Machine Learning in Finance (RSM338)

Lecture 4: Clustering

Table of contents

1 Introduction . . . 1
2 What is Clustering? . . . 1
3 Measuring Similarity . . . 2
4 K-Means Clustering . . . 5
4.1 A Finance Example . . . 5
4.2 Choosing the Number of Clusters . . . 8
5 Hierarchical Clustering . . . 11
6 Summary . . . 14

1 Introduction

last lecture we introduced the machine learning framework: models, loss functions, and learning algorithms. We focused on supervised learning, where we have labeled data—a target variable y that we want to predict from features $\mathbf{x}$. But supervised learning requires labels, and labels aren't always available.

this lecture we turn to **unsupervised learning**, where we have only features—no target variable. The goal is to discover structure in the data itself. The most fundamental unsupervised technique is **clustering**: grouping similar objects together.

Clustering has many potential applications in finance. Portfolio managers might group stocks into "peer groups" for relative valuation. Risk managers could cluster countries by economic indicators to identify similar risk profiles. Marketing teams might segment customers by behavior to target products. Quantitative researchers sometimes try to identify market "regimes"—periods with distinct statistical properties—and adapt their strategies accordingly.

The chapter covers five topics. First, we formalize what clustering means and why it's useful. Second, we develop the notion of distance—how we measure whether two objects are "similar." Third, we study the K-Means algorithm, the workhorse of clustering. Fourth, we address the critical question of how many clusters to use. Finally, we introduce hierarchical clustering, an alternative approach that builds a tree of increasingly coarse groupings.

2 What is Clustering?

In supervised learning, each observation comes with a label—a continuous outcome for regression (stock returns, house prices) or a category for classification (default/no default, spam/not spam). The algorithm learns to predict the label from the features. In unsupervised learning, there are no labels—just features. The algorithm must discover structure without being told what to look for. This is fundamentally different: we're not predicting anything specific, we're exploring. The main unsupervised techniques are clustering

 © Professor Kevin Mott

(grouping similar observations), dimensionality reduction (finding lower-dimensional representations), and anomaly detection (identifying observations that don't fit the pattern). This chapter focuses on clustering.

Formally, clustering aims to partition n objects into groups (clusters) such that objects *within* a cluster are similar to each other, while objects in *different* clusters are dissimilar. We don't know in advance how many clusters there should be, or what they represent—the algorithm discovers this from the data. This sounds vague, and it is. Unlike supervised learning, where we have a clear objective (minimize prediction error), clustering requires us to define what "similar" means and decide how many groups we want. These choices matter enormously.

Potential finance applications include customer segmentation (grouping clients by trading behavior and risk tolerance to tailor products), stock classification (discovering "peer groups" for relative valuation based on return patterns and fundamentals), country risk assessment (clustering countries by economic indicators to identify similar risk profiles), and regime detection (identifying market "states" like bull markets or high-volatility periods from historical data).

To cluster objects, we need to describe them numerically. Each object is represented by a vector of **features** (also called attributes or variables). Object i has p features:

$$\mathbf{x}_i = (x_{i1}, x_{i2}, \ldots, x_{ip})$$

where x_{ij} is the value of feature j for object i. The subscript i indexes objects (from 1 to n), and j indexes features (from 1 to p)—the same notation we used for data matrices in Lecture 1. For example, a stock might be described by $\mathbf{x}_i$ = (average return, volatility, beta, market cap), or a country by $\mathbf{x}_i$ = (GDP growth, corruption index, peace index, legal risk). Geometrically, each object is a point in p-dimensional space, and clustering means finding groups of points that are "close" to each other.

3 Measuring Similarity

To cluster objects, we need to quantify how similar—or dissimilar—two objects are. The standard approach is to measure **distance**: smaller distance means more similar. Given two objects with feature vectors $\mathbf{x}_i$ and $\mathbf{x}_j$, we need a function $d(\mathbf{x}_i, \mathbf{x}_j)$ that returns the distance between them. This function should satisfy intuitive properties: $d(\mathbf{x}_i, \mathbf{x}_i) = 0$ (an object is zero distance from itself), $d(\mathbf{x}_i, \mathbf{x}_j) = d(\mathbf{x}_j, \mathbf{x}_i)$ (distance is symmetric), and $d(\mathbf{x}_i, \mathbf{x}_j) \geq 0$ (distance is non-negative).

The most common choice is **Euclidean distance**—the "straight line" distance you learned in geometry. In two dimensions:

$$d(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{(x_{i1} - x_{j1})^2 + (x_{i2} - x_{j2})^2}$$

This is the Pythagorean theorem: the distance is the hypotenuse of a right triangle with legs equal to the differences in each coordinate.

Python

```
import numpy as np
import matplotlib.pyplot as plt

# Two points forming a 3-4-5 right triangle
p1 = np.array([1, 2])
p2 = np.array([4, 6])

fig, ax = plt.subplots()

# Plot the two points
ax.scatter([p1[0], p2[0]], [p1[1], p2[1]], s=100)
ax.annotate('$\\mathbf{x}_i = (1, 2)$', p1, xytext=(p1[0]-0.8, p1[1]-0.5))
ax.annotate('$\\mathbf{x}_j = (4, 6)$', p2, xytext=(p2[0]+0.1, p2[1]+0.3))
```

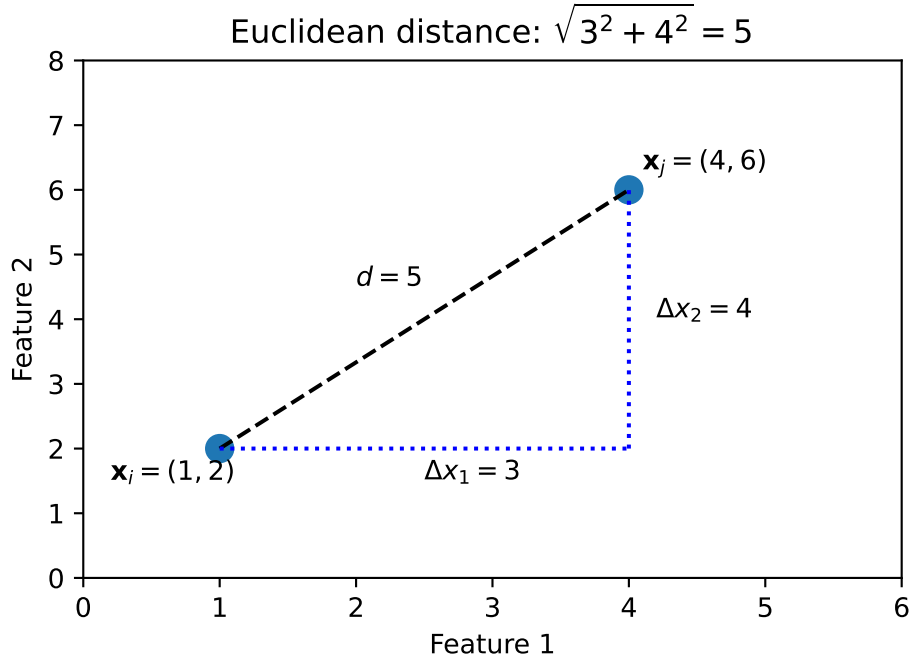
 © Professor Kevin Mott

```
# Draw the distance line (hypotenuse)
ax.plot([p1[0], p2[0]], [p1[1], p2[1]], 'k--')

# Draw the right triangle legs
ax.plot([p1[0], p2[0]], [p1[1], p1[1]], 'b:')  # horizontal leg
ax.plot([p2[0], p2[0]], [p1[1], p2[1]], 'b:')  # vertical leg

# Label the sides
ax.text(2.5, 1.5, '$\\Delta x_1 = 3$')
ax.text(4.2, 4, '$\\Delta x_2 = 4$')
ax.text(2, 4.5, '$d = 5$')

ax.set_xlabel('Feature 1')
ax.set_ylabel('Feature 2')
ax.set_title('Euclidean distance: $\\sqrt{3^2 + 4^2} = 5$')
ax.set_xlim(0, 6)
ax.set_ylim(0, 8)
plt.show()
```



In p dimensions, the formula generalizes naturally:

$$d(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{\sum_{k=1}^{p} (x_{ik} - x_{jk})^2}$$

The subscript k indexes the features from 1 to p. We take the difference in each feature, square it, sum all the squared differences, and take the square root. Using vector notation (from Lecture 1), this is the norm of the difference vector: $d(\mathbf{x}_i, \mathbf{x}_j) = \|\mathbf{x}_i - \mathbf{x}_j\|$. In Python, we can compute this with `np.linalg.norm`:

Python

```
# Two countries with 4 risk measures each
country_A = np.array([2.1, 45, 1.8, 60])  # (GDP growth, corruption, peace, legal)
```


© Professor Kevin Mott

```python
country_B = np.array([3.5, 72, 2.1, 45])

# Euclidean distance
distance = np.linalg.norm(country_A - country_B)
print(f"Distance between countries: {distance:.2f}")
```

Output

```
Distance between countries: 30.92
```

Euclidean distance has a critical flaw when features are on different scales. Consider clustering countries by GDP growth (ranging from -5% to +10%) and GDP level (ranging from $1 billion to $20,000 billion). If we compute distance on raw values, GDP level will completely dominate—a difference of $100 billion in GDP will swamp a difference of 5 percentage points in growth rate, even though both might be equally important for our analysis.

Python

```python
# Raw features (growth %, GDP in billions)
country_A = np.array([2.0, 500])    # 2% growth, $500B GDP
country_B = np.array([7.0, 510])    # 7% growth, $510B GDP
country_C = np.array([2.5, 5000])   # 2.5% growth, $5000B GDP

# Distances from A
d_AB = np.linalg.norm(country_A - country_B)
d_AC = np.linalg.norm(country_A - country_C)

print(f"Distance A to B: {d_AB:.1f}")
print(f"Distance A to C: {d_AC:.1f}")
```

Output

```
Distance A to B: 11.2
Distance A to C: 4500.0
```

Country B has a very different growth rate from A (7% vs 2%), while C has a similar growth rate (2.5% vs 2%). But the raw distance says B is "closer" to A than C is—because GDP level dominates the calculation. This is clearly wrong if growth rate matters for our analysis.

The solution is **standardization**: before clustering, we transform each feature to have mean zero and standard deviation one. For each feature j: $z_{ij} = (x_{ij} - \bar{x}_j)/s_j$, where $\bar{x}_j$ is the mean and s_j is the standard deviation. After standardization, a one-unit difference in any feature represents "one standard deviation," putting all features on comparable scales.

Python

```python
from sklearn.preprocessing import StandardScaler

# Stack countries into a matrix
X = np.array([[2.0, 500], [7.0, 510], [2.5, 5000]])

# Standardize
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Distances on standardized features
d_AB_scaled = np.linalg.norm(X_scaled[0] - X_scaled[1])
d_AC_scaled = np.linalg.norm(X_scaled[0] - X_scaled[2])

print(f"Distance A to B (standardized): {d_AB_scaled:.2f}")
```


© Professor Kevin Mott

```
print(f"Distance A to C (standardized): {d_AC_scaled:.2f}")
```

Output

```
Distance A to B (standardized): 2.22
Distance A to C (standardized): 2.14
```

After standardization, B is farther from A than C is—because A and B have very different growth rates. The standardization lets both features contribute meaningfully to the distance.

> ⚠ Warning
>
> **Always standardize before clustering** unless you have a specific reason not to. Most clustering algorithms implicitly assume features are on comparable scales. Failing to standardize can produce misleading results.

4 K-Means Clustering

K-Means is the most widely used clustering algorithm. The name reflects its two key elements: K is the number of clusters (which you specify), and the algorithm uses the mean of each cluster as its representative point. You provide the data (n objects, each with p features) and choose K. The algorithm returns cluster assignments (which cluster each object belongs to) and cluster centers, called **centroids** (the "average" location of each cluster).

K-Means minimizes the **within-cluster sum of squares (WCSS)**:

$$W_K = \sum_{k=1}^{K} \sum_{i:C(i)=k} \|\mathbf{x}_i - \mu_k\|^2$$

Let's unpack this notation. $C(i) \in \{1, 2, \ldots, K\}$ is the cluster assignment for object i, and μ_k is the centroid of cluster k. The outer sum loops over clusters; the inner sum loops over objects assigned to that cluster. The term $\|\mathbf{x}_i - \mu_k\|^2$ is the squared distance from object i to its centroid. In words: we want each object to be close to its assigned centroid, and the objective measures total squared distance. The squared distance can be expanded as $\|\mathbf{x}_i - \mu_k\|^2 = \sum_{j=1}^{p}(x_{ij} - \mu_{kj})^2$—the sum of squared differences across all features, which is just Euclidean distance squared.

Minimizing the WCSS directly is computationally difficult (it's NP-hard). K-Means uses an iterative procedure that converges to a local minimum. **Step 0 (Initialize):** Pick K objects randomly as initial centroids. **Step 1 (Assign):** For each object i, find the nearest centroid and assign i to that cluster: $C(i) = \arg\min_k \|\mathbf{x}_i - \mu_k\|^2$. **Step 2 (Update):** For each cluster k, update the centroid to be the mean of all objects in that cluster: $\mu_k = \frac{1}{n_k}\sum_{i:C(i)=k} \mathbf{x}_i$, where n_k is the number of objects in cluster k. **Repeat** Steps 1-2 until convergence (assignments stop changing).

Why does this work? The algorithm alternates between two sub-problems, each of which has an easy solution. Given centroids, the best assignment for each object is obviously the nearest centroid—any other assignment would increase the objective. Given assignments, the best centroid for cluster k is the mean of its members—this is a standard result: the mean minimizes the sum of squared distances. By alternating between these two optimizations, each step improves (or maintains) the objective. Since the objective is bounded below by zero, the algorithm must converge. However, it may converge to a local minimum rather than the global minimum, which is why we run it multiple times with different initializations and keep the best result.

4.1 A Finance Example

Let's apply K-Means to a finance problem: can we cluster stocks based on financial characteristics, and do the resulting clusters correspond to something meaningful (like sector)? We'll deliberately pick an extreme

example to illustrate the method clearly. We select 20 stocks from two very different sectors: utilities and technology. These sectors have almost opposite characteristics:

Sector	Stocks	Typical Characteristics
Utilities	NEE, DUK, SO, D, AEP, EXC, SRE, XEL, PEG, WEC	Low beta, high dividend yield (defensive, income-focused)
Technology	AAPL, MSFT, NVDA, GOOGL, META, AVGO, AMD, CRM, ADBE, NOW	High beta, low dividend yield (growth-focused, volatile)

We use two features: **beta** (sensitivity to market movements—utilities ~0.5, tech ~1.2+) and **dividend yield** (utilities ~3-4%, tech ~0-1%). Beta and dividend yield are pulled from Yahoo Finance via `yfinance` (data cached January 6, 2025).

Python

```
import os
import pandas as pd
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

# Stock data (pre-cached to avoid API calls)
# In practice, you would pull this from yfinance
stocks_df = pd.DataFrame({
    'ticker': ['NEE', 'DUK', 'SO', 'D', 'AEP', 'EXC', 'SRE', 'XEL', 'PEG', 'WEC',
               'AAPL', 'MSFT', 'NVDA', 'GOOGL', 'META', 'AVGO', 'AMD', 'CRM', 'ADBE', 'NOW'],
    'beta': [0.49, 0.44, 0.45, 0.54, 0.49, 0.64, 0.69, 0.39, 0.58, 0.44,
             1.24, 0.90, 1.67, 1.05, 1.23, 1.13, 1.56, 0.98, 1.16, 1.03],
    'dividendYield': [0.027, 0.037, 0.033, 0.048, 0.035, 0.032, 0.028, 0.027, 0.030, 0.032,
                      0.004, 0.007, 0.0003, 0.005, 0.003, 0.012, 0.0, 0.0, 0.0, 0.0],
    'sector': ['Utilities']*10 + ['Technology']*10
})

print(f"Sample size: {len(stocks_df)} stocks")
print(stocks_df)
```

Output

```
Sample size: 20 stocks
   ticker  beta  dividendYield      sector
0     NEE  0.49         0.0270   Utilities
1     DUK  0.44         0.0370   Utilities
2      SO  0.45         0.0330   Utilities
3       D  0.54         0.0480   Utilities
4     AEP  0.49         0.0350   Utilities
5     EXC  0.64         0.0320   Utilities
6     SRE  0.69         0.0280   Utilities
7     XEL  0.39         0.0270   Utilities
8     PEG  0.58         0.0300   Utilities
9     WEC  0.44         0.0320   Utilities
10   AAPL  1.24         0.0040  Technology
11   MSFT  0.90         0.0070  Technology
```

```
12   NVDA  1.67         0.0003  Technology
13  GOOGL  1.05         0.0050  Technology
14   META  1.23         0.0030  Technology
15   AVGO  1.13         0.0120  Technology
16    AMD  1.56         0.0000  Technology
17    CRM  0.98         0.0000  Technology
18   ADBE  1.16         0.0000  Technology
19    NOW  1.03         0.0000  Technology
```

Warning

This example is intentionally clean. We picked two sectors that are almost opposites in terms of risk and income characteristics. Real-world clustering problems are messier—clusters may not align neatly with any known category, and interpreting what the clusters "mean" requires judgment.

Now we prepare the features, standardize them, and run K-Means with $K = 2$ (since we expect two groups). The `n_init=10` parameter tells sklearn to run the algorithm 10 times with different initializations and keep the best result.

Python

```python
# Prepare features
X = stocks_df[["beta", "dividendYield"]].values
tickers = stocks_df["ticker"].values
true_sectors = stocks_df["sector"].values

# Standardize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Fit K-Means with K=2
kmeans = KMeans(n_clusters=2, random_state=42, n_init=10)
cluster_labels = kmeans.fit_predict(X_scaled)

print(f"Stocks in cluster 0: {list(tickers[cluster_labels == 0])}")
print(f"Stocks in cluster 1: {list(tickers[cluster_labels == 1])}")
```

Output

```
Stocks in cluster 0: ['NEE', 'DUK', 'SO', 'D', 'AEP', 'EXC', 'SRE', 'XEL', 'PEG', 'WEC']
Stocks in cluster 1: ['AAPL', 'MSFT', 'NVDA', 'GOOGL', 'META', 'AVGO', 'AMD', 'CRM', 'ADBE', 'NOW']
```

Did K-Means discover the sector groupings on its own? Let's compare:

Python

```python
fig, axes = plt.subplots(1, 2)

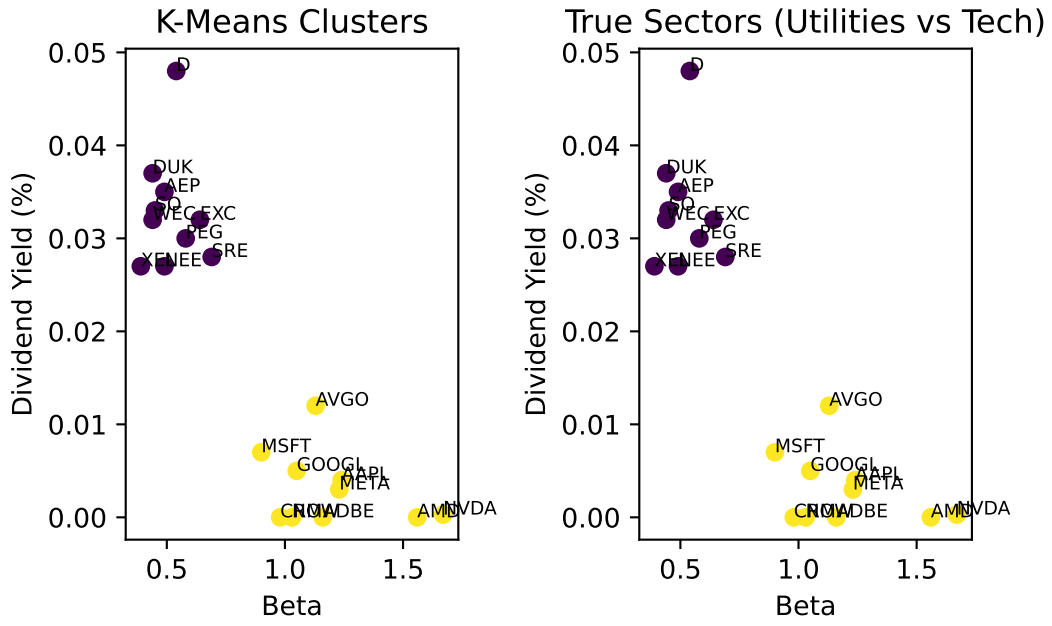
# K-Means clusters
axes[0].scatter(stocks_df["beta"], stocks_df["dividendYield"], c=cluster_labels)
for i, ticker in enumerate(tickers):
    axes[0].annotate(ticker, (stocks_df["beta"].iloc[i], stocks_df["dividendYield"].iloc[i]), fontsiz
axes[0].set_xlabel("Beta")
axes[0].set_ylabel("Dividend Yield (%)")
axes[0].set_title("K-Means Clusters")

# True sectors
```

 © Professor Kevin Mott

```
sector_colors = [0 if s == "Utilities" else 1 for s in true_sectors]
axes[1].scatter(stocks_df["beta"], stocks_df["dividendYield"], c=sector_colors)
for i, ticker in enumerate(tickers):
    axes[1].annotate(ticker, (stocks_df["beta"].iloc[i], stocks_df["dividendYield"].iloc[i]), fontsiz
axes[1].set_xlabel("Beta")
axes[1].set_ylabel("Dividend Yield (%)")
axes[1].set_title("True Sectors (Utilities vs Tech)")

plt.tight_layout()
plt.show()
```



K-Means recovered the sector groupings almost perfectly—with no labels! Utilities cluster together (low beta, high dividend yield), and tech stocks cluster together (high beta, low dividend yield). This illustrates the power of unsupervised learning: discovering structure that reflects real categories, without ever being told what those categories are. Again, this example was chosen to be clean. In practice, you might cluster all S&P 500 stocks and find that the resulting groups don't map neatly onto sectors—which could itself be interesting (maybe some "tech" companies behave more like utilities?).

K-Means has several limitations to keep in mind. **Local minima:** The algorithm finds a local minimum, not necessarily the global minimum—different random initializations can give different results, so always run multiple times (sklearn's `n_init` parameter). **You must specify K:** The algorithm requires you to choose the number of clusters in advance, but in many applications we don't know how many clusters there "should" be. We'll address this in the next section. **Spherical clusters:** K-Means assumes clusters are roughly spherical (ball-shaped) and of similar size, and struggles with elongated or irregularly shaped clusters. **Outlier sensitivity:** A single extreme observation can pull a centroid away from where it "should" be, distorting the clustering.

4.2 Choosing the Number of Clusters

K-Means requires you to specify K upfront, but how do you choose? There's a fundamental tradeoff: too few clusters means groups are too broad and we miss important distinctions; too many clusters means we're fitting noise rather than structure. At the extremes, $K = 1$ puts everything in one cluster (no structure discovered), while $K = n$ makes each object its own cluster (no compression). We want to find a K that

© Professor Kevin Mott

captures meaningful structure without overfitting.

Recall the K-Means objective: $W_K = \sum_{k=1}^{K} \sum_{i:C(i)=k} \|\mathbf{x}_i - \mu_k\|^2$. As we increase K, W_K always decreases—more clusters means smaller distances to the nearest centroid, and at the extreme, $K = n$ gives $W_K = 0$ (each point is its own centroid). So we can't just minimize W_K; that would always choose the largest possible K. We need to balance fit against complexity.

The **elbow method** plots W_K against K and looks for an "elbow"—a point where the improvement suddenly slows down:

Python

```python
from sklearn.datasets import make_blobs

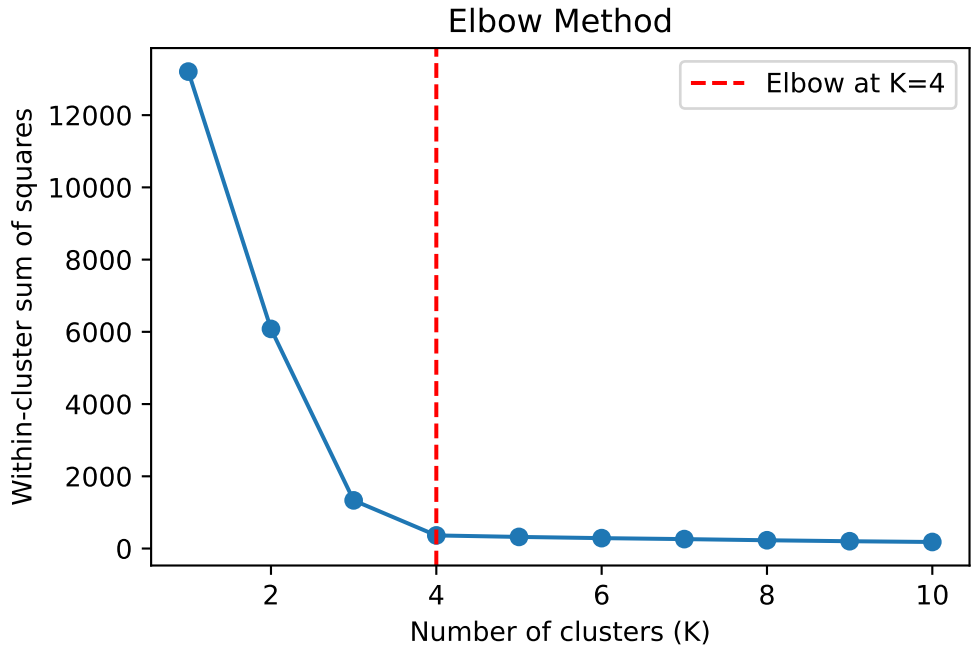
# Generate data with 4 natural clusters
np.random.seed(42)
X, _ = make_blobs(n_samples=200, centers=4, cluster_std=1.0)

# Compute WCSS for different values of K
wcss = []
K_range = range(1, 11)

for k in K_range:
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
    kmeans.fit(X)
    wcss.append(kmeans.inertia_)  # inertia_ is sklearn's name for WCSS

fig, ax = plt.subplots()
ax.plot(K_range, wcss, 'o-')
ax.set_xlabel('Number of clusters (K)')
ax.set_ylabel('Within-cluster sum of squares')
ax.set_title('Elbow Method')
ax.axvline(x=4, color='red', linestyle='--', label='Elbow at K=4')
ax.legend()
plt.show()
```


© Professor Kevin Mott



Before the elbow, adding clusters substantially reduces W_K. After the elbow, adding clusters provides diminishing returns. In this example, $K = 4$ is the elbow—which matches the true number of clusters in the synthetic data. The elbow is sometimes subtle or ambiguous; use it as a guide, not a rigid rule. Domain knowledge matters too: if business logic suggests there should be 5 customer segments, that's a good reason to use $K = 5$ even if the elbow is unclear.

The **silhouette score** provides another approach, measuring how well-separated the clusters are. For each object i, let $a(i)$ be the average distance to other objects in the same cluster, and $b(i)$ the average distance to objects in the nearest other cluster. The silhouette score is $s(i) = \frac{b(i)-a(i)}{\max(a(i),b(i))}$. Values close to 1 mean the object is well-matched to its cluster; close to 0 means it's on the boundary; negative values suggest it might be in the wrong cluster. The overall silhouette score averages across all objects—higher is better.

Python

```python
from sklearn.metrics import silhouette_score

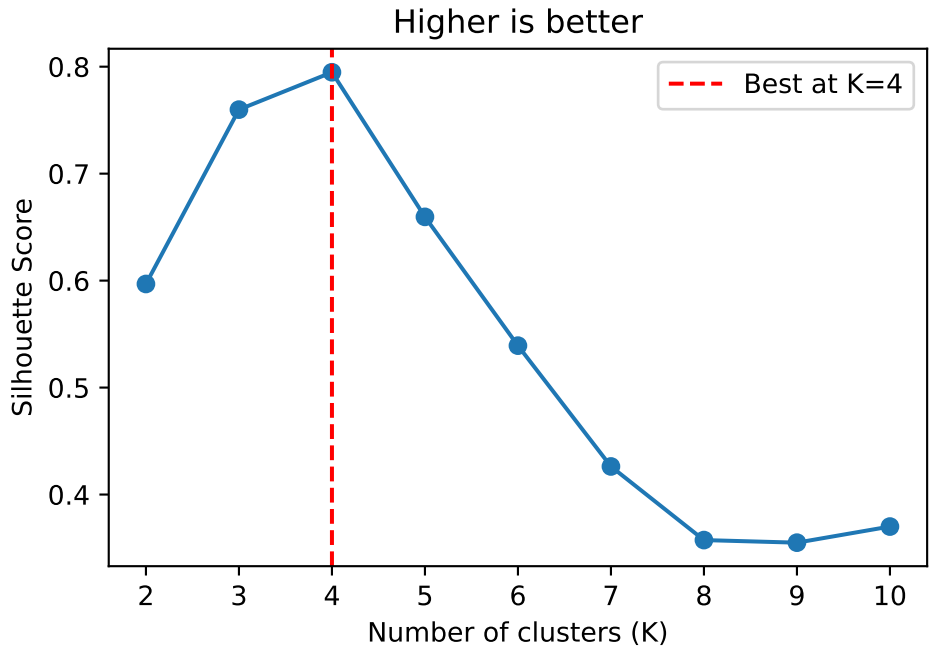
# Compute silhouette score for different K
silhouette_scores = []
K_range = range(2, 11)  # silhouette requires K >= 2

for k in K_range:
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
    labels = kmeans.fit_predict(X)
    score = silhouette_score(X, labels)
    silhouette_scores.append(score)

fig, ax = plt.subplots()
ax.plot(K_range, silhouette_scores, 'o-')
ax.set_xlabel('Number of clusters (K)')
ax.set_ylabel('Silhouette Score')
ax.set_title('Higher is better')
ax.axvline(x=4, color='red', linestyle='--', label='Best at K=4')
ax.legend()
```


© Professor Kevin Mott

```
plt.show()
```



The silhouette score peaks at $K = 4$, confirming what the elbow method suggested.

5 Hierarchical Clustering

K-Means requires you to specify the number of clusters K before you start. But what if you don't know how many clusters there should be? The elbow method and silhouette score can help, but they don't always give clear answers.

Hierarchical clustering takes a different approach: instead of committing to a specific K, it builds a complete hierarchy showing how objects group together at every level of similarity. Think of it like a family tree for your data—at the bottom, each object is its own "family" (most granular); as you move up, similar objects merge into larger families; at the top, everyone is in one big family (least granular). You can then "cut" this tree at any height to get as many or as few clusters as you want, *after* seeing the structure. This is the key advantage: you don't have to commit to K upfront.

The **agglomerative** (bottom-up) algorithm works as follows. Start with each of the n objects as its own cluster. Compute the distance between every pair of clusters, find the two that are closest, and merge them into one. Now you have $n - 1$ clusters. Repeat: find the closest pair, merge, and continue until everything is in one cluster. We record each merge as we go—after $n - 1$ merges, we have a complete record of how the clusters formed.

To make this concrete, suppose we have 5 objects (A, B, C, D, E) with pairwise distances given by:

	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

© Professor Kevin Mott

The smallest distance is $d(A, B) = 2$, so we merge A and B into cluster {A,B}. Next, the smallest remaining distance is $d(D, E) = 3$, so we merge D and E. Now we have {A,B}, C, and {D,E}. Continuing, suppose C is closest to {D,E}, so we merge them into {C,D,E}. Finally, we merge {A,B} and {C,D,E} into one cluster. At each step, we record which clusters merged and at what distance—this record is what the **dendrogram** visualizes.

But wait: once clusters contain multiple objects, how do we measure the distance between them? There are several options. **Single linkage** uses the minimum distance between any pair of objects (one from each cluster). **Complete linkage** uses the maximum. **Average linkage** uses the average of all pairwise distances. **Ward's method** chooses the merge that minimizes the increase in total within-cluster variance, similar in spirit to K-Means. Different linkage methods can give very different results: single linkage can create long, "chained" clusters; complete linkage creates compact, spherical clusters; Ward's method tends to create clusters similar to what K-Means would find. Ward's method is often a good default.

Let's build a dendrogram for our stock data:

Python

```python
from scipy.cluster.hierarchy import dendrogram, linkage, fcluster

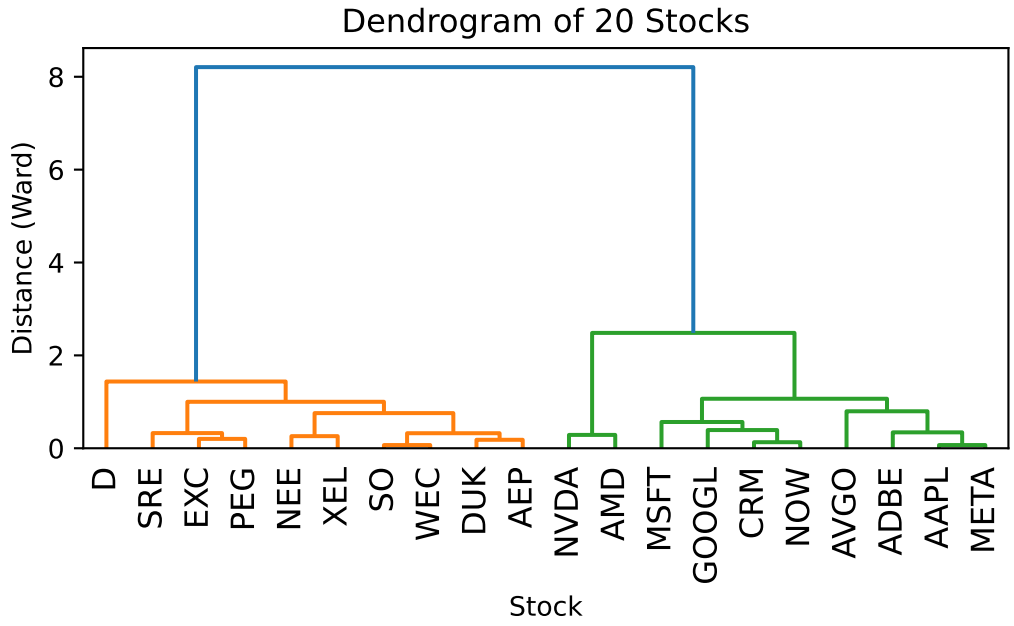
# Use our stock data
X_stocks = stocks_df[["beta", "dividendYield"]].values
stock_tickers = stocks_df["ticker"].values

# Standardize
scaler_hc = StandardScaler()
X_stocks_scaled = scaler_hc.fit_transform(X_stocks)

# Perform hierarchical clustering with Ward's method
Z = linkage(X_stocks_scaled, method='ward')

# Plot the dendrogram
fig, ax = plt.subplots()
dendrogram(Z, labels=stock_tickers, ax=ax, leaf_rotation=90)
ax.set_xlabel('Stock')
ax.set_ylabel('Distance (Ward)')
ax.set_title('Dendrogram of 20 Stocks')
plt.tight_layout()
plt.show()
```


© Professor Kevin Mott



Reading the dendrogram from bottom to top: each leaf at the bottom is one stock; the height where two branches merge shows how dissimilar those clusters were when merged; lower merges mean more similar objects, while the final merges at the top join quite different groups. In our stock dendrogram, the utilities cluster together on one side and tech stocks on the other, merging only at the very top—they're quite different from each other.

To get a specific number of clusters, draw a horizontal line across the dendrogram—the number of vertical lines it crosses equals the number of clusters. The key question is: where should you cut?

How to choose where to cut

Look for **large vertical gaps** in the dendrogram—places where the branches are tall before the next merge. A large gap means those clusters were quite dissimilar when they merged, suggesting they might be better left as separate groups.

In our stock dendrogram, there's a big gap between the utilities cluster and the tech cluster before they finally merge. That gap tells us "these two groups are very different"—a natural place to cut:

Python

```
fig, axes = plt.subplots(1, 2)

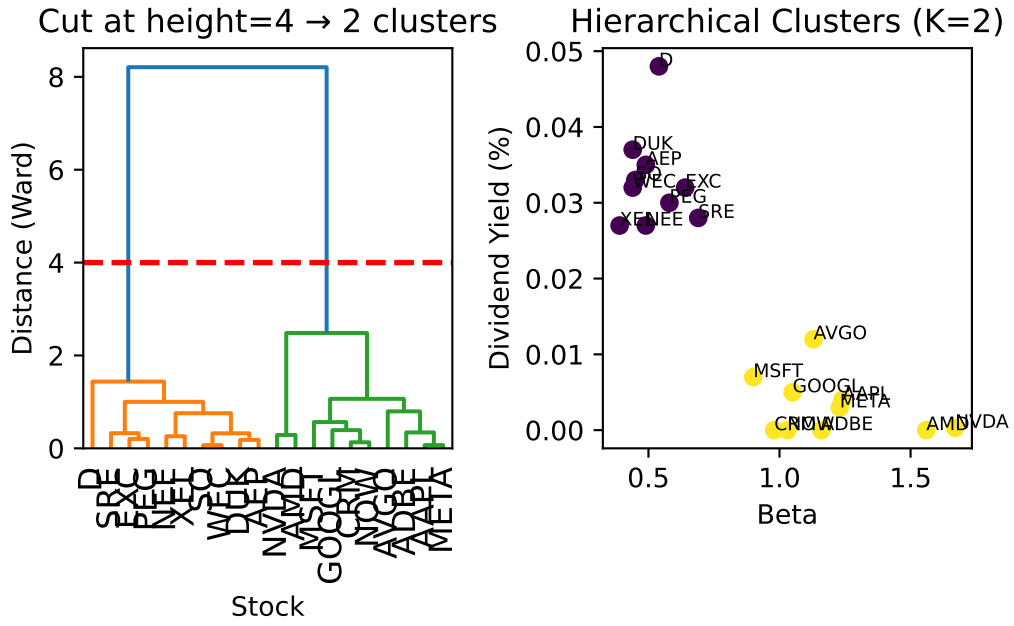
# Left: Dendrogram with cut line
dendrogram(Z, labels=stock_tickers, ax=axes[0], leaf_rotation=90)
axes[0].axhline(y=4, color='red', linestyle='--', linewidth=2)
axes[0].set_xlabel('Stock')
axes[0].set_ylabel('Distance (Ward)')
axes[0].set_title('Cut at height=4 → 2 clusters')

# Right: The resulting clusters
cluster_labels_hc = fcluster(Z, t=2, criterion='maxclust')
axes[1].scatter(stocks_df["beta"], stocks_df["dividendYield"], c=cluster_labels_hc)
for i, ticker in enumerate(stock_tickers):
    axes[1].annotate(ticker, (stocks_df["beta"].iloc[i], stocks_df["dividendYield"].iloc[i]), fontsiz
```


© Professor Kevin Mott

```
axes[1].set_xlabel("Beta")
axes[1].set_ylabel("Dividend Yield (%)")
axes[1].set_title("Hierarchical Clusters (K=2)")

plt.tight_layout()
plt.show()
```



The dendrogram lets you see the structure *before* committing to a number of clusters. You can experiment with different cuts to see what makes sense for your application.

How does hierarchical clustering compare to K-Means? The key differences are summarized in this table:

Aspect	K-Means	Hierarchical
Must specify K?	Yes, before running	No—choose after seeing the dendrogram
Output	Just cluster labels	Full tree structure showing relationships
Speed	Fast (scales to large n)	Slower (must compute all pairwise distances)
Deterministic?	No (random initialization)	Yes (same data → same tree)
Cluster shapes	Assumes spherical	More flexible (depends on linkage)

Use K-Means when you have a large dataset and a rough idea of K. Use hierarchical when you want to explore the cluster structure, or when n is small enough that speed isn't a concern (say, $n < 1000$). Using both as a sanity check is often wise—if they give very different answers, investigate why.

6 Summary

What is clustering?

Clustering is unsupervised learning: grouping objects by similarity without labels. Objects within a cluster should be similar; objects in different clusters should be dissimilar. Applications in finance include customer segmentation, stock classification, country risk assessment, and regime detection.

Measuring similarity

Distance quantifies how similar two objects are. Euclidean distance is most common:

$$d(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{\sum_{k=1}^{p} (x_{ik} - x_{jk})^2}$$

Always standardize features before clustering—otherwise, features on larger scales will dominate the distance calculation.

K-Means clustering

K-Means is the workhorse clustering algorithm. It minimizes within-cluster sum of squares by iterating between:

1. **Assign:** Each object goes to the nearest centroid
2. **Update:** Each centroid becomes the mean of its assigned objects

K-Means is fast and simple, but requires specifying K in advance and assumes spherical clusters.

Choosing K

The elbow method plots WCSS against K and looks for a bend. The silhouette score measures cluster separation (higher is better). Domain knowledge also matters—sometimes business logic dictates the number of clusters.

Hierarchical clustering

Hierarchical clustering builds a tree (dendrogram) by progressively merging clusters. Different linkage methods (single, complete, Ward) produce different structures. You can cut the tree at any level to get different numbers of clusters—look for large vertical gaps in the dendrogram to find natural cut points.

next lecture: We move to supervised learning—regression methods that predict continuous outcomes from features. The techniques we've developed for measuring distance and standardizing features will remain important.

© Professor Kevin Mott