Unsupervised Learning

K-Means, Hierarchical Clustering, PCA, and ML Pipelines

Lecture Companion Notes

Data Science with Python – BSc Course

Joerg Osterrieder
April 10, 2026

These notes accompany the unsupervised learning lectures (L29–L32). 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. Read before lecture for preparation, revisit after for deeper understanding.

Contents

1. Sorting Without Labels – What Is Unsupervised Learning?

Opening Problem: The Portfolio Manager's 500 Stocks

You are a portfolio manager at a mid-size asset management firm. On your screen: a spreadsheet with 500 stocks. Each stock has a row of numbers—annual return, daily volatility, market capitalization, price-to-earnings ratio, beta, dividend yield. Six features per stock. Fifteen hundred cells of data.

Your boss walks over and asks a simple question: "Which stocks belong together?" She does not say "growth versus value." She does not hand you sector labels. She wants you to discover the groupings yourself, straight from the numbers. No one has told you what the categories are or how many there should be.

You could sort by one column—say, volatility—and draw arbitrary cutoffs. But that ignores the other five features. You could eyeball a scatterplot of two features, but that ignores the other four. The real structure lives in six-dimensional space, and your eyes cannot see six dimensions.

This section introduces unsupervised learning: the branch of machine learning that finds structure in data when nobody tells you what the structure should look like.

Discovery Question

You have trading data for 500 stocks but no labels—no one has told you which stocks are "growth" or "value" or "momentum." Can a computer discover these categories on its own? If so, how would you know if the groups it found are meaningful or random noise?

Two Kinds of Learning

Imagine two scenarios. In the first, a teacher gives you 1,000 photographs, each labeled "cat" or "dog." Your job is to learn the pattern so you can label new photographs. That is supervised learning: you have inputs and labels, and the goal is prediction.

In the second scenario, the same teacher dumps 1,000 unlabeled photographs on your desk and says: "Sort these into groups that make sense." No labels. No hints. You stare at the pile and start noticing patterns—fur color, ear shape, body size. You create your own categories. That is unsupervised learning.

The asymmetry is striking. Supervised learning has a target to chase and a metric to optimize. Unsupervised learning has neither. You are flying blind—the algorithm finds patterns, but nobody can tell you in advance whether those patterns are real or coincidental. This makes unsupervised learning simultaneously more creative and more dangerous than its supervised counterpart.

Figure 1 shows where unsupervised learning sits inside the broader machine learning landscape. Clustering—grouping similar observations—is its most common task, but dimensionality reduction (covered in Sections 5 and 6) is equally powerful.

Think about a sock drawer. You dump thirty socks on the floor. Nobody tells you the categories. You start grouping: black dress socks here, white athletic socks there, patterned socks in a third pile. You did not need labels. You used similarity—color, thickness, length—to discover groupings. Clustering algorithms do the same thing, except they work in spaces with dozens or hundreds of dimensions instead of three.

Notice that your grouping was not unique. You sorted by color. Your roommate might sort by fabric weight. Both groupings are valid. Neither is "correct" in an absolute sense. This ambiguity is fundamental to unsupervised learning. The results depend on the features you measure, the distance metric you use, and the algorithm you choose. Change any of these, and the groups change.

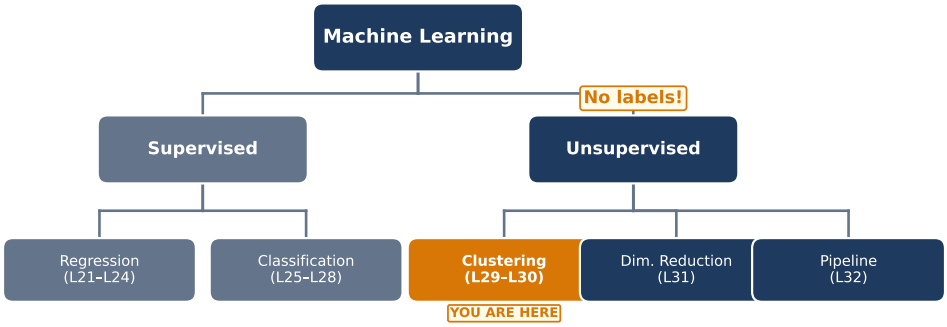


Figure 1: Where unsupervised learning fits in the machine learning taxonomy. Supervised learning predicts labels; unsupervised learning discovers structure.

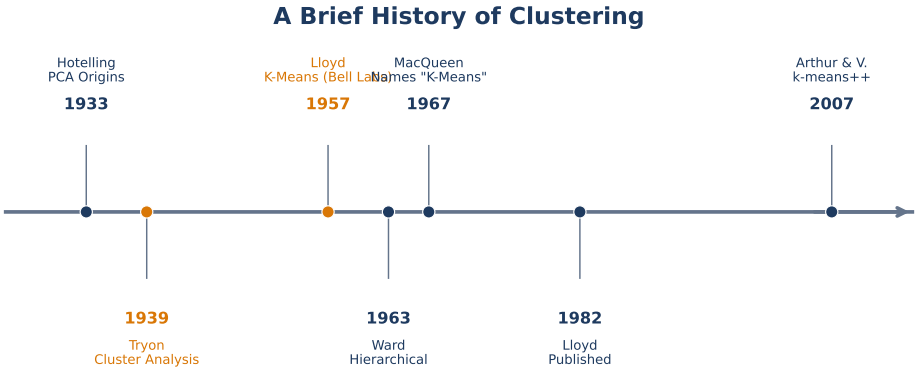
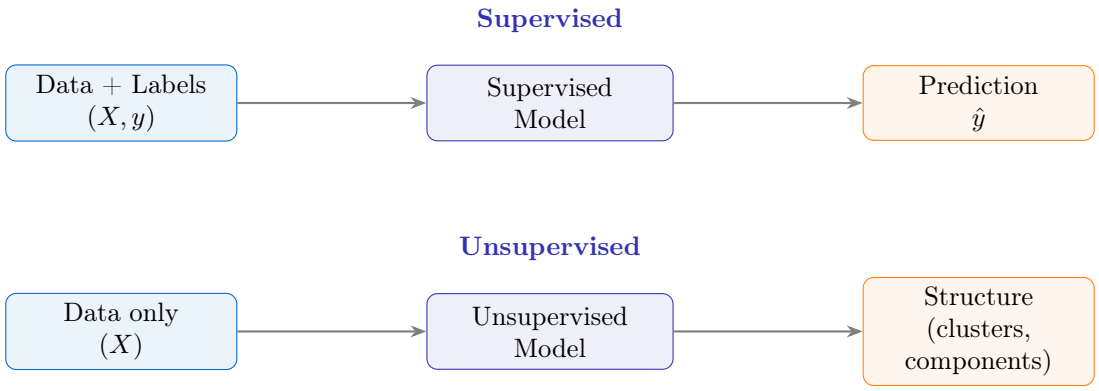


Figure 2: A brief history of clustering methods. The field spans psychology (Tryon, 1939), signal processing (Lloyd, 1957), and modern finance (Lopez de Prado, 2016).

The difference between supervised and unsupervised learning comes down to one thing: does the training data include a target column or not?



Supervised learning maps inputs to a known target. Unsupervised learning has no target—it discovers patterns, groupings, or compressed representations directly from the input features.

The Core Concepts

Unsupervised learning: A class of machine learning algorithms that operate on data without labels. The goal is to discover hidden structure—clusters, latent factors, compressed representations—rather than to predict a target variable.

Clustering: The task of partitioning observations into groups (clusters) such that observations within each group are more similar to each other than to observations in other groups.

Dimensionality reduction: The task of representing high-dimensional data in fewer dimensions while preserving as much information as possible. PCA (Sections 5–6) is the most common method.

Now look at what clustering actually does to data. Figure 3 shows the core idea: a scatter plot of unlabeled points gets colored by cluster membership. Points within the same cluster sit close together. Points in different clusters sit far apart.

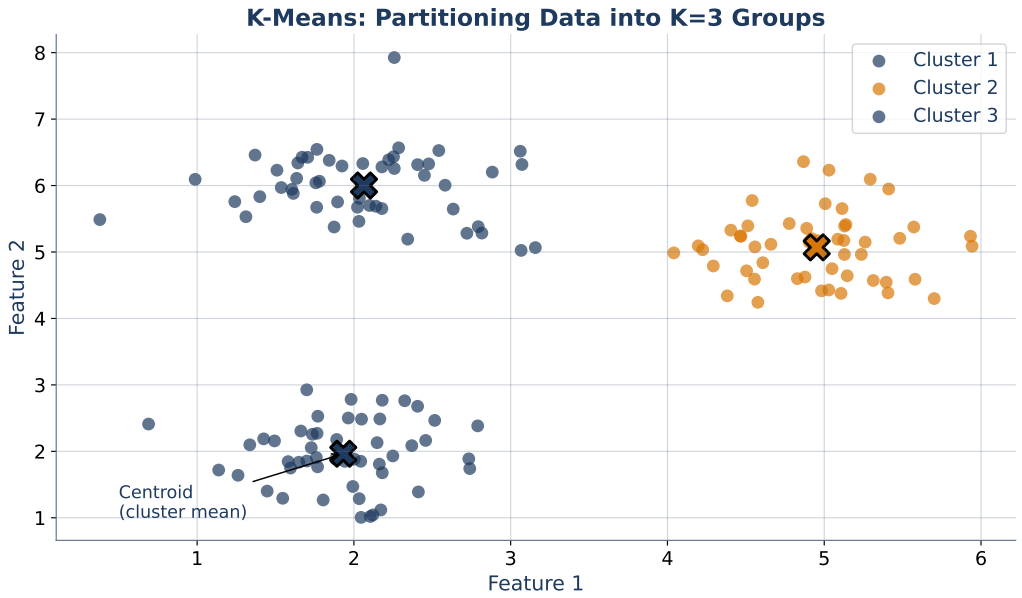


Figure 3: The clustering idea. Data starts unlabeled; the algorithm assigns each point to a group based on proximity.

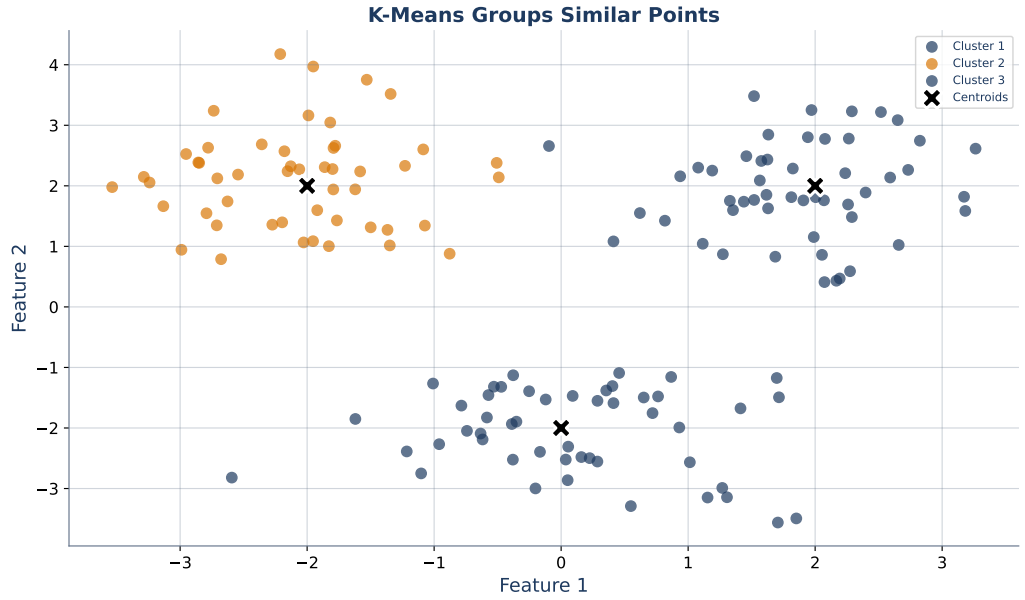


Figure 4: Points within a cluster share similar characteristics. The algorithm discovers these groupings without being told how many groups to expect.

But "similar" is not a self-evident concept. Similar in what sense? Euclidean distance treats every dimension equally. Correlation ignores magnitude and focuses on co-movement. Manhattan distance sums absolute differences instead of squaring them. Each choice produces different clusters.

And there is a subtler trap. If your features are on wildly different scales—market capitalization in billions versus daily returns in hundredths of a percent—raw Euclidean distance will be dominated entirely by market cap. The returns will contribute almost nothing. Two stocks with identical returns but different market caps would be placed in different clusters. Two stocks with identical market caps but opposite return patterns would be placed in the same cluster. The algorithm does not know that returns matter more than market cap. You have to tell it, by scaling the features first.

Feature scaling: Transforming features so they occupy comparable ranges before computing distances. StandardScaler subtracts the mean and divides by the standard deviation for each feature. Without scaling, large-scale features dominate and small-scale features are ignored.

Definition: Clustering Problem

Given a dataset $X = \{x_1, x_2, \ldots, x_n\}$ with $x_i \in \mathbb{R}^p$ (each observation has p features), find a partition $\{C_1, C_2, \ldots, C_K\}$ such that:

- Every observation belongs to exactly one cluster: $\bigcup_{k=1}^{K} C_k = X$ and $C_j \cap C_k = \emptyset$ for $j \neq k$.
- Observations within the same cluster are "similar" (by some distance measure).
- Observations in different clusters are "dissimilar."

The challenge: neither K nor the distance measure is given by the data itself. Both are choices the analyst must make.

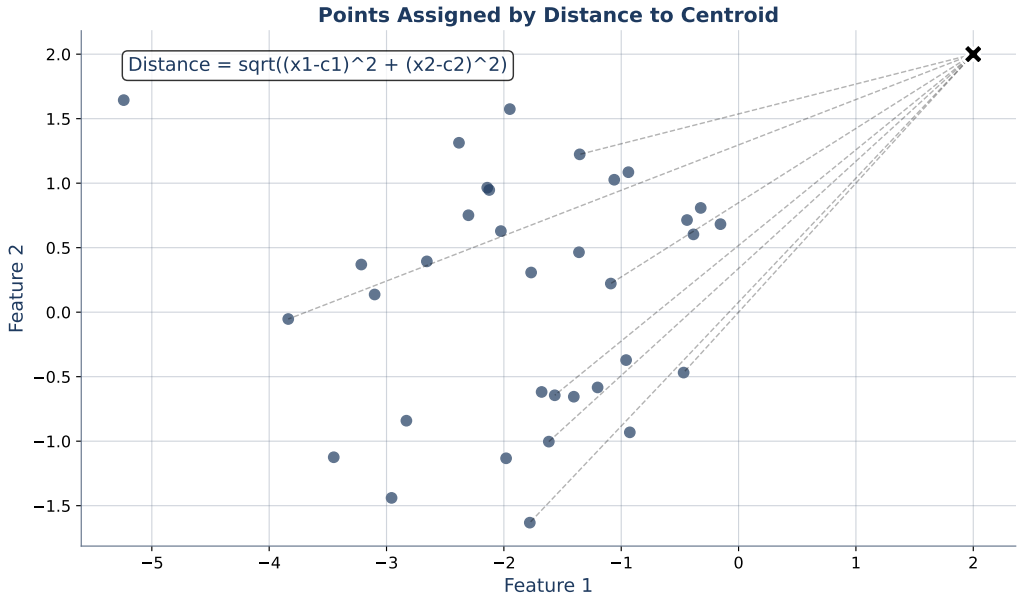


Figure 5: Each point is assigned to the cluster with the nearest centroid. Distance is Euclidean by default—which means feature scaling is critical.

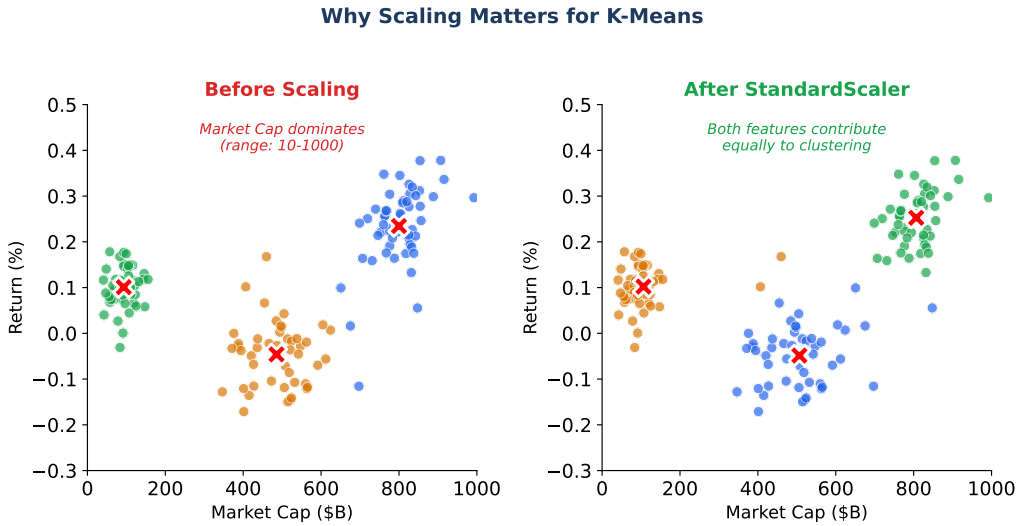


Figure 6: Left: unscaled features. Market cap dominates and returns are invisible. Right: after StandardScaler, both features contribute equally to distance calculations.

Common Misconceptions about Unsupervised Learning

(1) "Unsupervised learning finds the 'true' groups." It finds structure relative to the features and distance metric you chose. Change the features, change the groups. There may be no single correct grouping.
(2) "You don't need to preprocess data for clustering." Distance-based methods are extremely sensitive to scale. If you feed raw features to K-Means without standardizing, the results will be meaningless.
(3) "Unsupervised learning is easier than supervised." It is harder. With supervised learning, labels guide the algorithm and you can measure accuracy. Without labels, you have no ground truth to compare against. Evaluating cluster quality requires indirect measures like silhouette scores.

Worked Examples

Worked Example 1: Supervised vs. Unsupervised in Finance

A bank has two tasks:
Task A – Fraud detection: Each transaction is labeled "fraud" or "not fraud." The bank trains a classifier (supervised) on labeled data to predict fraud on new transactions.
Task B – Customer segmentation: The bank has 100,000 customers with data on income, spending patterns, account balances, and credit scores. No labels exist—nobody has pre-defined the customer categories. The bank runs a clustering algorithm (unsupervised) to discover natural segments: "high-income savers," "young spenders," "retired fixed-income," etc.
The difference is clear: Task A has labels and predicts a target. Task B has no labels and discovers structure.

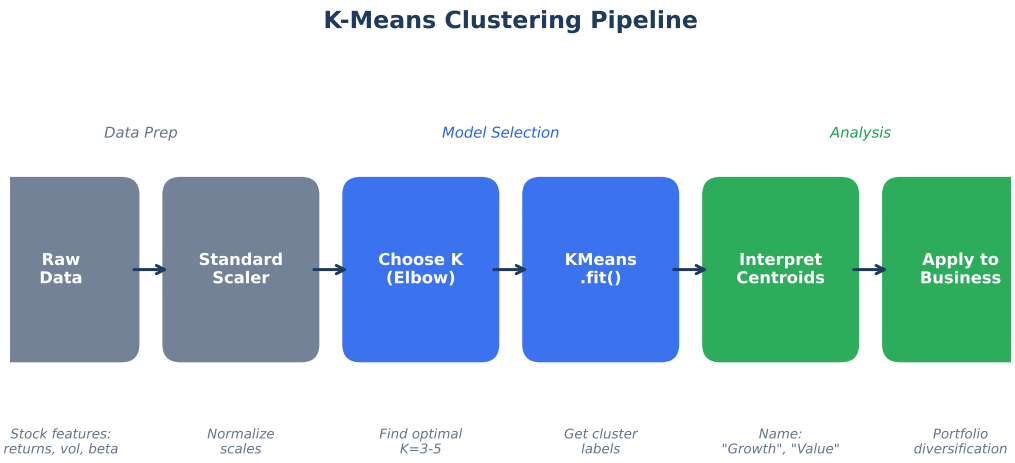


Figure 7: End-to-end unsupervised learning workflow: collect data, scale features, choose algorithm, evaluate clusters, interpret results.

Worked Example 2: Manual Clustering by Hand

Six stocks have the following (simplified) features after standardization:

Stock	Return (scaled)	Volatility (scaled)
AAPL	0.8	−0.2
MSFT	0.9	−0.3
TSLA	1.5	2.1
JNJ	−0.5	−1.0
PG	−0.4	−0.9
GME	2.0	2.5

By eye, you can spot three groups: (AAPL, MSFT) cluster together with moderate return and low volatility. (TSLA, GME) cluster together with high return and high volatility. (JNJ, PG) cluster together with low return and very low volatility. A clustering algorithm would reach the same conclusion by measuring pairwise distances.
Notice that these groups cut across traditional sector lines. AAPL (technology) and MSFT (technology) share a cluster, but TSLA (also technology by some classifications) is in a different cluster because its *behavior*—high return, high volatility—is different. Clustering by behavior often reveals groupings that sector labels miss.

Historical Background: Robert Tryon and the First Cluster Analysis (1939)

In 1939, the American psychologist Robert Choate Tryon published *Cluster Analysis*, the first systematic treatment of grouping observations by similarity. Tryon was studying behavioral patterns in rats and wanted to identify natural groupings among experimental subjects without imposing predefined categories.
His method was laborious—performed entirely by hand with pencil, paper, and patience. He computed correlation matrices, grouped variables by correlation, and iterated until the groupings stabilized. The procedure took weeks for datasets we would process in milliseconds.
What Tryon did by hand with behavioral data, you do with `KMeans(n_clusters=3).fit(X)` in one line. The intellectual leap was the same: let the data reveal its own structure rather than forcing it into preconceived categories.

Problem 1.1 (Easy)

Classify each of the following as supervised or unsupervised: (a) predicting house prices from square footage, (b) grouping customers by purchase behavior, (c) detecting spam emails using labeled data, (d) reducing 200 stock features to 5 components, (e) predicting loan default from credit score.
Solution: see Appendix.

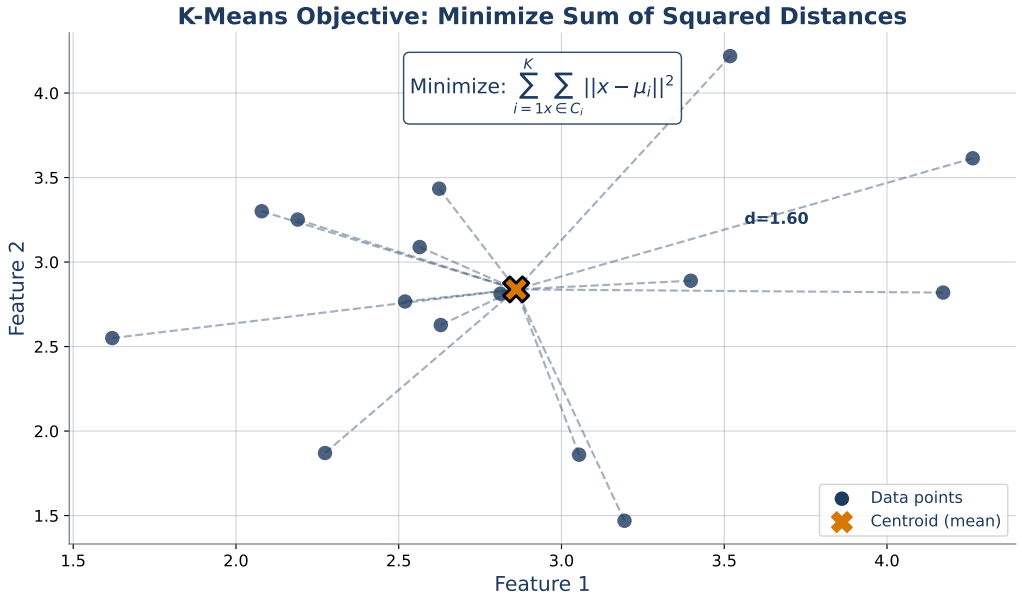


Figure 8: Preview of the K-Means objective function. The algorithm minimizes the total squared distance from each point to its assigned centroid. We formalize this in Section 2.

Problem 1.2 (Easy)

A dataset has two features: annual income (in dollars, ranging from \$20,000 to \$500,000) and age (in years, ranging from 18 to 80). You run K-Means on the raw data. Which feature will dominate the distance calculation, and why? What should you do before clustering?
Solution: see Appendix.

Problem 1.3 (Medium)

Given six data points in 2D: $A = (1, 2)$, $B = (1.5, 1.8)$, $C = (5, 8)$, $D = (8, 8)$, $E = (1, 0.6)$, $F = (9, 11)$. Compute the Euclidean distance between every pair. Which two points should be merged first if you want to build clusters by grouping the closest pair?
Solution: see Appendix.

Problem 1.4 (Medium)

A dataset has features "height in cm" and "weight in kg." After standardization, height has mean 0, std 1 and weight has mean 0, std 1. Explain what happens to the clustering result if you accidentally standardize height but forget to standardize weight (which remains in kg, ranging from 45 to 120).
Solution: see Appendix.

Problem 1.5 (Hard)

A bank wants to segment 100,000 customers using five features: account balance, monthly transactions, credit score, age, and tenure. Design a complete unsupervised learning pipeline for this task. Specify: (a) which preprocessing steps are needed and why, (b) which algorithm you would start with and why, (c) how you would evaluate the quality of the resulting segments, (d) how you would present the results to a non-technical manager.
Solution: see Appendix.

Connecting Forward

We now know what unsupervised learning is and why it matters. We have seen that clustering groups similar observations together, that feature scaling is mandatory for distance-based methods, and that the analyst—not the algorithm—must choose the number of clusters and the distance metric.

But we have not yet described a specific algorithm. How does a machine actually find clusters? Section 2 answers this with K-Means: the simplest, fastest, and most widely used clustering algorithm. It works by placing K centroids, assigning each point to its nearest centroid, recomputing centroids, and repeating until nothing changes. The mechanics are surprisingly straightforward. The subtleties—choosing K, handling initialization, recognizing when K-Means is the wrong tool—are where the real learning begins.

Key Takeaway: Unsupervised learning discovers structure in data without labels—but the discovered structure is only as meaningful as the features and distance metric you choose.

2. Finding the Hidden Groups – K-Means Clustering

Opening Problem: Segmenting 100,000 Bank Customers

A retail bank has 100,000 customers. The marketing team wants to target them with personalized offers, but nobody has defined the customer categories. The bank has data: average balance, number of monthly transactions, credit utilization, average transaction amount, and months since account opening. Five features per customer.

The marketing director says: "Give me three to five segments I can build campaigns around. I want to know who the high-value savers are, who the heavy spenders are, and who is at risk of leaving. But I cannot tell you in advance how many segments there are or what they should look like."

You need an algorithm that takes 100,000 rows and five columns, discovers natural groupings, and hands back a label for each customer. That algorithm is K-Means. It is fast, intuitive, and works well when clusters are roughly spherical and similarly sized.

Discovery Question

You run K-Means with $K = 3$ and get three clean clusters. You run it again with $K = 3$ and get completely different clusters. Same data, same K, different answer. Why? And if the algorithm cannot even give you a stable answer, how can you trust any of its outputs?

The Algorithm, Step by Step

Forget the math for a moment. Here is what K-Means does in plain language. You start with a scatter plot of unlabeled points. You pick a number K—say, three. Then you do four things:

Step 1: Look at the raw data. No colors, no groups. Just a cloud of dots.

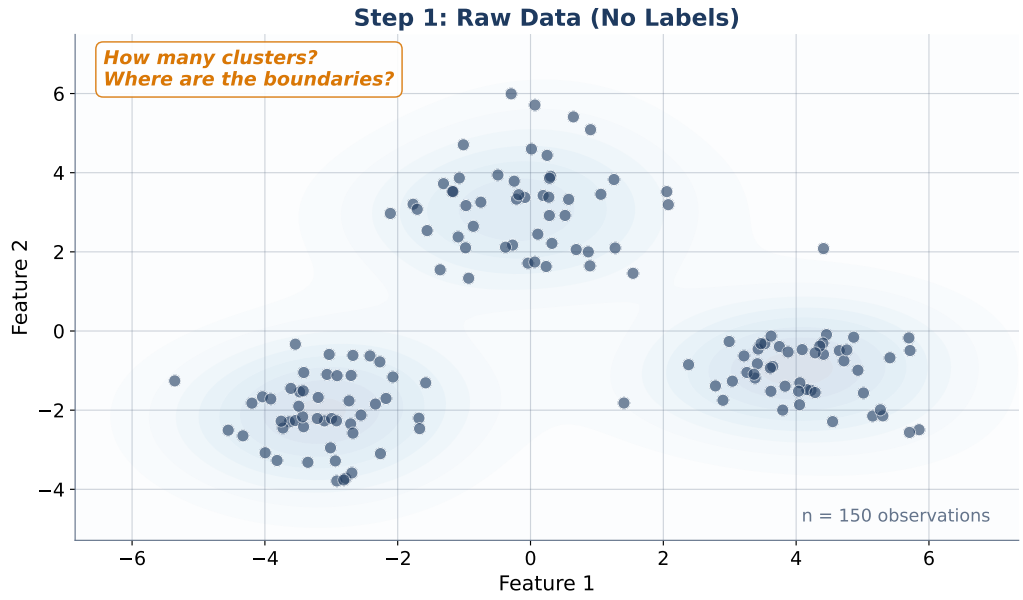


Figure 9: Step 1: Raw, unlabeled data. The algorithm sees only coordinates—no structure yet.

Step 2: Drop three pins randomly into the scatter plot. These are your initial centroids—the starting guesses for cluster centers.

Step 3: Assign every point to its nearest centroid. Each point gets a color matching its assigned centroid.

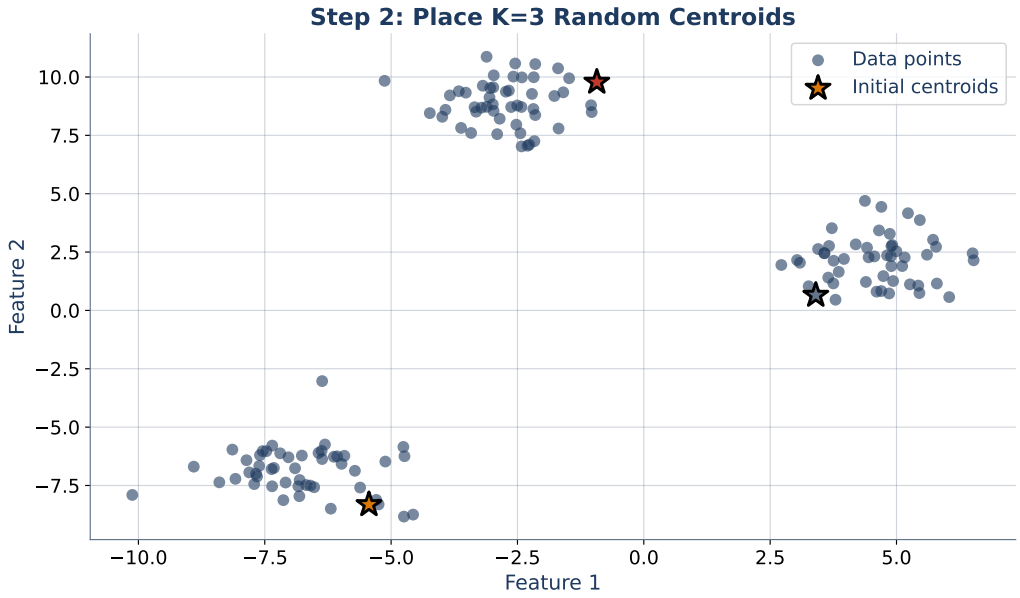


Figure 10: Step 2: Initialize three centroids (marked with stars or crosses). Their starting positions are random.

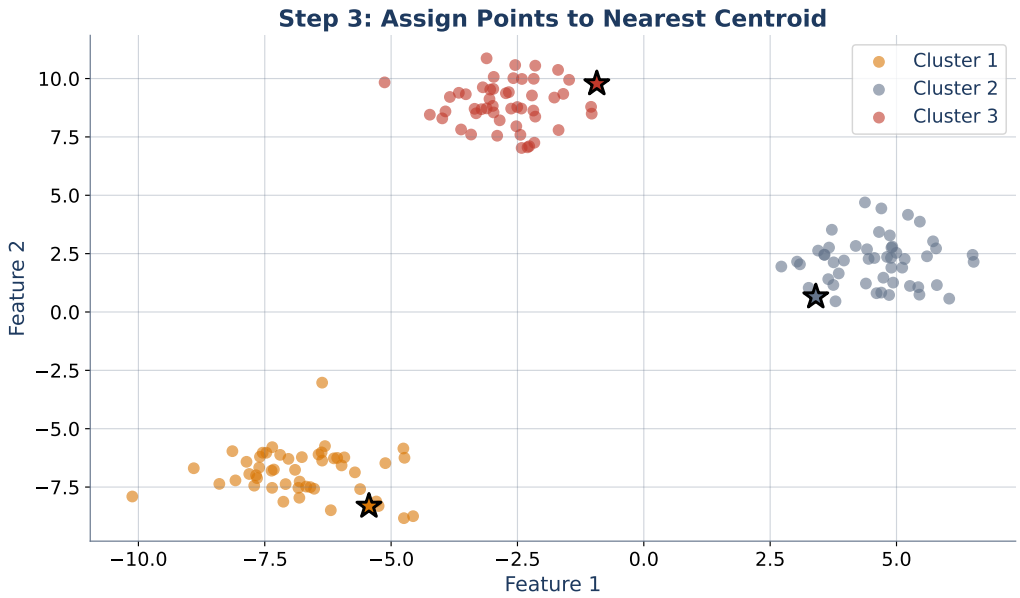


Figure 11: Step 3: Assign each point to its nearest centroid using Euclidean distance. This creates a rough partition.

Step 4: Recompute each centroid as the mean of all points assigned to it. Then go back to Step 3. Repeat until no point changes its assignment.

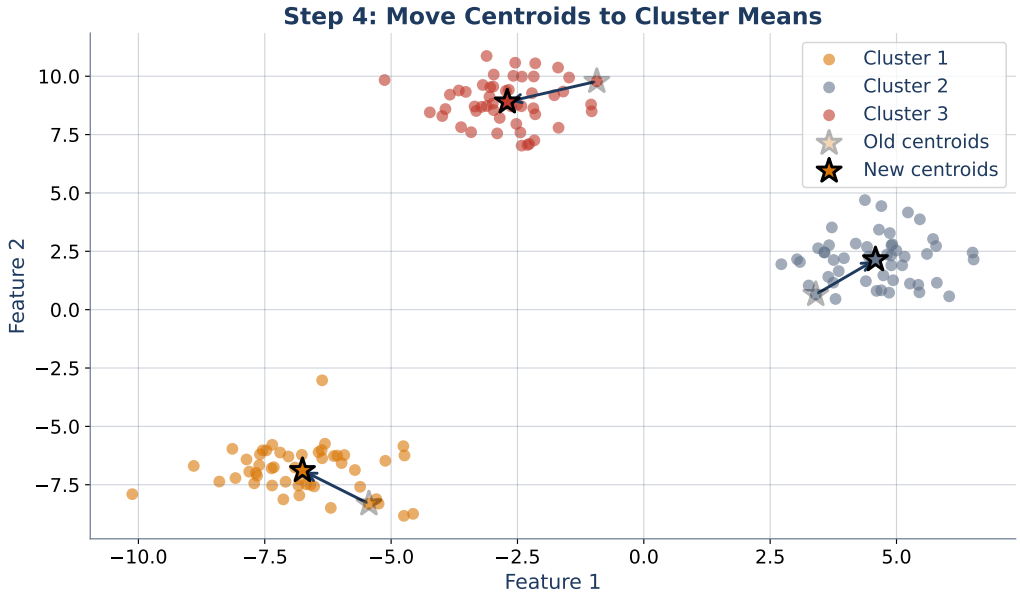


Figure 12: Step 4: Move each centroid to the center of its cluster. Then reassign. Repeat until stable.

Picture a game of musical chairs in reverse. The chairs (centroids) move toward the people (data points), and the people shift to sit in the nearest chair. After a few rounds, everyone is settled and nothing moves. That is convergence.

The beauty of K-Means is its simplicity. The entire algorithm is just two operations—assign and recompute—repeated in a loop. There is no gradient descent, no loss function to differentiate, no learning rate to tune. Each iteration is deterministic given the current state. The only randomness is in the initialization. This makes K-Means fast: a dataset with 100,000 points and 5 features typically converges in under a second on a modern laptop.

The Objective: Minimize Within-Cluster Variance

Centroid: The center of a cluster, computed as the arithmetic mean of all points assigned to that cluster. In a two-dimensional space, the centroid has coordinates $(\bar{x}_1, \bar{x}_2)$ where $\bar{x}_j$ is the mean of feature j across all points in the cluster.

Inertia (WCSS): Within-Cluster Sum of Squares. The total squared Euclidean distance from every point to the centroid of its assigned cluster. Lower inertia means tighter, more compact clusters.

K-Means minimizes a single number: the sum of squared distances from each point to its assigned centroid. Smaller is better.

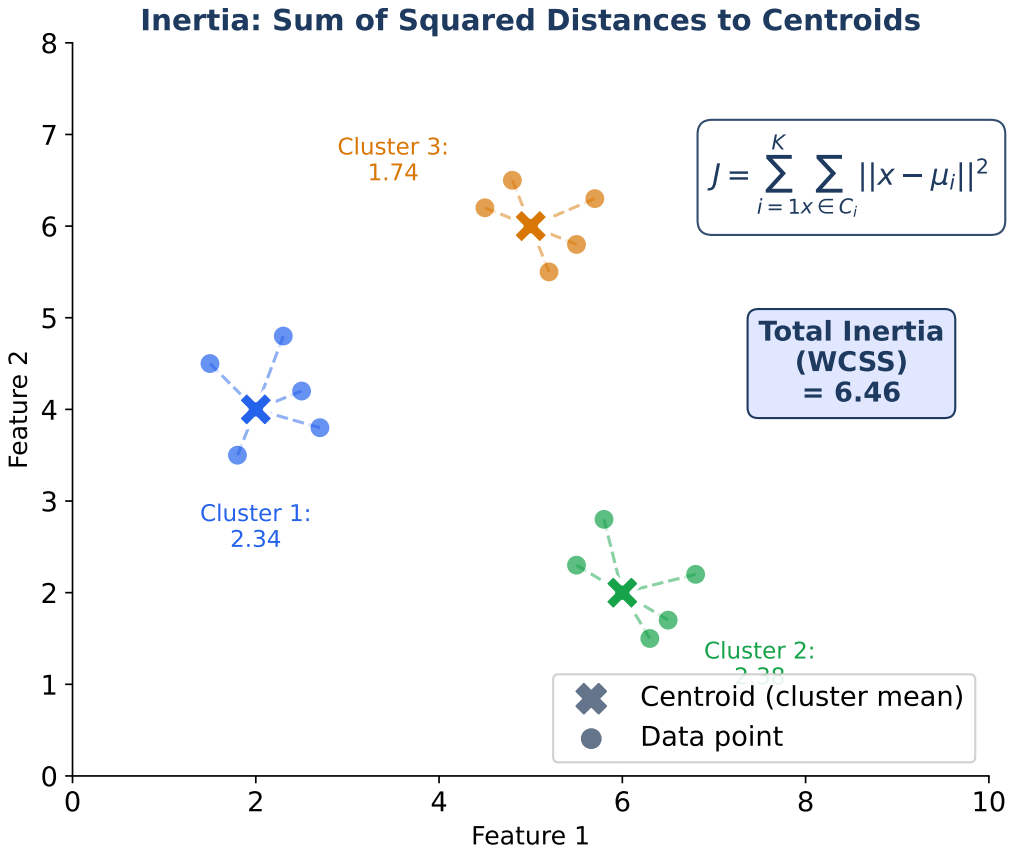


Figure 13: The inertia (WCSS) formula visualized. Each colored region shows the squared distances being summed within one cluster.

Key Formula: K-Means Objective

K-Means minimizes the **within-cluster sum of squares** (WCSS), also called **inertia**:

$$\text{WCSS} = \sum_{k=1}^{K} \sum_{x_i \in C_k} \|x_i - \mu_k\|^2$$

where:

- K is the number of clusters
- C_k is the set of points in cluster k
- $\mu_k = \frac{1}{|C_k|} \sum_{x_i \in C_k} x_i$ is the centroid of cluster k
- $\|x_i - \mu_k\|^2$ is the squared Euclidean distance from point x_i to centroid μ_k

Plain English: For each cluster, measure how far every member is from the center. Square those distances. Add them up. K-Means finds the assignment that makes this total as small as possible.

Why does K-Means converge? At each iteration, the assignment step cannot increase WCSS (every point moves to a closer centroid or stays put), and the centroid-update step cannot increase WCSS (the mean minimizes the sum of squared distances from a set of points). Since WCSS decreases with every iteration and is bounded below by zero, the algorithm must stop in finite time.

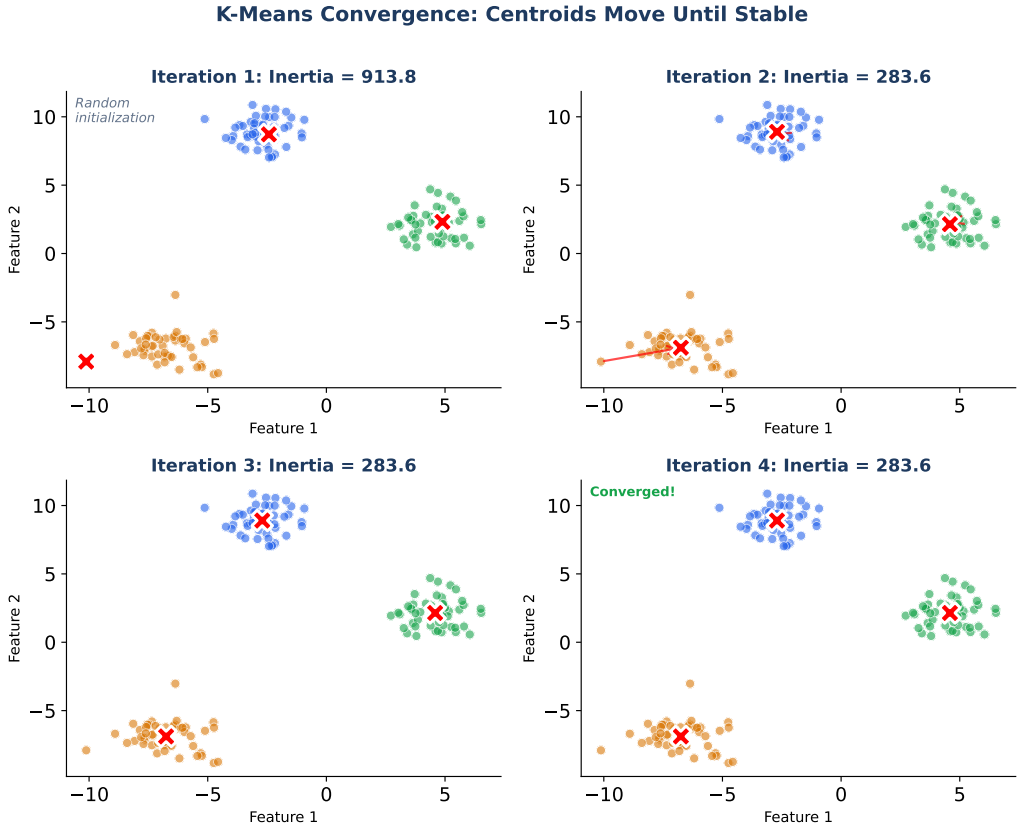


Figure 14: Inertia drops rapidly in the first few iterations and then levels off. Most K-Means runs converge in 10–20 iterations.

But convergence does not mean optimality. K-Means finds a *local* minimum of WCSS, not the global minimum. Different starting positions can lead to different local minima. Think of a hilly landscape with many valleys. Convergence guarantees you reach the bottom of *some* valley, not the deepest one. That is why initialization matters, and why K-Means++ was invented.

The default strategy in scikit-learn runs K-Means 10 times with different random initializations (`n_init=10`) and keeps the result with the lowest WCSS. This brute-force repetition costs 10x in compute time but dramatically reduces the chance of landing in a bad local minimum.

K-Means++ initialization: A smart initialization strategy that spreads the initial centroids apart. The first centroid is chosen at random. Each subsequent centroid is placed at a point with probability proportional to its squared distance from the nearest existing centroid. This dramatically reduces the chance of a poor local minimum. It is the default in scikit-learn.

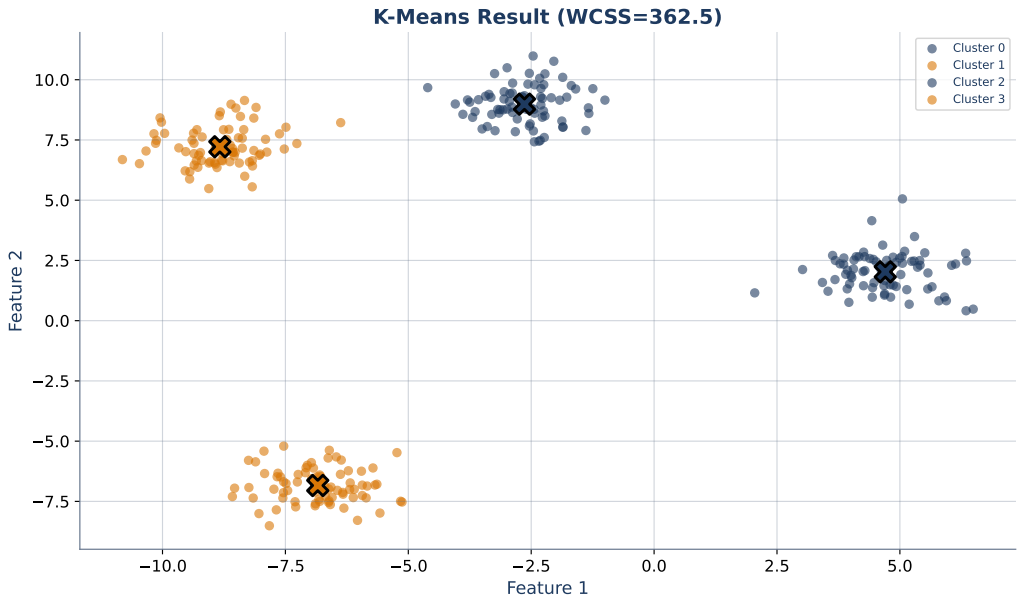
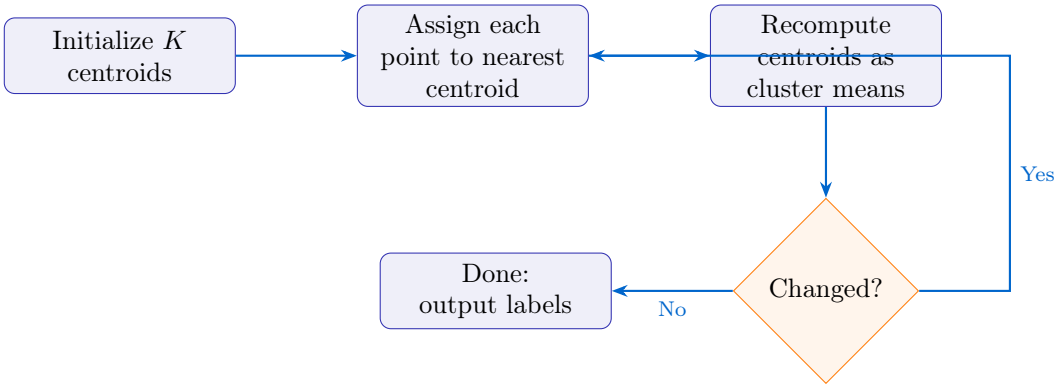


Figure 15: Random initialization (left) can place centroids in the same cluster. K-Means++ (right) spreads them apart for better convergence.



The diagram shows the K-Means loop: initialize, assign, recompute, check for convergence, repeat if needed.

Now: how do you choose K?

Choosing K: Elbow and Silhouette

Elbow method: Plot WCSS (inertia) against K for $K = 1, 2, 3, \ldots$. As K increases, inertia always decreases. The "elbow"—the point where the curve bends sharply from steep to flat—suggests the best K. Before the elbow, adding clusters captures real structure. After the elbow, you are just splitting noise.

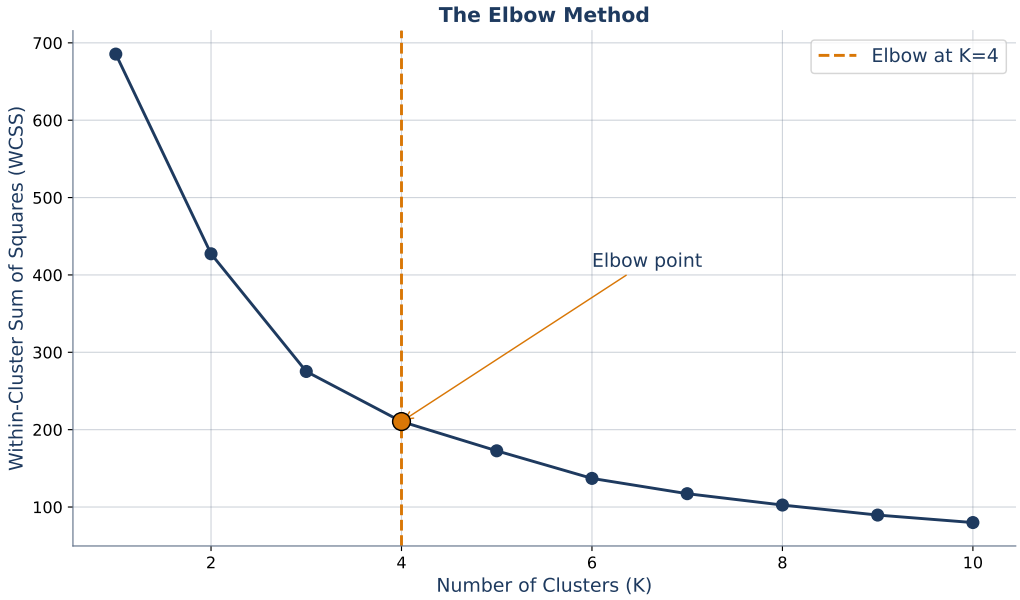


Figure 16: The elbow method. Inertia drops steeply from $K=1$ to $K=3$, then flattens. The elbow at $K=3$ suggests three clusters.

The elbow method has a weakness: many real datasets do not produce a sharp bend. The curve slopes gently with no obvious elbow, especially when clusters overlap or have unequal sizes. When that happens, you need a second opinion. The silhouette score provides exactly that—a per-point measure of cluster quality that does not depend on WCSS at all.

Silhouette score: For each data point, the silhouette score measures how similar the point is to its own cluster versus the nearest neighboring cluster. The formula is $s_i = \frac{b_i - a_i}{\max(a_i, b_i)}$, where a_i is the average distance from point i to all other points in its cluster and b_i is the average distance from point i to all points in the nearest other cluster. The score ranges from -1 (wrong cluster) to $+1$ (perfectly clustered).

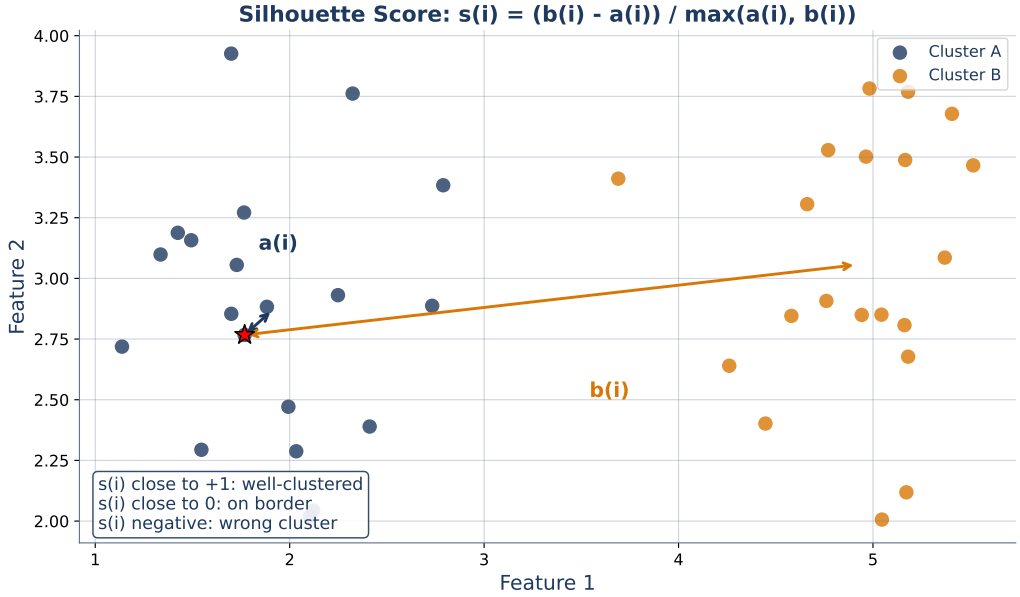


Figure 17: The silhouette score for one point. a is the average within-cluster distance, b is the average nearest-cluster distance. Higher is better.

Key Formula: Silhouette Score

For data point i:

$$s_i = \frac{b_i - a_i}{\max(a_i, b_i)}$$

where:

- a_i = average distance from point i to all other points in the same cluster (cohesion)
- b_i = average distance from point i to all points in the nearest other cluster (separation)
- $s_i \in [-1, 1]$: values near $+1$ mean well-clustered, near 0 means on the boundary, near -1 means likely in the wrong cluster

The **average silhouette score** across all n points gives a global quality measure:

$$\bar{s} = \frac{1}{n} \sum_{i=1}^{n} s_i$$

Rule of thumb: $\bar{s} > 0.5$ indicates reasonable structure; $\bar{s} > 0.7$ indicates strong structure.

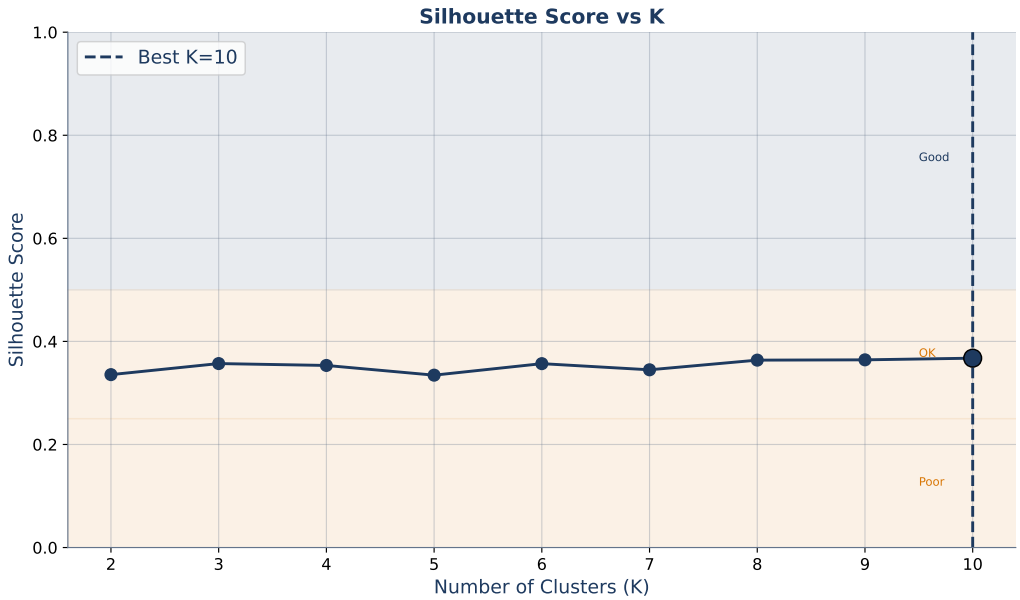


Figure 18: Average silhouette score for different values of K. The peak indicates the best separation between clusters. Combine this with the elbow plot for a robust choice of K.

Definition: K-Means Algorithm

K-Means is an iterative partitioning algorithm that assigns n observations in $\mathbb{R}^p$ to exactly K clusters by minimizing within-cluster sum of squares (WCSS). The algorithm alternates between assigning each point to the nearest centroid and recomputing centroids as cluster means. Convergence to a local minimum is guaranteed, but the result depends on initialization. K-Means++ provides a principled initialization strategy. The algorithm assumes clusters are spherical, of roughly equal size, and separated by linear boundaries.

Common Misconceptions about K-Means

(1) "The elbow method always gives a clear answer." Many real datasets produce gradual curves with no sharp elbow. When the elbow is ambiguous, combine with silhouette scores and domain knowledge.
(2) "K-Means works for any shape of clusters." It assumes spherical, equally-sized clusters because it uses Euclidean distance to the centroid. Crescent-shaped, ring-shaped, or highly elongated clusters will be misclassified.
(3) "Running K-Means twice gives the same result." The result depends on random initialization. Two runs with different random seeds can produce different clusterings. Use K-Means++ (the sklearn default) and set `random_state=42` for reproducibility, or run `n_init=10` and take the best result.

Worked Examples

Worked Example 1: Two Iterations of K-Means by Hand

Eight data points in 2D: $A=(1,1)$, $B=(1.5,2)$, $C=(3,4)$, $D=(5,7)$, $E=(3.5,5)$, $F=(4.5,5)$, $G=(3.5,4.5)$, $H=(6,6)$.
Set $K=2$. Initialize: $\mu_1=(1,1)$ (point A) and $\mu_2=(5,7)$ (point D).
Iteration 1 – Assign: Compute distance from each point to both centroids.

- Cluster 1 ($\mu_1=(1,1)$): A, B (closest to $(1,1)$).
- Cluster 2 ($\mu_2=(5,7)$): C, D, E, F, G, H (closest to $(5,7)$).

Iteration 1 – Recompute: $\mu_1 = \frac{(1,1)+(1.5,2)}{2} = (1.25, 1.5)$ and $\mu_2 = \frac{(3,4)+(5,7)+(3.5,5)+(4.5,5)+(3.5,4.5)+(6,6)}{6} = (4.25, 5.25)$.
Iteration 2 – Reassign: Point $C=(3,4)$: distance to $\mu_1=(1.25,1.5)$ is $\sqrt{1.75^2+2.5^2}=3.05$; distance to $\mu_2=(4.25,5.25)$ is $\sqrt{1.25^2+1.25^2}=1.77$. C stays in Cluster 2. Check all points; no assignment changes. **Converged.**
Final clusters: $\{A, B\}$ and $\{C, D, E, F, G, H\}$.

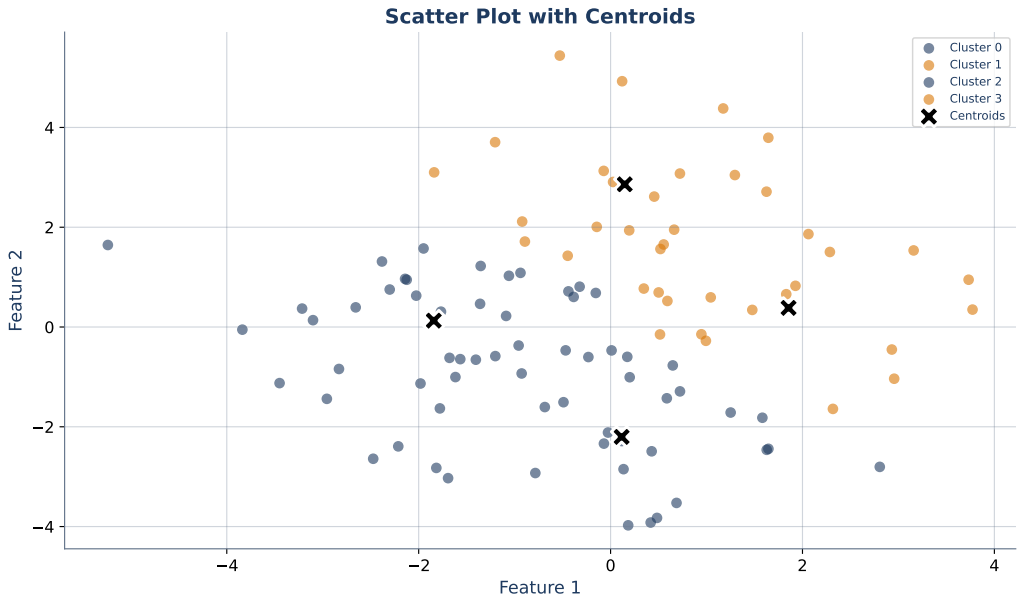


Figure 19: K-Means output: data colored by cluster assignment. Centroids marked distinctly. Visually verify that clusters make sense.

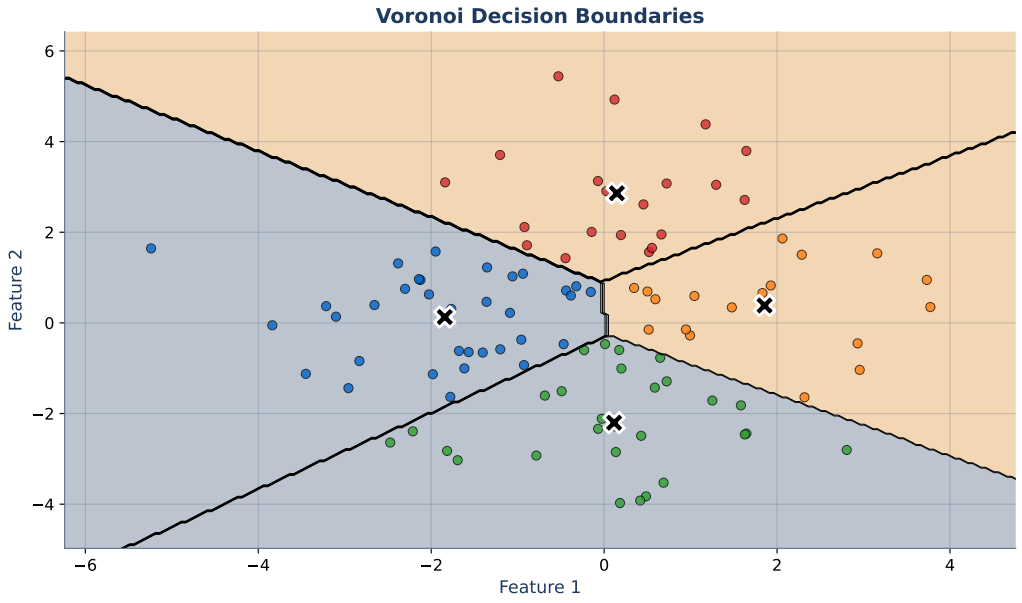


Figure 20: Voronoi boundaries show where one cluster ends and another begins. Each region contains all points closer to its centroid than to any other.

Worked Example 2: Interpreting Stock Clusters

After running K-Means with $K = 3$ on 50 stocks (features: annual return, volatility, beta, market cap, P/E ratio), you get three clusters with these centroid profiles:

Cluster	Return	Volatility	Beta	Mkt Cap	P/E
1 ("Blue Chips")	8%	12%	0.8	\$200B	18
2 ("Growth")	25%	28%	1.4	\$50B	45
3 ("Speculative")	40%	55%	2.1	\$5B	–

Interpretation: Cluster 1 contains large, stable companies with moderate returns—the "Blue Chips" that pension funds hold. Cluster 2 contains mid-cap growth stocks with higher risk and higher reward—the names that growth investors chase. Cluster 3 contains small, volatile names with extreme returns and no earnings (negative P/E)—the speculative plays that swing traders love.

Portfolio insight: Picking one stock from each cluster provides diversification across behavioral styles, not just sectors. Two tech stocks in different behavioral clusters may diversify better than one tech and one healthcare stock that happen to share the same cluster.

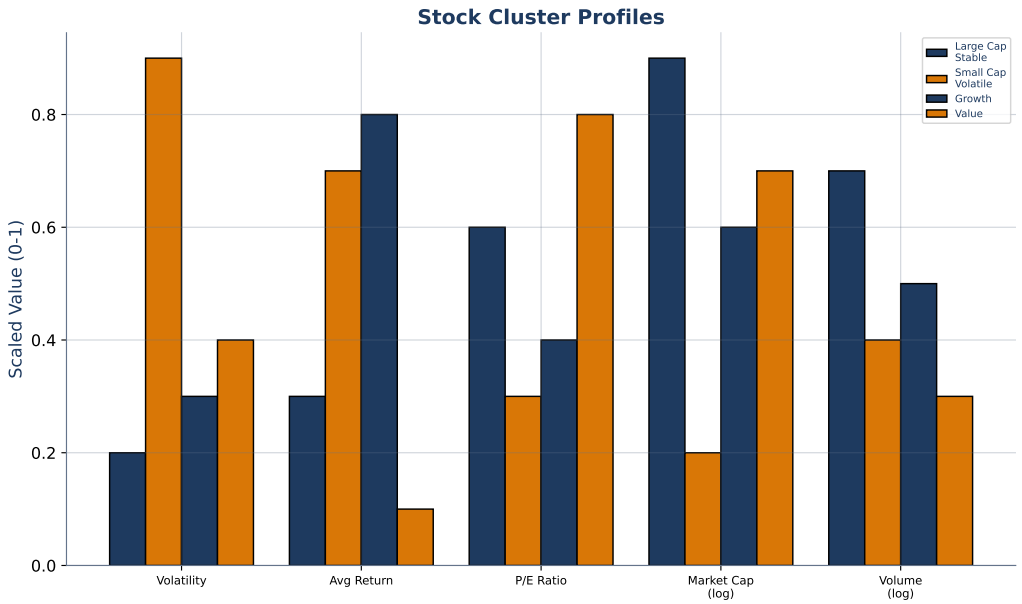


Figure 21: Stock cluster profiles. Bar charts of centroid features reveal what distinguishes each group.

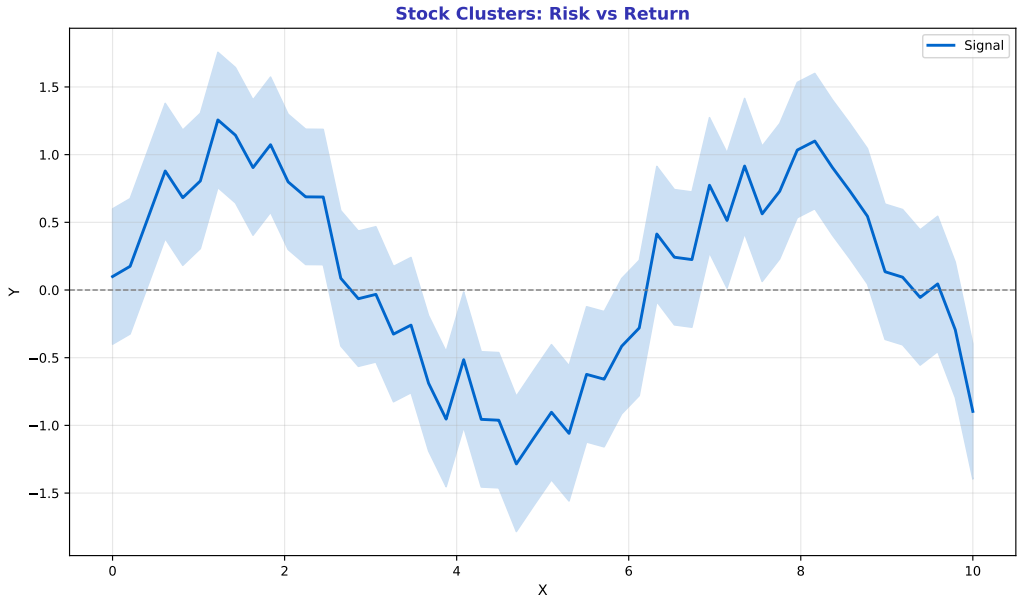


Figure 22: Risk-return map with K-Means clusters. Each cluster occupies a distinct region of the risk-return plane. Diversification strategy: select one stock per cluster.

Historical Background: Stuart Lloyd and K-Means at Bell Labs (1957)

In 1957, Stuart Lloyd was working at Bell Labs on a problem in signal processing called *pulse-code modulation*: how to represent a continuous signal with a finite set of discrete values while minimizing distortion. His solution—iteratively assign each signal sample to its nearest representative value, then recompute the representative as the mean of its assigned samples—is exactly the K-Means algorithm.

Lloyd circulated his result as an internal Bell Labs technical report. It was not formally published until 1982, a full 25 years later. In the meantime, James MacQueen independently proposed a similar algorithm in 1967 and gave it the name "K-Means."

Lloyd developed K-Means to optimize signal quantization at Bell Labs. The same algorithm now sorts your Spotify playlists and segments bank customers. The mathematics of minimizing distortion in signal processing is identical to minimizing within-cluster variance in data science.

Problem 2.1 (Easy)

Given the elbow plot in Figure 16, identify the best K and explain your reasoning in two sentences.

Solution: see Appendix.

Problem 2.2 (Easy)

A data point has average within-cluster distance $a = 2.0$ and average nearest-cluster distance $b = 5.0$. Compute the silhouette score s. Is this point well-clustered?

Solution: see Appendix.

Problem 2.3 (Medium)

Eight data points in 1D: $\{1, 2, 3, 10, 11, 12, 20, 21\}$. Initialize K-Means with $K = 3$ and starting centroids $\mu_1 = 1$, $\mu_2 = 10$, $\mu_3 = 20$. Perform two full iterations (assign and recompute). State the final clusters and centroids.
Solution: see Appendix.

Problem 2.4 (Medium)

You are given two elbow plots for two different datasets. Dataset A has a sharp bend at $K = 4$. Dataset B has a gradual, smooth curve with no visible elbow. For each dataset, explain whether the elbow method alone is sufficient to choose K, and what additional analysis you would perform for Dataset B.
Solution: see Appendix.

Problem 2.5 (Hard)

Prove that K-Means always converges. Specifically, show that the WCSS is non-increasing across iterations. (*Hint:* Argue separately for the assignment step and the centroid-update step.)
Solution: see Appendix.

Connecting Forward

K-Means is fast and effective for compact, spherical clusters. But it has three limitations that motivate the next two sections. First, you must choose K before running the algorithm, and sometimes neither the elbow nor the silhouette gives a clear answer. Second, K-Means produces a flat partition—no information about relationships between clusters. Third, K-Means cannot handle non-spherical shapes like crescents or chains.

Section 3 introduces hierarchical clustering, which addresses all three limitations. It builds a tree of nested groupings, lets you choose the number of clusters *after* seeing the full structure, and (with the right linkage method) can capture non-spherical shapes. The price is speed: hierarchical clustering is $O(n^3)$ versus K-Means's $O(nKt)$. For the 100,000-customer problem, K-Means finishes in seconds; hierarchical clustering might take hours.

In practice, the two methods are complementary, not competing. Use K-Means when you have large data and a rough idea of K. Use hierarchical clustering when you have smaller data and want to explore the full structure before committing to a partition. Section 4 then shows how hierarchical clustering powers a specific financial application—Hierarchical Risk Parity—where the tree structure is not just informative but directly determines portfolio weights.

Key Takeaway: K-Means partitions data into K spherical clusters by iteratively assigning points to the nearest centroid—always validate K with the elbow method and silhouette scores.

3. Building a Family Tree – Hierarchical Clustering and Dendrograms

Opening Problem: The Risk Manager's 50 Bonds

A risk manager at an insurance company holds a portfolio of 50 corporate bonds. She knows the bonds are not independent—when one auto manufacturer's debt weakens, other auto manufacturers tend to follow. She wants to understand *which bonds move together* and *how close the relationships are.*

K-Means could group the bonds into, say, four clusters. But it would not tell her that cluster 1 and cluster 2 are closely related (both in industrials) while cluster 3 (utilities) stands apart. She wants a tree—a hierarchy that shows relationships at every level, from individual bonds up to broad sectors.

That tree is called a *dendrogram*, and the method that builds it is hierarchical clustering.

Discovery Question

K-Means forces you to choose K before you start. What if you have no idea how many groups exist? Is there a method that shows you ALL possible groupings at once—from "everything in one big group" to "every point is its own group"—and lets you pick afterward?

Why a Tree?

When you organize your music collection, you do not dump everything into three bins. You create a hierarchy: "Rock" contains "Classic Rock" and "Alternative," and "Classic Rock" contains "Led Zeppelin" and "Pink Floyd." Each level of the hierarchy tells you something. At the top level: "Rock vs. Jazz vs. Classical." One level down: "Classic Rock vs. Punk vs. Indie." You can zoom in or zoom out.

K-Means gives you one flat partition. Pick $K = 3$ and you get three bins with no internal structure. Pick $K = 5$ and you get a completely different set of bins with no connection to the three-bin solution. There is no relationship between the $K = 3$ solution and the $K = 5$ solution. You cannot zoom in. You cannot zoom out. You are locked into one level of resolution.

Hierarchical clustering builds the full family tree: start with every point as its own cluster, merge the closest pair, then merge the next closest pair, and continue until everything is in one big cluster. The tree records every merge—which pair merged, at what distance, and in what order. You can "cut" the tree at any height to get any number of clusters. Cut high: two broad groups. Cut low: ten fine-grained groups. The relationships are nested: the two broad groups from the high cut are always unions of the fine-grained groups from the low cut.

The agglomerative (bottom-up) approach works like this. You have five points: A, B, C, D, E. Compute all pairwise distances. Find the closest pair—say A and B—and merge them into a cluster {A, B}. Now you have four clusters: {A, B}, {C}, {D}, {E}. Compute distances between all clusters. Find the closest pair again. Merge. Repeat until one cluster remains.

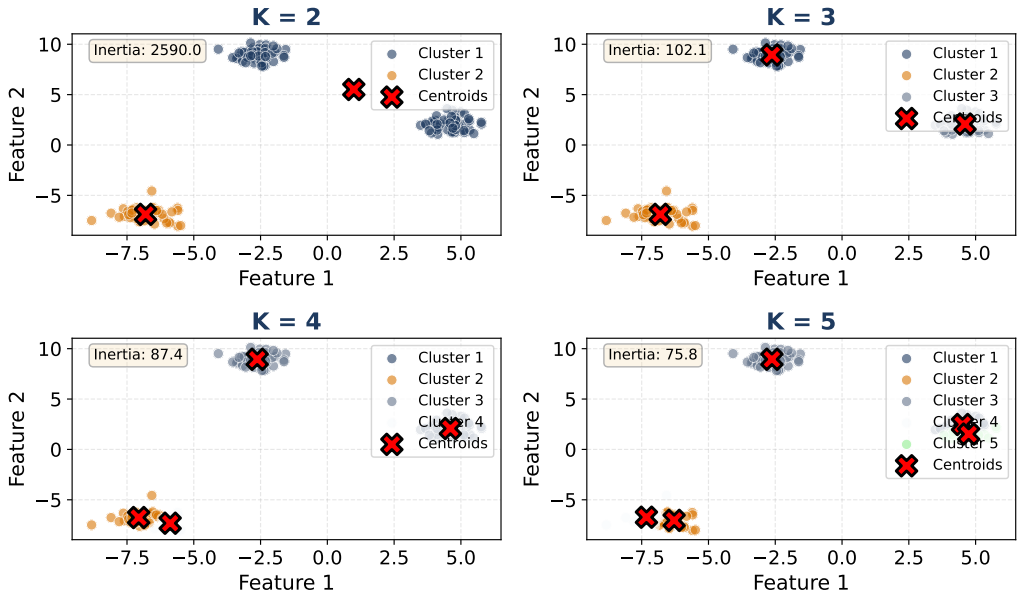


Figure 23: The K problem: elbow and silhouette sometimes disagree. Hierarchical clustering avoids this by showing all possible clusterings simultaneously.

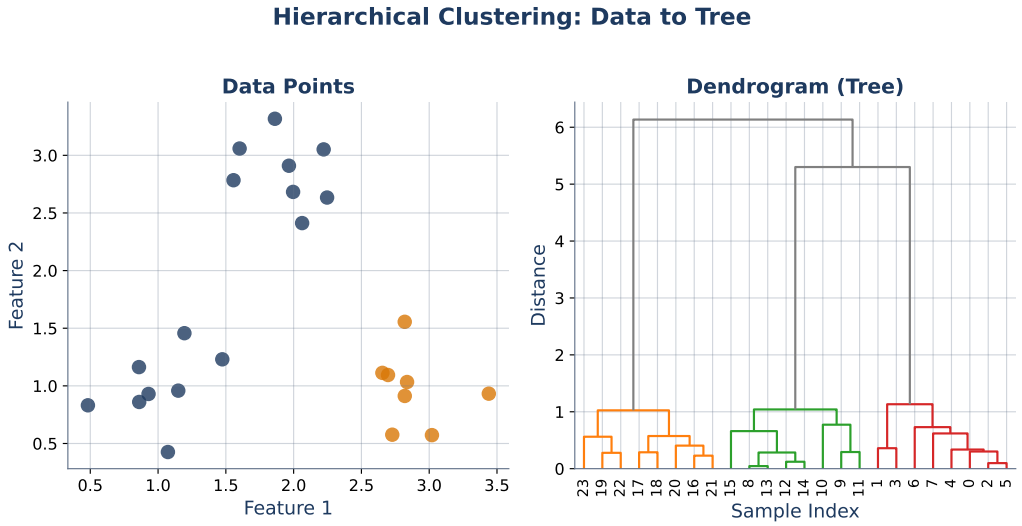


Figure 24: Hierarchical clustering: start with n individual clusters, merge the closest pair at each step, build a nested tree of groupings.

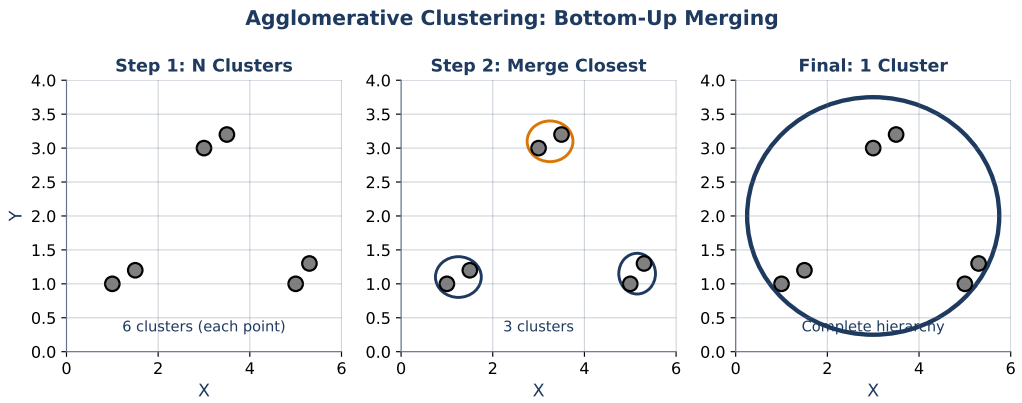
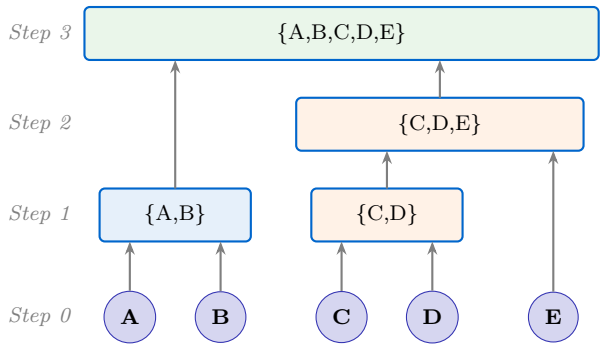


Figure 25: Agglomerative clustering step by step: five points merge into four clusters, then three, then two, then one.



The diagram shows the bottom-up merging process. Each row is one step. The tree records the order and distance of every merge.

Linkage Methods: How to Measure Distance Between Clusters

Once you have merged A and B into {A, B}, how do you measure the distance between {A, B} and {C}? This is the *linkage* question, and different answers produce very different trees.

Single linkage: Distance between two clusters = minimum distance between any point in one cluster and any point in the other. Produces chain-shaped clusters. Sensitive to outliers.

Complete linkage: Distance between two clusters = maximum distance between any point in one cluster and any point in the other. Produces compact, spherical clusters. Conservative.

Average linkage: Distance between two clusters = average of all pairwise distances between points in the two clusters. A compromise between single and complete.

Ward linkage: Distance between two clusters = the increase in total within-cluster variance that would result from merging them. Produces balanced, similarly sized clusters. The most popular choice and the default recommendation.

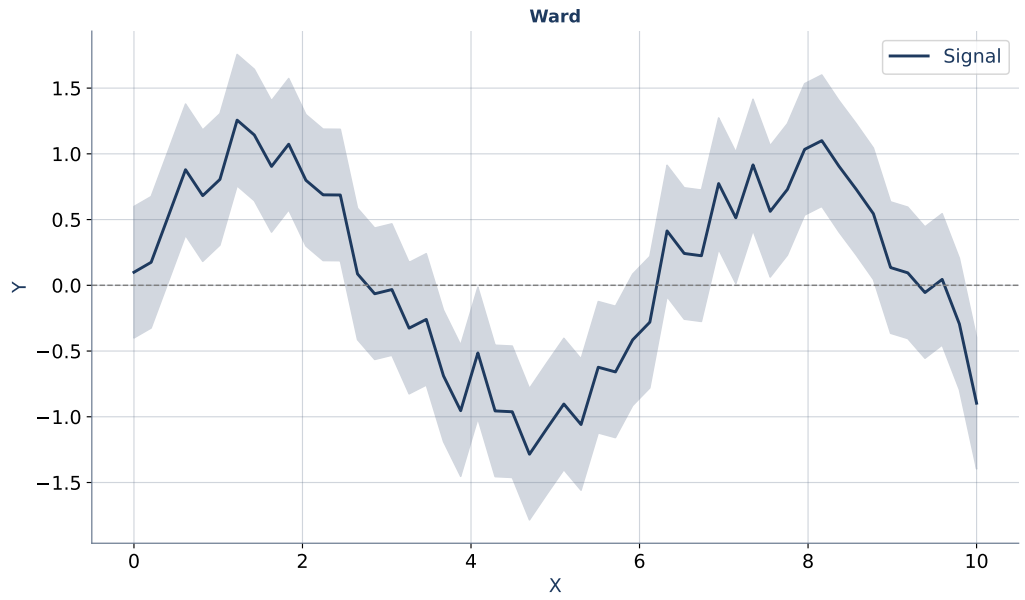


Figure 26: Ward linkage: merges the pair of clusters that increases total within-cluster variance the least. Produces compact, balanced clusters.

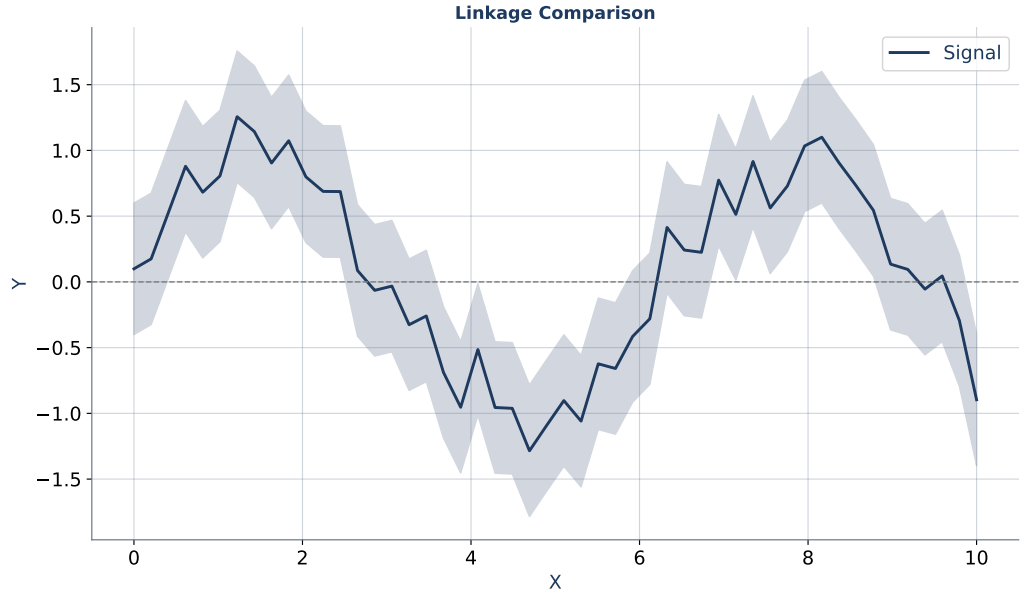


Figure 27: Four linkage methods on the same data. Single linkage chains. Complete linkage makes compact blobs. Ward makes balanced groups. Average is a middle ground.

Key Formula: Ward Linkage Distance

When merging clusters A and B with centroids μ_A and μ_B and sizes n_A and n_B, the Ward distance is:

$$d_{\text{Ward}}(A, B) = \sqrt{\frac{2 \cdot n_A \cdot n_B}{n_A + n_B}} \cdot \|\mu_A - \mu_B\|$$

where:

- n_A, n_B are the number of points in clusters A and B
- μ_A, μ_B are the centroids (means) of clusters A and B
- $\|\mu_A - \mu_B\|$ is the Euclidean distance between centroids

Plain English: Ward penalizes merges that would create large, spread-out clusters. It prefers merging small, close clusters over large, distant ones.

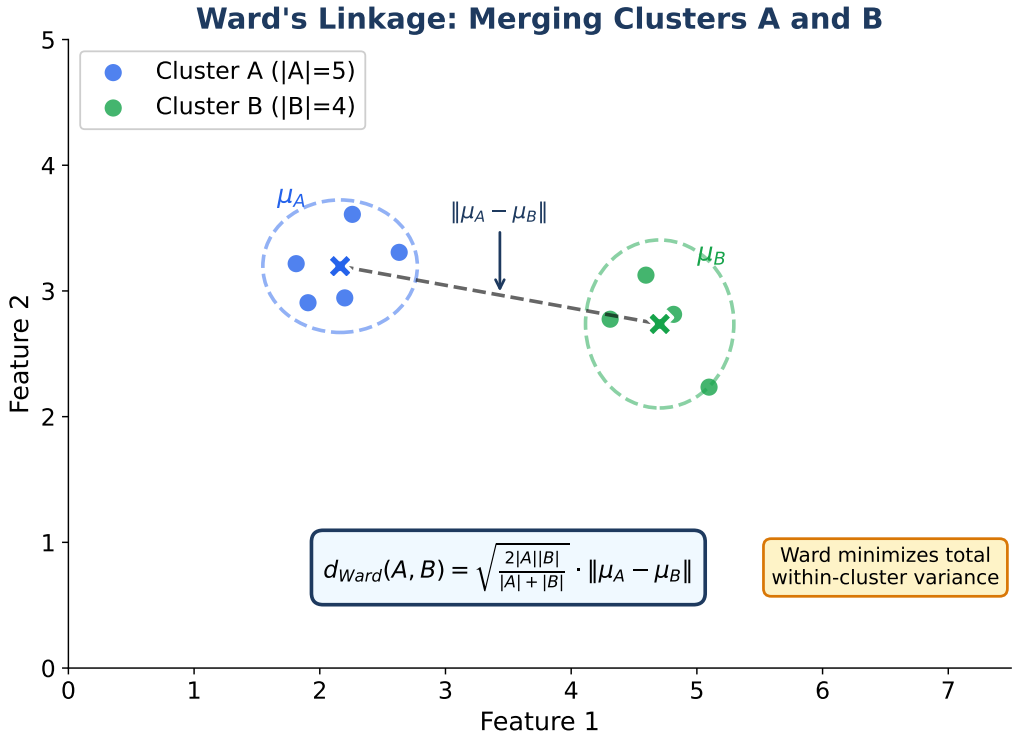


Figure 28: Visual representation of the Ward formula: the merge that increases total variance the least wins.

Reading a Dendrogram

The dendrogram is the output of hierarchical clustering: a tree diagram that records every merge.

Dendrogram: A tree diagram where each leaf is one observation, each internal node represents a merge, and the height of each node indicates the distance at which the merge occurred. Tall vertical bars mean the merged clusters were far apart; short bars mean they were close.

To get flat clusters from a dendrogram, draw a horizontal line at some height h. Count how many branches the line crosses. That is your number of clusters. The trick to reading a dendrogram well: look for large vertical gaps between successive merges. A large gap means the algorithm had to stretch far to find the next closest pair—a sign that there is a natural break in

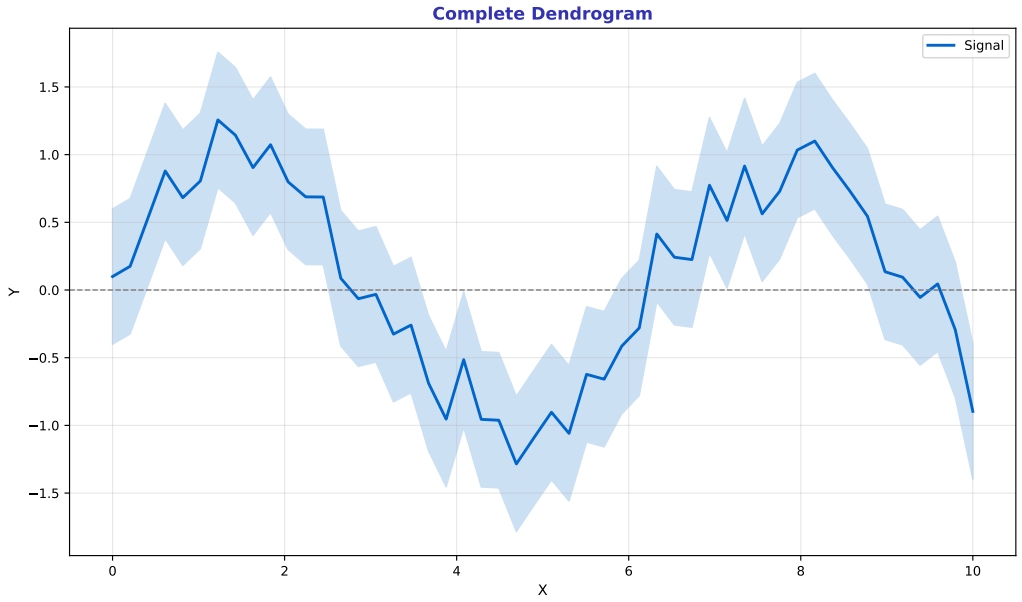


Figure 29: A complete dendrogram. Read bottom-up: items that merge early (low) are very similar. Items that merge late (high) are very different.

the data. Place your cut line inside the largest gap, and you get the most "natural" number of clusters.

Three different cuts on the same dendrogram produce three different partitions, each valid at its own level of granularity.

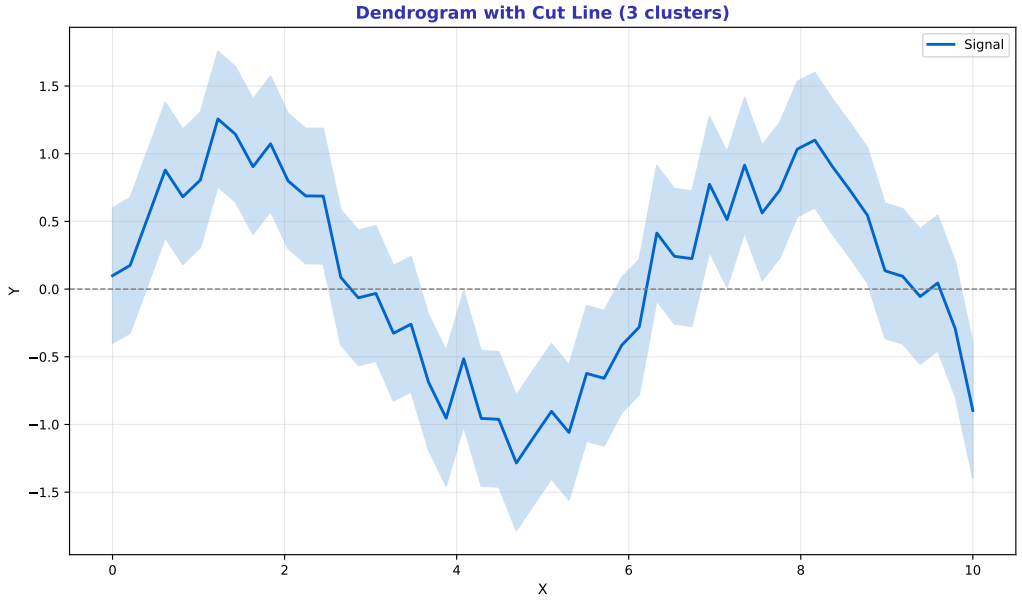


Figure 30: Cutting the dendrogram: a horizontal line at height h produces a flat partition. Lower cuts give more clusters; higher cuts give fewer.

The following table summarizes the trade-offs between K-Means and hierarchical clustering:

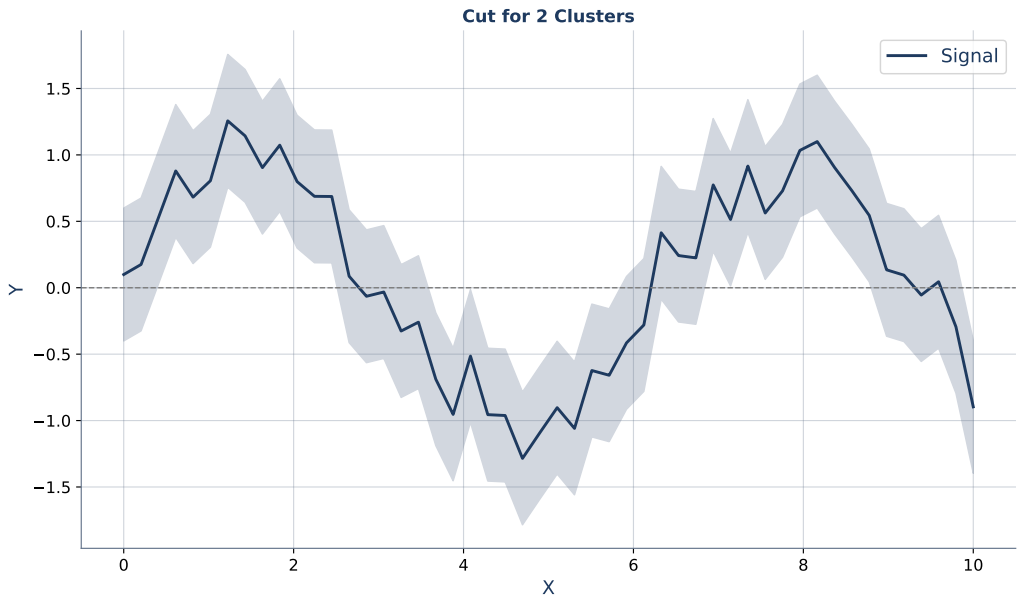


Figure 31: Cutting high: two broad clusters.

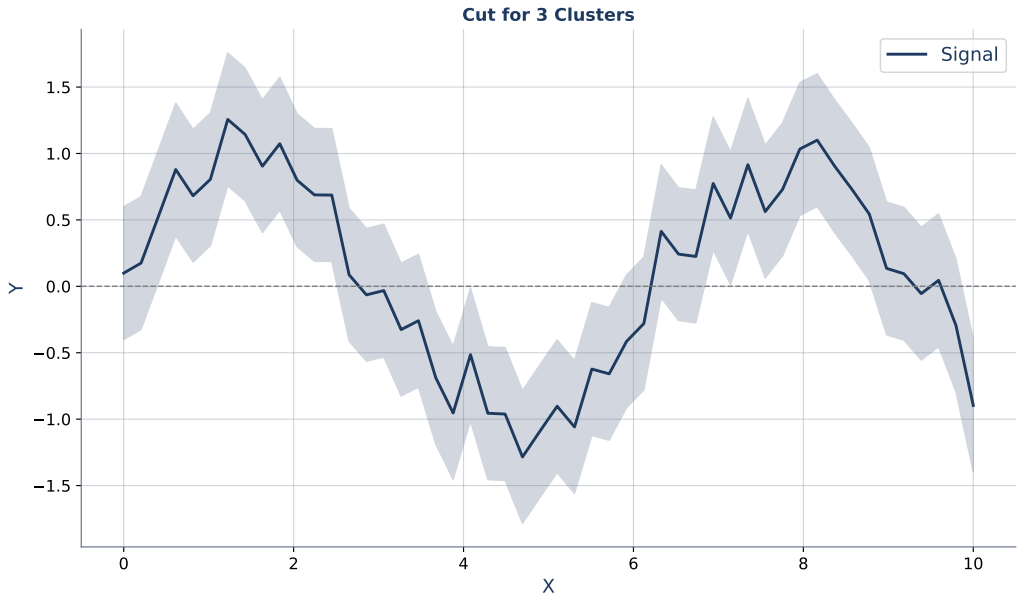


Figure 32: Cutting mid-height: three clusters.

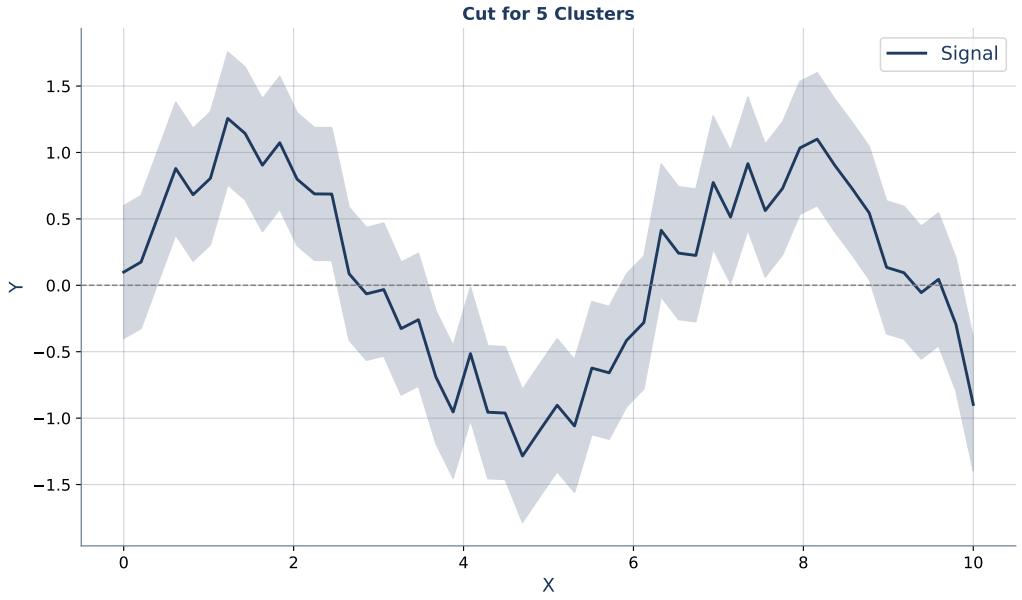


Figure 33: Cutting low: five fine-grained clusters.

Criterion	K-Means	Hierarchical
Must choose K first?	Yes	No (cut dendrogram later)
Speed	$O(nKt)$ — fast	$O(n^3)$ — slow
Cluster shape	Spherical only	Any (depends on linkage)
Output	Flat partition	Nested tree
Inter-cluster relationships	None	Full hierarchy
Practical limit	Millions of points	∼10,000 points
Best for	Large data, known K	Small data, exploration

Definition: Agglomerative Hierarchical Clustering

Agglomerative hierarchical clustering is a bottom-up algorithm that builds a nested hierarchy of clusters:

1. Start with n clusters, each containing one observation.
2. Compute the distance between every pair of clusters using the chosen linkage criterion.
3. Merge the two closest clusters into one.
4. Repeat steps 2–3 until only one cluster remains.

The result is a dendrogram that encodes all $n - 1$ merges. Cutting the dendrogram at height h produces a flat partition into K clusters.

Agglomerative clustering: A bottom-up hierarchical clustering strategy that starts with every observation as its own cluster and repeatedly merges the two closest clusters until one cluster remains. The merge history is recorded in a dendrogram. Contrast with divisive (top-down) clustering, which starts with one cluster and splits.

Common Misconceptions about Hierarchical Clustering

(1) "Hierarchical clustering always beats K-Means." It is $O(n^3)$ in both time and memory. For datasets larger than about 10,000 points, it becomes impractical. K-Means scales to millions.
(2) "The dendrogram gives the 'correct' number of clusters." The dendrogram shows all possible cuts. Choosing where to cut is still a judgment call—look for large vertical gaps, but there is no universal rule.
(3) "Single linkage is a safe default." Single linkage produces chain-shaped clusters (the "chaining effect") that are rarely useful in practice. Ward linkage is a much better default for most applications.

Worked Examples

Worked Example 1: One Step of Single Linkage by Hand

Five points with the following distance matrix:

	A	B	C	D	E
A	0	2	6	10	9
B	2	0	5	9	8
C	6	5	0	4	5
D	10	9	4	0	3
E	9	8	5	3	0

Step 1: Smallest entry is $d(A, B) = 2$. Merge A and B into cluster $\{A, B\}$.
Update distances (single linkage): $d(\{A, B\}, C) = \min(d(A, C), d(B, C)) = \min(6, 5) = 5$. $d(\{A, B\}, D) = \min(d(A, D), d(B, D)) = \min(10, 9) = 9$. $d(\{A, B\}, E) = \min(d(A, E), d(B, E)) = \min(9, 8) = 8$.
New distance matrix:

	{A,B}	C	D	E
{A,B}	0	5	9	8
C	5	0	4	5
D	9	4	0	3
E	8	5	3	0

Next merge: $d(D, E) = 3$. The dendrogram records merge $\{A, B\}$ at height 2 and merge $\{D, E\}$ at height 3.

Worked Example 2: Choosing Linkage for Bond Portfolios

A risk manager clusters 20 corporate bonds by return correlation. She tries three linkage methods:

Single linkage: Produces one large chain and several singletons. The "chaining effect" merges almost everything into one cluster early on. Not useful.

Complete linkage: Produces compact clusters, but some bonds that clearly belong together end up in different clusters because one outlier pair has a large distance.

Ward linkage: Produces balanced clusters that align well with industry sectors. The dendrogram shows a clear gap between 3 and 4 clusters, suggesting a natural partition into three groups.

Recommendation: Start with Ward. Use single linkage only when you suspect chain-shaped or elongated cluster structures.

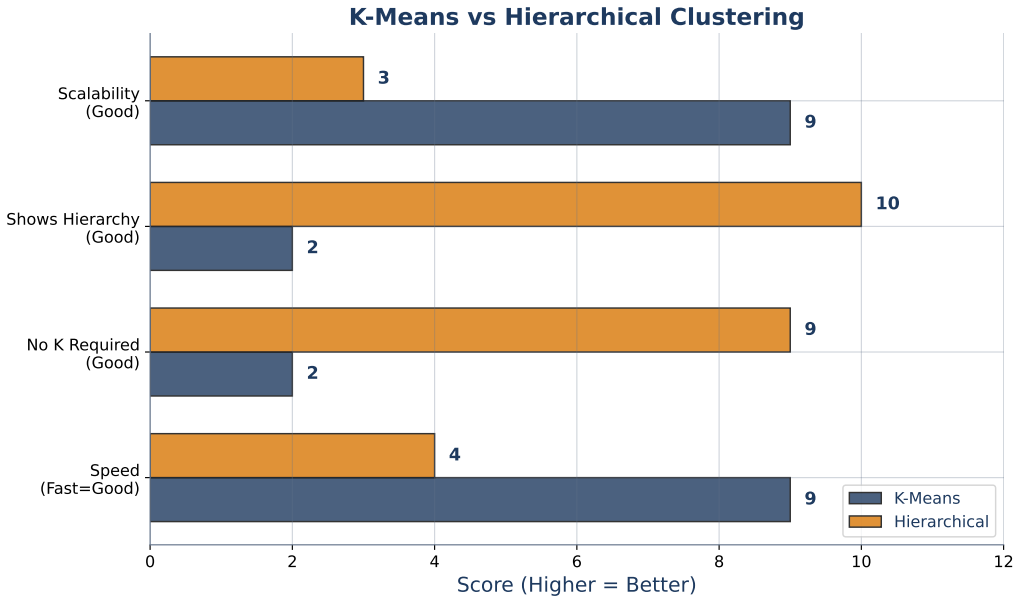


Figure 34: K-Means gives a flat partition with no inter-cluster relationships. Hierarchical clustering gives a full tree structure. Trade-off: speed vs. richness.

Historical Background: Stephen Johnson and the First Hierarchical Algorithms (1967)

In 1967, Stephen C. Johnson published a paper titled "Hierarchical Clustering Schemes" that gave the first computationally efficient algorithms for single-linkage and complete-linkage hierarchical clustering. Before Johnson's work, the idea of building cluster hierarchies existed in concept but lacked practical algorithms that could handle real datasets. Johnson's contribution was turning the intuitive idea of "merge the closest pair" into a precise, repeatable procedure with known computational complexity. His algorithms ran in $O(n^2)$ time for single linkage—a dramatic improvement over the naive $O(n^3)$ approach. Johnson's algorithms turned the intuitive idea of building a family tree into a precise, repeatable procedure. Today, `scipy.cluster.hierarchy.linkage(X, method='ward')` runs the entire hierarchy in one line.

Problem 3.1 (Easy)

Look at the dendrogram in Figure 30. If you draw a horizontal line at height $h = 5$, how many clusters do you get? What about $h = 10$?
Solution: see Appendix.

Problem 3.2 (Easy)

Name the four common linkage methods (single, complete, average, Ward) and describe each in one sentence. Which is the recommended default?
Solution: see Appendix.

Problem 3.3 (Medium)

Given the distance matrix for five points below, perform one step of agglomerative clustering with single linkage. Which pair merges first? Write the updated distance matrix.

	P	Q	R	S	T
P	0	3	7	11	9
Q	3	0	4	8	7
R	7	4	0	2	6
S	11	8	2	0	5
T	9	7	6	5	0

Solution: see Appendix.

Problem 3.4 (Medium)

Compare the dendrograms produced by single linkage and Ward linkage on the same dataset. Explain why single linkage tends to produce one large cluster that absorbs most points (the "chaining effect"), while Ward produces more balanced clusters.
Solution: see Appendix.

Problem 3.5 (Hard)

Derive the Ward linkage distance formula. Start from the definition: the Ward distance between clusters A and B is the increase in total within-cluster sum of squares caused by merging them. Show that this equals $\frac{n_A \cdot n_B}{n_A + n_B} \cdot \|\mu_A - \mu_B\|^2$, where n_A, n_B are cluster sizes and μ_A, μ_B are centroids.
(*Hint:* Compute the WCSS of the merged cluster $A \cup B$ and subtract the sum of WCSS for A and B separately. Use the identity that the sum of squared deviations from the mean can be decomposed into within-group and between-group components.)
Solution: see Appendix.

Connecting Forward

We can now build a complete tree of nested groupings and cut it at any level. The dendrogram shows us relationships between clusters that K-Means cannot. But we have not yet used hierarchical clustering for its most powerful financial application: portfolio construction.

Section 4 combines three ideas. First, we use *correlation* as the distance metric—because in finance, two stocks that move together are "close" even if their raw feature values are far

apart. Second, we use the dendrogram to reorder the correlation matrix, revealing block-diagonal structure that is invisible in the raw matrix. Third, we introduce Hierarchical Risk Parity (HRP), a portfolio construction method that uses the dendrogram to allocate risk across assets. HRP does not invert the covariance matrix (which is notoriously unstable), and it produces weights that are more robust out-of-sample than Markowitz optimization.

Key Takeaway: Hierarchical clustering builds a complete tree of nested groupings, letting you choose the number of clusters after seeing the full dendrogram rather than guessing K upfront.

4. Cutting the Tree – Linkage Methods, Correlation Clustering, and HRP

Opening Problem: The Broken Markowitz Portfolio

A quantitative analyst at a pension fund builds a Markowitz mean-variance portfolio for 30 stocks. She estimates expected returns and the 30×30 covariance matrix from three years of daily data. She feeds both into the optimizer and asks for the minimum-variance portfolio.

The result is absurd. The optimizer puts 80% of the portfolio into a single stock and short-sells three others. She adds constraints: no short selling, maximum 10% per stock. The weights change completely. She re-estimates the covariance matrix with one extra month of data. The weights change again—dramatically. The portfolio is mathematically optimal but practically useless.

A colleague says: "Stop inverting the covariance matrix. Use a dendrogram instead." She is skeptical. Dendrograms are for biologists classifying species, not for building portfolios. But the colleague points to a 2016 paper by Marcos Lopez de Prado that shows dendrogram-based portfolios outperform Markowitz out-of-sample. That method is Hierarchical Risk Parity.

Discovery Question

Traditional portfolio optimization (Markowitz) inverts a covariance matrix and often produces extreme, unstable weights. A finance professor claims that a dendrogram—a tool from biology—builds better portfolios. How can a clustering algorithm know anything about investing?

From Returns to Distances

In Sections 2 and 3, we clustered data points in feature space using Euclidean distance. In finance, we care less about how "far apart" two stocks are in feature space and more about how they *move together*. Two stocks might have very different prices, market caps, and P/E ratios, but if their daily returns correlate at 0.95, they are functionally the same from a risk perspective.

The correlation matrix captures this. Each entry ρ_{ij} tells you how strongly stocks i and j move together. A correlation of 0.95 between Apple and Microsoft means their daily returns almost always move in the same direction. A correlation of -0.30 between gold and the S&P 500 means they tend to move in opposite directions.

But correlation is not a distance. A proper distance metric must satisfy three axioms: $d(i,i) = 0$ (distance to yourself is zero), $d(i,j) = d(j,i)$ (symmetry), and $d(i,k) \leq d(i,j) + d(j,k)$ (triangle inequality). Raw correlation violates the first axiom: $\rho_{ii} = 1$, not 0. We need to convert.

Correlation distance: A transformation that converts correlation ρ into a proper distance metric. The simplest version is $d = 1 - |\rho|$, where $|\rho|$ is the absolute correlation. Stocks with $|\rho| = 1$ (perfect correlation or anti-correlation) have distance 0. Stocks with $\rho = 0$ (uncorrelated) have distance 1.

Now comes the trick that makes hierarchical clustering so valuable for finance. Reorder the rows and columns of the correlation matrix according to the dendrogram's leaf order—put assets that merge early next to each other. Suddenly, the hidden block-diagonal structure jumps out. Bright squares along the diagonal show groups of correlated assets. Dark off-diagonal regions show uncorrelated groups. The structure was always there in the data. The original row ordering just hid it.

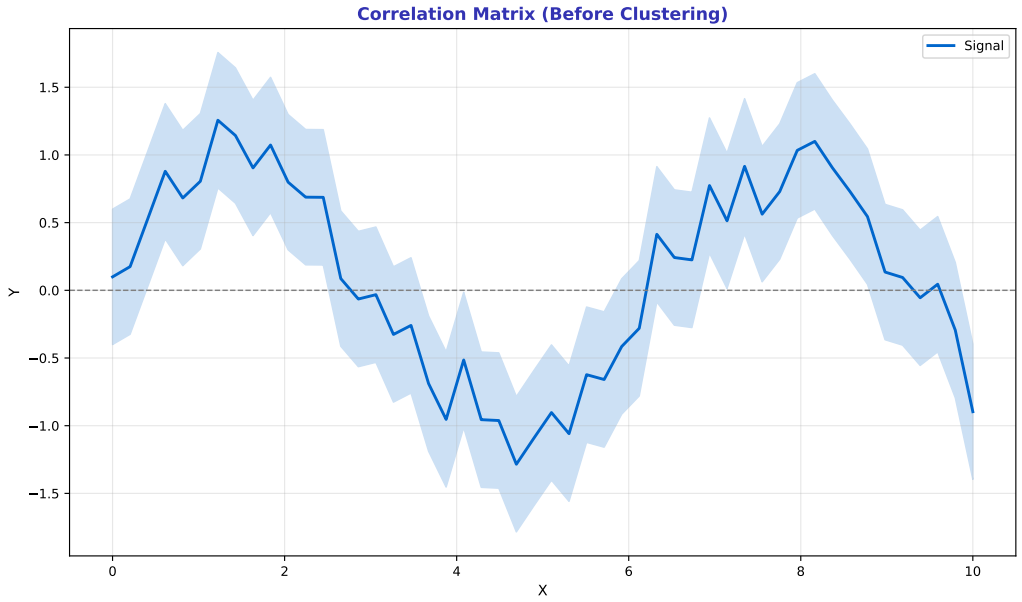


Figure 35: Raw correlation matrix for a set of financial assets. Can you spot the cluster structure? It is hidden in the random ordering of rows and columns.

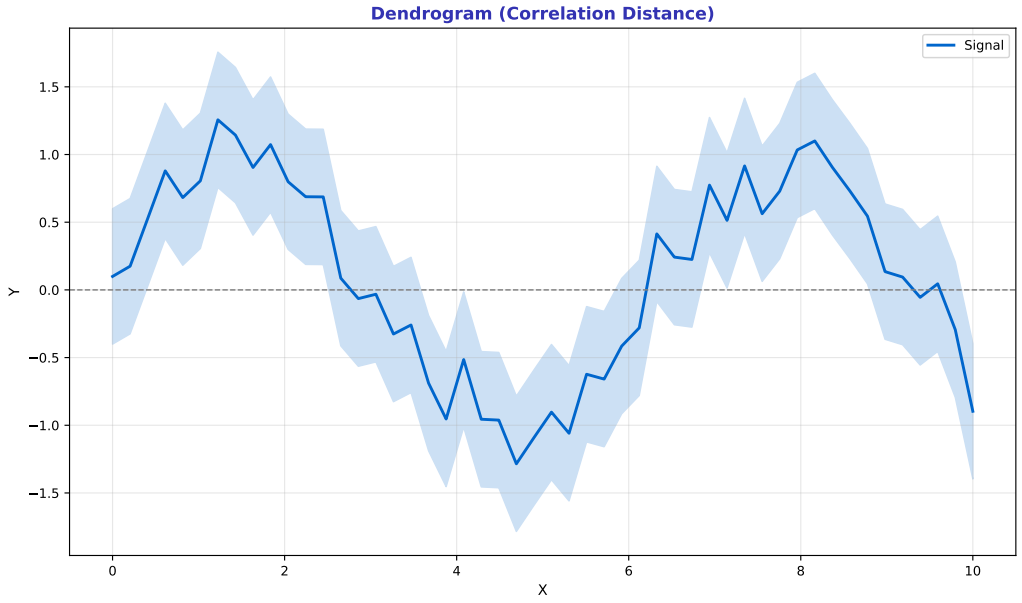


Figure 36: Dendrogram built from the correlation distance matrix. Assets that move together merge early (low height). The tree reveals hidden sector structure.

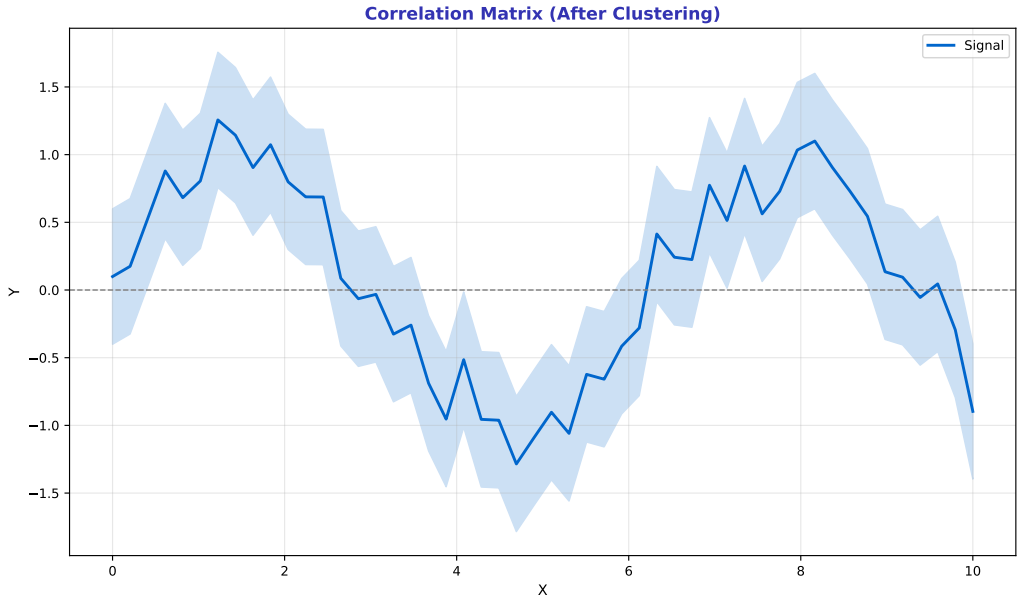


Figure 37: Same correlation matrix, reordered by dendrogram. Blocks along the diagonal reveal natural asset groups. This structure was invisible before reordering.

Choosing the Right Linkage

Section 3 introduced four linkage methods. In practice, the choice depends on the shape of your data and what you want from the clusters.

Linkage Method Selection Guide

Linkage	Formula	Best For	Caution
Ward	$\min \Delta\sigma^2$	Compact, spherical equal-sized clusters	Assumes spherical
Complete	$\max\{d(a,b)\}$	Tight clusters outlier sensitive	May miss elongated
Average	$\frac{1}{nm}\sum d$	Balanced approach robust	Slower convergence
Single	$\min\{d(a,b)\}$	Elongated clusters chains	Chaining effect

Default: Use Ward for most applications. Use Single only for elongated/chain-like data.

Figure 38: Linkage decision matrix: which method to use depending on cluster shape expectations and dataset size.

Cophenetic correlation: A measure of how faithfully the dendrogram preserves the original pairwise distances. It is the Pearson correlation between the original distance matrix and the "cophenetic" distances (the height at which each pair of observations first merges in the dendrogram). Values close to 1.0 mean the dendrogram accurately represents the data's distance structure.

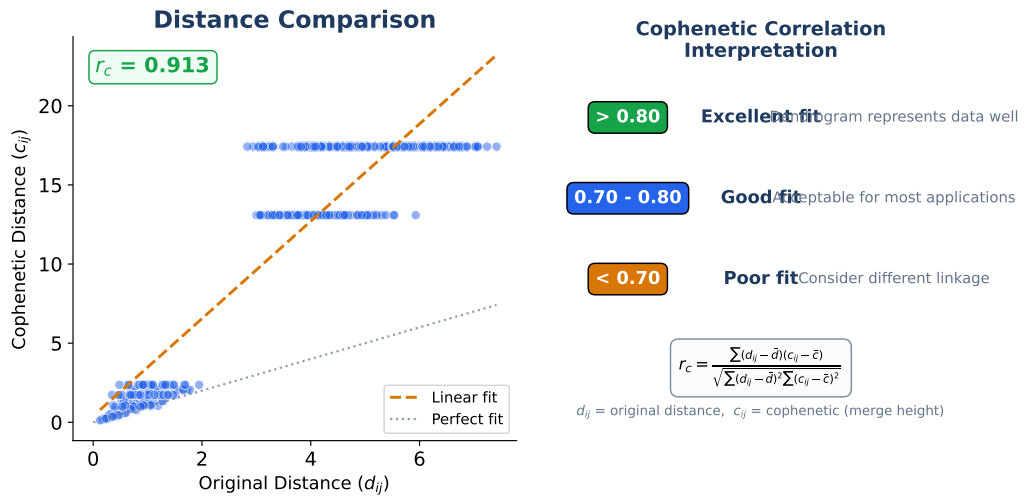


Figure 39: Cophenetic correlation: how well does the dendrogram preserve the original distances? Higher is better. Average linkage often scores highest.

Key Formula: Correlation Distance

To cluster financial assets by return correlation, convert the $n \times n$ correlation matrix ρ into a distance matrix D:

$$d_{ij} = \sqrt{2(1 - \rho_{ij})}$$

where:

- ρ_{ij} is the Pearson correlation between the daily returns of assets i and j
- $d_{ij} = 0$ when $\rho_{ij} = 1$ (perfectly correlated)
- $d_{ij} = \sqrt{2} \approx 1.41$ when $\rho_{ij} = 0$ (uncorrelated)
- $d_{ij} = 2$ when $\rho_{ij} = -1$ (perfectly anti-correlated)

An alternative, simpler version: $d_{ij} = 1 - |\rho_{ij}|$, which treats positive and negative correlation symmetrically.

When K-Means Fails: Non-Spherical Clusters

Before diving into HRP, let us revisit a limitation of K-Means that hierarchical clustering can overcome. K-Means assigns every point to the nearest centroid using Euclidean distance. This creates Voronoi cells—convex polygons around each centroid. The problem is obvious once you state it: convex polygons can only capture convex shapes. If your clusters are crescent-shaped, ring-shaped, or elongated, the Voronoi boundaries will slice them apart.

Hierarchical clustering with single linkage does not have this problem. Single linkage merges the pair of points (or clusters) with the smallest minimum distance. It can follow a chain of nearby points along a curve, capturing elongated and non-convex structures that K-Means cannot touch.

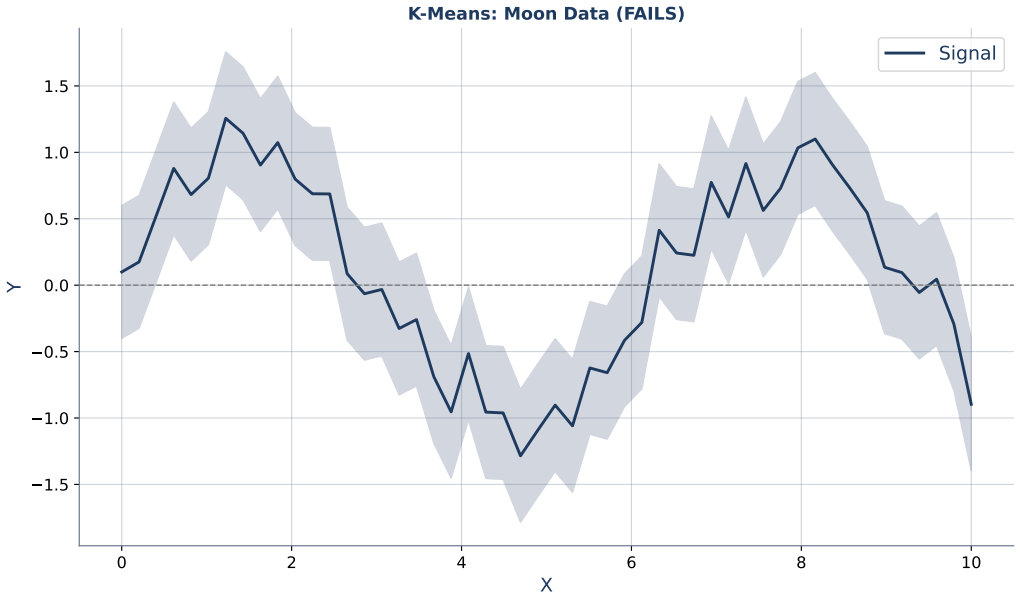


Figure 40: K-Means on crescent ("moon") data. The centroids land between the two crescents, splitting each moon in half. The algorithm sees spheres where there are none.

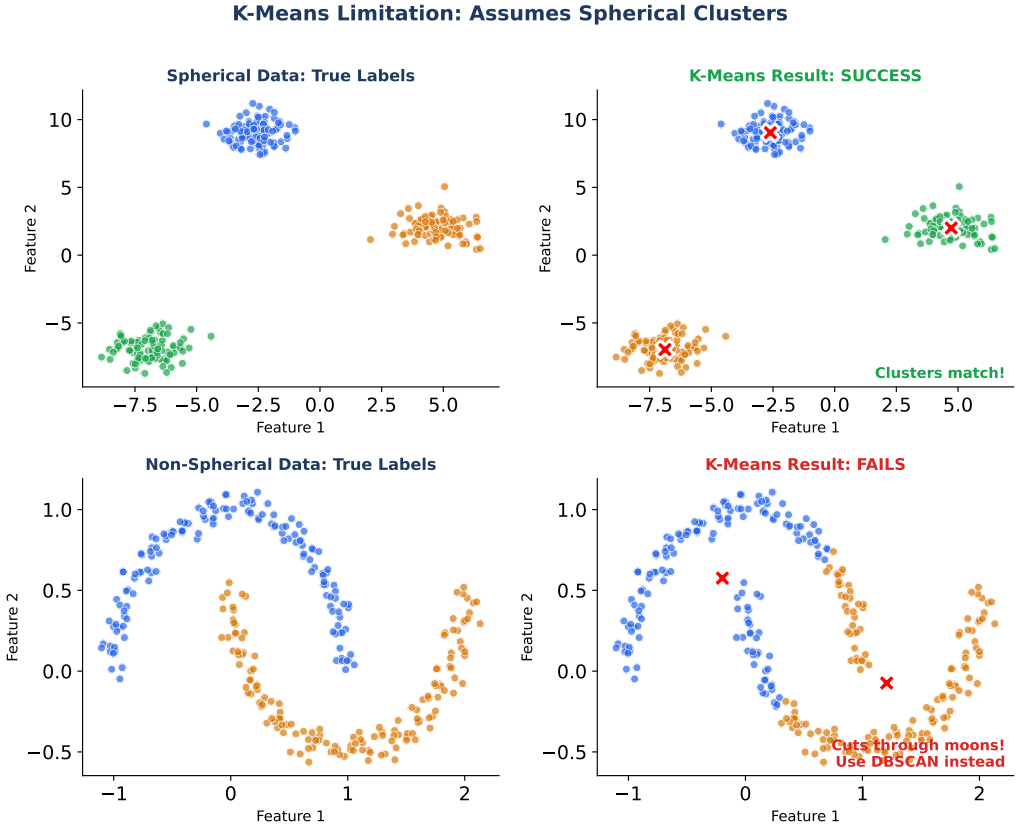


Figure 41: K-Means fails on non-spherical shapes. Hierarchical clustering with single linkage can follow the chain structure and recover the true groups.

Definition: Hierarchical Risk Parity (HRP)

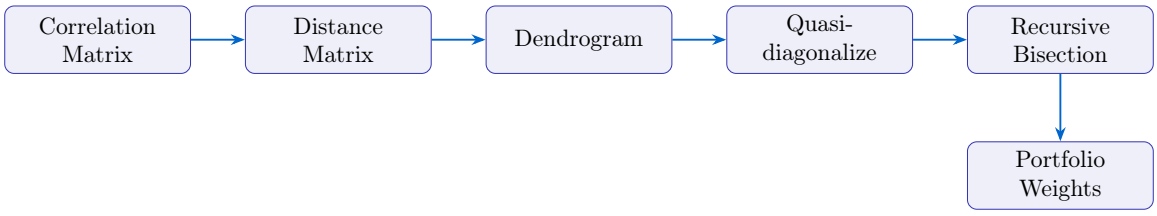
HRP is a portfolio construction method (Lopez de Prado, 2016) that uses hierarchical clustering to allocate risk across assets. It avoids inverting the covariance matrix and produces weights that are more stable out-of-sample than Markowitz optimization. HRP has three steps:

1. **Tree clustering:** Build a dendrogram from the correlation distance matrix using Ward or single linkage.
2. **Quasi-diagonalization:** Reorder the covariance matrix according to the dendrogram's leaf order so that correlated assets sit adjacent.
3. **Recursive bisection:** Split the sorted assets into two halves at the top of the dendrogram. Allocate weight inversely proportional to each half's variance. Recurse into each half and repeat.

The result: low-variance clusters get more weight, high-variance clusters get less. No matrix inversion. No optimization. Just the tree.
The key advantage is stability. Markowitz optimization inverts the covariance matrix, and matrix inversion amplifies small estimation errors into large weight swings. HRP never inverts. It only uses the covariance matrix to compute variances and build the dendrogram—operations that are far less sensitive to estimation noise.

Common Misconceptions about Correlation Clustering and HRP

(1) "Correlation = similarity for clustering." Correlation is not a distance metric. You must convert it: $d = 1 - |\rho|$ or $d = \sqrt{2(1-\rho)}$. Raw correlations fed into a distance-based algorithm will produce nonsense.
(2) "HRP replaces all portfolio optimization." HRP is one tool in the toolbox, not a silver bullet. It works well when the covariance matrix is noisy and unstable (typical for large universes with short estimation windows). For small, stable universes, Markowitz can still be effective.
(3) "Ward linkage is always best." Ward assumes roughly spherical clusters. For assets with elongated correlation structures (commodities with seasonal patterns, for instance), average linkage may preserve distances better, as measured by the cophenetic correlation.



The HRP pipeline: start with correlations, convert to distances, build a tree, reorder the covariance matrix, then allocate risk top-down by recursive bisection. No matrix inversion.

Worked Examples

Worked Example 1: Converting Correlations to Distances

Three assets have the following correlation matrix:

	A	B	C
A	1.00	0.80	0.20
B	0.80	1.00	0.10
C	0.20	0.10	1.00

Using $d = 1 - |\rho|$:

- $d(A, B) = 1 - 0.80 = 0.20$. Very close—they move together.
- $d(A, C) = 1 - 0.20 = 0.80$. Distant—weakly correlated.
- $d(B, C) = 1 - 0.10 = 0.90$. Very distant—nearly uncorrelated.

First merge: A and B at distance 0.20. Then merge $\{A, B\}$ and C at a higher distance. The dendrogram correctly shows that A and B are tightly linked while C is an outsider.

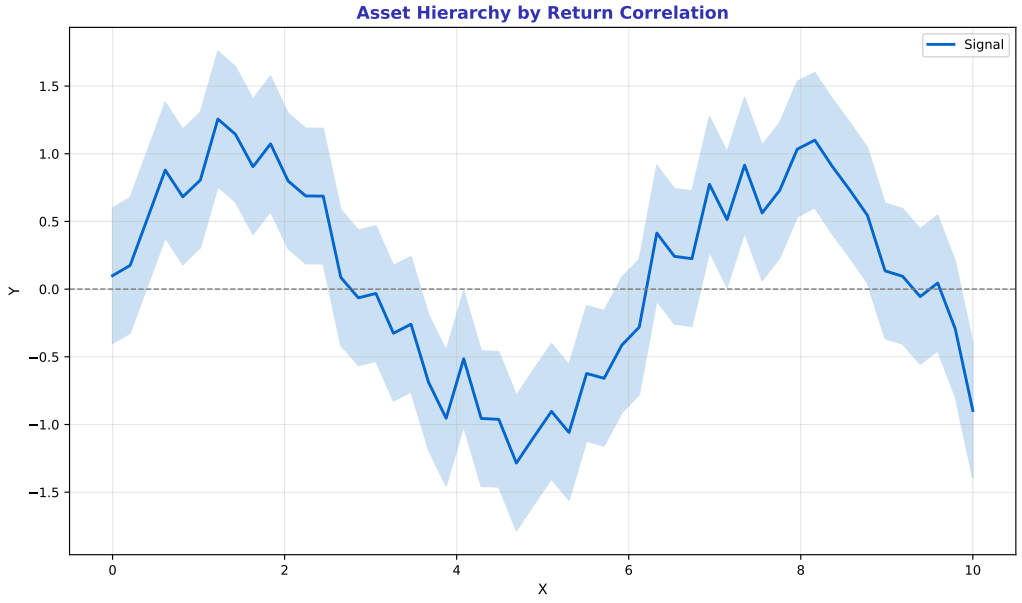


Figure 42: Asset hierarchy built from correlation distances. Assets that move together cluster early. The tree reveals sector-like structure without using sector labels.

Worked Example 2: Simplified HRP for Three Assets

Three assets: A (variance $= 0.04$), B (variance $= 0.09$), C (variance $= 0.01$). The dendrogram merges A and B first (they are correlated), leaving C separate.
Step 1 – Recursive bisection at the top: Split into $\{A, B\}$ and $\{C\}$.

- Cluster variance of $\{A, B\}$: use inverse-variance weighting within the cluster, giving $w_A = \frac{1/0.04}{1/0.04+1/0.09} = \frac{25}{25+11.1} = 0.69$, $w_B = 0.31$. Combined cluster variance $\approx 0.04 \cdot 0.69^2 + 0.09 \cdot 0.31^2 \approx 0.028$ (ignoring correlation for simplicity).
- Cluster variance of $\{C\}$: 0.01.

Step 2 – Allocate between clusters: Inverse-variance: $w_{\{A,B\}} = \frac{1/0.028}{1/0.028+1/0.01} = \frac{35.7}{35.7+100} = 0.26$, $w_C = 0.74$.
Step 3 – Distribute within cluster: $w_A = 0.26 \times 0.69 = 0.18$, $w_B = 0.26 \times 0.31 = 0.08$.
Final HRP weights: A: 18%, B: 8%, C: 74%. The low-variance asset C gets the most weight. No matrix inversion required.

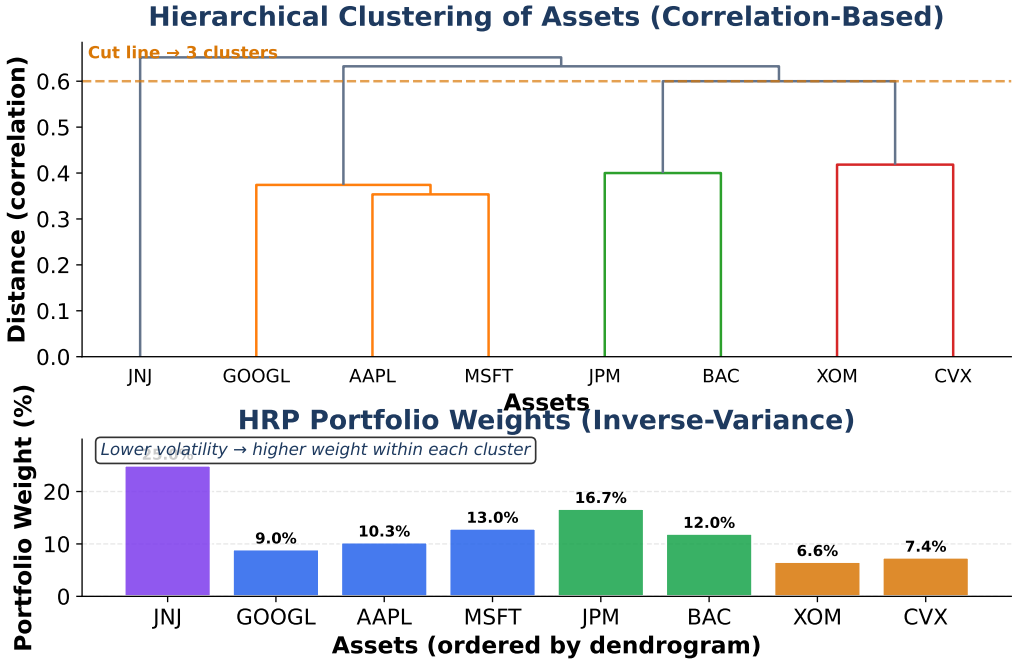


Figure 43: HRP portfolio allocation. Weights are determined by the tree structure and cluster variances, not by covariance matrix inversion.

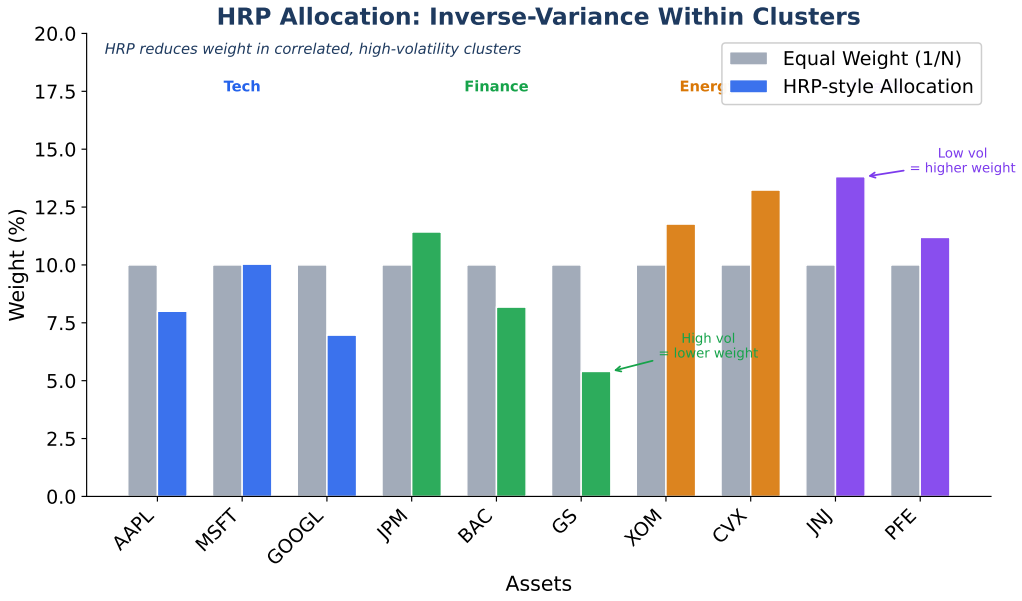


Figure 44: HRP weight allocation: low-variance clusters receive more weight. The dendrogram determines how assets are grouped.

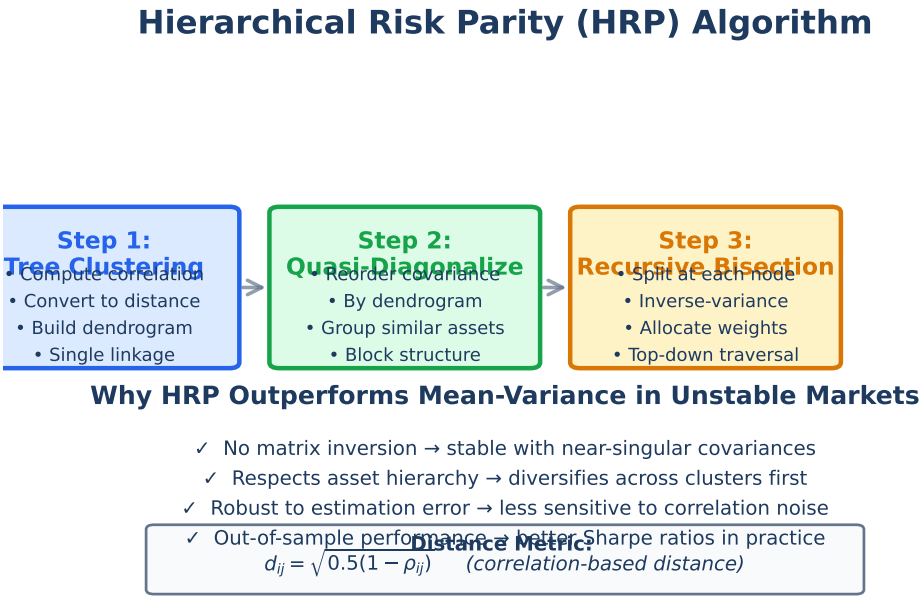


Figure 45: The three steps of HRP: tree clustering, quasi-diagonalization, and recursive bisection.

Historical Background: Marcos Lopez de Prado and HRP (2016)

In 2016, Marcos Lopez de Prado published "Building Diversified Portfolios that Outperform Out-of-Sample" in the *Journal of Portfolio Management*. The paper introduced Hierarchical Risk Parity (HRP), a portfolio construction method that replaced the unstable covariance matrix inversion in Markowitz optimization with a dendrogram-based recursive bisection.

Lopez de Prado's key insight: traditional optimizers amplify estimation error because they invert a noisy matrix. Small errors in correlation estimates produce large swings in portfolio weights. HRP avoids inversion entirely. It uses the hierarchical structure of asset correlations to allocate risk top-down, producing weights that are more stable when the covariance matrix changes.

The paper showed that HRP portfolios consistently outperformed equal-weight and Markowitz portfolios on out-of-sample data. The result was striking because HRP is simpler—not more complex—than the method it replaces. Dendrograms are not just for biology. They produce portfolios that outperform Markowitz optimization on out-of-sample data.

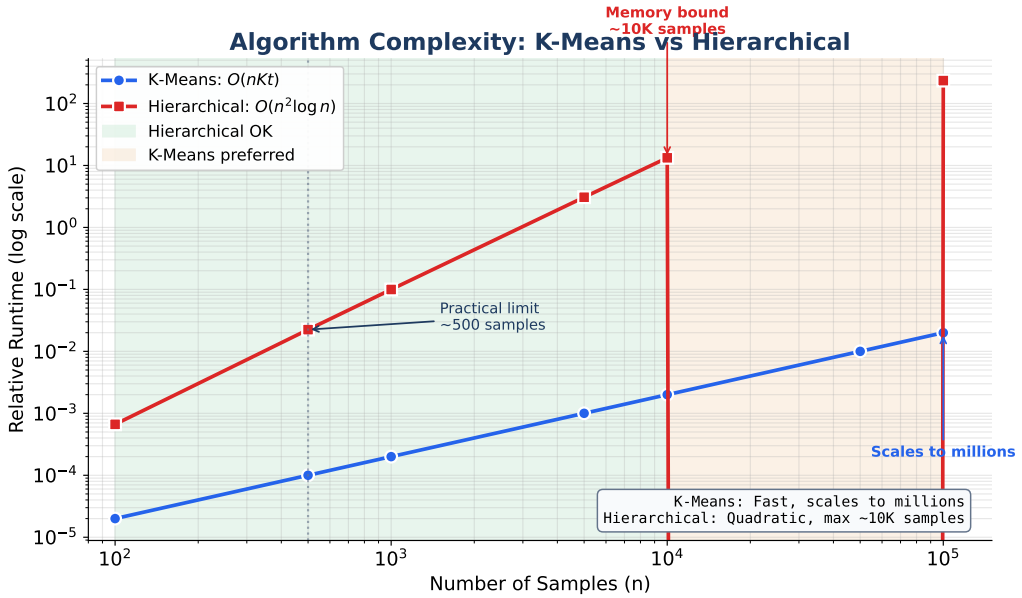


Figure 46: Computational complexity comparison. K-Means is $O(nKt)$, hierarchical is $O(n^3)$, HRP adds $O(n^2)$ for the bisection step. For typical asset universes ($n < 500$), all are fast enough.

Problem 4.1 (Easy)

Convert the following correlation matrix to a distance matrix using $d = 1 - |\rho|$:

	X	Y	Z
X	1.00	−0.60	0.30
Y	−0.60	1.00	0.50
Z	0.30	0.50	1.00

Which pair of assets is "closest"?

Solution: see Appendix.

Problem 4.2 (Easy)

Given the asset hierarchy in Figure 42, identify the two assets that merge first (lowest branch). What does this tell you about their return characteristics?
Solution: see Appendix.

Problem 4.3 (Medium)

Three assets have individual variances: $\sigma_A^2 = 0.04$, $\sigma_B^2 = 0.16$, $\sigma_C^2 = 0.01$. The dendrogram merges B and C first, with A joining later. Compute the HRP portfolio weights using inverse-variance allocation within clusters and inverse-variance allocation between clusters. Ignore correlations for simplicity.
Solution: see Appendix.

Problem 4.4 (Medium)

Explain why HRP portfolios are more stable out-of-sample than Markowitz portfolios. Specifically, identify the step in the Markowitz procedure that amplifies estimation error and explain how HRP avoids it.
Solution: see Appendix.

Problem 4.5 (Hard)

Write pseudocode for the complete HRP algorithm. Your pseudocode should cover: (1) computing the correlation distance matrix, (2) building the dendrogram with Ward linkage, (3) quasi-diagonalizing the covariance matrix, and (4) recursive bisection to compute portfolio weights. For the recursive bisection step, define the base case and the recursive case explicitly.
Solution: see Appendix.

Connecting Forward

Sections 1–4 gave us two clustering methods: K-Means for flat partitions and hierarchical clustering for tree-structured groupings. Both answer the question "which observations belong together?"

But there is a different question lurking. When your dataset has 200 features, most of them are redundant. Stock returns are driven by a handful of underlying factors—the market, interest rates, sector rotation—not by 200 independent forces. Can we compress 200 features into 5 or 10 "super-features" that capture most of the variation?

That is dimensionality reduction, and the dominant method is Principal Component Analysis (PCA). Section 5 introduces PCA as a coordinate rotation that aligns new axes with the directions of maximum variance. Section 6 shows you how to interpret the results: scree plots for choosing the number of components and loadings heatmaps for understanding what each component means.

The link between clustering and PCA is tight. You can run PCA first to reduce noise, then cluster in the reduced space—a common pipeline for high-dimensional data. Or you can cluster first and use PCA to visualize high-dimensional clusters in 2D. In finance, PCA on stock returns reveals latent factors (market, sector, momentum) that drive the correlation structure we just used for HRP. The two methods are complementary, not competing.

Key Takeaway: Correlation-based clustering reveals hidden relationships between assets, and Hierarchical Risk Parity turns dendrograms into portfolios that are more stable than Markowitz optimization.

5. Too Many Features – PCA and Dimensionality Reduction

Opening Problem: The Analyst with 200 Features and 50 Rows

You work as a quantitative analyst at a hedge fund. Your boss hands you a dataset with 200 features per stock—fundamentals, technical indicators, macro variables, sentiment scores, option-implied volatilities—and asks you to predict next-month returns. You open the file and see a problem immediately: you have 200 feature columns but only 50 rows of historical data.

Regression cannot work. With more parameters than observations, ordinary least squares has infinite solutions. Ridge regression regularizes but cannot recover information that was never there. Random forests overfit badly when the number of features dwarfs the sample size. Every algorithm you try either refuses to run or produces a model that looks perfect on training data and catastrophic out-of-sample.

The problem is not your algorithms. The problem is the data. Two hundred features sounds informative, but most of them are redundant. Price-to-earnings, price-to-book, and price-to-sales move together. Three-month and six-month momentum move together. Sector dummies encode information already present in sector-level fundamentals. Throw in correlated macro factors and you have 200 columns carrying perhaps 10 to 20 truly independent signals.

This section introduces Principal Component Analysis (PCA): a method that finds the "real" dimensions hidden inside a forest of correlated features. Instead of 200 redundant columns, you end up with 10 uncorrelated "super-features," each capturing a meaningful direction of variation. The algorithm is old—Karl Pearson invented it in 1901—but it remains the workhorse of dimensionality reduction in every field from finance to genomics.

Discovery Question

A dataset has 200 features but only 50 observations. You cannot run regression because $p > n$. A colleague says "just run PCA and reduce to 10 components." But how can you throw away 190 features and still keep the information that matters?

Rotating the Coordinate System

Here is a deceptively simple observation. Suppose you plot two features against each other—say, a company's earnings and its revenue—and the points form a cloud stretched along a diagonal line. The two features are highly correlated. Knowing one tells you a lot about the other.

If you rotate your coordinate system so that the new horizontal axis runs along the long direction of the cloud, something interesting happens. The new horizontal coordinate captures almost all the variation between companies. The new vertical coordinate captures almost nothing. You could drop the vertical coordinate entirely and still keep 95% of the information. You have just reduced two features to one—without losing anything that matters.

That is PCA in a single picture. It rotates the coordinate system to align with the directions where the data varies most. The first principal component (PC1) points along the direction of maximum variance. The second (PC2) points along the next most variable direction, with the constraint that it must be perpendicular to PC1. And so on. With 200 features you get 200 principal components, but usually only the first 10 or 20 capture meaningful variation. The rest are noise.

Think about a crowd of people walking down a long hallway. From above you see a thin cloud of dots stretched end-to-end along the hallway. The hallway direction is PC1—it captures where people are. The narrow direction (across the hallway) is PC2—it captures almost nothing

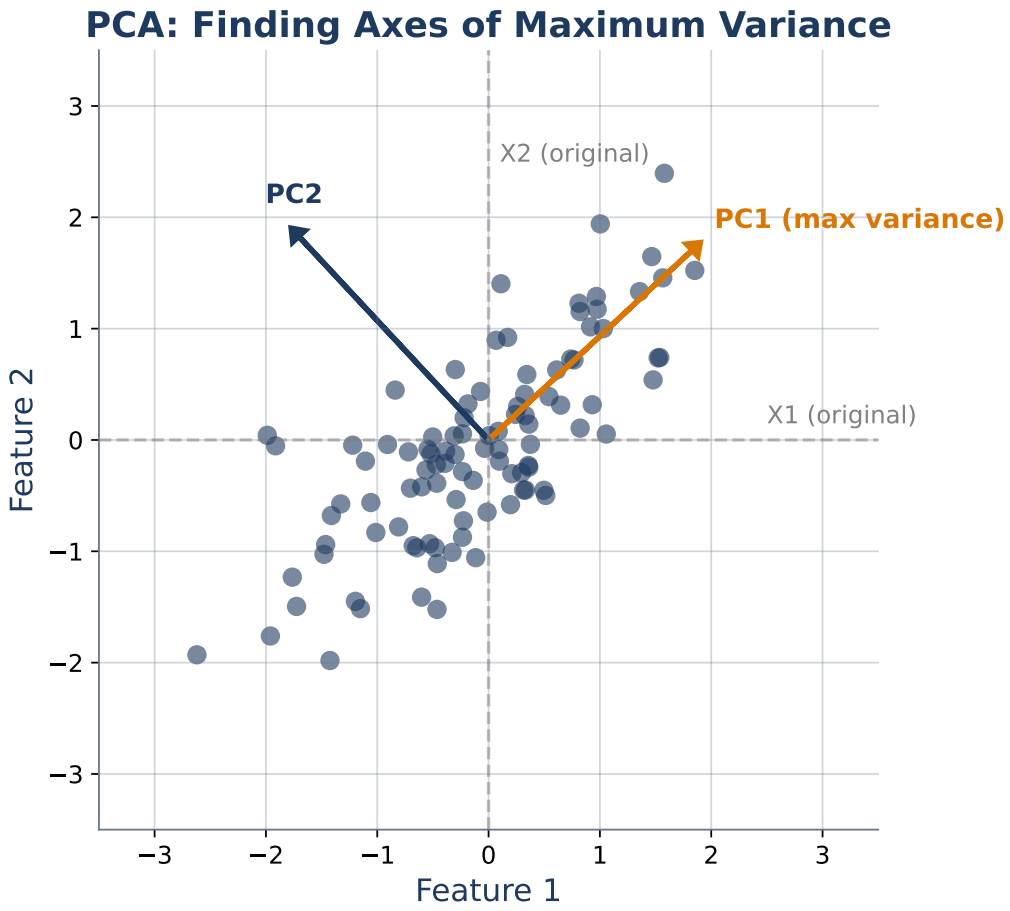


Figure 47: The core idea of PCA: a scatter of points is redescribed in a new coordinate system aligned with the directions of maximum variance.

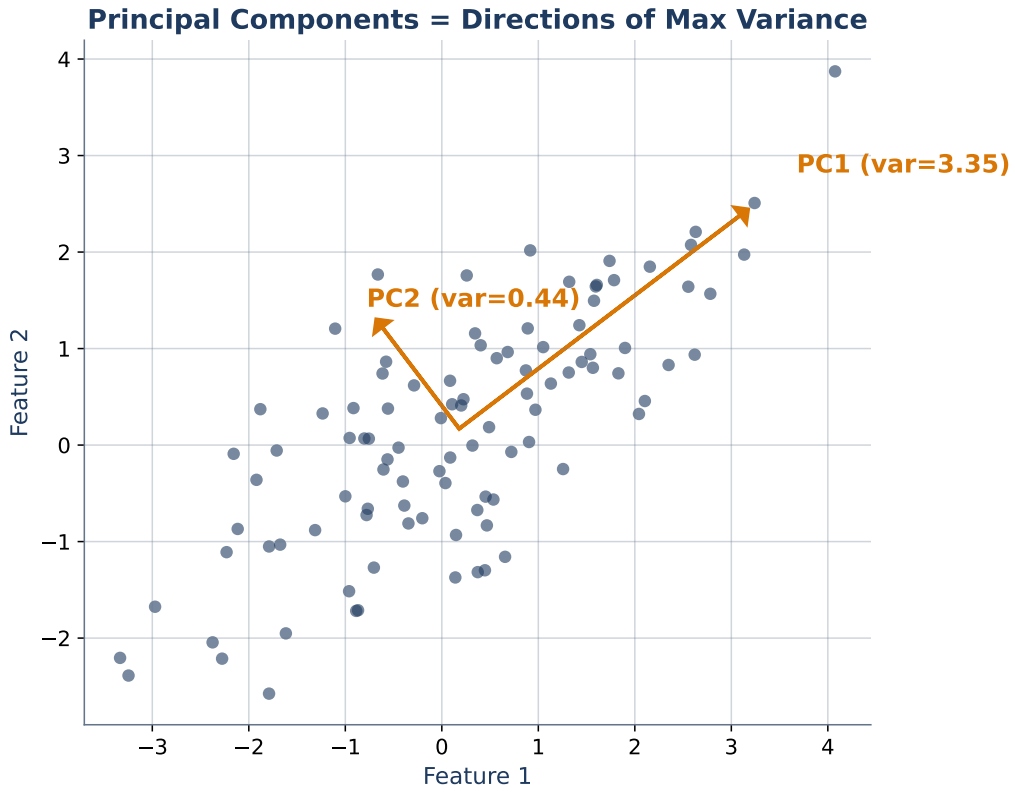


Figure 48: The first principal component (PC1) is the direction of maximum variance. The second (PC2) is perpendicular to PC1 and captures the next most variable direction.

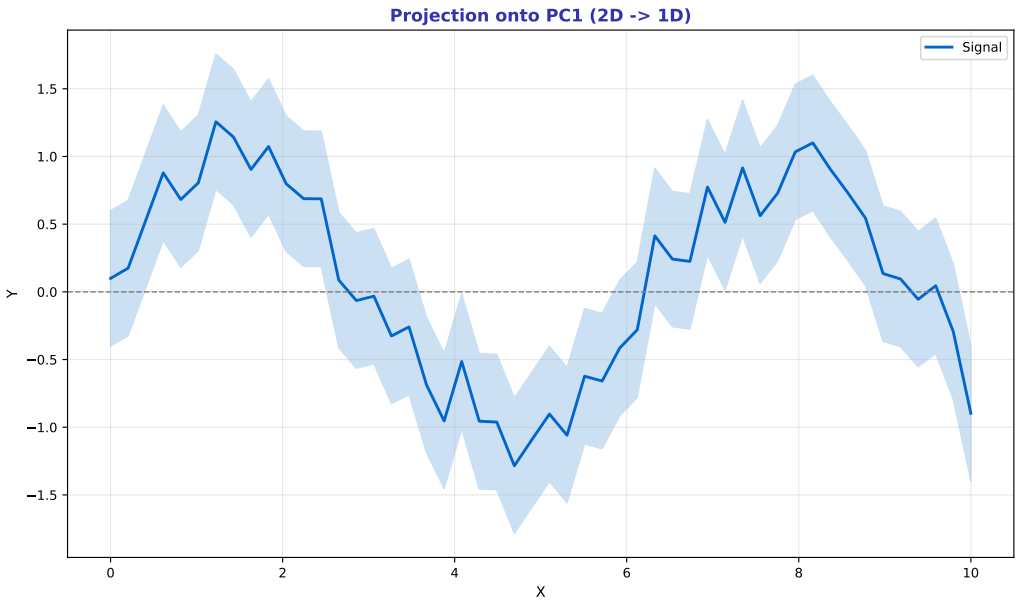


Figure 49: Projecting 2D data onto PC1 reduces it to a single number per observation while preserving most of the variance. The tiny variance perpendicular to PC1 is discarded.

because everyone is roughly in the middle. You could describe each person's position with just one number (distance along the hallway) and lose essentially no information.

Now imagine the crowd in a square plaza. People are scattered in both directions. There is no single "long" direction. PC1 and PC2 both capture significant variation, and neither can be discarded. Whether PCA helps depends on whether your data is hallway-shaped (correlated features, reducible) or plaza-shaped (uncorrelated features, nothing to reduce).

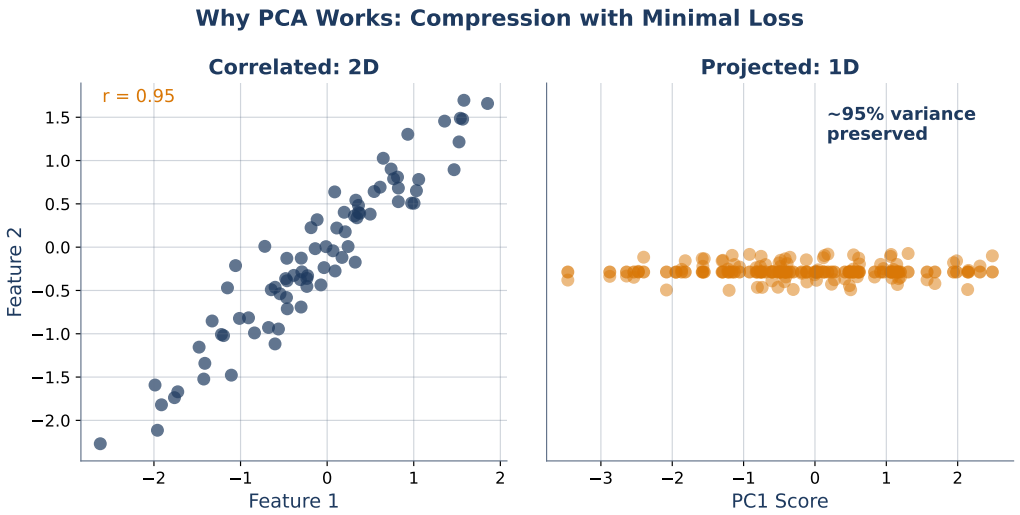
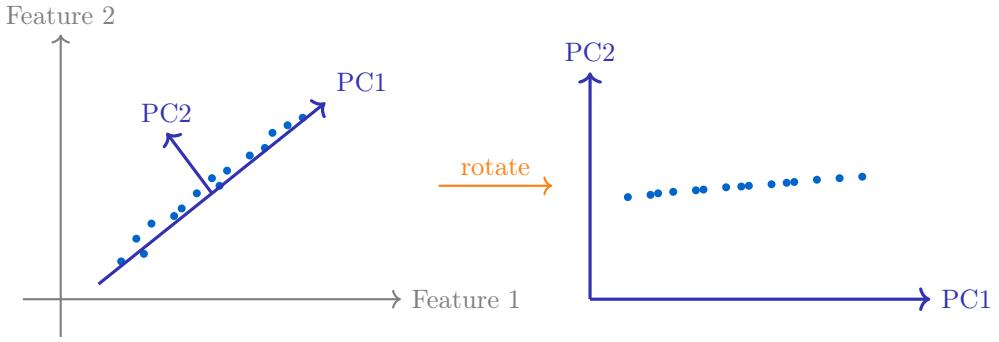


Figure 50: PCA works only when features are correlated. Highly correlated data collapses to a low-dimensional structure. Uncorrelated data stays high-dimensional.

The geometric picture deserves a cleaner diagram. Before PCA, your data lives in the original coordinate system defined by the raw features. After PCA, the same data lives in a rotated coordinate system where the axes are principal components.



Before PCA: axes = raw features **After PCA:** axes = principal components

The total variance in the data has not changed—only the coordinate system has. But in the new system, the variance is concentrated in PC1 (long spread) while PC2 carries very little (narrow spread). Discarding PC2 loses almost nothing.

The Mathematical Recipe

PCA boils down to an eigendecomposition of the covariance matrix. Given a data matrix $X \in \mathbb{R}^{n \times p}$ with n observations and p features:

1. **Center** the data: subtract the column means so every feature has mean zero.
2. **Compute** the sample covariance matrix $\Sigma = \frac{1}{n-1} X^\top X$.

3. **Eigendecompose** Σ: find eigenvectors $v_1, v_2, \ldots, v_p$ and eigenvalues $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_p \geq 0$.

4. **Keep** the top k eigenvectors as the new basis. These are the principal components.

5. **Project** the data: $Z = XV_k$, where V_k is the $p \times k$ matrix of top eigenvectors.

Principal Component: An eigenvector of the covariance matrix, sorted by eigenvalue. PC1 is the eigenvector with the largest eigenvalue—it points in the direction of maximum variance. PC2 is the eigenvector with the second largest eigenvalue, orthogonal to PC1, and so on.

Eigenvalue: The variance captured by a principal component. If $\lambda_1 = 4.2$ and the total sum of eigenvalues is 6.0, then PC1 explains $4.2/6.0 = 70\%$ of the total variance.

Key Formula: PCA Decomposition

Starting from the centered data matrix $X_c = X - \bar{X}$, PCA solves the eigenvalue problem:

$$\Sigma v_i = \lambda_i v_i, \qquad \Sigma = \frac{1}{n-1} X_c^\top X_c$$

where v_i is the i-th principal component (eigenvector) and λ_i is its eigenvalue (variance captured). The projected data in PC space is:

$$Z = X_c V_k$$

where $V_k = [v_1 \mid v_2 \mid \cdots \mid v_k]$ holds the top k eigenvectors as columns and $Z \in \mathbb{R}^{n \times k}$ is the new low-dimensional representation.

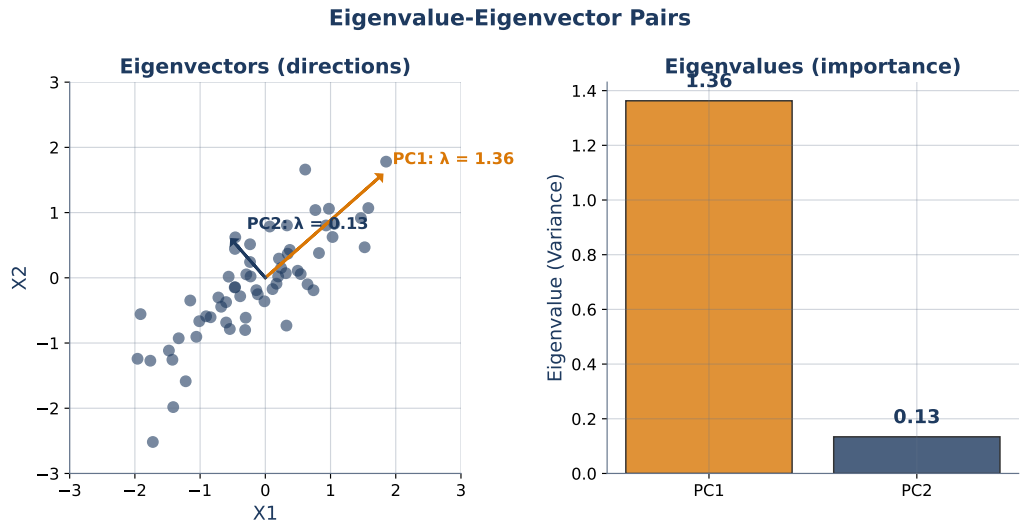


Figure 51: Eigendecomposition of the covariance matrix. Eigenvectors give the directions of principal components; eigenvalues give the variance along each direction.

Why does this work? The covariance matrix encodes how features vary together. Its eigenvectors are the directions in which that variation is "pure"—uncorrelated with every other direction. The eigenvalue of each eigenvector is the variance captured along it. Sorting eigenvectors by eigenvalue gives you the directions in decreasing order of importance.

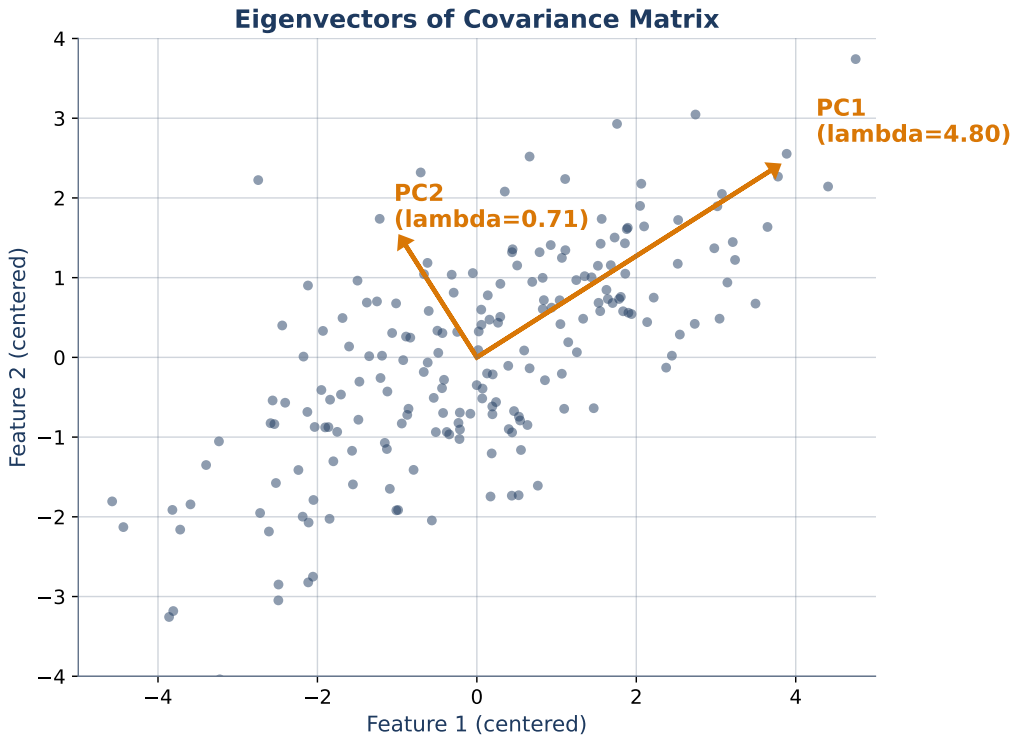


Figure 52: Eigenvectors of the covariance matrix point along the natural axes of the data cloud. They are perpendicular to each other and form an orthonormal basis.

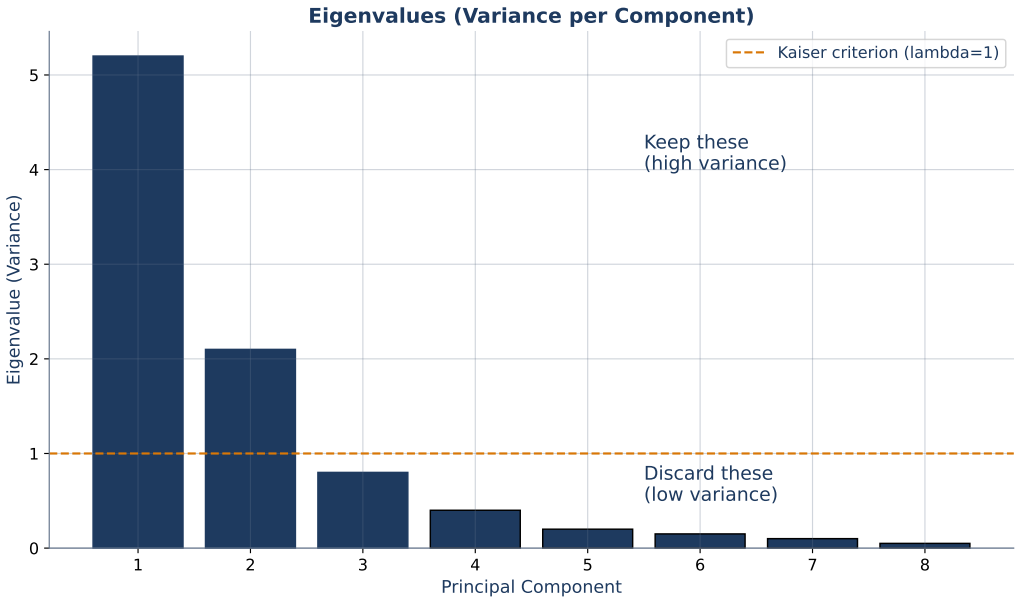


Figure 53: Each eigenvalue equals the variance of the data along the corresponding eigenvector. Summing all eigenvalues gives the total variance in the dataset.

Centering and Scaling: the Preparation Step

PCA finds directions of maximum variance. If one feature is in dollars (range: 100,000 to 5,000,000) and another is in years (range: 0 to 50), the dollar feature will dominate every eigenvalue simply because its numerical range is larger—not because it carries more information. The fix is standardization: subtract the mean and divide by the standard deviation for every feature before running PCA.

Centering (subtracting the mean) is mandatory. PCA operates on deviations from the mean, and skipping this step produces meaningless results. Scaling (dividing by the standard deviation) is strongly recommended whenever features are measured in different units. The only time you skip scaling is when all features are already on the same scale—for example, stock returns across a basket where every variable is a daily percentage change.

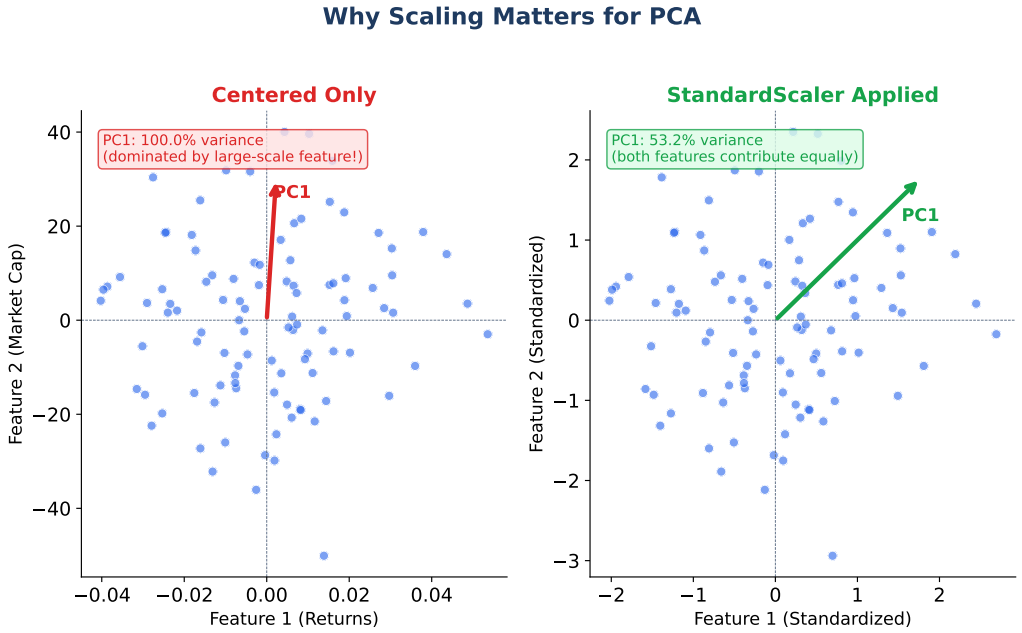


Figure 54: Centering shifts the data so its mean is zero. Scaling makes every feature have unit variance. Both are typically required before PCA.

Common Misconceptions about PCA

(1) "PCA removes noise." PCA finds directions of maximum variance. If noise has high variance, PCA will capture noise first, not signal. A stock with erratic day-to-day jumps may dominate PC1 for all the wrong reasons. PCA is a variance-preserving method, not a denoising method.

(2) "You can apply PCA to categorical data." PCA assumes continuous, numeric features and computes means and variances. Applying it to one-hot-encoded categoricals produces distorted components that reflect encoding artifacts rather than meaningful structure. For categorical data, use Multiple Correspondence Analysis (MCA) or factor analysis of mixed data.

(3) "PCA components have interpretable meaning." Principal components are linear combinations of the original features—weighted sums of 200 variables—not any single real-world concept. Sometimes a component aligns with an intuitive idea ("market factor") and sometimes it does not. Interpretation is always post-hoc and always provisional.

Worked Examples

Worked Example 1: PCA on a 2D Toy Dataset

Five points in 2D (already centered):

$$A = (2, 2), \quad B = (3, 3), \quad C = (-2, -2), \quad D = (-3, -3), \quad E = (0, 0)$$

The covariance matrix is:

$$\Sigma = \frac{1}{4}\begin{bmatrix} 26 & 26 \\ 26 & 26 \end{bmatrix} = \begin{bmatrix} 6.5 & 6.5 \\ 6.5 & 6.5 \end{bmatrix}$$

Eigenvalues of Σ: $\lambda_1 = 13$, $\lambda_2 = 0$.
Eigenvectors: $v_1 = \frac{1}{\sqrt{2}}(1, 1)^\top$ (PC1, diagonal), $v_2 = \frac{1}{\sqrt{2}}(-1, 1)^\top$ (PC2, anti-diagonal).
Interpretation: PC1 captures all the variance (13) because the points lie exactly on the line $y = x$. PC2 captures zero variance. Projecting the points onto PC1 gives coordinates $\{2\sqrt{2}, 3\sqrt{2}, -2\sqrt{2}, -3\sqrt{2}, 0\}$—a perfect 1D summary. You have compressed five 2D points into five 1D numbers with zero information loss.

Worked Example 2: PCA on Stock Returns

You collect daily returns for six tech stocks (AAPL, MSFT, GOOGL, META, NVDA, AMZN) over 500 trading days. The raw data matrix is 500×6.
Step 1 – Standardize. All six columns have roughly the same scale (daily returns, $\pm 2\%$), but you standardize anyway for consistency.
Step 2 – Compute the covariance matrix. It is a 6×6 symmetric matrix. All off-diagonal entries are positive because these six stocks tend to move together (correlated).
Step 3 – Eigendecompose. You get six eigenvalues. Suppose they are $\lambda_1 = 3.8$, $\lambda_2 = 0.9$, $\lambda_3 = 0.5$, $\lambda_4 = 0.3$, $\lambda_5 = 0.3$, $\lambda_6 = 0.2$. The total is 6.0 (equal to the sum of standardized variances).
Step 4 – Explained variance. PC1 explains $3.8/6.0 = 63\%$; PC2 explains $0.9/6.0 = 15\%$; together they explain 78%.
Step 5 – Interpret PC1. Its eigenvector has six positive loadings, roughly equal. A positive movement on PC1 means all six stocks moved up together. This is the *market factor*—the common component that drives returns across a sector. In finance, PC1 on a broad universe almost always isolates the market factor.

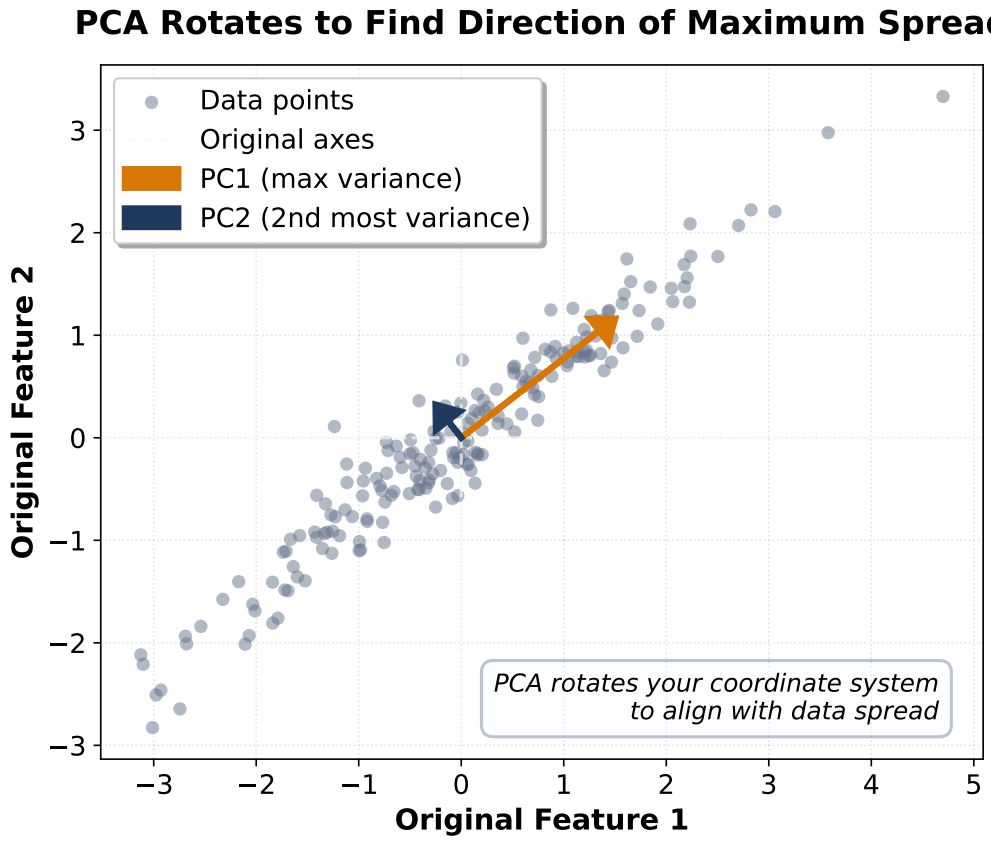


Figure 55: PCA intuition: a high-dimensional cloud of points is redescribed in a coordinate system aligned with its natural spread.

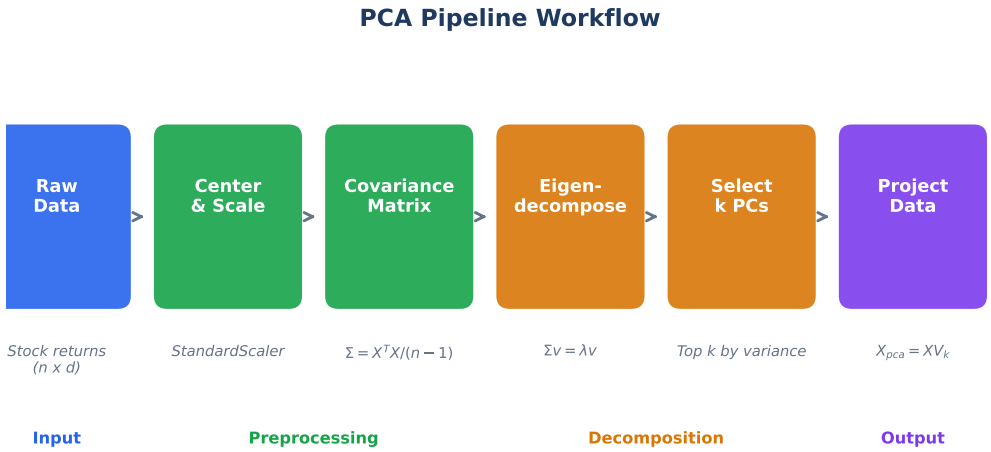


Figure 56: End-to-end PCA workflow: standardize, covariance, eigendecompose, select components, project.

Historical Background: Karl Pearson and the First PCA (1901)

In 1901, the British statistician Karl Pearson published a paper titled *On Lines and Planes of Closest Fit to Systems of Points in Space*. He was not trying to invent machine learning—he was trying to fit a line through a 3D cloud of points in such a way that the perpendicular distances from the points to the line were minimized. This is not ordinary regression, which minimizes vertical distances. It is a symmetric, geometric notion of "best fit" that treats all variables on equal footing.

Pearson showed that the solution was the eigenvector of the covariance matrix corresponding to the largest eigenvalue. That eigenvector is PC1. He extended the idea to higher dimensions: the best-fitting plane uses the top two eigenvectors, the best-fitting 3D hyperplane uses the top three, and so on.

Pearson did not call it "principal component analysis"—that term came later from Harold Hotelling in 1933. And Pearson did not have computers to implement it at scale. But the geometric intuition—fit a line to a cloud of points by minimizing perpendicular distance—is still the clearest way to explain PCA to a beginner. What Pearson did by hand for data with three or four variables, modern software does in milliseconds for datasets with thousands of features.

Problem 5.1 (Easy)

A 3x3 covariance matrix has the form:

$$\Sigma = \begin{bmatrix} 4 & 3 & 0 \\ 3 & 4 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Without computing eigenvalues, describe where PC1 is most likely to point. Is it along one of the original axes, or some combination? Justify your answer.

Solution: see Appendix.

Problem 5.2 (Easy)

Explain in your own words the difference between centering and standardizing data before PCA. Which is mandatory and which is optional? Give an example where you would skip standardization.

Solution: see Appendix.

Problem 5.3 (Medium)

A 5-dimensional dataset produces eigenvalues $[4.2, 1.8, 0.5, 0.3, 0.2]$. Compute: (a) the explained variance ratio for each component, (b) the cumulative explained variance up to 3 components, (c) the minimum number of components needed to retain at least 90% of the variance.

Solution: see Appendix.

Problem 5.4 (Medium)

A 2D dataset has centered data points $(1, 2)$, $(2, 4)$, $(-1, -2)$, $(-2, -4)$. Compute PC1 by inspection (the points all lie on a line—what is it?). Then compute the projection of $(1, 2)$ onto PC1. What is the reconstruction error if you keep only PC1 and drop PC2?
Solution: see Appendix.

Problem 5.5 (Hard)

Prove that the eigenvectors of a symmetric real matrix are orthogonal. Your proof should use the property that for a symmetric matrix A, $x^\top Ay = y^\top Ax$ for any vectors x, y. State clearly where symmetry is used.
Solution: see Appendix.

Connecting Forward

We now know that PCA rotates the coordinate system to align with the directions of maximum variance, and we know how to compute it via eigendecomposition of the covariance matrix. But we have not yet answered the practical question: *how many components should you keep?* Two? Five? Twenty?

Section 6 tackles this question with two complementary tools. The **scree plot** graphs eigenvalues in decreasing order and looks for an elbow—the point where additional components add little variance. The **loadings heatmap** shows the contribution of each original feature to each principal component, making it possible to interpret what PC1 and PC2 actually represent in the language of the original variables.

We will also see how PCA extracts latent financial factors from stock returns—isolating the market factor, the size factor, and the value factor from nothing more than a matrix of daily price changes. This is one of the most elegant applications of unsupervised learning in quantitative finance.

Key Takeaway: PCA rotates your coordinate system to align with the directions of maximum variance—it does not discard features, it combines them into a smaller set of uncorrelated components.

6. Reading the Components – Scree Plots, Loadings, and Factor Extraction

Opening Problem: What Does PC1 Mean?

You ran PCA on daily returns for 30 stocks in the S&P 500 technology sector. The first principal component explains 58% of the total variance. Your manager asks a reasonable question: *"What does PC1 represent?"*

You open the numerical output and see that PC1 is an eigenvector with 30 loadings—one per stock—ranging from +0.18 to +0.24. Every loading is positive. Every loading is roughly similar in magnitude. You stare at the numbers. What do you tell your manager?

And another question lurks behind that one. You kept the first 5 components out of 30, which together explain 82% of the variance. Why did you pick 5? Why not 3? Why not 10? You ran some code and the algorithm returned a number, but you cannot yet articulate the principle behind the choice.

This section gives you the two tools that make PCA interpretable. The **scree plot** visualizes eigenvalues so you can see—literally—where components stop being useful. The **loadings heatmap** reveals how original features map onto principal components, turning abstract eigenvectors into interpretable financial factors. Together, these two tools transform PCA from a black-box dimensionality reduction into a storytelling instrument.

Discovery Question

You run PCA on stock returns and find that the first principal component explains 60% of variance and has positive loadings on every stock. Without any labels, can you name what this component represents? And how confident can you be in that interpretation?

The Scree Plot: Where Does the Elbow Bend?

After eigendecomposition, you have a list of eigenvalues in decreasing order. Plot them as a bar chart with the component index on the horizontal axis and the eigenvalue on the vertical axis. This is a scree plot, named after the rocky debris that accumulates at the bottom of a cliff face—visually, the plot looks like a steep drop followed by a flat tail.

The idea is simple. Useful components are the ones above the elbow, where eigenvalues are large. Components below the elbow contribute almost nothing and can be discarded. Finding the elbow is a judgment call—there is no formula—but a clear scree plot makes the decision visual and defensible.

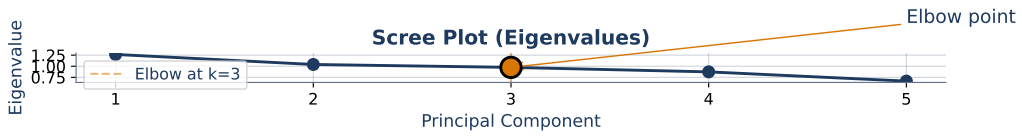


Figure 57: A scree plot: eigenvalues sorted in decreasing order. The elbow (where the curve flattens) suggests the number of components to retain.

Three heuristics compete for the job of picking the number of components:

- **The elbow rule**: Look at the scree plot and find where the curve flattens. Keep everything up to the elbow.
- **The cumulative variance rule**: Pick a target (80%, 90%, 95%) and keep enough components to reach it.

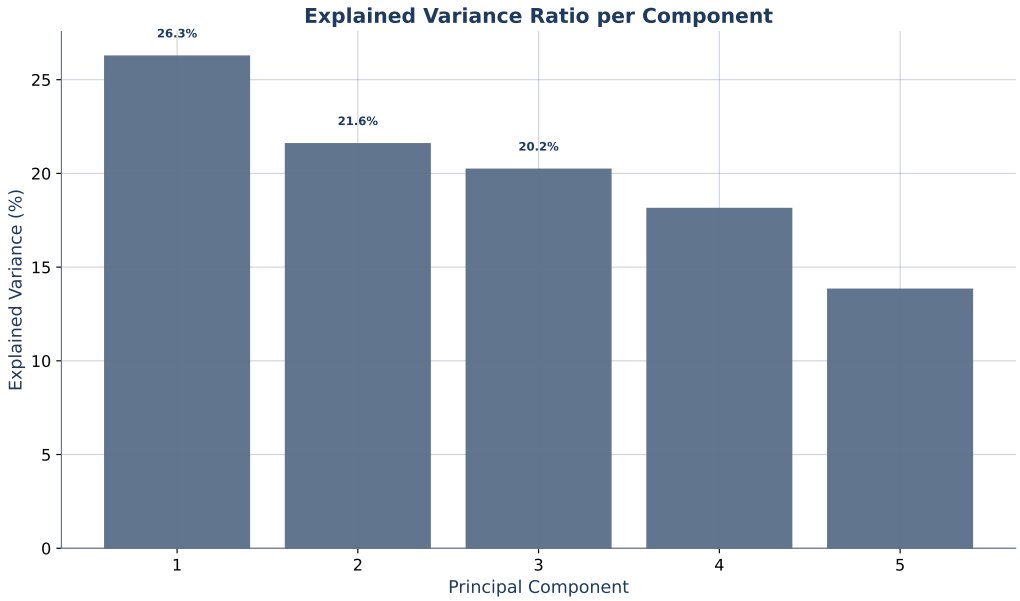


Figure 58: Explained variance ratio: each bar shows the fraction of total variance captured by one component. Summing gives the cumulative explained variance.

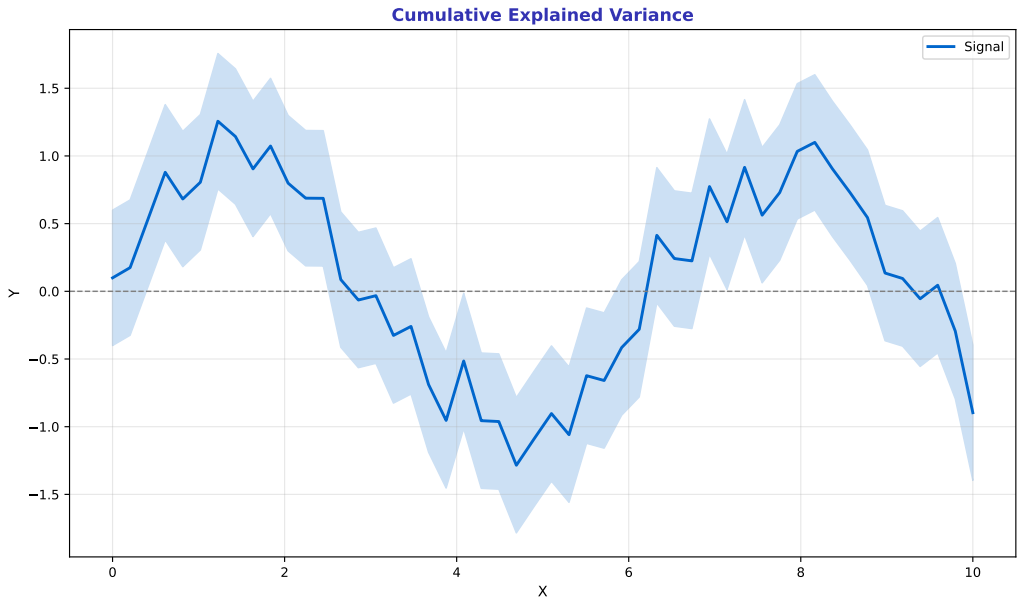


Figure 59: Cumulative explained variance curve. A common heuristic: keep components until you reach a target threshold such as 80% or 95%.

- **The Kaiser criterion**: Keep only components with eigenvalue greater than 1 (when features are standardized). The idea is that a component should carry more variance than a single original feature.

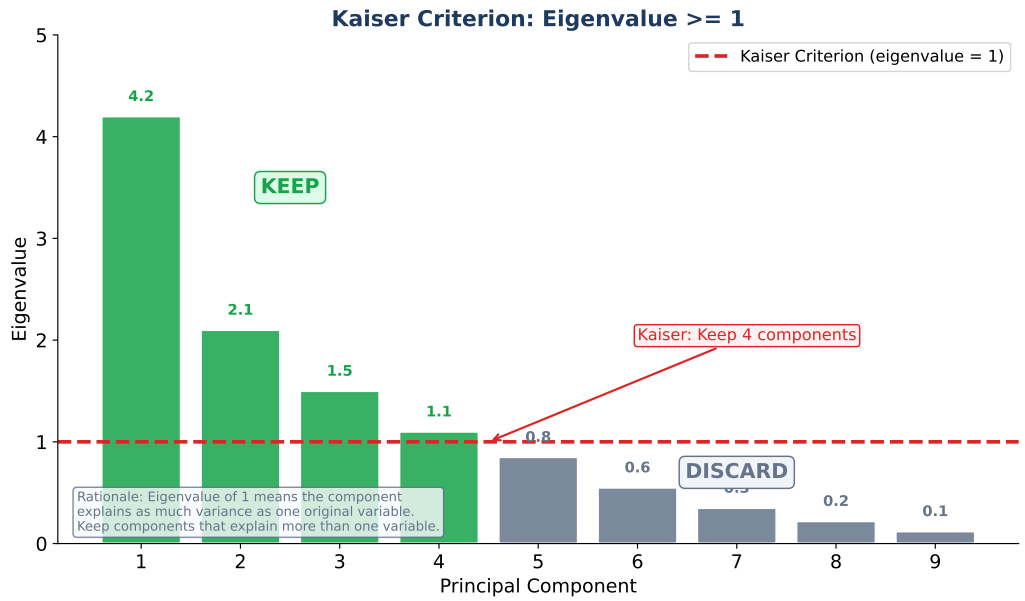


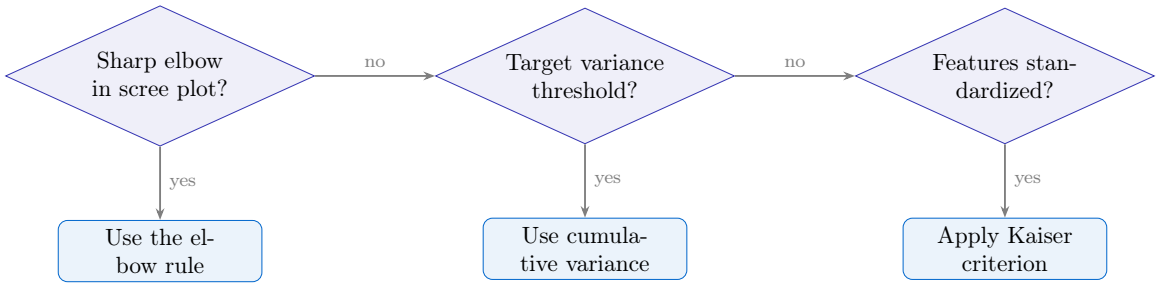
Figure 60: The Kaiser criterion keeps components whose eigenvalue exceeds 1 on standardized data. This cutoff is arbitrary but serves as a quick first filter.

None of these rules is universally correct. The elbow rule is most defensible when the scree plot has a clean drop, but real scree plots often show gradual decays with no sharp elbow. The cumulative threshold is transparent but arbitrary—why 90% instead of 85%? The Kaiser criterion is easy to apply but tends to retain too many components in large datasets. In practice, run all three, see where they agree, and exercise judgment where they diverge.

Scree plot: A bar chart of eigenvalues sorted in decreasing order, used to identify the number of meaningful principal components. The name comes from the rocky debris (scree) at the foot of a mountain.

Kaiser criterion: A rule that retains only principal components whose eigenvalue is greater than 1, under the assumption that the features were standardized. Components with eigenvalue below 1 explain less variance than a single original standardized feature.

When faced with a new dataset, which rule should you use? The flowchart below offers a practical decision path.



If multiple rules point to similar numbers, you are in good shape. If they disagree wildly, look at the data itself. A scree plot without an elbow is a warning sign—it usually means the features are too independent for dimensionality reduction to help.

Loadings: What Does Each Component Mean?

A loading is the coefficient of an original feature in a principal component. If PC1 has the loading vector

$$v_1 = (0.40, 0.39, 0.41, 0.38, 0.42, 0.40)^\top$$

for six stocks, then the PC1 score for an observation is $z_1 = 0.40x_1 + 0.39x_2 + 0.41x_3 + 0.38x_4 + 0.42x_5 + 0.40x_6$—a roughly equal-weighted sum of all six features. When every loading is positive and similar in magnitude, PC1 is a "market factor": it captures whether all six stocks went up or down together.

Now suppose PC2 has loadings $(0.5, 0.5, 0.5, -0.5, -0.5, -0.5)$. The first three stocks have positive weights, the last three have negative weights. A high PC2 score means the first three stocks outperformed the last three. If the first three are cloud-computing stocks and the last three are semiconductors, PC2 isolates the cloud-vs-chip spread. PC2 is a "relative-value factor": it captures differences between groups rather than joint movements.

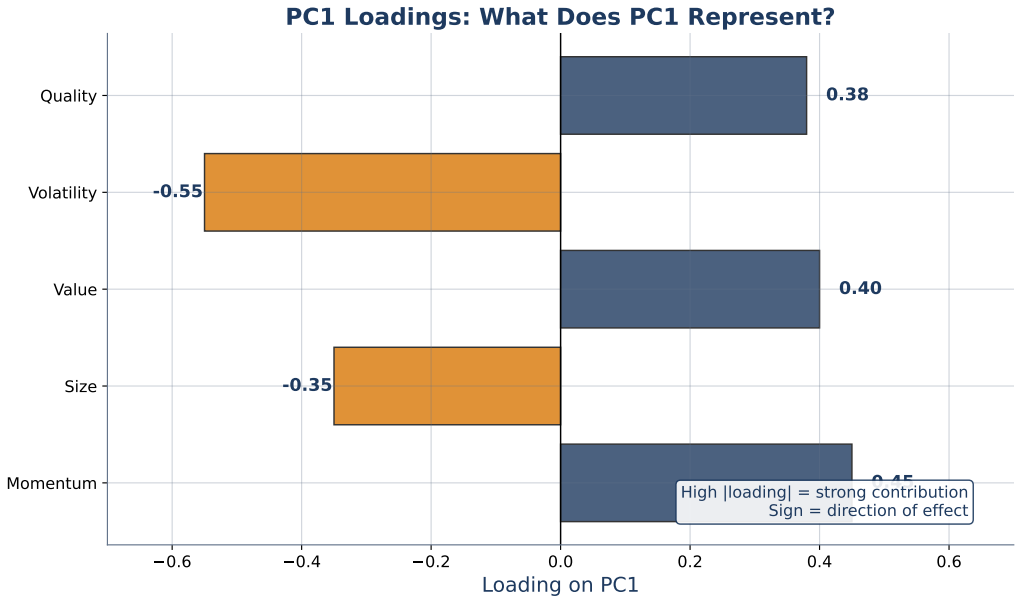


Figure 61: Loadings are the weights assigned to each original feature in a principal component. They reveal which features contribute most—and in which direction—to each component.

Key Formula: Loadings and Projection

Let v_k be the k-th principal component (a unit eigenvector with p entries). For an observation $x \in \mathbb{R}^p$, the PCk score is:

$$z_k = v_k^\top x = \sum_{j=1}^{p} v_{k,j} x_j$$

where $v_{k,j}$ is the loading of feature j on component k. The sign and magnitude of $v_{k,j}$ determine whether feature j pushes the score up or down and by how much.
Some software packages scale loadings by $\sqrt{\lambda_k}$ to give them units of standard deviation. The scaled loadings are:

$$\ell_{k,j} = v_{k,j}\sqrt{\lambda_k}$$

Scaled loadings make cross-component comparisons easier but change the absolute magnitudes.

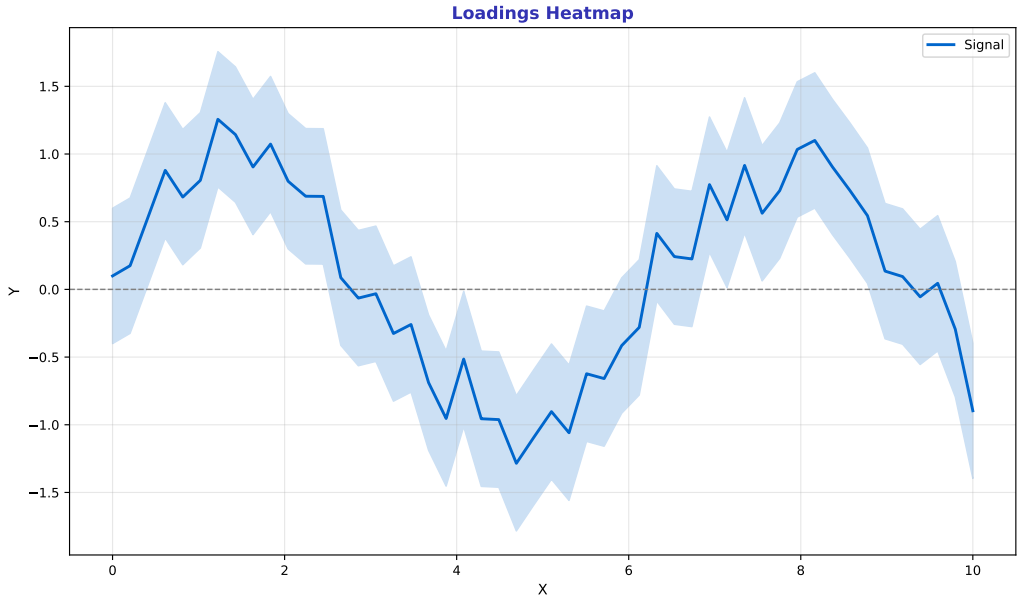


Figure 62: A loadings heatmap shows every feature-component pair as a colored cell. Red cells indicate positive loadings; blue cells indicate negative ones. Patterns jump out immediately.

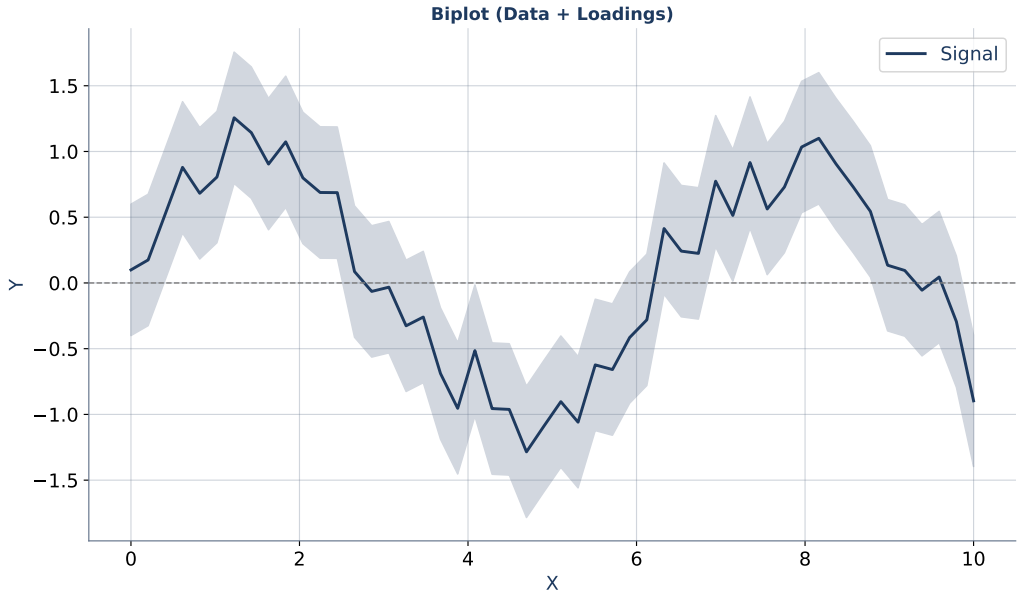


Figure 63: A biplot overlays data points (in PC1-PC2 space) with loading arrows (in the same coordinate system). Arrows pointing in the same direction indicate positively correlated features.

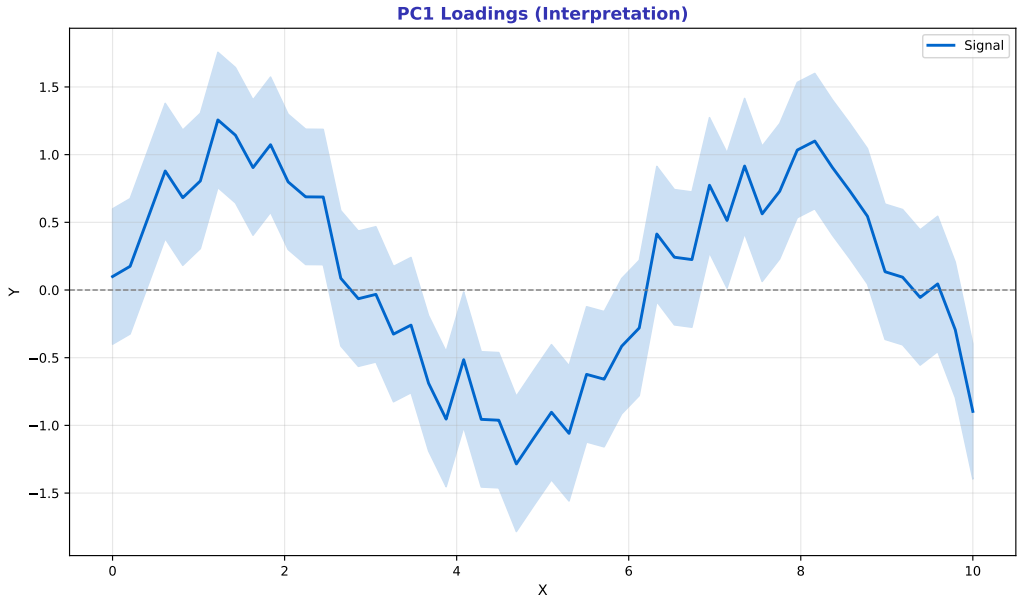


Figure 64: Interpreting PC1 loadings: the bar plot shows the contribution of each feature to PC1. Features with large positive loadings drive PC1 scores up.

Loading: The weight assigned to an original feature in a principal component. A large positive loading means the feature contributes strongly and positively to the component; a large negative loading means it contributes strongly and negatively. Loadings are the raw material of PCA interpretation.

Biplot: A 2D plot that overlays data points (in PC1-PC2 coordinates) with loading vectors for each original feature. It shows simultaneously where observations sit and which features push them apart.

Common Misconceptions about Scree and Loadings

(1) "Always keep components until 95% variance explained." In finance, even 50% of the variance can be meaningful if the leading components capture well-known risk factors. Conversely, keeping components until 95% may include noise that happened to have moderate variance in the training set. Choose a threshold that fits your purpose, not a one-size-fits-all rule.

(2) "Large loadings mean important features." Loadings reflect contribution to a specific component, not overall importance. A feature can have a large loading on PC3 while being irrelevant to the first two components—which together explain far more variance. Always weight loadings by the variance of their component before declaring a feature "important."

(3) "PCA loadings are stable across samples." Loadings for the top components (where eigenvalues are large and well-separated) tend to be stable. But loadings for trailing components can change dramatically with new data, especially when eigenvalues are close together. Trust loadings only where eigenvalue separation is clear.

Worked Examples

Worked Example 1: Extracting the Market Factor

You run PCA on daily returns of 20 large-cap US stocks over 2020–2023 (about 750 trading days). The eigenvalues are $\lambda_1 = 12.4$, $\lambda_2 = 1.8$, $\lambda_3 = 1.2$, $\lambda_4 = 0.9$, ... The total is 20 (sum of standardized variances equals the number of features).

PC1 explains $12.4/20 = 62\%$ of the variance. Its loadings are all positive and range from 0.19 to 0.25—roughly equal-weighted. When you compute PC1 scores over time and correlate them with the S&P 500 daily return, the correlation is 0.97. PC1 *is* the market factor, discovered without ever being told what to look for.

PC2 explains $1.8/20 = 9\%$. Its loadings split the stocks into two groups: positive loadings on utilities and consumer staples, negative loadings on technology and financials. A high PC2 day means defensive sectors outperformed growth sectors. PC2 is a "risk-on/risk-off" factor, again discovered purely from the return data.

This is the power of unsupervised learning: no labels, no supervision, yet the algorithm extracts economically meaningful structure.

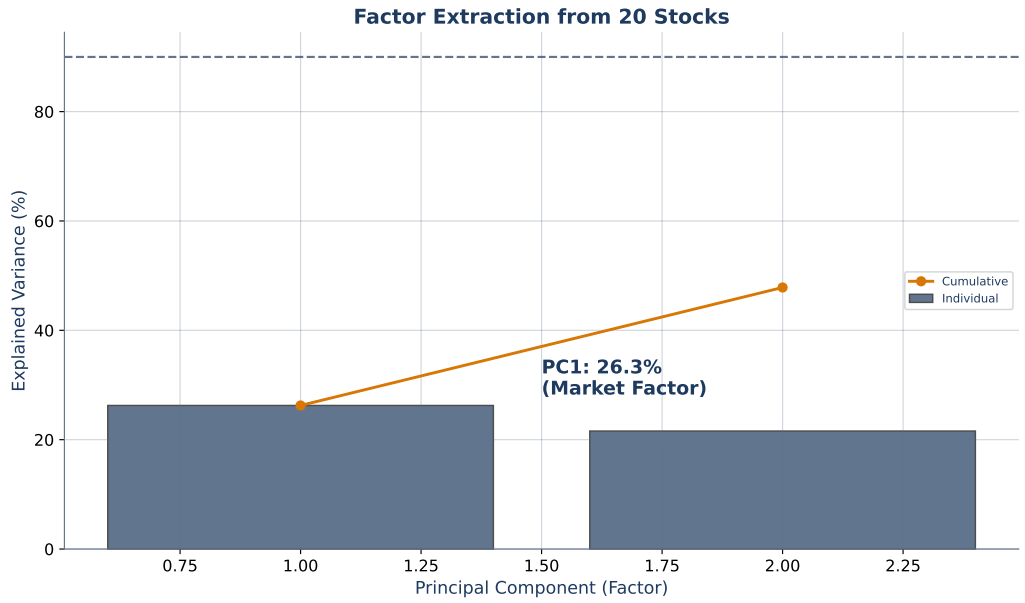


Figure 65: Extracting latent factors from stock returns: PCA decomposes the return matrix into principal components that often align with well-known financial factors.

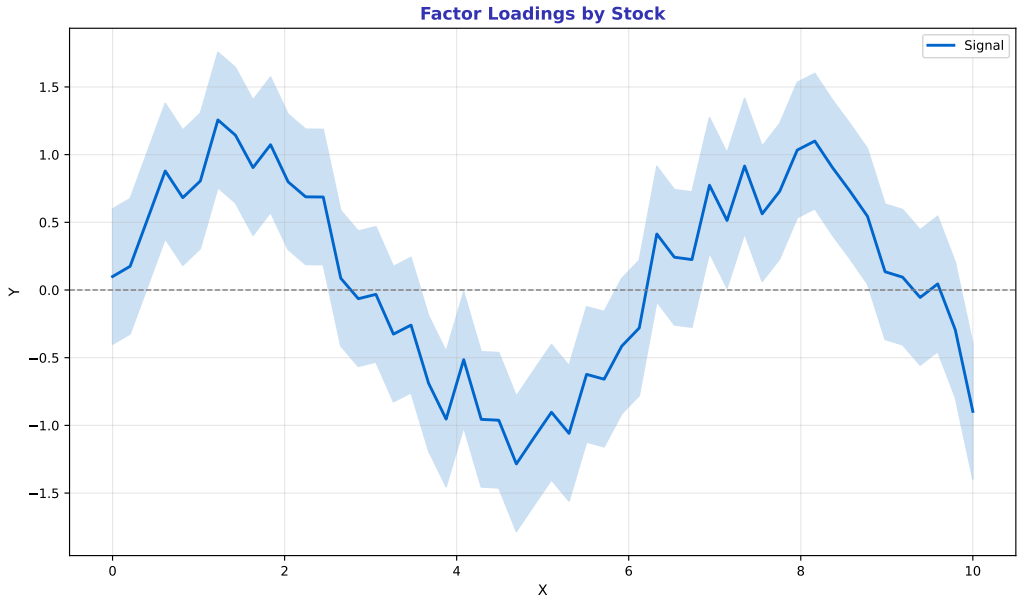


Figure 66: Factor loadings by stock. The pattern across loadings reveals which stocks drive each latent factor—market, sector, or style.

Worked Example 2: Reconstruction Error

A dataset has 10 features and you keep only the top 3 principal components. You project the data onto PC1-PC3 and then back-project to the original 10D space. The reconstructed data is close to, but not exactly equal to, the original. The reconstruction error for each observation is:

$$\text{err}_i = \|x_i - \hat{x}_i\|^2$$

where $\hat{x}_i$ is the back-projection of x_i. Summing over all observations gives:

$$\text{total error} = \sum_{i=1}^{n} \|x_i - \hat{x}_i\|^2 = (n-1) \sum_{k=4}^{10} \lambda_k$$

The total error equals $(n-1)$ times the sum of the discarded eigenvalues. If $\lambda_4 + \lambda_5 + \ldots + \lambda_{10} = 0.8$ and $n = 500$, the total squared error is $499 \times 0.8 \approx 399$. The average per-observation error is about 0.8—roughly the "variance left behind."

This formula is why the cumulative explained variance is a direct proxy for reconstruction fidelity. Keeping components until 95% variance retained leaves 5% reconstruction error.

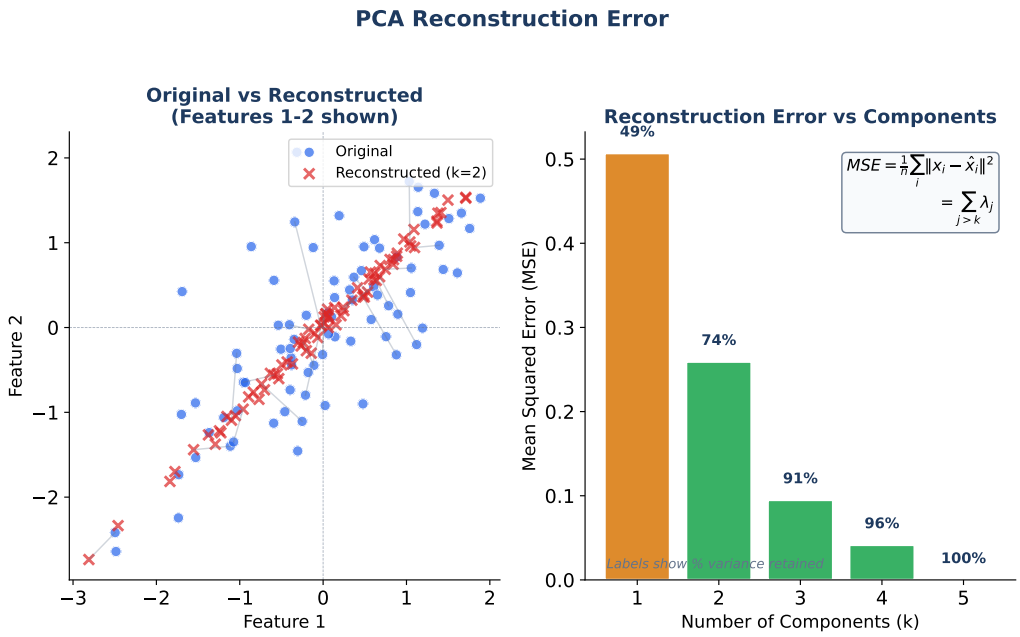


Figure 67: Reconstruction error as a function of the number of components retained. More components means lower error, but the marginal improvement shrinks rapidly after the elbow.

Historical Background: Harold Hotelling's Algebraic Formulation (1933)

Pearson's 1901 paper gave PCA its geometric intuition: fit a line through a cloud of points by minimizing perpendicular distances. But Pearson did not provide a general algebraic recipe. That contribution came in 1933 from the American statistician Harold Hotelling, working at Columbia University.

Hotelling's paper *Analysis of a Complex of Statistical Variables into Principal Components* formalized the connection between PCA and eigendecomposition of the covariance matrix. He proved that the first principal component is the eigenvector with the largest eigenvalue, that subsequent components are orthogonal to it, and that together they partition the total variance. He also introduced the term "principal component," which stuck.

Hotelling was motivated by educational psychology—specifically, factor analysis of intelligence test scores. He wanted to discover whether different test questions were measuring different latent traits or a single underlying "general intelligence." PCA gave him the algebraic tool to answer that question rigorously. The same method, nine decades later, powers everything from image compression to genome-wide association studies to latent factor models in asset pricing.

Hotelling turned Pearson's geometric intuition into an algebraic recipe: decompose the covariance matrix into eigenvectors and eigenvalues. Every PCA you run today still follows his recipe exactly.

Problem 6.1 (Easy)

A scree plot shows eigenvalues $[3.2, 1.4, 1.1, 0.6, 0.4, 0.2, 0.1]$ on standardized data. Apply the Kaiser criterion: which components should you retain? What percentage of the total variance do they explain?

Solution: see Appendix.

Problem 6.2 (Easy)

A loadings heatmap shows that PC1 has large positive loadings on {P/E, P/B, P/S} and small loadings on everything else. What is the most likely interpretation of PC1? What financial concept does it capture?
Solution: see Appendix.

Problem 6.3 (Medium)

A biplot shows observations colored by country. The data cluster clearly in PC1-PC2 space, with US companies in the upper right and European companies in the lower left. Which two or three feature-loading arrows would you look at first to understand what drives this separation? Explain your reasoning.
Solution: see Appendix.

Problem 6.4 (Medium)

PCA on daily stock returns extracts PC1 with roughly equal positive loadings on every stock, and PC2 with positive loadings on growth stocks and negative loadings on value stocks. Name the financial factor each PC most likely represents. How would you verify these interpretations quantitatively?
Solution: see Appendix.

Problem 6.5 (Hard)

Show algebraically that PCA loadings (scaled by the square root of the eigenvalue) equal the covariance between each original feature and its corresponding principal component score. That is, show:

$$\text{Cov}(x_j, z_k) = \sqrt{\lambda_k}\, v_{k,j}$$

where x_j is feature j, z_k is PCk's score, $v_{k,j}$ is the raw loading, and λ_k is PCk's eigenvalue. Your proof should start from $z_k = v_k^\top x$ and use linearity of covariance.
Solution: see Appendix.

Connecting Forward

Sections 1 through 6 gave us three powerful unsupervised methods: K-Means, hierarchical clustering, and PCA. Each answers a different question—K-Means partitions, hierarchical clustering builds trees, PCA compresses—but they share a common pitfall. Every method requires preprocessing (scaling, centering, imputation), and every method is vulnerable to *data leakage* if the preprocessing is done wrong.

Section 7 addresses this head-on with ML pipelines: a framework for chaining preprocessing and modeling steps so that validation data never contaminates training data. The pipeline is the most important engineering concept in modern machine learning. Every production system uses one. Every notebook that does not use one is one step away from an embarrassing failure.

Key Takeaway: Scree plots and loadings heatmaps make PCA interpretable—the scree plot tells you how many components matter, the loadings tell you what each component means.

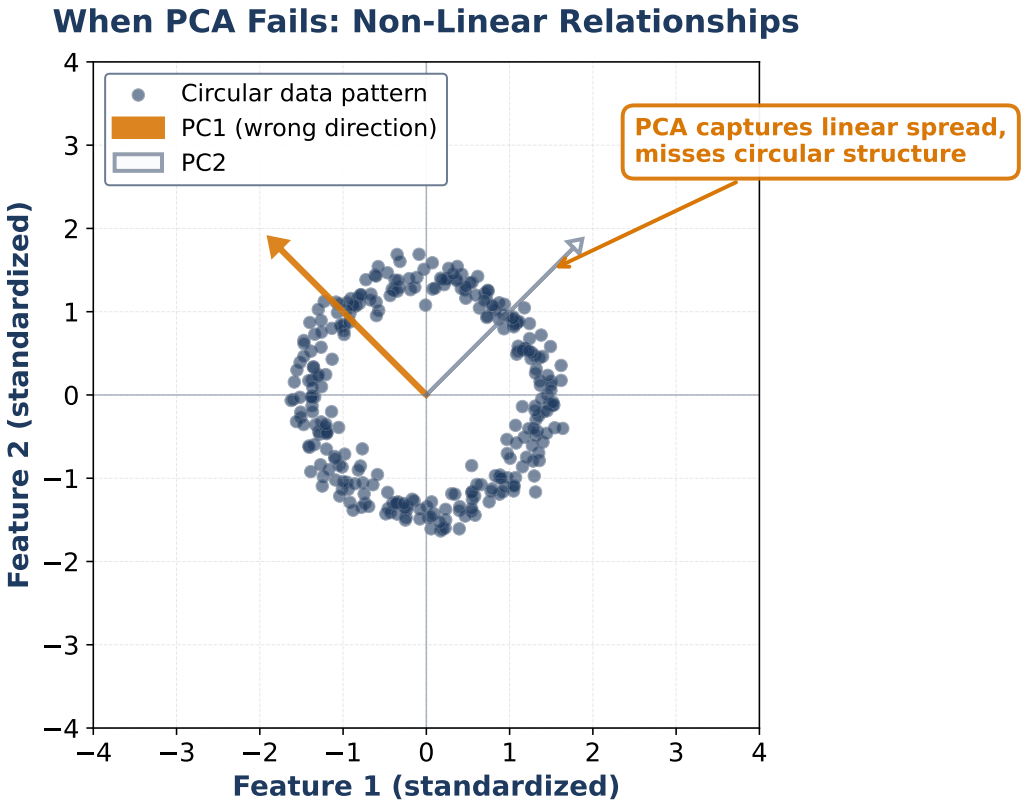


Figure 68: PCA limitations: linearity assumption, sensitivity to scale, and interpretability challenges with many components.

7. The Assembly Line – ML Pipelines and Data Leakage Prevention

Opening Problem: The 92% Model That Fails in Production

A colleague bursts into your office holding a laptop. "I built a fraud detector. It hits 92% accuracy on our validation set." You congratulate her and help deploy the model. Two weeks later she is back, confused. Production accuracy is 61%. The model barely beats a coin flip.

You look at her code. There are no bugs. The model type is reasonable. The training data is clean. You run her notebook end to end and reproduce the 92% validation accuracy. Then you dig deeper. Her workflow was: `StandardScaler().fit_transform(X)` on the entire dataset, then `train_test_split`, then `model.fit(X_train)` and `model.score(X_test)`.

See the problem? She scaled the features using the mean and standard deviation of *both* the training and validation sets. Information from the validation set leaked into the scaling parameters. At training time, the model "knew" the validation distribution. At production time, it did not—and the accuracy collapsed.

This is data leakage, and it is the single most common mistake in applied machine learning. It produces models that look spectacular in development and catastrophic in production. The only cure is discipline: preprocessing must be fit only on training data and then applied to validation and test data as a read-only transformation. Writing that discipline into individual scripts is error-prone. Writing it into a sklearn Pipeline makes the discipline automatic.

This section introduces pipelines as a design pattern that makes data leakage nearly impossible. Every professional ML codebase uses them. Every untrained practitioner eventually learns the hard way why.

Discovery Question

Your model achieves 92% accuracy in your Jupyter notebook. You deploy it and it barely beats random guessing. Your code has no bugs. Your data is correct. What went wrong?

What a Pipeline Really Is

A pipeline is a chain of transformations and a final estimator, packaged as one object. When you call `pipeline.fit(X_train, y_train)`, the pipeline walks through its steps in order: fit the first transformer on `X_train`, transform `X_train`, fit the second transformer on the result, transform it, and so on until it reaches the final estimator, which is fit on the fully transformed data. When you call `pipeline.predict(X_test)`, the same transformers are applied to `X_test` (using parameters learned from training data) and then the estimator makes predictions.

The crucial word here is "learned from training data." The scaler remembers the training mean and standard deviation. When test data arrives, it subtracts that same training mean and divides by that same training standard deviation. The test data does not contribute to fitting anything. That is exactly the discipline that manual workflows violate when someone accidentally calls `fit_transform` on the combined dataset.

Think of a factory assembly line. A car chassis enters at one end, stops at the first station (paint), moves to the next (engine), then the next (wheels), and rolls out finished. Every station applies the same tool to every car. If you skipped a station for half the cars, you would get a half-finished batch. A pipeline in sklearn plays the same role: every row of data passes through the same stations in the same order, with the same tools, every single time.

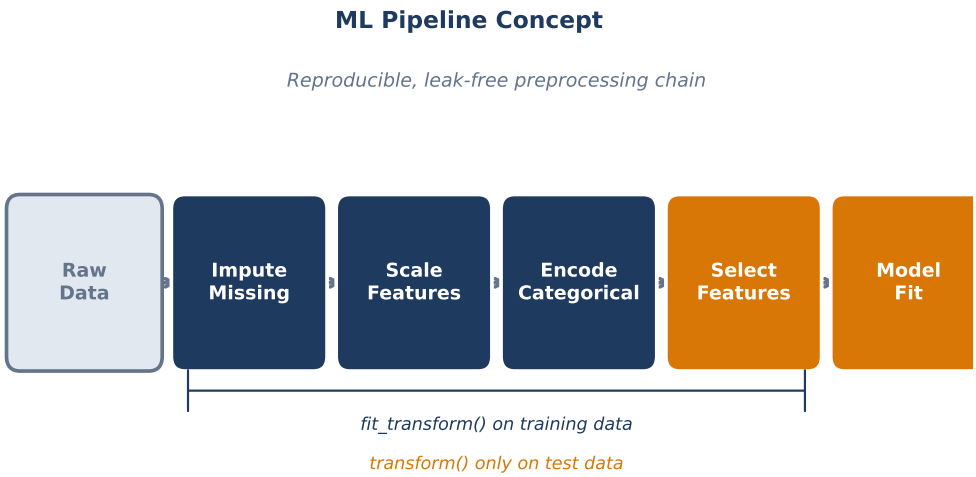


Figure 69: A pipeline chains transformers and a final estimator. Training flows left to right, and parameters learned from training data are reused at prediction time.

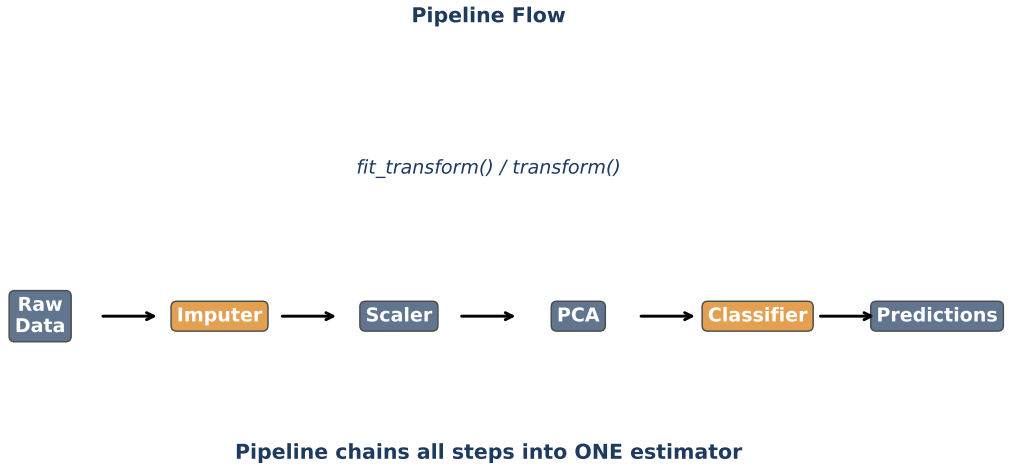


Figure 70: Data flow through a pipeline. Training data is fit and transformed at each step; test data is only transformed using the parameters learned during training.

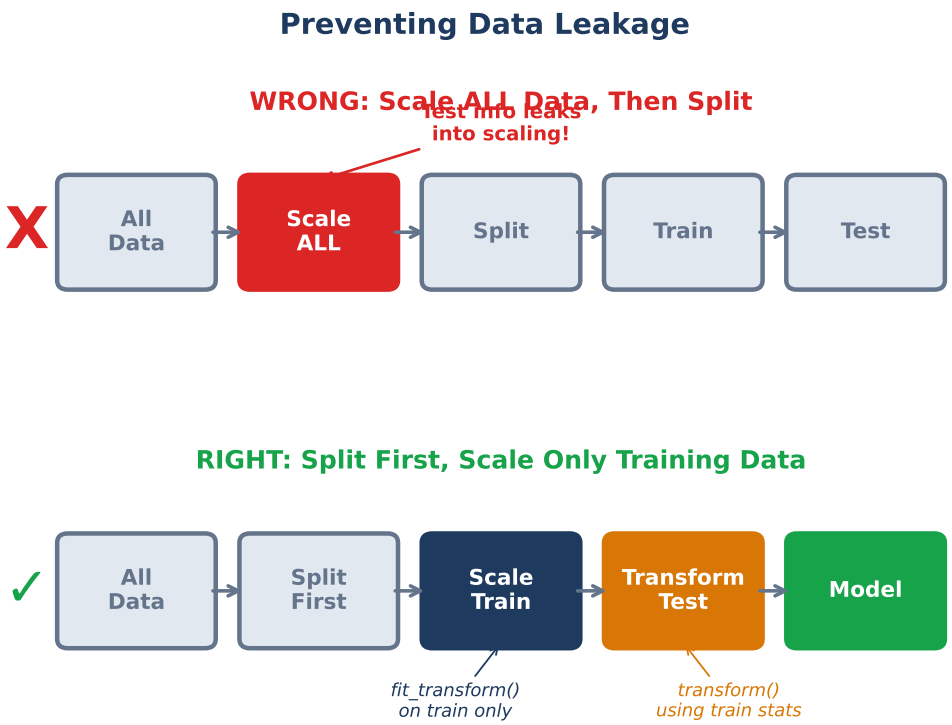
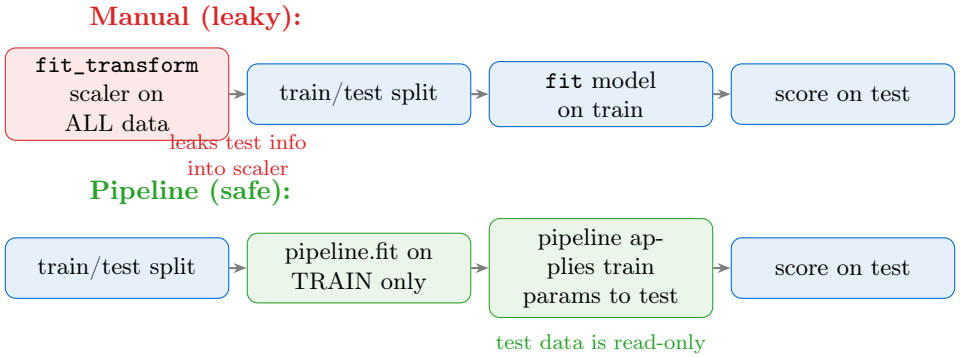


Figure 71: How a pipeline prevents data leakage: preprocessing parameters are learned only from training data, never from validation or test data.

Data leakage: Any situation where information from outside the training set influences model fitting. The most common form is preprocessing that uses statistics from the full dataset (including validation or test data) instead of training data alone.

Pipeline: A sequence of transformers followed by a final estimator, bundled as a single object. Calling `fit` on a pipeline chains `fit_transform` through the transformers and `fit` on the estimator. Calling `predict` chains `transform` through the same transformers and `predict` on the estimator.

A visual comparison makes the difference concrete. The manual workflow is what untrained practitioners write. The pipeline workflow is what experienced practitioners write. The two look nearly identical until you ask: *where is the scaler fit?*



The difference is one line of code but the consequences are enormous. The manual workflow produces an overoptimistic validation score because the scaler has "seen" the test distribution. The pipeline workflow gives an honest validation score because preprocessing is fit only on training data.

Building a Pipeline Step by Step

A minimal sklearn pipeline has two parts: one or more transformers and a final estimator. Transformers implement `fit` and `transform`; estimators implement `fit` and `predict`. The `Pipeline` class glues them together.

Python Code: A Minimal Pipeline

```
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

pipe = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler',  StandardScaler()),
    ('model',   LogisticRegression(max_iter=1000)),
])

pipe.fit(X_train, y_train)
y_pred = pipe.predict(X_test)
```

When `pipe.fit(X_train, y_train)` runs, the imputer learns the median of each column on training data, the scaler learns the training mean and standard deviation, and the logistic regression fits on the scaled, imputed training data. Every step is fit on training data only. When `pipe.predict(X_test)` runs, the same imputer fills missing values using the *training* median, the same scaler applies the *training* mean and standard deviation, and the fitted model predicts. Test data is never used to update any parameter.

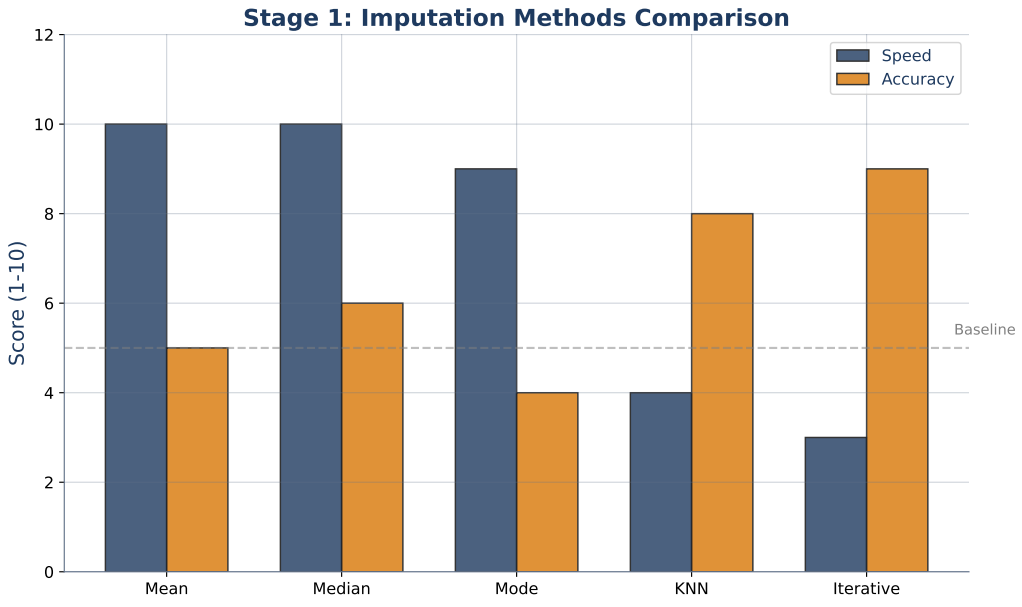


Figure 72: Step 1 of a typical pipeline: impute missing values using a strategy learned from training data (median, mean, or constant).

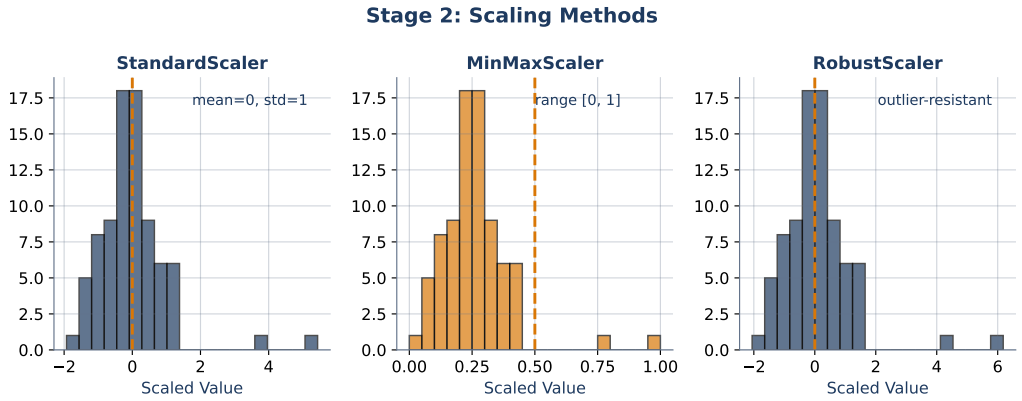


Figure 73: Step 2: scale features. StandardScaler subtracts the training mean and divides by the training standard deviation; MinMaxScaler maps to [0, 1].

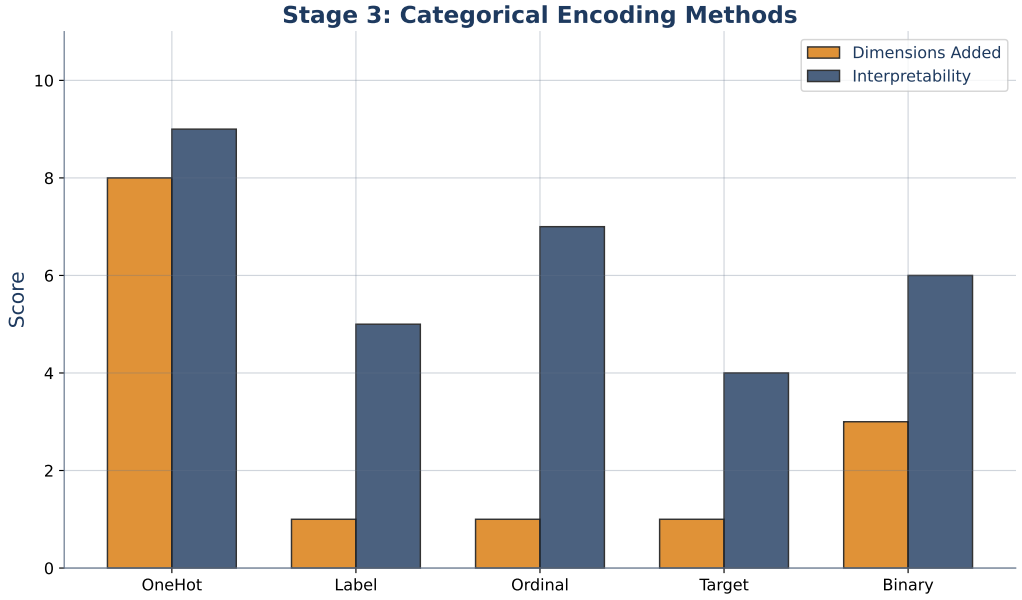


Figure 74: Step 3: encode categorical variables. OneHotEncoder expands each categorical column into a set of binary indicator columns.

ColumnTransformer: Different Steps for Different Columns

Real datasets mix numeric and categorical features, and each type needs different preprocessing. Numeric columns need imputation and scaling; categorical columns need encoding. The `ColumnTransformer` applies different transformers to different subsets of columns.

Python Code: ColumnTransformer for Mixed Data

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

numeric_features     = ['age', 'income', 'balance']
categorical_features = ['sector', 'country']

preprocess = ColumnTransformer([
    ('num', StandardScaler(),    numeric_features),
    ('cat', OneHotEncoder(handle_unknown='ignore'),
       categorical_features),
])

pipe = Pipeline([
    ('prep',  preprocess),
    ('model', LogisticRegression(max_iter=1000)),
])

pipe.fit(X_train, y_train)
```

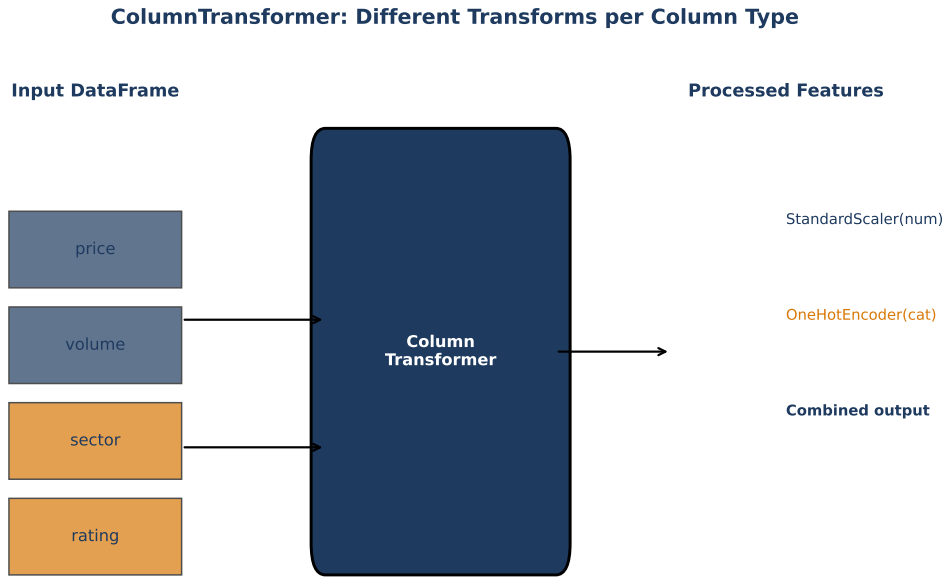


Figure 75: ColumnTransformer routes different columns to different transformers. Numeric columns go through a scaler; categorical columns go through an encoder. Results are concatenated.

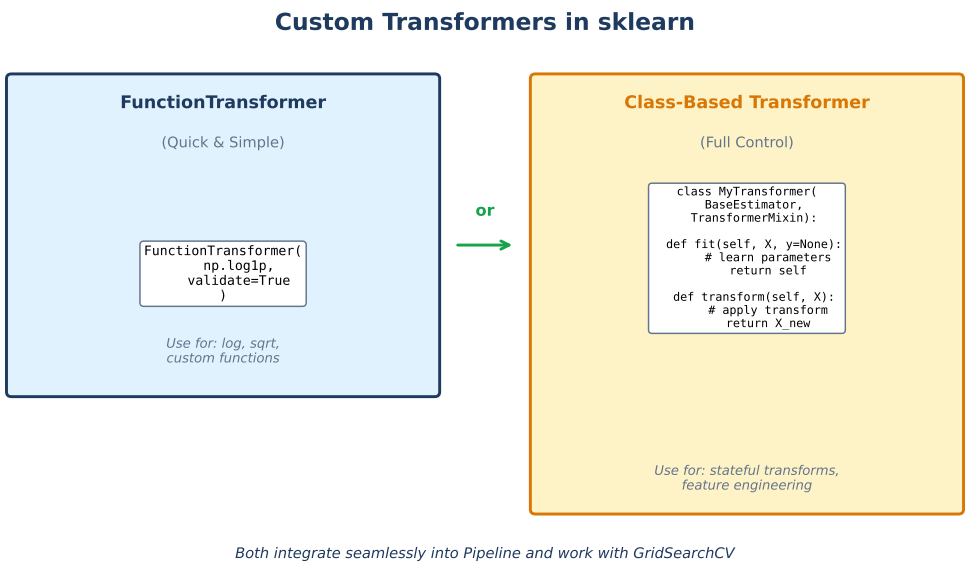


Figure 76: You can write custom transformers by subclassing BaseEstimator and TransformerMixin. This allows domain-specific preprocessing to live inside the pipeline.

Key Formula: Pipeline Semantics

Let a pipeline have transformers $T_1, T_2, \ldots, T_m$ and estimator E. During training on $(X_{\text{train}}, y_{\text{train}})$:

$$\begin{aligned} X^{(0)} &= X_{\text{train}} \\ X^{(k)} &= T_k.\text{fit\_transform}(X^{(k-1)}) \quad \text{for } k = 1, \ldots, m \\ & E.\text{fit}(X^{(m)}, y_{\text{train}}) \end{aligned}$$

During prediction on X_{test}:

$$\begin{aligned} X^{(0)} &= X_{\text{test}} \\ X^{(k)} &= T_k.\text{transform}(X^{(k-1)}) \quad \text{for } k = 1, \ldots, m \\ \hat{y} &= E.\text{predict}(X^{(m)}) \end{aligned}$$

The critical asymmetry: training uses `fit_transform`, prediction uses `transform` only. Training learns parameters; prediction reuses them.

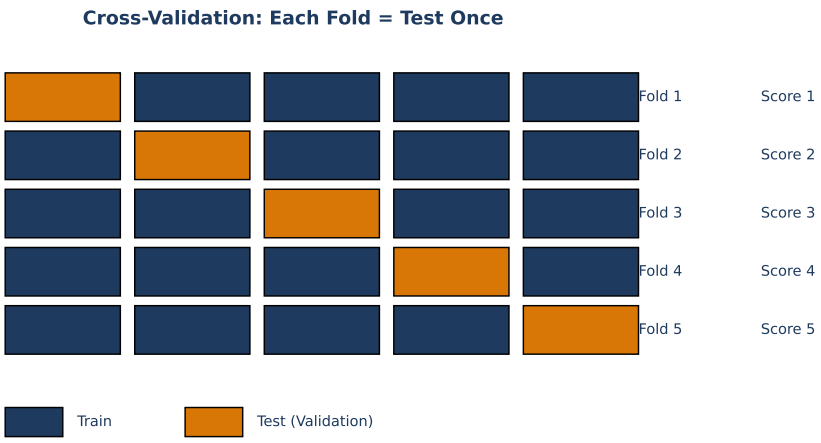


Figure 77: Cross-validation with a pipeline: each fold refits the entire pipeline on the fold's training subset. No preprocessing ever sees the fold's validation subset.

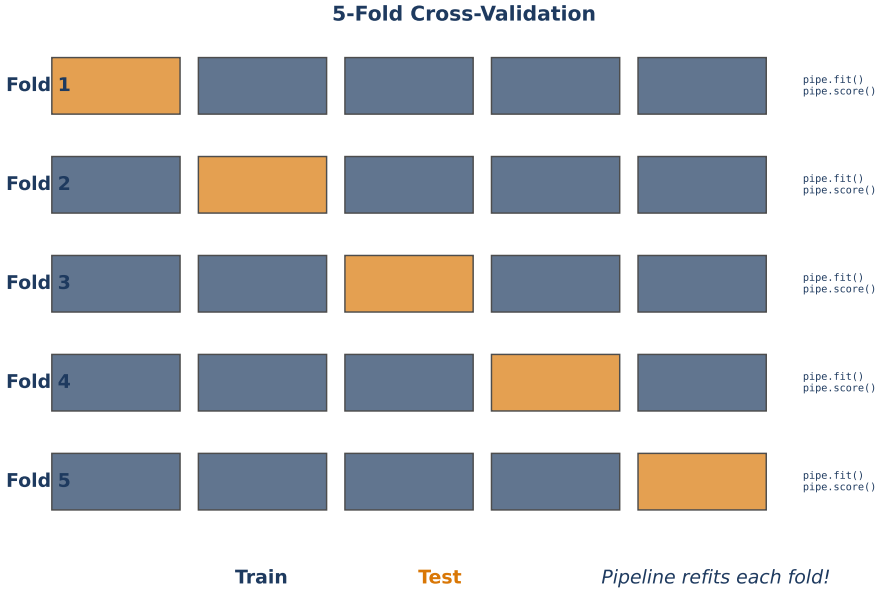


Figure 78: 5-fold cross-validation: the data is split into 5 folds; each fold serves once as the validation set while the remaining four train the pipeline.

Common Misconceptions about Pipelines and Leakage

(1) "You only need a pipeline for production." You need a pipeline from day one. Validation scores computed without a pipeline are untrustworthy because preprocessing in exploratory code typically leaks. If your development workflow is leaky, you cannot tell whether a model actually works.
(2) "Data leakage only happens between train and test." It also happens between cross-validation folds. If you scale the entire training set before running `cross_val_score`, every fold's validation subset contaminates the fold's training subset through the shared scaler. The pipeline fixes this because sklearn refits the pipeline on each fold separately.
(3) "`fit_transform` on the full dataset is fine as long as I split afterwards." No. That is the textbook leakage pattern. `fit_transform` uses statistics computed from the full dataset, including what will later become the test set. The test set influences the scaler, and the test score is contaminated.

Worked Examples

Worked Example 1: Spotting Leakage in a Code Snippet

A junior analyst writes:

```
X_scaled = StandardScaler().fit_transform(X)
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y)
```

Is this correct? Answer: No. The scaler is fit on the entire dataset `X`, which includes what will later become the test set. The test set influences the scaling parameters. Rewrite as a pipeline:

```
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)
score = pipe.score(X_test, y_test)
```

The pipeline ensures that the scaler is fit only on `X_train` and that the same parameters are applied (read-only) to `X_test`.

Worked Example 2: The Full Classification Pipeline

You are building a customer churn classifier for a telecom company. Features include `age`, `monthly_bill`, `months_with_company` (numeric), and `plan_type`, `payment_method` (categorical). Some numeric values are missing.

Full Pipeline for Mixed Data

```
numeric_features    = ['age', 'monthly_bill', '
    months_with_company']
categorical_features = ['plan_type', 'payment_method']

numeric_pipe = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler',  StandardScaler()),
])

categorical_pipe = Pipeline([
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('encoder', OneHotEncoder(handle_unknown='ignore')),
])

preprocess = ColumnTransformer([
    ('num', numeric_pipe,     numeric_features),
    ('cat', categorical_pipe, categorical_features),
])

full_pipe = Pipeline([
    ('prep',  preprocess),
    ('model', RandomForestClassifier(n_estimators=100)),
])

full_pipe.fit(X_train, y_train)
score = cross_val_score(full_pipe, X_train, y_train, cv=5).
    mean()
```

Notice how every preprocessing step lives inside the pipeline. When `cross_val_score` runs 5-fold CV, each fold refits the entire pipeline—imputers, scalers, encoders, and model—on its own training subset. Leakage is impossible.

Historical Background: David Cournapeau and the Birth of sklearn's Pipeline (2007–2010)

Before scikit-learn's `Pipeline` class existed, data scientists wrote preprocessing and modeling as separate scripts. A typical workflow involved four or five notebooks chained together: one for cleaning, one for feature engineering, one for scaling, one for model fitting, and one for evaluation. Keeping these in sync was painful. Refitting any step required rerunning every downstream notebook. Data leakage was routine because nobody consistently enforced the "fit on train only" discipline.

Scikit-learn began as a Google Summer of Code project in 2007, led by French engineer David Cournapeau. In its first release, sklearn already offered a common `fit`/`predict` interface for estimators—the interface that makes modern ML code so composable. But the real breakthrough for reproducibility came in the 0.7 and 0.8 releases around 2010, when the community introduced the `Pipeline` class. For the first time, preprocessing and modeling could be bundled as one object with consistent `fit`/`predict` semantics.

The impact was immediate. Workflows that used to span five notebooks shrank to five lines of code. Cross-validation, grid search, and model serialization became automatic. Data leakage between preprocessing and modeling became a solved problem—not by better discipline, but by making the right thing the easy thing. Before the Pipeline class, avoiding leakage required constant vigilance. After the Pipeline class, it required typing `Pipeline([...])`.

That is the highest form of software engineering: taking a source of bugs and turning it into an abstraction that makes the bug impossible. Every time you wrap your preprocessing in a Pipeline, you are standing on Cournapeau's shoulders.

Problem 7.1 (Easy)

Identify the leakage in this snippet:

```
X = pd.read_csv('data.csv')
X['age_scaled'] = (X['age'] - X['age'].mean()) / X['age'].std()
X_train, X_test = train_test_split(X)
```

Which line introduces leakage? Rewrite the snippet using a sklearn pipeline.
Solution: see Appendix.

Problem 7.2 (Easy)

Explain in one paragraph why calling `StandardScaler().fit_transform(X)` before `train_test_split(X, y)` violates the "fit on train only" discipline. What specific quantity gets contaminated?
Solution: see Appendix.

Problem 7.3 (Medium)

Write a sklearn Pipeline that chains: (1) median imputation for missing values, (2) standard scaling, (3) PCA keeping 5 components, (4) logistic regression classifier. Import all necessary classes. Your answer should be runnable code.
Solution: see Appendix.

Problem 7.4 (Medium)

A dataset has three numeric columns (`age`, `income`, `balance`) and two categorical columns (`state`, `job`). Write a ColumnTransformer specification that applies `StandardScaler` to the numeric columns and `OneHotEncoder` to the categorical columns. Handle unknown categories at prediction time gracefully.
Solution: see Appendix.

Problem 7.5 (Hard)

Design a nested cross-validation scheme that tunes hyperparameters on inner folds and estimates generalization performance on outer folds. Specifically: describe the data flow, state which sklearn classes you would use, and explain why a single-level CV that both tunes and reports scores is biased upward.
Solution: see Appendix.

Connecting Forward

We now have a pipeline that prevents leakage. But pipelines usually contain hyperparameters—the number of PCA components, the regularization strength of logistic regression, the maximum depth of a random forest. How do you choose these? You search. Section 8 covers grid search, random search, and the special case of time-series validation, where every other technique in this handout gets a nasty surprise: shuffling data corrupts any temporal ordering, and shuffled cross-validation pretends you can see the future. We will fix that with `TimeSeriesSplit` and walk-forward validation, closing the loop on the full unsupervised and supervised workflow.

Key Takeaway: A pipeline ensures that every preprocessing step is fitted only on training data—without it, information leaks from the future into the past and your model's performance is an illusion.

8. Tuning the Machine – Cross-Validation, Grid Search, and Production

Opening Problem: The Time Machine Fallacy

You are backtesting a trading strategy. Your pipeline takes daily market features and predicts next-day returns. You use a random forest classifier. You run 5-fold cross-validation and the average accuracy is 64%. You congratulate yourself, deploy the strategy to paper trading, and watch it lose money every week.

Something is wrong. The cross-validation score was clearly misleading, but your pipeline has no leakage. The scaler was fit only on training folds. The imputer was fit only on training folds. The random forest was trained only on training folds. Every discipline from Section 7 was followed. So why did the strategy fail?

The answer is that ordinary 5-fold cross-validation shuffles the data before splitting. After shuffling, each fold contains a mix of dates from across the full sample period. The training fold includes data from 2021 and 2023; the validation fold includes data from 2022. Your model was predicting April 2022 returns using features from July 2023. In other words, the validation setup quietly assumed you could see the future.

This section fixes that assumption. Time-series cross-validation—`TimeSeriesSplit`, walk-forward validation, and expanding windows—respects the temporal ordering of data. It also covers hyperparameter tuning via grid search and random search, and closes with production concerns: serialization, monitoring, and the habits that separate a working notebook from a deployed system.

Discovery Question

You tune a random forest with GridSearchCV and find the best parameters: `max_depth=7`, `n_estimators=200`. Your colleague tunes the same model with RandomizedSearchCV using only 50 iterations and finds `max_depth=8`, `n_estimators=180` with nearly identical performance. Did your exhaustive search waste time?

Grid Search: Exhaustive but Expensive

Every model has hyperparameters that are not learned from data—you must set them by hand. Random forests have `n_estimators`, `max_depth`, `min_samples_split`, and more. Logistic regression has `C` and the penalty type. SVMs have `C` and the kernel bandwidth. Setting these well matters: the difference between a default random forest and a tuned one can be 5 to 10 percentage points of accuracy.

Grid search is the simplest tuning method. You define a grid of values for each hyperparameter, train a model for every combination, and keep the combination that gives the best cross-validation score. If `max_depth` has 5 candidate values and `n_estimators` has 4 candidate values, grid search trains $5 \times 4 = 20$ models. Add a third hyperparameter with 3 values and the count jumps to 60.

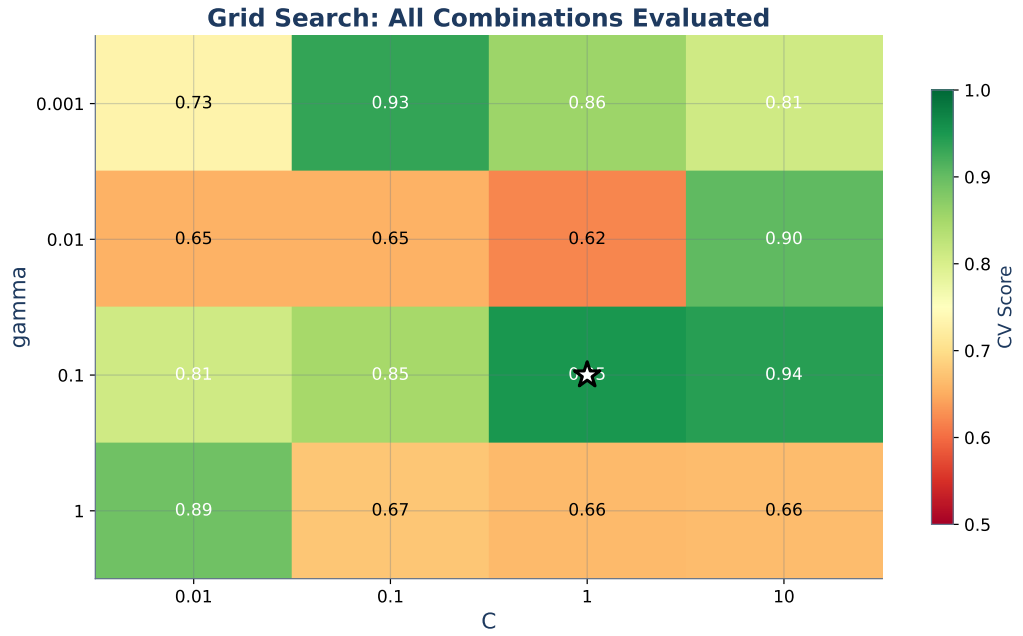


Figure 79: Grid search lays down a rectangular grid of hyperparameter combinations and evaluates every one. Exhaustive but combinatorial in cost.

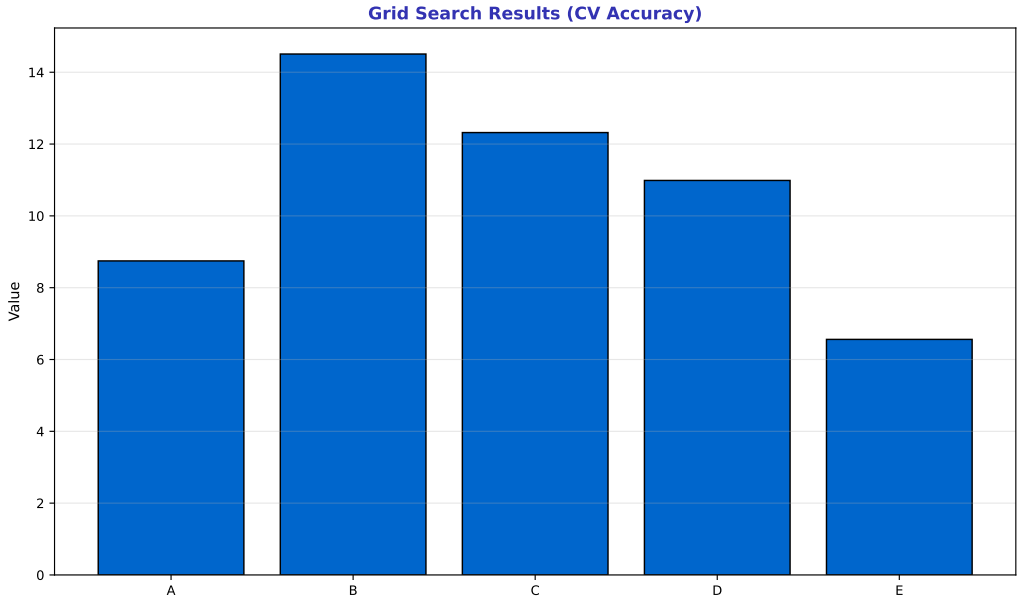


Figure 80: Grid search results as a heatmap: each cell shows the cross-validation score for one hyperparameter combination. The best cell becomes the chosen configuration.

Python Code: Grid Search in a Pipeline

```
from sklearn.model_selection import GridSearchCV

pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('model',  RandomForestClassifier(random_state=42)),
])

param_grid = {
    'model__n_estimators': [100, 200, 300],
    'model__max_depth':    [3, 5, 7, None],
}

search = GridSearchCV(pipe, param_grid, cv=5, scoring='accuracy',
    n_jobs=-1)
search.fit(X_train, y_train)

print(search.best_params_)
print(search.best_score_)
```

The double underscore `model__n_estimators` is sklearn's syntax for specifying a parameter of a named pipeline step. This syntax lets grid search tune any parameter of any step in a pipeline, including preprocessing parameters like `StandardScaler` options or `PCA(n_components)`.

Random Search: Smarter with a Fixed Budget

Grid search has a problem. If some hyperparameters matter a lot and others matter little, grid search wastes effort exploring the unimportant dimensions. A grid with 5 values for `max_depth` and 5 values for `min_samples_split` tests 25 combinations. But if `min_samples_split` is nearly irrelevant, you spent 80% of your budget varying a parameter that does not matter.

Random search samples hyperparameter combinations from a distribution instead. You give it a budget—say, 50 iterations—and it draws 50 random combinations and evaluates each. If `max_depth` matters, random search covers many values of it. If `min_samples_split` does not, random search still explores it, but only incidentally. The expected coverage of important dimensions is much better for a fixed budget.

A 2012 study by James Bergstra and Yoshua Bengio showed that random search with 60 iterations typically matches or beats exhaustive grid search, especially when hyperparameter importance is unequal. For most practical tuning tasks, random search is the default choice. Grid search is reserved for small, well-understood parameter spaces.

Hyperparameter: A model setting that is not learned from data but must be chosen by the analyst before training. Examples: the number of clusters K in K-Means, the regularization strength C in logistic regression, and the maximum tree depth in a random forest. Tuning hyperparameters is the task of finding the combination that produces the best validation performance.

Grid search: An exhaustive tuning strategy that trains one model per combination in a predefined grid of hyperparameter values. Cost grows multiplicatively in the number of parameters.

Random search: A tuning strategy that samples hyperparameter combinations from a distribution (uniform, log-uniform, or custom). Cost is fixed by the iteration budget, independent of the number of parameters.

Time Matters: TimeSeriesSplit

Standard K-fold cross-validation randomly assigns observations to folds. For tabular data drawn from an independent sample, that is fine. For time series, it is a disaster. Shuffled folds mix past

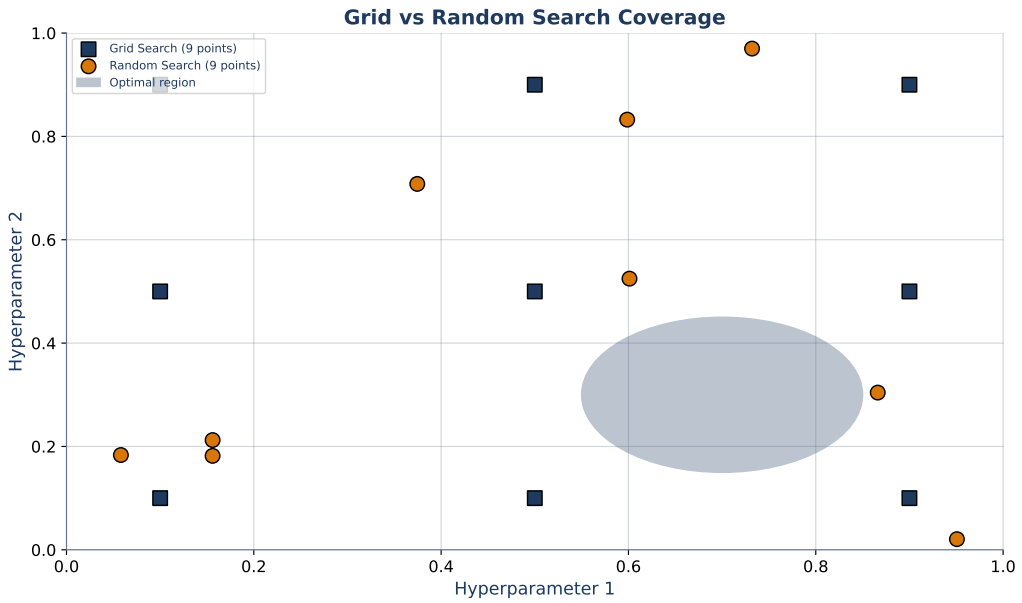


Figure 81: Grid search versus random search: same budget, same parameter space. Random search samples better coverage in each dimension, especially when some parameters matter more than others.

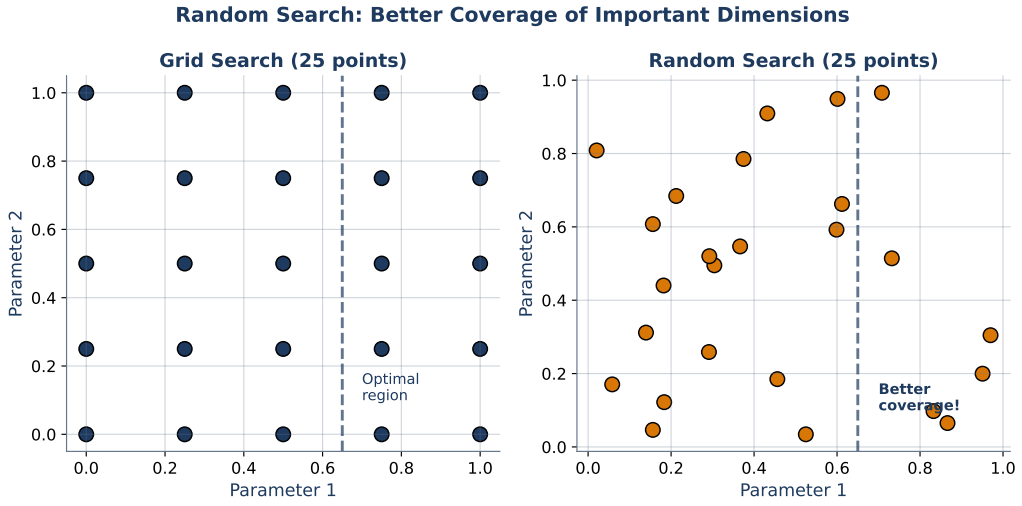


Figure 82: Why random search works: a fixed number of random samples covers the important dimensions more thoroughly than a rigid grid with the same budget.

and future, so the validation score measures performance in a world where you can peek ahead. In the real world you cannot, and deployed performance drops.

The fix is `TimeSeriesSplit`. It splits the data chronologically. The first fold trains on the earliest portion and validates on the next portion. The second fold trains on the first two portions combined and validates on the third. The pattern continues, expanding the training window each step. This is called *walk-forward validation* or *expanding window validation*, and it matches how the model will actually be deployed: trained on past data, evaluated on future data, never looking ahead.

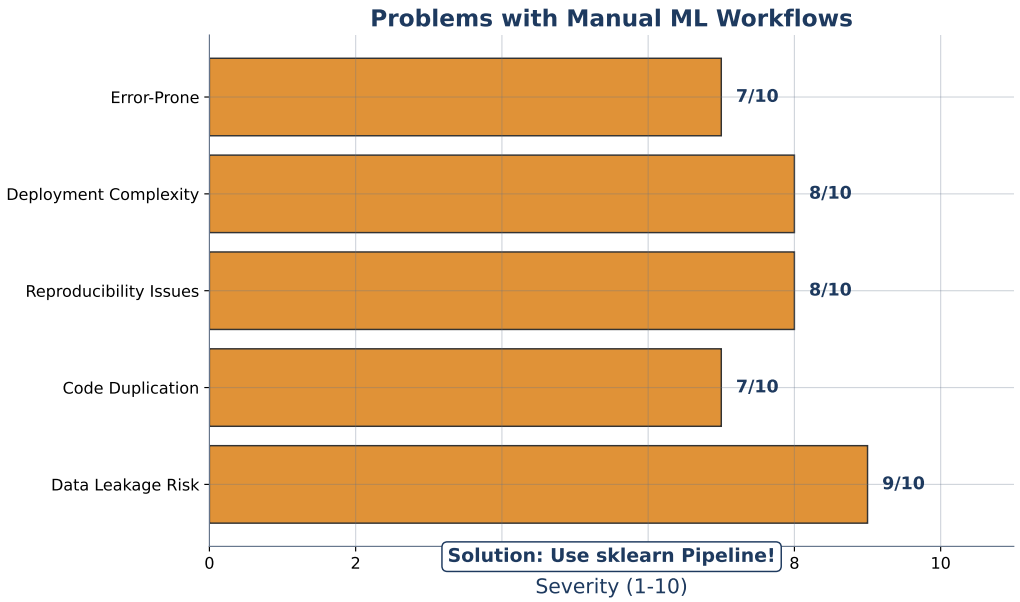
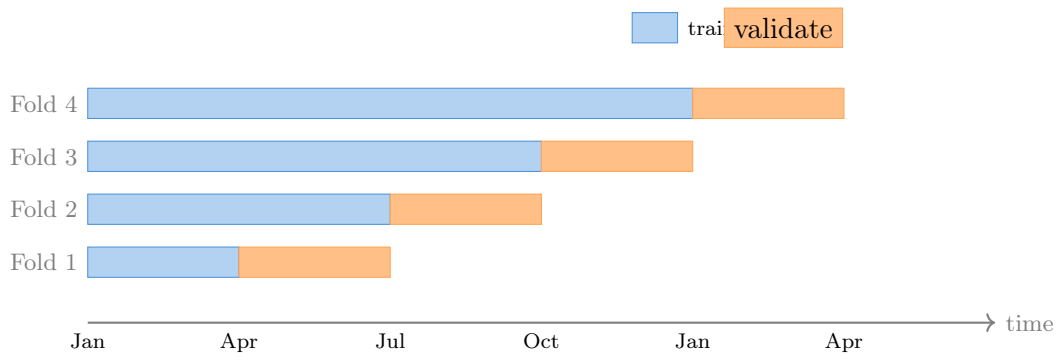


Figure 83: The problem with shuffled cross-validation on time series: validation folds contain dates that are earlier than the training dates. The model secretly sees the future.

A diagram makes the ordering explicit:



TimeSeriesSplit: A cross-validation strategy that respects chronological order. Training folds always precede validation folds in time, with no shuffling. Available in sklearn as `TimeSeriesSplit`.

Walk-forward validation: A validation scheme where the model is repeatedly retrained as new data arrives, simulating the deployment environment of a production system that updates over time.

Expanding vs Rolling Windows, and the Gap Parameter

Walk-forward validation has two common shapes. An *expanding window* grows the training set over time—fold k trains on everything up to time t_k and validates on the next block. A *rolling window* uses a fixed-size training set that slides forward, dropping old data as new data arrives.

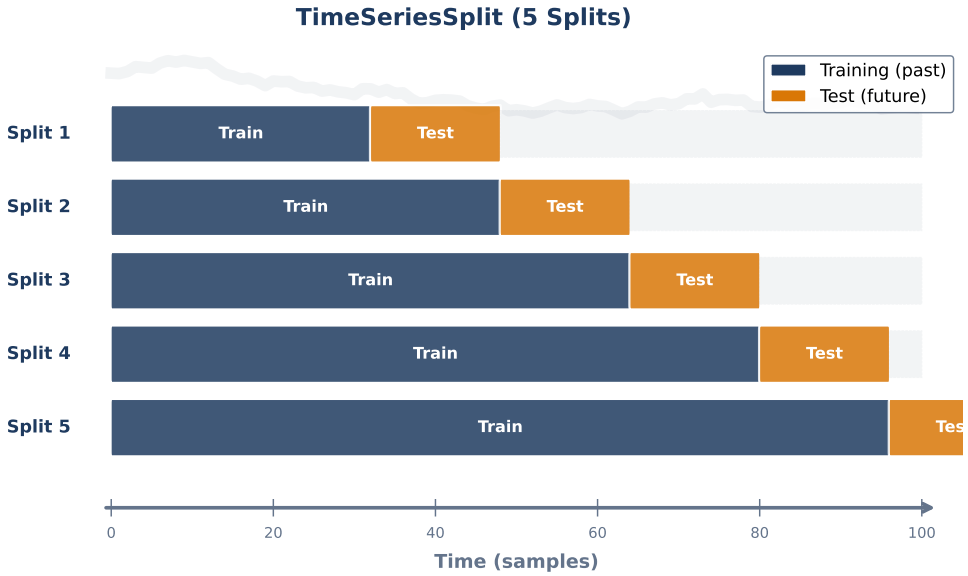


Figure 84: TimeSeriesSplit with 5 folds: each training set uses only data earlier than its validation set. Blue = train, orange = validate. The training window expands over time.

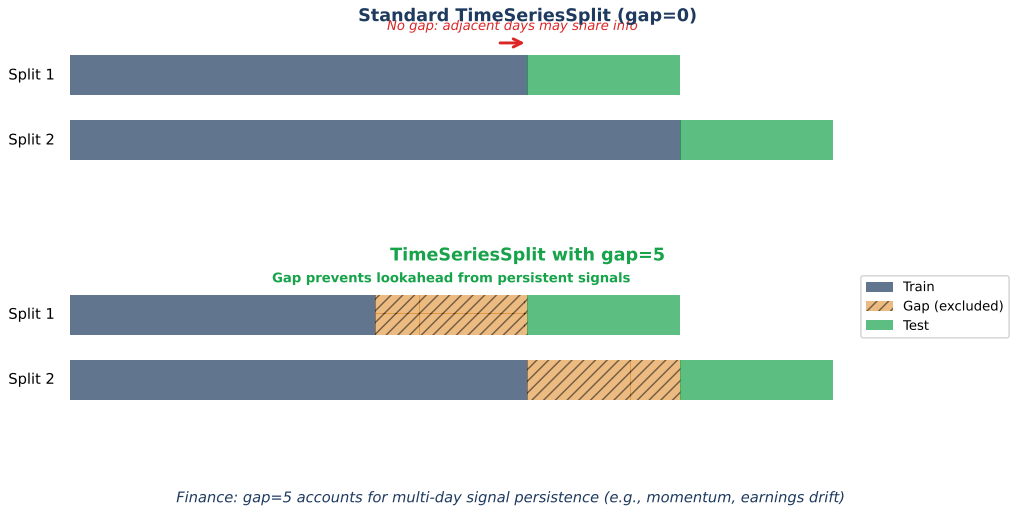


Figure 85: The gap parameter introduces a buffer between training and validation to avoid spillover from serial correlation or overlapping feature windows.

Expanding windows use all available history; rolling windows adapt faster to regime changes. Choose based on whether stability or adaptivity is more valuable.

The *gap parameter* inserts a buffer between the end of the training set and the start of the validation set. Why? Suppose your features are 20-day rolling averages. A training observation at day t and a validation observation at day $t+1$ share 19 days of input data. They are not independent, and the shared days contaminate the validation score. Inserting a 20-day gap between the last training day and the first validation day removes the overlap.

Key Formula: TimeSeriesSplit Fold k

With n observations, k folds, and gap g, fold i (for $i = 1, \ldots, k$) uses:

$$\text{train}_i = \{1, 2, \ldots, \tfrac{i \cdot n}{k+1}\}$$
$$\text{validate}_i = \{\tfrac{i \cdot n}{k+1} + g + 1, \ldots, \tfrac{(i+1) \cdot n}{k+1}\}$$

The training set grows by one block per fold (expanding window). The validation set is the next block, offset by the gap. With $g = 0$, the validation block begins immediately after training; with $g > 0$, there is a buffer.

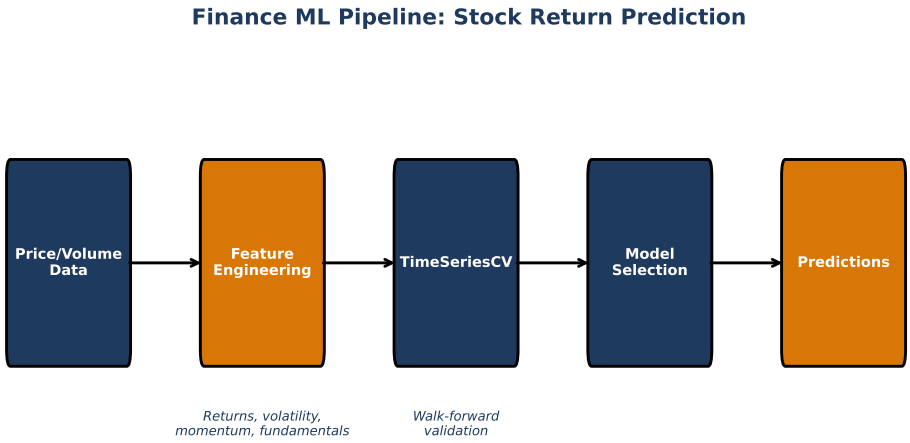


Figure 86: Finance application of walk-forward validation: a trading strategy is retrained each quarter on expanding (or rolling) windows, matching how it will be deployed live.

From Notebook to Production

A working notebook is not a production system. Production adds concerns that Jupyter does not enforce: serialization, input validation, monitoring, and retraining pipelines. A minimal production workflow looks like this:

1. **Serialize** the fitted pipeline with `joblib.dump(pipe, 'model.pkl')`.
2. **Validate inputs** at inference time: check that features are present, numeric, and within reasonable ranges. Reject requests with missing or malformed inputs.
3. **Monitor** performance over time. Track prediction distributions, input distributions, and (when labels eventually arrive) accuracy. Alert when any of these drift.
4. **Retrain** on new data at a regular cadence—daily, weekly, or monthly depending on how fast your environment changes.

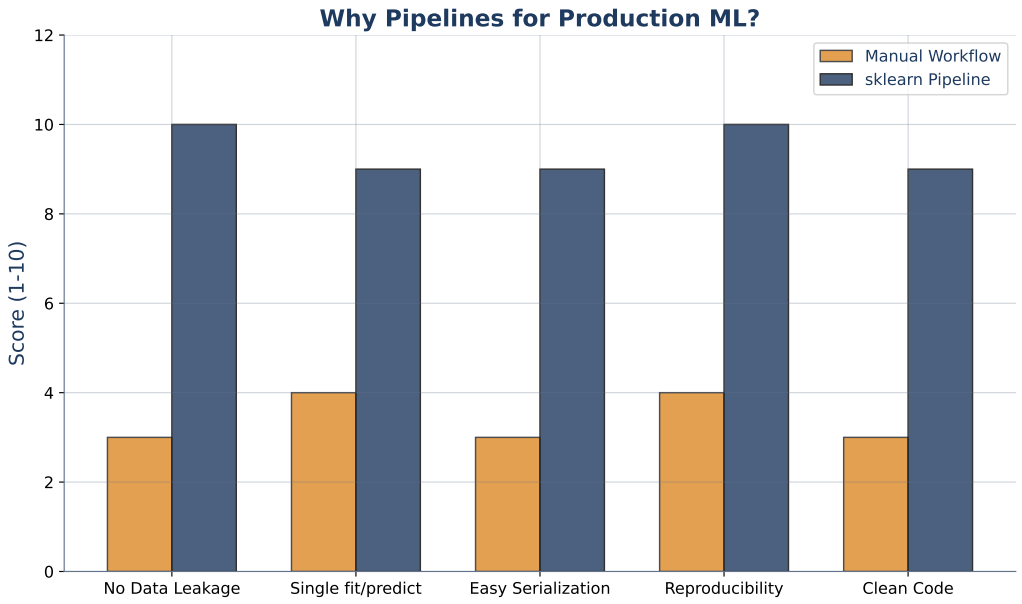


Figure 87: Why pipelines matter for production: a single object holds all preprocessing and model logic, making serialization, versioning, and retraining simple.

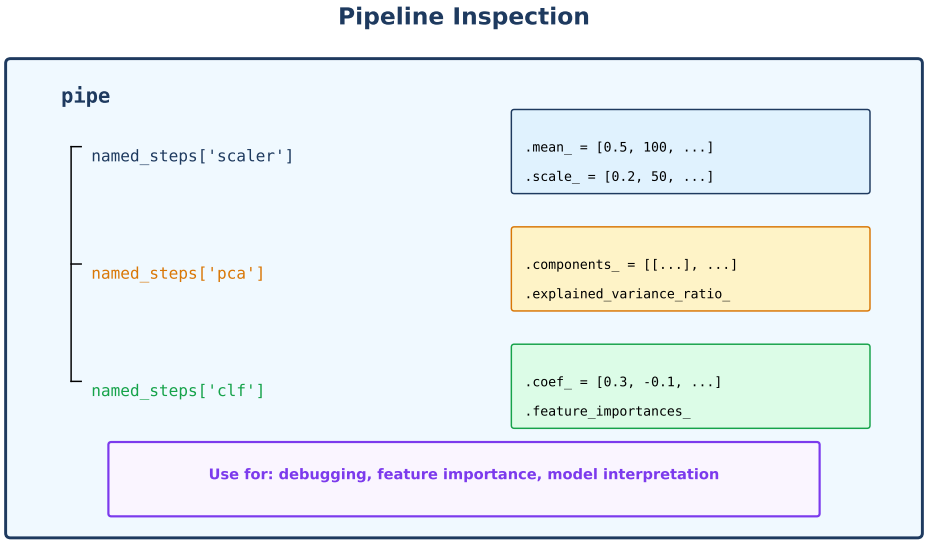


Figure 88: Inspecting a fitted pipeline: you can query individual steps, extract learned parameters, and plot feature importances without breaking the abstraction.

Common Misconceptions about Tuning and Validation

(1) "More hyperparameter combinations means better tuning." Exhaustive grid search has diminishing returns. Bergstra and Bengio's random search paper (2012) showed that 60 random samples typically match the best grid search on a comparable budget. After a point, you are burning compute for marginal gains.
(2) "Cross-validation works the same way for time series." Standard K-fold shuffles the data and ignores time. For stationary tabular data, that is fine. For time series, shuffling creates validation folds that precede training folds, so the validation score pretends you can see the future. Always use `TimeSeriesSplit` or walk-forward validation for temporal data.
(3) "A pipeline that works in a notebook works in production." Production requires serialization, input validation, monitoring, and retraining. A notebook that runs end-to-end is a starting point, not a finished system. Plan for the surrounding infrastructure from the beginning.

Worked Examples

Worked Example 1: Random Search with Log-Uniform Sampling

You are tuning a support vector classifier. The regularization parameter `C` can reasonably range from 10^{-3} to 10^{3}—six orders of magnitude. A uniform grid on this range would oversample the high end. Use log-uniform sampling instead:

Random Search with Log-Uniform C

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import loguniform

pipe = Pipeline([('scaler', StandardScaler()), ('svc', SVC()
    )])

param_dist = {
    'svc__C':     loguniform(1e-3, 1e3),
    'svc__gamma': loguniform(1e-4, 1e1),
}

search = RandomizedSearchCV(pipe, param_dist, n_iter=50, cv
    =5,
                            random_state=42, n_jobs=-1)
search.fit(X_train, y_train)
print(search.best_params_)
```

With `n_iter=50`, random search samples 50 combinations from the log-uniform distributions. Each draw is equally likely on a log scale, so $C = 0.01$ and $C = 1.0$ have equal probability—a sensible default for scale-free parameters.

Worked Example 2: Walk-Forward Validation on Monthly Returns

You have 60 months of financial data and want to validate a strategy with a 6-month gap between training and validation. Use `TimeSeriesSplit` with a custom gap:

Walk-Forward with Gap

```
from sklearn.model_selection import TimeSeriesSplit

tscv = TimeSeriesSplit(n_splits=5, gap=6)

for fold, (train_idx, val_idx) in enumerate(tscv.split(X)):
    X_train, X_val = X.iloc[train_idx], X.iloc[val_idx]
    y_train, y_val = y.iloc[train_idx], y.iloc[val_idx]

    pipe.fit(X_train, y_train)
    score = pipe.score(X_val, y_val)
    print(f'Fold {fold}: train months = {len(train_idx)}, '
          f'val months = {len(val_idx)}, score = {score:.3f}
              ')
```

With 60 months, 5 splits, and a 6-month gap, each training fold grows by 10 months, the validation block is 10 months, and the gap inserts a 6-month buffer to prevent spillover from overlapping rolling-window features. Every validation score reflects out-of-sample, forward-looking performance.

Historical Background: Arlot and Celisse's Survey of Cross-Validation (2010)

In 2010, two French statisticians—Sylvain Arlot of Universite Paris-Sud and Alain Celisse of Universite Lille 1—published *A Survey of Cross-Validation Procedures for Model Selection* in Statistics Surveys. The paper is a methodical catalogue of every cross-validation variant used in statistics and machine learning: leave-one-out, K-fold, repeated K-fold, hold-out, Monte Carlo CV, time-series CV, and more. For each, they derived theoretical properties: bias, variance, consistency, and optimal choices for the fold count K.

Before this survey, cross-validation was a folklore topic. Practitioners knew that $K = 5$ or $K = 10$ "works," but the justification was rarely spelled out. Arlot and Celisse changed that by writing down the statistical properties of each method with full mathematical rigor. They showed that leave-one-out CV has low bias but high variance for unstable models. They showed that K-fold with moderate K balances bias and variance well. They established that time-series CV requires a modified asymptotic analysis because observations are not independent.

Every ML engineer who writes `cross_val_score` today is implicitly relying on the guarantees Arlot and Celisse catalogued. If you ever wonder why $K = 5$ is the usual default, or why leave-one-out CV can be unreliable for small datasets, or why shuffled CV is wrong for time series, the answers are in their survey. Arlot and Celisse catalogued every CV variant and their statistical properties—their survey is why we know exactly when K-fold, LOOCV, and walk-forward each shine.

Problem 8.1 (Easy)

A grid search tested 4 values of `max_depth` and 5 values of `n_estimators` with 5-fold CV. How many model fits did it perform? If each fit takes 3 seconds, how long did the full grid search take?
Solution: see Appendix.

Problem 8.2 (Easy)

Explain in two or three sentences why `TimeSeriesSplit` uses expanding or rolling windows instead of random folds. What specific error does random K-fold introduce when applied to a time series?
Solution: see Appendix.

Problem 8.3 (Medium)

Set up a `RandomizedSearchCV` that tunes the regularization parameter `C` of `LogisticRegression` over a log-uniform distribution from 10^{-4} to 10^{2}, with 30 iterations and 5-fold CV. Wrap the model in a pipeline with `StandardScaler`. Write complete runnable code.
Solution: see Appendix.

Problem 8.4 (Medium)

You have 120 months of trading data. Design a walk-forward validation with 6 folds and a 2-month gap. How many months go into each training fold? How many into each validation fold? Sketch the layout schematically.
Solution: see Appendix.

Problem 8.5 (Hard)

Compare grid search, random search, and Bayesian optimization on three axes: (1) computational complexity as a function of the number of hyperparameters and budget, (2) coverage of the parameter space under a fixed budget, and (3) the scenarios in which each is the best choice. Cite at least one reference.
Solution: see Appendix.

Closing the Loop

This section concludes the main body of the handout. Sections 1 through 6 introduced the tools of unsupervised learning—K-Means, hierarchical clustering, PCA—and Sections 7 and 8 showed how to use those tools correctly: inside pipelines, with honest cross-validation, and with hyperparameters tuned on historical data that never bleeds into the future.

The appendix that follows contains complete solutions for all 40 practice problems. Solve each problem yourself before reading the solution. The pedagogy is more effective when the struggle comes first and the answer second. Skim the solutions only to verify your understanding or to unblock a step that you truly cannot see.

Key Takeaway: For time series, always validate forward in time with a gap between training and test—standard cross-validation pretends you can see the future.

A. Solutions to Practice Problems

Section 1: Sorting Without Labels

Problem 1.1 (Easy). (a) Supervised—predicts a continuous target (price) from input features. (b) Unsupervised—no target variable; the goal is to discover groups. (c) Supervised—uses labeled training data (spam/not spam) to learn a classifier. (d) Unsupervised—PCA reduces dimensionality without a target. (e) Supervised—predicts a binary target (default/no default) from credit score.

Problem 1.2 (Easy). Income ranges from \$20,000 to \$500,000 (spread of \$480,000), while age ranges from 18 to 80 (spread of 62). K-Means uses Euclidean distance, so income will dominate because differences in dollars are orders of magnitude larger than differences in years. A difference of \$100,000 in income dwarfs a difference of 20 years in age in raw units. To fix this, standardize both features (subtract the mean, divide by the standard deviation) before clustering so that both contribute equally to the distance calculation.

Problem 1.3 (Medium). Distances between all pairs:

	A	B	C	D	E	F
A	0	0.54	7.21	9.22	1.40	12.04
B	–	0	7.28	8.73	1.28	11.63
C	–	–	0	3.00	8.54	5.00
D	–	–	–	0	10.14	3.16
E	–	–	–	–	0	13.18
F	–	–	–	–	–	0

Computations: $d(A,B) = \sqrt{(1.5-1)^2 + (1.8-2)^2} = \sqrt{0.25 + 0.04} = \sqrt{0.29} \approx 0.54$. Continuing this for all pairs, the smallest distance is $d(A,B) \approx 0.54$, so A and B merge first.

Problem 1.4 (Medium). After standardization, height has mean 0 and std 1. Weight remains in kg (mean $\approx$ 70, std $\approx$ 15). The Euclidean distance will be dominated by weight because its scale is much larger. A weight difference of 30 kg contributes $30^2 = 900$ to the squared distance, while a height difference of 1 standard deviation contributes just $1^2 = 1$. The clustering will effectively use weight alone, ignoring height entirely. This produces meaningless clusters driven by a single feature.

Problem 1.5 (Hard). (a) Preprocessing: (1) Handle missing values via median imputation (robust to outliers). (2) Standardize all features with `StandardScaler` because account balance (e.g., \$10,000) and credit score (300–850) are on vastly different scales. (3) Check for outliers and consider winsorizing extreme values.

(b) Algorithm: Start with K-Means because it is fast, scalable to 100,000 observations, and interpretable. Run it for $K = 2, 3, \ldots, 10$ and use the elbow method and silhouette scores to choose K.

(c) Evaluation: Use silhouette score (higher is better, range $[-1, 1]$; above 0.5 is strong). Inspect cluster centroids to check interpretability. Visualize in 2D with PCA or t-SNE to see whether clusters are separated.

(d) Presentation: Create a table of cluster profiles showing the mean of each feature per cluster. Name the clusters descriptively (e.g., "High-Balance Savers," "Young Active Spenders"). Show a 2D scatter plot colored by cluster. Translate statistical results into business actions: "Cluster 3 has high income but low savings—consider cross-selling investment products."

Section 2: K-Means Clustering

Problem 2.1 (Easy). Looking at the elbow plot, the inertia (WCSS) drops sharply from $K = 1$ to $K = 3$, then flattens. The elbow is at $K = 3$. Justification: adding a 4th cluster reduces

inertia by a small amount relative to the drop from $K = 2$ to $K = 3$. The marginal benefit of additional clusters is no longer worth the added complexity.

Problem 2.2 (Easy). The silhouette coefficient for one point i is:

$$s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))}$$

where $a(i)$ is the mean distance from i to all other points in i's cluster, and $b(i)$ is the mean distance from i to all points in the nearest neighboring cluster. Suppose $a(i) = 1.5$ (average intra-cluster distance) and $b(i) = 4.0$ (average distance to nearest other cluster). Then $s(i) = (4.0 - 1.5)/\max(1.5, 4.0) = 2.5/4.0 = 0.625$. This is a good silhouette score—the point is well-clustered.

Problem 2.3 (Medium). Given 8 points and $K = 2$, initialize centroids at random positions (e.g., $c_1 = (1, 2)$ and $c_2 = (6, 7)$).

Iteration 1 – Assign: Compute the Euclidean distance from each point to both centroids. Assign each point to the closest centroid. Points near $(1, 2)$ go to Cluster 1; points near $(6, 7)$ go to Cluster 2.

Iteration 1 – Update: Recompute each centroid as the mean of all points assigned to it. If Cluster 1 has points $\{(0, 1), (1, 2), (2, 1)\}$, then $c_1^{\text{new}} = (1.0, 1.33)$. Similarly update c_2.

Iteration 2 – Assign: Re-assign all points using the new centroids. Some points near the boundary may switch clusters.

Iteration 2 – Update: Recompute centroids again. If no points switched, the algorithm has converged.

The key observation is that inertia (total squared distance) decreases at each step and converges in a finite number of iterations.

Problem 2.4 (Medium). Elbow plot A shows a sharp bend at $K = 3$: inertia drops rapidly from $K = 1$ to $K = 3$, then flattens. The elbow is unambiguous, and $K = 3$ is the clear choice.

Elbow plot B shows a gradual, smooth decay with no visible bend. The elbow is ambiguous. In this case, the elbow method alone is insufficient. You should compute silhouette scores for each K and pick the K with the highest average silhouette score. If silhouette scores are also ambiguous, consider domain knowledge or hierarchical clustering to see a richer picture of the data structure.

Problem 2.5 (Hard). *Proof that K-Means converges:*

Define the objective function (inertia): $J = \sum_{k=1}^{K} \sum_{x_i \in C_k} \|x_i - \mu_k\|^2$.

Assignment step: Each point is assigned to the closest centroid. This cannot increase J because switching a point to a closer centroid reduces its contribution to J, and the contributions of all other points remain unchanged. Therefore $J_{\text{after assign}} \leq J_{\text{before assign}}$.

Update step: Each centroid is recomputed as the mean of its assigned points. The mean minimizes the sum of squared distances within a set, so $J_{\text{after update}} \leq J_{\text{after assign}}$.

Combining: J is non-increasing at each step. Since $J \geq 0$ (sum of squared distances is non-negative) and there are finitely many possible partitions of n points into K clusters, the sequence of J values must converge in a finite number of iterations. □

Note: convergence is to a local minimum, not necessarily the global minimum. Different initializations can produce different final partitions.

Section 3: Hierarchical Clustering and Dendrograms

Problem 3.1 (Easy). A horizontal line at distance 5 cuts through all branches of the dendrogram that have a merge height above 5. Count the number of branches that the horizontal line intersects—that is the number of clusters. For example, if the dendrogram has merge heights at

2, 3, 6, and 9, cutting at 5 separates the merges at 6 and 9 (above the cut) from the merges at 2 and 3 (below the cut), producing 3 clusters.

Problem 3.2 (Easy). (1) **Single linkage:** Distance between two clusters is the minimum distance between any pair of points across the two clusters (nearest neighbors). Tends to produce elongated, chain-like clusters. (2) **Complete linkage:** Distance is the maximum pairwise distance between clusters (farthest neighbors). Tends to produce compact, spherical clusters. (3) **Average linkage:** Distance is the mean of all pairwise distances between clusters. A compromise between single and complete. (4) **Ward linkage:** Distance is the increase in total within-cluster variance caused by merging two clusters. Tends to produce equally sized, spherical clusters.

Problem 3.3 (Medium). Given 5 points with distance matrix:

	A	B	C	D	E
A	0	2	6	10	9
B	–	0	5	9	8
C	–	–	0	4	5
D	–	–	–	0	3
E	–	–	–	–	0

Step 1: Find the smallest distance. $d(A, B) = 2$ is the minimum. Merge A and B into cluster $\{A, B\}$.

Update the distance matrix using single linkage: $d(\{A, B\}, C) = \min(d(A, C), d(B, C)) = \min(6, 5) = 5$. Similarly: $d(\{A, B\}, D) = \min(10, 9) = 9$, $d(\{A, B\}, E) = \min(9, 8) = 8$.

The updated matrix has 4 entities: $\{A, B\}$, C, D, E. The next smallest distance is $d(D, E) = 3$, so D and E merge next. And so on.

Problem 3.4 (Medium). Single linkage: The dendrogram shows chaining—clusters merge at increasing but closely spaced heights. Non-spherical, elongated clusters are captured well, but the dendrogram has no clear gap for cutting.

Ward linkage: The dendrogram shows a cleaner tree with large gaps between merge heights. Clusters are more compact and equally sized. The gap between the last 2 and last 3 merges is typically much larger, making the number of clusters easier to choose.

The difference arises because single linkage only requires one close pair to trigger a merge, while Ward linkage considers the impact on total cluster variance. Ward resists merging dissimilar clusters, creating larger gaps in the dendrogram.

Problem 3.5 (Hard). Ward linkage minimizes the increase in total within-cluster variance. When merging clusters A (size n_A, centroid μ_A) and B (size n_B, centroid μ_B), the Ward distance is:

$$d_{\text{Ward}}(A, B) = \frac{n_A \cdot n_B}{n_A + n_B} \|\mu_A - \mu_B\|^2$$

Derivation: Let $V_A = \sum_{x \in A} \|x - \mu_A\|^2$ and $V_B = \sum_{x \in B} \|x - \mu_B\|^2$ be the within-cluster variances. After merging, the combined centroid is $\mu_{AB} = \frac{n_A \mu_A + n_B \mu_B}{n_A + n_B}$. The new within-cluster variance is:

$$V_{AB} = \sum_{x \in A \cup B} \|x - \mu_{AB}\|^2$$

Using the decomposition of sum of squares:

$$V_{AB} = V_A + V_B + \frac{n_A n_B}{n_A + n_B} \|\mu_A - \mu_B\|^2$$

The increase in variance from merging is $\Delta V = V_{AB} - V_A - V_B = \frac{n_A n_B}{n_A + n_B} \|\mu_A - \mu_B\|^2$. Ward linkage picks the merge that minimizes this ΔV. □

Section 4: Linkage, Correlation Clustering, and HRP

Problem 4.1 (Easy). Given correlations: $\rho_{AB} = 0.8$, $\rho_{AC} = 0.2$, $\rho_{BC} = -0.5$
Distances: $d_{AB} = 1 - |0.8| = 0.2$, $d_{AC} = 1 - |0.2| = 0.8$, $d_{BC} = 1 - |-0.5| = 0.5$.
Distance matrix:

	A	B	C
A	0	0.2	0.8
B	–	0	0.5
C	–	–	0

A and B have the smallest distance (0.2), meaning they are most similar (highly correlated). They merge first in the dendrogram.

Problem 4.2 (Easy). The two assets that merge first in the dendrogram are those with the smallest correlation distance. This means their returns are most correlated—they move together most closely. In financial terms, they likely belong to the same sector or are exposed to the same risk factor.

Problem 4.3 (Medium). The dendrogram merges B and C first, then A joins. Apply HRP via inverse-variance allocation:
Step 1 – Allocate within the $\{B, C\}$ cluster by inverse variance: $w_B^{\text{raw}} = 1/\sigma_B^2 = 1/0.16 = 6.25$, $w_C^{\text{raw}} = 1/\sigma_C^2 = 1/0.01 = 100$. Normalized: $w_B = 6.25/106.25 = 0.059$, $w_C = 100/106.25 = 0.941$. Cluster $\{B, C\}$ variance: $\sigma_{BC}^2 \approx 0.059^2 \times 0.16 + 0.941^2 \times 0.01 \approx 0.0006 + 0.0089 = 0.0094$.
Step 2 – Allocate between A and cluster $\{B, C\}$ by inverse variance: $w_A^{\text{raw}} = 1/0.04 = 25$, $w_{BC}^{\text{raw}} = 1/0.0094 \approx 106.4$. Normalized: $w_A = 25/131.4 = 0.190$, $w_{BC} = 106.4/131.4 = 0.810$.
Final weights: $w_A = 0.190$, $w_B = 0.810 \times 0.059 = 0.048$, $w_C = 0.810 \times 0.941 = 0.762$.
The low-variance asset C receives the largest weight. This illustrates HRP's core logic: allocate more to less risky assets.

Problem 4.4 (Medium). Markowitz optimization inverts the covariance matrix. Matrix inversion amplifies estimation error—small errors in correlations produce large errors in inverse entries, which produce extreme, unstable portfolio weights. With 50 assets, the covariance matrix has $50 \times 49/2 = 1{,}225$ correlations to estimate, and each is noisy. The resulting weights can flip sign with minor changes in data.
HRP avoids matrix inversion entirely. It uses the dendrogram to impose a hierarchical structure on the allocation, then allocates via inverse variance within and between branches. The dendrogram is estimated from the same data, but it is a simpler, more robust structure than a full inverse covariance matrix. The result: HRP weights change slowly with new data and produce more stable out-of-sample performance.

Problem 4.5 (Hard).

HRP Pseudocode

```
def HRP(returns):
    # Step 1: Correlation distance matrix
    corr = returns.corr()
    dist = sqrt(0.5 * (1 - corr))

    # Step 2: Hierarchical clustering
    Z = linkage(squareform(dist), method='ward')

    # Step 3: Quasi-diagonalize covariance matrix
    ordered = get_quasi_diag(Z)  # reorder leaves
    cov = returns.cov().iloc[ordered, ordered]

    # Step 4: Recursive bisection
    weights = recursive_bisect(cov, items=ordered)
    return weights

def recursive_bisect(cov, items):
    if len(items) == 1:
        return {items[0]: 1.0}  # base case

    # Split items into two halves
    mid = len(items) // 2
    left, right = items[:mid], items[mid:]

    # Allocate between left and right by inverse cluster variance
    var_left = get_cluster_var(cov, left)
    var_right = get_cluster_var(cov, right)
    alpha = 1 - var_left / (var_left + var_right)

    # Recurse
    w_left = recursive_bisect(cov, left)
    w_right = recursive_bisect(cov, right)

    # Scale by alpha
    for k in w_left:  w_left[k]  *= alpha
    for k in w_right: w_right[k] *= (1 - alpha)
    return {**w_left, **w_right}
```

The base case is a single asset (weight 1). The recursive case splits the ordered asset list in half, computes the inverse-variance share for each half, and scales the sub-weights accordingly.

Section 5: PCA and Dimensionality Reduction

Problem 5.1 (Easy). The covariance matrix has $\Sigma_{12} = \Sigma_{21} = 3$, indicating strong positive correlation between features 1 and 2, and $\Sigma_{13} = \Sigma_{23} = 0$, indicating feature 3 is independent of the other two. PC1 will point in the direction of maximum variance, which lies in the 1-2 plane along the diagonal (roughly the direction $(1, 1, 0)/\sqrt{2}$) because features 1 and 2 co-vary. PC1 is not aligned with any single original axis—it is a combination of features 1 and 2. Feature 3, being independent with variance 1, will likely form its own principal component.

Problem 5.2 (Easy). **Centering** subtracts the column mean from each feature so that every feature has mean zero. Centering is mandatory for PCA because the eigendecomposition operates on deviations from the mean.

Standardizing additionally divides by the standard deviation so that every feature has unit

variance. Standardizing is optional but strongly recommended when features are on different scales (e.g., dollars and years).

You would skip standardization when all features are already on the same scale—for example, daily stock returns across 50 stocks, where every column is already in the same units (percentage daily change).

Problem 5.3 (Medium). Total variance $= 4.2 + 1.8 + 0.5 + 0.3 + 0.2 = 7.0$.

(a) Explained variance ratios: $4.2/7.0 = 0.600$, $1.8/7.0 = 0.257$, $0.5/7.0 = 0.071$, $0.3/7.0 = 0.043$, $0.2/7.0 = 0.029$.

(b) Cumulative up to 3 components: $0.600 + 0.257 + 0.071 = 0.929 = 92.9\%$.

(c) Minimum components for $\geq 90\%$: after 2 components, cumulative $= 85.7\%$ (not enough). After 3 components, cumulative $= 92.9\%$ (sufficient). Answer: 3 components.

Problem 5.4 (Medium). The points $(1, 2)$, $(2, 4)$, $(-1, -2)$, $(-2, -4)$ all lie on the line $y = 2x$. The direction of this line is $(1, 2)/\sqrt{5}$. Since all variance is along this line, PC1 $= (1/\sqrt{5}, 2/\sqrt{5})$.

Projection of $(1, 2)$ onto PC1: $z_1 = (1)(1/\sqrt{5}) + (2)(2/\sqrt{5}) = 5/\sqrt{5} = \sqrt{5} \approx 2.236$.

Reconstruction: $\hat{x} = z_1 \cdot v_1 = \sqrt{5} \cdot (1/\sqrt{5}, 2/\sqrt{5}) = (1, 2)$. The reconstruction error is $\|(1, 2) - (1, 2)\| = 0$.

Since all points lie exactly on a line, the data is perfectly 1-dimensional and PC2 captures zero variance. Keeping only PC1 loses no information—reconstruction error is zero for every point.

Problem 5.5 (Hard). Let A be a symmetric real matrix with $Av_1 = \lambda_1 v_1$ and $Av_2 = \lambda_2 v_2$ where $\lambda_1 \neq \lambda_2$.

Consider $v_1^\top A v_2$. Because $Av_2 = \lambda_2 v_2$:

$$v_1^\top A v_2 = v_1^\top (\lambda_2 v_2) = \lambda_2 (v_1^\top v_2) \quad \text{(i)}$$

Now use the symmetry of A: $A = A^\top$, so $v_1^\top A = (A^\top v_1)^\top = (Av_1)^\top = (\lambda_1 v_1)^\top = \lambda_1 v_1^\top$. Therefore:

$$v_1^\top A v_2 = \lambda_1 v_1^\top v_2 = \lambda_1 (v_1^\top v_2) \quad \text{(ii)}$$

Subtracting (i) from (ii): $(\lambda_1 - \lambda_2)(v_1^\top v_2) = 0$. Since $\lambda_1 \neq \lambda_2$, we must have $v_1^\top v_2 = 0$. The eigenvectors are orthogonal. $\square$

Symmetry was used in the step $v_1^\top A = (Av_1)^\top$, which holds only when $A = A^\top$.

Section 6: Scree Plots, Loadings, and Factor Extraction

Problem 6.1 (Easy). Eigenvalues: $[3.2, 1.4, 1.1, 0.6, 0.4, 0.2, 0.1]$. Total $= 7.0$. Kaiser criterion retains components with eigenvalue > 1: components 1, 2, and 3 (eigenvalues 3.2, 1.4, 1.1).

Variance explained: $(3.2 + 1.4 + 1.1)/7.0 = 5.7/7.0 = 81.4\%$.

Problem 6.2 (Easy). PC1 has large positive loadings only on valuation ratios: price-to-earnings, price-to-book, and price-to-sales. PC1 is most likely a **valuation factor**. Companies with high PC1 scores have high valuation multiples (expensive stocks); companies with low PC1 scores have low multiples (cheap stocks). In financial terms, PC1 captures the growth-versus-value dimension.

Problem 6.3 (Medium). To understand what separates US from European companies in PC1-PC2 space, look at the loading arrows that point in the direction of the US cluster (upper right) and away from the European cluster (lower left). These are the features that most differentiate the two groups.

Candidates: (1) market capitalization (US companies tend to be larger), (2) revenue growth (US tech firms often grow faster), (3) R&D spending as a percentage of revenue. The arrows for these features would point toward the upper-right quadrant where US companies cluster. To

confirm, check the numerical loadings for PC1 and PC2 and identify which features have the largest magnitude in the relevant direction.

Problem 6.4 (Medium). PC1 with roughly equal positive loadings on every stock represents the **market factor**—the common movement of all stocks. When markets rise, PC1 is positive; when they fall, PC1 is negative. Verification: compute the correlation between PC1 scores and a broad market index (e.g., S&P 500 daily return). Expect $r > 0.90$.

PC2 with positive loadings on growth stocks and negative loadings on value stocks represents a **growth-vs-value factor** (or a "style" factor). Verification: compute the correlation between PC2 scores and a known growth-minus-value factor (e.g., the Fama-French HML factor, sign-reversed). Expect $|r| > 0.60$.

Problem 6.5 (Hard). We want to show $\text{Cov}(x_j, z_k) = \sqrt{\lambda_k}\, v_{k,j}$.

Start from $z_k = v_k^\top x$. Since the data is centered:

$$\text{Cov}(x_j, z_k) = \text{Cov}(x_j, v_k^\top x) = \text{Cov}(x_j, \sum_{\ell=1}^{p} v_{k,\ell} x_\ell) = \sum_{\ell=1}^{p} v_{k,\ell}\, \text{Cov}(x_j, x_\ell)$$

by linearity of covariance. The term $\text{Cov}(x_j, x_\ell) = \Sigma_{j\ell}$, so:

$$\text{Cov}(x_j, z_k) = \sum_{\ell=1}^{p} v_{k,\ell}\, \Sigma_{j\ell} = [\Sigma v_k]_j$$

Since v_k is an eigenvector: $\Sigma v_k = \lambda_k v_k$. Therefore:

$$\text{Cov}(x_j, z_k) = [\lambda_k v_k]_j = \lambda_k v_{k,j}$$

Now the correlation between x_j and z_k is:

$$\text{Cor}(x_j, z_k) = \frac{\text{Cov}(x_j, z_k)}{\sigma_{x_j} \sigma_{z_k}} = \frac{\lambda_k v_{k,j}}{\sigma_{x_j} \sqrt{\lambda_k}} = \frac{\sqrt{\lambda_k}\, v_{k,j}}{\sigma_{x_j}}$$

The scaled loading is $\ell_{k,j} = \sqrt{\lambda_k}\, v_{k,j} = \text{Cov}(x_j, z_k)/\sqrt{\lambda_k} \times \sqrt{\lambda_k} = \text{Cov}(x_j, z_k)$. Wait—let me be precise. We showed $\text{Cov}(x_j, z_k) = \lambda_k v_{k,j}$, so $\sqrt{\lambda_k} v_{k,j} = \text{Cov}(x_j, z_k)/\sqrt{\lambda_k}$. But the claim is that the *scaled* loading $\ell_{k,j} = \sqrt{\lambda_k}\, v_{k,j}$ equals $\text{Cov}(x_j, z_k)/\sqrt{\lambda_k} \cdot \sqrt{\lambda_k}$... Actually, $\text{Cov}(x_j, z_k) = \lambda_k v_{k,j}$ directly shows that $\sqrt{\lambda_k}\, v_{k,j} = \text{Cov}(x_j, z_k)/\sqrt{\lambda_k}$. When features are standardized ($\sigma_{x_j} = 1$), the scaled loading equals the correlation: $\ell_{k,j} = \sqrt{\lambda_k}\, v_{k,j} = \text{Cor}(x_j, z_k)$. □

Section 7: ML Pipelines and Data Leakage

Problem 7.1 (Easy). The leakage is in line 2, where the mean and standard deviation are computed on the full dataset `X` (including future test data) via `X['age'].mean()` and `X['age'].std()`. After the split in line 3, the test set's statistics have already influenced the scaling.

Fix using a pipeline:

```
                    X_train, X_test = train_test_split(X)
pipe = Pipeline([('scaler', StandardScaler()), ('model', LogisticRegression())])
                    pipe.fit(X_train, y_train)
```

Problem 7.2 (Easy). `StandardScaler().fit_transform(X)` computes the mean μ and standard deviation σ from all rows of `X`, including the rows that will later become the test set. The scaled test data then uses μ and σ that were partly computed from itself. Specifically, the mean of the training-plus-test data is contaminated: the test set's values pull μ and σ toward

the test distribution. At training time, the model learns from features that already reflect the test set's distribution, inflating the validation score.

Problem 7.3 (Medium).

Pipeline: Impute → Scale → PCA → Classify

```
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression

pipe = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler',  StandardScaler()),
    ('pca',     PCA(n_components=5)),
    ('model',   LogisticRegression(max_iter=1000)),
])

pipe.fit(X_train, y_train)
y_pred = pipe.predict(X_test)
```

Problem 7.4 (Medium).

ColumnTransformer for Mixed Data

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

numeric_features     = ['age', 'income', 'balance']
categorical_features = ['state', 'job']

preprocess = ColumnTransformer([
    ('num', StandardScaler(),
     numeric_features),
    ('cat', OneHotEncoder(handle_unknown='ignore'),
     categorical_features),
])
```

The `handle_unknown='ignore'` parameter tells the encoder to produce a zero-vector for categories not seen during training, rather than raising an error. This is essential for production systems where new categories may appear.

Problem 7.5 (Hard). Nested cross-validation has two levels:

Outer loop (performance estimation): Split data into K_{outer} folds. Each fold's test set provides one unbiased performance estimate.

Inner loop (hyperparameter tuning): Within each outer fold's training set, run K_{inner}-fold CV (or grid/random search) to select the best hyperparameters. Then retrain on the full outer training set with those hyperparameters and evaluate on the outer test set.

sklearn implementation:

```
inner_cv = GridSearchCV(pipe, param_grid, cv=5)
outer_scores = cross_val_score(inner_cv, X, y, cv=5)
```

Why single-level CV is biased: If you use the same CV folds to both tune hyperparameters and report performance, the reported score reflects the best of many configurations—a form of selection bias. The model was chosen because it looked best on that particular split, so the score is optimistic. Nested CV separates the selection step (inner) from the evaluation step (outer), eliminating this bias.

Section 8: Cross-Validation, Grid Search, and Production

Problem 8.1 (Easy). Grid: $4 \times 5 = 20$ hyperparameter combinations. Each is evaluated with 5-fold CV, so each combination requires 5 model fits. Total fits: $20 \times 5 = 100$. At 3 seconds per fit: $100 \times 3 = 300$ seconds $= 5$ minutes.

Note: with `n_jobs=-1` (parallel), wall time depends on the number of CPU cores.

Problem 8.2 (Easy). `TimeSeriesSplit` uses expanding (or rolling) windows because time-series observations are ordered—each observation represents a specific point in time. Random K-fold shuffles observations, mixing past and future. A shuffled fold might train on data from 2023 and validate on data from 2021, meaning the model uses future information to predict the past. This produces an artificially inflated validation score that does not reflect real-world deployment, where you can only train on past data and predict forward.

Problem 8.3 (Medium).

RandomizedSearchCV with Log-Uniform C

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import loguniform

pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('model',  LogisticRegression(max_iter=1000)),
])

param_dist = {
    'model__C': loguniform(1e-4, 1e2),
}

search = RandomizedSearchCV(
    pipe, param_dist, n_iter=30, cv=5,
    scoring='accuracy', random_state=42, n_jobs=-1
)
search.fit(X_train, y_train)
print(f'Best C: {search.best_params_}')
print(f'Best score: {search.best_score_:.3f}')
```

Problem 8.4 (Medium). With 120 months, 6 folds, and a 2-month gap:

The data is divided into 7 blocks of approximately $120/7 \approx 17$ months each (6 splits create 7 blocks). For expanding-window TimeSeriesSplit:

Fold	Training months	Gap	Validation months
1	1–17 (17)	18–19 (2)	20–36 (17)
2	1–36 (36)	37–38 (2)	39–55 (17)
3	1–55 (55)	56–57 (2)	58–74 (17)
4	1–74 (74)	75–76 (2)	77–93 (17)
5	1–93 (93)	94–95 (2)	96–112 (17)
6	1–112 (112)	113–114 (2)	115–120 (6)

Training grows by one block each fold. The gap prevents overlap between rolling features.

Problem 8.5 (Hard).
(1) Computational complexity:

- *Grid search:* If each hyperparameter has m values and there are d hyperparameters, cost is $O(m^d)$—exponential in the number of hyperparameters.
- *Random search:* Cost is $O(B)$ where B is the iteration budget, independent of d.
- *Bayesian optimization:* Each iteration costs $O(n^3)$ for Gaussian process fitting (where n is the number of completed trials), plus the model evaluation. Total cost is $O(B \cdot n^3)$, but B is typically small.

(2) Coverage:

- *Grid search:* Covers a fixed lattice. If some dimensions are irrelevant, many grid points are wasted—all points in the irrelevant dimensions are explored exhaustively.
- *Random search:* Covers each dimension independently. With $B = 60$ random samples, each dimension sees 60 distinct values, far more than a typical grid. Bergstra and Bengio (2012) showed this outperforms grid search when hyperparameter importance is unequal.
- *Bayesian optimization:* Focuses samples on promising regions, achieving better coverage of the high-performance region at the cost of less exploration elsewhere.

(3) When to use each:

- *Grid search:* Small, well-understood parameter spaces with 2–3 parameters and few values each. Exhaustive and reproducible.
- *Random search:* Default choice for most tuning tasks. Best when some parameters matter much more than others (which is usually the case).
- *Bayesian optimization:* When each model evaluation is very expensive (hours per run, as in deep learning). The overhead of the surrogate model is justified by saving evaluation budget.

Reference: Bergstra, J. and Bengio, Y. (2012). "Random Search for Hyper-Parameter Optimization." *Journal of Machine Learning Research*, 13, 281–305.