, so the region belonging to each cluster is bounded by the straight lines equidistant between centres. Two stretched, parallel groups get cut across rather than separated along their length, and no value of k repairs it. Density-based methods such as DBSCAN and HDBSCAN follow shape instead and find them.

2. C

Silhouette, Davies-Bouldin and Calinski-Harabasz all presuppose at least two clusters and can only rank candidate values of k. The gap statistic compares the achieved inertia against what uniformly random data with the same bounding box would give, so it can prefer k equal to one. Its poor-man's version, clustering a shuffled copy and comparing, needs no library at all.

3. A

PCA orders directions by variance, and variance is not usefulness. The projection is fitted without ever looking at the target, so the signal you needed can sit in the twelfth component carrying 0.4 per cent of the variance and be thrown away. Linear discriminant analysis is the supervised counterpart, finding directions that separate the classes instead.

4. D

t-SNE optimises for local neighbourhoods and deliberately expands dense regions and contracts sparse ones, so cluster sizes and between-cluster distances carry no information. Only membership is reliable. Run the same procedure on shuffled data before believing any structure, because convincing clusters appear readily from pure noise at low perplexity.

5. B

Two documents on the same topic at different lengths point in the same direction with different magnitudes. The angle captures the topic and the length captures something else — word frequency, document length, how confident the model was. Normalise every vector to unit length and cosine, dot product and Euclidean distance all give the same ranking, which is why vector stores normalise on the way in.

6. C

One per cent of 999,900 clean transactions is 9,999 false positives against 90 true ones, so precision is about 0.9 per cent. No team can work that queue and within a fortnight they will ignore it, which is worse than no detector because everyone believes the problem is covered. The design work is narrowing the population, cascading, or ranking to a fixed capacity.

Module test — 8. Neural networks, built from what you already know

1. D

Matrix multiplication is associative, so W3 times W2 times W1 applied to x is one matrix applied to x. Fifty layers collapse to one with a great deal of wasted arithmetic. Everything a network can do beyond a linear model comes from the bend applied between the layers, not from the layers themselves.

2. B

Backpropagation multiplies local derivatives layer by layer. At most 0.25 per layer means 0.25 to the tenth power by ten layers, so the gradient reaching the bottom is about a millionth of what arrived at the top and the early layers stop learning. ReLU's derivative is exactly 1 where it is active, which is why a crude hinge replaced a smooth curve.

3. C

ReLU's derivative is 1 for positive pre-activations and 0 for negative ones, and backpropagation multiplies by the local derivative. Multiplying by zero stops the signal. If a unit's pre-activation is negative for every training example, its weights never update and the unit is dead permanently, which is the dying-ReLU failure in one line of arithmetic.

4. B

Adam divides each step by the square root of the running average of squared gradients. An L2 term added to the loss is divided by the same quantity, so the parameters receiving the largest gradients get the least decay, which is the opposite of what regularisation is for. AdamW subtracts the decay from the weights directly, outside that scaling, and it is the one-letter fix worth making.

5. A

Dropout zeroes a random fraction of activations during training and must be switched off at inference, which is what evaluation mode does. Forgetting it is among the most common bugs in PyTorch code, and its signature is exactly this: predictions that change per call with no other explanation. Batch normalisation also behaves differently between the two modes.

6. C

Each level of adaptation needs more data than the last. Full fine-tuning has enough freedom to reshape features that took an enormous amount of compute to build, and 300 images across several classes cannot constrain it. The recipe is feature extraction, then the last block, then everything, stopping at whichever step stops improving — and on a few hundred images that is usually the first.

Module test — 9. Into the world, and keeping it honest

1. C

A boosted tree traversal is a few milliseconds; fetching the customer's 30-day transaction count from a database is usually more. The design consequence is to precompute slow-moving features nightly and compute only genuinely real-time ones in the request. Optimising the model when it is 12 per cent of the latency caps your possible improvement at 12 per cent.

2. B

Predictions from the two paths should be identical to floating-point precision. A discrepancy in the fourth decimal is usually a library version; a discrepancy in the first is a feature computed differently. Making it an automated test on every deploy catches a class of error that reading the code does not, in the same way the label-shuffle check does for leakage.

3. A

This is label shift: the base rate moved and the relationship held. Ranking is intact, so AUC does not move, but every predicted probability is now wrong and the threshold chosen for the old distribution produces a different flag rate. Recalibration on recent data fixes it and is far cheaper than retraining, which is the remedy for concept drift instead.

4. D

The threshold was chosen for the previous model's score distribution. A retrained model spreads its scores differently, so the same cut point produces a different number of flags, and the system can behave worse while the model is better. Re-choosing the threshold on validation data, by the same cost or capacity criterion, belongs in every retraining run.

5. B

A search that maximises validation score will enthusiastically find the configuration that best uses the leaked column, so AutoML makes leakage worse rather than better. It automates model selection and hyperparameter tuning, which was never where most of the value sat: framing, features, leakage and the choice of metric all stay with you.

6. C

That logged probability is the propensity, and it is what makes inverse propensity weighting possible, which is how you estimate what a different policy would have done. It is nearly free at logging time and impossible to reconstruct afterwards. Outcomes for declined applicants do not exist, which is the selective-labelling problem the propensity is there to work around.

Machine Learning, Foundations — final examination

1. A ***1. What learning is, and what it is not***

Parameters come from the data; hyperparameters come from you. Split thresholds are fitted on the training rows, but depth, learning rate and penalty strength are set before fitting and chosen by trial on validation data. Choosing them on the training set means picking whatever fits it hardest, which is always the most complex option available.

2. C ***1. What learning is, and what it is not***

A supervised model learns the label, not the reality behind it. The label here is an administrative act performed by particular clinics under particular insurers, so the model reproduces the coding process. It may still be useful, but the sentence to write before looking at any metric is: this label was created by whom, and what does it miss.

3. B ***1. What learning is, and what it is not***

Unsupervised methods generate hypotheses and cannot confirm them. k-means partitions whatever it is given, returning tidy, describable, reproducible groups from pure noise. Before believing a clustering, run the same procedure on shuffled data and see whether the output looks different, and check whether the groups survive resampling.

4. D ***1. What learning is, and what it is not***

Only a dataset containing rejected applications can speak about the decision to approve. It is also the grain you almost never have labels for, because a rejected applicant never got the chance to repay. Naming the grain first is what surfaces that problem before three months of work rather than after.

5. B ***1. What learning is, and what it is not***

Log loss for a positive row is minus the log of the predicted probability, and minus the natural log of 0.0001 is about 9.2. As the prediction approaches zero the loss goes to infinity, which is the property that punishes confident mistakes savagely and lets a single row dominate a validation score.

6. C ***1. What learning is, and what it is not***

Weighting deliberately misrepresents the base rate to the model so that minority errors cost more, and the predicted probabilities come out shifted as a result. If you need probabilities you can compute with, leave the weights out and move the threshold instead, or recalibrate on held-out data afterwards.

7. A ***1. What learning is, and what it is not***

Human disagreement bounds what any model can be measured to achieve, because the labels themselves contain that much noise. Between the baseline and the human agreement rate is the whole range in which the work can make a difference, and it is often much narrower than anyone assumed. Measuring it costs two people and an afternoon.

8. C ***2. Linear models and the machinery of fitting***

A linear model extrapolates the fitted line indefinitely and nothing in it warns you when you have left the range of the training data. This is the failure that most often reaches production quietly. A tree has the opposite behaviour, returning the mean of its rightmost leaf, which is different rather than better.

9. A ***2. Linear models and the machinery of fitting***

Enormous coefficients of opposite sign that cancel are the signature of a singular design matrix. The classic cause is one-hot encoding every level of a category while keeping the intercept, so the dummies sum to the intercept column. Drop one level, or add a ridge penalty, which makes the matrix invertible by construction.

10. B ***2. Linear models and the machinery of fitting***

An epoch is a fixed number of rows, so a larger batch means fewer steps. Averaging over more rows cuts the gradient noise, and that noise is doing something useful: it shakes the walk out of flat regions and poor local minima. This is why very large batches sometimes generalise slightly worse at equal training accuracy.

11. D ***2. Linear models and the machinery of fitting***

Training is measured on a handicapped model when dropout is on, and training loss is often averaged across the epoch while validation is measured at the end, so training looks worse by the amount the model improved during it. An easier validation set is the guilty explanation and usually means the split was not random.

12. C ***2. Linear models and the machinery of fitting***

With complete separation, log loss keeps improving as the coefficient grows, because pushing a prediction from 0.999 to 0.9999 still reduces the loss. The optimiser chases it to the iteration limit. A perfect separator in training data is usually a column recorded after the outcome, and the default L2 penalty is what normally hides this.

13. A ***2. Linear models and the machinery of fitting***

Softmax makes the classes compete for a fixed budget of probability, which is correct when exactly one label is true and wrong here: a genuinely three-topic article would have to divide its confidence three ways. Independent sigmoids with their own binary cross-entropy and their own thresholds let each label be true on its own terms.

14. B ***2. Linear models and the machinery of fitting***

Weighted averaging is dominated by the common classes, so two dead classes barely move it. Macro takes the plain mean across classes, so a class with twelve examples counts as much as one with forty thousand and the failure appears immediately. If the rare classes are the point, use macro and say that you did.

15. D ***3. The data is the job***

A tree asks whether income exceeds a threshold, and the answer is unchanged if the whole column is divided by a thousand. Scaling matters for distance-based methods, for penalised linear models and for anything fitted by gradient descent on the features. Before a boosting library it is harmless and pointless.

16. B ***3. The data is the job***

On the raw scale, 10,000 to 20,000 and 1,000,000 to 1,010,000 are the same difference. On the log scale, 10,000 to 20,000 and 1,000,000 to 2,000,000 are. The second framing is usually closer to how the world works, which makes the log a claim about the domain rather than a cosmetic change to a plot.

17. C ***3. The data is the job***

The unseen category is encoded as none of the above and a prediction still comes out. That is a genuine choice with a downside, since silently treating an unknown city as absent may be exactly wrong for the problem, but a crash at two in the morning is worse. The important part is that the behaviour was decided rather than discovered.

18. A ***3. The data is the job***

Count encoding replaces a category with how often it appears, which is a property of the feature column alone. Nothing about y enters it, so the mechanism that makes target encoding leak cannot occur. It is cheap, safe, often surprisingly effective — a rare merchant is a different kind of thing from a common one — and badly underused.

19. D ***3. The data is the job***

The cyclical encoding exists because linear and distance-based models need continuity, and 23 and 0 being 23 units apart breaks them. A tree needs no such help: two splits reconstruct night exactly. Turning a sharp boundary into a curve the tree must approximate with several splits makes it a little worse rather than a little better.

20. B ***3. The data is the job***

Every method in the course assumes training rows are drawn from the same distribution as the rows you will predict on, and a held-out set inherits whatever selection produced the training rows. This is the one class of problem holding data out cannot find. It has to be reasoned about from how the rows came to exist.

21. C ***3. The data is the job***

SMOTE interpolates between a minority point and its neighbours, so a synthetic row can be built from a point that later lands in the held-out fold. Resampling before splitting puts information from the held-out rows into training, which is the most common way SMOTE produces a wonderful number that means nothing.

22. A ***4. Generalisation, and the discipline of held-out data***

Twenty positives across five folds is four per fold on average, and unstratified sampling can easily give one fold twelve and another none. Scores then swing for reasons that have nothing to do with the model, and at extreme imbalance some metrics are undefined while others silently return zero. StratifiedKFold is the default for classifiers and should stay.

23. C ***4. Generalisation, and the discipline of held-out data***

This is adversarial validation. A correct random split should be indistinguishable, giving an AUC near 0.5. Well above that means the sets come from different populations, whether from a split bug, hidden time ordering, or genuine distribution shift. It takes ten lines and finds problems no amount of reading the code would surface.

24. B ***4. Generalisation, and the discipline of held-out data***

Leave-one-out has very low bias and famously high variance, and it costs one fit per row. It is occasionally right for very small datasets and is more often a sign that somebody has more compute than data. Five folds is the default and the right answer most of the time.

25. D ***4. Generalisation, and the discipline of held-out data***

If moving the fold boundaries changes the score by as much as changing the model does, the comparison has not resolved anything. More repeats do not help, because the repeats share almost all their training data and are highly correlated. The fold-to-fold standard deviation is a description of stability, not a confidence interval.

26. A ***4. Generalisation, and the discipline of held-out data***

Bias is being wrong the same way every time and variance is being wrong differently every time. Fifty refits agreeing with each other and disagreeing with the truth is the definition of the first. Adding regularisation here makes it worse; the model family cannot represent the pattern, or the features do not carry it.

27. B ***4. Generalisation, and the discipline of held-out data***

Sampling uniformly between 0.001 and 100 puts 99.9 per cent of the draws above 0.1, which is almost certainly the wrong end of a range whose useful values span several orders of magnitude. Log-scaled sampling spends the budget evenly across those orders, which is what makes random search beat grid search at equal cost.

28. D ***4. Generalisation, and the discipline of held-out data***

The standard error of a proportion at $p = 0.85$ is about 1.1 points at a thousand rows and about 0.36 points at ten thousand. To resolve a one-point difference you need the noise well below one point, so ten thousand is the order required. The calculation takes a minute and almost nobody does it before starting.

29. C ***5. Measuring a model honestly***

Precision is a column of the confusion matrix and recall is a row. The sentence that fixes it permanently: precision is how much of my alarm was justified, and recall is how much of the danger I noticed. Mixing them up is the most common error in this area, and precision alone is not even a property of the model, since it moves with the base rate.

30. A ***5. Measuring a model honestly***

When capacity is the constraint, the question is not which transactions are fraud but which two hundred are most worth looking at. That changes what you optimise: only the ranking at the very top matters, and a model that is better there and worse elsewhere is the better model. Average metrics across the whole score range cannot tell you that.

31. B ***5. Measuring a model honestly***

The harmonic mean bakes in a judgement that should be made deliberately. In cancer screening a miss and a false alarm are not equally costly, and in spam filtering they are not either. Report precision and recall at a stated operating point instead, so a reader can see which side of the trade-off you chose.

32. D ***5. Measuring a model honestly***

Log loss is a proper scoring rule, minimised exactly when the predicted probabilities equal the true ones, so a model fitted on it has no incentive to distort them. Naive Bayes over-counts correlated evidence and pushes towards the extremes, forests are under-confident at the ends, and an SVM produces distances that need a calibration step bolted on.

33. C ***5. Measuring a model honestly***

A funnel widening to the right means the errors grow with the size of the prediction, which is what a multiplicative relationship looks like on an additive scale. Fitting log of y makes the errors multiplicative to match. The plot takes a minute and carries information no summary metric retains.

34. A ***5. Measuring a model honestly***

RMSE squares before averaging, so large errors dominate it, and it is always at least as large as MAE. A ratio near one means errors of consistent size; a ratio near three means a small number of very large ones. Reporting both is more informative than reporting either, and the gap is the part that carries the information.

35. B ***5. Measuring a model honestly***

Slicing a hundred ways is a hundred comparisons, and a group of eleven rows scoring 0.45 is noise. This is the multiple comparisons problem from module four wearing new clothes. Guard on group size, and confirm any finding on a second period or a second sample before anyone acts on it.

36. B ***6. Trees, ensembles, and the algorithms that win on tables***

A tree can only cut perpendicular to an axis, so a diagonal boundary becomes a staircase that needs many splits and a great deal of data to do badly what one division does exactly. This is why engineering the ratio as a column helps trees as well as linear models, even though trees find interactions without being asked.

37. D ***6. Trees, ensembles, and the algorithms that win on tables***

Drawing n rows from n with replacement leaves about 1/e of them, roughly 36.8 per cent, unused by that tree. Each tree can be scored on the rows it did not see, and averaging across trees gives a generalisation estimate with no separate holdout and no cross-validation loop, which is genuinely useful on small datasets.

38. A ***6. Trees, ensembles, and the algorithms that win on tables***

Each tree contributes the learning rate times its correction, so halving the rate halves the progress per tree. Halve one and double the other is the exact trade, and lower rates generalise slightly better at proportionally more time. Set the tree count high and let early stopping choose it rather than putting it in a grid search.

39. C ***6. Trees, ensembles, and the algorithms that win on tables***

In LightGBM, setting the row subsample fraction alone does nothing without also setting how often the subsample is redrawn. It is a long-standing trap that costs people an afternoon, and it is the reason a run that should have been regularised behaves exactly as it did before.

40. B ***6. Trees, ensembles, and the algorithms that win on tables***

The RBF kernel is exp of minus gamma times the squared distance between two points, so a feature measured in rupees swamps one measured in years exactly as it does for k-nearest neighbours. Scaling is not an optimisation nicety here; without it the similarity being computed is essentially one feature.

41. D ***6. Trees, ensembles, and the algorithms that win on tables***

SHAP explains the model, not the world. It divides one prediction's departure from the average among the features with an exact additive decomposition, which is genuinely useful for debugging, for spotting leakage, and for an adverse-action notice. It is not an estimate of what would happen if the feature were changed, which needs an experiment.

42. A ***6. Trees, ensembles, and the algorithms that win on tables***

Partial dependence plots an average, and an average over two opposite populations is flat. Individual conditional expectation lines draw one curve per row: parallel lines mean the average is a fair summary, and crossing lines mean the effect of this feature depends on something else in the row. The check costs nothing and stops a false report.

43. C ***7. Learning without labels, and learning a representation***

Clustering has no target to tell it what matters, so what matters is entirely encoded in how you scaled. Standardising says that a standard deviation of age and a standard deviation of income are equally important, which is a choice about similarity. If age genuinely matters twice as much for your purpose, you can and should say so by scaling it.

44. A ***7. Learning without labels, and learning a representation***

A single radius that separates a dense region will merge everything in a sparse one, and a radius tuned for the sparse region fragments the dense one. This is DBSCAN's main practical limitation. HDBSCAN removes it by building a hierarchy across all density levels and extracting the most stable clusters from it.

45. B ***7. Learning without labels, and learning a representation***

PCA maximises variance, and a column measured in rupees has more variance than one measured in years for reasons that have nothing to do with importance. Without standardising, the first component is whichever feature has the largest units. Fitting on the whole dataset before splitting is a separate and also serious error, in the opposite direction from the option.

46. C ***7. Learning without labels, and learning a representation***

t-SNE has no transform: new points require rerunning the whole embedding. UMAP can place them, which is its main practical advantage along with speed. It preserves more global structure than t-SNE and still not enough for between-cluster distances to be quoted, and both benefit from a PCA reduction to thirty or fifty components first.

47. D ***7. Learning without labels, and learning a representation***

Implicit feedback has no negatives. A film you never watched might be one you would love. Treating absence as dislike is wrong, and the standard approach is to treat observed interactions as positives with a confidence weight and unobserved ones as low-confidence negatives. Getting this framing right matters more than the choice of algorithm.

48. A ***7. Learning without labels, and learning a representation***

Embeddings represent meaning, and a part code or an order number has no meaning to represent. Keyword search wins outright on exact identifiers, on rare exact terms, and on negation, which embeddings handle badly. The standard answer is hybrid search: run BM25 and embedding search in parallel and fuse the rankings.

49. B ***7. Learning without labels, and learning a representation***

Uncertainty sampling deliberately concentrates labels near the decision boundary, which is what makes it efficient for training and what disqualifies it for evaluation. A test set drawn that way describes boundary cases rather than the population. Keep a randomly labelled set alongside, always, or you cannot tell whether any of it worked.

50. C ***8. Neural networks, built from what you already know***

The theorem says such a network exists. It says nothing about whether the optimiser can reach it or how many units it would need, and for many functions a shallow network needs exponentially more units than a deep one. Depth buys reuse: early layers learn pieces that later layers combine, and the pieces are shared.

51. A ***8. Neural networks, built from what you already know***

You want the variance of the activations to stay roughly constant as the signal passes through, in both directions. ReLU sets negative pre-activations to zero, so about half the variance disappears at every layer, and the extra factor of two in the initialisation compensates. Xavier was derived for a symmetric activation such as tanh, where that does not happen.

52. B ***8. Neural networks, built from what you already know***

Batch statistics are used during training and a running average at inference, so a model can train well and behave differently when it serves. It also performs badly with small batches and creates a dependence between the examples in a batch. Layer normalisation removes all three problems, which is why transformers use it.

53. D ***8. Neural networks, built from what you already know***

The gradient flows through both branches and the identity path multiplies by nothing, so it cannot shrink. That direct route around the multiplication chain is what made networks of fifty, a hundred and a thousand layers trainable. Before residual connections, adding layers past about twenty made networks worse on the training set, which is a failure to optimise rather than overfitting.

54. A ***8. Neural networks, built from what you already know***

Backpropagation multiplies the gradient arriving from above by each operation's local derivative, and those local derivatives are computed from the values saved on the way forward. Every intermediate activation therefore has to be retained until it is consumed, which is why the standard fix when a model will not fit is to reduce the batch size. Adam's two extra values per parameter add to it.

55. C ***8. Neural networks, built from what you already know***

Each filter spans the full depth of the input, so it holds 3 times 3 times 3 weights, and there are 64 filters. Compare that with the 150 million a fully connected first layer would need on the same image. The saving comes from local connectivity and from applying the same filter at every position rather than learning a separate detector per location.

56. B ***8. Neural networks, built from what you already know***

Every query is compared against every key, so doubling the context quadruples the computation. This is the central engineering constraint of the architecture and the reason context windows were 512 tokens in 2018. Sparse patterns, linear approximations and memory-efficient exact implementations work around it; none of them changes the quadratic relationship itself.

57. D ***9. Into the world, and keeping it honest***

Six months after you publish a score, teams you have never met are reading it, and one of them uses it in a way you would not endorse. Your change breaks their system with no visible connection to what you did. Access to model outputs needs to be as deliberate as access to a database, which usually means an interface rather than a shared table.

58. B ***9. Into the world, and keeping it honest***

Most teams need less online serving than they assume. An hourly batch job avoids latency budgets, autoscaling, availability requirements and the constraint that every feature be computable inside a request. Asking what the freshness requirement actually is, before choosing an architecture, is free and frequently changes the answer.

59. C ***9. Into the world, and keeping it honest***

When a training set is assembled, the store returns each feature as of the moment of the event rather than as of today. That is the leakage guard from module three implemented as infrastructure, and it is what stops a nightly-refreshed balance column from describing today rather than the application date. Consistency between paths is the store's other property, and a different one.

60. A ***9. Into the world, and keeping it honest***

Most problems are upstream data problems, and they appear in the input layer within hours: a column that becomes fully null, a currency changing units, a category renamed. System health stays green while the model returns confident nonsense, and actual performance waits for labels that may take ninety days. The first three layers are early warnings for the fourth.

61. B ***9. Into the world, and keeping it honest***

They are conventional cut points, not theory, and their value lies in being shared so that two teams mean the same thing by a moderate shift. Whatever the number says, plot the two distributions before acting: the index tells you something changed and the plot tells you what.

62. C ***9. Into the world, and keeping it honest***

Unstructured pruning reduces the count of non-zero parameters, which is a different thing from reducing time or memory, because standard hardware processes a sparse matrix with scattered zeros exactly as it processes a dense one. Structured pruning removes whole channels, filters or attention heads, achieves lower sparsity, and gives a real speedup.

63. A ***9. Into the world, and keeping it honest***

Out-of-scope use is the entry that outlives everyone involved: trained on applicants aged 21 to 65 in three states, and not to be used outside that population. Hyperparameters and versions belong in the run log and matter for reproducibility. The card exists so that somebody in two years knows what the model is not for.