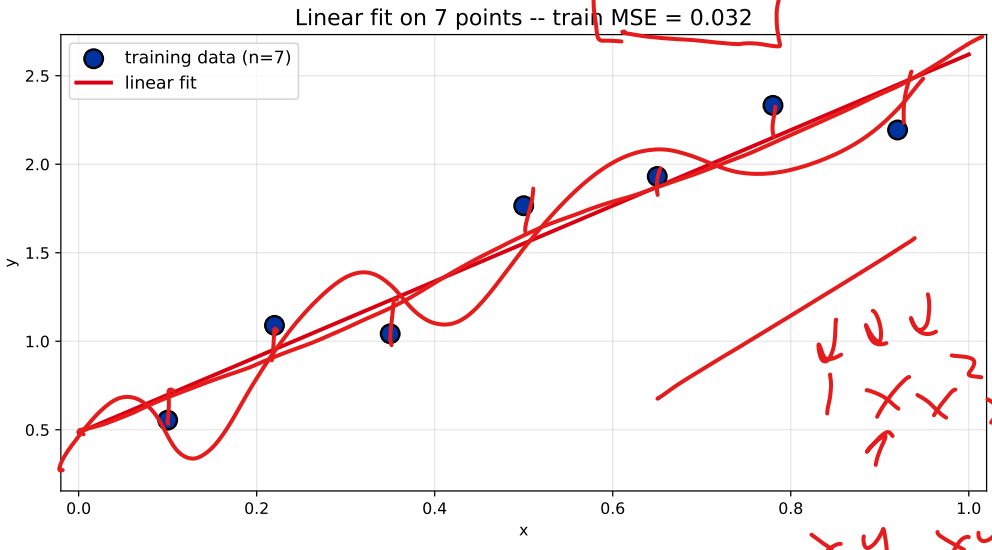
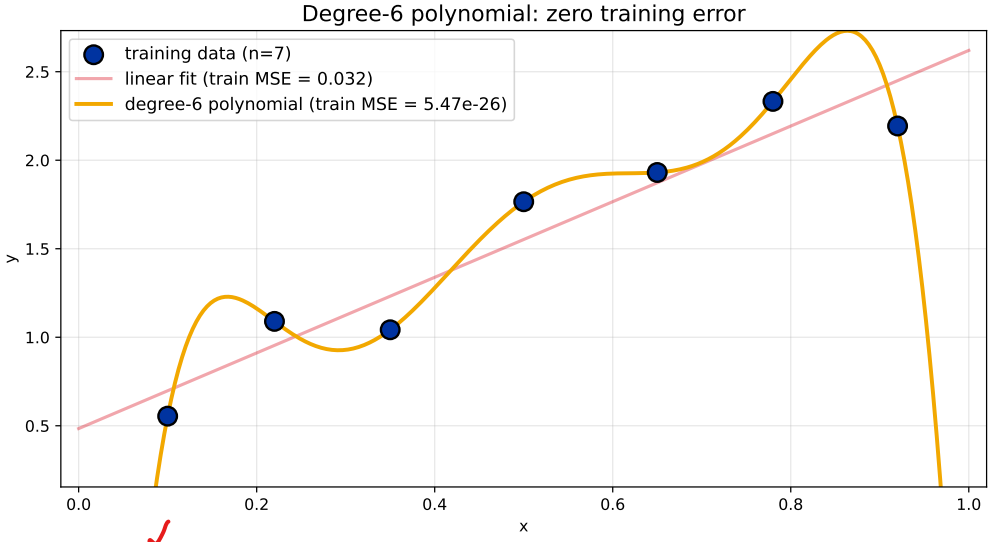
A few noisy points. Fit a line.



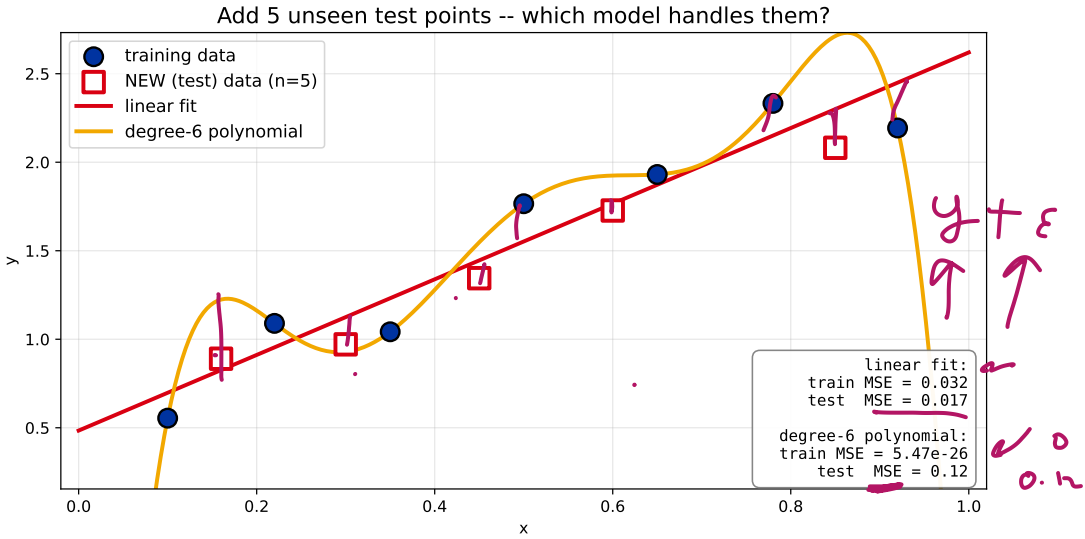
Train MSE ≈ 0.03. Not bad.

Same points. Fit a degree-6 polynomial.



Train MSE $\approx 10^{-26}$. *Should we ship the polynomial?*

Add 5 unseen points. Which model handles them?



Polynomial test MSE $\approx 0.12 - 7\times$ worse than the line (≈ 0.02). Training error lies. *Today: how to*

Outline

The problem: training error lies

Train/test split

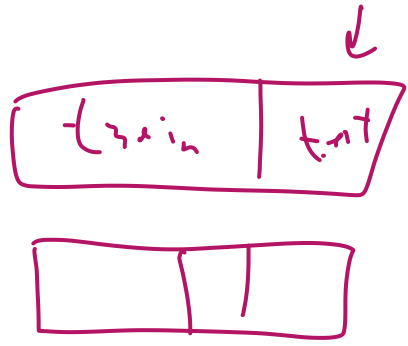
Why train/test isn't enough: validation

Cross-validation

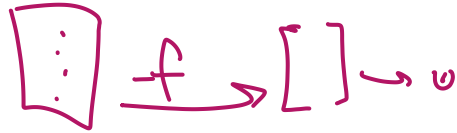
Stratified CV (and friends)

Nested CV

Wrap-up



Why we need to hold data out

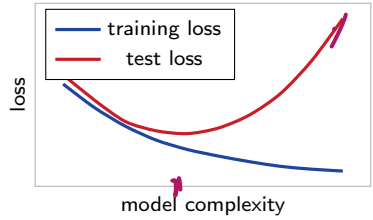


We fit a model by minimizing empirical risk on the training data (L01, L01b). The fitted model is the one with the lowest training loss.

Problem: the training loss is also the thing we used to choose the model. We're *grading the model on the questions we wrote the answer key from*.

We need an **independent estimate** of how the model will do on data it hasn't seen.

The picture every ML student should have in mind:

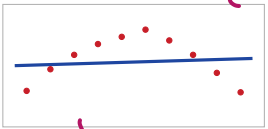


Training loss keeps falling. Test loss has a sweet spot, then climbs.

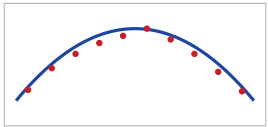
Two ways a model can be wrong

Same noisy curved data fitted three ways:

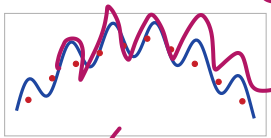
Underfit (degree 1)



Sweet spot (degree 3)



Overfit (degree 9)



Underfit (high bias). The model is too simple to capture the pattern. Training error AND test error are both large; the model is missing real structure.

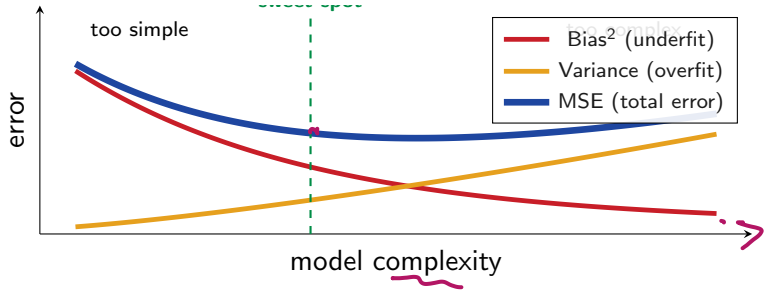
Overfit (high variance). The model is too flexible – it captures the training data and its noise. Training error is tiny; test error is huge.

Both are bad. The art of ML is finding the sweet spot between them.

Bias-variance tradeoff

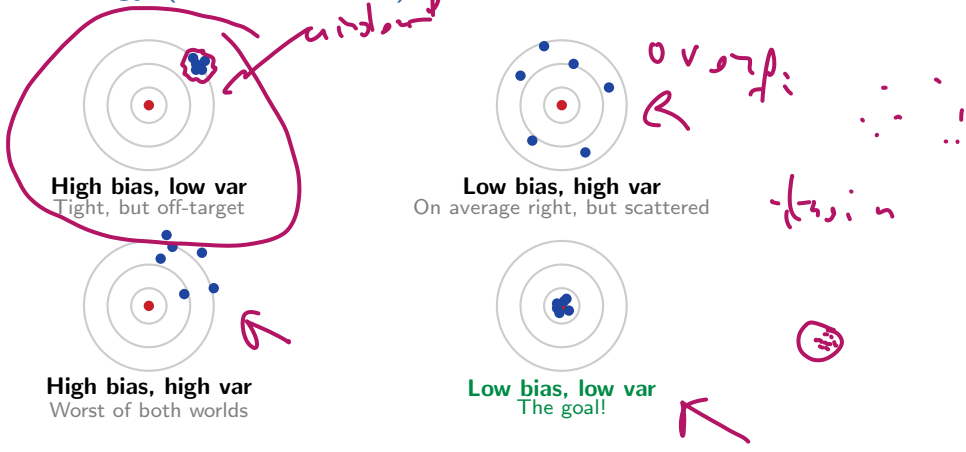
These two failure modes have formal names you already met in stat lectures (`03_stat`): **bias** and **variance**, with the identity

$$\mathrm{MSE}(\hat{\theta}) = \mathrm{Bias}^2(\hat{\theta}) + \mathrm{Var}(\hat{\theta})$$



Bias falls as the model gets richer; variance rises. MSE bottoms out somewhere in between. *The whole job of validation is to find that sweet spot empirically.*

The dartboard analogy (from 03_stat)



Bullseye = truth. Blue dots = estimates from many training sets.

Bias-variance is everywhere in ML

The only model class you've met so far is **polynomial regression**, where the complexity knob is the degree d:

- Too simple: $d = 1$ (a line through curved data) – high bias.
- Too complex: $d = 20$ (wiggly, like Demo 1) – high variance.
- Sweet spot: somewhere in between, found by validation.

Every model class you'll meet later has its own "complexity knob" and the same U-shape applies:

Model (upcoming chapter)	**Complexity knob**
Polynomial regression (now)	degree d
Ridge / Lasso (next chapter)	penalty strength λ
KNN (classification chapter)	number of neighbors k
Decision trees (later)	maximum depth
Neural networks (much later)	width / depth, dropout

Key insight. Different knobs, same shape. This is why we need **held-out test sets** and **cross-validation** – two empirical ways to find the sweet spot, regardless of the model.

Historical examples

Overfit: Babylonian astrology.
For centuries, astrologers built detailed rules connecting planetary positions to historical events: *"When Mars is in Taurus and the moon is half full, there will be a great harvest if the newborn prince is left-handed."*
They squeezed meaning out of every alignment – coincidences and noise treated as signal.

Underfit: medieval bloodletting.
For centuries, European doctors had one rule: *"Something's wrong? Drain blood."*
Used for headaches, fevers, plague, melancholy alike – a single too-simple rule for every disease the model couldn't tell apart.

The art of ML is in the middle: complex enough to match real structure, simple enough not to chase noise.

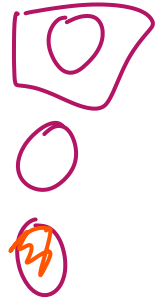
Underfit – Kargin Haxordum example

Serjant Soghomonyan

https://youtu.be/pv_4MVcjEik

Overfit – Kargin Haxordum example

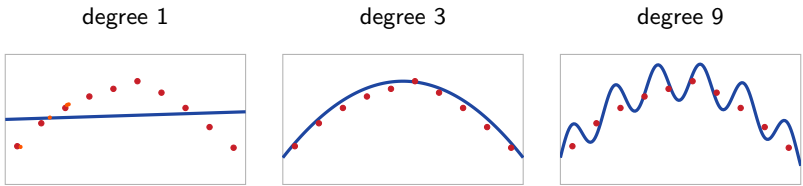
Ete qarakusi a meju klor a



https://youtu.be/7arnxebkEUU

Predict-first: can a degree-9 polynomial fit 10 points perfectly?

You have 10 noisy samples from $y = \sin(\pi x) + \varepsilon$. You fit polynomials of degree 1, 3, and 9. Reminder from L01b:



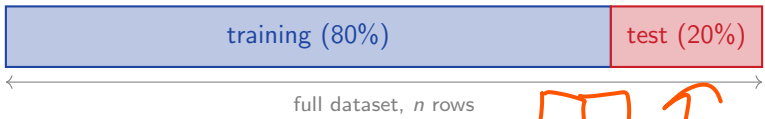
Question: which has the smallest *training* error? Is that the model you should deploy?

Degree 9 has the smallest training error (it interpolates every point). It is also the *worst* model. Training error is not a measure of model quality.

The fix: hold out data the model never saw

Split the data into two pieces:

- **Training set** - used to fit the model.
- **Test set** - used *once* at the very end to estimate generalization.



Typical splits: 80/20, 70/30 for small data, 90/10 for large data.

Tradeoff: smaller test set ⇒ more data to train on, but noisier estimate. Bigger test set ⇒ honest estimate, but less training data. The right ratio depends on how much data you have.

Parameters vs. hyperparameters

Before we go further, one piece of vocabulary that the rest of this lecture (and the rest of the course) leans on:

Parameter θ. A value the model *learns* from data during training.

- Linear regression: the coefficients $\theta_0, \theta_1, \ldots$
- Polynomial regression: same – one per basis column.

Found by minimizing the empirical risk (L01, L01b).

Hyperparameter. A setting *you* choose before fitting. The model can't learn it because changing it would change the model class itself.

- Polynomial regression: the degree d.
- Gradient descent: the learning rate α.
- Train/test split: `test_size`, `random_state`.

More hyperparameters appear as we meet new model classes in upcoming chapters.

Why this matters here: validation exists to help us pick hyperparameters honestly. "Try a few values, look at the score, pick the best" is the whole game for the next several frames.

Mechanics: sklearn train_test_split

Call signature:

```
train_test_split(X, y, test_size=, random_state=, shuffle=, stratify=)
```

Returns: `X_train, X_test, y_train, y_test`.

argument	what it controls
`test_size=0.2`	fraction held out (80/20 split)
`random_state=509`	seed – same split every run; required for debugging
`shuffle=True` (default)	randomize row order before split
`shuffle=False`	keep chronological order – use for time-series data
`stratify=y`	preserve class ratio (classification only – see Section 5)

Critical rules

Ranked by how often students miss them:

1. **Don't iterate on the test set.** The most-violated rule. If you make any modeling decision based on test-set behavior, the test set becomes part of training. Look at test *once*, at the end.
2. **Split before any preprocessing.** The leakage rule from L01c (*fit on train, transform on train and test*) requires you to have split first. Without this, mean-imputation and scaling silently leak.
3. **Group-structured data needs group splits.** If you have 10 photos per person and randomly split, the same person appears in both train and test. The model memorizes faces, not features. Use `GroupShuffleSplit`.
4. **Time series needs chronological splits.** A random shuffle leaks the future into the past. Use `TimeSeriesSplit` or a single chronological cutoff.

What about EDA? Distribution-level facts (sizes, classes present, ranges) are fine to check on the full data before splitting. The rule is about *modeling decisions* and *performance numbers*, not about awareness.

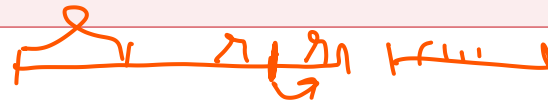
Nuance: when the rows aren't independent

Random shuffle assumes every row is exchangeable. Two cases where this is wildly wrong:

1. Multiple records per entity (patients, users, photos).
100 patients $\times$ 5 visits = 500 rows. Random 80/20 puts $\sim$ 4 visits of the same patient in train and $\sim$ 1 in test $\Rightarrow$ the model memorizes *the patient*, not the disease pattern. Use `GroupKFold(...).split(X, y, groups=patient_id)` – the `groups=` argument pins all rows of an entity to a single fold.

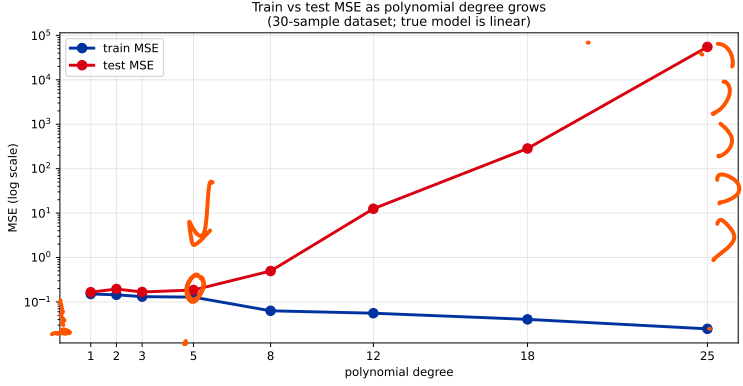
2. Time-ordered data (forecasting, sensor readings, prices).
Random shuffle puts tomorrow's row in training and yesterday's row in the test set – training on the future to predict the past. Use `TimeSeriesSplit(n_splits=5)` (expanding-window CV), or simply a single chronological cutoff: first 80% train, last 20% test.

Diagnosis: (a) no group ID in both train and test, (b) no test timestamp before the latest train timestamp. Violate either $\Rightarrow$ leaking.



Demo: train vs test MSE as polynomial degree grows

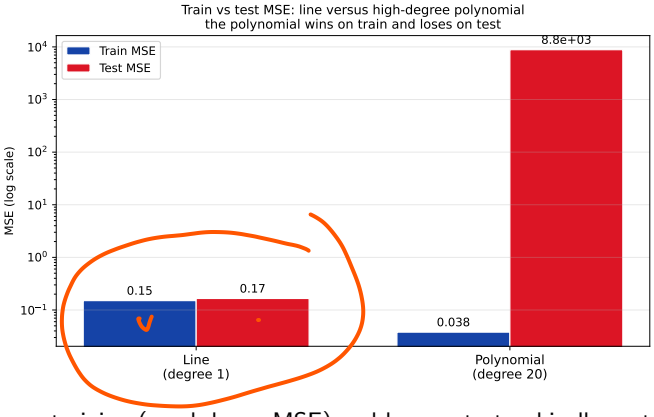
30 noisy samples from a linear truth ($y = 4 + 3x + \text{noise}$). Fit polynomials of degree 1 to 25 on the training split. For each, measure train MSE and test MSE on the test split.



Train MSE drops monotonically. Test MSE bottoms out around degree 1–3 (matching the true model), then explodes. *The train-test gap is the empirical fingerprint of overfitting.*

Demo: line versus high-degree polynomial

Same 30-sample dataset. Two models: a straight line, and a degree-20 polynomial. Train MSE and test MSE side by side:



The polynomial wins on training (much lower MSE) and loses catastrophically on test. *If you only looked at the train bars you would pick the polynomial. That is exactly the trap that train/test split exists to prevent.*

The hyperparameter tuning trap

After splitting, you try several models or hyperparameter settings. You evaluate each on the test set, then pick the best.

Common workflow. Let's check whether the reported number is honest.

Each comparison *uses* the test set. If you pick the configuration with the best test score, you've absorbed the test set into your model-selection process.

Analogy. You take a practice exam 100 times until you score perfectly. Do you expect to score perfectly on the real exam?

Three sets, three jobs

Each split has exactly one purpose. Mixing them is how reported numbers stop predicting reality.

set	purpose	how often you may touch it
training	fit **parameters** (θ)	as often as you want
validation	fit **hyperparameters** (degree d, α, λ)	many times, while tuning
test	estimate **true generalization**	**exactly once**, at the end

Why? Every time you look at a score and *decide* something, you've used that data to fit. Training data fits θ (allowed). Validation data fits hyperparameters (allowed). If you peek at the test set and adjust anything, you've used it to fit too – and the number you report is no longer an honest estimate.

One validation set is noisy

You have 200 Yerevan listings. 60/20/20 split ⇒ 40 validation rows.

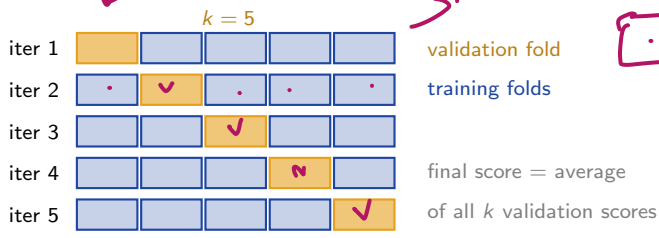
Model A scores 0.81 MSE on validation, model B scores 0.79. Is B really better, or is the gap just noise on 40 samples?

Same data, different random split can flip which model looks better. On a 200-row dataset the validation score has a standard error of several percent.

Solution: use k different validation sets and average their scores. Averaging k roughly-independent estimates reduces the variance of the mean by a factor of $\approx 1/k$ (CLT-style, math module 20). This is k-**fold cross-validation**.

k-fold cross-validation: how it works

Split the (non-test) data into k equal chunks called *folds*. For each of k iterations, hold out one fold as validation and train on the rest.



Subtle point: you train k different models (one per iteration), not one. The CV score estimates how well the *training protocol* generalizes, not any specific model's performance. The final model you ship is usually fit on *all* the training data after CV picks the hyperparameters.

Choosing k and the cost

Standard choices:

- $k = 5$ - the default. Gives 80/20 train/val per fold (matches standard hold-out), bias is low, compute manageable.
- $k = 10$ - lower variance, 2x more compute. Use when you can afford it.
- $k = n$ - leave-one-out (LOO). Train n models. Used only when n is tiny; the LOO score has surprisingly high variance because folds are nearly identical.
- `RepeatedKFold(n_splits=5, n_repeats=3)` - repeats k-fold with different random splits and averages. Lower variance when compute allows.

When CV is worth it:

- Small datasets ($n < 10{,}000$) - one validation set is noisy.
- Model selection / hyperparameter tuning.
- Reporting a confidence interval on the score (mean $\pm$ std over folds).

When a single split is enough:

- Big data ($n > 10^6$) - one split is already low-noise.
- Quick prototyping iterations.
- Compute is the bottleneck.

CV in code

Call signature:

```
cross_val_score(model, X, y, cv=, scoring=)
```

Returns: array of k scores (one per fold).

argument	what it controls
`model`	any sklearn estimator; refit per fold
`X, y`	pass *train* subsets only – test stays untouched
`cv=KFold(...)`	the splitter (see `n_splits`, `shuffle`, `random_state` above)
`scoring="neg_mean_squared_error"`	error metric (note the sign $*$)

$*$ **sklearn footgun:** `cross_val_score` *maximizes* by convention, so error metrics are negated. Flip the sign when reporting: `-scores.mean()`. Always report `scores.std()` too – a single number hides instability.

Pipeline + CV: how the leakage rule survives folds

CV trains k models. If you fit your preprocessor (scaler, imputer, encoder) on the full dataset *before* the CV loop, you leak into every fold. The clean fix: wrap preprocessing + model in a `Pipeline`, then pass the *pipeline* (not just the model) to `cross_val_score`.

```
pipe = Pipeline([("scaler", StandardScaler()), ("lr", LinearRegression())])
cross_val_score(pipe, X_train, y_train, cv=KFold(...))
```

Why this works. On each fold, sklearn refits *every* step of the pipeline on that fold's training rows only. The scaler sees fold-train means/stds, never the fold-val rows. Same for OHE categories, imputer medians, target-encoder per-category means.

HW2 used `ColumnTransformer + LinearRegression`; that pipeline drops straight into `cross_val_score`, no change needed.

Demo: what happens when you leak

Toy demo: 100 rows, 1 numeric feature, scale it. Two ways:

WRONG – scaler fit on full data *before* CV	**RIGHT** – scaler inside Pipeline, refit per fold
CV MSE ≈ 0.142	CV MSE ≈ 0.158
looks $\sim 11\%$ *better*	*honest number, the truth*

On this toy dataset the gap is small ($\sim 11\%$). For `target encoding` or `KNNImputer` (which use the target or whole-row similarity), the gap can be **5–10 percentage points of** R^2. Same rule, far worse consequence.

Diagnosis: if your CV score looks suspiciously good, check whether *anything* that was fit before the CV loop touched all the rows. Imputers, scalers, encoders, target encoders, PCA – all common culprits.

Always have a dumb baseline

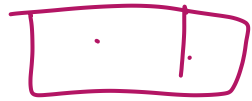
A CV-mean MSE of 0.16 sounds good. *Is it?* Beats predicting the mean every time? Beats predicting the most common class?

```
DummyRegressor(strategy="mean")            # always predicts y.mean()
DummyRegressor(strategy="median")
DummyClassifier(strategy="most_frequent") # always predicts the modal class
DummyClassifier(strategy="stratified")    # predicts proportional to class
    frequencies
```

Run the dummy through the same `cross_val_score(...)` as your model. The gap is the part of the score you can claim credit for.

Workflow rule: fit a dummy baseline first, every project. Then your fancy model has to clear that bar before you tune it. Saves you from celebrating models that aren't doing anything.

The imbalanced-data problem



Many real classification tasks are imbalanced:

- Credit-card fraud: 0.2% fraudulent transactions
- Disease screening: 1% positives in screened population
- Click-through prediction: 2% click rate
- Apartment rent fraud detection: maybe 5% suspicious listings

Random k-fold CV draws folds at random. With a 5% positive class, the positives are scattered uniformly - but *uniformly* doesn't mean *evenly*. Some folds get more, some fewer, sometimes zero.

Why care? A fold with 0% positives breaks the protocol in two ways: the model trained without it never sees the rare class, and the score from that fold isn't meaningful.

Next frame: how often does this actually happen?

Predict-first: 5-fold CV on 5% positives

You have 200 rows of which 10 are positive (5%). You run vanilla 5-fold CV.

Questions:

- How many positives does each fold get *on average*?
- What's the probability that at least one fold has zero positives?

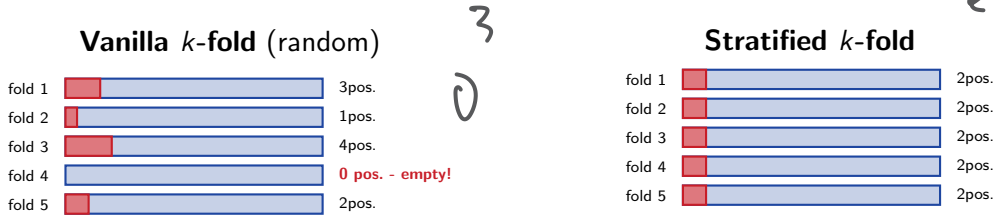
Answers:

- **On average 2 positives per fold** (10 split across 5).
- **Roughly 6% chance** at least one fold is empty - the 10 positives can cluster by chance. (Each fold's probability of being empty is $\binom{190}{40}/\binom{200}{40} \approx 0.8^{10} \approx 11\%$; with 5 folds, $\approx 1-(1-0.11)^5 \approx 44\%$ if independent, but folds are negatively correlated $\Rightarrow$ true probability is $\approx 6\%$.) With smaller imbalance (1% positives, $n = 200$) it's nearly guaranteed.

Solution: split each class separately, so every fold ends up with its share of the minority class. **Stratified k-fold.**

Stratified k-fold: vanilla vs stratified

Same data, two CV strategies. Each row is one fold's positive count (out of 10).



Stratified k-fold splits each class separately. Every fold receives its share. No more empty-positive folds.

For classification: default to `StratifiedKFold`. There's almost no reason to use plain `KFold` on classification data. The sklearn helpers (`cross_val_score`, `GridSearchCV`) auto-stratify if `y` is categorical, but be explicit.

Stratified CV in code

Preview of Ch. 2 (classification). We use a generic "classifier" here; specifics like Logistic Regression, F1, and ROC-AUC come later.

```
from sklearn.model_selection import StratifiedKFold, cross_val_score

skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=509)
scores = cross_val_score(
    classifier,           # any classification model
    X_train, y_train,
    cv=skf,
    scoring="accuracy",   # or a metric that handles imbalance
        better (Ch.2)
)
print(f"score = {scores.mean():.3f} +/- {scores.std():.3f}")
```

- Use `StratifiedKFold` *by default* for classification. Almost no reason to use plain `KFold`.
- In Ch. 2 we'll see why `accuracy` is the wrong metric on imbalanced data (a 1% positive class gets 99% accuracy from the constant predictor).

The problem: tuning + evaluating with the same CV

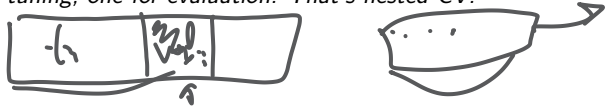
The standard three-way split has clean roles: train for parameters, validation for hyperparameters, test for the final number.

But what if our dataset is small and we already need CV for tuning? Naive workflow:

1. Run 5-fold CV across polynomial degrees $d \in \{1, 2, 3, 5, 9\}$.
2. Pick d^* with the best CV score.
3. Report that CV score as the model's quality.

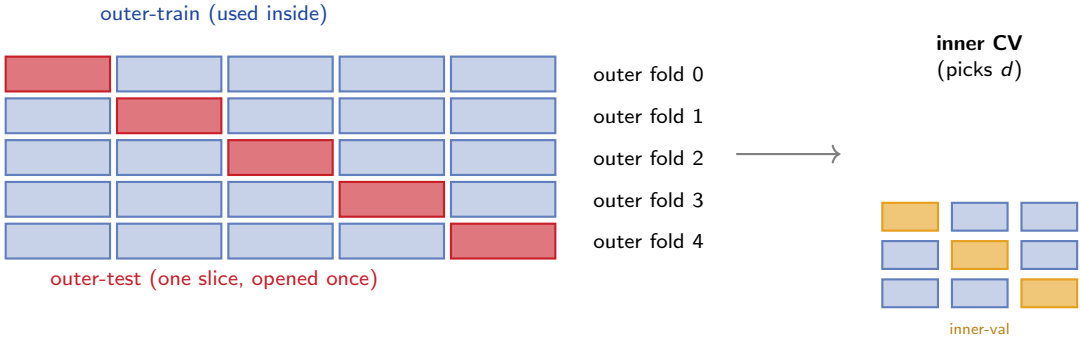
The bug: step 3 reports a score that step 2 *used to pick the winner*. Whichever d happened to do well on these specific folds gets chosen – and we report its score. Same data fitted the choice and graded the choice. Optimism bias: typically 3–10 percentage points.

We need a second layer of CV: one for tuning, one for evaluation. That's nested CV.



Nested CV: how it works

Two CV loops, one inside the other.



For *each* outer fold: run a small inner CV on the outer-train to pick the best d. Then refit on the whole outer-train and score on the outer-test. The K outer-test scores are the honest estimate.

Nested CV: pattern + when to use

Pseudocode:

```
for tr_idx, te_idx in OuterKFold.split(X):          # outer: evaluation
best_d = inner_cv_pick_degree(X[tr_idx], y[tr_idx])      # inner: tuning
score = fit_and_score(X[tr_idx], y[tr_idx],
X[te_idx], y[te_idx], degree=best_d)
```

Key properties.

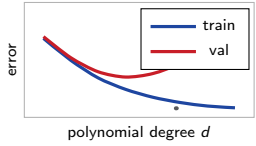
- The outer-test slice is touched **once** per outer fold (just like a hold-out test set).
- Picking d uses only the outer-train – the outer-test never influenced which d was chosen.
- Different outer folds may pick *different* d^* – that's fine. We're not trying to ship one specific d; we're estimating how a *tuning procedure* generalizes.

When to use: any time the number you report is the score of a *tuned* model. The gap between optimism-biased single-CV and honest nested CV can be 3–10 percentage points – enough to flip "best in class" to "worse than baseline".

Three diagnostic curves

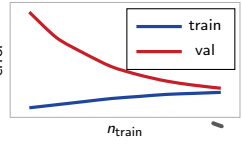
Each plots train (blue) and validation (red) error vs. a single axis. Different axes diagnose different failure modes.

Validation curve
(error vs. hyperparameter)



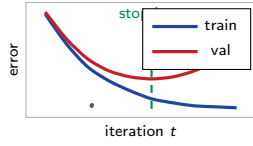
Picks the *best HP*.

Learning curve
(error vs. train size)



Would more data help?

Training curve
(error vs. iteration)



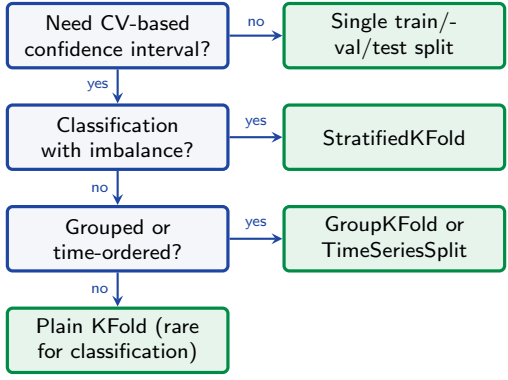
When to *stop training*.

What to do with each shape:

- **Validation curve** sweet spot at the *edge* of your range ⇒ widen the hyperparameter range.
- **Learning curve** gap not closing ⇒ bias-dominated, more data won't help. Closing slowly ⇒ variance, more data WILL help.
- **Training curve** val error climbing while train still falls ⇒ early stopping. Halt at the dashed line above.

Decision flowchart

Single train/val/test split is fine if you don't need a CI on your score - for example, big data, fast prototyping, or compute-constrained settings. CV is for when score variance matters.



Recap

- **Training error lies.** Always evaluate on data the model didn't see.
- **Three sets, three jobs.** Train fits θ; validation fits hyperparameters; test is the final number, touched exactly once.
- **Cross-validation.** Rotates validation through k folds. Default $k = 5$.
- **Stratified CV.** Default for classification. **Group / Time splits.** For non-independent rows.
- **Nested CV.** Honest evaluation when you also tune.
- **Three diagnostic curves.** Validation curve (best HP), learning curve (would more data help?), training curve (when to stop).

The workflow this lecture supports:

1. **First fight underfitting.** Pick a model class rich enough to capture the pattern (more features, higher polynomial degree, etc.) – until train error is low.
2. **Then fight overfitting.** Use validation to detect when the model is now memorizing noise, and use **regularization** (Ridge, Lasso – *next chapter*) to push complexity back down without losing capacity.

Coming next: bias-variance decomposition (L01d2), then regularization (L03) – the principled tool for the overfitting half of the workflow above.