Supervised Learning

From Linear Regression to Gradient Boosting

Lecture Companion Notes (Standalone Edition)

Data Science with Python – BSc Course

Joerg Osterrieder
April 12, 2026

These notes accompany the supervised learning lectures (L21–L28). They follow a problem-first structure: each section opens with a concrete challenge, builds intuition through visuals and analogies, then formalizes the concept with worked examples. The handout is fully self-contained: every derivation, formula, and finance example is developed in-line, with no cross-references to other documents. Read before lecture for preparation, revisit after for deeper understanding.

Contents

1. From Correlation to Prediction – What Is Supervised Learning?

Opening Problem: A Portfolio Manager's 500 Stocks

You are a portfolio manager at a mid-size asset-management firm. On your screen is a spreadsheet with 500 rows—one per stock—and a dozen columns: twelve-month return, daily volatility, price-to-earnings ratio, price-to-book ratio, market capitalization, sector code, analyst rating, and a handful of technical indicators. On a separate sheet sits the quantity you actually care about: next-month return, measured after the fact for stocks you held last year. Twelve features per stock, one label per stock, 500 rows.

Your boss wants a model that takes the twelve features of any new stock and predicts its next-month return. She does not want the model to predict past returns (you already know those). She wants generalization: accurate predictions on stocks the model has never seen. That is the entire job of supervised learning—take a table of features X and labels y, and learn a function f such that $f(X_{\text{new}})$ is close to y_{new} for new inputs.

This section introduces supervised learning as a distinct branch of machine learning. It defines the core concepts (regression, classification, features, labels), sketches the learning paradigm (data, loss, optimization, prediction), and explains why splitting data into train and test sets is the one non-negotiable discipline that separates rigorous ML from wishful thinking.

Discovery Question

You fit a model that predicts stock returns with 99% accuracy on your historical data. You deploy it next quarter and it loses money. Same math, same features, same market. What went wrong? How can a model succeed on the past and fail on the future?

Supervised vs. Unsupervised Learning

Machine learning splits into two large families. In *supervised learning* the training data includes both inputs and labels. You hand the algorithm a table of features X with a column y attached, and the algorithm learns to predict y from X. In *unsupervised learning* there is no label column. The algorithm is asked to discover structure—clusters, components, anomalies—without being told what the right answer looks like.

The asymmetry is sharp. Supervised learning has a target to chase and a metric to optimize. You can measure how well you are doing: compare predictions to labels, compute an error, iterate. Unsupervised learning has no ground truth, so evaluation is indirect. Most practical ML systems you encounter—spam filters, credit scoring, price prediction, medical diagnosis—are supervised. Unsupervised methods appear as preprocessing steps (PCA, clustering) or as exploratory tools that precede a supervised pipeline.

Supervised learning: A class of machine learning algorithms that learn a mapping $f : X \to y$ from labeled examples (x_i, y_i). The goal is prediction: produce $\hat{y} = f(x_{\text{new}})$ for new inputs that were not in the training set.

Features (inputs): The columns of the data matrix X used as inputs to the model. In finance, features might include historical returns, volatility, valuation ratios, or macroeconomic indicators. Each row is one observation; each column is one feature.

Labels (targets): The column y that the model is trying to predict. In regression, y is continuous (e.g., next-month return in percent). In classification, y is discrete (e.g., fraud versus not fraud).

Regression versus Classification

Supervised learning itself splits into two tasks, distinguished entirely by the type of the label y.

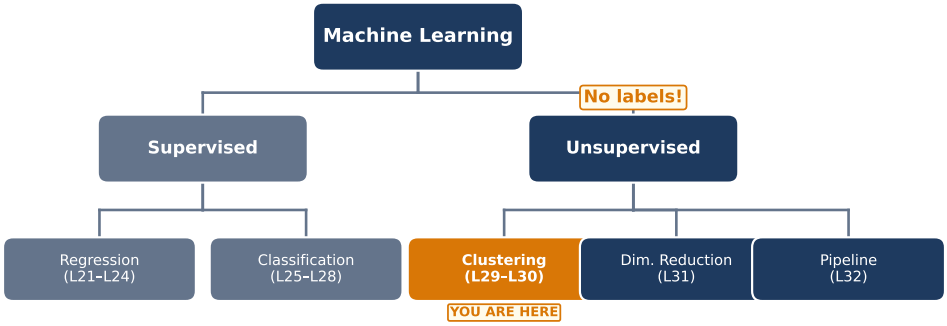


Figure 1: Where supervised learning sits in the machine learning taxonomy. Supervised learning has labels; unsupervised learning does not.

- **Regression:** y is continuous. Examples: next-month return (a number between -0.5 and $+0.5$), house price (dollars), blood pressure (mmHg).
- **Classification:** y is discrete. Examples: default / no-default (binary), stock sector (multi-class), fraud / not-fraud (binary with heavy class imbalance).

The two tasks share most of the machinery—features, train/test splits, loss functions, cross-validation—but they use different loss functions (squared error for regression, cross-entropy for classification) and different metrics (RMSE, R^2 for regression; precision, recall, AUC for classification). Sections 2 through 4 develop the regression side. Sections 5 through 8 develop the classification side.

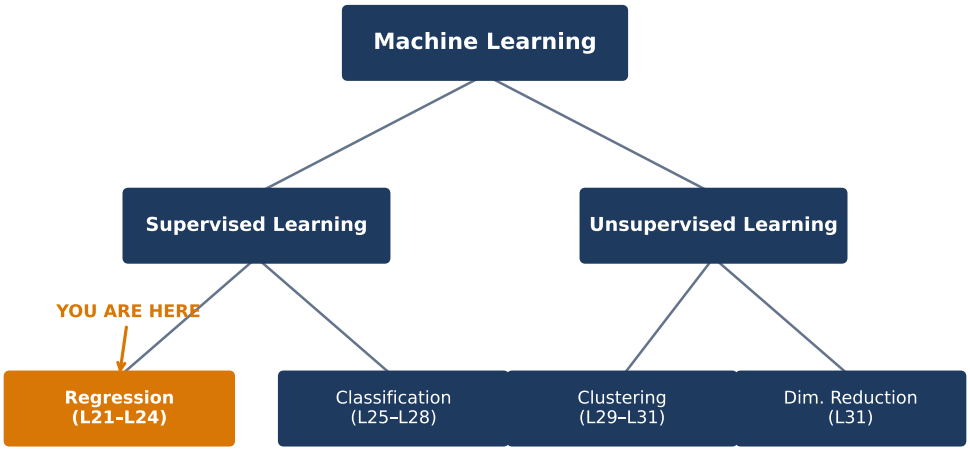


Figure 2: Regression predicts continuous values; classification predicts discrete categories. The same algorithms (trees, neural networks, boosting) often have both regression and classification variants.

The Methods Map

By the end of this handout you will meet a menagerie of supervised learning algorithms. They are not random; they form a progression that starts simple and adds complexity one idea at a time.

1. **Linear regression** (Section 2): the simplest regressor. Assumes a straight-line relationship, fits coefficients by minimizing squared error.
2. **Regularized regression—Ridge, Lasso, ElasticNet** (Section 3): penalize large coefficients to prevent overfitting and select features.
3. **Logistic regression** (Section 5): linear regression for classification, with a sigmoid squashing function.
4. **Decision trees** (Section 6): partition the feature space with axis-aligned splits. Interpretable but unstable.
5. **Random forests and gradient boosting** (Section 6): ensemble methods that combine many trees. The state of the art on most tabular data.
6. **Support vector machines, K-nearest neighbors, Naive Bayes** (Section 7): the remaining classical tools, each with its own strengths.

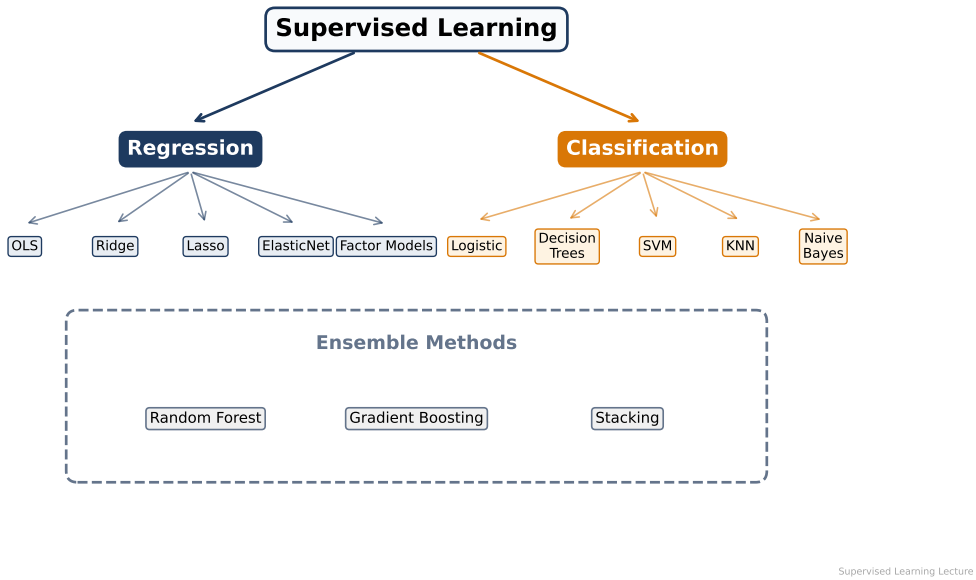
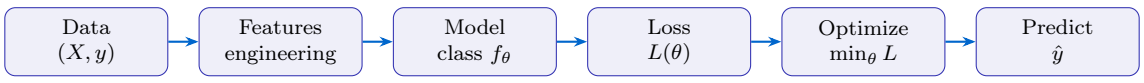


Figure 3: The supervised learning methods map. Each branch adds a new idea: linear to nonlinear, dense to sparse, single model to ensemble.

The Learning Paradigm

Every supervised learning algorithm follows the same five-step choreography:



The same five steps, over and over, for every algorithm in this handout. What changes is the model class, the loss function, and the optimization algorithm.

Train/Test Splits and Generalization

Here is the one rule you must never break: *do not evaluate your model on the same data you used to train it.* If you train a model on 500 rows and ask how well it predicts those 500 rows, you will get a score that tells you nothing about future performance. A model with enough capacity (a deep tree, a many-feature linear model) can memorize any training set exactly. A memoriser gets 100% on training data and 0% on new data.

The remedy is to hold out a *test set*—typically 20% of the data—that the model never sees during training. You fit the model on the training 80% and evaluate on the held-out 20%. The test-set score is your estimate of how the model will perform on unseen data, which is what you actually care about.

Training set: The portion of the data used to fit the model parameters. Typically 70–80% of the observations.

Test set: A held-out portion of the data used only to estimate generalization performance. Typically 20–30% of the observations. The test set must not influence model fitting or hyperparameter choice.

Generalization: The ability of a model to make accurate predictions on data not seen during training. The whole point of supervised learning is generalization; the training error is just a means to this end.

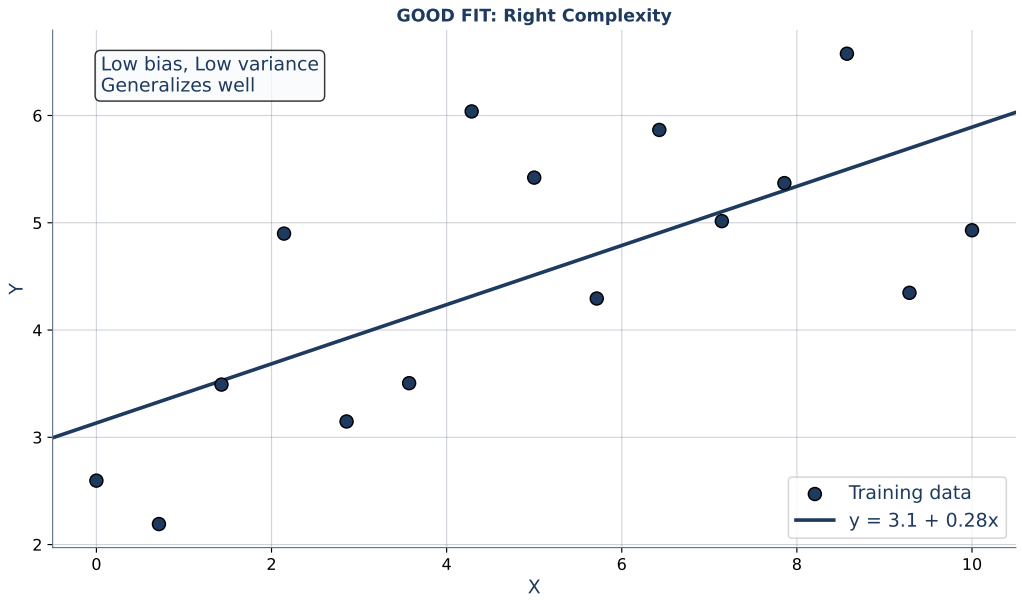


Figure 4: A good fit generalizes. The model captures the signal (trend) without chasing the noise (individual points). Both training and test error are low.

Generalization is harder than it sounds because real data is contaminated with noise. Every dataset contains some random variation that no model should reproduce—if you try, you end up memorizing noise instead of signal. A thousand-parameter model on a hundred-row dataset will always fit the training set perfectly but fail catastrophically on new data. The rest of this handout is, to a first approximation, a sequence of techniques for keeping this tension in balance.

Independence, i.i.d., and Why Finance Breaks the Rules

Most ML theory assumes that training examples are *independent and identically distributed* (i.i.d.): each row was drawn from the same underlying distribution, independently of every other row. The i.i.d. assumption lets us treat the training set as a random sample and the test set as a fresh draw from the same population. Under i.i.d., test-set performance is an unbiased estimate of real-world performance.

Financial data violates i.i.d. in two important ways. First, observations are *time-ordered*: today's return is correlated with yesterday's return (autocorrelation), so rows are not independent.

Second, the distribution *drifts*: the joint distribution of returns, volatility, and valuation ratios in 2008 is not the same as in 2022. A model trained on 2015–2020 and tested on 2023 faces a genuinely new distribution. We will return to this problem in Section 4 (walk-forward validation) and Section 8 (production concerns).

Definition: Supervised Learning Problem

Given a dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$ with $x_i \in \mathbb{R}^p$ and $y_i \in \mathcal{Y}$ (either $\mathbb{R}$ for regression or a finite set for classification), the **supervised learning problem** is:

- **Assume** that (x_i, y_i) are i.i.d. draws from an unknown joint distribution $P(X, Y)$.
- **Choose** a hypothesis class $\mathcal{F}$ (e.g., all linear functions, all depth-5 trees, all 3-layer neural networks).
- **Find** $\hat{f} \in \mathcal{F}$ that minimizes empirical loss $\sum_i L(y_i, f(x_i))$ on the training data.
- **Evaluate** $\hat{f}$ on held-out data to estimate true generalization loss $\mathbb{E}_{(X,Y)\sim P}[L(Y, \hat{f}(X))]$.

The *training error* is $\frac{1}{n}\sum_i L(y_i, \hat{f}(x_i))$; the *generalization error* is the expectation above. Good ML minimizes the latter, not the former.

Common Misconceptions about Supervised Learning

(1) "More features always help." Adding irrelevant features increases variance without decreasing bias. A model with 50 noise features and 5 real features is almost always worse than a model with just 5 real features.
(2) "High training accuracy is what matters." Training accuracy only measures how well you memorised the training data. You care about *test* accuracy—performance on data the model has never seen.
(3) "If my model works on the training set it will work in production." Training-set performance is an upper bound on generalization, not an estimate of it. The gap between train and test error is often large, especially for complex models.
(4) "i.i.d. is a harmless assumption." For finance, it is the opposite: it is almost always violated. Treat any ML workflow on time-series data with suspicion unless it uses walk-forward validation.

Worked Examples

Worked Example 1: Supervised vs. Unsupervised in a Bank

A retail bank has two tasks on its data-science roadmap.

Task A – Credit default prediction. For every past loan the bank knows whether the borrower defaulted ($y = 1$) or not ($y = 0$). The features include debt-to-income ratio, credit score, employment length, loan amount. The goal is to predict default for a new applicant. Because the historical data has labels, this is **supervised** learning—specifically, binary classification.

Task B – Customer segmentation. The marketing team wants to split 100,000 customers into groups for targeted campaigns. They have demographic and transaction features but no "correct" segment labels—no one has pre-defined the segments. This is **unsupervised** learning (clustering), because there are no labels to predict.

The key question in both cases is "do we have a target column?" If yes, supervised. If no, unsupervised. The same feature matrix X can appear in both settings; the difference is whether a y is attached.

Worked Example 2: A Train/Test Split in Numbers

Suppose you have 1000 house-price records with features (square footage, bedrooms, ZIP code) and label (price). You do an 80/20 split: 800 rows for training, 200 for testing.

Scenario A. You fit a small linear regression. Training RMSE is \$45,000 and test RMSE is \$47,000. The gap is small; the model generalizes well.

Scenario B. You fit a 20-layer neural network with 10,000 parameters on the same 800 training rows. Training RMSE drops to \$5,000 but test RMSE is \$80,000. The network memorised the training set (tiny train error) but performs terribly on new houses (huge test error). This is **overfitting**.

Scenario C. You fit a model so simple that it only uses the mean price. Training RMSE is \$120,000 and test RMSE is \$118,000. Both errors are large and close to each other. This is **underfitting**: the model is too simple to capture real structure.

The art of supervised learning is finding a model class complex enough to capture signal but not so complex that it captures noise. Sections 2 through 8 develop this art systematically.

Historical Background: Carl Friedrich Gauss and the Invention of Least Squares (1809)

The first supervised learning algorithm predates the term "machine learning" by 150 years. In 1809, the German mathematician Carl Friedrich Gauss published *Theoria Motus Corporum Coelestium in Sectionibus Conicis Solem Ambientium* ("Theory of the motion of the heavenly bodies moving about the sun in conic sections"). In it he described the method of least squares: given a set of observations of a celestial body's position, find the orbital parameters that minimize the sum of squared errors between predicted and observed positions.

Gauss claimed he had used the method as early as 1795, though the French mathematician Adrien-Marie Legendre published it independently in 1805. A priority dispute followed—Gauss was right that he had used it earlier, but Legendre published first. The modern consensus is that both should share credit.

Least squares is a supervised learning algorithm in every modern sense: it takes labeled data (x_i, y_i) (observations and times), chooses a hypothesis class (Keplerian orbits), defines a loss function (sum of squared residuals), and finds the parameters that minimize the loss. Two centuries later, Gauss's machinery still powers linear regression, logistic regression, the entire regularization family, and the optimization loops inside neural networks. When you write `LinearRegression().fit(X, y)` in scikit-learn, you are using Gauss's 1809 method with 2024 computing power.

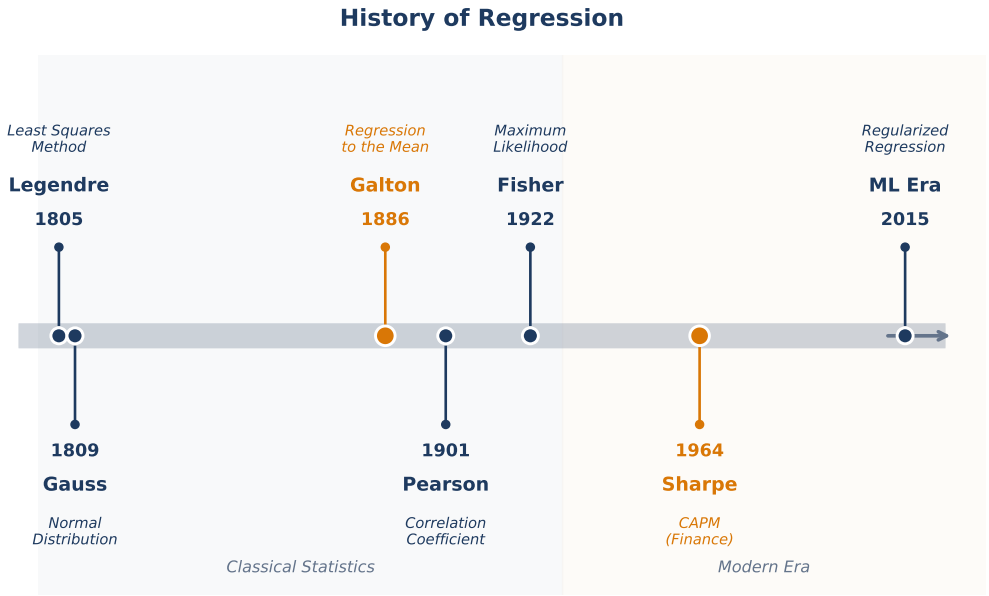


Figure 5: A brief history of supervised learning. From Gauss (1809) through Fisher (1936), Rosenblatt (1958), Breiman (1984), and the deep learning revolution of the 2010s.

Python Quickstart

Minimal Supervised Learning Pipeline in scikit-learn

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import numpy as np

# X is (n_samples, n_features), y is (n_samples,)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

model = LinearRegression()
model.fit(X_train, y_train)

y_pred_train = model.predict(X_train)
y_pred_test  = model.predict(X_test)

rmse_train = np.sqrt(mean_squared_error(y_train, y_pred_train))
rmse_test  = np.sqrt(mean_squared_error(y_test,  y_pred_test))
print(f'Train RMSE = {rmse_train:.3f},  Test RMSE = {rmse_test:.3
    f}')
```

Every supervised learning pipeline in this handout is a variation on these nine lines. Change `LinearRegression()` to `RandomForestClassifier()` and you have a forest classifier. Change the metric to `roc_auc_score` and you have classification evaluation. Add a `StandardScaler` and a `Pipeline` and you have a leakage-safe workflow. The ritual is always: split, fit, predict, score.

Classification Variant (same structure)

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, roc_auc_score

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

clf = LogisticRegression(max_iter=1000)
clf.fit(X_train, y_train)

y_pred  = clf.predict(X_test)
y_proba = clf.predict_proba(X_test)[:, 1]

print(f'Accuracy = {accuracy_score(y_test, y_pred):.3f}')
print(f'AUC      = {roc_auc_score(y_test, y_proba):.3f}')
```

Problem 1.1 (Easy) ⋆

Classify each of the following as regression or classification: (a) predicting a stock's next-month return; (b) predicting whether a loan will default; (c) predicting tomorrow's S&P 500 closing price; (d) classifying emails as spam or not; (e) predicting a patient's blood-glucose level in two hours.
Solution: see Appendix.

Problem 1.2 (Medium) ⋆⋆

You have ten years of daily returns for the S&P 500 (roughly 2,500 trading days). You want to build a model that predicts tomorrow's return from the last 20 days of returns. Design an appropriate train/test split. Why is a random 80/20 split the wrong choice here? What split would you use instead, and why?
Solution: see Appendix.

Problem 1.3 (Medium) ⋆⋆

Explain why the i.i.d. assumption is violated for financial return data. Give two concrete examples of how this violation could inflate a naïve train/test evaluation. How would you redesign the evaluation to diagnose each problem?
Solution: see Appendix.

Problem 1.4 (Hard) ⋆⋆⋆

Derive the **bias-variance decomposition** from first principles. Let $y = f(x) + \epsilon$ where $\mathbb{E}[\epsilon] = 0$ and $\mathrm{Var}(\epsilon) = \sigma^2$. For an estimator $\hat{f}(x)$ trained on a random sample, show that the expected squared prediction error at a fixed x decomposes as

$$\mathbb{E}\Big[(y - \hat{f}(x))^2\Big] = \underbrace{(\mathbb{E}[\hat{f}(x)] - f(x))^2}_{\text{Bias}^2} + \underbrace{\mathrm{Var}(\hat{f}(x))}_{\text{Variance}} + \sigma^2.$$

Expectations are over both the randomness in the training sample and the noise ϵ.
Solution: see Appendix.

Problem 1.5 (Hard) ⋆⋆⋆

You have a dataset with 10,000 rows and 500 features. You plan to tune hyperparameters by grid search and report final performance. Design a complete validation strategy that (a) produces an unbiased estimate of production performance, (b) uses cross-validation to tune hyperparameters, and (c) prevents any form of data leakage. Draw a schematic showing which data is used at each step.
Solution: see Appendix.

Connecting Forward

We now know what supervised learning is: learning a function $f : X \to y$ from labeled examples, with generalization as the true goal. We have seen the two sub-families (regression and classification), the five-step learning paradigm (data, features, model, loss, optimization), and the one discipline that separates rigorous ML from theatre (train/test splits and honest evaluation).

What we have not yet done is fit a single model. Section 2 corrects that. It introduces linear

regression—the humblest regressor, the oldest learning algorithm in the book, and the foundation for almost everything that follows. We will derive its closed-form solution from pure calculus, work through a small numerical example by hand, and apply it to two finance cornerstones: the Capital Asset Pricing Model (CAPM) and the Fama-French three-factor model.

Key Takeaway: Supervised learning turns labeled historical data into predictions about new observations; a train/test split and a clear generalization goal are the two non-negotiable ingredients.

2. The Starting Point – OLS and Factor Models

Opening Problem: Monthly Returns from Three Features

You are a junior quant at a long-only equity fund. Your boss hands you a spreadsheet: for each of 200 US stocks, monthly returns over 60 months, plus three features per stock per month—price-to-earnings ratio, 12-month price momentum, and realized volatility. She wants a single sentence answer: "Given the P/E, momentum, and volatility at the end of a month, what do you predict the return will be during the next month?"

You could squint at scatter plots forever. You could build a twenty-line machine learning pipeline. But the fastest, most interpretable, and often most competitive starting point is the oldest supervised learning algorithm: ordinary least squares (OLS) linear regression. Fit three coefficients—one for each feature—and an intercept, and you have a model you can explain to the investment committee in one slide.

This section derives OLS from first principles, walks through a numerical example by hand, states the LINE assumptions, and connects the math to two famous finance applications: the CAPM single-factor model and the Fama-French three-factor model.

Discovery Question

Two features—P/E and momentum—each have a correlation of 0.2 with next-month returns. Does a linear model using *both* features have a correlation close to 0.4 with returns, or close to 0.2? Why? What if the two features are themselves perfectly correlated with each other?

The Linear Regression Model

The simplest relationship between two variables is a straight line. Given one feature x and a label y, a linear regression model posits

$$\hat{y} = \beta_0 + \beta_1 x,$$

where $\hat{y}$ is the predicted value, β_0 is the intercept (the predicted y when $x = 0$), and β_1 is the slope (the change in predicted y per unit change in x). With p features the model generalizes to

$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p = \beta_0 + \boldsymbol{\beta}^\top x,$$

where $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_p)$ is the coefficient vector. The model has $p + 1$ parameters: p slopes and one intercept.

Linear regression: A supervised learning model that predicts a continuous target as an affine function of the features: $\hat{y} = \beta_0 + \beta^\top x$. Fitting the model means choosing coefficients β to minimize the sum of squared residuals on the training data.

Residual: The difference between the observed label and the predicted value for a given data point: $e_i = y_i - \hat{y}_i$. Residuals are the raw material of regression diagnostics.

The OLS Objective

Ordinary least squares (OLS) picks the coefficients that minimize the sum of squared residuals:

$$L(\beta) = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{n} \left(y_i - \beta_0 - \beta^\top x_i\right)^2.$$

Why squared residuals and not absolute residuals? Three answers. First, squaring makes L differentiable everywhere, which allows closed-form solutions. Second, squared loss corresponds

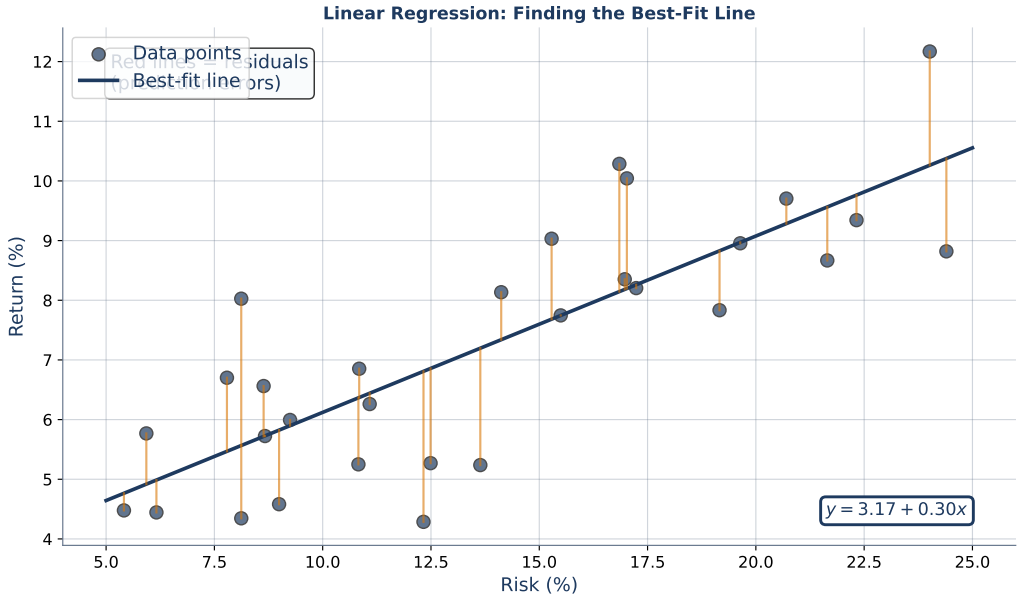


Figure 6: The linear regression concept. Data points form a cloud; the line $\hat{y} = \beta_0 + \beta_1 x$ threads through the cloud so that the vertical distances from points to line are as small as possible.

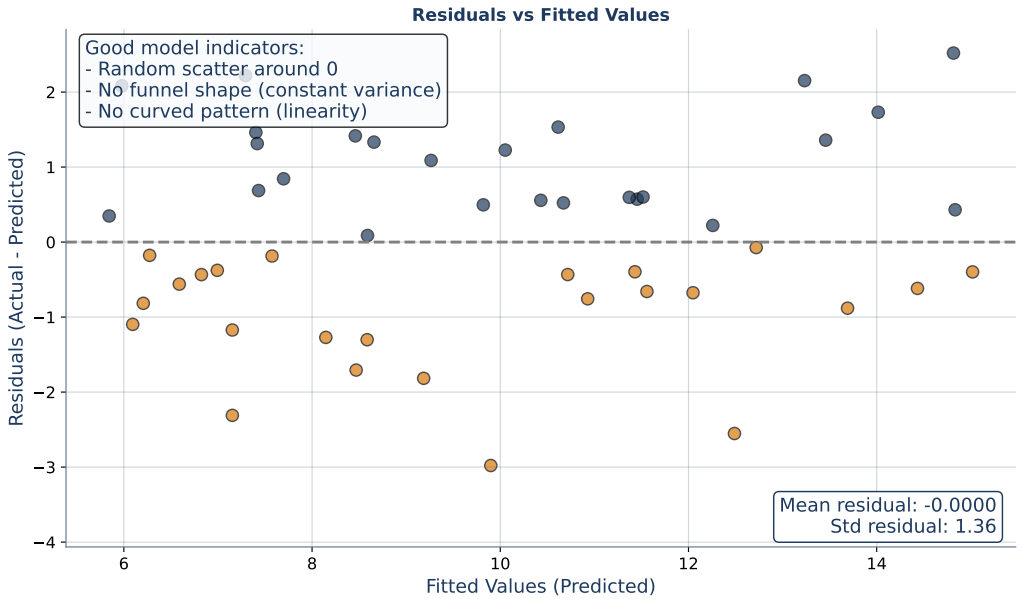


Figure 7: Residuals as vertical distances. Each orange segment measures how far a data point lies from the fitted line. OLS picks the line that minimizes the sum of squared lengths of these segments.

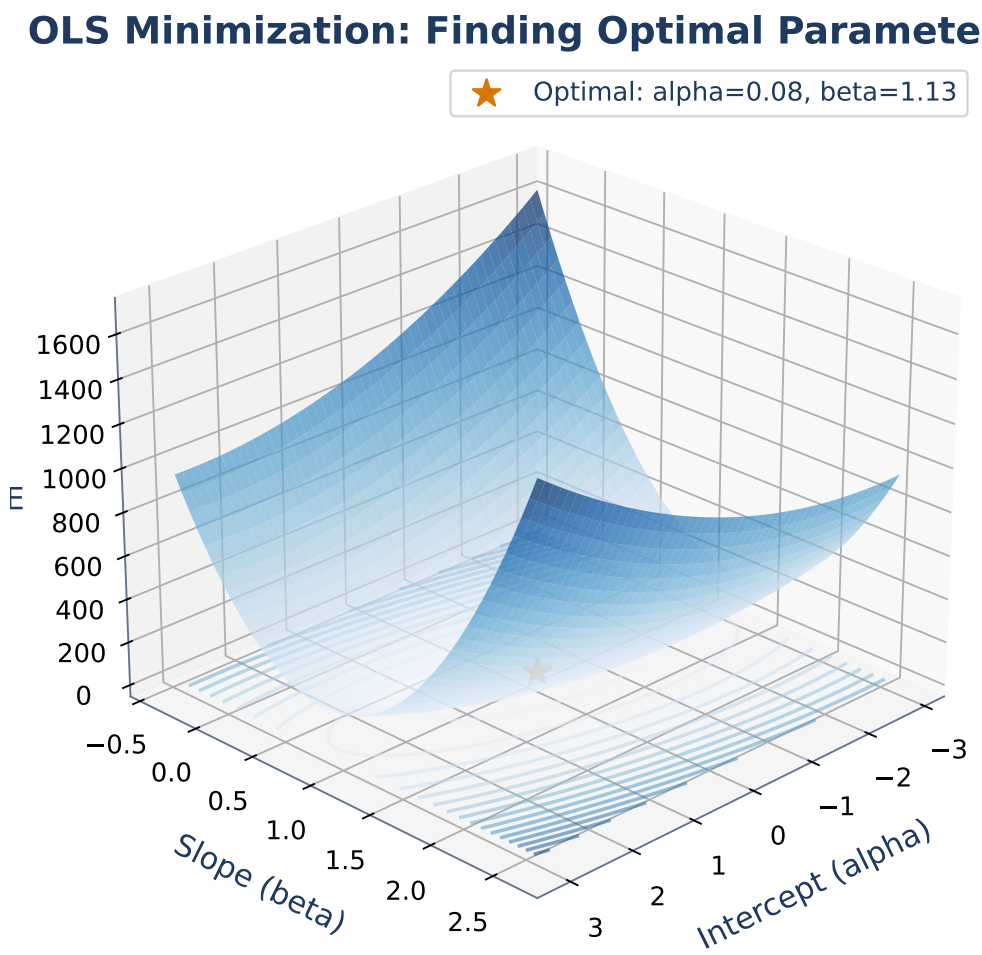


Figure 8: OLS minimizes the sum of squared residuals. The fitted line is chosen so that the total squared vertical distance from points to line is as small as possible.

to the maximum-likelihood estimate under Gaussian noise (more on this below). Third, historical inertia: Gauss picked squared loss in 1809, and it has been the default ever since.

Key Formula: OLS Closed-Form Solution

Let $X \in \mathbb{R}^{n\times(p+1)}$ be the feature matrix with a column of ones prepended (for the intercept), and let $y \in \mathbb{R}^n$ be the label vector. The OLS coefficient vector is

$$\hat{\beta} = (X^\top X)^{-1} X^\top y,$$

provided $X^\top X$ is invertible. The prediction for a new input x_{new} (also with a 1 prepended) is $\hat{y} = x_{\text{new}}^\top \hat{\beta}$.
Plain English: form $X^\top X$ (a small $(p+1)\times(p+1)$ matrix), invert it, multiply by $X^\top y$. The whole fit is one line of matrix algebra.

Deriving the Closed-Form Solution

The OLS solution is one of the few machine-learning results where you can see every step. Write the loss in matrix form:

$$L(\beta) = \|y - X\beta\|^2 = (y - X\beta)^\top (y - X\beta) = y^\top y - 2y^\top X\beta + \beta^\top X^\top X\beta.$$

Take the gradient with respect to β:

$$\nabla_\beta L = -2X^\top y + 2X^\top X\beta.$$

Set the gradient to zero:

$$X^\top X\beta = X^\top y \implies \hat{\beta} = (X^\top X)^{-1} X^\top y.$$

This is the *normal equation*. To check that it gives a minimum and not a maximum, compute the Hessian: $\nabla^2 L = 2X^\top X$, which is positive semi-definite for any X. If $X^\top X$ is invertible then the Hessian is positive definite and the critical point is a unique minimum. The OLS estimator is the global minimizer of the squared-residual loss.

Geometric Interpretation

OLS has a beautiful geometric interpretation. Think of y as a vector in $\mathbb{R}^n$ (one component per observation). The column space of X is the set of all vectors expressible as linear combinations of the columns of X; it is a $(p+1)$-dimensional subspace of $\mathbb{R}^n$. OLS finds the vector $\hat{y} = X\hat{\beta}$ in this column space that is closest to y in Euclidean distance. The residual vector $e = y - \hat{y}$ is orthogonal to every column of X; this is exactly the statement $X^\top e = 0$, which is equivalent to the normal equation.

Geometrically, $\hat{y}$ is the orthogonal projection of y onto the column space of X. You can verify this with a ruler: in any OLS fit, the residual vector is perpendicular to every feature column. This perpendicularity is the reason squared error is the right loss: the Pythagorean theorem says that when the residual is perpendicular to the fit, you have the shortest residual, hence the best least-squares fit.

A Numerical Example by Hand

Let's work an OLS problem from start to finish with a tiny dataset. Suppose we have five observations of monthly return (y) predicted from lagged momentum (x):

x (momentum)	y (next-month return)
-0.2	-0.03
-0.1	-0.01
0.0	0.00
0.1	0.02
0.2	0.03

Compute the OLS estimates step by step. Means: $\bar{x} = 0$, $\bar{y} = 0.002$. The slope estimator is

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}.$$

Numerator: $(-0.2)(-0.032) + (-0.1)(-0.012) + (0)(-0.002) + (0.1)(0.018) + (0.2)(0.028) = 0.0064 + 0.0012 + 0 + 0.0018 + 0.0056 = 0.015$. Denominator: $0.04 + 0.01 + 0 + 0.01 + 0.04 = 0.10$. So $\hat{\beta}_1 = 0.015/0.10 = 0.15$.

Intercept: $\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x} = 0.002 - 0.15 \cdot 0 = 0.002$.

Fitted line: $\hat{y} = 0.002 + 0.15x$. For next-month momentum of $+0.15$, predicted return is $0.002 + 0.15 \cdot 0.15 = 0.0245$. For negative momentum of -0.1, predicted return is $0.002 - 0.015 = -0.013$. The slope of 0.15 means: *a one-unit increase in lagged momentum predicts a 0.15-unit increase in next-month return.* In the actual financial scale (returns are small), a 10% momentum increase predicts a 1.5% return increase.

Interpreting Coefficients

Each coefficient β_j has a precise meaning: *the expected change in y when x_j increases by one unit, holding all other features constant.* The "holding all other features constant" clause is the distinguishing feature of multiple regression; it is what separates a regression coefficient from a raw correlation.

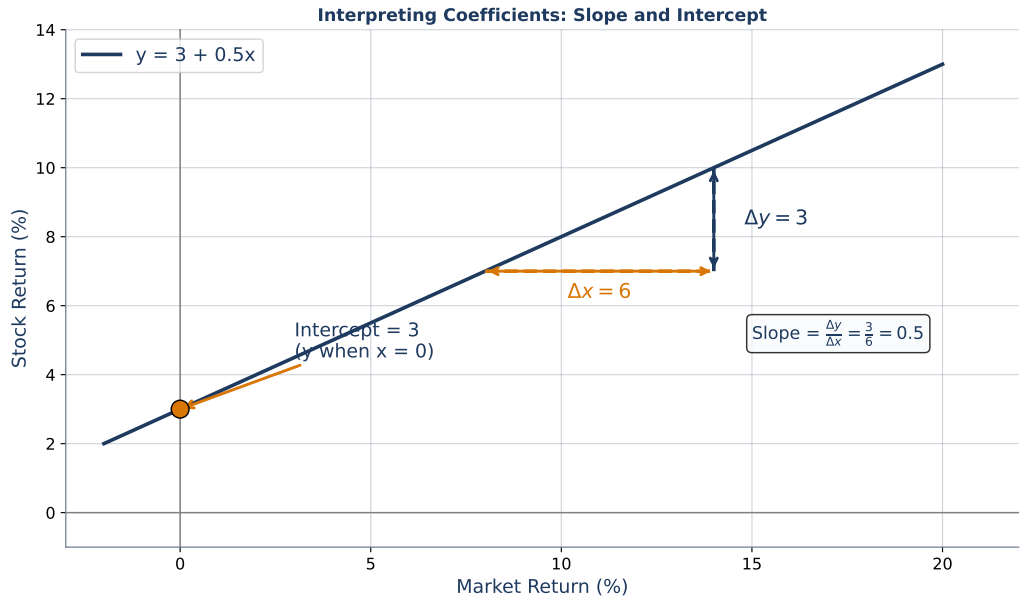


Figure 9: Coefficient interpretation. Each slope β_j is the change in the predicted label per unit change in x_j, with all other features held fixed. The size and sign of the coefficient encode both direction and strength.

When features are *uncorrelated* with each other, multiple regression coefficients coincide with the univariate slopes you would get by fitting each feature separately. When features are correlated (which is almost always the case in finance), the multiple regression coefficients can differ

substantially from univariate slopes. This is the "confounding" effect: a univariate regression of returns on momentum attributes all explanatory power to momentum, even if some of that power actually belongs to volatility (which is correlated with momentum).

The LINE Assumptions

OLS is the *best linear unbiased estimator* (BLUE)—meaning among all unbiased estimators that are linear functions of y, OLS has the smallest variance—*under four assumptions*. The assumptions are called LINE:

1. **Linearity** (**L**): The true relationship between features and label is linear: $y = \beta_0 + \beta^\top x + \epsilon$ with a linear systematic part.
2. **Independence** (**I**): The errors ϵ_i are independent across observations.
3. **Normality** (**N**): The errors ϵ_i are normally distributed. (Strictly needed only for confidence intervals and hypothesis tests, not for point estimates.)
4. **Equal variance** (**E**, also called *homoscedasticity*): The variance of ϵ_i is the same for all observations: $\mathrm{Var}(\epsilon_i) = \sigma^2$ regardless of x.

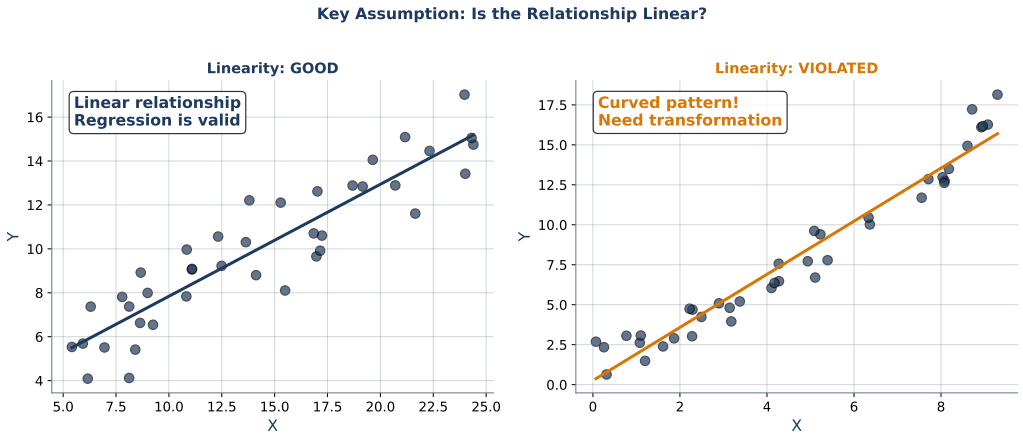


Figure 10: The four LINE assumptions underlying OLS: linearity, independence, normality, equal variance. Each can be checked with a specific diagnostic plot.

The Gauss-Markov Theorem (Proof Sketch)

Under assumptions L, I, and E (normality is not needed for this result), the **Gauss-Markov theorem** states that OLS is the best linear unbiased estimator (BLUE): among all estimators of β that are linear in y and unbiased, the OLS estimator has the smallest variance.

Proof sketch. Any linear estimator of β can be written as $\tilde{\beta} = Ay$ for some $(p+1) \times n$ matrix A. Unbiasedness requires $\mathbb{E}[\tilde{\beta}] = \beta$, which gives $AX = I$. Write $A = (X^\top X)^{-1} X^\top + B$ for some B; unbiasedness gives $BX = 0$. The variance is

$$\mathrm{Var}(\tilde{\beta}) = \sigma^2 AA^\top = \sigma^2 \left[(X^\top X)^{-1} + BB^\top \right].$$

The OLS estimator corresponds to $B = 0$ and has variance $\sigma^2 (X^\top X)^{-1}$. Any other linear unbiased estimator has variance $\sigma^2 (X^\top X)^{-1} + \sigma^2 BB^\top$, which is larger by the positive semi-definite matrix $\sigma^2 BB^\top$. Thus OLS wins in the BLUE sense. □

The Capital Asset Pricing Model (CAPM)

The first great finance application of OLS is the Capital Asset Pricing Model, introduced by William Sharpe (1964), John Lintner (1965), and Jan Mossin (1966). CAPM says that the excess return of any asset equals a risk-free return premium proportional to the asset's sensitivity to the market:

$$R_i - R_f = \alpha_i + \beta_i(R_m - R_f) + \epsilon_i.$$

Here R_i is the return on asset i, R_f is the risk-free rate, R_m is the return on the market portfolio. The slope β_i (the CAPM beta) measures how much asset i moves per unit of market move. The intercept α_i (the CAPM alpha) measures the "free lunch"—excess return not explained by market exposure. Under the efficient-market hypothesis, α_i should be zero; any positive α_i is evidence of skill or mispricing.

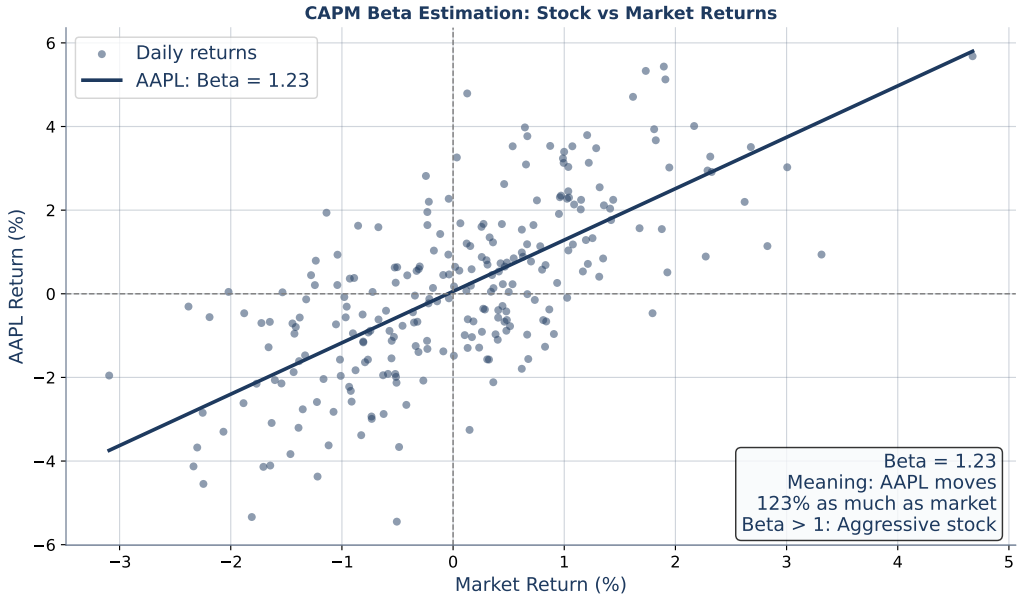


Figure 11: CAPM as a regression. The slope of a scatter plot of (excess stock return vs. excess market return) is the CAPM beta. The intercept is alpha.

Estimating CAPM is pure OLS: regress the asset's excess returns on the market's excess returns. A beta of 1 means the asset moves with the market; a beta of 1.5 means it amplifies market moves (aggressive); a beta of 0.5 means it dampens market moves (defensive). Utilities typically have betas around 0.5; growth tech stocks often have betas above 1.5.

Fama-French Three-Factor Model

In 1993, Eugene Fama and Kenneth French extended CAPM with two additional factors: SMB (Small Minus Big—the return difference between small-cap and large-cap stocks) and HML (High Minus Low—the return difference between value and growth stocks). The three-factor model is

$$R_i - R_f = \alpha_i + \beta_{M,i}(R_m - R_f) + \beta_{S,i}\,\mathrm{SMB} + \beta_{H,i}\,\mathrm{HML} + \epsilon_i.$$

This is a multiple regression with three features. The coefficients tell you the asset's exposure to three risk factors: market movements (β_M), size exposure (β_S), and value exposure (β_H). Fitting is again pure OLS: build a feature matrix with columns for $R_m - R_f$, SMB, and HML, and solve the normal equation.

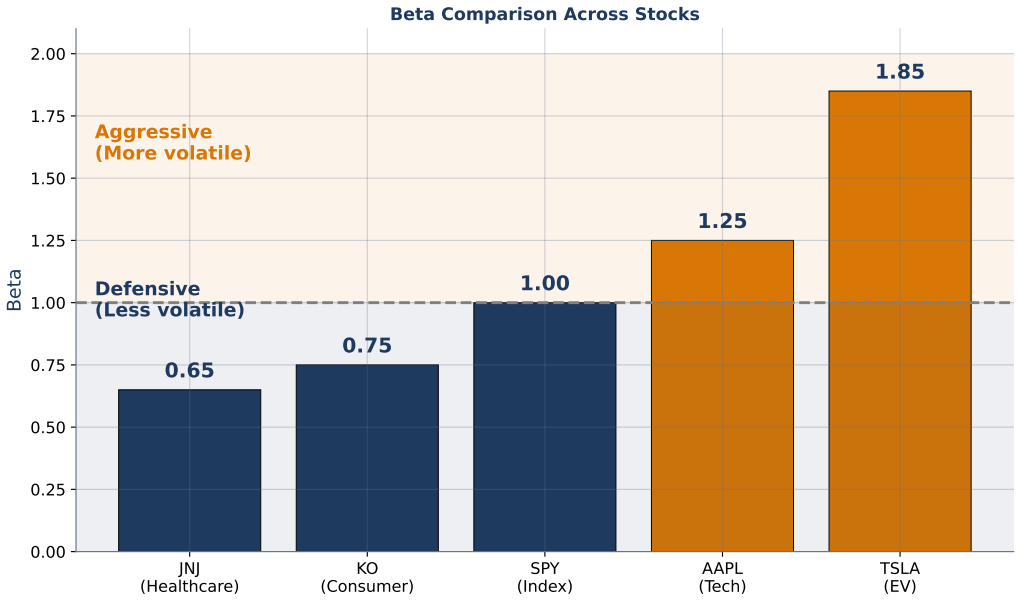


Figure 12: Three stocks with betas of 0.5, 1.0, and 1.5. The steeper the regression line, the more the stock amplifies market moves.

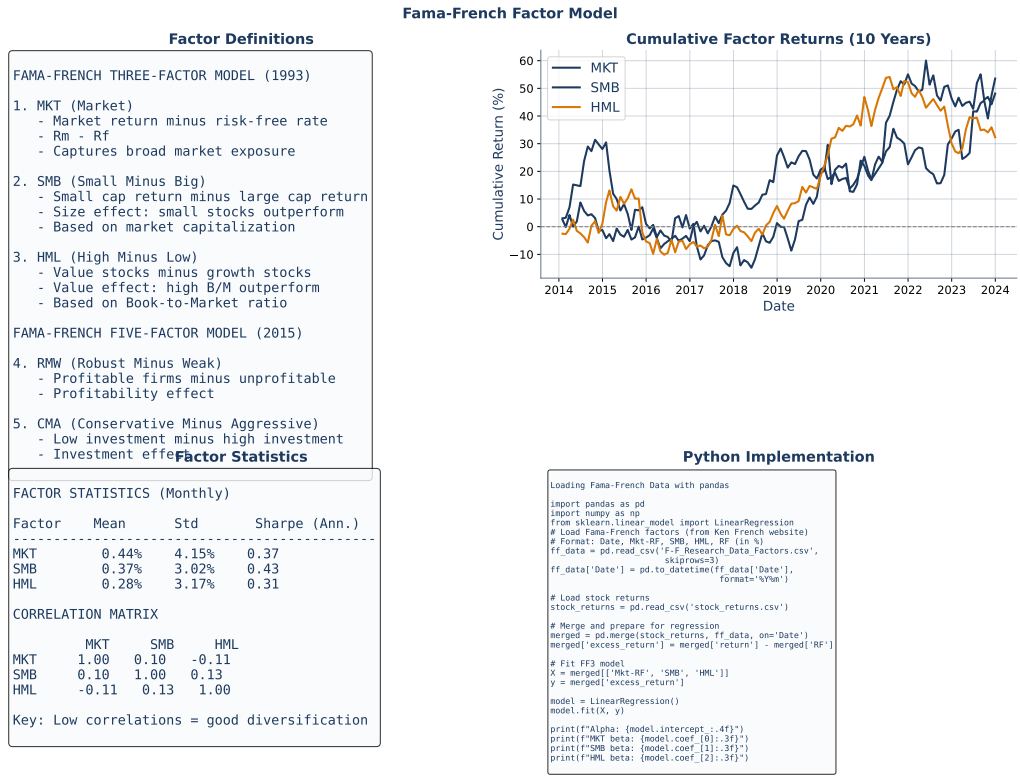


Figure 13: Fama-French three-factor model: market, size (SMB), and value (HML) factors explain the bulk of cross-sectional variation in stock returns—far more than CAPM alone.

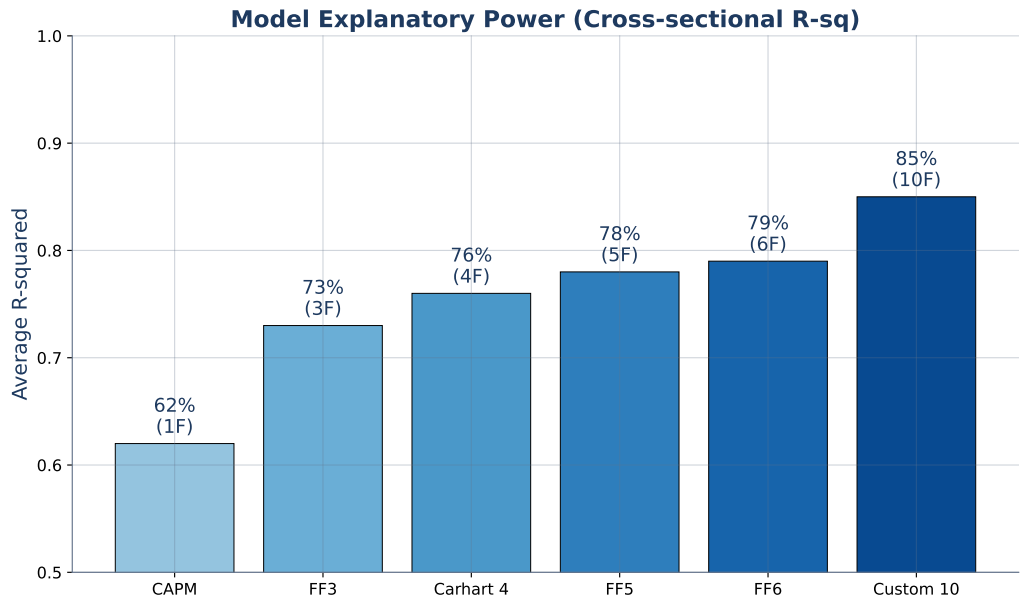


Figure 14: R^2 comparison: CAPM (one factor) versus Fama-French (three factors). Adding SMB and HML substantially increases explanatory power.

Worked Example: Apple's Beta by Hand

Worked Example: Estimating AAPL's CAPM Beta

Suppose we have 60 months of data. The market excess return $R_m - R_f$ has sample mean 0.009 (monthly) and sample variance 0.0022. Apple's excess return has sample mean 0.014, and the sample covariance between Apple's excess return and market excess return is 0.0027.

OLS beta: $\hat{\beta} = \text{Cov}(R_{AAPL}, R_m)/\text{Var}(R_m) = 0.0027/0.0022 = 1.227$. OLS alpha: $\hat{\alpha} = \bar{R}_{AAPL} - \hat{\beta}\,\bar{R}_m = 0.014 - 1.227 \cdot 0.009 = 0.014 - 0.01104 = 0.00296$.

So AAPL has a CAPM beta of ≈ 1.23 (moderately aggressive) and an alpha of $\approx 0.30\%$ per month. The alpha is positive, suggesting AAPL outperformed what CAPM alone would predict over this window. Of course, this simple single-factor fit ignores size, value, momentum, and other risk exposures; a Fama-French three-factor fit would almost certainly reduce the alpha closer to zero.

Worked Example: A Two-Feature OLS Fit

Predict monthly return y from momentum x_1 and volatility x_2 with 4 observations.

x_1	x_2	y
0.1	0.02	0.03
0.2	0.03	0.05
-0.1	0.02	-0.01
0.0	0.04	0.00

Build X (with intercept column) and y:

$$X = \begin{pmatrix} 1 & 0.1 & 0.02 \\ 1 & 0.2 & 0.03 \\ 1 & -0.1 & 0.02 \\ 1 & 0 & 0.04 \end{pmatrix}, \qquad y = \begin{pmatrix} 0.03 \\ 0.05 \\ -0.01 \\ 0.00 \end{pmatrix}.$$

Compute $X^\top X$ (a 3×3 matrix) and $X^\top y$ (a 3-vector). The normal equation $\hat{\beta} = (X^\top X)^{-1} X^\top y$ yields approximately $\hat{\beta} \approx (0.021, 0.20, -0.55)^\top$. Interpretation: a 1-unit increase in momentum raises the predicted return by 0.20, holding volatility fixed; a 1-unit increase in volatility reduces the predicted return by 0.55, holding momentum fixed.
In practice, we read these off `model.coef_` and `model.intercept_` after running `LinearRegression().fit(X, y)`. The point of the hand calculation is to reinforce what the black box is actually doing.

When OLS Goes Wrong

OLS is powerful but brittle. Four failure modes are worth naming explicitly.

- **Nonlinearity.** If the true relationship bends, OLS's straight line leaves residuals with systematic patterns. Diagnostic: plot residuals versus fitted values; a random cloud is good, a curved band is bad.
- **Heteroscedasticity.** If error variance grows with the feature (common in finance, where high-volatility regimes have larger residuals), OLS estimates remain unbiased but are no longer minimum-variance. The standard errors are also wrong; p-values are unreliable.
- **Multicollinearity.** If two features are highly correlated, $X^\top X$ is nearly singular and the inversion in $(X^\top X)^{-1}$ becomes numerically unstable. Coefficients become erratic; a small change in the data can flip a coefficient's sign.
- **Outliers.** One extreme point can pull the fitted line far from the bulk of the data because squared loss penalizes outliers quadratically. Robust regression methods (Huber loss, quantile regression) exist for this case.

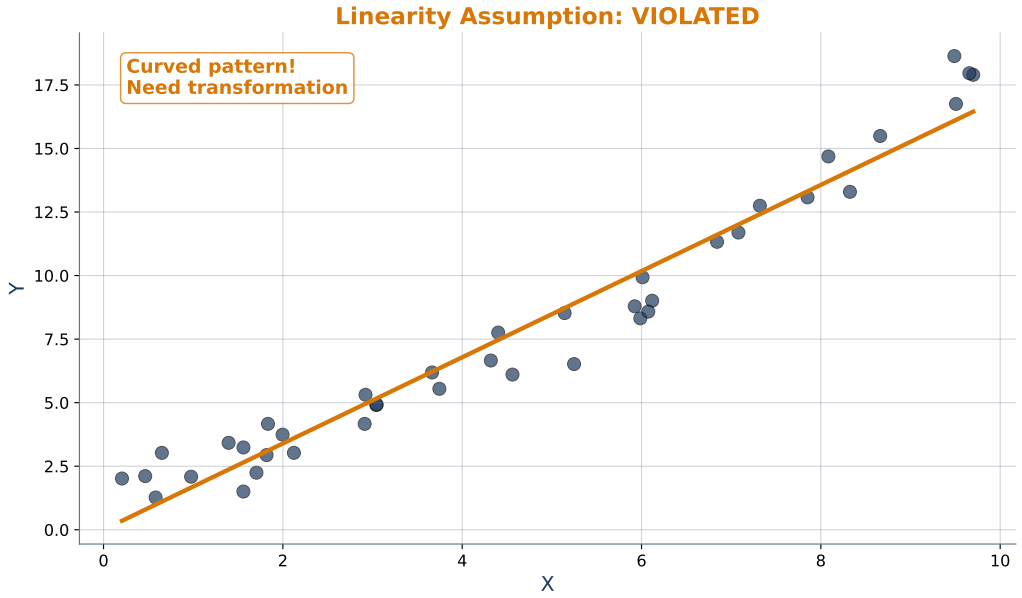


Figure 15: Linearity violated. When the true relationship is curved, a straight-line fit produces residuals with systematic structure—a sign that the model class is wrong.

Definition: Ordinary Least Squares (OLS)

Ordinary least squares is the supervised learning algorithm that fits a linear regression model by minimizing the sum of squared residuals:

$$\hat{\beta} = \arg\min_{\beta} \sum_{i=1}^{n} (y_i - \beta^\top x_i)^2 = (X^\top X)^{-1} X^\top y.$$

Under the LINE assumptions (linearity, independence, normality of errors, equal variance), OLS is unbiased, consistent, and efficient (BLUE). When assumptions fail, OLS estimates remain unbiased but lose their efficiency guarantee and the standard errors become unreliable.

Common Misconceptions about Linear Regression

(1) "A high R^2 means the model is correct." A high R^2 only means the model explains a lot of variance in the training data. It does not guarantee correct functional form or correct causal interpretation. A quadratic relationship fitted with a line will have a lower R^2 than a quadratic fit, but the real issue is model misspecification, not R^2.
(2) "Correlation between features is a minor issue." Strong correlation (multicollinearity) makes individual coefficients unstable and uninterpretable, even when the overall prediction is fine. If you intend to interpret β_j as the "effect of x_j holding others fixed," multicollinearity can make that interpretation meaningless.
(3) "CAPM beta is a fixed property of a stock." Betas change over time. A 60-month rolling OLS will show that most betas drift substantially over the life of a stock. Use rolling or Kalman-filtered estimates if you care about the current value.

Historical Background: Gauss, Legendre, and Fama-French

The priority dispute over the method of least squares between Gauss and Legendre is one of the most famous in the history of mathematics. Legendre published the method in 1805 in an appendix on comet orbits. Gauss, in his 1809 treatise on celestial motion, claimed to have been using the method since 1795. Contemporary evidence (Gauss's lecture notes and correspondence) supports his claim, but Legendre published first. The modern tradition credits both.

The statistical theory of regression was developed by Francis Galton (1886) and Karl Pearson (1896), working on heredity: Galton noticed that tall fathers have tall but shorter sons, which he called "regression toward mediocrity"—the origin of the term "regression."

Multiple regression in finance reached its peak with Eugene Fama and Kenneth French's 1993 paper "Common Risk Factors in the Returns on Stocks and Bonds," which introduced the SMB (Small Minus Big) and HML (High Minus Low) factors. The paper extended CAPM to a three-factor model and showed that size and value exposures capture a substantial portion of cross-sectional return variation that market exposure alone misses. Fama received the Nobel Prize in Economics in 2013, in part for this work.

The three-factor model has since been extended further: the Carhart four-factor model adds momentum (1997); the Fama-French five-factor model adds profitability and investment (2015). All are multiple regressions—just longer formulas, with more coefficients to estimate via the same 1809 least-squares machinery.

Problem 2.1 (Easy) ⋆

Compute the OLS slope and intercept for the following three data points by hand: $(x_1, y_1) = (1, 2)$, $(x_2, y_2) = (2, 3)$, $(x_3, y_3) = (3, 5)$. What are the residuals? Does their sum equal zero?

Solution: see Appendix.

Problem 2.2 (Easy) ⋆

Stock A has a CAPM beta of 1.5. Stock B has a CAPM beta of 0.8. The market returns +2% in a given month. What are the expected excess returns of A and B according to CAPM (ignore alpha)? What about when the market drops −3%?

Solution: see Appendix.

Problem 2.3 (Medium) ⋆⋆

Sketch a proof of the Gauss-Markov theorem. State precisely which assumptions are needed and which are not. Show explicitly why normality is unnecessary for BLUE but becomes essential for exact confidence intervals.

Solution: see Appendix.

Problem 2.4 (Medium) ⋆⋆

You fit a linear regression and obtain the following residual-vs-fitted-value plot: residuals form a clear U-shape (large positive at low and high fitted values, negative in the middle). Which LINE assumption is violated? What would you do to fix the problem—while still using linear regression?

Solution: see Appendix.

Problem 2.5 (Hard) ⋆⋆⋆

Derive the closed-form Fama-French three-factor estimator. Given T months of data for one stock and the three factors (MKT-RF, SMB, HML), write out the feature matrix X, the label vector y, and solve $\hat{\beta} = (X^\top X)^{-1} X^\top y$. Show how the result reduces to CAPM when SMB and HML columns are zero. What happens when SMB and HML are highly correlated with MKT-RF?
Solution: see Appendix.

Connecting Forward

OLS gives us a first working supervised learning algorithm, a derivation we can follow end-to-end, and two finance applications (CAPM and Fama-French) that cover a substantial fraction of real quantitative workflows. But OLS has a weakness we have not yet addressed: in high dimensions, with correlated features, or with more features than observations, OLS becomes unstable or outright undefined. Section 3 fixes this with *regularization*: penalize large coefficients to trade a little bias for a lot less variance. Ridge regression, Lasso, and ElasticNet all emerge from the same principle, and each comes with its own geometric story about when and why to prefer one over another.

Key Takeaway: OLS fits a linear model by minimizing squared residuals; the closed-form $\hat{\beta} = (X^\top X)^{-1} X^\top y$ connects finance's CAPM and Fama-French models to the 1809 method of least squares.

3. Taming Complexity – Regularization and Bias-Variance

Opening Problem: 200 Features, 100 Observations

You work at a quantitative hedge fund. Your data scientist has pulled 200 candidate factors—macroeconomic indicators, technical signals, fundamental ratios, sentiment scores—and only 100 months of history. She hands you the feature matrix and asks you to fit a return-prediction model. You plug it into `LinearRegression().fit(X, y)`. The code crashes with a singular-matrix warning, or worse, it runs and returns wild coefficients: some factors have $\beta = 10^8$, others have $\beta = -10^8$. The predictions look like a random walk.

What went wrong? With 200 features and 100 observations, the feature matrix X is 100×200, and $X^\top X$ is 200×200—but it has rank at most 100. The inverse does not exist. Even if you add a few more rows to make $X^\top X$ technically invertible, its condition number is enormous: tiny changes in data produce massive changes in $\hat{\beta}$. OLS has failed.

The fix is *regularization*: add a penalty on coefficient size to the loss function. The penalty shrinks coefficients toward zero, stabilizes the inversion, and often improves generalization by trading a little bias for a lot less variance. This section derives Ridge regression (L2 penalty) and Lasso (L1 penalty), explains when each wins, and builds the bias-variance decomposition from first principles.

Discovery Question

Two models predict next-month stock returns. Model A uses 200 features with unregularized OLS; it has training RMSE of 0.01 and test RMSE of 0.08. Model B uses the same 200 features with regularization; it has training RMSE of 0.05 and test RMSE of 0.055. Which model is better for production? Why does a *worse* training fit sometimes produce a *better* production model?

Bias-Variance Decomposition

Before introducing regularization we need the vocabulary to say why regularization helps. That vocabulary is the *bias-variance decomposition*. Let $y = f(x) + \epsilon$ be the true data-generating process, with $\mathbb{E}[\epsilon] = 0$ and $\text{Var}(\epsilon) = \sigma^2$. Let $\hat{f}(x)$ be a prediction produced by a model trained on a random sample. Then the expected squared prediction error at a fixed test point x decomposes as:

$$\mathbb{E}\Big[(y - \hat{f}(x))^2\Big] = \underbrace{(\mathbb{E}[\hat{f}(x)] - f(x))^2}_{\text{Bias}^2} + \underbrace{\text{Var}(\hat{f}(x))}_{\text{Variance}} + \underbrace{\sigma^2}_{\text{Irreducible}}.$$

Deriving the Decomposition

Expand inside the expectation. Insert $\mathbb{E}[\hat{f}(x)]$ (a constant, so the expectation passes through):

$$\begin{aligned}(y - \hat{f})^2 &= (f + \epsilon - \hat{f})^2 \\ &= (f - \mathbb{E}[\hat{f}] + \mathbb{E}[\hat{f}] - \hat{f} + \epsilon)^2.\end{aligned}$$

Let $A = f - \mathbb{E}[\hat{f}]$ (a constant), $B = \mathbb{E}[\hat{f}] - \hat{f}$ (zero-mean random), $C = \epsilon$ (zero-mean random and independent of B). Then:

$$(A + B + C)^2 = A^2 + B^2 + C^2 + 2AB + 2AC + 2BC.$$

Take expectations. $\mathbb{E}[A^2] = A^2$ because A is constant. $\mathbb{E}[B^2] = \text{Var}(\hat{f})$ because B has zero mean. $\mathbb{E}[C^2] = \sigma^2$. The cross terms: $\mathbb{E}[2AB] = 2A\mathbb{E}[B] = 0$ because $\mathbb{E}[B] = 0$. $\mathbb{E}[2AC] = 2A\mathbb{E}[\epsilon] = 0$.

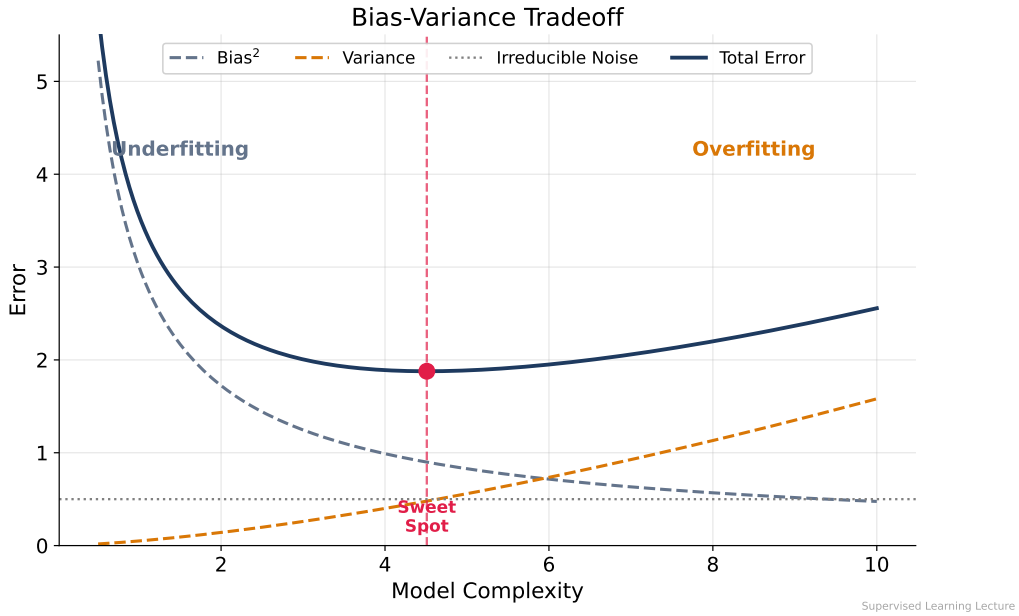


Figure 16: The bias-variance tradeoff. Simple models have high bias and low variance; complex models have low bias and high variance. Total prediction error is the sum, minimized by an intermediate complexity.

$\mathbb{E}[2BC] = 2\mathbb{E}[BC] = 0$ because B depends only on training data and $C = \epsilon$ is the noise at test time, independent of training data.

Collecting what survives:

$$\mathbb{E}[(y - \hat{f})^2] = A^2 + \mathrm{Var}(\hat{f}) + \sigma^2 = (\mathbb{E}[\hat{f}] - f)^2 + \mathrm{Var}(\hat{f}) + \sigma^2.$$

This is the decomposition. Three distinct sources of error:

- **Bias**: how far, on average, the model's predictions are from the truth. Underfitting produces high bias.
- **Variance**: how much the model's predictions jump around when we retrain on different samples. Overfitting produces high variance.
- **Irreducible**: the intrinsic noise σ^2. No amount of modeling can remove this term.

Simple models (e.g., constant prediction) have low variance but high bias. Complex models (e.g., 200-feature OLS on 100 samples) have low bias but enormous variance. The sweet spot is somewhere in between, and regularization is the tool we use to land there.

Bias: The squared difference between the average prediction $\mathbb{E}[\hat{f}(x)]$ and the true function $f(x)$. High bias indicates underfitting: the model class cannot capture the real structure.

Variance: The expected squared deviation of predictions from their average: $\mathbb{E}[(\hat{f}(x) - \mathbb{E}[\hat{f}(x)])^2]$. High variance indicates overfitting: the model is too sensitive to the particular training sample.

Irreducible error: The noise variance σ^2 intrinsic to the data. No model, no matter how clever, can predict a truly random component. This sets a floor on achievable test error.

Ridge Regression (L2 Regularization)

Ridge regression, introduced by Hoerl and Kennard in 1970, adds a squared L2 penalty to the OLS loss:

$$L_{\text{ridge}}(\beta) = \sum_{i=1}^{n}(y_i - \beta^\top x_i)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 = \|y - X\beta\|^2 + \lambda\|\beta\|_2^2.$$

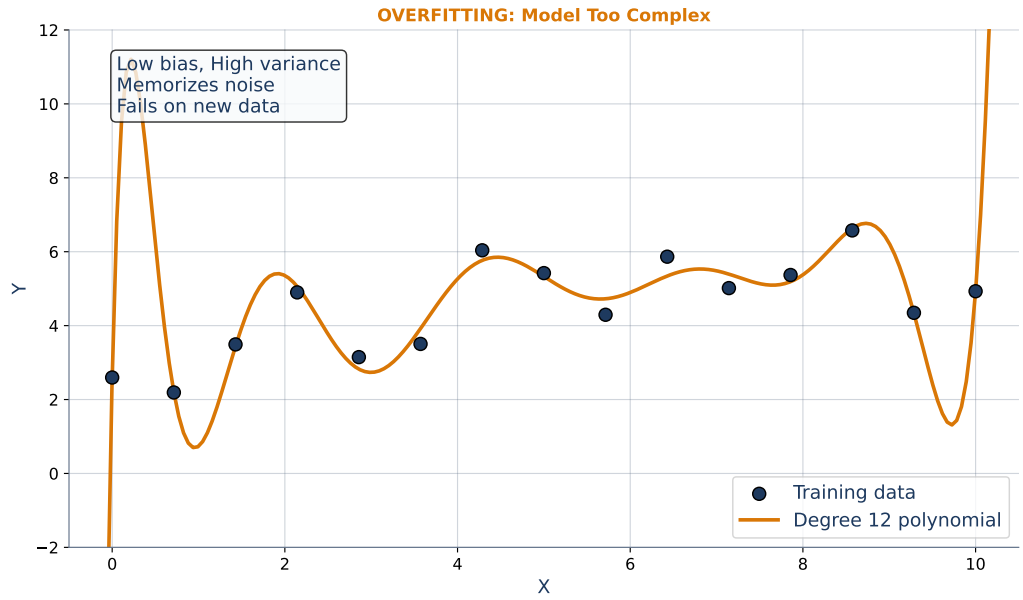


Figure 17: Overfitting: a high-capacity model chases every data point, including the noise. Training error is near zero but test error explodes. Variance is high; bias is low.

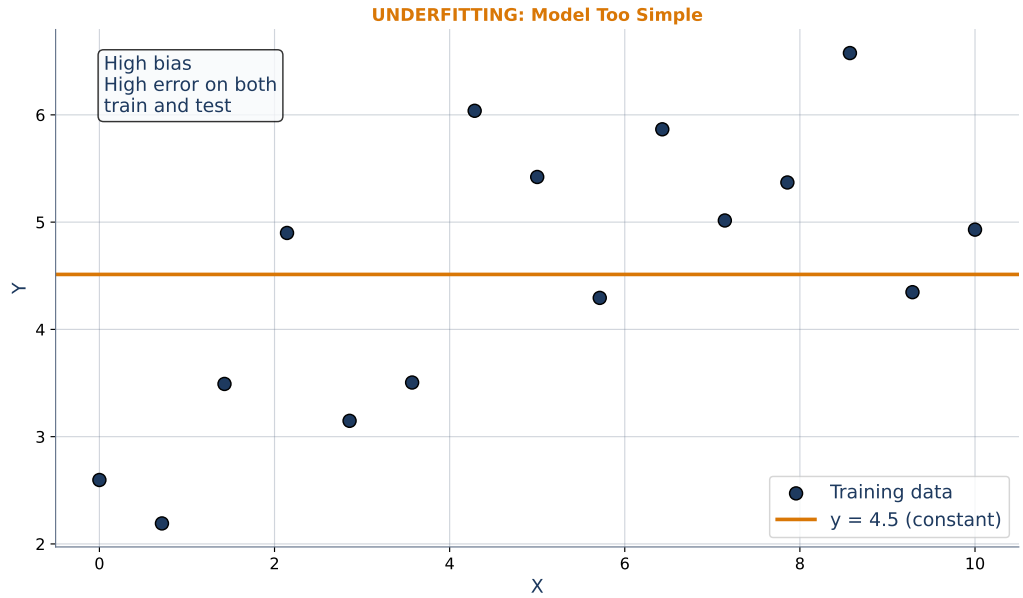


Figure 18: Underfitting: a too-simple model cannot capture the true structure. Both training and test error are high. Bias is high; variance is low.

Here $\lambda \geq 0$ is a hyperparameter that controls the strength of regularization: $\lambda = 0$ recovers OLS; larger λ pulls coefficients toward zero.

The normal equation gains a simple modification. Set the gradient to zero:

$$\nabla L_{\text{ridge}} = -2X^\top y + 2X^\top X\beta + 2\lambda\beta = 0 \implies (X^\top X + \lambda I)\beta = X^\top y.$$

So:

$$\hat{\beta}_{\text{ridge}} = (X^\top X + \lambda I)^{-1} X^\top y.$$

The magical consequence is that $X^\top X + \lambda I$ is *always* invertible for $\lambda > 0$, even when $X^\top X$ is singular. Ridge rescues OLS in the high-dimensional regime.

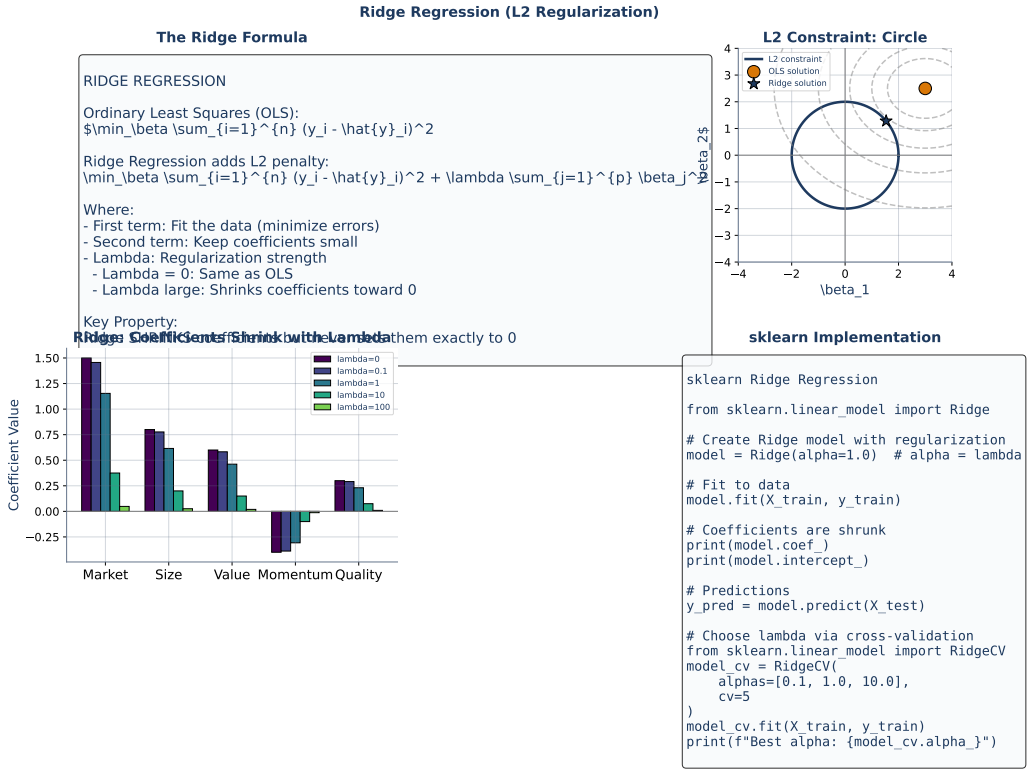


Figure 19: Ridge regression shrinks coefficients toward zero continuously. As λ increases, all coefficients shrink but none are set exactly to zero.

Key Formula: Ridge Regression Closed Form

Given feature matrix $X \in \mathbb{R}^{n \times p}$, label vector $y \in \mathbb{R}^n$, and regularization strength $\lambda > 0$:

$$\hat{\beta}_{\text{ridge}} = (X^\top X + \lambda I)^{-1} X^\top y.$$

Plain English: add λ times the identity matrix to $X^\top X$ before inverting. This one trick makes the inversion stable, shrinks coefficients toward zero, and trades a little bias for much less variance.

Deriving Ridge from Scratch

Start with the ridge loss and expand:

$$L(\beta) = (y - X\beta)^\top (y - X\beta) + \lambda\beta^\top\beta = y^\top y - 2y^\top X\beta + \beta^\top X^\top X\beta + \lambda\beta^\top\beta.$$

Take the gradient:

$$\nabla L = -2X^\top y + 2X^\top X\beta + 2\lambda\beta = -2X^\top y + 2(X^\top X + \lambda I)\beta.$$

Set to zero and solve:

$$(X^\top X + \lambda I)\beta = X^\top y \implies \hat{\beta}_{\text{ridge}} = (X^\top X + \lambda I)^{-1} X^\top y.$$

Check the Hessian: $\nabla^2 L = 2(X^\top X + \lambda I)$, which is positive definite for $\lambda > 0$ (even when $X^\top X$ is singular). The solution is the unique global minimum.

Lasso Regression (L1 Regularization)

Lasso (Least Absolute Shrinkage and Selection Operator), introduced by Robert Tibshirani in 1996, uses an L1 penalty instead:

$$L_{\text{lasso}}(\beta) = \|y - X\beta\|^2 + \lambda\|\beta\|_1 = \|y - X\beta\|^2 + \lambda \sum_{j=1}^{p} |\beta_j|.$$

The L1 penalty has a crucial difference from L2: it produces *sparse* solutions. For large enough λ, many coefficients $\hat{\beta}_j$ are exactly zero. Lasso is both a regularizer and a feature-selection mechanism.

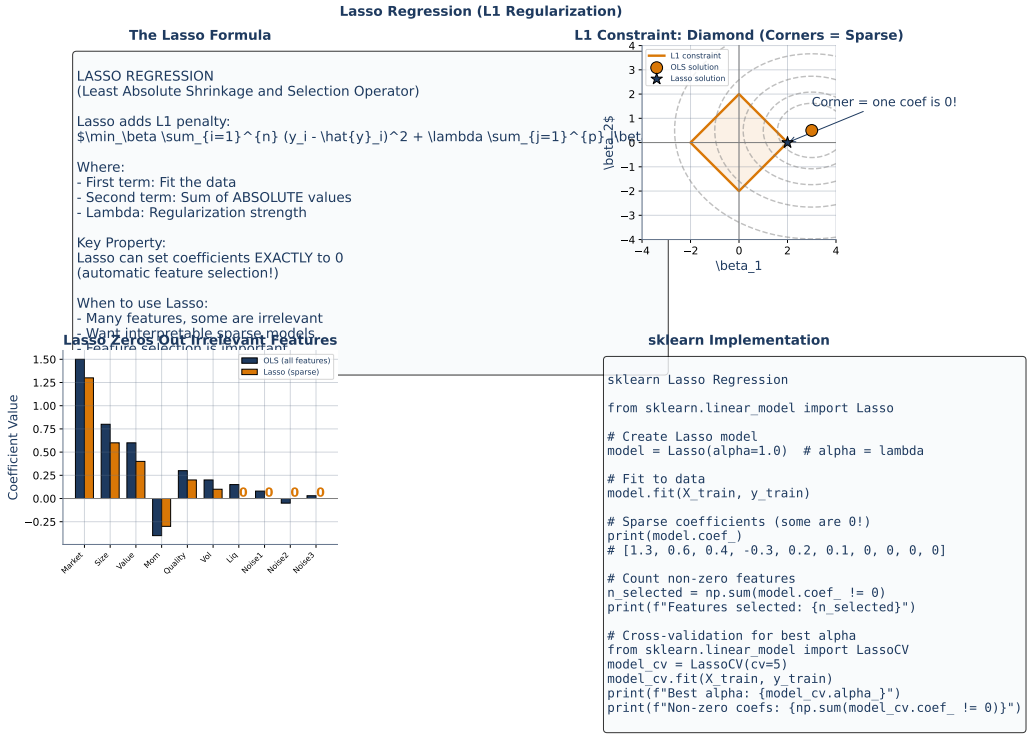


Figure 20: Lasso regression drives some coefficients to exactly zero. The sparsity is the source of Lasso's feature-selection capability.

Unlike Ridge, Lasso has no closed-form solution because $|\beta_j|$ is not differentiable at $\beta_j = 0$. Fitting requires iterative methods: coordinate descent or proximal gradient algorithms. scikit-learn's `Lasso` class hides these details.

The Geometry of L1 versus L2

Why does L1 produce sparsity but L2 does not? The answer is geometric. Both Lasso and Ridge can be reformulated as constrained optimization:

$$\text{Lasso:} \quad \min_{\beta} \|y - X\beta\|^2 \text{ subject to } \|\beta\|_1 \leq t,$$

$$\text{Ridge:} \quad \min_{\beta} \|y - X\beta\|^2 \text{ subject to } \|\beta\|_2 \leq t.$$

The L1 constraint set $\{\beta : \|\beta\|_1 \leq t\}$ is a diamond (in 2D) with corners on the coordinate axes. The L2 constraint set $\{\beta : \|\beta\|_2 \leq t\}$ is a disk. The OLS loss contours are ellipses. When the loss ellipse first touches the constraint boundary, the touch point is the regularized estimate. For a diamond, the contact is almost always at a corner—and corners lie on the axes, where one or more coefficients are zero. For a disk, the contact is almost never at a coordinate axis.

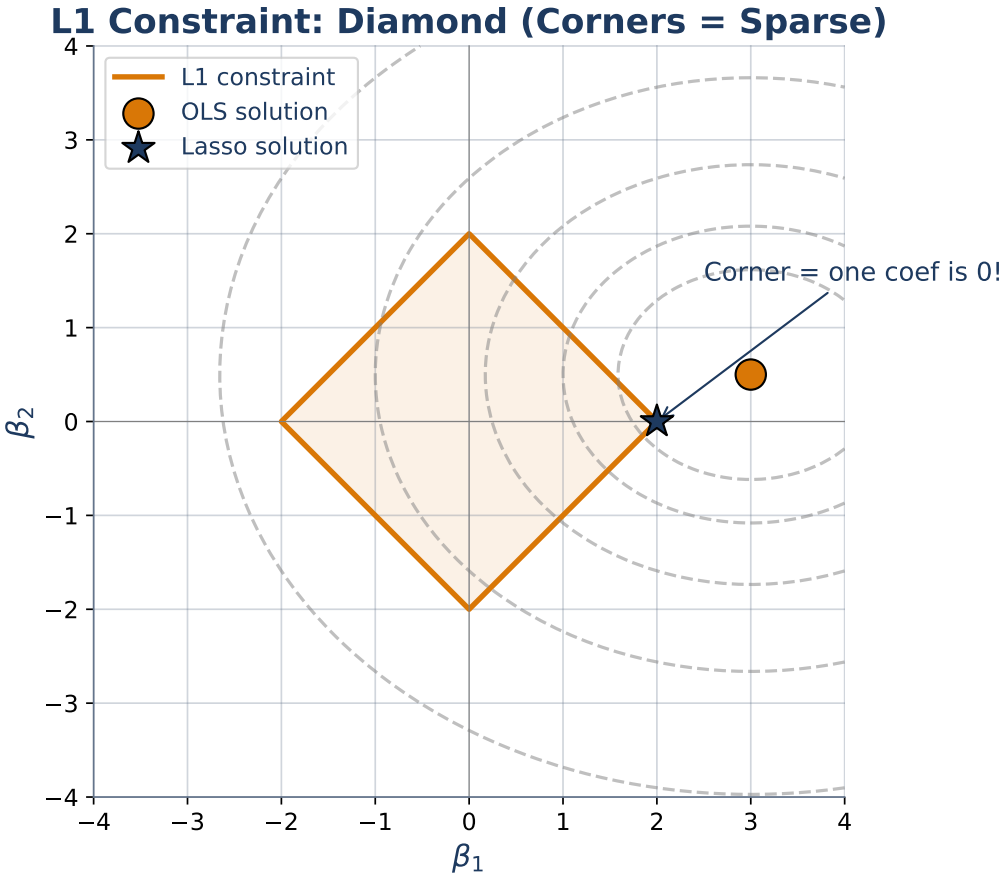


Figure 21: The L1 diamond has corners on the axes. The OLS loss ellipse touches a corner, forcing one coefficient to zero: sparsity.

This geometric distinction is why Lasso does feature selection and Ridge does not. The "corners on the axes" property is specific to the L1 norm; any Lp norm with $p \leq 1$ produces sparsity, any Lp with $p > 1$ does not.

ElasticNet: Best of Both Worlds

ElasticNet, introduced by Zou and Hastie in 2005, combines L1 and L2 penalties:

$$L_{\text{en}}(\beta) = \|y - X\beta\|^2 + \lambda\left[\alpha\|\beta\|_1 + (1-\alpha)\|\beta\|_2^2\right].$$

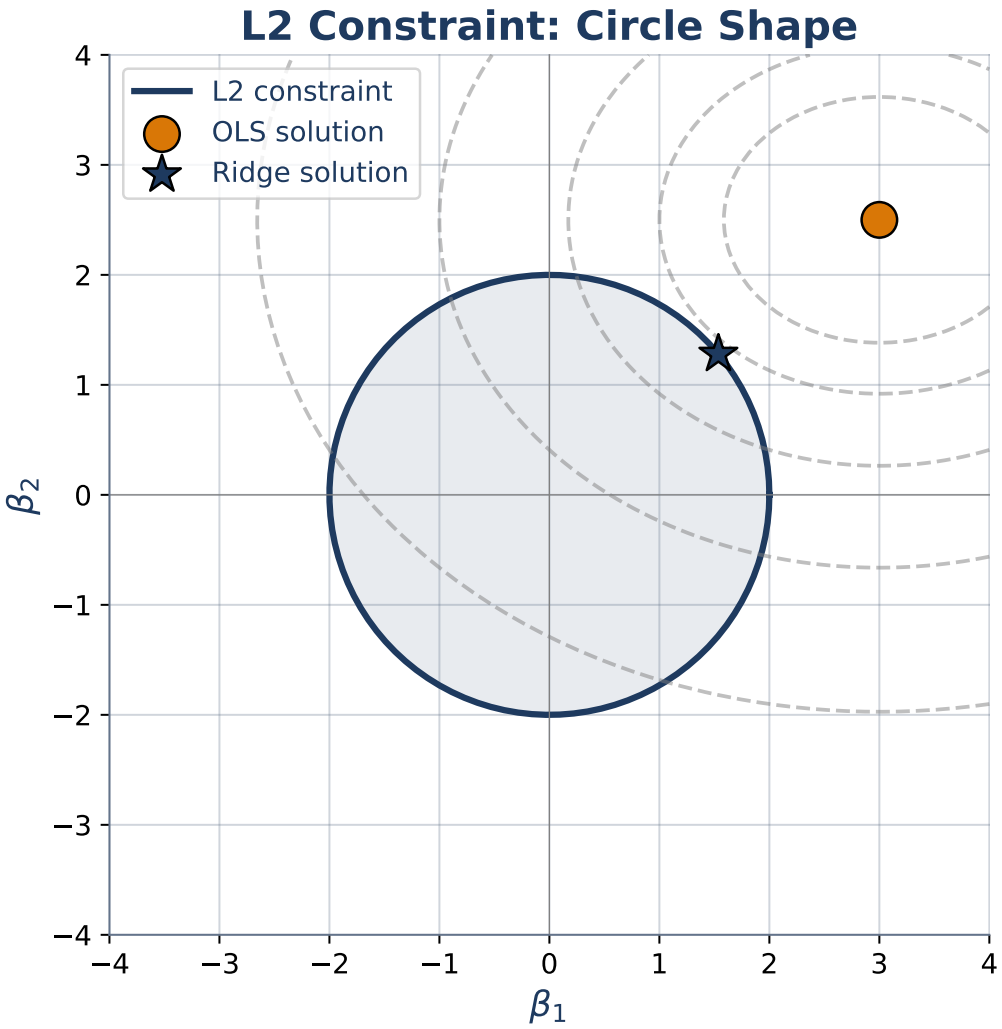


Figure 22: The L2 disk has no corners. The OLS loss ellipse touches a smooth point on the boundary; coefficients shrink smoothly but stay nonzero.

The mixing parameter $\alpha \in [0, 1]$ controls the blend: $\alpha = 1$ is pure Lasso, $\alpha = 0$ is pure Ridge. ElasticNet is particularly useful when features come in correlated groups: Lasso tends to pick one feature from the group and zero out the others arbitrarily, while ElasticNet's L2 component keeps correlated features together.

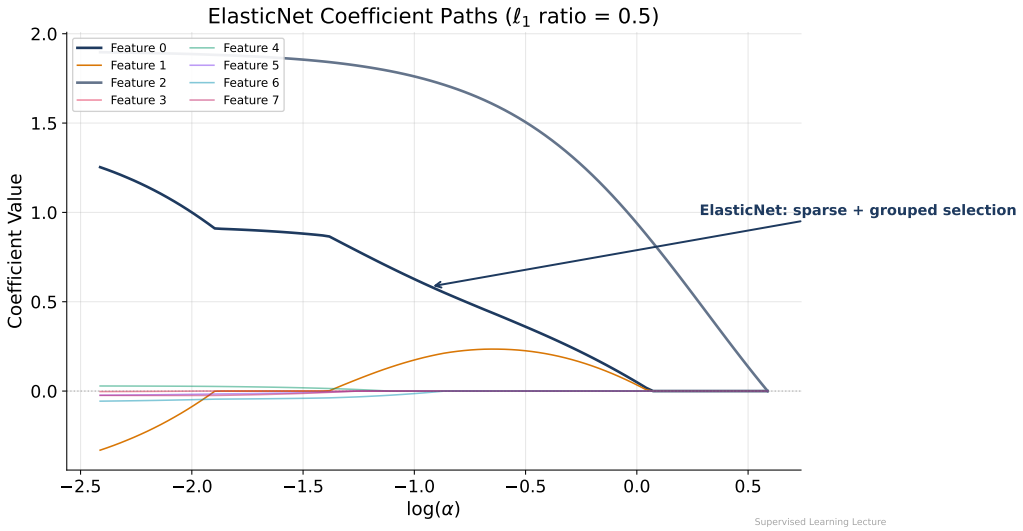


Figure 23: Coefficient paths under ElasticNet as λ varies. The L1 component drives coefficients to zero; the L2 component keeps correlated features bundled together.

Coefficient Paths

A *coefficient path* plots each $\hat{\beta}_j$ as a function of λ. At $\lambda = 0$ you see the unregularized OLS values. As λ grows, Ridge shrinks all coefficients smoothly toward zero without ever reaching it; Lasso drives them to exact zeros one by one. The paths are diagnostic: they show which features survive heavy regularization.

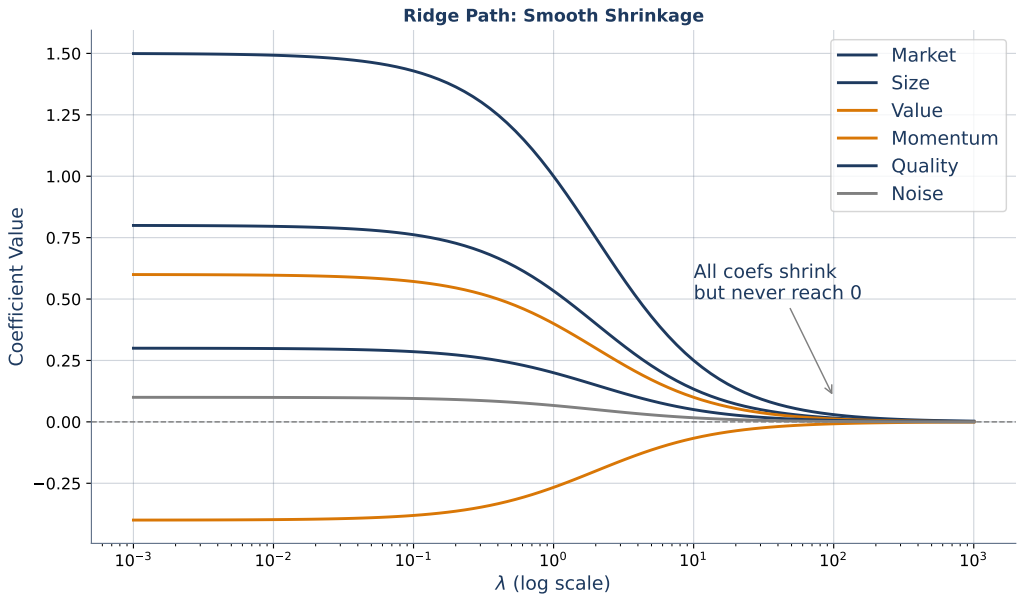


Figure 24: Ridge coefficient path. All coefficients shrink smoothly toward zero as λ grows, but never reach it.

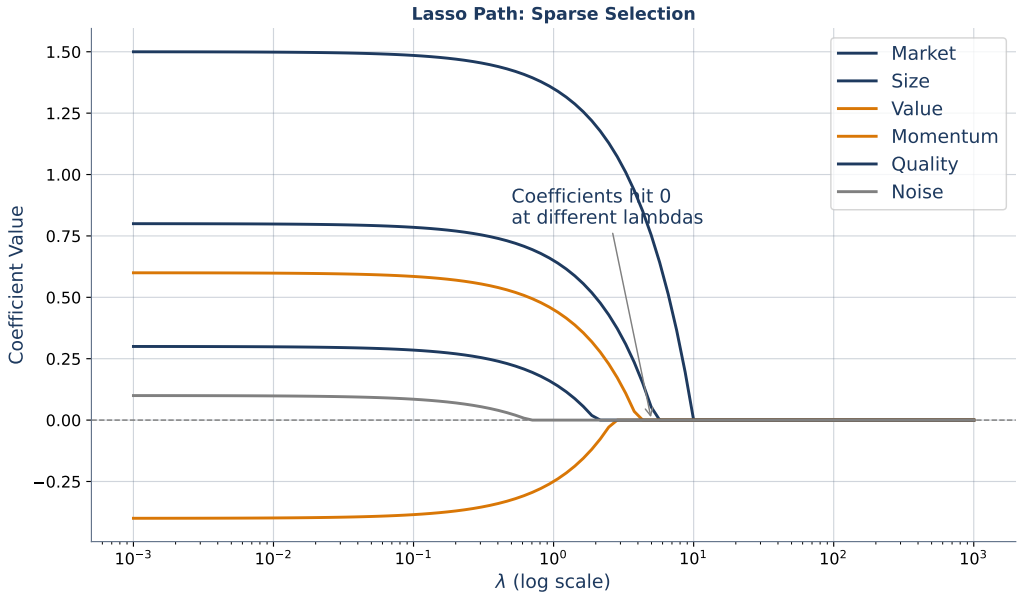


Figure 25: Lasso coefficient path. Coefficients hit zero one by one as λ increases. The features that survive the largest λ are the most important.

Choosing λ: Cross-Validation

The regularization strength λ is a *hyperparameter*: it cannot be fit by minimizing training loss (which always prefers $\lambda = 0$). Instead, we choose λ using *cross-validation*.

In K-fold cross-validation, the training data is split into K equal folds. For each candidate λ, we train on K-1 folds and evaluate on the held-out fold. Averaging across folds gives an out-of-sample estimate of generalization error for that λ. We pick the λ that minimizes this cross-validated error.

Sparse Factor Models in Finance

The typical finance factor zoo has dozens or hundreds of candidate factors: market, size, value, momentum, quality, low-volatility, profitability, investment, sentiment, short interest, and on. For a specific asset class or time window, only a handful of factors actually matter. Lasso regression is ideal for factor selection: fit a Lasso with cross-validated λ, and the nonzero coefficients are your active factors.

This approach has two major advantages over OLS with all factors. First, Lasso produces a sparse model that a portfolio manager can read: "The top-4 nonzero factors are momentum, value, quality, and low-volatility." Second, Lasso avoids multicollinearity disasters: when two factors are highly correlated, Lasso picks one and zeros the other, rather than giving both large and opposing coefficients.

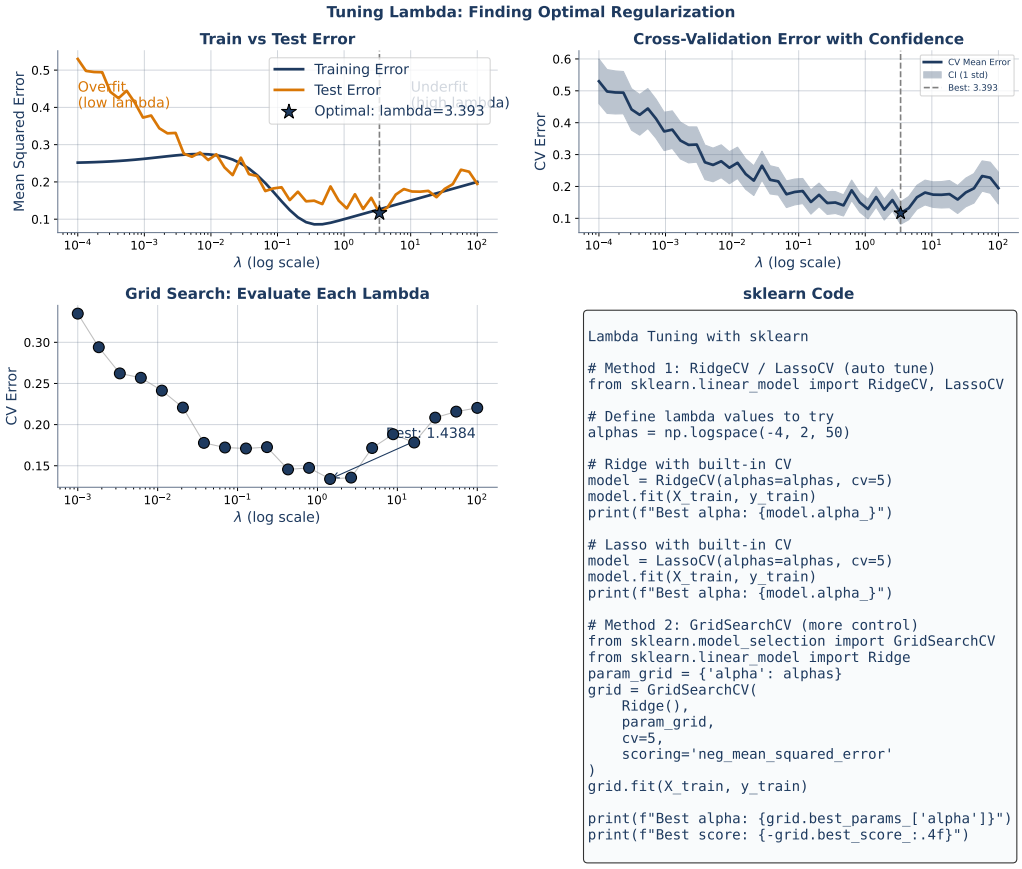


Figure 26: Cross-validation picks the λ that minimizes the K-fold mean squared error. Too small λ overfits (left of minimum); too large underfits (right of minimum).

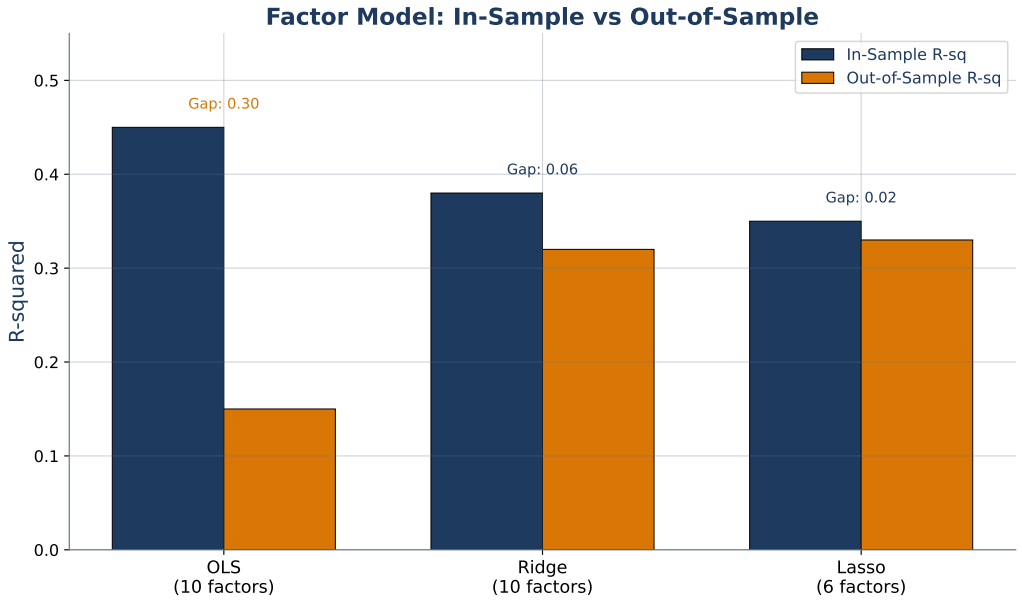


Figure 27: Sparse factor model via Lasso. Out of 50 candidate factors, cross-validation selects about a dozen. The sparse model often outperforms the full OLS model on held-out returns.

Worked Examples

Worked Example 1: Ridge Rescues a Singular $X^\top X$

Suppose $X = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$ and $y = (2, 4)^\top$. OLS requires $X^\top X = \begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix}$, which has rank 1 and is singular. OLS is undefined.

Ridge with $\lambda = 1$: $X^\top X + I = \begin{pmatrix} 3 & 2 \\ 2 & 3 \end{pmatrix}$, which is nonsingular with determinant $9 - 4 = 5$ and inverse $\frac{1}{5}\begin{pmatrix} 3 & -2 \\ -2 & 3 \end{pmatrix}$. Compute $X^\top y = \begin{pmatrix} 6 \\ 6 \end{pmatrix}$. Then $\hat{\beta}_{\text{ridge}} = \frac{1}{5}\begin{pmatrix} 3 & -2 \\ -2 & 3 \end{pmatrix}\begin{pmatrix} 6 \\ 6 \end{pmatrix} = \frac{1}{5}\begin{pmatrix} 6 \\ 6 \end{pmatrix} = (1.2, 1.2)^\top$.

Sanity check: with both features identical, Ridge splits the effect equally between them. OLS cannot; Ridge can. This is why Ridge is the default rescue when features are collinear.

Worked Example 2: Bias-Variance Numbers

A toy setup. True model: $f(x) = x^2$, with noise $\epsilon \sim \mathcal{N}(0, 0.25)$. Training set: 20 points on $x \in [-1, 1]$.

Model A: predict the constant 0. Then $\mathbb{E}[\hat{f}] = 0$, so $\text{Bias}^2 = (0 - x^2)^2$. Average over $x \in [-1, 1]$: $\int_{-1}^{1} x^4 \cdot \frac{1}{2} dx = \frac{1}{5}$. Variance is zero (the model is a constant). Irreducible noise: 0.25. Expected MSE: $0.2 + 0 + 0.25 = 0.45$.

Model B: a 20-parameter polynomial fitted to the 20 training points exactly. Training MSE is zero. Bias is small (the polynomial class can express x^2). But variance is enormous: a single point that shifts by its noise can move the polynomial substantially. Expected MSE: roughly $0 + 2.5 + 0.25 = 2.75$.

Model C: a 2-parameter quadratic. Bias is essentially zero (the correct form). Variance is small (only 2 parameters). Expected MSE: roughly $0 + 0.05 + 0.25 = 0.30$.

Model C wins by a wide margin over both the over-simple Model A and the over-complex Model B. Ridge effectively converts Model B into something between B and C by penalizing large coefficients.

Historical Background: Hoerl, Kennard, Tibshirani, and the Rise of Regularization

Ridge regression was introduced by Arthur Hoerl in 1962 in a chemical-engineering journal (his PhD work on reaction rates), and developed formally in a 1970 pair of papers by Hoerl and Robert Kennard: "Ridge Regression: Biased Estimation for Nonorthogonal Problems" and "Ridge Regression: Applications to Nonorthogonal Problems." The original motivation was numerical stability in industrial statistics: when engineers ran factorial experiments with correlated factors, the $X^\top X$ matrix became nearly singular and OLS coefficients exploded. Ridge was a pragmatic fix for the practitioner.

Lasso has a very different origin. In 1996, Robert Tibshirani (then at the University of Toronto) published "Regression Shrinkage and Selection via the Lasso." The paper was motivated by a desire for *interpretable* models: OLS coefficients are hard to read with many features, and Ridge keeps everything nonzero. Tibshirani observed that the L1 penalty produces sparse solutions—a property already known in geophysics via the LAD (Least Absolute Deviation) literature—and cast it in the regression framework. Lasso became the default method for feature selection in high-dimensional statistics.

ElasticNet (Zou and Hastie, 2005) blended L1 and L2 to address a specific weakness: Lasso picks one feature from a correlated group arbitrarily. ElasticNet keeps them bundled. Hui Zou's PhD thesis, supervised by Trevor Hastie, laid out the theory.

The modern deep-learning era added many more regularization techniques (dropout, weight decay, data augmentation), but the L1/L2/ElasticNet triumvirate remains the default for linear models. Every `LogisticRegression` call in scikit-learn has an L2 default (`penalty='l2'`), and every `Lasso` or `ElasticNet` is a direct descendant of Hoerl, Kennard, and Tibshirani.

Ridge, Lasso, and ElasticNet with Cross-Validation

```
from sklearn.linear_model import RidgeCV, LassoCV, ElasticNetCV
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
import numpy as np

# Grid of lambdas (alphas in sklearn)
alphas = np.logspace(-4, 2, 50)

# Ridge
ridge = Pipeline([
    ('scaler', StandardScaler()),
    ('model',  RidgeCV(alphas=alphas, cv=5)),
])
ridge.fit(X_train, y_train)
print(f'Best ridge alpha: {ridge.named_steps["model"].alpha_}')

# Lasso
lasso = Pipeline([
    ('scaler', StandardScaler()),
    ('model',  LassoCV(alphas=alphas, cv=5, max_iter=10000)),
])
lasso.fit(X_train, y_train)
print(f'Best lasso alpha: {lasso.named_steps["model"].alpha_}')
print(f'Nonzero coefs: {np.sum(lasso.named_steps["model"].coef_
    != 0)}')

# ElasticNet (grid over L1-ratio)
en = Pipeline([
    ('scaler', StandardScaler()),
    ('model',  ElasticNetCV(l1_ratio=[0.1, 0.5, 0.9],
                            alphas=alphas, cv=5, max_iter=10000))
                               ,
])
en.fit(X_train, y_train)
```

A Note on Feature Scaling

Regularization penalizes the size of coefficients. If features are on wildly different scales (income in dollars versus age in years), the coefficients for small-scale features will appear large simply because they multiply small numbers. Ridge and Lasso will then penalize them disproportionately. The fix is universal: *standardize features before regularization.* `StandardScaler` subtracts the mean and divides by the standard deviation for each feature, so all features are on the same scale.

Failing to standardize before regularization is one of the most common bugs in applied ML. The scikit-learn default regularization behavior assumes standardized features; if you skip the scaler, your results are garbage.

Definition: Regularized Linear Regression

A regularized linear regression model is defined by the loss

$$L(\beta) = \|y - X\beta\|^2 + \lambda R(\beta),$$

where $R(\beta)$ is a regularization functional (penalty) and $\lambda > 0$ is a regularization strength. Common choices:

- **Ridge** (L2): $R(\beta) = \|\beta\|_2^2 = \sum_j \beta_j^2$. Closed-form solution $\hat{\beta} = (X^\top X + \lambda I)^{-1} X^\top y$.
- **Lasso** (L1): $R(\beta) = \|\beta\|_1 = \sum_j |\beta_j|$. Solved via coordinate descent; produces sparse $\hat{\beta}$.
- **ElasticNet**: $R(\beta) = \alpha\|\beta\|_1 + (1-\alpha)\|\beta\|_2^2$. Combines sparsity (L1) with stability on correlated features (L2).

λ is selected by cross-validation to balance training error against generalization error.

Common Misconceptions about Regularization

(1) "Regularization is cheating because it ignores data." Regularization deliberately introduces bias to reduce variance. The net effect is lower test error, not a worse model. It is the *sine qua non* of high-dimensional statistics.
(2) "Lasso and Ridge give similar answers." They are fundamentally different: Lasso produces sparse solutions (feature selection), Ridge does not. On the same data they can give qualitatively different coefficient patterns.
(3) "A large λ is always safer." Too large λ crushes all coefficients to nearly zero and produces underfitting. Cross-validation picks λ honestly; don't override it without a reason.
(4) "Standardization only matters for distance-based methods." It matters just as much for regularization. Unscaled features make the penalty depend arbitrarily on feature units—grams versus kilograms change your $\hat{\beta}$.

Problem 3.1 (Easy) ⋆

A Ridge regression was fitted with $\lambda_1 = 0.1$ and with $\lambda_2 = 100$. Which set of coefficients, $\hat{\beta}^{(1)}$ or $\hat{\beta}^{(2)}$, is more heavily regularized (smaller in L2 norm)? Which is more likely to underfit the data?
Solution: see Appendix.

Problem 3.2 (Medium) ⋆⋆

For the true model $f(x) = 2x$, the learner uses the single-parameter family $\hat{f}(x) = cx$ and estimates c from a random sample of size $n = 10$ where $y = 2x + \epsilon$, $\epsilon \sim \mathcal{N}(0, 1)$. Suppose the training x-values are fixed at $1, 2, \ldots, 10$. Compute the bias and variance of the OLS estimate of c at a fixed test point $x = 5$.
Solution: see Appendix.

Problem 3.3 (Medium) ⋆⋆

Sketch the L1 unit ball (diamond) and the L2 unit ball (disk) in 2D. Draw a family of elliptical OLS loss contours centered at $(2, 1)$. Use the sketch to explain why the L1-constrained minimum is typically at a corner while the L2-constrained minimum is not.
Solution: see Appendix.

Problem 3.4 (Hard) ⋆⋆⋆

Derive the closed-form Ridge solution $\hat{\beta} = (X^\top X + \lambda I)^{-1} X^\top y$ from scratch: write down the loss, take the gradient, set to zero, solve. Also verify that the Hessian is positive definite for $\lambda > 0$. Finally, show that Ridge is equivalent to OLS on an augmented dataset with $\sqrt{\lambda}$ pseudo-observations appended.
Solution: see Appendix.

Problem 3.5 (Hard) ⋆⋆⋆

Design a cross-validation strategy for tuning the Lasso λ on a 5-year time series of daily returns with 500 features. Explain why standard K-fold CV is inappropriate and what you would do instead. Provide pseudocode.
Solution: see Appendix.

Connecting Forward

Regularization extends OLS from a clean but fragile closed-form solution to a robust tool that works even when features outnumber observations. Ridge stabilizes the inversion; Lasso performs feature selection. Both require a hyperparameter λ tuned by cross-validation.

Section 4 turns to the question of how we *measure* regression performance in the first place. Squared error is the training loss, but there are many metrics for evaluation: MSE, RMSE, MAE, R^2, and adjusted R^2. We will derive each, compare their properties, and explain why stock-return predictions almost always have tiny R^2 values—yet can still be economically valuable.

Key Takeaway: Regularization trades a little bias for a lot less variance by penalizing coefficient size; Ridge shrinks, Lasso selects, ElasticNet combines—and cross-validation picks λ.

4. Measuring What Matters – Regression Metrics and Validation

Opening Problem: Is an R^2 of 0.05 Any Good?

You train two quant models on five years of monthly stock returns. Model A reports an R^2 of 0.05. Model B reports an R^2 of 0.08. Your director asks which is better and whether either is worth deploying. A colleague from the climate-modeling team walks by and scoffs: "R^2 of 0.08? Our climate models have R^2 of 0.95. You call that predictive?"
The climate colleague is not wrong that 0.95 is higher than 0.08. But he is wrong that 0.08 is meaningless. For monthly stock returns, an R^2 of 0.05 is remarkable; an R^2 of 0.08 is almost suspiciously good. A model with $R^2 = 0.02$ that earns \$50 million per year is more valuable than a model with $R^2 = 0.95$ that earns nothing. The question "is $R^2 = 0.08$ good?" has no universal answer—it depends on the domain, the data, and the economics of being right.
This section builds the toolbox for measuring regression performance: MSE, RMSE, MAE, R^2, and adjusted R^2. It explains why each exists, when to use which, and why stock-return R^2 values are tiny yet still valuable. It also treats cross-validation in depth—including the subtle problem of applying K-fold CV to time series.

Discovery Question

Two regression models have the same RMSE. Model A's errors are uniformly distributed around zero; Model B has a few large errors and many small ones. Which model would you prefer? Under what circumstances would RMSE mislead you about real-world performance?

Mean Squared Error (MSE) and RMSE

The most common regression metric is mean squared error:

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

MSE is the natural loss for squared-error regression: it is exactly what OLS minimizes (divided by n). Its units are the square of the label's units, which is awkward: if y is dollars, MSE is dollars-squared.

Root mean squared error (RMSE) fixes the units:

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

RMSE is in the same units as the label. An RMSE of \$10,000 for a house-price model means the typical prediction error is on the order of \$10,000.

Mean Absolute Error (MAE)

MSE/RMSE squares errors. Mean absolute error (MAE) takes absolute values instead:

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

MAE is more robust to outliers than MSE: a single enormous error contributes its own magnitude to MAE, but its *square* to MSE. If your data has heavy-tailed errors (stock returns during 2008), MAE can be a better summary of "typical" error.

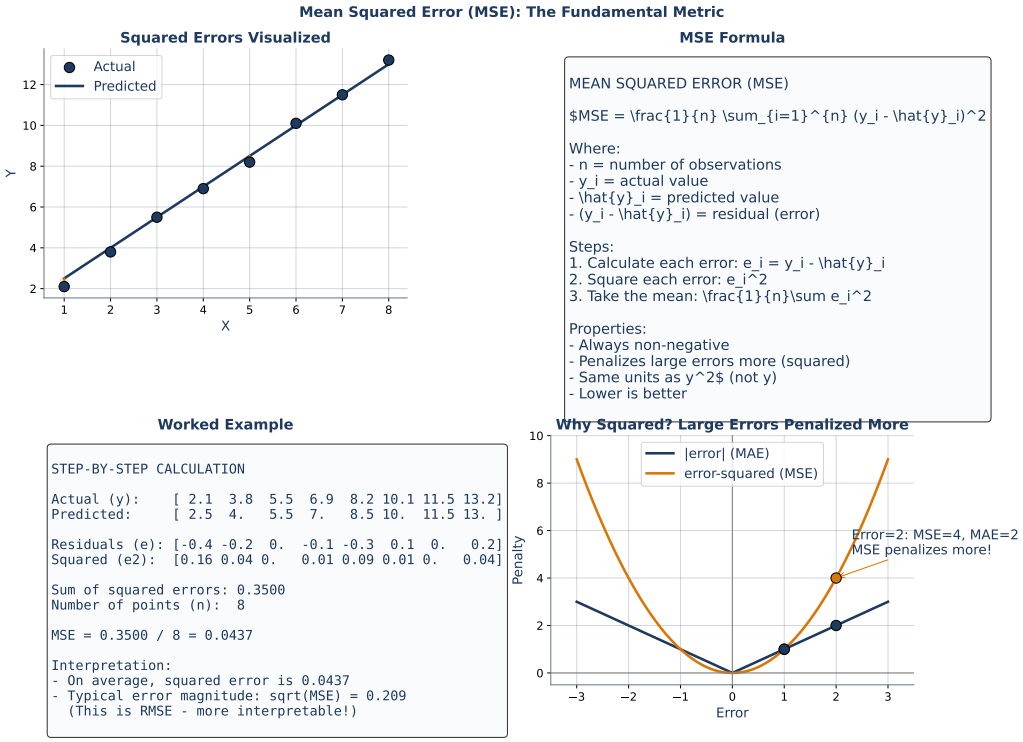


Figure 28: MSE measures the average squared residual. Squaring penalizes large errors more heavily than small ones.

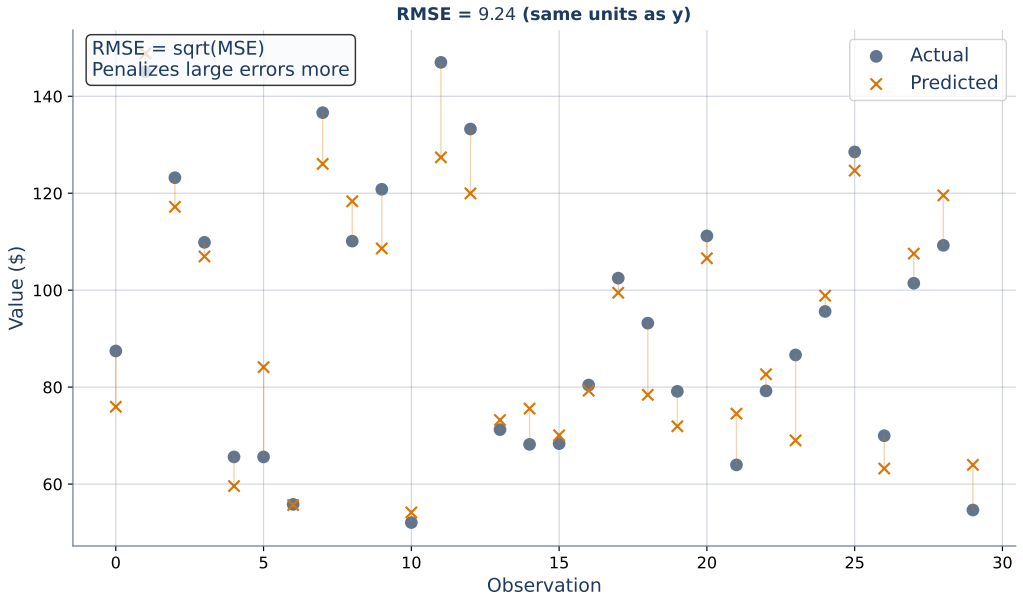


Figure 29: RMSE is the square root of MSE. Same information, more interpretable units (same as the label).

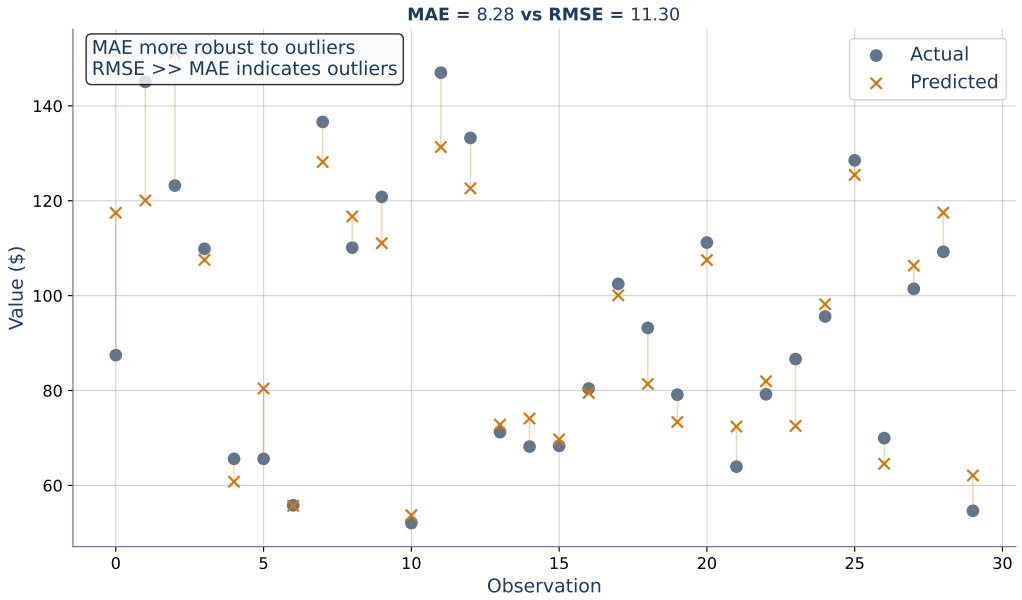


Figure 30: MAE treats all errors linearly. It is less sensitive to outliers than MSE/RMSE, but it is not differentiable at zero, which complicates gradient-based training.

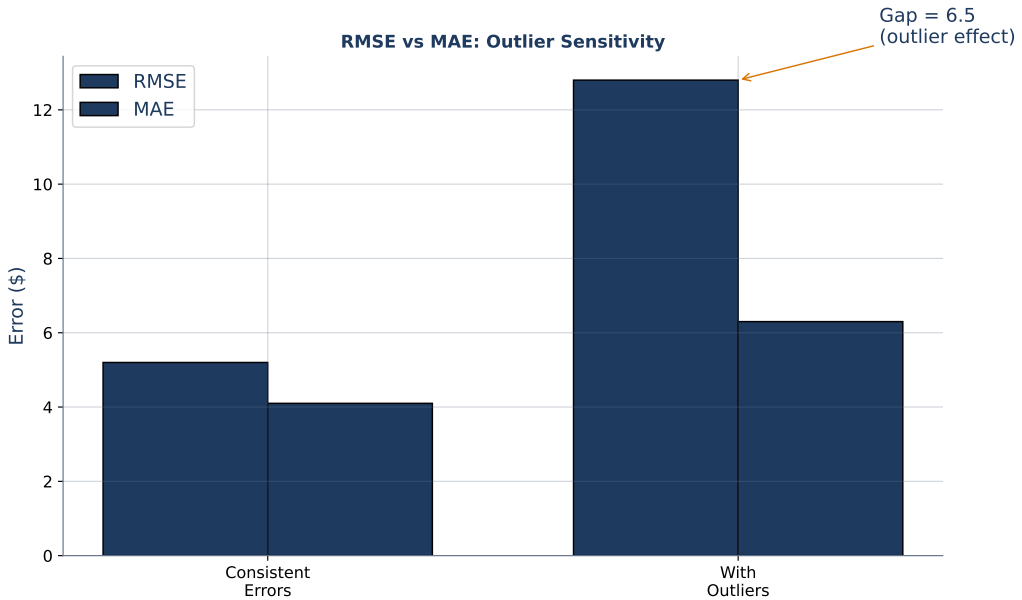


Figure 31: Comparing MSE, RMSE, MAE on the same residuals. MSE amplifies the impact of the largest residuals; MAE treats them all equally.

R^2: Coefficient of Determination

R^2 measures how much of the variance in y the model explains:

$$R^2 = 1 - \frac{\text{SS}_{\text{res}}}{\text{SS}_{\text{tot}}} = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2}.$$

The denominator is the total sum of squares (variance of y around its mean, times n). The numerator is the residual sum of squares (unexplained variance). Thus R^2 is the fraction of variance *explained* by the model.

$R^2 = 1$ means perfect prediction. $R^2 = 0$ means the model does no better than predicting the mean of y. $R^2 < 0$ is possible on a test set: a poorly fit model can be worse than the mean predictor, which gives a negative R^2.

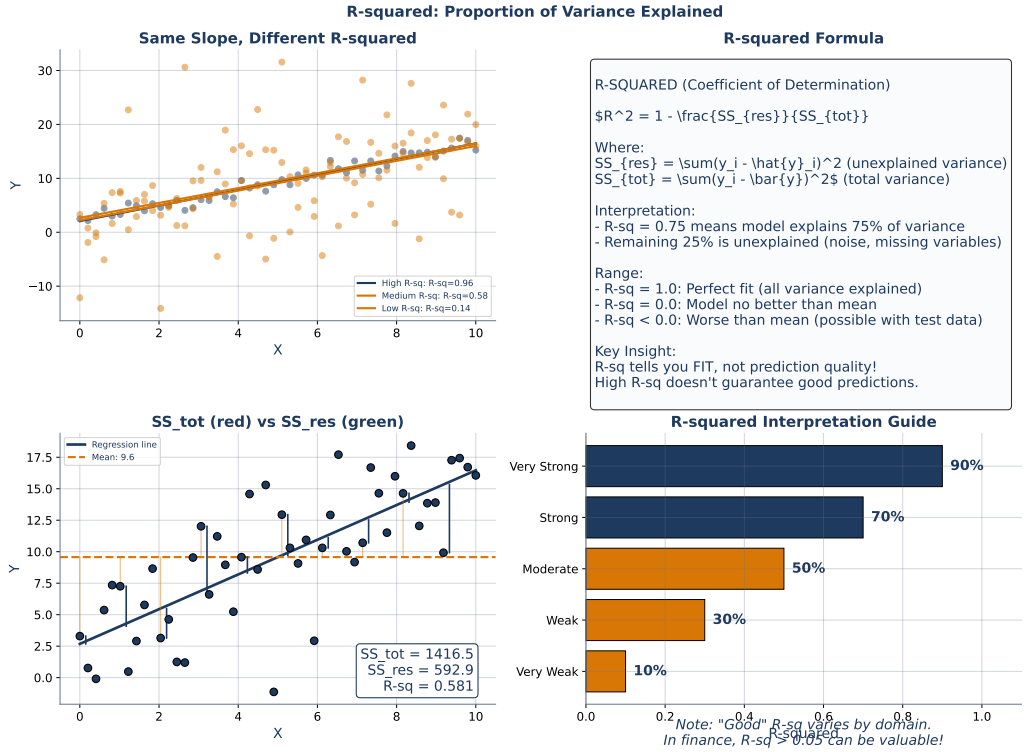


Figure 32: R^2 visualized: the ratio of explained sum of squares to total sum of squares. $R^2 = 1 - \text{SS}_{\text{res}}/\text{SS}_{\text{tot}}$.

Key Formula: R^2

Given observations $y_1, \ldots, y_n$ with mean $\bar{y}$ and model predictions $\hat{y}_1, \ldots, \hat{y}_n$:

$$R^2 = 1 - \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{n} (y_i - \bar{y})^2}.$$

Equivalently, $R^2 = 1 - \text{MSE}_{\text{model}}/\text{MSE}_{\text{mean-only}}$. The baseline is the constant predictor $\hat{y}_i = \bar{y}$.

Plain English: "what fraction of the variance in y did our model explain?" Close to 1 is ideal; close to 0 means the model is no better than the sample mean.

Adjusted R^2

Plain R^2 has a flaw: it cannot decrease when you add a new feature, even if the new feature is pure noise. Any feature, no matter how irrelevant, will fit some of the training-set variance just by chance. Adjusted R^2 penalizes feature count:

$$R^2_{\text{adj}} = 1 - (1 - R^2)\,\frac{n-1}{n-p-1},$$

where p is the number of features. When a useless feature is added, R^2 rises by a tiny amount, but $n-p-1$ drops by one, typically pushing R^2_{adj} down. Adjusted R^2 therefore prefers parsimonious models: it rewards features that add more explanatory power than they cost in degrees of freedom.

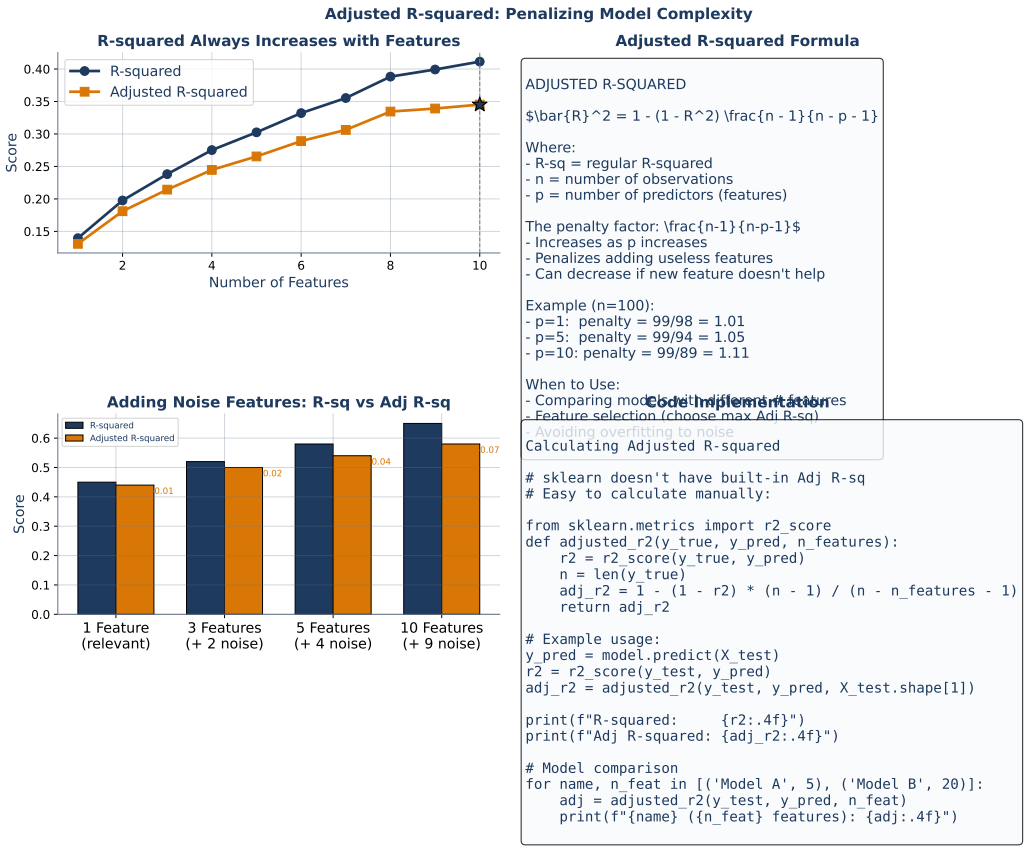


Figure 33: Adjusted R^2 penalizes the addition of features. Plain R^2 monotonically increases; adjusted R^2 peaks near the true number of useful features.

A Worked Example on Five Data Points

Worked Example: Computing All Metrics by Hand

Predict y from x with five observations. Suppose the fit gives:

i	y_i	$\hat{y}_i$	$e_i = y_i - \hat{y}_i$
1	2	2.1	-0.1
2	3	2.8	$+0.2$
3	5	4.9	$+0.1$
4	7	7.3	-0.3
5	9	8.9	$+0.1$

MSE: $((-0.1)^2+(0.2)^2+(0.1)^2+(-0.3)^2+(0.1)^2)/5 = (0.01+0.04+0.01+0.09+0.01)/5 = 0.16/5 = 0.032$.
RMSE: $\sqrt{0.032} \approx 0.179$.
MAE: $(0.1 + 0.2 + 0.1 + 0.3 + 0.1)/5 = 0.8/5 = 0.16$.
R^2: First $\bar{y} = (2+3+5+7+9)/5 = 5.2$. Total SS: $(2-5.2)^2 + (3-5.2)^2 + (5-5.2)^2 + (7-5.2)^2 + (9-5.2)^2 = 10.24 + 4.84 + 0.04 + 3.24 + 14.44 = 32.80$. Residual SS: 0.16 (from MSE calculation, times $n = 5$). $R^2 = 1 - 0.16/32.80 = 1 - 0.00488 = 0.995$.
Adjusted R^2 (with $p = 1$): $R^2_{\text{adj}} = 1-(1-0.995)(5-1)/(5-1-1) = 1-0.005 \cdot (4/3) = 1 - 0.00667 = 0.993$.
All metrics show a strong fit: RMSE and MAE are small relative to the range of y (spanning 2 to 9), and $R^2 \approx 0.995$ means the line captures nearly all the variance.

Why Stock Returns Have Tiny R^2

Financial return prediction routinely reports R^2 values of 0.01 to 0.10 on monthly data. This seems pathetic by physical-sciences standards (astrophysics often gets $R^2 > 0.99$) but reflects a fundamental truth: financial markets are *almost* efficient. If monthly stock returns were very predictable, traders would arbitrage the predictability away within days. The steady state has only small, hard-won predictable components.

An R^2 of 0.05 means the model explains 5% of the variance in monthly returns. The remaining 95% is pure noise relative to the model's information set. But on a portfolio of 500 stocks, even a 5% R^2 can produce meaningful risk-adjusted returns: the law of large numbers converts a small edge per stock into a large aggregate profit. Industrial quantitative investing operates at $R^2 \approx 0.02$–0.05; that is the bar.

Residual Analysis

Beyond a single summary number, you should always plot residuals. A good fit has residuals that look like random noise; a bad fit has residuals with visible structure.

Cross-Validation: The Fundamental Technique

A single train/test split is a point estimate of test performance; different splits give different estimates. Cross-validation averages over many splits to produce a more reliable estimate.

In K-fold cross-validation, split the training data into K equal folds. For each fold $k = 1, \ldots, K$:

1. Train the model on the $K-1$ folds excluding fold k.
2. Evaluate on fold k.

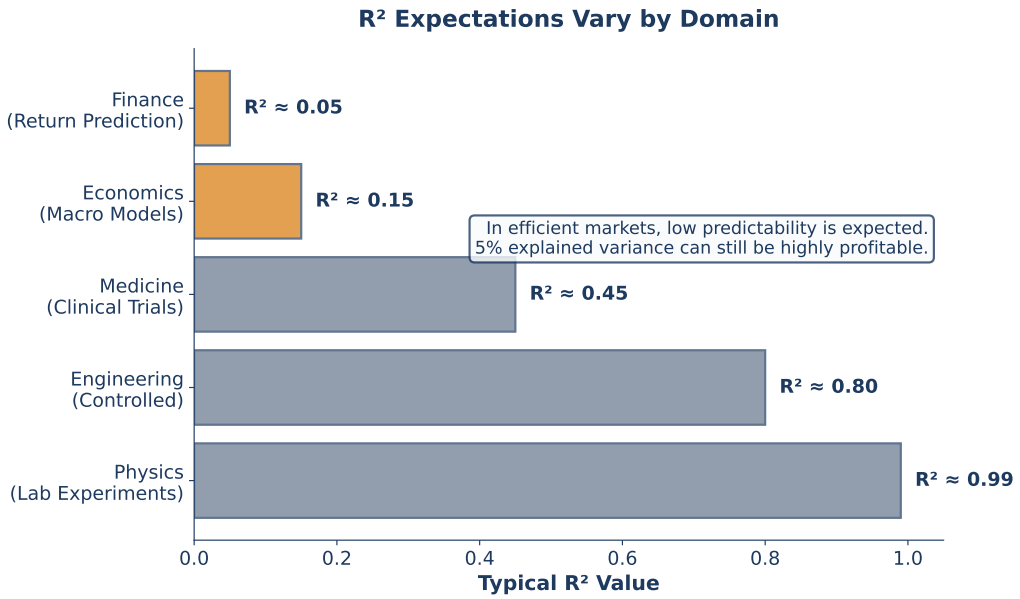


Figure 34: R^2 benchmarks across domains. Physical sciences: $R^2 > 0.9$. Engineering: 0.7–0.9. Social sciences: 0.3–0.5. Financial returns: 0.02–0.10.

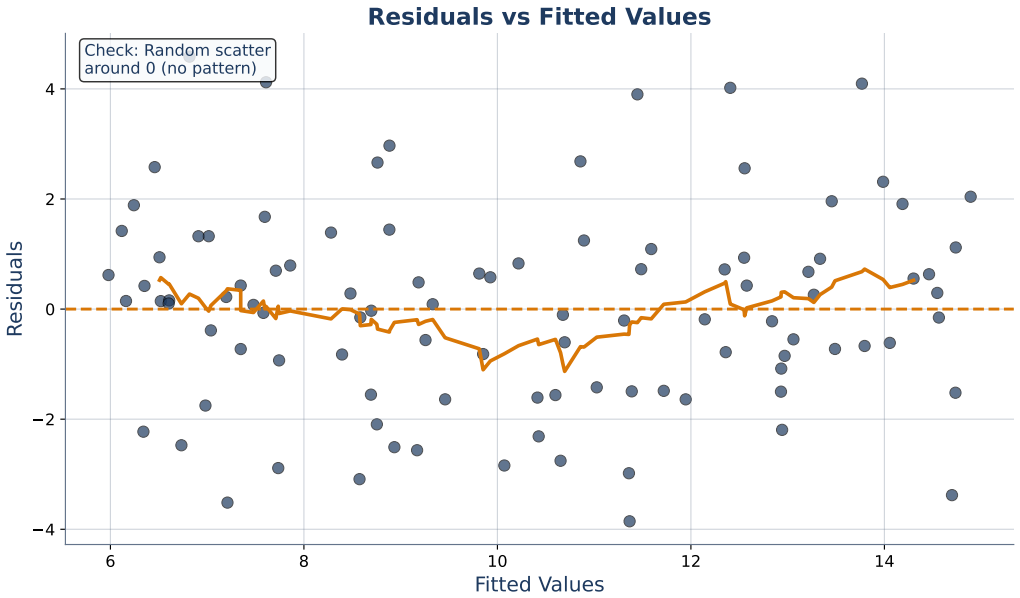


Figure 35: Residuals versus fitted values. A good fit shows a random cloud centered at zero. Curvature in the cloud indicates nonlinearity; a funnel shape indicates heteroscedasticity.

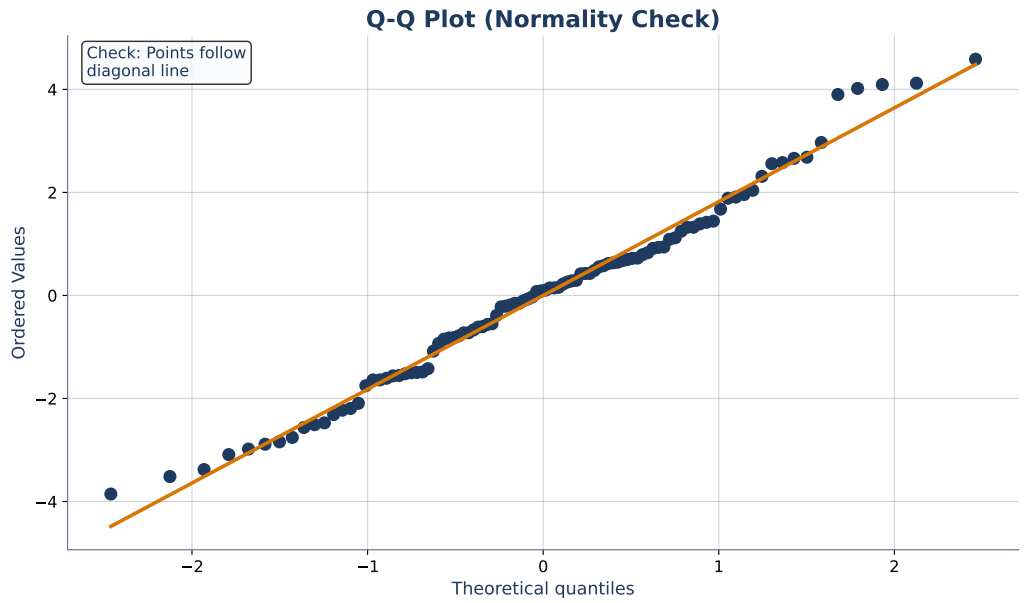


Figure 36: Quantile-quantile (Q-Q) plot of residuals versus a normal distribution. Points along the diagonal indicate normally distributed residuals; deviations indicate heavy tails or skew.

Average the K test scores to get the cross-validated score. Common choices: $K = 5$ or $K = 10$. Leave-one-out CV ($K = n$) gives the lowest-bias estimate but highest variance and highest compute cost.

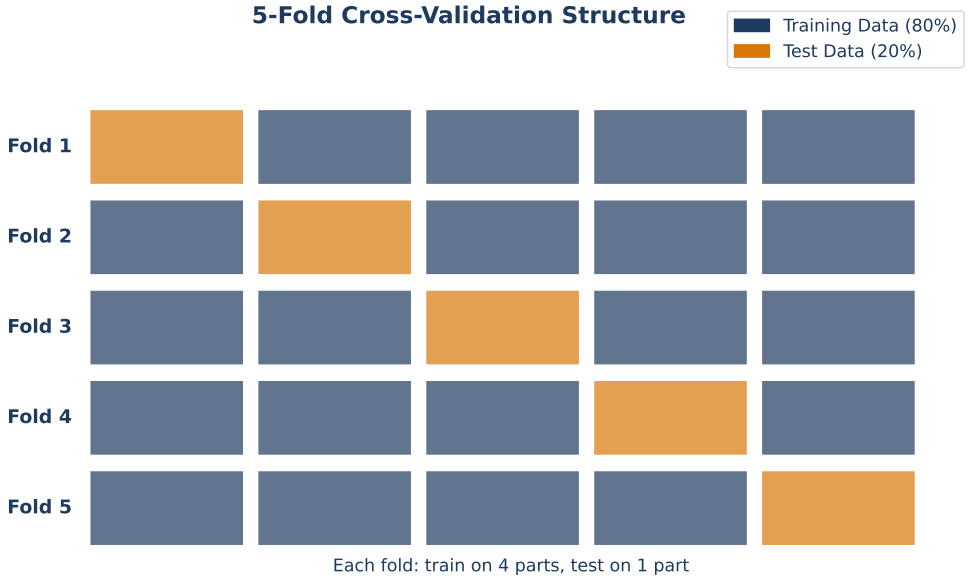


Figure 37: K-fold cross-validation. The data is split into K equal folds; each fold serves once as the test set while the rest form the training set. The K test scores are averaged.

Key Formula: K-Fold Cross-Validation

Given data $\mathcal{D}$ and an integer $K \geq 2$:

1. Partition $\mathcal{D}$ into K disjoint, equal-size folds $\mathcal{D}_1, \ldots, \mathcal{D}_K$.
2. For $k = 1, \ldots, K$: train a model on $\mathcal{D} \setminus \mathcal{D}_k$ and compute test error e_k on $\mathcal{D}_k$.
3. Report cross-validated error: $\bar{e} = \frac{1}{K} \sum_{k=1}^{K} e_k$.

The CV error $\bar{e}$ is an unbiased estimator of expected test error for a model trained on $n(1-1/K)$ observations. Its variance is typically lower than that of a single held-out test split.

Why K-Fold Breaks on Time Series

K-fold CV assumes i.i.d. data. For time series, this assumption is wrong in a specific and damaging way: random K-fold mixes past and future. A fold might contain data from 2020 in training and 2019 in the test set, so the model trains on the future to predict the past. The resulting CV score overestimates real production performance because the model has implicitly seen "future" information.

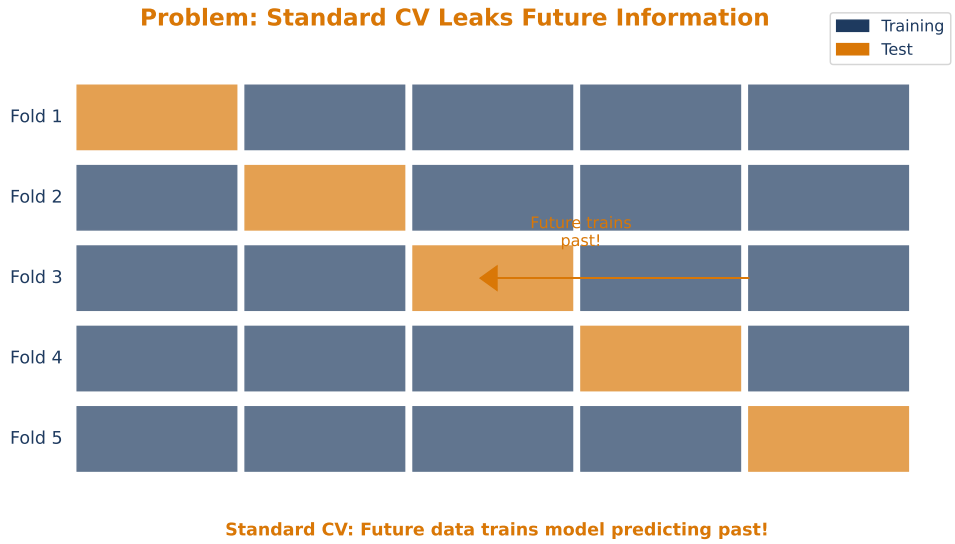


Figure 38: Standard K-fold CV on a time series: random folds mix past and future, producing optimistic estimates. A model validated this way will underperform in production.

Walk-Forward Cross-Validation

The fix is *walk-forward* (also called expanding-window) validation. Split the time series into K blocks in chronological order. For the first fold, train on block 1 and test on block 2. For the second, train on blocks 1–2 and test on block 3. Continue until all blocks have been used as test sets. The training window always *precedes* the test window in time.

scikit-learn implements this as `TimeSeriesSplit`. Use it for any time-ordered data.

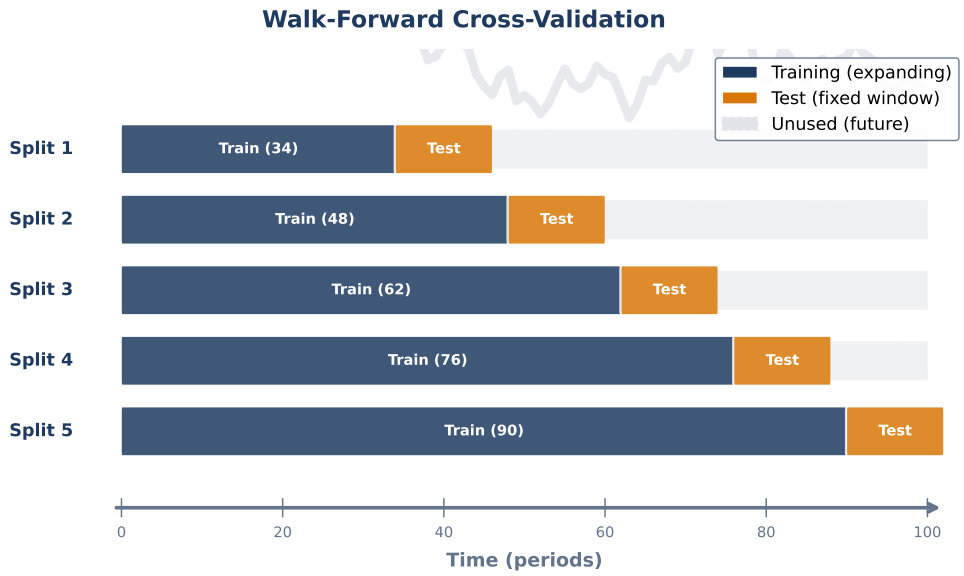


Figure 39: Walk-forward cross-validation. The training window always precedes the test window in time. Never trains on the future to predict the past.

Purging and Embargo

A subtler leakage problem appears when features are computed from *overlapping* windows. If feature x_t is a 30-day rolling average ending at t, then x_t and x_{t-5} overlap in their data. Training on x_{t-5} and testing on x_t leaks information.

Marcos Lopez de Prado's 2018 book *Advances in Financial Machine Learning* introduced *purged* cross-validation: drop training observations whose feature windows overlap with test observations. Add an *embargo*: skip a further window after the test period to prevent autocorrelation leaks. The combined Purged-Embargoed CV is the de-facto standard for serious financial ML.

Information Coefficient

A finance-specific metric worth mentioning is the Information Coefficient (IC), defined as the Spearman rank correlation between predicted and realized returns:

$$\text{IC} = \rho_{\text{Spearman}}(\hat{y}, y).$$

IC measures whether the model ranks stocks correctly, regardless of magnitude. A monthly IC of 0.05 is typical for a successful factor; 0.10 is strong; 0.15 is exceptional. The IC is often more stable than R^2 and more directly tied to portfolio performance (rank correctness matters more than magnitude for long-short strategies).

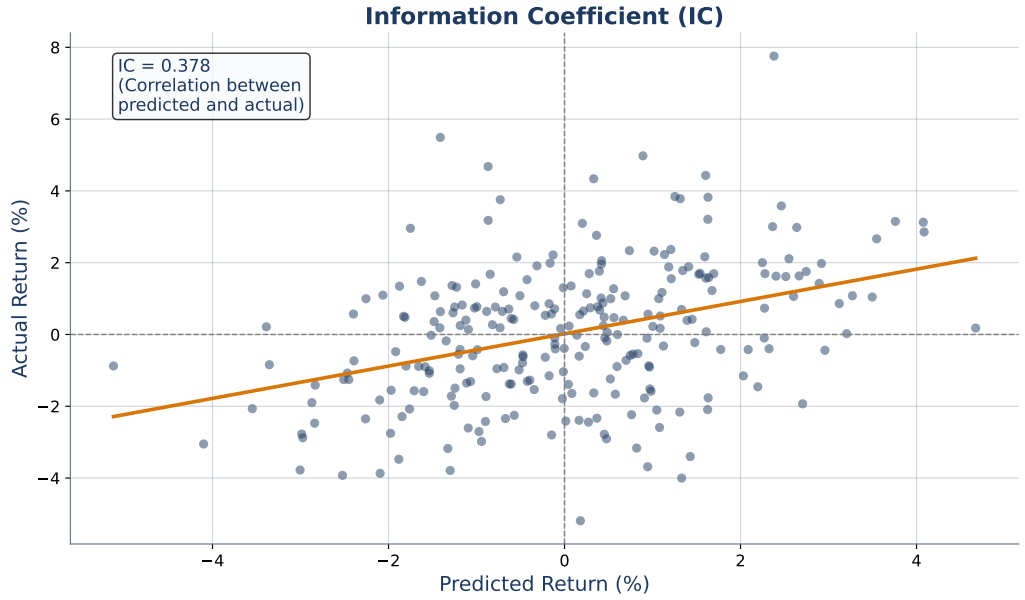


Figure 40: Information Coefficient (IC): Spearman rank correlation between predictions and realized returns. A monthly IC of 5–10% is a strong finance signal.

Definition: Regression Evaluation Metrics

Given observations $y_1, \ldots, y_n$ and predictions $\hat{y}_1, \ldots, \hat{y}_n$:

- **MSE** $= \frac{1}{n}\sum(y_i - \hat{y}_i)^2$. Scale: squared units. Sensitive to outliers.
- **RMSE** $= \sqrt{\text{MSE}}$. Scale: same as y. Most common summary.
- **MAE** $= \frac{1}{n}\sum|y_i - \hat{y}_i|$. Scale: same as y. Robust to outliers.
- $R^2 = 1 - \text{SS}_{\text{res}}/\text{SS}_{\text{tot}}$. Scale: unitless, $\in [-\infty, 1]$.
- **Adjusted** $R^2 = 1 - (1 - R^2)(n-1)/(n-p-1)$. Penalizes feature count.
- **Information Coefficient**: Spearman rank correlation. Used in finance for ranking models.

Common Misconceptions about Regression Metrics

(1) "Low R^2 means a bad model." Low R^2 means the signal-to-noise ratio is low. In finance and social sciences, this is normal. The question is whether the predictable component is large enough to be economically meaningful, not whether R^2 crosses some universal threshold.
(2) "Adding features always improves R^2." Plain R^2 can only increase or stay flat when you add features. Adjusted R^2 corrects this by penalizing feature count, but practitioners who report only R^2 can fool themselves.
(3) "K-fold CV is always the right choice." For time-series data, standard K-fold gives overoptimistic estimates because it mixes past and future. Always use walk-forward CV (`TimeSeriesSplit`) on time-ordered data.
(4) "RMSE and MAE tell the same story." RMSE is sensitive to outliers (squares errors); MAE is robust. On heavy-tailed data, they can give very different rankings of competing models.

Historical Background: Karl Pearson and Mervyn Stone

The coefficient of determination R^2 traces back to Karl Pearson (1895) and Francis Galton (1886), who first formalized correlation and regression. Pearson introduced R^2 as the "proportion of variance explained" in his 1897 paper on bivariate regression. The formula was a natural by-product of least squares: the total variance of y decomposes into an explained and an unexplained part, and R^2 measures the explained fraction.

Cross-validation is younger. Mervyn Stone published the foundational paper "Cross-Validatory Choice and Assessment of Statistical Predictions" in 1974 in the *Journal of the Royal Statistical Society*. The paper established leave-one-out CV as an asymptotically unbiased estimator of generalization error and introduced the K-fold generalization in the discussion. Seymour Geisser independently developed similar ideas. The computational cost was prohibitive in 1974; CV became routine only with modern CPUs.

The tailoring of CV to time series is a 2000s development. Hansen and Timmermann (2008) and others formalized walk-forward validation as the appropriate tool for financial data. Marcos Lopez de Prado's 2018 book pushed the state of the art further with purged and embargoed variants.

Regression Metrics and Cross-Validation in scikit-learn

```python
from sklearn.metrics import (
    mean_squared_error, mean_absolute_error, r2_score
)
from sklearn.model_selection import (
    cross_val_score, KFold, TimeSeriesSplit
)
from sklearn.linear_model import Ridge
import numpy as np

# Point metrics on a held-out split
y_pred = model.predict(X_test)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
mae  = mean_absolute_error(y_test, y_pred)
r2   = r2_score(y_test, y_pred)

# Standard K-fold (i.i.d. data only)
cv = KFold(n_splits=5, shuffle=True, random_state=42)
scores = cross_val_score(Ridge(alpha=1.0), X, y, cv=cv,
                         scoring='neg_root_mean_squared_error')
print(f'K-fold RMSE: {-scores.mean():.4f} +- {scores.std():.4f}')

# Walk-forward for time series
tscv = TimeSeriesSplit(n_splits=5)
ts_scores = cross_val_score(Ridge(alpha=1.0), X, y, cv=tscv,
                            scoring='neg_root_mean_squared_error'
                              )
print(f'Walk-forward RMSE: {-ts_scores.mean():.4f}')
```

Problem 4.1 (Easy) ⋆

Given the residuals $(0.2, -0.3, 0.1, -0.1, 0.5)$, compute MSE, RMSE, and MAE. Which metric is most affected by the largest residual (0.5)?

Solution: see Appendix.

Problem 4.2 (Easy) ⋆

Compute R^2 given MSE $= 0.20$ and $\text{Var}(y) = 1.0$. What does this value mean in plain English?
Solution: see Appendix.

Problem 4.3 (Medium) ⋆⋆

You have 10 years of daily stock returns (about 2,500 trading days) and a feature matrix of 50 technical indicators. Design a 5-fold walk-forward cross-validation scheme. For each fold, state the training window, test window, and any gap or embargo you would use, and justify.
Solution: see Appendix.

Problem 4.4 (Medium) ⋆⋆

A colleague fits a model to monthly stock returns and reports $R^2 = 0.05$. He concludes: "Our model is useless. 5% is nothing." Do you agree? Explain the context, cite typical benchmark R^2 values for stock returns, and describe conditions under which $R^2 = 0.05$ could support a profitable trading strategy.
Solution: see Appendix.

Problem 4.5 (Hard) ⋆⋆⋆

Write clean pseudocode for K-fold CV on a generic model. Then identify two sources of bias and two sources of variance in the CV estimator of generalization error. Discuss how the choice of K affects each. Why is $K = 5$ the common default, and when would you prefer $K = 10$ or leave-one-out?
Solution: see Appendix.

Connecting Forward

We now have a complete regression toolkit: fit with OLS or regularization, evaluate with RMSE/MAE/R^2, validate with K-fold or walk-forward CV. Section 5 changes the target: from continuous y (regression) to discrete y (classification). We will see how logistic regression extends linear regression to binary classification via the sigmoid squashing function, how cross-entropy replaces squared error as the natural loss, and how log-odds give us an interpretable coefficient scale for credit scoring and default prediction.

Key Takeaway: Pick metrics by task: RMSE/MAE/R^2 for magnitude-oriented problems, IC for rank-oriented problems; pick CV by data structure: K-fold for i.i.d., walk-forward for time series.

5. From Numbers to Categories – Logistic Regression

Opening Problem: 100,000 Loan Applications

You are the lead data scientist at a consumer bank. Every week the bank receives 100,000 loan applications. For each applicant the bank has features—credit score, debt-to-income ratio, employment length, loan amount, previous loan history—and wants a single number in return: the probability that this applicant will default within 24 months. Approve if the probability is below 5%; review if 5–15%; decline if above 15%.

Linear regression fails here for a specific reason. Suppose we encode default as $y = 1$ and no-default as $y = 0$ and fit a line. The line can predict $\hat{y} = -0.3$ for a safe applicant and $\hat{y} = 1.2$ for a risky one. Both predictions are *outside the probability range* $[0, 1]$. They also do not correspond to the actual probabilities the bank needs. The bank needs a model whose output is, by construction, a valid probability.

This section introduces logistic regression: the smallest change to linear regression that produces valid probabilities. We will derive the sigmoid squashing function, motivate the log-likelihood loss, interpret coefficients as log-odds, and walk through a numerical example of credit scoring.

Discovery Question

You want a model that outputs a probability in $[0, 1]$. The model's core computation, $\beta^\top x$, can produce any real number. What is the simplest function that maps $\mathbb{R}$ to $(0, 1)$ smoothly, monotonically, and symmetrically around zero? Why does the answer involve an exponential?

Why OLS Fails on Classification

Suppose we encode the label as $y \in \{0, 1\}$ and fit OLS. Two things go wrong:

- **Predictions escape $[0, 1]$.** A linear combination $\beta^\top x$ can be any real number. Interpreting $\hat{y} = -0.5$ or $\hat{y} = 1.3$ as a probability is nonsense.
- **Errors are structurally heteroscedastic.** The variance of a Bernoulli label y depends on its mean p: $\text{Var}(y) = p(1 - p)$. OLS assumes constant error variance, which is badly wrong here.

The fix is to squash the linear combination through a function σ whose range is $(0, 1)$. Linear regression becomes

$$\hat{y} = \sigma(\beta^\top x),$$

a *generalized linear model* with sigmoid link.

The Sigmoid Function

The sigmoid (also called the logistic function) is:

$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$

It maps $\mathbb{R}$ to $(0, 1)$, is smooth, strictly increasing, and symmetric around $z = 0$: $\sigma(0) = 0.5$, $\sigma(z) = 1 - \sigma(-z)$.

Why this particular function? Three reasons. First, its range is exactly $(0, 1)$, matching probabilities. Second, its derivative has a beautiful closed form $\sigma'(z) = \sigma(z)(1-\sigma(z))$, which simplifies

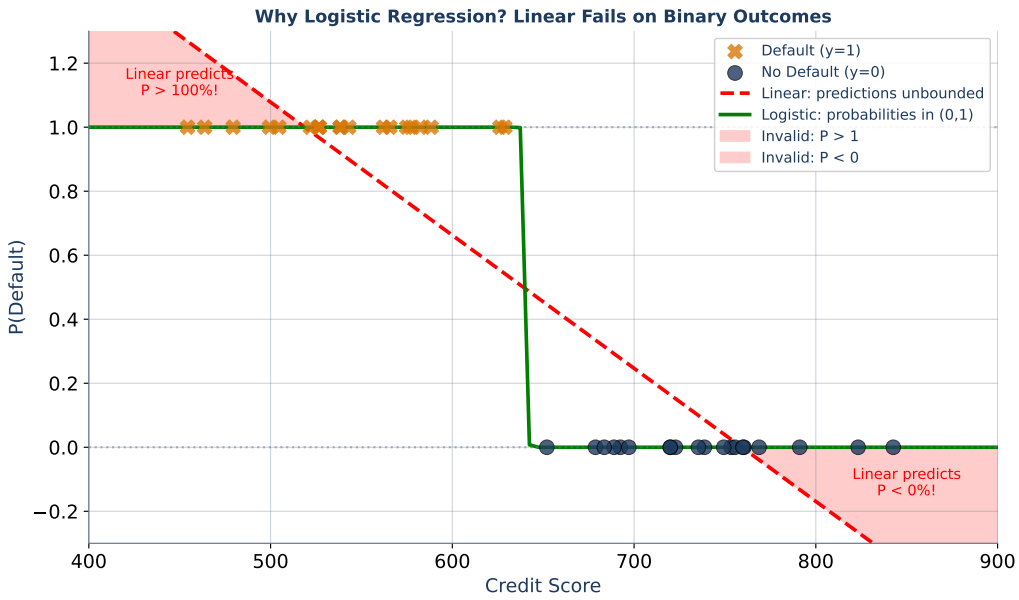


Figure 41: Linear regression (left) produces probabilities outside $[0, 1]$. Logistic regression (right) bends the output to stay in the valid range.

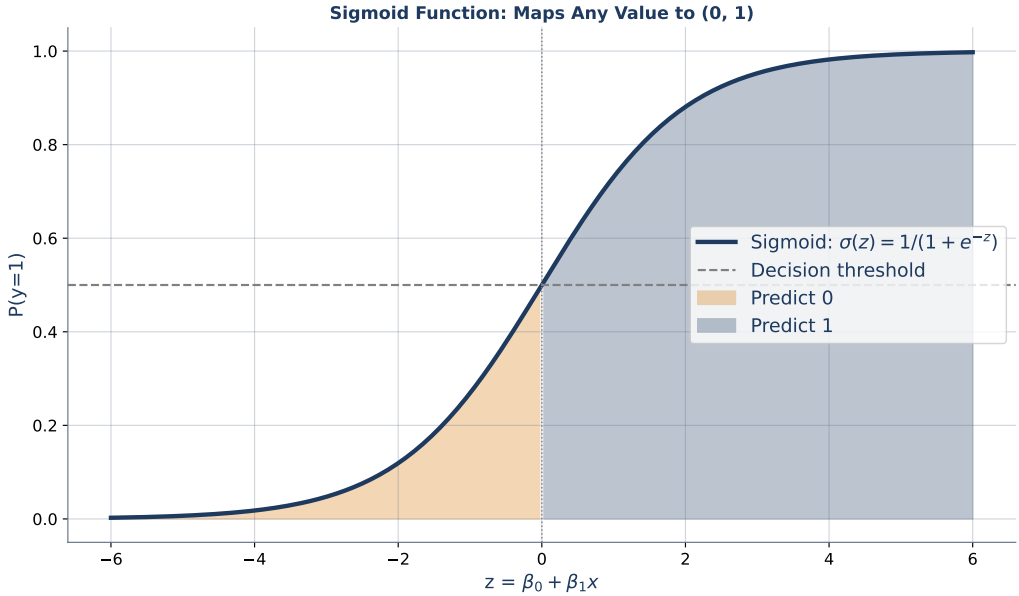


Figure 42: The sigmoid $\sigma(z) = 1/(1 + e^{-z})$ squashes any real number into $(0, 1)$. At $z = 0$, the output is exactly 0.5 (the decision boundary).

gradient-descent training. Third, it arises naturally from the maximum-likelihood derivation below: if you assume the class-conditional distributions are Gaussian with equal variance, the posterior probability of class 1 takes exactly the logistic form.

The Logistic Regression Model

Logistic regression is the model

$$P(y = 1 \mid x) = \sigma(\beta_0 + \beta^\top x),$$

with $P(y = 0 \mid x) = 1 - P(y = 1 \mid x)$. Fitting means choosing β_0, β to fit labeled data.

The *decision boundary*—the locus where $P(y = 1 \mid x) = 0.5$—is the set where $\beta_0 + \beta^\top x = 0$. This is a hyperplane. Logistic regression is thus a *linear* classifier: its decision boundary is a hyperplane in input space.

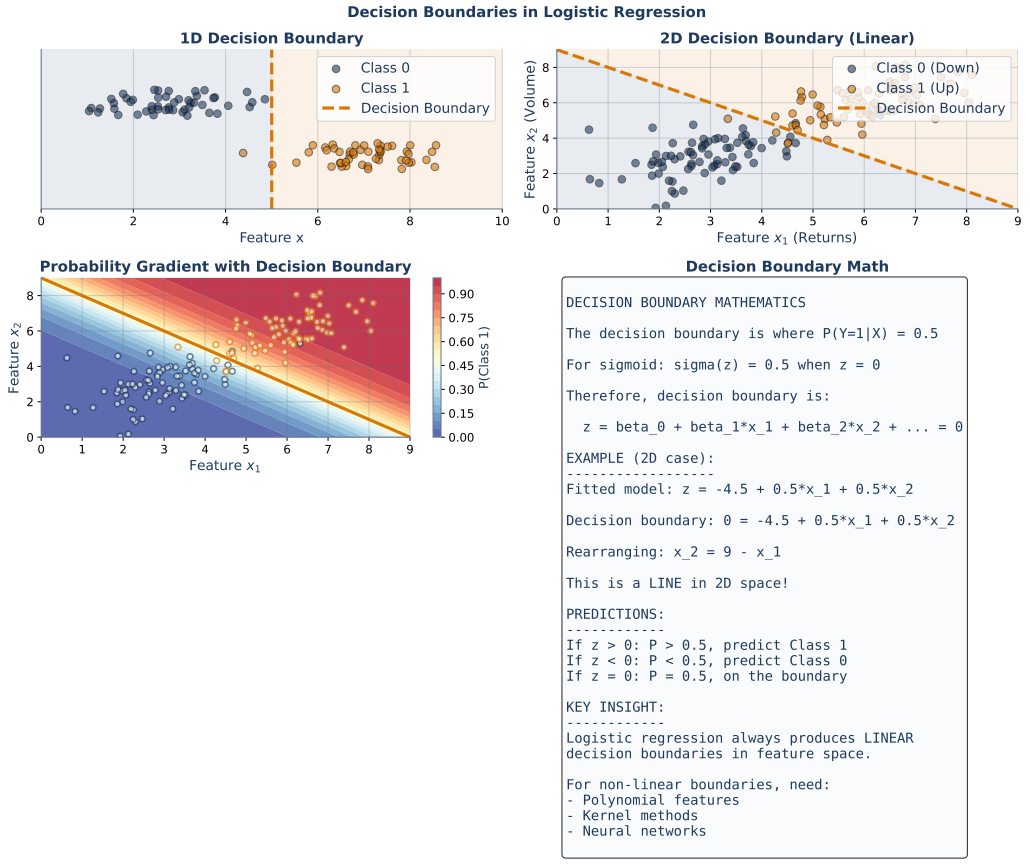


Figure 43: The logistic regression decision boundary is a hyperplane: the set of inputs for which $\beta^\top x + \beta_0 = 0$ (equivalently, $P(y = 1|x) = 0.5$).

Maximum Likelihood Estimation

Given labeled data $\{(x_i, y_i)\}_{i=1}^n$ with $y_i \in \{0, 1\}$, the likelihood of the observations under the logistic model is

$$L(\beta) = \prod_{i=1}^{n} P(y_i \mid x_i; \beta) = \prod_{i=1}^{n} p_i^{y_i}(1 - p_i)^{1-y_i},$$

where $p_i = \sigma(\beta^\top x_i)$. The log-likelihood is

$$\ell(\beta) = \log L(\beta) = \sum_{i=1}^{n} \big[y_i \log p_i + (1 - y_i) \log(1 - p_i)\big].$$

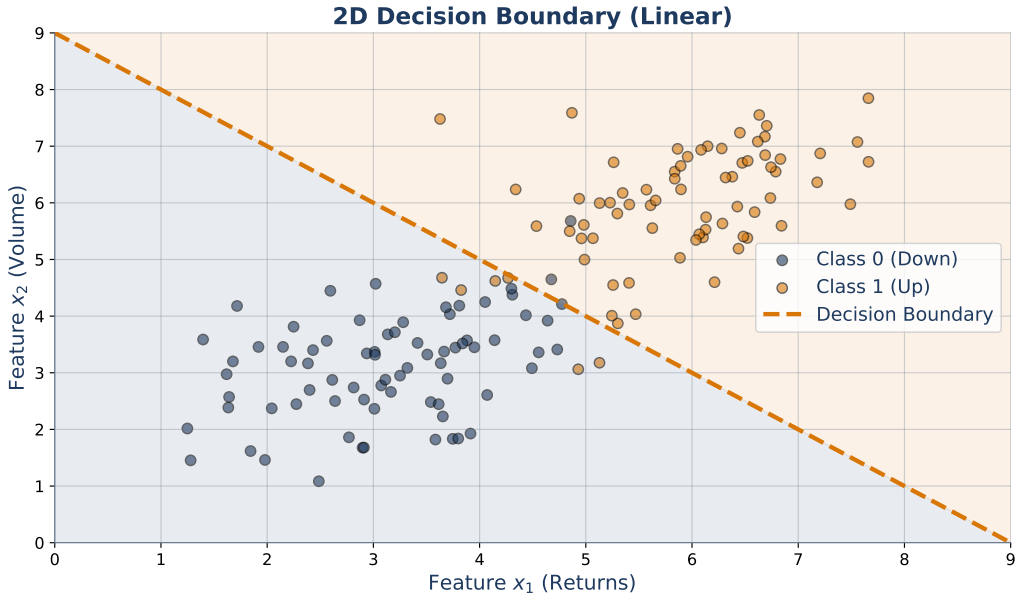


Figure 44: In 2D feature space, the logistic decision boundary is a straight line. Points on one side have $P(y = 1|x) > 0.5$, points on the other side have $P(y = 1|x) < 0.5$.

We want to maximize ℓ, or equivalently minimize the *negative log-likelihood*, which is exactly *cross-entropy loss*:

$$L_{\text{CE}}(\beta) = -\ell(\beta) = -\sum_{i=1}^{n} \big[y_i \log p_i + (1 - y_i) \log(1 - p_i) \big].$$

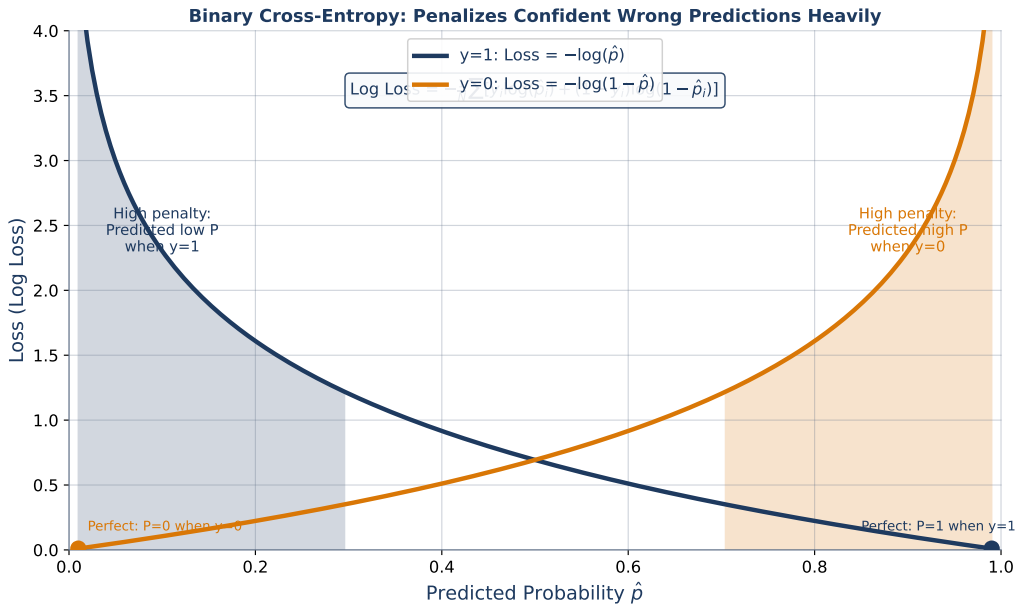


Figure 45: The cross-entropy loss explodes when the model is confident and wrong. It grows gently when the model is uncertain. This asymmetry penalizes overconfidence in bad predictions.

Key Formula: Logistic Regression

Given features $x \in \mathbb{R}^p$, parameters $\beta_0 \in \mathbb{R}$ and $\beta \in \mathbb{R}^p$:

- **Prediction**: $P(y = 1 \mid x) = \sigma(\beta_0 + \beta^\top x)$.
- **Loss (cross-entropy)**: $L = -\sum_{i=1}^{n}[y_i \log p_i + (1 - y_i)\log(1 - p_i)]$.
- **Fitting**: no closed form. Use gradient descent or its variants (Newton-Raphson, LBFGS).

Unlike OLS, logistic regression has no closed-form solution because the log-likelihood is transcendental in β. Modern implementations solve it with iterative optimization.

Deriving the Log-Likelihood from Bernoulli

Each observation is a Bernoulli trial: y_i is 1 with probability p_i and 0 with probability $1 - p_i$. The Bernoulli probability mass function is $P(y \mid p) = p^y(1 - p)^{1-y}$, which compactly represents both cases. For observation i:

$$\log P(y_i \mid p_i) = y_i \log p_i + (1 - y_i)\log(1 - p_i).$$

Summing over independent observations (i.i.d. assumption):

$$\ell(\beta) = \sum_{i=1}^{n} \log P(y_i \mid p_i) = \sum_{i=1}^{n}[y_i \log p_i + (1 - y_i)\log(1 - p_i)].$$

This is the log-likelihood used in maximum-likelihood estimation. Minimizing its negation is equivalent to maximizing the likelihood. The beautiful thing is that this loss—often called *binary cross-entropy*—is the canonical classification loss in machine learning from logistic regression to deep learning: a single concept spans the entire supervised classification literature.

The Gradient and Why No Closed Form Exists

Differentiating the log-likelihood with respect to β_j:

$$\frac{\partial \ell}{\partial \beta_j} = \sum_{i=1}^{n}(y_i - p_i)x_{ij}.$$

Setting to zero gives

$$\sum_{i=1}^{n}(y_i - p_i)x_{ij} = 0 \quad \text{for all } j.$$

Because $p_i = \sigma(\beta^\top x_i)$ depends nonlinearly on β, this system has no closed-form solution. We solve it iteratively. The most common methods are Newton-Raphson (uses second derivatives), BFGS/LBFGS (quasi-Newton), and stochastic gradient descent.

Odds and Log-Odds

The *odds* of an event with probability p is $p/(1 - p)$. The *log-odds* (or logit) is $\log[p/(1 - p)]$. Logistic regression has a beautifully simple interpretation in log-odds:

$$\log \frac{p}{1 - p} = \log \frac{\sigma(\beta^\top x)}{1 - \sigma(\beta^\top x)} = \beta^\top x.$$

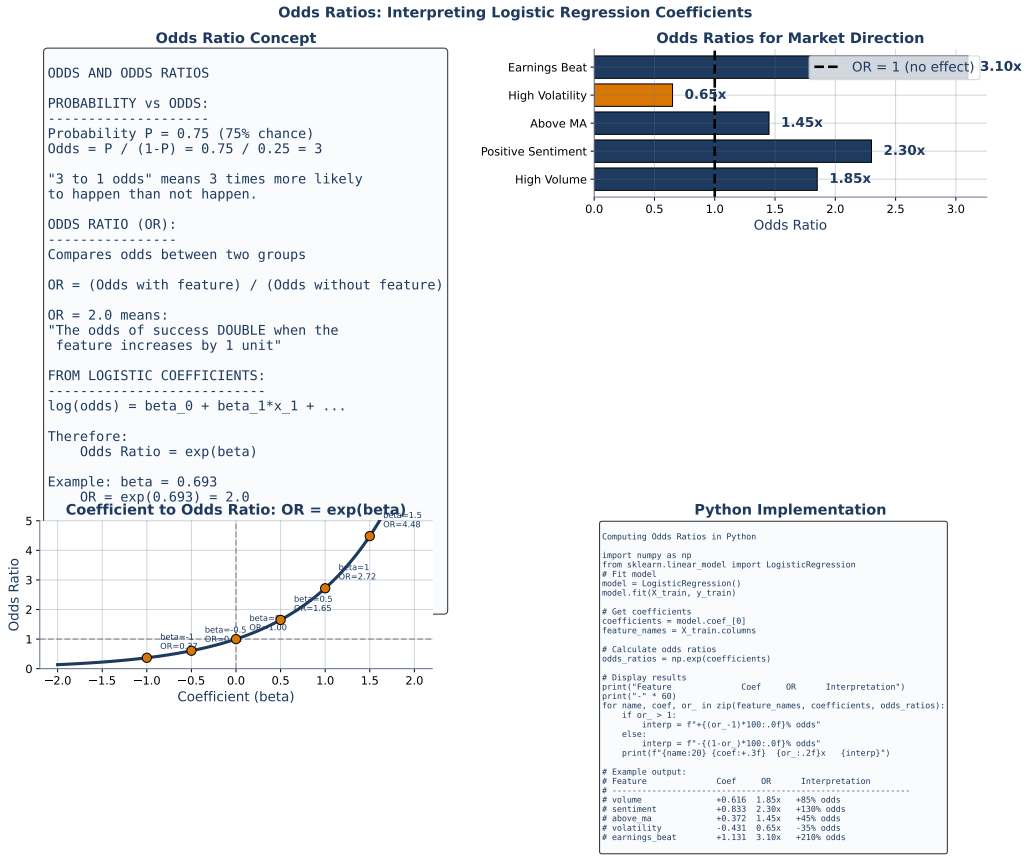


Figure 46: Odds ratios: $\exp(\beta_j)$ is the multiplicative change in odds of $y = 1$ per unit change in x_j. An odds ratio of 2 means the event is twice as likely for each unit increase in the feature.

The log-odds are a *linear* function of the features. Each coefficient β_j is the change in log-odds per unit change in x_j. An exponentiated coefficient e^{β_j} is the *odds ratio*: the multiplicative change in odds per unit change in x_j.

For credit scoring: if $\beta_{\text{credit\_score}} = -0.005$, then a 100-point increase in credit score changes log-odds by -0.5, i.e., odds ratio of $e^{-0.5} \approx 0.61$. Odds of default drop by 39% per 100-point credit-score increase.

A Worked Credit Scoring Example

Worked Example: Computing Default Probability

A logistic regression has been fitted for loan default prediction. The estimated coefficients are (intercept $\beta_0 = -3$, $\beta_{\text{DTI}} = 2.5$, $\beta_{\text{credit\_score}} = -0.005$). Features:

- DTI (debt-to-income ratio): 0.4 for this applicant.
- Credit score: 680 for this applicant.

Compute:

$$z = \beta_0 + \beta_{\text{DTI}} \cdot 0.4 + \beta_{\text{credit\_score}} \cdot 680 = -3 + 1 - 3.4 = -5.4.$$

Then

$$P(\text{default}) = \sigma(-5.4) = \frac{1}{1 + e^{5.4}} \approx \frac{1}{1 + 221.4} \approx 0.0045 = 0.45\%.$$

This applicant has a 0.45% predicted default probability. The bank's policy says approve if below 5%, so this loan is approved.
Now vary one feature: suppose the credit score were 580 instead.

$$z = -3 + 1 - 2.9 = -4.9, \quad P(\text{default}) = \sigma(-4.9) \approx 0.74\%.$$

Still below 5%—still approved. At credit score 500:

$$z = -3 + 1 - 2.5 = -4.5, \quad P(\text{default}) = \sigma(-4.5) \approx 1.1\%.$$

The probability rises by a factor of about 2.4 from the 680 scorer to the 500 scorer, even though the linear term only moved by 0.9. That exponential nonlinearity is the essence of the sigmoid.

Multi-class: Softmax

For $K > 2$ classes, the binary sigmoid generalizes to the *softmax* function:

$$P(y = k \mid x) = \frac{e^{\beta_k^\top x}}{\sum_{j=1}^{K} e^{\beta_j^\top x}}.$$

Each class has its own coefficient vector β_k. The exponential ensures positivity; the normalization ensures probabilities sum to 1. The gradient and training procedure mirror binary logistic regression.

An alternative for multi-class is *one-vs-rest* (OvR): fit K binary logistic regressions, one for each class against all others. Simpler but less principled than softmax.

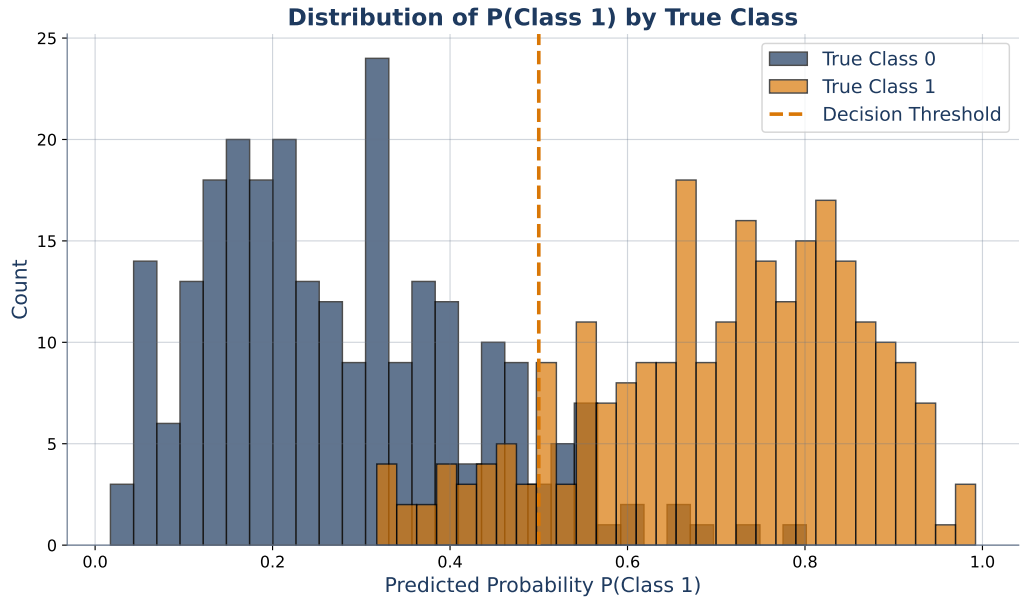


Figure 47: Predicted default probability as a function of credit score. The sigmoid shape means marginal effects are largest in the middle of the distribution.

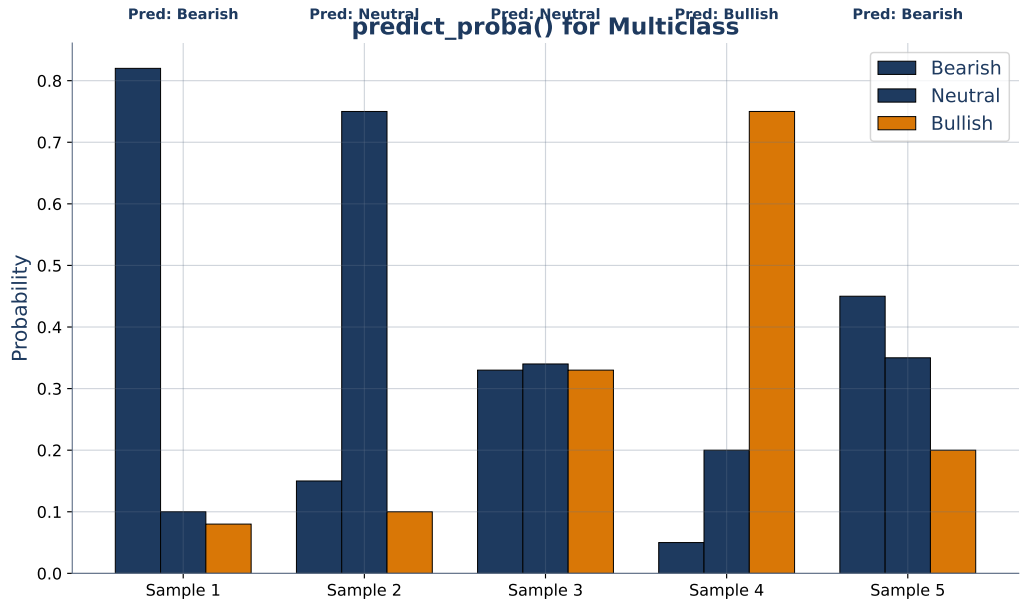


Figure 48: Softmax for 3-class classification: exponentiate each class's linear combination, then normalize so probabilities sum to 1.

Regularization for Logistic Regression

Just as linear regression has Ridge and Lasso, logistic regression supports L2 and L1 regularization:

$$L(\beta) = L_{\text{CE}}(\beta) + \lambda R(\beta),$$

with $R(\beta) = \|\beta\|_2^2$ (Ridge) or $\|\beta\|_1$ (Lasso). scikit-learn uses the parameter $C = 1/\lambda$, so large C means weak regularization.

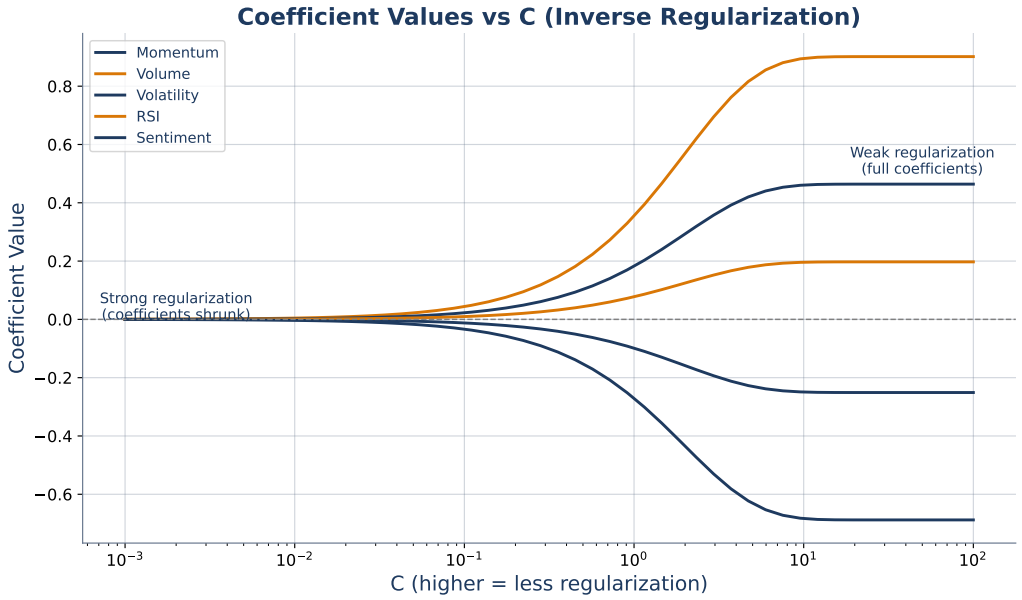


Figure 49: Effect of regularization on logistic regression. Strong regularization (small C) produces a smoother decision boundary; weak regularization lets the boundary wiggle to fit noise.

Definition: Logistic Regression

Logistic regression is a binary classification model that predicts

$$P(y = 1 \mid x) = \sigma(\beta_0 + \beta^\top x), \quad \sigma(z) = \frac{1}{1 + e^{-z}}.$$

Parameters β_0, β are fit by maximizing the log-likelihood (equivalently, minimizing cross-entropy) via iterative optimization. The decision boundary $\{x : \beta_0 + \beta^\top x = 0\}$ is a hyperplane, making logistic regression a linear classifier. For $K > 2$ classes, softmax replaces sigmoid; the per-class probabilities are

$$P(y = k \mid x) = e^{\beta_k^\top x} / \textstyle\sum_j e^{\beta_j^\top x}.$$

Common Misconceptions about Logistic Regression

(1) "Logistic regression is a regression algorithm." Despite the name, it is a *classification* algorithm. It estimates a conditional probability, but the goal is predicting a class.
(2) "Coefficients are probability changes." Coefficients are changes in *log-odds*, not probabilities. The actual probability change depends on the starting probability (marginal effects are largest at $p = 0.5$ and smallest at the extremes).
(3) "Logistic regression fits nonlinear boundaries." In the original feature space, the decision boundary is always a hyperplane. To fit nonlinear boundaries you must engineer nonlinear features (polynomials, interactions) or use a different model (trees, SVMs, neural networks).
(4) "OLS would work if we just thresholded the output at 0.5**."** OLS predictions can exceed $[0, 1]$ and treat all residuals symmetrically, which is statistically wrong for Bernoulli labels. Logistic regression is not just cosmetic; it encodes the right likelihood.

Historical Background: David Cox and the Logistic Model (1958)

The logistic function itself was introduced by Pierre-Francois Verhulst in 1838 to model population growth. In 1927, E. B. Wilson used it to model bioassay dose-response curves. These were applications of the curve as a descriptive tool, not a regression model.
The modern statistical framework is due to David Roxbee Cox, a British statistician who published "The Regression Analysis of Binary Sequences" in 1958 in the *Journal of the Royal Statistical Society*. Cox formalized logistic regression as a generalized linear model with log-odds link, derived the maximum-likelihood estimator, and developed the associated tests and diagnostics. The paper is one of the most cited in modern statistics. Logistic regression became the workhorse of binary classification in medicine (bioassay, epidemiology), credit scoring (since the 1960s FICO-era), and marketing (response modeling). It is still the most common model in credit and in clinical medicine because of its interpretability—odds ratios translate cleanly to stories regulators and clinicians can understand. Even in the deep-learning era, "logistic regression on good features" remains a competitive baseline that many modern models fail to beat on medium-sized tabular datasets.

Logistic Regression in scikit-learn

```python
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('model',  LogisticRegression(C=1.0, penalty='l2', max_iter
       =1000)),
])

pipe.fit(X_train, y_train)

# Class predictions (0 or 1) and probabilities
y_pred  = pipe.predict(X_test)
y_proba = pipe.predict_proba(X_test)[:, 1]   # P(y=1|x)

# Coefficients in log-odds space
print(f'Intercept: {pipe.named_steps["model"].intercept_}')
print(f'Coefs:     {pipe.named_steps["model"].coef_}')

# Odds ratios
import numpy as np
print(f'Odds ratios: {np.exp(pipe.named_steps["model"].coef_)}')
```

Problem 5.1 (Easy) ⋆

Given $\beta^\top x = 1.2$, compute $P(y = 1 \mid x)$. Then compute the corresponding odds and log-odds.
Solution: see Appendix.

Problem 5.2 (Easy) ⋆

A binary feature x_j has a logistic regression coefficient $\beta_j = \ln(2.5) \approx 0.916$. Interpret the odds ratio. If the baseline probability of $y = 1$ is 0.1, what is the new probability when x_j flips from 0 to 1?
Solution: see Appendix.

Problem 5.3 (Medium) ⋆⋆

Show that the logistic regression decision boundary $\{x : P(y = 1 \mid x) = 0.5\}$ is a hyperplane. Identify its equation in terms of β_0, β.
Solution: see Appendix.

Problem 5.4 (Medium) ⋆⋆

Derive the gradient of the binary cross-entropy loss with respect to β. Show that setting the gradient to zero reduces to $\sum_i (y_i - p_i)x_i = 0$ and explain why this system has no closed-form solution.
Solution: see Appendix.

Problem 5.5 (Hard) ⋆⋆⋆

Extend the derivation to the 3-class softmax. Write the per-class probabilities, the cross-entropy loss (with one-hot labels), and compute the gradient $\partial L/\partial \beta_k$. Verify that your formula reduces to the binary case when $K = 2$.
Solution: see Appendix.

Connecting Forward

Logistic regression is the linear classifier: a single hyperplane separates classes, with probabilities obtained via the sigmoid squashing function. Its coefficients read as log-odds, its loss reads as cross-entropy, and its fitting routine reads as gradient descent on a convex objective.

But linearity is also its limit. If the true decision boundary is curved (nonlinear interaction of features), no logistic regression can capture it without feature engineering. Section 6 tackles this by introducing *decision trees*: recursive partitioning algorithms that produce piecewise-constant decision boundaries, handle interactions automatically, and compose naturally into ensembles (random forests, gradient boosting). Trees are the main reason "classical" ML still dominates tabular data problems.

Key Takeaway: Logistic regression predicts probabilities via the sigmoid; it fits via maximum likelihood (cross-entropy); its coefficients are log-odds that read cleanly for credit, medicine, and marketing.

6. The Tree Family – Trees, Forests, Boosting

Opening Problem: An Interpretable Credit Model

A bank's chief risk officer walks into your office. "Your logistic regression for credit scoring is accurate but regulators keep asking why specific applicants were declined. Our compliance team needs to explain each decision in plain English. Show me a model where the logic is obvious—where I can look at an applicant and trace the path from features to decision without squinting at a coefficient matrix."

Logistic regression is interpretable but global: all features contribute to every decision through a single linear combination. For communicating individual decisions, a different structure helps: a sequence of if-then rules. "If credit score < 600, decline. Otherwise, if debt-to-income > 0.5, send to manual review. Otherwise, approve." This is a *decision tree*, and it is among the most transparent supervised learning algorithms.

This section builds decision trees from first principles, shows why a single tree tends to overfit, and introduces the two ensemble ideas that rescue tree models: *bagging* (random forests) and *boosting* (gradient boosting, XGBoost). The tree family dominates tabular machine learning in finance and business applications because it combines interpretability (or at least feature importance) with state-of-the-art accuracy.

Discovery Question

A decision tree fits by asking: "Which feature, split at which value, produces the purest child nodes?" But there are infinitely many possible splits. How does the tree actually decide, computationally, without trying every possibility? And why does a single tree overfit so dramatically that hardly anyone uses one tree anymore?

The Decision Tree Idea

A decision tree is a flowchart of binary decisions. Starting at the root, each internal node tests a single feature against a threshold ("credit_score < 600?"). Depending on the answer, we descend to the left or right child. We continue until we reach a leaf, which carries a predicted class (classification) or value (regression).

Training a tree means choosing the split at each node to maximally separate the classes or reduce prediction error. The CART algorithm (Breiman et al. 1984) greedily chooses the best split at each node without looking ahead.

Splitting Criteria: Gini, Entropy, MSE

At each node the algorithm needs a measure of "impurity"—how mixed the classes are—so it can pick the split that produces the purest children.

For classification, the two most common impurity measures are:

Gini impurity:

$$\text{Gini}(S) = 1 - \sum_{k=1}^{K} p_k^2,$$

where p_k is the fraction of samples in S belonging to class k. Gini is 0 for a pure node (all one class) and peaks when classes are equally distributed: Gini $= 0.5$ for two equal classes, ≈ 0.667 for three equal classes.

Entropy:

$$H(S) = -\sum_{k=1}^{K} p_k \log_2 p_k.$$

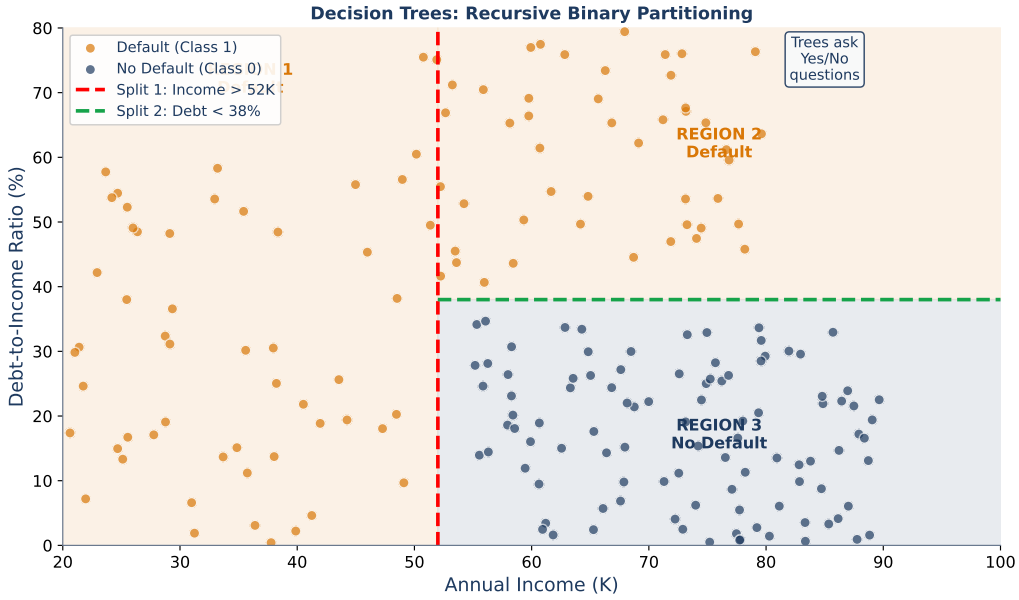


Figure 50: The decision tree intuition: a sequence of binary questions leads to a prediction. Each path from root to leaf is an if-then rule.

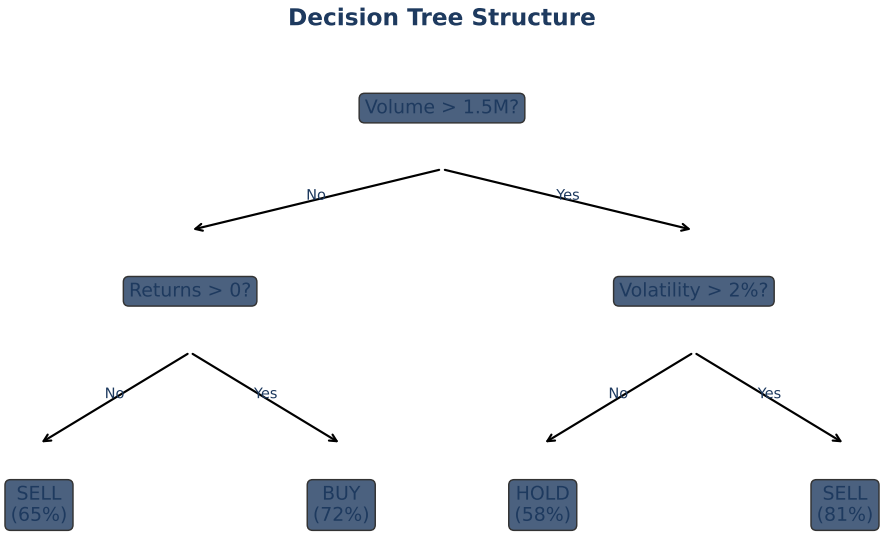


Figure 51: A tree with four internal nodes and five leaves. Each internal node tests one feature; each leaf outputs a prediction.

Entropy measures information content; it is 0 for pure nodes and 1 for two equally distributed classes.

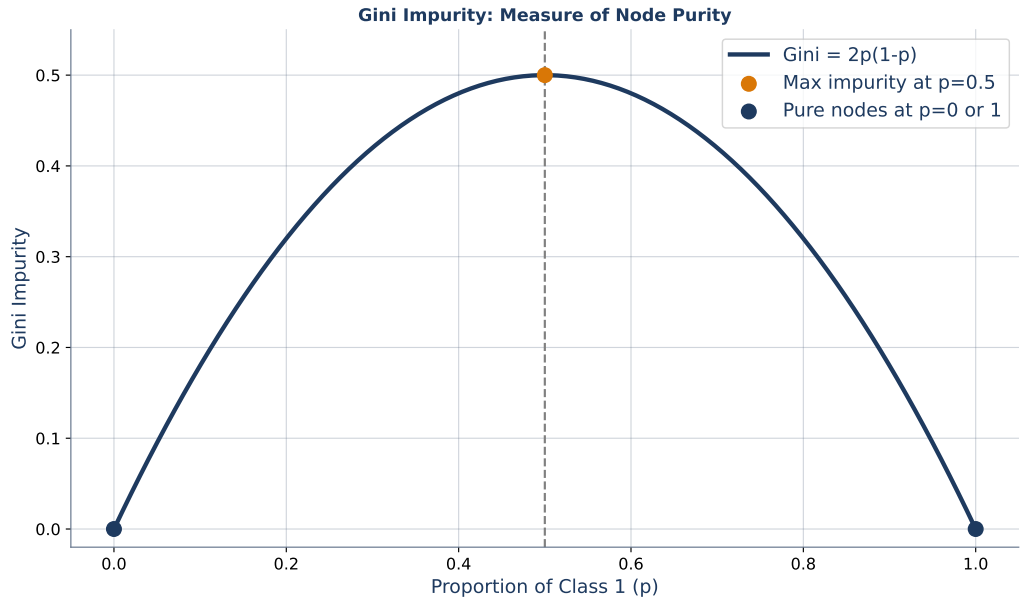


Figure 52: Gini impurity as a function of class proportion. Peaks at $p = 0.5$ (maximum uncertainty) and is zero at $p = 0$ or $p = 1$ (pure node).

Why do Gini and entropy give similar trees? Compute their derivatives at $p = 0.5$. For binary classification, Gini $= 2p(1-p)$ and its derivative is $2-4p$. Entropy's derivative at $p = 0.5$ is also zero, with similar curvature. Both functions are concave, both vanish at the extremes, and both peak at $p = 0.5$. Empirically, Gini and entropy produce trees that differ by a few splits, rarely in any meaningful way. Gini is slightly cheaper to compute because it avoids the logarithm.

For regression, the split criterion is MSE reduction: pick the split that produces child nodes with smallest total squared-error variance.

The Greedy CART Algorithm

The full algorithm is simple:

1. Start with all training data at the root.
2. For each candidate split (feature j, threshold t), compute the impurity of the left and right children. Weighted-average their impurities by the number of points in each child.
3. Pick the split that minimizes the weighted-average child impurity (equivalently, maximizes impurity reduction from parent to children).
4. Recurse on each child. Stop when a stopping criterion is met (max depth, min samples per leaf, or pure node).

The number of candidate thresholds per feature is finite: for a feature with distinct values $v_1 < v_2 < \ldots < v_m$, the sensible thresholds are the midpoints $(v_i + v_{i+1})/2$, giving $m - 1$ candidates. For p features and n samples per node, the per-node cost is $O(np)$ (after sorting), and the total tree cost is $O(np \log n)$ for a balanced tree.

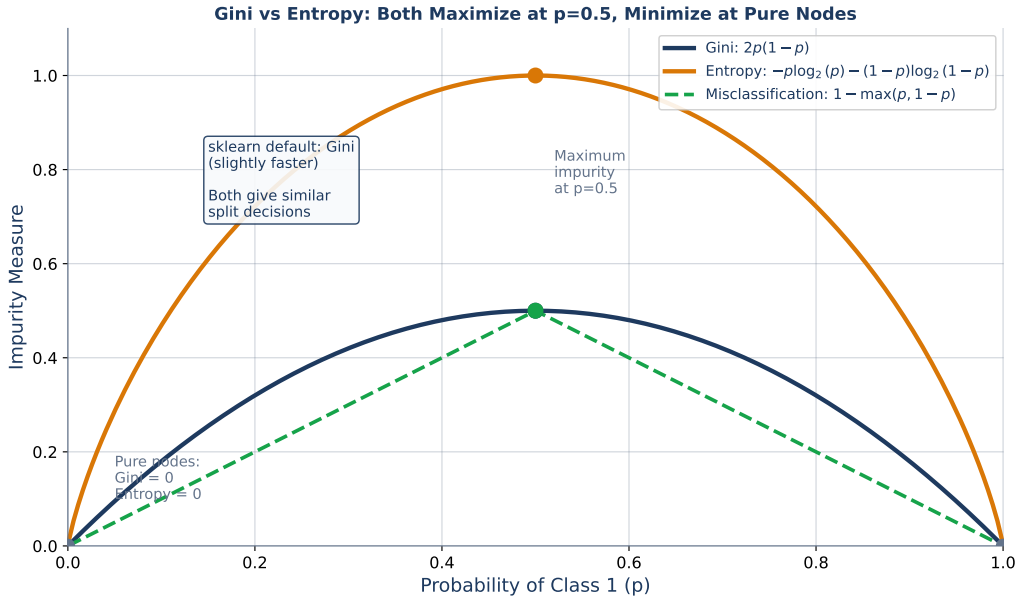


Figure 53: Gini vs entropy for a binary classification. Both peak at $p = 0.5$ and vanish at the extremes; they differ only slightly in curvature and rarely produce different trees in practice.

A Split-by-Hand Example

Worked Example: One Tree Split

Ten training samples at a node with labels: 6 positive, 4 negative. Gini of node: $1 - (0.6^2 + 0.4^2) = 1 - (0.36 + 0.16) = 0.48$.
Consider splitting on feature x_1 at threshold $t = 3.5$. Suppose this splits into a left child with 5 samples (4 positive, 1 negative) and a right child with 5 samples (2 positive, 3 negative).
Left Gini: $1-(0.8^2+0.2^2) = 1-0.68 = 0.32$. Right Gini: $1-(0.4^2+0.6^2) = 1-0.52 = 0.48$.
Weighted average: $(5/10) \cdot 0.32 + (5/10) \cdot 0.48 = 0.16 + 0.24 = 0.40$.
Impurity reduction: $0.48 - 0.40 = 0.08$.
Consider an alternative split at threshold $t = 5.5$. Suppose it produces a left child with 7 samples (6 positive, 1 negative) and a right child with 3 samples (0 positive, 3 negative).
Left Gini: $1 - ((6/7)^2 + (1/7)^2) = 1 - (0.735 + 0.020) = 0.245$. Right Gini: $1 - (0 + 1) = 0$ (pure!). Weighted average: $(7/10) \cdot 0.245 + (3/10) \cdot 0 = 0.1715$.
Impurity reduction: $0.48 - 0.1715 = 0.3085$.
The second split is far better (reduction of 0.31 vs 0.08). The algorithm will iterate over all candidate features and thresholds, pick the one with largest reduction, and recurse on the resulting children.

Overfitting in Decision Trees

A deep tree can memorize the training set exactly: with enough depth, every leaf contains a single training sample. The training error is zero, but the test error is catastrophic because the tree has memorized noise. This is the purest form of high-variance, low-bias behavior.

Three standard remedies:

- **Limit depth** (hyperparameter `max_depth`): prune before it overfits.
- **Minimum samples per leaf** (`min_samples_leaf`): refuse splits that leave tiny leaves.

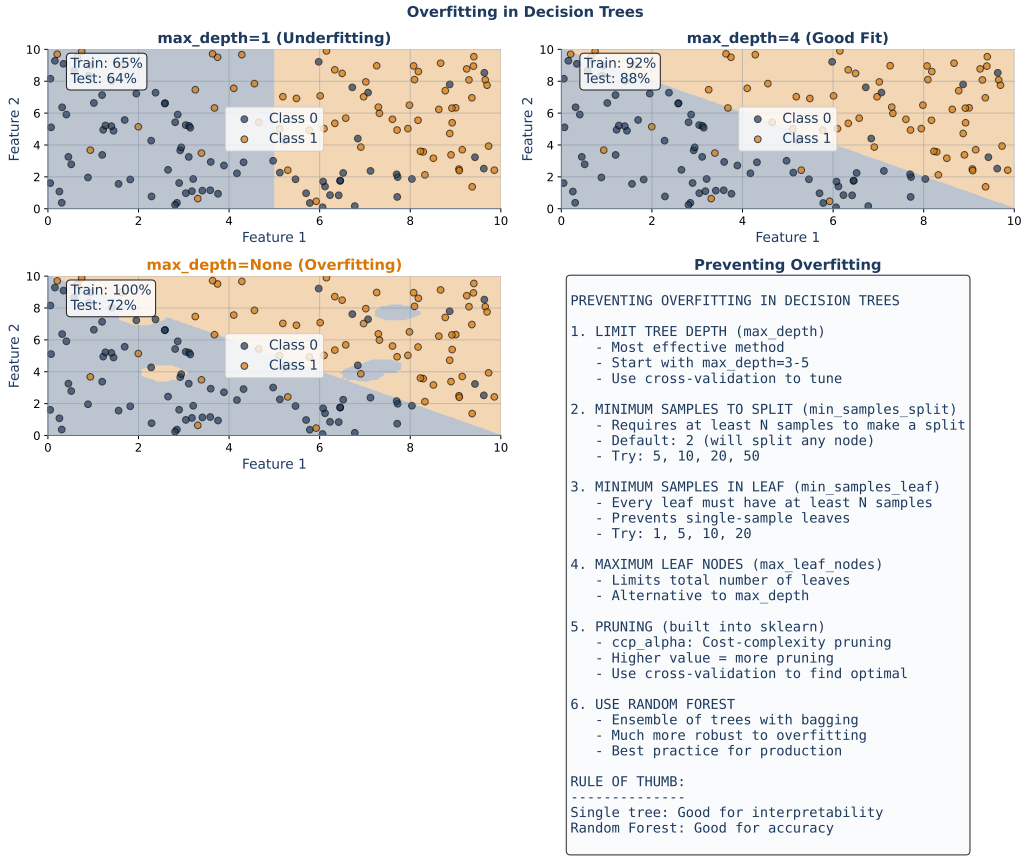


Figure 54: Tree depth versus test error: shallow trees underfit, very deep trees overfit. Between these extremes is a sweet spot, typically depth 3–8 for most tabular problems.

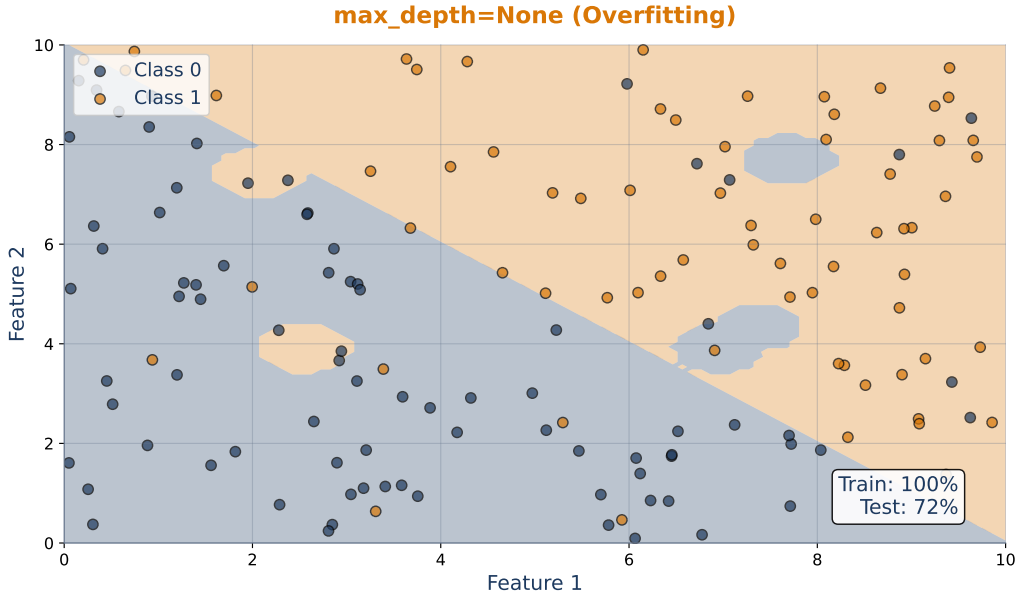


Figure 55: A very deep tree on 2D data produces jagged, overly specific boundaries that fit noise rather than structure.

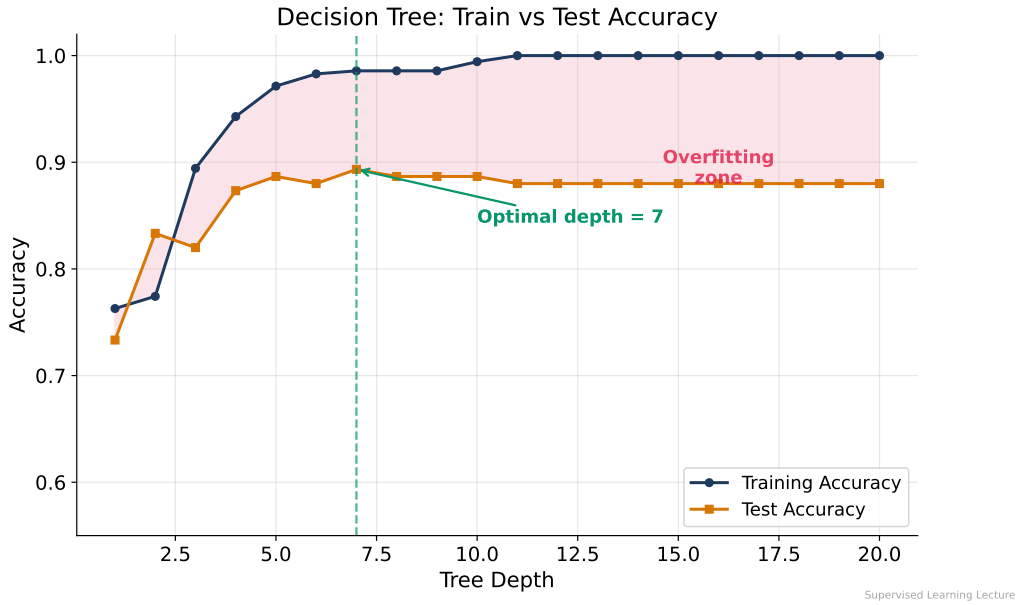


Figure 56: Decision boundaries for a single decision tree at three depths. Depth 3 underfits; depth 20 overfits; depth 6 is near-optimal.

- **Cost-complexity pruning** (`ccp_alpha`): fit a large tree, then prune back. Penalize the number of leaves.

Bagging and Random Forests

A single tree is high-variance but low-bias. If we average many *independent* trees, variance drops by a factor of $1/B$ (where B is the number of trees) while bias stays the same. This is the *bagging* idea (*bootstrap aggregation*, Breiman 1996).

But we can only get independent trees if we train them on different data. Bagging does this by generating *bootstrap samples*: draw n samples with replacement from the training set, fit a tree on the bootstrap, repeat B times, average the predictions. Each tree sees a slightly different dataset and learns slightly different splits.

Random forests (Breiman 2001) add one more idea: at each split, consider only a random subset of features. This decorrelates the trees further (without random feature subsets, every tree tends to split on the single strongest feature first, making trees too similar). Typical random forest: 100 trees, $\sqrt{p}$ features considered per split.

Why Bagging Reduces Variance but Not Bias

Let $\hat{f}_b(x)$ be the prediction of tree b. The bagged prediction is $\bar{f}(x) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}_b(x)$.

$$\mathbb{E}[\bar{f}(x)] = \mathbb{E}\left[\frac{1}{B}\sum_{b=1}^{B} \hat{f}_b(x)\right] = \frac{1}{B}\sum_{b=1}^{B} \mathbb{E}[\hat{f}_b(x)].$$

If each tree has the same bias (they are trained identically in expectation), then $\mathbb{E}[\hat{f}_b] = \mathbb{E}[\hat{f}_1]$ for all b, so $\mathbb{E}[\bar{f}] = \mathbb{E}[\hat{f}_1]$. The ensemble has the same bias as a single tree.

For variance:

$$\mathrm{Var}(\bar{f}(x)) = \mathrm{Var}\left(\frac{1}{B}\sum_{b=1}^{B} \hat{f}_b\right) = \frac{1}{B^2}\left[\sum_{b} \mathrm{Var}(\hat{f}_b) + \sum_{b\neq c} \mathrm{Cov}(\hat{f}_b, \hat{f}_c)\right].$$

Figure 57: Random forest: many trees trained on different bootstrap samples with random feature subsets. Predictions are averaged (regression) or voted (classification).

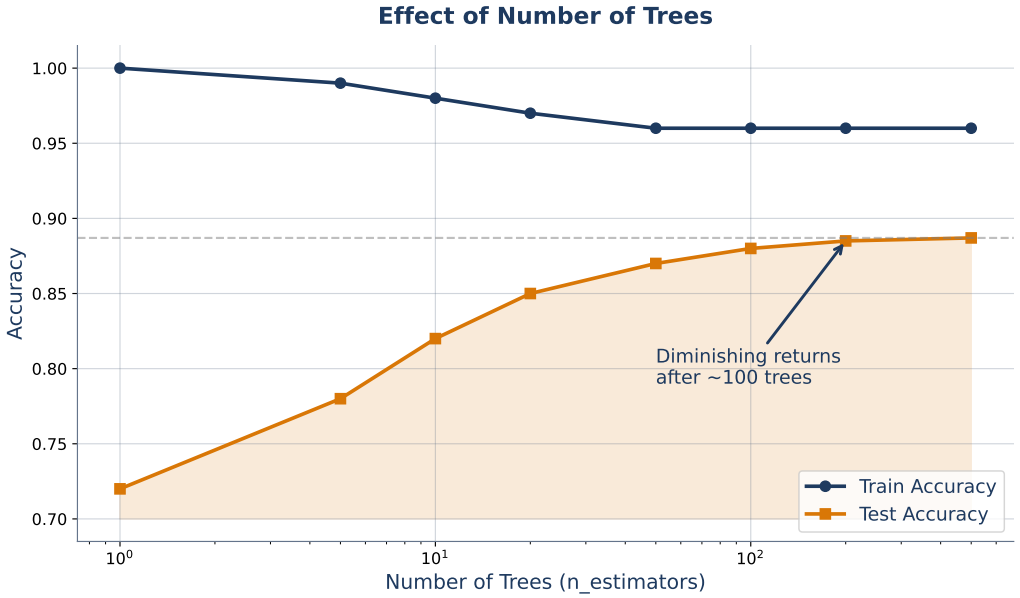


Figure 58: Random forest accuracy as a function of the number of trees. Accuracy climbs quickly in the first 20 trees, then flattens. Typical defaults: 100–500 trees.

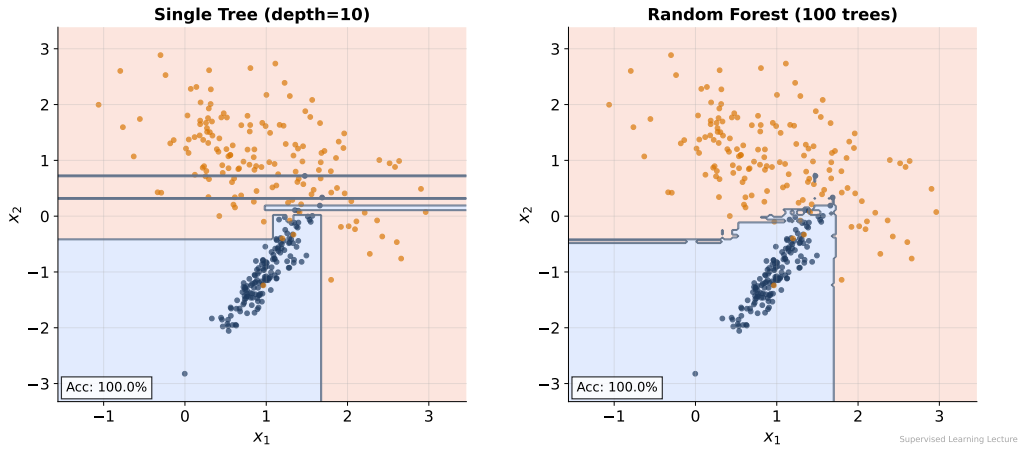


Figure 59: Decision boundaries: single tree (jagged, overfit) vs random forest (smooth, well-regularized). The ensemble averages out tree-level noise.

If trees were fully independent (covariance zero), variance would drop to $\text{Var}(\hat{f}_1)/B$. In practice, bootstrap samples overlap, so trees are correlated. The variance reduction is

$$\text{Var}(\bar{f}) = \rho\sigma^2 + \frac{1-\rho}{B}\sigma^2,$$

where ρ is the average pairwise correlation and σ^2 is the single-tree variance. As $B \to \infty$, variance approaches $\rho\sigma^2$—the floor set by correlation. Random feature subsets reduce ρ, pushing the variance floor lower.

OOB Error: Free Validation

Each bootstrap sample leaves roughly $1 - 1/e \approx 37\%$ of the training data *out of the sample*. For each observation, about 37% of the trees never saw it during training. Aggregate the predictions of these "out-of-bag" trees to get an honest test-set-like estimate of performance, without needing a separate validation set.

OOB error is a free gift of the bagging framework: no explicit validation set needed, yet you get an unbiased estimate of test performance. scikit-learn exposes it as `oob_score=True`.

Feature Importance

Trees produce a natural feature-importance score. For each feature j, sum the impurity reduction across all splits on j, weighted by the number of samples reaching each split. Features that split near the root with many samples and large impurity reduction rank highest.

An alternative is *permutation importance*: shuffle the values of one feature, re-evaluate the model, and measure the drop in accuracy. Permutation importance is model-agnostic and less biased toward features with many levels.

Gradient Boosting

Bagging averages independently trained trees. *Boosting* trains trees *sequentially*, where each new tree corrects the errors of the ensemble so far. Gradient boosting (Friedman 2001) formalizes this as gradient descent in function space.

The setup. Start with an initial prediction (e.g., the mean of y). At each iteration m:

1. Compute the negative gradient of the loss with respect to current predictions: for squared error, this is the *residual* $r_i = y_i - \hat{y}_i^{(m-1)}$.

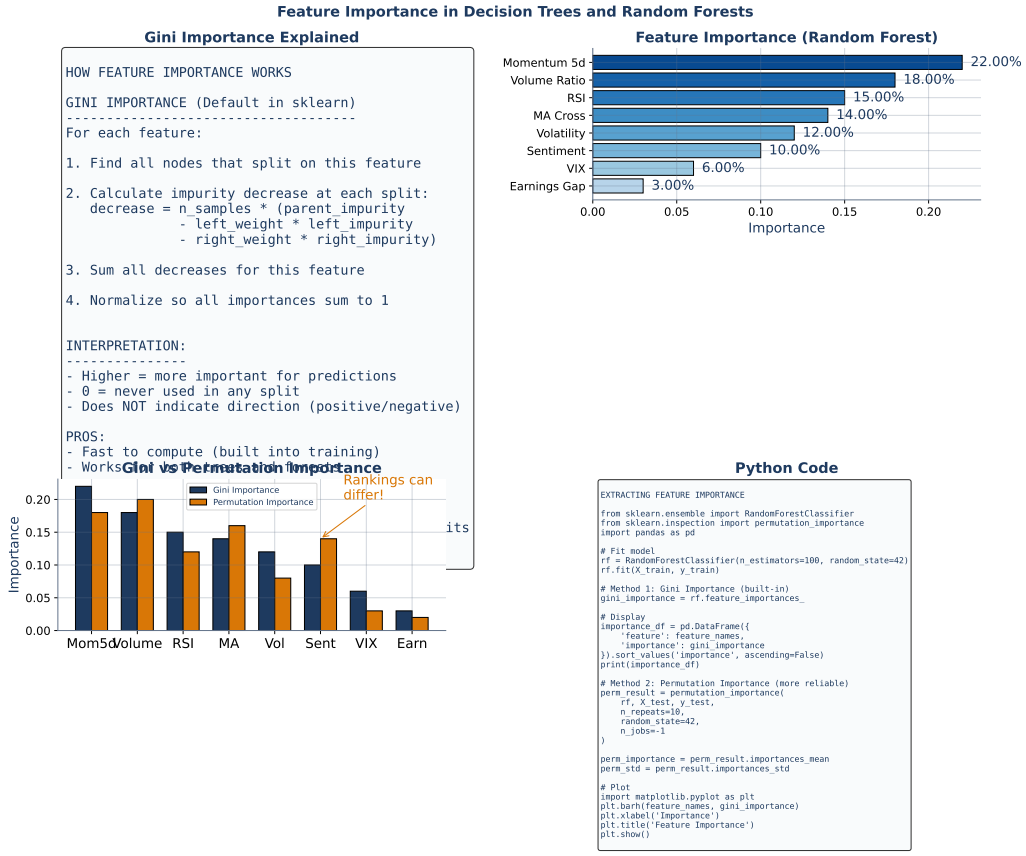


Figure 60: Feature importance bars from a random forest. Features are ranked by cumulative impurity reduction across all trees and splits.

2. Fit a small tree to predict these residuals.
3. Add the tree's prediction to the ensemble, scaled by a learning rate η: $\hat{y}_i^{(m)} = \hat{y}_i^{(m-1)} + \eta \cdot \text{tree}_m(x_i)$.

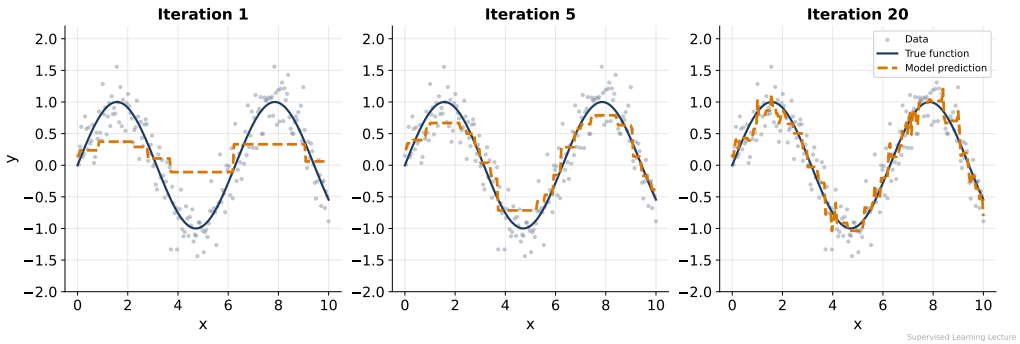


Figure 61: Gradient boosting as sequential residual fitting: each tree corrects the errors of the ensemble so far. The sum of all trees approaches the target.

The beautiful insight is that this is *gradient descent in function space*: each tree takes a step along the negative gradient of the loss, with tree structure chosen to best approximate that gradient. For squared loss, the gradient is the residual; for cross-entropy, it is something similar.

XGBoost and Modern Boosting

XGBoost (Chen and Guestrin, 2016) is the industrial-strength gradient-boosting library. It adds several engineering and algorithmic improvements:

- L1 and L2 regularization on leaf weights.
- A second-order (Taylor expansion) split-finding objective.
- Histogram-based splitting for speed on large data.
- Efficient handling of missing values and sparse features.
- Parallelization at the split level.

XGBoost and its siblings (LightGBM, CatBoost) dominate Kaggle competitions on tabular data and are the industry default for structured-data prediction in finance and e-commerce.

Why Boosting Targets Bias, Bagging Targets Variance

Here is the deep connection between tree ensembles and the bias-variance decomposition.

Bagging (random forests) takes high-variance, low-bias base learners (deep trees) and averages them to reduce variance while keeping bias unchanged. The result is a high-capacity model with lower variance.

Boosting takes low-variance, high-bias base learners (shallow trees, typically depth 3–6) and combines them sequentially to reduce bias while variance grows slowly. The result is a high-capacity model with low bias.

Both strategies land at ensembles with lower total error than a single tree, but through opposite mechanisms. In practice, gradient boosting with careful regularization (small trees, small learning rate, early stopping) is typically the higher-performing of the two on tabular data.

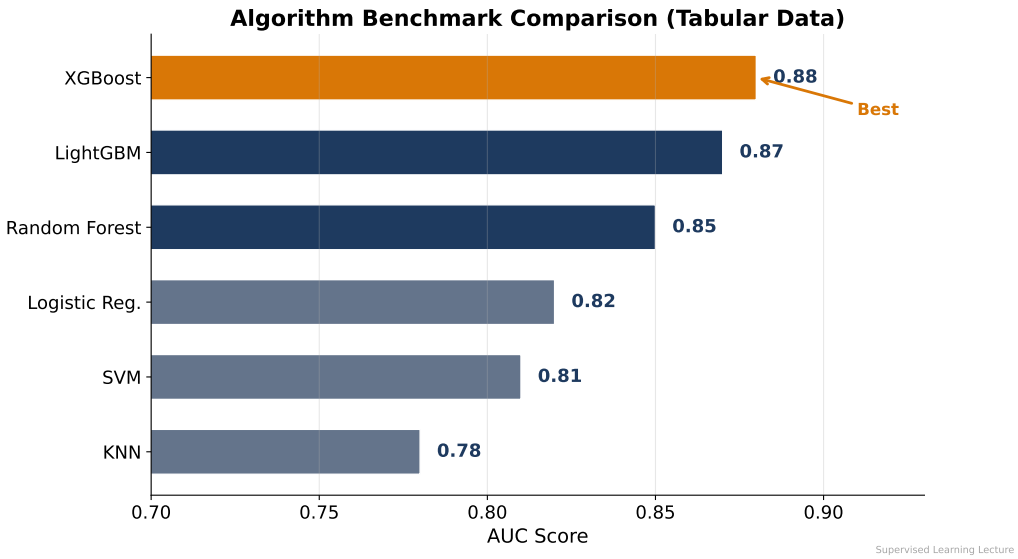


Figure 62: XGBoost benchmarks. On tabular data, XGBoost routinely outperforms logistic regression and matches or beats neural networks, at a fraction of the training time.

A Credit-Default Feature-Importance Study

Worked Example: Credit Features in a Random Forest

A retail bank fits a random forest on 100,000 loan records with 30 features. Out-of-bag accuracy: 87%. The top-5 features by importance:

Feature	Importance
Credit score	0.28
Debt-to-income ratio	0.18
Loan-to-value ratio	0.11
Employment length	0.08
Previous delinquencies	0.07

The top-5 features account for 72% of total importance. The loan officer can explain any decline as: "Credit score was low (28% of our model), and DTI was high (18%), pushing your default probability above our threshold."

If the bank wanted a simpler model for regulatory transparency, fitting logistic regression on just the top-5 features often achieves near-random-forest accuracy (say, 83% vs 87%) at the price of 4 percentage points, which may be worth the interpretability.

Definition: Decision Tree, Random Forest, Gradient Boosting

- **Decision tree (CART)**: a recursive binary partition of the feature space. Each internal node tests one feature against a threshold; each leaf outputs a prediction. Training greedily minimizes impurity (Gini, entropy, or MSE) at each split.
- **Random forest**: an ensemble of B decision trees, each trained on a bootstrap sample of the data and with a random subset of features considered at each split. Predictions are averaged (regression) or voted (classification). Reduces variance without increasing bias.
- **Gradient boosting**: a sequential ensemble where each new tree fits the negative gradient (residuals for squared loss) of the ensemble so far. Equivalent to gradient descent in function space. Reduces bias; modern implementations include XGBoost, LightGBM, CatBoost.

Common Misconceptions about Tree Models

(1) "Decision trees are interpretable." A shallow tree with 3–4 leaves is interpretable; a tree with 100 leaves is not. Beyond a handful of splits, you lose the "transparent reasoning" advantage.
(2) "Random forests are just averages of trees." They are averages of *decorrelated* trees. The bootstrap and random feature subsets are what make the ensemble work; without them you have correlated trees and variance does not drop.
(3) "More trees always help." In random forests yes (with diminishing returns). In boosting no—adding too many boosting rounds overfits. Early stopping based on a held-out set is critical for boosting.
(4) "Gini feature importance is unbiased." Gini importance is biased toward high-cardinality features (many distinct values) because they provide more candidate split points. Permutation importance is less biased.

Historical Background: Breiman, Friedman, and the Tree Revolution

Decision trees have a long history in rule-based AI (ID3, Quinlan 1986), but the modern CART algorithm (Classification and Regression Trees) was published by Leo Breiman, Jerome Friedman, Richard Olshen, and Charles Stone in 1984 as the book *Classification and Regression Trees*. CART introduced the binary splitting strategy, cost-complexity pruning, and the Gini/entropy impurity measures that remain the standard.

Bagging was Breiman's 1996 answer to the instability of single trees. He showed empirically that averaging trees trained on bootstrap samples could reduce variance dramatically. Random forests (Breiman 2001) added the random-feature-subsets trick, producing one of the most robust general-purpose ML algorithms.

Gradient boosting has two parents. Yoav Freund and Robert Schapire invented AdaBoost in 1996 as an exponential-loss boosting algorithm. Jerome Friedman generalized it in 2001 as "gradient boosting": a single framework that subsumes AdaBoost, L2Boost, LogitBoost, and arbitrary loss functions. Friedman's paper is one of the most cited in machine learning.

XGBoost (Chen and Guestrin, 2016) was the industrial leap. It combined Friedman's gradient-boosting math with aggressive engineering: parallelization, out-of-core learning, sparsity-aware splits, regularized objective. From 2015 to today, XGBoost and its successors (LightGBM by Microsoft, CatBoost by Yandex) have dominated virtually every Kaggle competition on tabular data. Breiman passed away in 2005 but Friedman remains active at Stanford. Their collective contribution is arguably the backbone of applied ML on tabular data.

Tree Family in scikit-learn and XGBoost

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
import xgboost as xgb

# Single decision tree
dt = DecisionTreeClassifier(max_depth=5, min_samples_leaf=10,
    random_state=42)
dt.fit(X_train, y_train)

# Random forest (100 trees, sqrt(p) features per split)
rf = RandomForestClassifier(n_estimators=100, max_depth=None,
                            max_features='sqrt', oob_score=True,
                            n_jobs=-1, random_state=42)
rf.fit(X_train, y_train)
print(f'OOB accuracy: {rf.oob_score_:.3f}')
print(f'Feature importances: {rf.feature_importances_}')

# XGBoost
xgb_clf = xgb.XGBClassifier(
    n_estimators=500, max_depth=4, learning_rate=0.05,
    subsample=0.8, colsample_bytree=0.8,
    reg_alpha=0.1, reg_lambda=1.0,
    early_stopping_rounds=20, eval_metric='logloss',
)
xgb_clf.fit(X_train, y_train, eval_set=[(X_val, y_val)], verbose=
    False)
```

Problem 6.1 (Easy) ⋆

Compute Gini impurity for a node with 70% class A and 30% class B. Compute the entropy for the same node (base-2 logarithm). Is the node pure?
Solution: see Appendix.

Problem 6.2 (Medium) ⋆⋆

A node has 12 samples: 8 positive, 4 negative. Three candidate splits produce children with positive/negative counts of (a) $(6/2, 2/2)$, (b) $(8/0, 0/4)$, (c) $(4/3, 4/1)$. Which split yields the largest Gini reduction? Compute explicitly.
Solution: see Appendix.

Problem 6.3 (Medium) ⋆⋆

Explain the OOB error mechanism: which samples are "out of bag" for tree b, what fraction of the training data is OOB for a typical tree, and why the average OOB prediction is approximately unbiased for the test error.
Solution: see Appendix.

Problem 6.4 (Hard) ⋆⋆⋆

Derive the variance-reduction formula for bagging:

$$\mathrm{Var}(\bar{f}) = \rho\sigma^2 + \tfrac{1-\rho}{B}\sigma^2,$$

where ρ is the pairwise correlation of the trees and σ^2 is the single-tree variance. Use this to explain why random feature subsets (in random forests, not just bagging) reduce the variance floor.
Solution: see Appendix.

Problem 6.5 (Hard) ⋆⋆⋆

Show that gradient boosting is gradient descent in function space for any differentiable loss L. Specifically, at iteration m with current prediction $F_{m-1}(x)$, the next tree h_m is fitted to the negative gradient $-\partial L/\partial F$ evaluated at F_{m-1}. For squared loss, show that this gradient is the residual $y - F_{m-1}(x)$.
Solution: see Appendix.

Connecting Forward

Decision trees are the second great family of classical supervised learning after linear models. Bagging and boosting turn single trees (which overfit) into ensembles (which generalize). On tabular data, random forests and gradient boosting routinely outperform both linear models and deep neural networks.

Section 7 completes the classical toolkit with three more methods: support vector machines (large-margin classifiers), K-nearest neighbors (lazy, distance-based prediction), and naive Bayes (probabilistic, simple, surprisingly effective). Each has its own niche. Together they give you the complete classical-ML arsenal.

Key Takeaway: Trees recursively partition feature space; bagging reduces variance, boosting

reduces bias; random forests and XGBoost remain state-of-the-art on tabular data.

7. Completing the Toolkit – SVM, KNN, Naive Bayes

Opening Problem: When Trees and Logistic Regression Fail

You have a small dataset: 200 observations, 300 features (gene expression data for a cancer classification task). Logistic regression overfits catastrophically because there are more features than observations. Random forests help but also struggle with the small sample size. What else is there?
You try a second dataset: 50,000 examples with only 3 features, but the decision boundary is a non-convex blob. Logistic regression draws a straight line; trees draw axis-aligned staircases; neither fits the blob well.
A third dataset: text documents with 10,000 word-count features, of which any single article uses only 100 or so. You want a fast, simple classifier that can be retrained nightly.
Each of these problems has a natural tool in the classical-ML toolkit that is not a linear model and not a tree. This section fills in the remaining three: support vector machines (SVMs) for the high-dimensional small-sample case, K-nearest neighbors (KNN) for the non-convex-blob case, and naive Bayes for the high-dimensional sparse-feature case.

Discovery Question

You are given two classes that cannot be separated by any line in their original 2D feature space. Yet if you lift the data to 3D by adding $x_3 = x_1^2 + x_2^2$ (distance from origin), a single plane separates them cleanly. Is there a way to *compute* the classifier in the higher-dimensional space without actually constructing the lifted features? The answer—the "kernel trick"—is one of the most beautiful ideas in machine learning.

Support Vector Machines: Maximum Margin Classifiers

SVMs were developed by Vladimir Vapnik and colleagues in the 1990s. The core idea: among all hyperplanes that separate two classes, pick the one that maximizes the *margin*—the distance to the closest training point on each side. Large-margin classifiers tend to generalize better than small-margin ones, under the intuition that a wide margin is more robust to small perturbations in the data.

Hard-Margin SVM (Derivation)

For linearly separable data, the hard-margin SVM solves

$$\min_{w,b} \tfrac{1}{2}\|w\|^2 \quad \text{subject to} \quad y_i(w^\top x_i + b) \geq 1,\ i = 1, \ldots, n,$$

where $y_i \in \{-1, +1\}$. The constraint says each training point must be on the correct side of the hyperplane, at least one unit away (after appropriate scaling). The objective minimizes $\|w\|$, which is equivalent to maximizing the margin $2/\|w\|$.

The support vectors are the points that sit exactly on the margin: $y_i(w^\top x_i + b) = 1$. They are the only training points that determine the classifier—remove non-support-vector points and the solution is unchanged.

Soft-Margin SVM

Real data are rarely linearly separable. Soft-margin SVM (Cortes and Vapnik, 1995) allows some points to violate the margin, with penalty:

$$\min_{w,b,\xi} \tfrac{1}{2}\|w\|^2 + C\sum_{i=1}^{n} \xi_i \quad \text{subject to} \quad y_i(w^\top x_i + b) \geq 1 - \xi_i,\ \xi_i \geq 0.$$

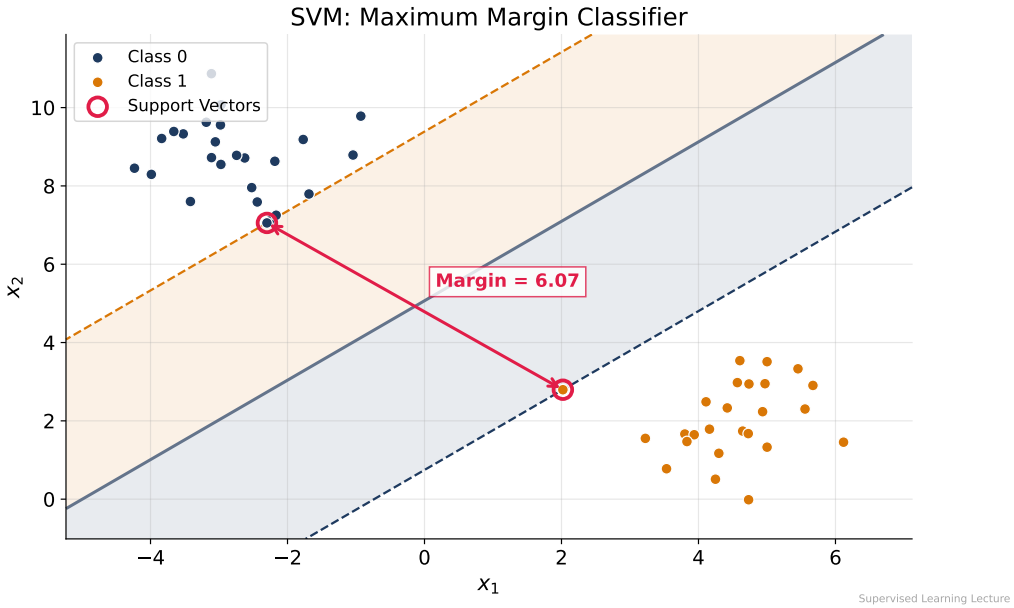


Figure 63: SVM maximum margin. The chosen hyperplane is the one whose distance to the closest training points (the support vectors) is as large as possible.

Here $\xi_i \geq 0$ is the amount by which point i violates the margin (0 if correctly classified by at least one unit of margin). The hyperparameter $C > 0$ trades off margin width against misclassifications: large C means strict (hard-margin limit); small C means tolerant (wider margin, more violations).

The Kernel Trick

The key insight that made SVMs famous: both the training and the prediction of an SVM can be written *entirely in terms of inner products* $x_i^\top x_j$. Replace these inner products with a *kernel function* $K(x_i, x_j)$, and the SVM effectively operates in the higher-dimensional space where K is an inner product.

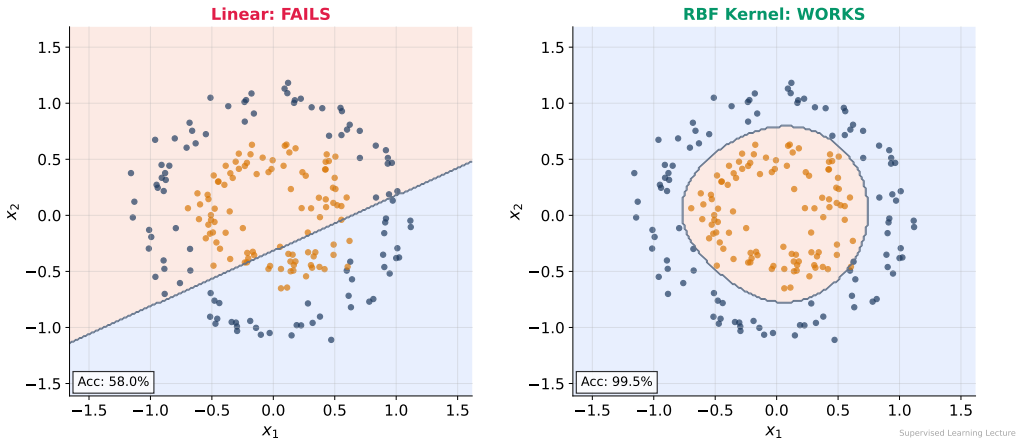


Figure 64: The kernel trick: a non-linearly-separable problem in 2D becomes linearly separable in a higher-dimensional feature space. The kernel function computes the inner product in that space without ever constructing the lifted features.

Common kernels:

- **Linear**: $K(x, x') = x^\top x'$ (no lifting; recovers linear SVM).

- **Polynomial**: $K(x, x') = (x^\top x' + 1)^d$ (lifts to degree-d polynomial features).
- **Radial Basis Function (RBF, Gaussian)**: $K(x, x') = \exp(-\gamma\|x - x'\|^2)$ (lifts to infinite-dimensional feature space).

The RBF kernel is the default in scikit-learn's SVM. It has one hyperparameter γ that controls the "locality" of the decision boundary: large γ makes narrow, overfitting boundaries; small γ makes broad, underfitting boundaries.

Worked Example: Kernel Trick by Hand

Suppose in 2D we define the map $\phi(x) = (x_1^2, \sqrt{2}x_1x_2, x_2^2)$, lifting to 3D. The inner product in the lifted space is

$$\phi(x)^\top \phi(x') = x_1^2x_1'^2 + 2x_1x_2x_1'x_2' + x_2^2x_2'^2 = (x_1x_1' + x_2x_2')^2 = (x^\top x')^2.$$

So the kernel $K(x, x') = (x^\top x')^2$ corresponds to this specific lifting. The SVM trained with this kernel is equivalent to a linear SVM applied to the 3D features $\phi(x)$—but we never have to compute $\phi(x)$ explicitly. For higher-degree polynomial kernels or the RBF kernel (which corresponds to infinite-dimensional feature space), this trick is essential: we could never store or compute the lifted features directly.

When SVMs Still Shine

SVMs dominated ML in the 1990s and early 2000s before being partially eclipsed by gradient boosting (for tabular) and deep learning (for unstructured). They remain competitive in three settings:

- **Small data with high dimensions**: Gene-expression studies (hundreds of samples, thousands of genes) where SVMs with RBF kernel often beat random forests.
- **Clear margin structure**: When classes are genuinely well-separated, the large-margin bias of SVMs gives excellent generalization.
- **Kernel design**: When domain knowledge suggests a specific similarity function (string kernels for text, graph kernels for molecules), SVMs plug it in directly.

K-Nearest Neighbors (KNN)

KNN is the laziest possible learning algorithm: *no training at all.* Given a new input x, find the K training points closest to x by some distance metric, and predict the majority class (classification) or the average label (regression) among those neighbors.

Choosing K

Small K (e.g., $K = 1$) is high-variance: a single mislabeled neighbor can flip the prediction. Large K (e.g., $K = n$) is high-bias: the prediction converges to the global class majority regardless of x. The sweet spot, typically $K \in [5, 50]$, is found by cross-validation.

Distance Metrics

KNN depends entirely on the distance metric. Common choices:

- **Euclidean**: $d(x, x') = \sqrt{\sum_j (x_j - x'_j)^2}$. The default.

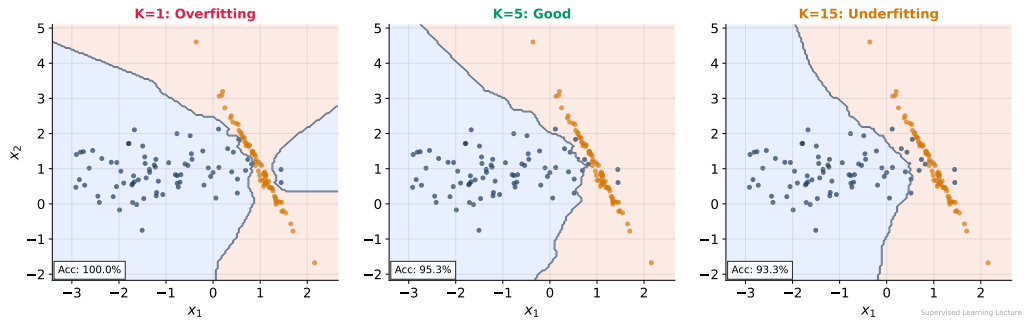


Figure 65: KNN decision boundaries for $K = 1$, $K = 5$, $K = 20$. Small K produces jagged, overfit boundaries; large K produces smoother boundaries that underfit if K is too large.

- **Manhattan** (L1): $d(x, x') = \sum_j |x_j - x'_j|$. More robust to extreme feature values.
- **Cosine**: $d(x, x') = 1 - \cos\theta$ where $\cos\theta = x^\top x' / (\|x\|\|x'\|)$. Used for high-dimensional sparse data (text).

Feature scaling is mandatory. Without standardization, features with large numerical ranges dominate the distance. If one feature is income in dollars and another is age in years, income differences will drown out age differences.

The Curse of Dimensionality

KNN performs badly in high dimensions. The reason is counterintuitive: in high-dimensional space, *all points become approximately equidistant*. Specifically, the ratio between the farthest and nearest neighbor distances tends to 1 as dimensionality grows. Nearest neighbors stop being meaningfully "nearest."

This "curse of dimensionality" hits KNN harder than tree-based methods because KNN relies on meaningful proximity. Typical guidance: KNN works well below 20 features, struggles between 20–100, and is near-useless above 1000 features unless combined with aggressive dimensionality reduction (PCA, feature selection).

Naive Bayes

Naive Bayes is probably the simplest supervised learning method that remains useful. It applies Bayes' rule to a simplifying ("naive") assumption that all features are conditionally independent given the class:

$$P(y \mid x) \propto P(y) \prod_{j=1}^{p} P(x_j \mid y).$$

The predicted class is the one that maximizes this posterior.

Variants

- **Gaussian Naive Bayes**: For continuous features, assume $P(x_j \mid y) = \mathcal{N}(\mu_{jy}, \sigma^2_{jy})$. Estimate class-conditional means and variances from training data.
- **Multinomial Naive Bayes**: For count features (e.g., word counts in documents), model each feature as a multinomial sample given the class. Standard for text classification.
- **Bernoulli Naive Bayes**: For binary features, model each as Bernoulli given the class.

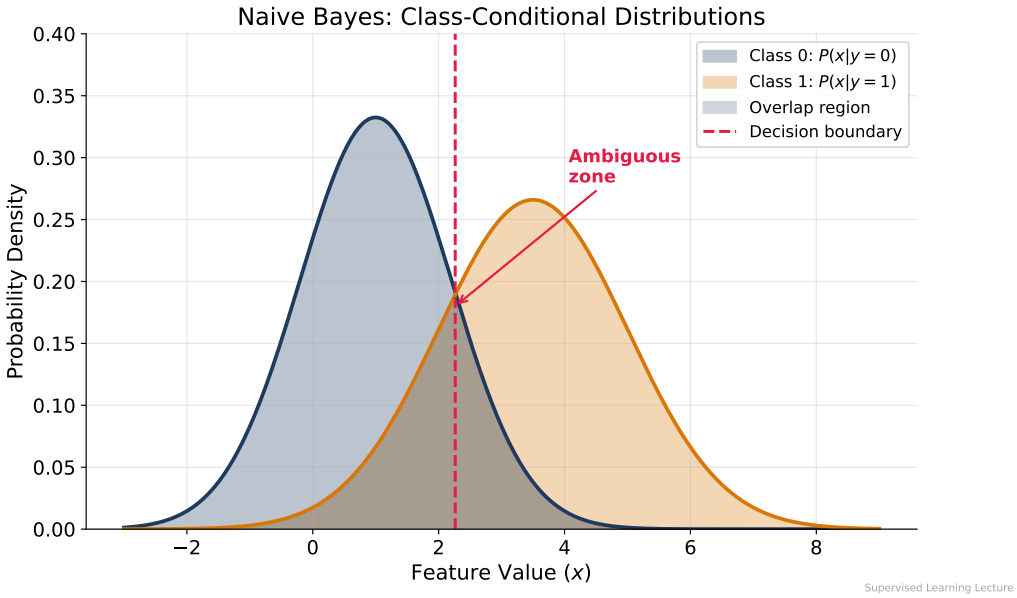


Figure 66: Naive Bayes' "naive" assumption: features are conditionally independent given the class. The joint probability factorizes into a product of per-feature probabilities.

Why Naive Bayes Works Despite Wrong Assumptions

The conditional-independence assumption is almost always wrong in real data—features are correlated within classes. Yet Naive Bayes often delivers surprisingly competitive performance. Why?

Zhang (2004) showed: even when conditional independence is violated, the *decision* (argmax over classes) can be correct as long as the errors in per-feature probability estimates roughly cancel across classes. For classification, you only need the correct *ranking* of class probabilities; the absolute values can be off. Naive Bayes often gets the ranking right even when the absolute probabilities are mis-calibrated.

This is why spam filters of the 2000s and early text classifiers relied heavily on Naive Bayes: fast, robust, nearly as accurate as more sophisticated methods for text-like data, and trivially parallelizable.

Naive Bayes in Text Classification

Worked Example: Email as Spam or Not

Two classes: *spam* (S) with prior $P(S) = 0.4$, *ham* (H) with $P(H) = 0.6$.
Three features (binary: word present or not):

- "offer": $P(\text{offer} \mid S) = 0.8$, $P(\text{offer} \mid H) = 0.1$.
- "meeting": $P(\text{meeting} \mid S) = 0.05$, $P(\text{meeting} \mid H) = 0.4$.
- "urgent": $P(\text{urgent} \mid S) = 0.6$, $P(\text{urgent} \mid H) = 0.2$.

A new email contains "offer" and "urgent" but not "meeting."

$$P(S \mid x) \propto P(S) \cdot P(\text{offer}|S) \cdot P(\text{not meeting}|S) \cdot P(\text{urgent}|S)$$

$$= 0.4 \cdot 0.8 \cdot (1 - 0.05) \cdot 0.6 = 0.4 \cdot 0.8 \cdot 0.95 \cdot 0.6 = 0.1824.$$

$$P(H \mid x) \propto 0.6 \cdot 0.1 \cdot (1 - 0.4) \cdot 0.2 = 0.6 \cdot 0.1 \cdot 0.6 \cdot 0.2 = 0.0072.$$

Normalize: $P(S \mid x) = 0.1824/(0.1824 + 0.0072) = 0.962$. Almost certain to be spam. The naive independence assumption is clearly wrong: "offer" and "urgent" probably co-occur in genuine spam more than product. But the classification is correct regardless.

Choosing Among Classical Methods

When should you use SVM, KNN, Naive Bayes, logistic regression, or trees? A rough guide:

Situation	First try	Runner-up
Tabular, $n > 1000$, $p < 100$	Gradient boosting	Random forest
Tabular, $n < 100$, $p > 1000$	SVM (RBF) or L1 logistic	Lasso + linear SVM
Text classification	Naive Bayes / linear SVM	Logistic regression
Very small n, low p	KNN or logistic	Naive Bayes
Low-dim dense, complex boundary	SVM (RBF) or KNN	Random forest
High-dim sparse (e.g., word counts)	Multinomial NB	Linear SVM

Definition: SVM, KNN, Naive Bayes

- **Support Vector Machine**: A maximum-margin classifier. For linearly separable data, solves $\min \frac{1}{2}\|w\|^2$ subject to $y_i(w^\top x_i + b) \geq 1$. Soft-margin variant tolerates violations with a penalty C. The kernel trick allows nonlinear boundaries by replacing $x^\top x'$ with $K(x, x')$.
- **K-Nearest Neighbors**: A lazy, non-parametric method. For a new input x, predict the majority class (classification) or average label (regression) of the K closest training points under some distance metric. No training phase; all work happens at prediction time.
- **Naive Bayes**: A probabilistic classifier based on Bayes' rule with the "naive" assumption that features are conditionally independent given the class. Estimates $P(x_j|y)$ from training data, multiplies them, and picks the class with maximum posterior.

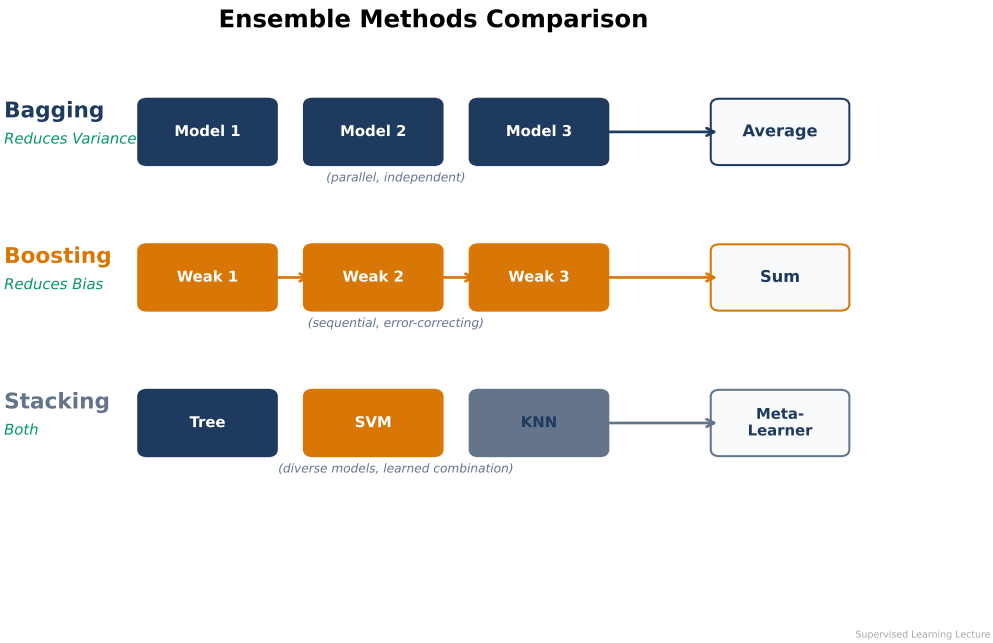


Figure 67: Classical-ML method comparison on five benchmark datasets. No single method dominates; the winner depends on data characteristics.

Common Misconceptions about SVM, KNN, NB

(1) "SVMs are always nonlinear." Linear SVM is the default for high-dimensional sparse data. RBF-kernel SVM is nonlinear but expensive; linear SVM scales to millions of features.

(2) "KNN has no training." It has no parameter learning, but the choice of K and distance metric are hyperparameters that must be tuned by cross-validation.

(3) "Naive Bayes assumes features are independent." It assumes *conditional* independence given the class, a weaker and more plausible assumption. Even so, the assumption is usually violated; the method works anyway because classification only requires correct ranking of class probabilities.

(4) "KNN works out of the box." Distance-based methods require feature scaling. Unscaled features produce distances dominated by the feature with the largest range—and nonsensical neighborhoods.

Historical Background: Cover, Hart, Vapnik

K-nearest neighbors traces back to Thomas Cover and Peter Hart's 1967 paper "Nearest Neighbor Pattern Classification" in *IEEE Transactions on Information Theory*. They proved a landmark result: the asymptotic error of 1-NN is at most *twice* the Bayes optimal error. KNN has no training, no parameters, and provable worst-case behavior—a rare combination.

Support vector machines emerged from the Soviet statistical-learning tradition. Vladimir Vapnik and Alexey Chervonenkis developed the theory of structural risk minimization in the 1960s and 70s. In 1992, Vapnik and colleagues (Boser, Guyon) introduced the kernel trick at COLT, allowing SVMs to handle nonlinear classification. In 1995, Cortes and Vapnik published "Support-Vector Networks" in *Machine Learning*, introducing the soft-margin SVM that handles noisy data. Vapnik emigrated to AT&T Bell Labs in 1990 and later to Facebook Research.

Naive Bayes dates to early Bayesian text classification work in the 1960s, but its modern use in computing was popularized by Paul Graham's 2002 essay "A Plan for Spam," which used Naive Bayes to build a highly effective spam filter. Within two years, every major email provider had deployed Naive Bayes filters. Graham's essay is a delightful read for the history.

These three methods represent three different intellectual traditions: information theory (KNN), statistical learning theory (SVM), and Bayesian inference (NB). Each is worth knowing because each succeeds where the others fail.

SVM, KNN, and Naive Bayes in scikit-learn

```python
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB, MultinomialNB
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

# RBF-kernel SVM (scale features!)
svm = Pipeline([
    ('scaler', StandardScaler()),
    ('model',  SVC(C=1.0, kernel='rbf', gamma='scale',
       probability=True)),
])
svm.fit(X_train, y_train)

# KNN (scale features!)
knn = Pipeline([
    ('scaler', StandardScaler()),
    ('model',  KNeighborsClassifier(n_neighbors=5, metric='
       euclidean')),
])
knn.fit(X_train, y_train)

# Gaussian Naive Bayes (no scaling needed)
gnb = GaussianNB()
gnb.fit(X_train, y_train)

# Multinomial Naive Bayes (for count features like TF-IDF)
# X must be non-negative
mnb = MultinomialNB(alpha=1.0)  # alpha is Laplace smoothing
mnb.fit(X_train_counts, y_train)
```

Problem 7.1 (Easy) ⋆

Explain why KNN has no training phase. What does "lazy learning" mean in practical terms, and what cost does the laziness impose at prediction time?
Solution: see Appendix.

Problem 7.2 (Medium) ⋆⋆

Given three support vectors and their coefficients (labels and dual weights) for a linear SVM, write down the equation of the hyperplane. Use a small concrete example: $(x_1, x_2, y) = (1, 1, +1), (2, 3, +1), (0, 0, -1)$ with equal weights, and derive w and b.
Solution: see Appendix.

Problem 7.3 (Medium) ⋆⋆

A simple text classification problem: three documents labeled as "positive" and three as "negative," with three words in the vocabulary. Compute $P(\text{class} \mid \text{document})$ for a new document using Multinomial Naive Bayes. Show all intermediate calculations.
Solution: see Appendix.

Problem 7.4 (Hard) ⋆⋆⋆

Derive the kernel trick for the polynomial kernel of degree 2: $K(x, x') = (x^\top x')^2$. Find the explicit feature map ϕ such that $K(x, x') = \phi(x)^\top \phi(x')$ for 2D inputs. How many dimensions does $\phi(x)$ have? What about for degree 3 in 2D inputs?
Solution: see Appendix.

Problem 7.5 (Hard) ⋆⋆⋆

Show that KNN suffers from the curse of dimensionality. Specifically, for uniform random points in a d-dimensional unit cube, compute (or argue heuristically) that the expected distance between any two points grows with d, and the ratio between farthest and nearest distances tends to 1. What does this imply about the usefulness of KNN as d grows?
Solution: see Appendix.

Connecting Forward

We have now covered the full classical supervised-learning toolkit: linear regression, regularized regression, logistic regression, trees, forests, boosting, SVMs, KNN, and Naive Bayes. Section 8 turns to the meta-skill: how do you evaluate classification models, especially when classes are imbalanced? How do you avoid data leakage, handle concept drift, and deploy a model to production without shooting yourself in the foot?

Key Takeaway: SVM dominates small-n, high-p problems via maximum margins and kernels; KNN excels on low-dim smooth boundaries but suffers in high dimensions; Naive Bayes remains the simplest useful classifier and is the baseline for text.

8. Judging Models – Classification Metrics, Imbalance, Production

Opening Problem: A 99.5%-Accurate Fraud Model

The product manager bursts into your office. "Our new fraud model has 99.5% accuracy on the test set! This is huge—we're ready to deploy." You ask a quick clarifying question: "What fraction of transactions are actually fraudulent?" She checks. "About 0.5%." You sigh.

A classifier that predicts "not fraud" for every transaction achieves 99.5% accuracy on this dataset—without detecting a single fraud. The product manager's "huge" model might be literally that dumb. You need metrics that are not fooled by class imbalance.

This final section tackles the full battery of classification metrics (confusion matrix, precision, recall, F1, ROC, AUC, PR curves), handles class imbalance (class weights, SMOTE, threshold tuning), and addresses the subtle but deadly production issues that turn 99.5%-accurate prototypes into worthless deployments: data leakage, label leakage, concept drift, and the mechanical importance of sklearn pipelines.

Discovery Question

You change a model's decision threshold from 0.5 to 0.3 and its precision drops from 0.9 to 0.6, while recall rises from 0.4 to 0.8. Which setting is "better"? Can one model be strictly better than another, or is it always a tradeoff? What decides which point on the curve you should pick?

The Confusion Matrix

The confusion matrix organizes predictions vs truth:

	Predicted +	Predicted −
Actual +	TP (true positive)	FN (false negative)
Actual −	FP (false positive)	TN (true negative)

From TP, FP, TN, FN we build every classification metric.

Accuracy, Precision, Recall, F1

Accuracy $= (TP + TN)/(TP + TN + FP + FN)$. Fraction of predictions that are correct.

Precision $= TP/(TP + FP)$. Of all *predicted positives*, what fraction are truly positive? "Don't cry wolf."

Recall (Sensitivity, True Positive Rate) $= TP/(TP + FN)$. Of all *actual positives*, what fraction did we catch? "Don't miss the wolf."

F1-Score $= 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$. The harmonic mean of precision and recall, balancing the two.

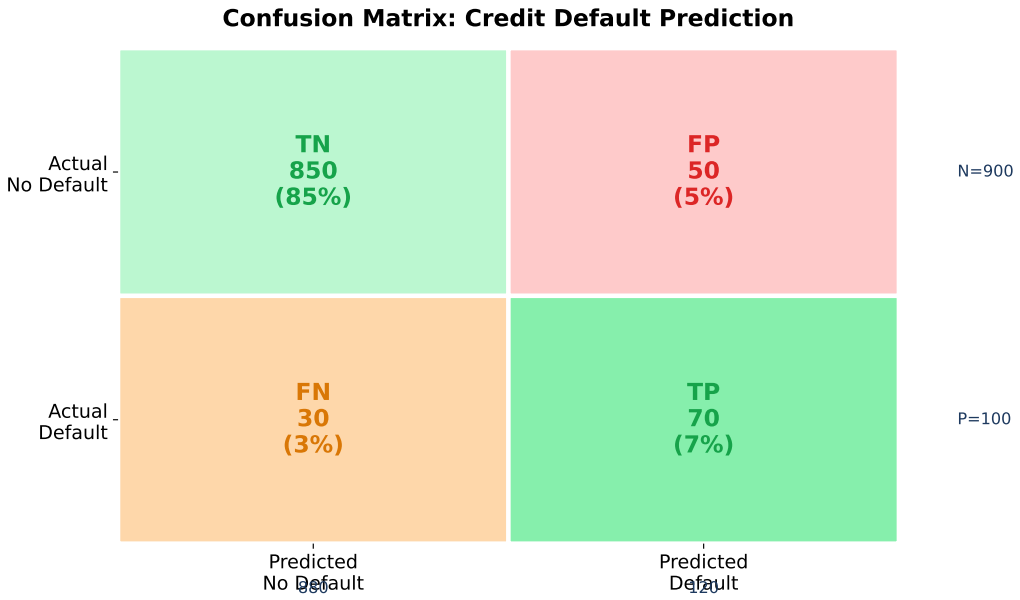


Figure 68: The confusion matrix. Four outcomes: true positive, false positive, true negative, false negative. Every classification metric is built from these four counts.

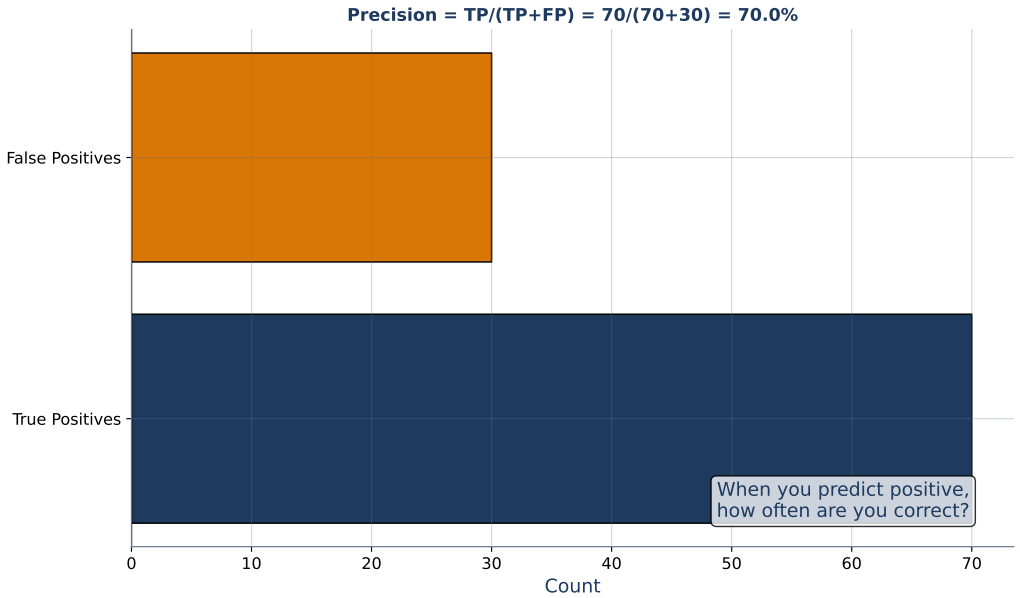


Figure 69: Precision: fraction of predicted positives that are truly positive. High precision = few false alarms.

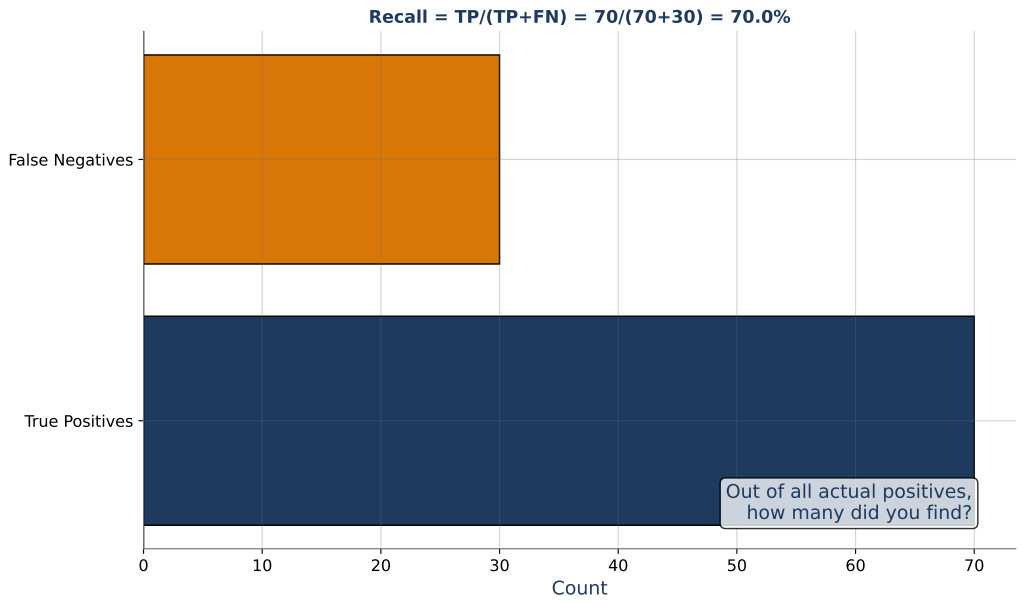


Figure 70: Recall: fraction of actual positives caught. High recall = few missed cases.

Key Formulas: Classification Metrics

Given a confusion matrix with counts TP, FP, TN, FN:

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN}$$

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

$$\text{Recall (TPR)} = \frac{TP}{TP + FN}$$

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

$$\text{Specificity (TNR)} = \frac{TN}{TN + FP}$$

$$\text{FPR} = \frac{FP}{TN + FP} = 1 - \text{Specificity}$$

Rule of thumb: Accuracy is okay only when classes are balanced. On imbalanced data, use precision/recall/F1 or ROC AUC and PR AUC.

A Worked Fraud-Detector Example

Worked Example: 1000 Transactions, 5 Frauds

A fraud classifier outputs predictions on 1000 test transactions. The confusion matrix:

	Predicted fraud	Predicted not fraud
Actual fraud	4 (TP)	1 (FN)
Actual not fraud	3 (FP)	992 (TN)

Accuracy $= (4 + 992)/1000 = 996/1000 = 99.6\%$.
Precision $= 4/(4 + 3) = 4/7 \approx 0.571$.
Recall $= 4/(4 + 1) = 4/5 = 0.800$.
F1 $= 2 \cdot (0.571 \cdot 0.800)/(0.571 + 0.800) = 0.914/1.371 \approx 0.667$.
The 99.6% accuracy is overly flattering; the real story is that the model catches 80% of frauds (recall) but 43% of its alerts are wrong (precision). Whether this is acceptable depends on the cost of a missed fraud versus a false alarm.
Compare to a "never fraud" model on the same data: 995 actual non-frauds and 5 actual frauds. Predicting always-not-fraud gives $995/1000 = 99.5\%$ accuracy—higher than our model's accuracy if we rounded! But precision and recall are both *zero*: no frauds detected, no true positives. Accuracy is a terrible summary for imbalanced problems.

Why Accuracy Misleads on Imbalanced Data

The intuition: accuracy weighs every sample equally. On a 0.5%-fraud dataset, the 99.5% majority class dominates, and a "always not-fraud" classifier scores 99.5%. On balanced data (50/50) accuracy is meaningful; on imbalanced data (99/1) it is almost meaningless.

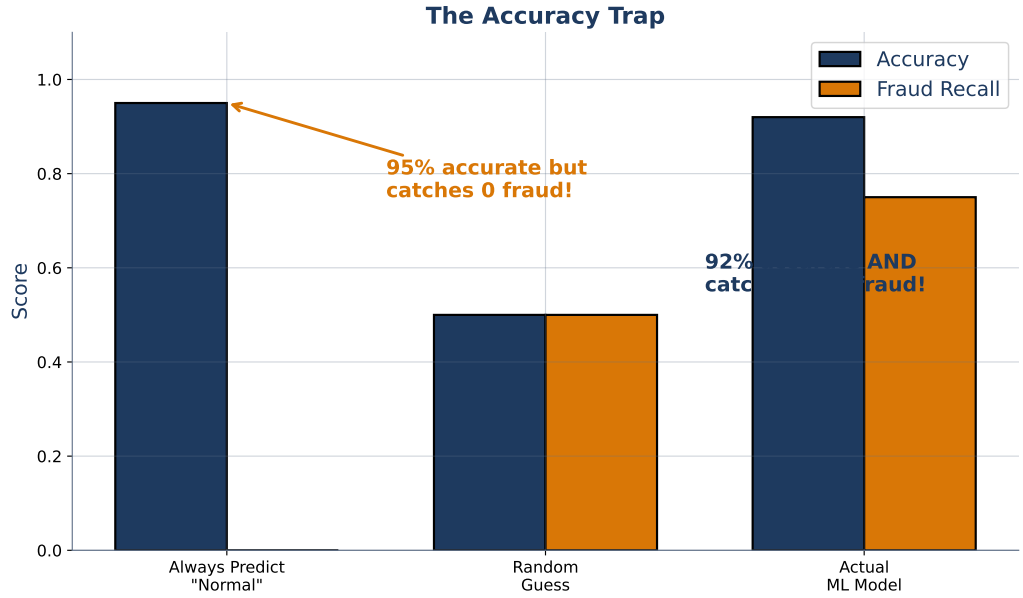


Figure 71: Accuracy on imbalanced data. The "majority-class" baseline achieves high accuracy without learning anything. Use F1, AUC, or PR AUC instead.

ROC Curves and AUC

ROC (Receiver Operating Characteristic) curves plot True Positive Rate (recall) against False Positive Rate as the classification threshold varies from 1 (strictest) to 0 (loosest). Every classifier that outputs a probability or score produces an ROC curve.

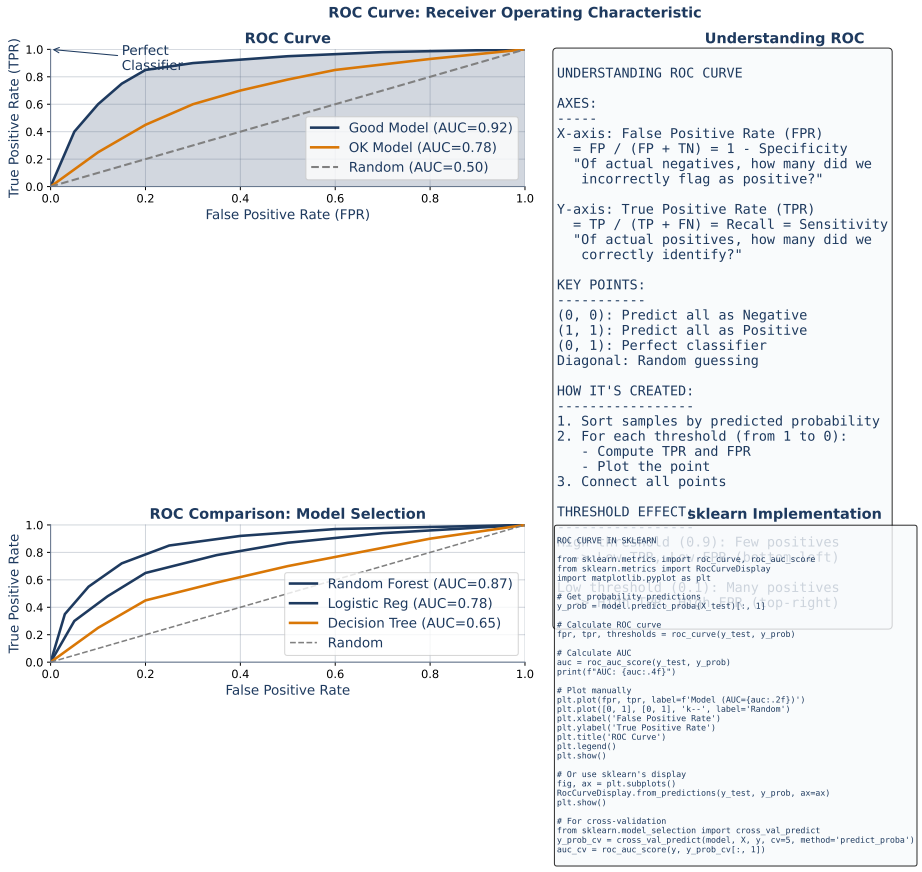


Figure 72: ROC curve: TPR vs FPR as threshold varies. The diagonal is random guessing; the top-left corner is a perfect classifier.

AUC (Area Under the ROC Curve) summarizes the curve as a single number in $[0, 1]$. AUC $= 0.5$ is random guessing; AUC $= 1.0$ is perfect. AUC has a beautiful probabilistic interpretation: *the probability that the classifier ranks a random positive example higher than a random negative example.* This ranking interpretation is why AUC is common in credit scoring and medical diagnostics—what matters is sorting high-risk above low-risk, not the exact threshold.

Precision-Recall Curves

ROC curves can be misleading on heavily imbalanced data because FPR has a large denominator (all the true negatives). A classifier can achieve a respectable-looking ROC curve while missing nearly all positives. PR curves address this: plot precision against recall as the threshold varies. For imbalanced data, PR curves are more informative.

Threshold Tuning

Most classifiers output a probability or score; a fixed threshold (usually 0.5) converts the score to a class prediction. But 0.5 is not always the right threshold. For fraud detection you might want threshold 0.1 (catch more fraud at the cost of false alarms); for spam filtering you might want 0.8 (avoid blocking real email).

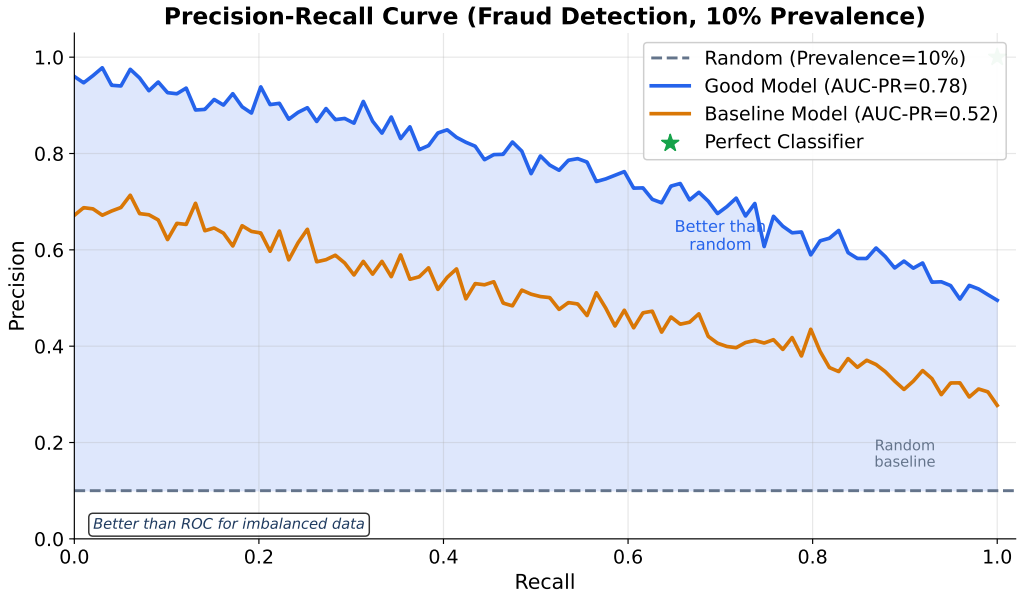


Figure 73: Precision-recall curve. Better for imbalanced data than ROC because FP counts only against precision, not swamped by a huge TN count.

The optimal threshold depends on the *costs* of FP vs FN. If a missed fraud costs \$10,000 and a false alarm costs \$50 (analyst time), you want low threshold: even a 1%-probability fraud is worth flagging.

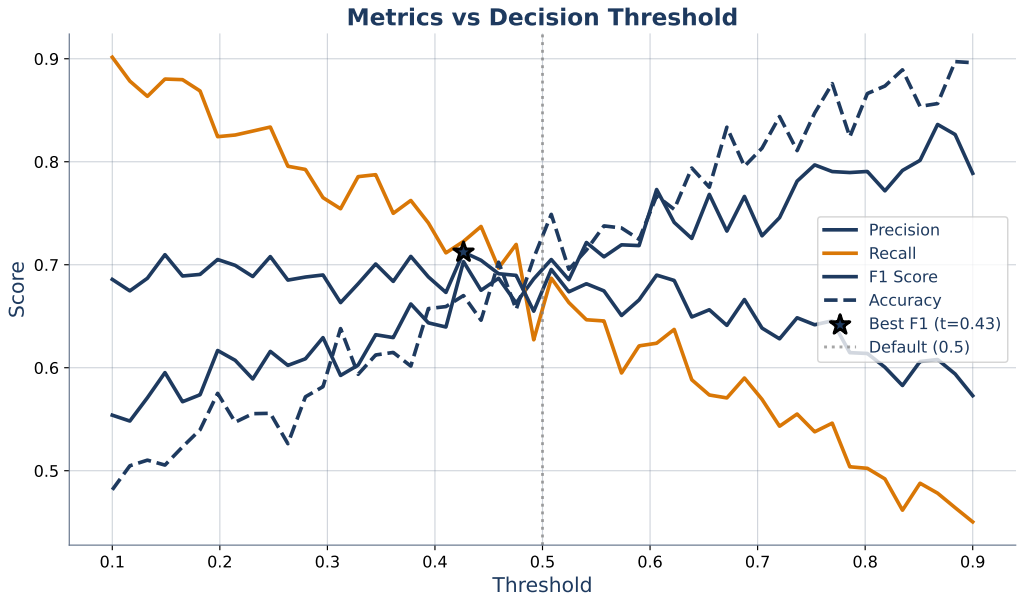


Figure 74: Metrics as functions of the decision threshold. Precision rises; recall falls; F1 peaks in between. The business problem determines which point to pick.

Class Imbalance Solutions

Three standard tools for imbalanced data:

Class weights. Pass `class_weight='balanced'` to sklearn estimators. The loss function is reweighted so that minority-class errors count more—effectively compensating for the imbalance during training.

Undersampling. Drop majority-class samples until the classes are balanced. Simple but wastes data.

Oversampling / SMOTE. Duplicate or interpolate minority-class samples. Plain oversampling replicates existing points; SMOTE creates *synthetic* new minority points by interpolating between existing ones.

Figure 75: Comparing class-imbalance strategies. Class weights, oversampling, undersampling, and SMOTE each trade off different metrics.

SMOTE: How It Works

SMOTE (Synthetic Minority Over-sampling Technique, Chawla et al. 2002) creates a synthetic minority example as follows:

1. Pick a real minority-class point x at random.
2. Find its k nearest minority-class neighbors (typically $k = 5$).
3. Pick a random neighbor x' among them.
4. Create a synthetic point at $\alpha x + (1 - \alpha)x'$ for random $\alpha \in [0, 1]$.

This "interpolates" minority-class points, effectively oversampling the minority without exact duplication.

Cost-Sensitive Decisions

For each decision you make, consider explicitly the cost matrix:

Expected cost per prediction $= P(y = 1|x) \cdot [\text{TP cost}] + P(y = 0|x) \cdot [\text{FP cost}]$ (if you predict 1).

Predict 1 when this expected cost is less than the expected cost of predicting 0. Rearrange: predict 1 iff

$$\frac{P(y = 1|x)}{P(y = 0|x)} > \frac{C_{FP} - C_{TN}}{C_{FN} - C_{TP}}.$$

The right-hand side is a threshold on the odds ratio derived entirely from the cost matrix.

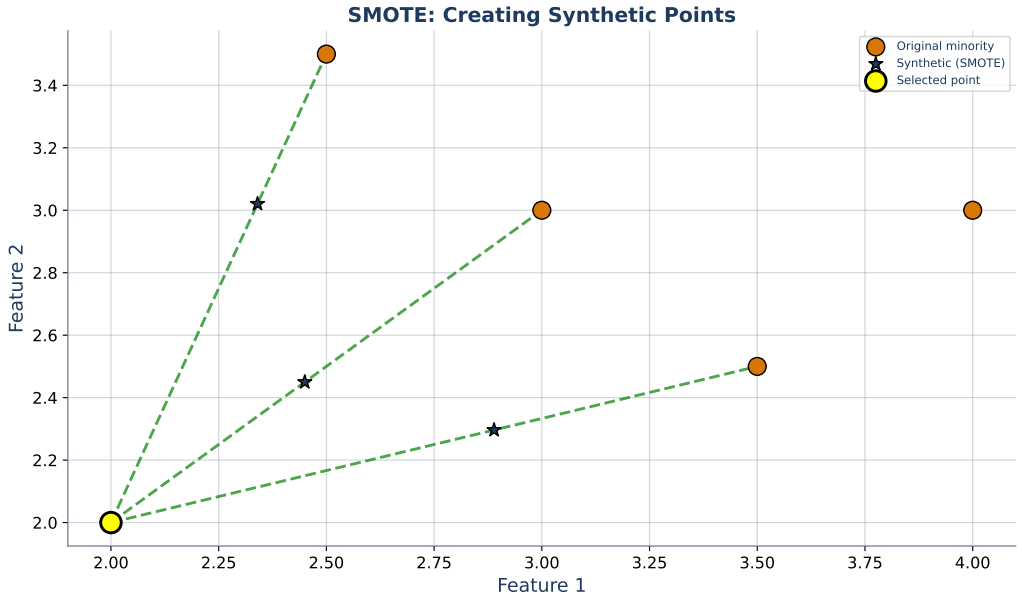


Figure 76: SMOTE creates synthetic minority points by interpolating between real minority points. Less overfitting than plain oversampling, more realistic than random noise.

Data Leakage: The Silent Killer

Data leakage happens when information from the test set (or the future) silently slips into training. The result is an overoptimistic validation score that does not generalize to production.

Classic leakage example: scale features *before* splitting:

```
X_scaled = StandardScaler().fit_transform(X)
X_train, X_test = train_test_split(X_scaled)  # WRONG
```

The scaler's mean and variance are computed from *all* rows, including the test set. The test set has already influenced the scaling. Result: test-set performance is optimistic.

Correct version: scale inside a pipeline

```
pipe = Pipeline([('scaler', StandardScaler()), ('model', ...)])
X_train, X_test = train_test_split(X)
pipe.fit(X_train, y_train)  # scaler fits on train only
```

Label Leakage

Label leakage is worse: a feature reveals the label directly. Example: building a churn model with the feature "days since last login." Customers who churned by the time of scoring have enormous "days since last login"—the feature is effectively a copy of the target. Label leakage in production causes the model to fail silently the moment the leak feature goes stale.

Detect label leakage by (1) examining feature-label correlations before building the model, (2) checking for features that are only available at prediction time in the training data but not in production, and (3) reviewing features with domain experts.

Concept Drift

Concept drift is the slow change of the data distribution over time. Customer behavior evolves; market regimes shift; the mapping from features to labels is not stationary. A model trained in 2022 may degrade on 2024 data simply because the world has changed.

Figure 77: Data leakage turns rigorous ML into illusory ML. The fitted scaler absorbs test-set information; the model appears better than it is.

Monitor production models: track prediction distribution drift (KS test on predicted scores), feature distribution drift (KS test per feature), and true performance (when you get labels back). Retrain on a regular cadence or when drift is detected.

Pipelines: Leakage Prevention by Construction

scikit-learn's `Pipeline` object makes leakage hard to commit. All preprocessing steps (scaling, imputation, PCA) and the final model are chained. The pipeline fits all preprocessing only on the training data and applies the same transformations at test time. Using Pipelines is the default defense against scaling-before-split and related errors.

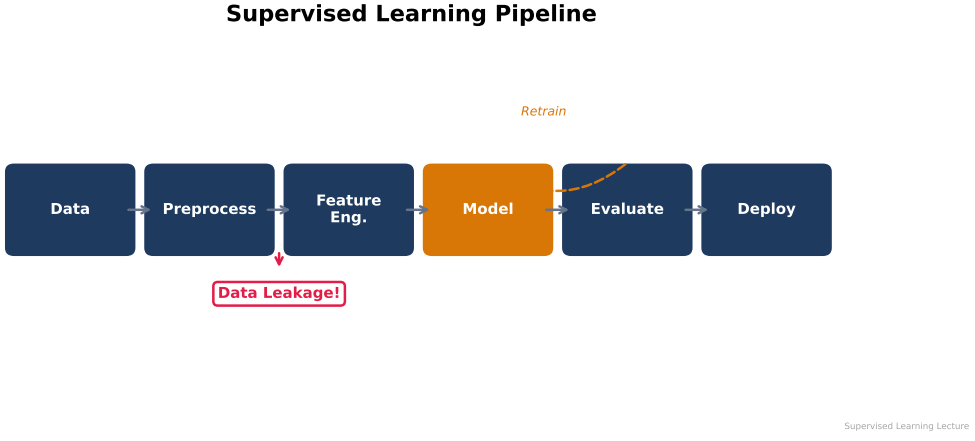


Figure 78: A complete ML pipeline: imputation → scaling → PCA → model. Pipelines prevent leakage by fitting every stage only on training data.

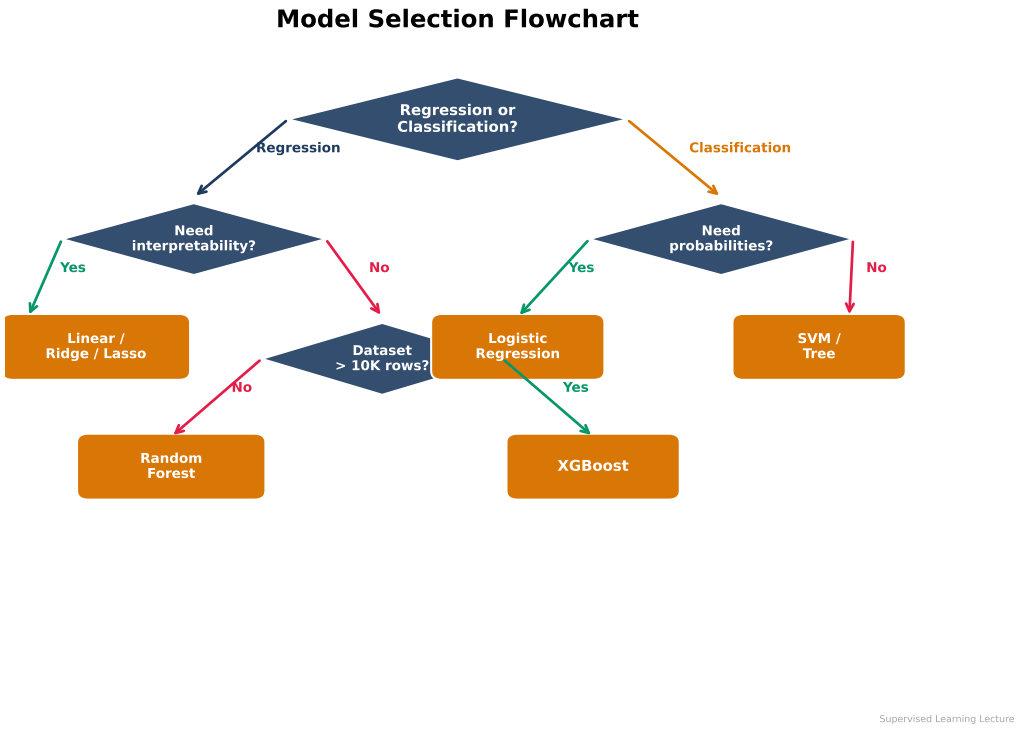


Figure 79: Which algorithm for which problem? A quick-reference flowchart for classical ML method selection.

Model Selection Flowchart

Definition: Classification Evaluation and Production Concerns

- **Confusion matrix**: 2×2 (for binary) table of TP/FP/TN/FN counts.
- **Precision** $= TP/(TP + FP)$; **Recall** $= TP/(TP + FN)$; **F1** $= 2PR/(P + R)$.
- **ROC curve**: TPR vs FPR as threshold varies. AUC is its area, with probabilistic ranking interpretation.
- **PR curve**: Precision vs Recall. Better than ROC for imbalanced data.
- **Class imbalance**: Techniques include class weights, SMOTE, undersampling, threshold tuning.
- **Data leakage**: Test-set or future information contaminates training, inflating validation scores.
- **Concept drift**: The data distribution changes over time, degrading production performance.
- **Pipelines**: Chain preprocessing with the model so that transforms fit on training data only.

Common Misconceptions about Classification Metrics

(1) "99% accuracy means a great model." On imbalanced data (1% minority), the constant "always majority" classifier achieves 99% without learning anything. Always check against a naive baseline.
(2) "AUC alone tells you if the model is good." AUC measures ranking quality, not calibrated probability or operating-point performance. A model with great AUC might still be useless at your chosen threshold; always pair AUC with a confusion matrix at the deployed threshold.
(3) "SMOTE always helps." SMOTE can hurt in high-dim data where interpolations land in regions with no real samples. Try class weights first; reach for SMOTE only if simpler tools fail.
(4) "Data leakage is a beginner mistake." Sophisticated leakage—leaky features, pre-split preprocessing, feature engineering that peeks at the test set—is one of the most common causes of models that "work in the notebook but fail in production."

Historical Background: Peterson, Chawla, and Applied ML

ROC curves originated in signal-detection theory during World War II. Peterson, Birdsall, and Fox developed them at the University of Michigan around 1954 to quantify radar operators' ability to distinguish signal from noise. The Receiver Operating Characteristic graphed the operator's trade-off between true positives (enemy aircraft detected) and false positives (false alarms) as they adjusted their sensitivity threshold. The name stuck.
ROC analysis moved into medicine in the 1960s (radiological diagnosis) and into machine learning in the late 1990s (Provost, Fawcett, Kubat). Fawcett's 2006 paper "An Introduction to ROC Analysis" remains the standard reference.
SMOTE was introduced by Nitesh Chawla, Kevin Bowyer, Lawrence Hall, and W. Philip Kegelmeyer in "SMOTE: Synthetic Minority Over-sampling Technique" published in 2002 in the *Journal of Artificial Intelligence Research*. It became the de-facto method for imbalanced classification, especially in medical and financial applications.
Concept drift formalization dates to Widmer and Kubat's 1996 paper "Learning in the Presence of Concept Drift and Hidden Contexts," but production-ML drift monitoring became mainstream only in the 2010s with the rise of deployed ML systems at scale. Today every major ML platform (Vertex AI, SageMaker, Azure ML) ships drift-monitoring tools by default.

Complete Classification Evaluation Pipeline

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score,
    TimeSeriesSplit
from sklearn.metrics import (
    confusion_matrix, classification_report,
    roc_auc_score, precision_recall_curve
)

# Pipeline prevents leakage
pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('pca',    PCA(n_components=0.95)),
    ('model',  LogisticRegression(class_weight='balanced',
                                  max_iter=1000, C=0.1)),
])

# Time-series CV for honest validation
tscv = TimeSeriesSplit(n_splits=5)
auc_scores = cross_val_score(pipe, X, y, cv=tscv, scoring='
    roc_auc')
print(f'AUC: {auc_scores.mean():.3f} +- {auc_scores.std():.3f}')

# Deploy: fit on all training data, evaluate on held-out
pipe.fit(X_train, y_train)
y_proba = pipe.predict_proba(X_test)[:, 1]
y_pred  = pipe.predict(X_test)

print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
print(f'AUC: {roc_auc_score(y_test, y_proba):.3f}')

# Threshold tuning by profit curve
import numpy as np
precisions, recalls, thresholds = precision_recall_curve(y_test,
    y_proba)
f1s = 2 * precisions * recalls / (precisions + recalls + 1e-9)
best_threshold = thresholds[np.argmax(f1s[:-1])]
print(f'Best F1 threshold: {best_threshold:.3f}')
```

Problem 8.1 (Easy) ⋆

Given TP = 80, FP = 20, FN = 10, TN = 890, compute accuracy, precision, recall, specificity, and F1.
Solution: see Appendix.

Problem 8.2 (Easy) ⋆

A fraud detection model achieves 99.5% accuracy. The fraud rate in the test set is 0.5%. What does this accuracy tell you about the model's real performance? What additional metrics should you examine?
Solution: see Appendix.

Problem 8.3 (Medium) ⋆⋆

A cost matrix: false negative (missed fraud) costs \$10,000; false positive (false alarm) costs \$100. Your model outputs probabilities. Compute the optimal threshold using the cost-sensitive decision rule. How does your threshold change if the cost of FN doubles?
Solution: see Appendix.

Problem 8.4 (Medium) ⋆⋆

Identify the data leakage in the following code:

```
scaler = StandardScaler().fit(X)
X_scaled = scaler.transform(X)
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y)
model = LogisticRegression().fit(X_train, y_train)
```

Rewrite the code to prevent leakage.
Solution: see Appendix.

Problem 8.5 (Hard) ⋆⋆⋆

Design a complete ML pipeline for fraud detection: 100,000 credit-card transactions per day, 0.3% fraud rate, features include transaction amount, merchant category, time of day, and 30-day rolling aggregates. Specify: (a) preprocessing, (b) choice of model, (c) cross-validation strategy, (d) handling of class imbalance, (e) threshold selection procedure, (f) at least three production concerns you would address before deployment.
Solution: see Appendix.

Closing the Loop

This section concludes the main body of the handout. Sections 1 through 4 built the regression side of supervised learning (linear models, regularization, metrics, cross-validation). Sections 5 through 7 built the classification side (logistic regression, trees and ensembles, SVMs/KNN/NB). Section 8 closed with the evaluation discipline that separates honest ML from fantasy.

The appendix contains complete solutions for all 40 practice problems. Solve each problem yourself first—the pedagogy is far more effective when the struggle precedes the answer. Skim the solutions only to verify your understanding or to unblock a step you genuinely cannot see.

Key Takeaway: On imbalanced data, accuracy lies: use precision/recall/F1/AUC and tune thresholds to your cost matrix; pipelines prevent leakage; walk-forward validation prevents temporal leakage; monitor for drift in production.

A. Solutions to Practice Problems

Section 1: From Correlation to Prediction

Problem 1.1 (Easy). (a) Regression—continuous target (return). (b) Classification—binary target (default/no-default). (c) Regression—continuous target (price). (d) Classification—binary target (spam/not). (e) Regression—continuous target (glucose level).

Problem 1.2 (Medium). Random 80/20 split is wrong because it ignores time order. A random draw could place some 2023 days in training and some 2015 days in testing, letting the model "see" the future to predict the past. The correct split is chronological: train on days 1–2000, test on days 2001–2500. The test set must follow the training set in time.

Problem 1.3 (Medium). Financial data violates i.i.d. in two ways: (1) autocorrelation—today's return is correlated with yesterday's, so observations are not independent; (2) non-stationarity—the joint distribution drifts over time (2008 is not 2022).

Concrete leakage examples: (a) fitting a scaler on all data before splitting means the test set's statistics influence the scaling; (b) using rolling features (e.g., 30-day momentum) lets x_t overlap with x_{t-1}, so a random fold split leaks information via these overlapping windows.

Fixes: (a) use `Pipeline` so the scaler fits only on training data; (b) use `TimeSeriesSplit` with a gap and embargo to prevent overlapping-window leakage.

Problem 1.4 (Hard). Let $\mu(x) = \mathbb{E}[\hat{f}(x)]$. Decompose:

$$y - \hat{f}(x) = f(x) + \epsilon - \hat{f}(x) = (f(x) - \mu(x)) + (\mu(x) - \hat{f}(x)) + \epsilon.$$

Let $A = f(x) - \mu(x)$ (a constant, not random), $B = \mu(x) - \hat{f}(x)$ (random, zero mean), $C = \epsilon$ (random, zero mean, independent of training). Square and take expectations:

$$\mathbb{E}[(y - \hat{f})^2] = A^2 + \mathbb{E}[B^2] + \mathbb{E}[C^2] + 2A\mathbb{E}[B] + 2A\mathbb{E}[C] + 2\mathbb{E}[BC].$$

Cross terms vanish: $\mathbb{E}[B] = 0$, $\mathbb{E}[C] = 0$, and $\mathbb{E}[BC] = 0$ because B depends only on training and C is test noise, independent of training. Remaining: $A^2 + \mathrm{Var}(\hat{f}) + \sigma^2$. □

Problem 1.5 (Hard). Strategy:

1. **Hold-out test set**: set aside 20% of rows as a pure evaluation set that is never used until the end.
2. **Nested CV on the training 80%**: outer loop (5-fold) estimates performance; inner loop (5-fold grid search) tunes hyperparameters. Use `GridSearchCV(pipe, param_grid, cv=5)` passed to `cross_val_score(_, X, y, cv=5)`.
3. **Prevent leakage**: wrap preprocessing (scaler, imputer, feature selection) in a `Pipeline` so they are refit on each inner-fold training set.
4. **Final evaluation**: fit the tuned pipeline on the full 80% training set, evaluate once on the held-out 20% test set. Report the test-set metric as the production estimate.

If the data are time-ordered, replace outer random CV with `TimeSeriesSplit`; otherwise use `StratifiedKFold` for classification.

Section 2: OLS and Factor Models

Problem 2.1 (Easy). Means: $\bar{x} = 2$, $\bar{y} = (2 + 3 + 5)/3 = 10/3 \approx 3.333$. Slope formula and computation:

$$\begin{aligned}\hat{\beta}_1 &= \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sum_i (x_i - \bar{x})^2} \\ &= \frac{(-1)(-4/3) + 0 + (1)(5/3)}{1 + 0 + 1} = \frac{3}{2} = 1.5.\end{aligned}$$

Intercept: $\hat{\beta}_0 = 10/3 - 1.5 \cdot 2 = 10/3 - 3 = 1/3 \approx 0.333$.

Predictions: $\hat{y}_1 = 1/3 + 1.5 \cdot 1 = 11/6 \approx 1.833$; $\hat{y}_2 = 1/3 + 3 = 10/3 \approx 3.333$; $\hat{y}_3 = 1/3 + 4.5 = 29/6 \approx 4.833$.

Residuals: $e_1 = 2 - 11/6 = 1/6$; $e_2 = 3 - 10/3 = -1/3$; $e_3 = 5 - 29/6 = 1/6$. Sum: $1/6 - 1/3 + 1/6 = 2/6 - 2/6 = 0$. ✓

Problem 2.2 (Easy). CAPM: $R_i - R_f = \beta_i(R_m - R_f)$. If $R_m - R_f = 0.02$ (market excess), stock A (beta 1.5) has expected excess return $1.5 \cdot 0.02 = 0.03$ (3%); stock B (beta 0.8) has $0.8 \cdot 0.02 = 0.016$ (1.6%).

If $R_m - R_f = -0.03$, stock A: $1.5 \cdot (-0.03) = -0.045$ (drops 4.5%); stock B: $0.8 \cdot (-0.03) = -0.024$ (drops 2.4%). High-beta stocks amplify both gains and losses.

Problem 2.3 (Medium). *Assumptions needed:* linearity (L), independence of errors (I), equal variance (E). Normality (N) is *not* needed for BLUE.

Proof: Any linear estimator is $\tilde{\beta} = Ay$. Unbiasedness requires $\mathbb{E}[\tilde{\beta}] = AX\beta = \beta$ for all β, so $AX = I$. Write $A = (X^\top X)^{-1}X^\top + B$; the condition becomes $BX = 0$. Variance: $\mathrm{Var}(\tilde{\beta}) = \sigma^2 AA^\top = \sigma^2[(X^\top X)^{-1} + BB^\top]$ (cross terms vanish because $BX = 0$). The OLS choice $B = 0$ minimizes this; any other B adds the positive-semidefinite matrix $\sigma^2 BB^\top$.

Why normality is needed for CIs: The distribution of $\hat{\beta}$ is normal only if errors are normal; in that case $\hat{\beta} \sim \mathcal{N}(\beta, \sigma^2(X^\top X)^{-1})$, giving exact t-distributed confidence intervals. Without normality, the central limit theorem provides only asymptotic (large-n) approximations.

Problem 2.4 (Medium). A U-shape in residuals-vs-fitted indicates **nonlinearity**: the true relationship bends, and a straight line cannot fit it. Fix: add polynomial features (e.g., x^2, x^3) to the feature matrix. This keeps the model linear in parameters but can fit nonlinear shapes. Alternatively, apply a transformation to the label (e.g., $\log y$) if the nonlinearity reflects a multiplicative structure.

Problem 2.5 (Hard). Build feature matrix $X \in \mathbb{R}^{T \times 4}$ with columns (intercept, MKT-RF, SMB, HML). Label vector $y \in \mathbb{R}^T$ of stock excess returns. Compute $\hat{\beta} = (X^\top X)^{-1}X^\top y$. This gives $(\hat{\alpha}, \hat{\beta}_M, \hat{\beta}_S, \hat{\beta}_H)^\top$.

If SMB and HML columns are identically zero, $X^\top X$ has two zero rows/columns; its submatrix for (intercept, MKT-RF) is invertible, and the solution reduces to the CAPM fit (with $\hat{\beta}_S, \hat{\beta}_H$ undefined but effectively zero).

When SMB, HML are highly correlated with MKT-RF, $X^\top X$ is near-singular (condition number explodes). Coefficients become unstable; small data changes produce large coefficient swings. The fix is regularization (Ridge or Lasso), or dropping redundant factors, or orthogonalizing the factors before regression.

Section 3: Regularization and Bias-Variance

Problem 3.1 (Easy). $\lambda_2 = 100 \gg \lambda_1 = 0.1$, so $\hat{\beta}^{(2)}$ is heavily regularized and smaller in L2 norm. $\hat{\beta}^{(2)}$ is more likely to **underfit**: a tiny coefficient vector means the model relies less on the features, pushing toward a constant predictor.

Problem 3.2 (Medium). OLS estimator of c with x-values fixed: $\hat{c} = \sum_i x_i y_i / \sum_i x_i^2$. Since $y_i = 2x_i + \epsilon_i$,

$$\hat{c} = \frac{\sum_i x_i(2x_i + \epsilon_i)}{\sum_i x_i^2} = 2 + \frac{\sum_i x_i \epsilon_i}{\sum_i x_i^2}.$$

$\mathbb{E}[\hat{c}] = 2$, so bias is zero. Variance:

$$\mathrm{Var}(\hat{c}) = \frac{\sum_i x_i^2 \cdot \mathrm{Var}(\epsilon_i)}{(\sum_i x_i^2)^2} = \frac{\sigma^2}{\sum_i x_i^2}.$$

For $x_i \in \{1, 2, \ldots, 10\}$: $\sum x_i^2 = 1+4+9+\ldots+100 = 385$. With $\sigma^2 = 1$: $\mathrm{Var}(\hat{c}) = 1/385 \approx 0.0026$.

At test point $x = 5$: $\text{Var}(\hat{f}(5)) = 25 \cdot \text{Var}(\hat{c}) \approx 0.065$. Bias is zero, so expected squared error is $0 + 0.065 + 1 = 1.065$.

Problem 3.3 (Medium). Sketch: In 2D, the L1 unit ball is a diamond with corners at $(\pm 1, 0)$ and $(0, \pm 1)$. The L2 unit ball is a unit disk. An OLS ellipse centered at $(2, 1)$ grows outward as the loss increases; the constrained optimum is the first touch point with the constraint boundary.

For the diamond: generic ellipses touch the diamond at a corner, forcing one coordinate (e.g., β_2) to be zero. The probability of touching an edge (instead of a corner) is measure-zero. Hence Lasso produces sparse solutions.

For the disk: generic ellipses touch at a smooth point on the circle. Both coordinates are nonzero; Ridge shrinks coefficients but keeps them nonzero.

Problem 3.4 (Hard). *Derivation of Ridge.* Loss: $L(\beta) = (y - X\beta)^\top (y - X\beta) + \lambda \beta^\top \beta$. Expand:

$$L = y^\top y - 2y^\top X\beta + \beta^\top X^\top X\beta + \lambda \beta^\top \beta.$$

Gradient: $\nabla L = -2X^\top y + 2X^\top X\beta + 2\lambda\beta$. Set to zero: $(X^\top X + \lambda I)\beta = X^\top y \implies \hat{\beta} = (X^\top X + \lambda I)^{-1} X^\top y$.

Hessian: $\nabla^2 L = 2(X^\top X + \lambda I)$. $X^\top X$ is positive semidefinite; adding λI with $\lambda > 0$ makes the sum positive definite, so the Hessian is PD and the critical point is the unique global minimum.

Data augmentation equivalence. Construct augmented data: stack X with $\sqrt{\lambda} I_p$ (an identity matrix scaled by $\sqrt{\lambda}$), and y with zeros. New feature matrix $\tilde{X} = [X; \sqrt{\lambda} I_p]$ of shape $(n + p) \times p$. OLS on the augmented data:

$$\tilde{X}^\top \tilde{X} = X^\top X + \lambda I, \quad \tilde{X}^\top \tilde{y} = X^\top y + 0 = X^\top y.$$

Hence OLS on augmented data = Ridge on original. □

Problem 3.5 (Hard). Standard K-fold CV is wrong because it mixes past and future. Use `TimeSeriesSplit`:

```
from sklearn.model_selection import TimeSeriesSplit
tscv = TimeSeriesSplit(n_splits=5, gap=10)  # 10-day embargo

best_alpha = None
best_rmse = float('inf')
for alpha in alpha_grid:
    rmse_per_fold = []
    for train_idx, val_idx in tscv.split(X):
        X_tr, X_va = X[train_idx], X[val_idx]
        y_tr, y_va = y[train_idx], y[val_idx]
        pipe = Pipeline([('scaler', StandardScaler()),
                         ('model', Lasso(alpha=alpha))])
        pipe.fit(X_tr, y_tr)
        rmse = mean_squared_error(y_va, pipe.predict(X_va), squared=False)
        rmse_per_fold.append(rmse)
    mean_rmse = np.mean(rmse_per_fold)
    if mean_rmse < best_rmse:
        best_rmse = mean_rmse
        best_alpha = alpha
```

Gap/embargo of $\sim$10 days prevents overlapping-window leakage. The training window grows; the validation window is always in the future. The chosen α is the one with lowest out-of-sample RMSE.

Section 4: Regression Metrics and Validation

Problem 4.1 (Easy). MSE: $(0.04 + 0.09 + 0.01 + 0.01 + 0.25)/5 = 0.40/5 = 0.08$. RMSE: $\sqrt{0.08} \approx 0.283$. MAE: $(0.2 + 0.3 + 0.1 + 0.1 + 0.5)/5 = 1.2/5 = 0.24$.

MSE and RMSE are most affected by the 0.5 residual (quadratic penalty: contributes 0.25 of total 0.40, i.e., 62.5%). MAE only reflects it linearly (0.5 of total 1.2, i.e., 42%).

Problem 4.2 (Easy). $R^2 = 1 - \text{MSE}/\text{Var}(y) = 1 - 0.20/1.0 = 0.80$. In plain English: the model explains 80% of the variance in y. The remaining 20% of variance is not captured by the features.

Problem 4.3 (Medium). 5-fold walk-forward on 2500 trading days: split into 5 consecutive blocks of ≈500 days each. Fold 1: train on days 1–500, validate on 501–1000. Fold 2: train on 1–1000, validate on 1001–1500. And so on (expanding window). Embargo: 10–20 days between training end and validation start to prevent rolling-feature leakage (if features include 30-day rolling averages).

Justification: the training window always precedes the validation window in time, never training on the future to predict the past. The expanding window means later folds see more data, reflecting the growing history available in production.

Problem 4.4 (Medium). $R^2 = 0.05$ is *not* useless for stock returns. Monthly return R^2 is typically 0.01–0.10; 0.05 is at the high end of "good factor" territory. A model with $R^2 = 0.05$ implies a Pearson correlation of $\sqrt{0.05} \approx 0.22$—meaningful enough to support a portfolio strategy if used properly.

Conditions for profitability: (1) apply across many stocks (law of large numbers smooths out idiosyncratic noise); (2) combine with risk management (position sizing, drawdown controls); (3) account for trading costs (a 0.05 R^2 signal might be swamped by round-trip costs of 5–10 bps per trade).

Problem 4.5 (Hard). *Pseudocode:*

```
def k_fold_cv(model_factory, X, y, k=5):
    n = len(X)
    indices = np.random.permutation(n)
    fold_size = n // k
    scores = []
    for i in range(k):
        val_idx = indices[i*fold_size : (i+1)*fold_size]
        train_idx = np.setdiff1d(indices, val_idx)
        model = model_factory()
        model.fit(X[train_idx], y[train_idx])
        score = metric(y[val_idx], model.predict(X[val_idx]))
        scores.append(score)
    return np.mean(scores), np.std(scores)
```

Bias sources: (1) each fold trains on $n(1 - 1/K)$ observations, slightly less than n; the CV estimator is biased toward smaller training-set performance. (2) Model selection based on CV can be optimistic if CV is used both to tune and report.

Variance sources: (1) the random fold assignment; different shuffles give different estimates. (2) Model instability: high-variance models produce high CV variance.

Choice of K: $K = 5$ is a good default because training sets are 80% of n (close enough to full), CV variance is moderate, compute cost is 5 fits. $K = 10$ reduces bias (training on 90% of n) at the cost of compute; useful when computing one fit is cheap. Leave-one-out ($K = n$) has minimal bias but maximal variance and n-fold compute—useful for very small datasets where the bias of $K = 5$ matters more than variance.

Section 5: Logistic Regression

Problem 5.1 (Easy). $P(y=1|x) = \sigma(1.2) = 1/(1+e^{-1.2}) = 1/(1+0.3012) \approx 0.7685$.

Odds: $P/(1-P) = 0.7685/0.2315 \approx 3.32$. Log-odds: $\log(3.32) \approx 1.2$. ✓ (the log-odds equal $\beta^\top x$ by construction).

Problem 5.2 (Easy). Odds ratio $e^{\beta_j} = 2.5$: flipping the feature from 0 to 1 multiplies the odds by 2.5.

Baseline: odds $= 0.1/0.9 \approx 0.111$. After flip: new odds $= 0.111 \cdot 2.5 = 0.278$. New probability: $0.278/(1+0.278) \approx 0.217$.

So the probability rises from 0.1 to ≈ 0.217. The probability more than doubles, but does not simply multiply by 2.5—that is the odds ratio, not the probability ratio.

Problem 5.3 (Medium). Decision boundary: $\{x : P(y=1|x) = 0.5\}$. $\sigma(z) = 0.5 \iff z = 0$. So the boundary is $\{x : \beta_0 + \beta^\top x = 0\}$—a hyperplane in $\mathbb{R}^p$. In 2D, a straight line. In 3D, a plane. Always linear in x.

Problem 5.4 (Medium). Cross-entropy loss: $L(\beta) = -\sum_i [y_i \log p_i + (1-y_i)\log(1-p_i)]$ where $p_i = \sigma(\beta^\top x_i)$.

Using $\partial p_i/\partial \beta_j = p_i(1-p_i)x_{ij}$:

$$\frac{\partial L}{\partial \beta_j} = -\sum_i \left[\frac{y_i}{p_i} - \frac{1-y_i}{1-p_i}\right] p_i(1-p_i)x_{ij}.$$

Simplify the bracket: $\frac{y_i(1-p_i)-(1-y_i)p_i}{p_i(1-p_i)} = \frac{y_i-p_i}{p_i(1-p_i)}$. The denominator cancels with $p_i(1-p_i)$ outside, leaving:

$$\frac{\partial L}{\partial \beta_j} = -\sum_i (y_i - p_i)x_{ij} = \sum_i (p_i - y_i)x_{ij}.$$

Setting the gradient to zero: $\sum_i (p_i - y_i)x_{ij} = 0$ for all j. No closed form because $p_i = \sigma(\beta^\top x_i)$ is a nonlinear function of β; the system is transcendental and must be solved iteratively.

Problem 5.5 (Hard). 3-class softmax: $p_k = e^{\beta_k^\top x} / \sum_j e^{\beta_j^\top x}$. One-hot label y with $y_k \in \{0,1\}$ and $\sum_k y_k = 1$. Cross-entropy: $L = -\sum_i \sum_k y_{ik} \log p_{ik}$.

Gradient with respect to β_k: using $\partial p_{il}/\partial \beta_{kj} = p_{il}(\delta_{lk} - p_{ik})x_{ij}$,

$$\frac{\partial L}{\partial \beta_{kj}} = -\sum_i \sum_l \frac{y_{il}}{p_{il}} p_{il}(\delta_{lk} - p_{ik})x_{ij} = -\sum_i \sum_l y_{il}(\delta_{lk} - p_{ik})x_{ij}.$$

Since $\sum_l y_{il} = 1$ and $\sum_l y_{il}\delta_{lk} = y_{ik}$:

$$\frac{\partial L}{\partial \beta_{kj}} = -\sum_i (y_{ik} - p_{ik})x_{ij} = \sum_i (p_{ik} - y_{ik})x_{ij}.$$

Same form as binary! Sum over mis-prediction residuals weighted by features.

When $K = 2$: softmax reduces to sigmoid (fix $\beta_0 = 0$ for identifiability; $p_1 = e^{\beta_1^\top x}/(1+e^{\beta_1^\top x}) = \sigma(\beta_1^\top x)$). The gradient matches the binary formula.

Section 6: Trees, Forests, Boosting

Problem 6.1 (Easy). Gini: $1 - (0.7^2 + 0.3^2) = 1 - (0.49 + 0.09) = 0.42$.

Entropy (base 2): $-0.7\log_2(0.7) - 0.3\log_2(0.3) = -0.7 \cdot (-0.515) - 0.3 \cdot (-1.737) = 0.361 + 0.521 = 0.882$.

Not pure (pure = 0 for both). The node is moderately impure; splitting further could reduce impurity.

Problem 6.2 (Medium). Parent Gini: $1-((8/12)^2+(4/12)^2) = 1-(0.444+0.111) = 0.445$.

Split (a): left $(6+, 2-)$, right $(2+, 2-)$. Left Gini: $1-(0.75^2+0.25^2) = 0.375$. Right Gini: $1-(0.5^2+0.5^2) = 0.5$. Weighted: $(8/12)(0.375)+(4/12)(0.5) = 0.25+0.167 = 0.417$. Reduction: $0.445 - 0.417 = 0.028$.

Split (b): left $(8+, 0-)$, right $(0+, 4-)$. Both pure, Gini $= 0$. Weighted: 0. Reduction: 0.445. **Best split.**

Split (c): left $(4+, 3-)$, right $(4+, 1-)$. Left Gini: $1-((4/7)^2+(3/7)^2) = 1-(0.327+0.184) = 0.489$. Right Gini: $1-(0.8^2+0.2^2) = 0.32$. Weighted: $(7/12)(0.489)+(5/12)(0.32) = 0.285+0.133 = 0.418$. Reduction: 0.027.

Split (b) is the best: it produces two pure children (Gini 0), reduction $= 0.445$.

Problem 6.3 (Medium). Bootstrap sample: n draws with replacement from the training set of size n. Probability any specific observation is excluded from a single draw: $1-1/n$. Probability excluded from all n draws: $(1-1/n)^n \to e^{-1} \approx 0.368$ as $n \to \infty$.

So about 37% of observations are out-of-bag for each tree. For any given observation, about 37% of the trees in the ensemble never saw it; aggregating predictions from these OOB trees gives an unbiased estimate of generalization performance without a separate validation set. OOB predictions use only trees trained on data that excluded the query point—exactly like a test set for that point.

Problem 6.4 (Hard). Let $\hat{f}_b$ be the prediction of tree b, with $\mathrm{Var}(\hat{f}_b) = \sigma^2$ and pairwise correlation $\mathrm{Corr}(\hat{f}_b, \hat{f}_c) = \rho$ for $b \neq c$. The bagged average:

$$\mathrm{Var}(\bar{f}) = \frac{1}{B^2}\left[\sum_b \mathrm{Var}(\hat{f}_b) + \sum_{b\neq c} \mathrm{Cov}(\hat{f}_b, \hat{f}_c)\right] = \frac{B\sigma^2 + B(B-1)\rho\sigma^2}{B^2} = \frac{\sigma^2}{B} + \frac{(B-1)\rho\sigma^2}{B}.$$

As $B \to \infty$: $\mathrm{Var}(\bar{f}) \to \rho\sigma^2$. The variance floor is $\rho\sigma^2$, not zero.

Random feature subsets (random forests beyond plain bagging) decorrelate trees by preventing them from all splitting on the strongest feature first. Lower $\rho \implies$ lower variance floor $\implies$ better generalization.

Problem 6.5 (Hard). At iteration m, current prediction function is F_{m-1}. Loss per sample: $L(y_i, F_{m-1}(x_i))$. Gradient boosting chooses the next tree h_m to approximate the *negative gradient* of the loss with respect to F:

$$r_i^{(m)} = -\left.\frac{\partial L(y_i, F)}{\partial F}\right|_{F=F_{m-1}(x_i)}.$$

Fit h_m to predict these $r_i^{(m)}$ values using the training inputs x_i, and update $F_m = F_{m-1} + \eta h_m$ for some learning rate η.

For squared loss $L(y, F) = \frac{1}{2}(y-F)^2$: $\partial L/\partial F = -(y-F)$. So $r_i^{(m)} = y_i - F_{m-1}(x_i)$, the residual. Each boosting tree fits the residuals of the ensemble so far—a gradient step in function space.

Section 7: SVM, KNN, Naive Bayes

Problem 7.1 (Easy). KNN has no training because no parameters are learned. All work happens at prediction time: given a new input, find its K nearest training points and return their majority class (or average label). The cost: prediction requires computing distances to every training point, $O(nd)$ per query (or $O(\log n)$ with KD-trees in low dimensions). Training-free but prediction-expensive—opposite of most ML methods.

Problem 7.2 (Medium). For a simple illustration, consider the three points with labels $y_1 = y_2 = +1, y_3 = -1$ and assume the dual solution places weights $\alpha_1 = \alpha_2 = \alpha_3 = 1$. Then $w = \sum \alpha_i y_i x_i = (1)(+1)(1,1) + (1)(+1)(2,3) + (1)(-1)(0,0) = (3,4)$.

Bias: for any support vector on the margin, $y_i(w^\top x_i + b) = 1$. Using $i = 1$ ($y_1 = +1, x_1 = (1,1)$): $1 \cdot (3 \cdot 1 + 4 \cdot 1 + b) = 1 \implies b = 1 - 7 = -6$.

Hyperplane equation: $3x_1 + 4x_2 - 6 = 0$ or $3x_1 + 4x_2 = 6$. (This is a simplified illustrative example; a real SVM would compute dual weights via quadratic programming.)

Problem 7.3 (Medium). Setup: 6 documents, vocabulary of 3 words $\{w_1, w_2, w_3\}$. Training label counts (suppose): positive class has word counts $(5, 2, 1)$ total across its 3 documents (8 words total); negative class has $(1, 4, 3)$ total (8 words total).

Priors: $P(+) = P(-) = 0.5$ (balanced).

Class-conditional probabilities (Laplace-smoothed with $\alpha = 1$, vocab size $V = 3$):

$$P(w_1|+) = (5+1)/(8+3) = 6/11 \approx 0.545, \quad P(w_2|+) = 3/11, \quad P(w_3|+) = 2/11.$$

$$P(w_1|-) = 2/11, \quad P(w_2|-) = 5/11, \quad P(w_3|-) = 4/11.$$

New document contains w_1 once and w_2 once:

$$P(+|\text{doc}) \propto 0.5 \cdot 0.545 \cdot 0.273 = 0.0744.$$

$$P(-|\text{doc}) \propto 0.5 \cdot 0.182 \cdot 0.455 = 0.0414.$$

Normalize: $P(+|\text{doc}) = 0.0744/(0.0744 + 0.0414) \approx 0.642$. Predict **positive**.

Problem 7.4 (Hard). Polynomial kernel degree 2: $K(x, x') = (x^\top x')^2 = (x_1 x_1' + x_2 x_2')^2 = x_1^2 x_1'^2 + 2x_1 x_2 x_1' x_2' + x_2^2 x_2'^2$.

This equals $\phi(x)^\top \phi(x')$ with $\phi(x) = (x_1^2, \sqrt{2}x_1 x_2, x_2^2)$, a 3-dimensional feature map.

Alternatively, with the inhomogeneous version $K(x, x') = (x^\top x' + 1)^2$, the feature map is 6-dimensional: $(1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, \sqrt{2}x_1 x_2, x_2^2)$.

For degree 3 in 2D: $(x_1 x_1' + x_2 x_2')^3$ expands into terms like $x_1^3 x_1'^3, x_1^2 x_2 \cdot x_1'^2 x_2', \ldots$. The feature map has dimension $\binom{2+3-1}{3} = 4$ for the homogeneous version; the inhomogeneous version with constant term $+1$ includes more. General formula: polynomial kernel of degree d over p-dim inputs has feature-map dimension $\binom{p+d}{d}$.

The deep lesson: degree-d polynomial features blow up combinatorially, but the kernel computes the inner product in $O(p)$ without ever materializing them.

Problem 7.5 (Hard). For uniform random points in a d-dim unit cube, the Euclidean distance between two points has expected value $\mathbb{E}[\|x - x'\|] \approx \sqrt{d/6}$ (each coordinate contributes $\mathbb{E}[(x_i - x_i')^2] = 1/6$ for uniform coordinates). So pairwise distances *grow* with $\sqrt{d}$.

More important is the *relative* spread. The ratio between farthest and nearest neighbor distances, $R = d_{\max}/d_{\min}$, tends to 1 as $d \to \infty$ for most distributions (Beyer et al., 1999). Distances concentrate: all points look equidistant.

Consequence for KNN: "nearest" loses meaning in high dimensions. The nearest neighbor is scarcely closer than the farthest, so taking the K nearest neighbors returns a random-looking subset that does not reflect local structure. KNN degenerates to random guessing as d grows unless combined with dimensionality reduction.

Section 8: Classification Metrics and Production

Problem 8.1 (Easy). Total: $80 + 20 + 10 + 890 = 1000$.

- Accuracy: $(80 + 890)/1000 = 0.97$.
- Precision: $80/(80 + 20) = 0.80$.
- Recall: $80/(80 + 10) = 0.889$.
- Specificity: $890/(890 + 20) = 0.978$.

- F1: $2 \cdot 0.80 \cdot 0.889/(0.80 + 0.889) = 1.422/1.689 \approx 0.842$.

Problem 8.2 (Easy). 99.5% accuracy matches exactly the base rate of the majority class (0.5% minority means predicting "not fraud" always gives 99.5% accuracy). The model may be trivial.

Additional metrics: precision, recall, F1 *on the fraud class*; ROC AUC; PR AUC; confusion matrix. A real model should have non-trivial recall (catches most fraud) and reasonable precision (few false alarms). Accuracy alone on imbalanced data is a red flag.

Problem 8.3 (Medium). Cost-sensitive rule: predict positive iff

$$\frac{P(y=1|x)}{P(y=0|x)} > \frac{C_{FP} - C_{TN}}{C_{FN} - C_{TP}}.$$

Assuming zero cost for correct decisions ($C_{TN} = C_{TP} = 0$), $C_{FP} = 100$, $C_{FN} = 10{,}000$:

$$\frac{P(y=1)}{P(y=0)} > \frac{100}{10{,}000} = 0.01.$$

Threshold: $P(y=1|x) > 0.01/(1 + 0.01) \approx 0.0099$. Very low threshold: flag almost anything.

If C_{FN} doubles to \$20,000: threshold becomes $100/20{,}000 = 0.005$, or $P(y=1|x) > 0.005/1.005 \approx 0.00498$. Even lower threshold—flag more aggressively as the cost of missing increases.

Problem 8.4 (Medium). Leakage: the scaler is fit on *all* of X (including test data) before splitting. The scaling uses the test set's mean and standard deviation—test information has already influenced training-data transformation.

Fix with Pipeline:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y)
pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('model',  LogisticRegression()),
])
pipe.fit(X_train, y_train)  # scaler fits on train only
y_pred = pipe.predict(X_test)
```

Problem 8.5 (Hard). *Pipeline:*

1. **Preprocessing:** (a) robust outlier handling (winsorize transaction amount). (b) One-HotEncode merchant category. (c) cyclical encoding for time-of-day. (d) rolling features already computed; verify no future leakage.

2. **Model:** gradient boosting (XGBoost or LightGBM). Choices: shallow trees (depth 4–6), learning rate 0.05, 500 iterations with early stopping. Calibration via Platt scaling or isotonic regression.

3. **Cross-validation:** `TimeSeriesSplit` with gap of 1–2 days (embargo to prevent feature-window leakage). Never shuffle.

4. **Imbalance:** use `scale_pos_weight` $= n_{\text{neg}}/n_{\text{pos}}$ in XGBoost, or sample majority class down (undersampling) in training folds only. SMOTE is risky on mixed categorical/numerical features.

5. **Threshold selection:** compute expected cost over grid of thresholds using cost matrix (FP cost = analyst review time; FN cost = lost-fraud amount); pick the threshold that minimizes expected cost.

6. **Production concerns:**
   - *Drift monitoring*: daily KS tests on input features and predicted-probability distributions.
   - *Label feedback delay*: chargebacks arrive 30–60 days later; use provisional labels with a delay buffer.
   - *Feature availability*: verify every production feature is computable at inference time without future information (especially rolling features anchored to "now").
   - *A/B testing*: roll out to 5% of transactions, compare against the incumbent model on realized fraud rate and analyst workload.
   - *Retraining cadence*: monthly retrain on the latest 6–12 months; keep a champion/challenger setup.
   - *Explainability*: SHAP values for analyst review and regulatory compliance (Basel, SR 11-7).