# IMEN574/IDSC574 Programming for Data Science

## 11. **Clustering**

Spring 2026

Instructor: **Sangwoo Mo**

# Today's Agenda

## 01 What Is Clustering?

Unsupervised learning,
discovering hidden structure

## 02 K-Means Clustering

Core algorithm, K-Means++,
choosing K, limitations

## 03 From K-Means to GMM

Why K-Means is EM, extending to soft probabilistic assignments
GMM formulation, EM algorithm

## 04 Density-Based Clustering

DBSCAN, HDBSCAN,
non-convex shapes & varying density

## 05 Evaluation & Metrics

Silhouette, internal/external
metrics, common pitfalls

## 06 Full Example & Practice

Customer segmentation,
algorithm comparison

# What Is Clustering?

Given data X (no labels!), discover natural groups or structure in the data.

**Supervised (Classification)**

**Given:** X and y (labels)
**Goal:** predict y for new X
**Examples:** spam/ham, cat/dog

**Unsupervised (Clustering)**

**Given:** X only (no labels)
**Goal:** discover groups in X
**Examples:** customer segments, topics

Key challenge: no ground truth → how do we know if clusters are "good"? We need internal metrics and domain knowledge.

# Clustering Is Everywhere

### Customer Segmentation

Group customers by behavior for targeted marketing

### Anomaly Detection

Normal clusters vs. outliers in fraud/network security

### Document Clustering

Group similar articles, emails, or search results by topic

### Image Segmentation

Group pixels by color/texture for object detection

### Gene Expression

Group genes with similar expression patterns

### Recommendation

Group similar users/items for collaborative filtering

# How Do We Cluster?

There are many different philosophies for grouping data. We'll start with the simplest and build up.

① **Partitional**

Divide data into K groups
by minimizing distance to centers. Hard assignments only
— what if we want probabilities?

`K-Means, MiniBatch`

② **Model-Based**

Extends K-Means with soft (probabilistic)
assignments. Each cluster = one Gaussian distribution

`GMM`

③ **Density-Based**

Density-based: clusters are dense regions
separated by sparse areas

`DBSCAN, HDBSCAN`

④ **Hierarchical**

Build a tree of clusters
by merging or splitting

`Agglomerative`

*We'll start with K-Means, extend it to GMM, then tackle density-based and hierarchical approaches.*

# K-Means Clustering

The most fundamental clustering algorithm

# K-Means — The Algorithm

Partition n data points into K clusters by minimizing within-cluster variance.

1. Choose K (number of clusters)
2. Initialize K centroids randomly
3. Assign each point to nearest centroid
4. Recompute centroids as cluster means
5. Repeat 3–4 until convergence

## The EM Connection

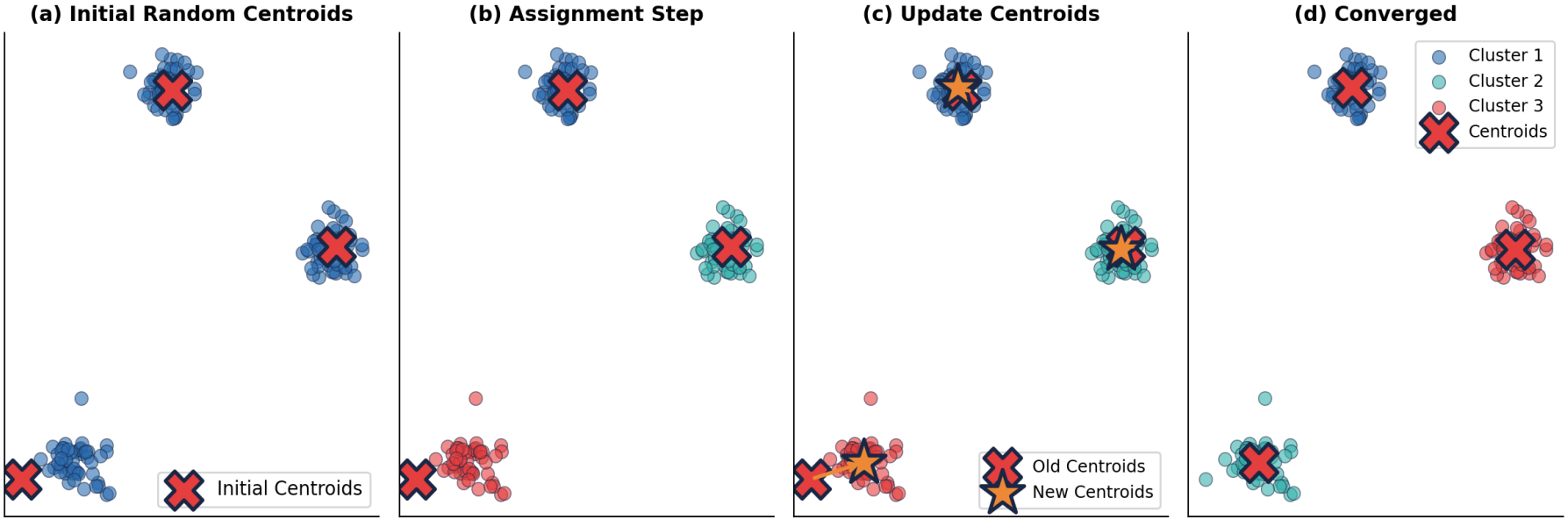
**E-step (Expectation):**
Assign each point to the nearest centroid

**M-step (Maximization):**
Recompute each centroid as the mean of its assigned points

Repeat until centroid positions stabilize.

**Tip:** K-Means is an EM-style algorithm: iterate between assigning points (E) and updating centers (M).

# K-Means — Step by Step




*Random init → Assign to nearest centroid → Update centroids → Converged*

# K-Means — Objective & Convergence

## Objective: Minimize Inertia

$$J = \sum_k \sum ||x_i - \mu_k||^2$$

**μk** = centroid (mean) of cluster k

Sum of squared distances from each point to its assigned centroid.

**Lower inertia = tighter, more compact clusters**

## Convergence & Complexity

**Guaranteed to converge:**
Inertia J decreases monotonically

**NP-hard problem:**
Lloyd's algorithm finds a local minimum

**Time complexity:**
O(nKi) where n=points, K=clusters, i=iterations

# K-Means — Code

```python
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
import numpy as np

# Always scale before clustering!
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Fit K-Means
kmeans = KMeans(n_clusters=3, init='k-means++',
                n_init=10, random_state=42)
kmeans.fit(X_scaled)

labels = kmeans.labels_           # cluster assignments
centers = kmeans.cluster_centers_# centroids
inertia = kmeans.inertia_         # sum of squared dist
```

## Key Parameters

**`n_clusters=3`**
Number of clusters K

**`init='k-means++'`**
Smart initialization (default)

**`n_init=10`**
Run 10x, keep best result

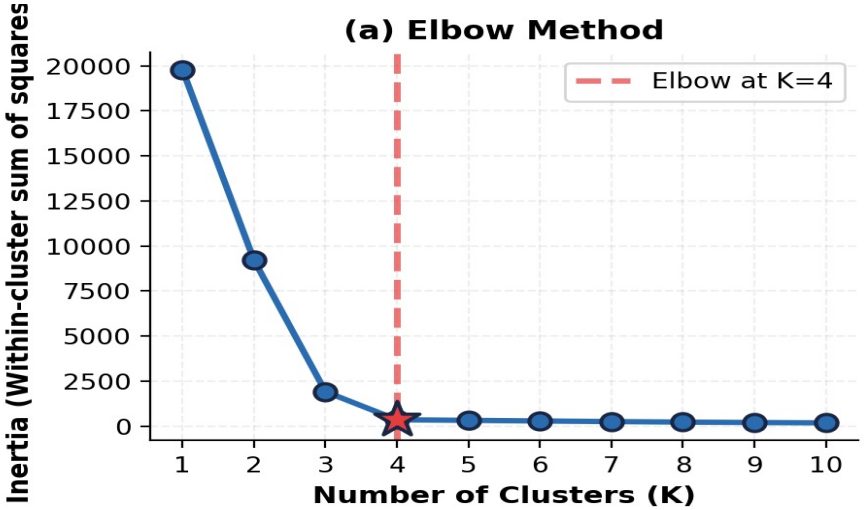
**`random_state=42`**
Reproducibility

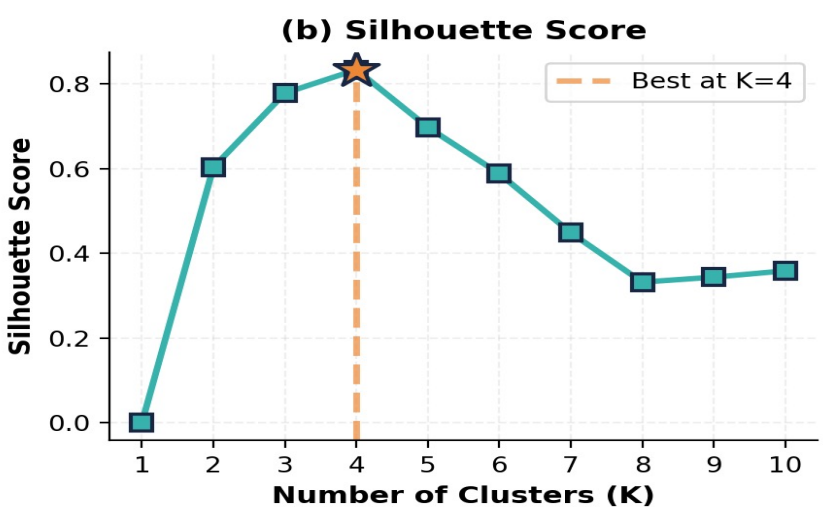
# K-Means++ & Choosing K

### K-Means++ Initialization

1. First centroid = random point
2. Each next centroid chosen with probability $\propto D(x)^2$
3. Spreads centroids across data, avoids poor local minima

*O(log K)-competitive with the optimal solution.*



# Silhouette Score — Measuring Cluster Quality

$$s(i) = ( b(i) - a(i) ) / max( a(i), b(i) )$$

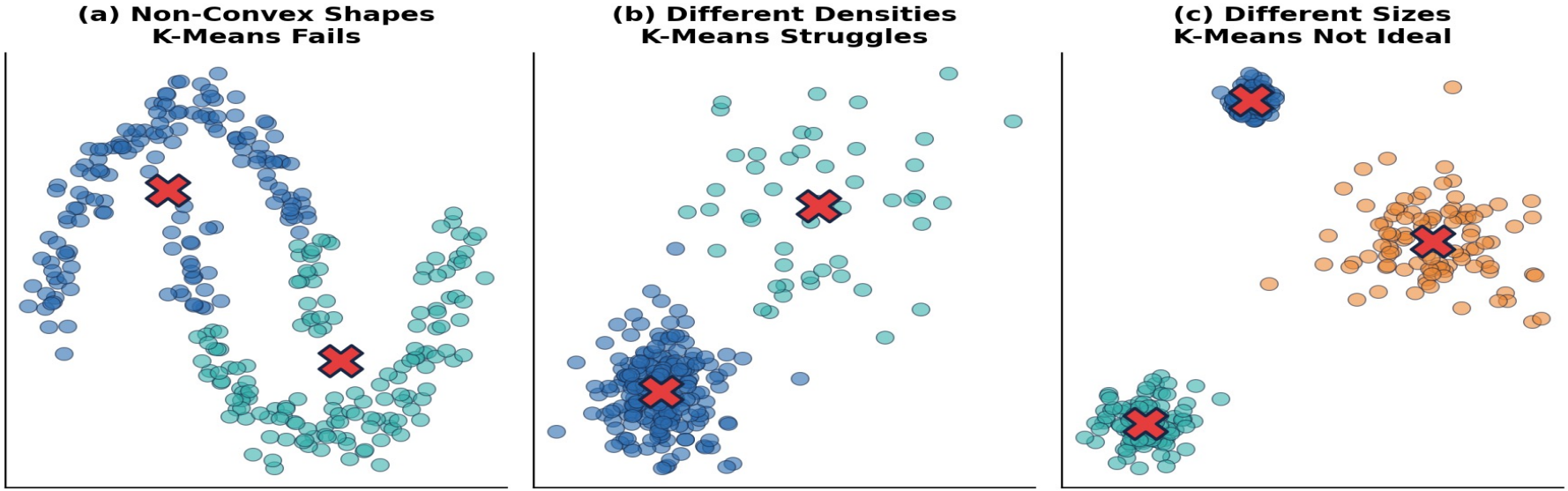
**a(i)** = mean distance to points in **same cluster** (cohesion)
**b(i)** = mean distance to **nearest other cluster** (separation)

**Range: −1 to +1.** Higher = better. s > 0.5 is generally good.
**+1:** well-clustered | **0:** on boundary | **−1:** misplaced

```
from sklearn.metrics import silhouette_score

scores = []
for k in range(2, 11):
    km = KMeans(n_clusters=k, n_init=10,
                random_state=42).fit(X_scaled)
    sc = silhouette_score(X_scaled, km.labels_)
    scores.append(sc)
    print(f"K={k}: silhouette={sc: 3f}")
```

# K-Means — Limitations




**Spherical Only**

Hard assignments, convex shapes only → GMM fixes this

**Must Specify K**

Need to guess K in advance

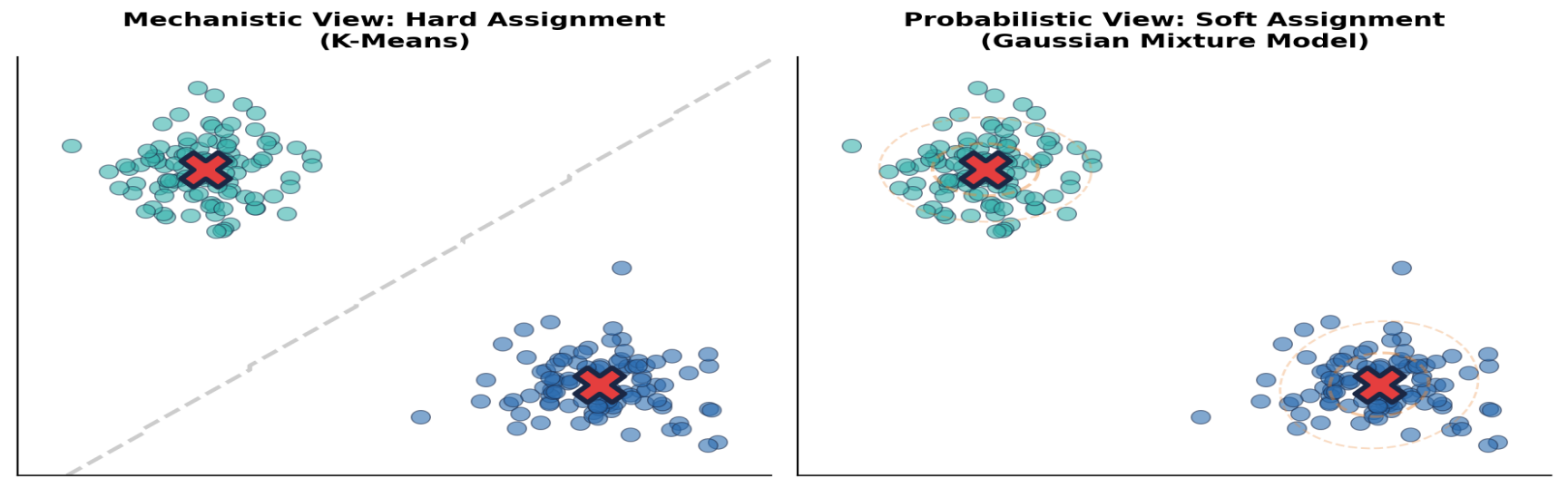
**Sensitive to Outliers**

Outliers pull centroids away from centers

**Needs Scaling**

Larger-scale features dominate distance

# K-Means: Mechanistic → Probabilistic




**Mechanistic View**

Hard assignment, Voronoi boundaries
Each point → exactly 1 cluster

**Probabilistic View**

Soft assignment, Gaussian contours
Each point → probability distribution

This probabilistic view leads us to EM and GMM →

# K-Means as an EM Algorithm

## K-Means Steps

1) Initialize centroids

**2) E-step:** Assign each point to nearest centroid (**hard** assignment)

**3) M-step:** Recompute centroid as mean

## General EM Framework

1) Initialize parameters θ

**2) E-step:** Compute posterior probabilities (**soft** assignment)

**3) M-step:** Update θ to maximize E[log-likelihood]

**K-Means is a special case of EM** where assignments are **hard (0 or 1)** instead of **soft (probabilities)**.
What if we allow soft assignments with Gaussian probabilities? →
**Gaussian Mixture Model (GMM)**

# Gaussian Mixture Models

Soft clustering with probabilistic assignments

# GMM — Soft Assignment

Each cluster = one Gaussian distribution. Each point has a probability of belonging to each cluster.

$$p(x) = \sum_k \pi_k \, N(x \mid \mu_k, \Sigma_k)$$

where $N(x \mid \mu, \Sigma) = (2\pi)^{-d/2} \, |\Sigma|^{-1/2} \, \exp(-\tfrac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu))$

### K-Means (Hard Assignment)

Each point belongs to exactly one cluster. Binary: 0 or 1.

### GMM (Soft Assignment)

Each point has a probability distribution over all clusters.

## Each Gaussian Component Has Three Parameters:

**μk**

Mean (center)

**Σk**

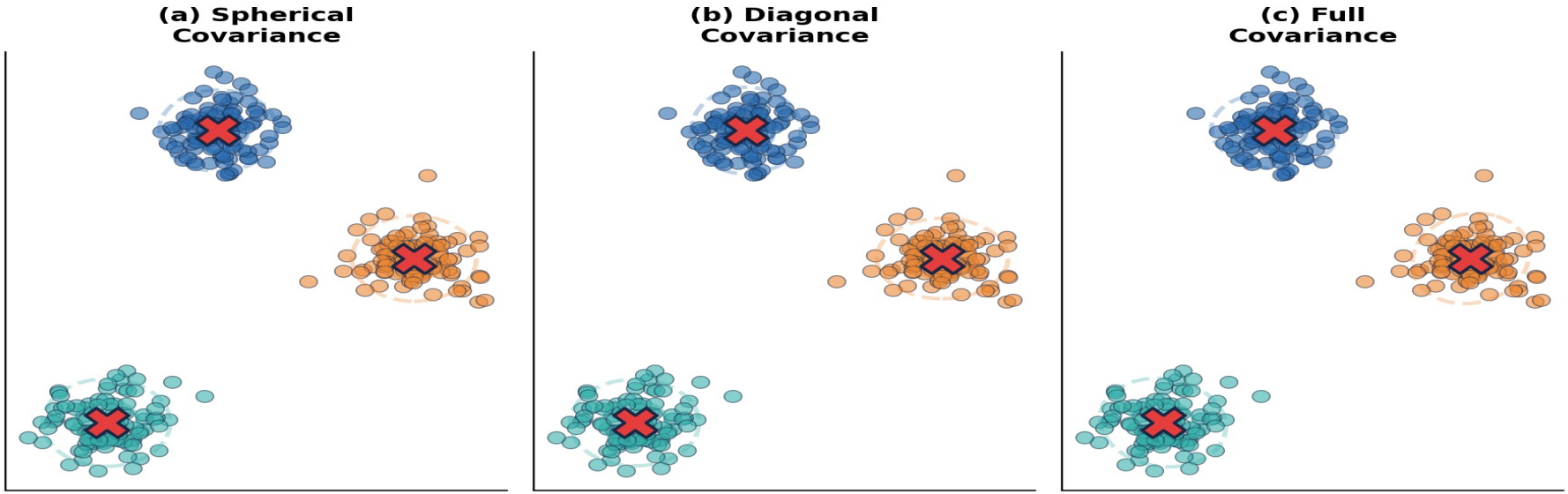
Covariance (shape/spread)

**πk**

Weight (mixing coefficient)

**Tip:** GMM gives uncertainty: "80% cluster 2, 15% cluster 1, 5% cluster 3" (probabilities sum to 1)

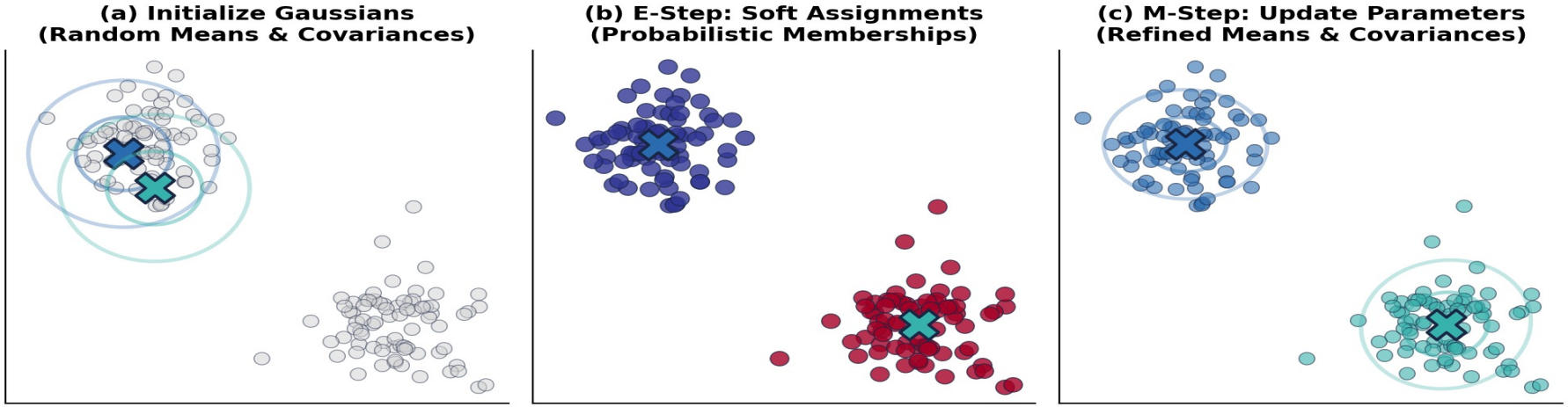
# GMM — Covariance & Cluster Shapes




| | |
|---|---|
| `spherical` | All clusters are circular (like K-Means) |
| `diag` | Axis-aligned ellipses |
| `full` | Arbitrary ellipses (most flexible) |

*Use covariance_type='full' for maximum flexibility, 'spherical' if you want K-Means-like behavior.*

# EM Algorithm for GMM




## E-step (Expectation)

Compute soft assignments:
responsibility of each Gaussian
for each data point

## M-step (Maximization)

Update μ, Σ, π to maximize
likelihood given soft assignments.
Guaranteed to increase monotonically.

# GMM — Code & Choosing K

```python
from sklearn.mixture import GaussianMixture

gmm = GaussianMixture(n_components=3,
    covariance_type='full', random_state=42)
gmm.fit(X_scaled)

labels = gmm.predict(X_scaled)        # hard labels
probs = gmm.predict_proba(X_scaled)   # soft probs

print(f"BIC: {gmm.bic(X_scaled):.1f}")
print(f"AIC: {gmm.aic(X_scaled):.1f}")
```

**Model Selection: BIC / AIC**

BIC = −2 log L + p log(n)
AIC = −2 log L + 2p

Lower = better model.
BIC penalizes complexity more.

```python
# Choosing K with BIC
bics = []
K_range = range(2, 11)
for k in K_range:
    gm = GaussianMixture(n_components=k,
        covariance_type='full', random_state=42)
    gm.fit(X_scaled)
    bics.append(gm.bic(X_scaled))

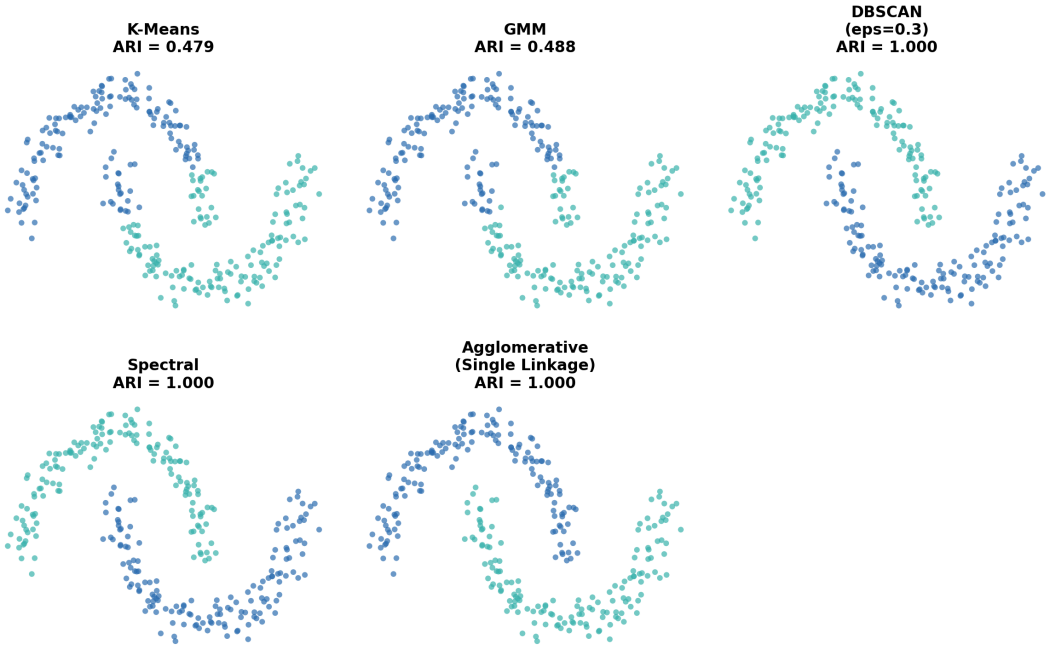
best_k = K_range[np.argmin(bics)]
print(f"Best K by BIC: {best_k}")
```

# Density-Based Clustering DBSCAN & HDBSCAN

When convex clusters aren't enough

# Why K-Means Fails — The Two Moons Problem

Some datasets have non-convex structure that K-Means and GMM cannot capture.




**DBSCAN, Spectral, and Agglomerative (single linkage) all achieve ARI = 1.0 — K-Means and GMM cannot.**

# DBSCAN — Density-Based Clustering

Groups together points that are closely packed. Points in low-density regions become outliers.

## Core Concept: What Is Density?

Density = how many points exist within a certain distance (eps). Dense regions have many neighbors; sparse regions have few.

### `eps (ε)`

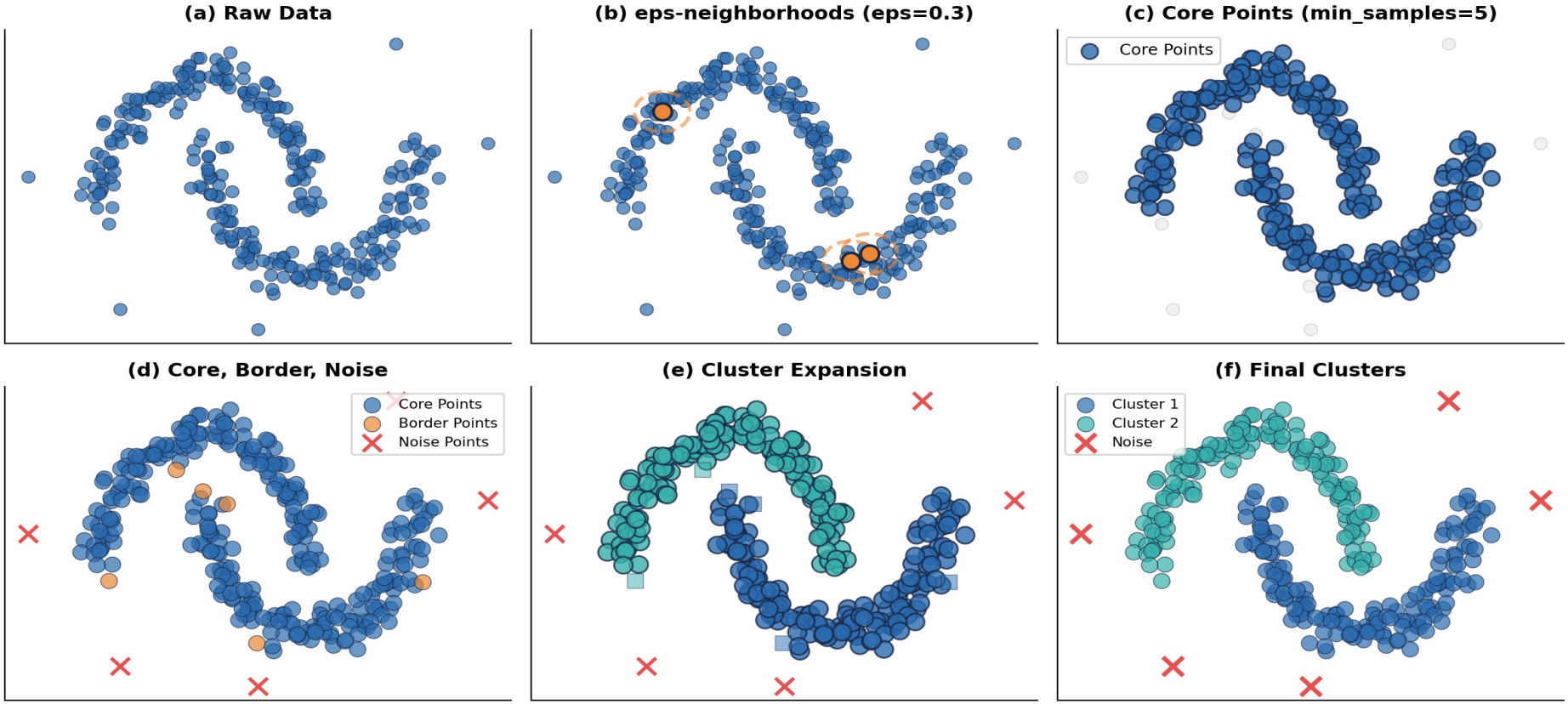
Radius of neighborhood.
Too small = everything is noise.
Too large = one big cluster.

### `min_samples`

Minimum points in eps-neighborhood to be a core point. Higher = stricter, fewer clusters.

**Tip:** No need to specify K upfront! DBSCAN discovers the number of clusters from the density structure.

# DBSCAN — Algorithm Walkthrough




*Start from any unvisited core point → expand by adding density-connected neighbors → repeat until all points classified*

# DBSCAN — Core, Border & Noise Points

## Core Point

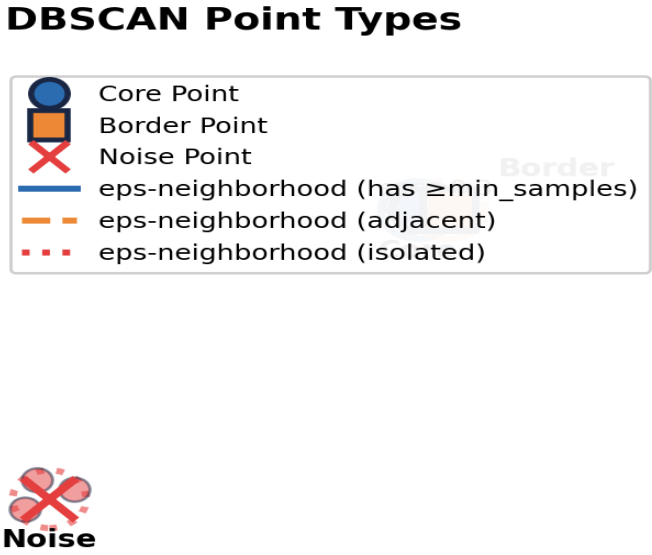
≥ min_samples neighbors within eps radius. Forms the backbone of clusters.

## Border Point

Within eps of a core point, but has fewer than min_samples neighbors. Belongs to cluster of its core neighbor.

## Noise Point

Neither core nor border. Isolated outlier (label = −1). Can be discarded or studied separately.




**Tip:** Algorithm: start from a core point, recursively add all density-connected points. Repeat until all classified.

# DBSCAN — Code & Parameter Tuning

```python
from sklearn.cluster import DBSCAN

db = DBSCAN(eps=0.3, min_samples=5)
labels = db.fit_predict(X_scaled)

n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
n_noise = (labels == -1).sum()
print(f"Clusters: {n_clusters}, Noise: {n_noise}")
```

### Tuning eps

k-distance plot: sort distances to k-th nearest neighbor. The elbow point → good eps value.

```python
# Tune eps: k-distance plot
from sklearn.neighbors import NearestNeighbors

nn = NearestNeighbors(n_neighbors=5).fit(X_scaled)
distances, _ = nn.kneighbors(X_scaled)
k_dist = np.sort(distances[:, -1])  # 5th-NN distance

plt.plot(k_dist)
plt.ylabel('5-NN distance')
plt.title('k-distance plot')
plt.show()
```

# DBSCAN vs K-Means

| | K-Means | DBSCAN |
|---|---|---|
| **Cluster shape** | Spherical only | Any shape |
| **Must specify K?** | Yes | No (auto) |
| **Handles outliers?** | No | Yes (label = -1) |
| **Varying density?** | OK | Struggles (single eps) |
| **Scalability** | O(nKi) — fast | $O(n^2)$ or O(n log n) |
| **Deterministic?** | No (random init) | Yes (core points) |

# HDBSCAN — Adaptive Density-Based

## Why HDBSCAN?

**Problem:** DBSCAN uses a single global ε for the whole dataset. If clusters have different densities, one ε cannot work for all: dense clusters fragment or sparse ones vanish.

**Solution — HDBSCAN:** Still density-based (not hierarchical/dendrogram). Scans across all density thresholds and extracts clusters that persist, adapting to local density.

**Key idea:**
1. Build a mutual-reachability graph over all points
2. Construct a minimum spanning tree (MST)
3. Walk down the condensed cluster tree, pick clusters by stability

**Advantages over DBSCAN:**
- No ε parameter — only min_cluster_size
- Handles varying-density clusters naturally
- Produces soft cluster membership probabilities
- More robust to parameter choice than DBSCAN

# Hierarchical & Spectral Clustering

Dendrograms, graph-based methods, and beyond

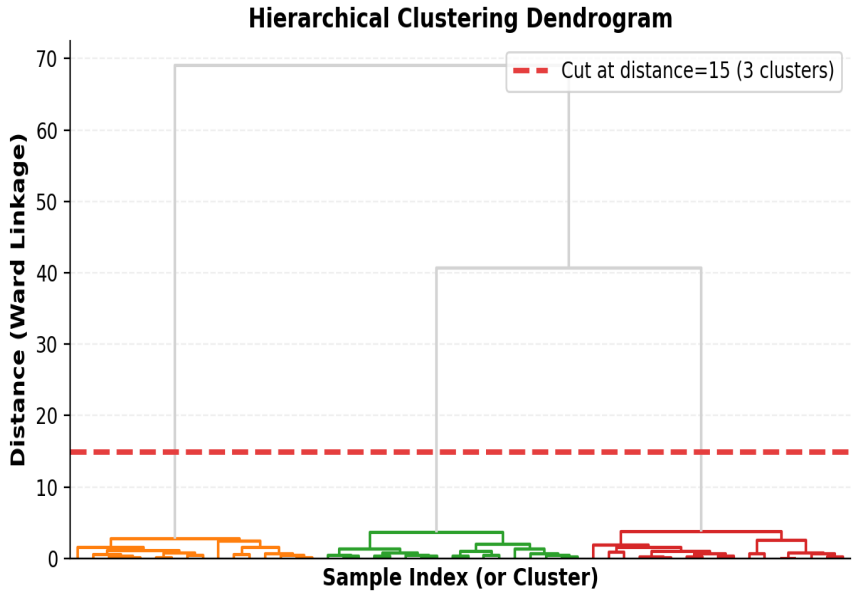
# Hierarchical Clustering & Dendrogram

## Agglomerative (Bottom-Up)

1. Start: each point = its own cluster
2. Merge two closest clusters at each step
3. Repeat until one cluster remains
4. Cut dendrogram at desired level for K

### Linkage Methods:

`ward:` minimize variance increase (default)
`complete:` max distance between clusters
`average:` mean distance (balanced)
`single:` min distance (elongated shapes)



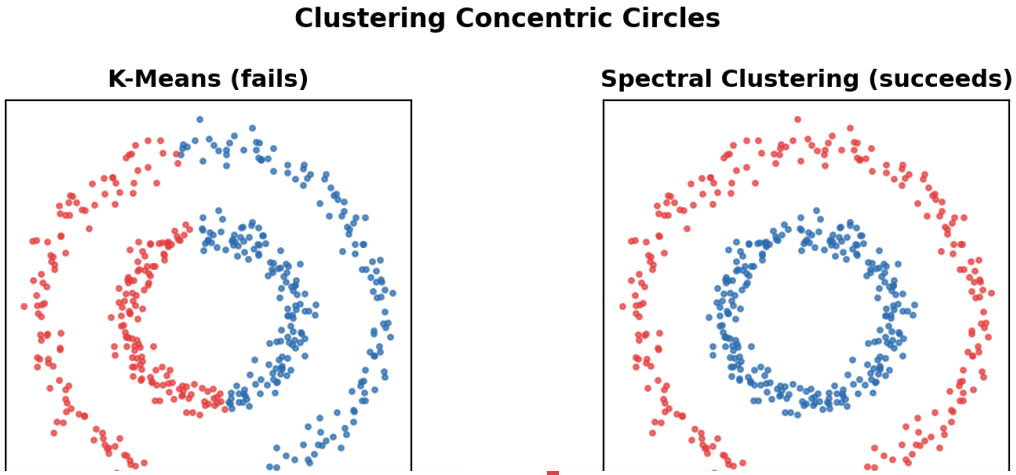

```
from scipy.cluster.hierarchy import dendrogram, linkage
from sklearn.cluster import AgglomerativeClustering

agg = AgglomerativeClustering(n_clusters=3, linkage='ward')
labels = agg.fit_predict(X_scaled)

Z = linkage(X_scaled, method='ward')
```

# Spectral Clustering

Uses the eigenvalues of a similarity (affinity) matrix to reduce dimensions, then clusters in the new space.




### Strengths

Handles non-convex shapes (circles, moons)
No assumption on cluster shape
Works via graph Laplacian eigenvectors

### Limitations

Must specify K upfront
$O(n^3)$ — slow for large datasets
Sensitive to affinity & n_neighbors

# Clustering Taxonomy — Core Ideas (1/2)

## Partitional Clustering

**Core Idea:** Divide data into K non-overlapping groups by minimizing distance to cluster centers. Each point belongs to exactly one cluster.

✓ **Strengths:** Fast, scalable, simple to implement and interpret
✗ **Weaknesses:** Must specify K, spherical cluster assumption, sensitive to outliers

**`K-Means`**
Classic. Inertia minimization via Lloyd's algorithm.
**`MiniBatchKMeans`**
Random batches → 10x faster for n > 10K.
**`K-Medoids (PAM)`**
Uses actual data points as centers. Robust to outliers.

## Density-Based Clustering

**Core Idea:** Clusters are dense regions separated by sparse regions. Points in low-density areas are outliers. No assumption about cluster shape.

✓ **Strengths:** Arbitrary shapes, auto K, built-in outlier detection
✗ **Weaknesses:** Sensitive to eps/bandwidth, struggles with varying density (except HDBSCAN)

**`DBSCAN`**
eps-neighborhood density. Single global eps.
**`HDBSCAN`**
Hierarchical DBSCAN. Adapts to local density.
**`OPTICS`**
Reachability ordering. Like DBSCAN but variable eps.
**`MeanShift`**
Shifts points to density peaks. Auto-discovers K.

# Clustering Taxonomy — Core Ideas (2/2)

## Model-Based

**Core Idea:** Data comes from a mixture of probability distributions. Fit parameters via EM.
`GMM, BayesianGMM`

✓ Soft assignment, elliptical shapes, BIC model selection
✗ Assumes Gaussian, sensitive to init, slow for high d

## Hierarchical

**Core Idea:** Build a tree (dendrogram) by merging closest clusters bottom-up. Cut at desired level.
`Agglomerative (4 linkage types)`

✓ No K needed upfront, dendrogram for exploration
✗ $O(n^2)$ memory, irreversible merges, no re-assignment

## Spectral

**Core Idea:** Build affinity graph → compute Laplacian eigenvectors → cluster in spectral space.
`SpectralClustering`

✓ Non-convex shapes (circles, moons, spirals)
✗ $O(n^3)$ slow, must specify K, sensitive to graph params

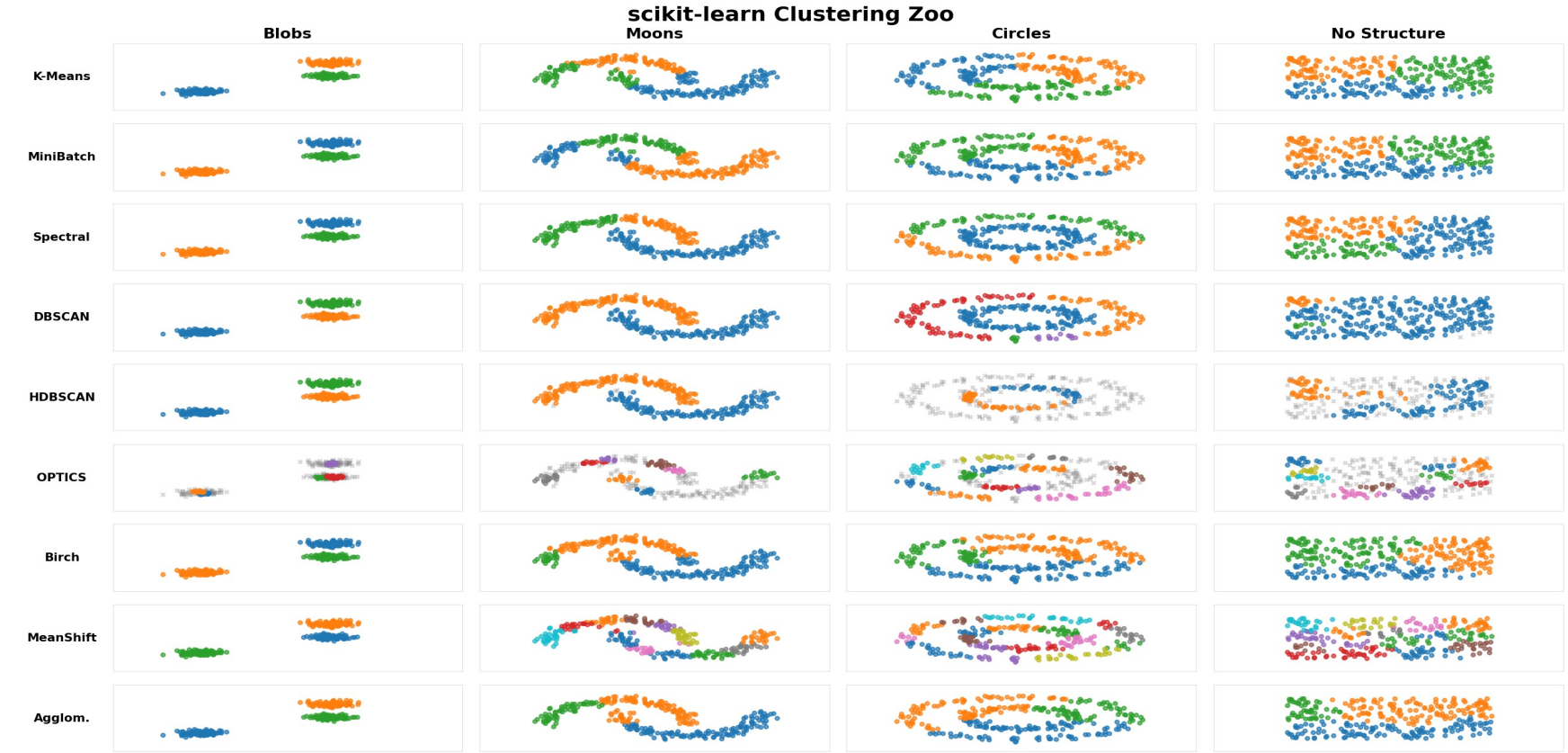
## Other Approaches

`BIRCH` — CF-tree for streaming/very large data. Scalable but spherical bias.

`AffinityPropagation` — Message-passing between data points. Auto-discovers K using exemplars.

*Each has its niche — no single algorithm wins everywhere.*

# scikit-learn Clustering Zoo




*Rows = algorithms, Columns = data shapes. No single algorithm wins everywhere.*

# More sklearn Clustering Algorithms

## MeanShift

Density-based, auto-discovers K via bandwidth parameter. Shifts points to density peaks.

*No K needed, slow $O(n^2)$*

## OPTICS

Like DBSCAN but handles variable density. Produces reachability plot instead of flat clusters.

*No single eps, ordering-based*

## BIRCH

Incremental, memory-efficient for very large datasets. Builds a CF-tree. Good as preprocessing step.

*Scalable, spherical bias*

## AffinityPropagation

Message-passing between data points.
Auto-discovers K from data.
Uses exemplars (real data points).

*Auto K, slow $O(n^2)$*

## MiniBatchKMeans

K-Means with random mini-batches.
10x faster for large n (>10K).
Slightly worse inertia.

*Fast, approx. K-Means*

## AgglomerativeClustering

Bottom-up hierarchical merging.
4 linkage options (ward, complete, average, single). Dendrogram.

*Flexible, $O(n^2)$ memory*

# Algorithm Comparison

| Algorithm | Shape | K? | Outliers? | Soft? | Scalability | Best For |
|---|---|---|---|---|---|---|
| K-Means | Spherical | Yes | ✗ | ✗ | O(nKi) fast | Baseline, large data |
| MiniBatch KM | Spherical | Yes | ✗ | ✗ | Very fast | n > 10K |
| GMM | Elliptical | Yes | ✗ | ✓ | $O(nK^2d)$ | Probabilities needed |
| DBSCAN | Arbitrary | No | ✓ | ✗ | O(n log n) | Non-convex + outliers |
| HDBSCAN | Arbitrary | No | ✓ | ✗ | O(n log n) | Varying density |
| OPTICS | Arbitrary | No | ✓ | ✗ | O(n log n) | Reachability analysis |
| Agglomerative | Depends | Optional | ✗ | ✗ | $O(n^2)$ | Dendrogram exploration |
| Spectral | Arbitrary | Yes | ✗ | ✗ | $O(n^3)$ | Non-convex, small n |
| MeanShift | Arbitrary | No | ✗ | ✗ | $O(n^2)$ | Auto K, density peaks |
| BIRCH | Spherical | Yes | ✗ | ✗ | O(n) | Very large data, stream |
| AffinityProp | Arbitrary | No | ✗ | ✗ | $O(n^2)$ | Exemplar-based, auto K |

## Decision Guide

Quick baseline? → **K-Means**

Non-convex shapes? → **DBSCAN / HDBSCAN**

Need probabilities? → **GMM**

Varying density? → **HDBSCAN**

Explore hierarchy? → **Agglomerative**

# Evaluation & Visualization

How to assess clusters without labels

# Internal Metrics (No Labels Needed)

### Silhouette

`(b−a) / max(a,b)`

Cohesion vs. separation
Range [−1, 1]. Higher = better

### Inertia (SSE)

`Σ||xi−μk||²`

Sum of squared distances
Lower = tighter (K-Means only)

### Calinski-Harabasz

`between / within`

Between vs. within variance
Higher = better

### Davies-Bouldin

`avg similarity`

Avg cluster pair similarity
Lower = better

```python
from sklearn.metrics import (silhouette_score,
    calinski_harabasz_score, davies_bouldin_score)

print(f"Silhouette:  {silhouette_score(X, labels):.3f}")
print(f"Calinski-H:  {calinski_harabasz_score(X, labels):.1f}")
print(f"Davies-B:    {davies_bouldin_score(X, labels):.3f}")
```

# External Metrics (When Labels Available)

If you have ground truth labels (e.g., for benchmarking), use these to measure agreement.

| Metric | Range | What It Measures |
|---|---|---|
| Adjusted Rand Index (ARI) | −0.5 to 1.0 | Agreement between clusterings, adjusted for chance |
| Normalized Mutual Info (NMI) | 0 to 1.0 | Shared information between predicted and true labels |
| V-Measure | 0 to 1.0 | Harmonic mean of homogeneity and completeness |

```python
from sklearn.metrics import (adjusted_rand_score,
    normalized_mutual_info_score, v_measure_score)

# y_true = actual labels, labels = cluster labels
ari = adjusted_rand_score(y_true, labels)
nmi = normalized_mutual_info_score(y_true, labels)
vm  = v_measure_score(y_true, labels)
print(f"ARI={ari:.3f}  NMI={nmi:.3f}  V={vm:.3f}")
```

*Note: cluster labels are permutation-invariant — cluster 0 ≠ class 0. ARI/NMI handle this automatically.*

# Common Pitfalls in Clustering

### Forgetting to Scale

Different units dominate distance.
Always StandardScaler first.

### K by Inertia Alone

Inertia always decreases.
Use silhouette + domain knowledge.

### Ignoring Domain Knowledge

Metrics are guides, not answers.
Best K should make business sense.

### High Dims Without PCA

Distance becomes meaningless.
Reduce with PCA first.

### Clustering With Labels

If you have labels, use classification.
Clustering is for discovery.

### Over-interpreting

Algorithms always find clusters —
even in uniform data. Validate!

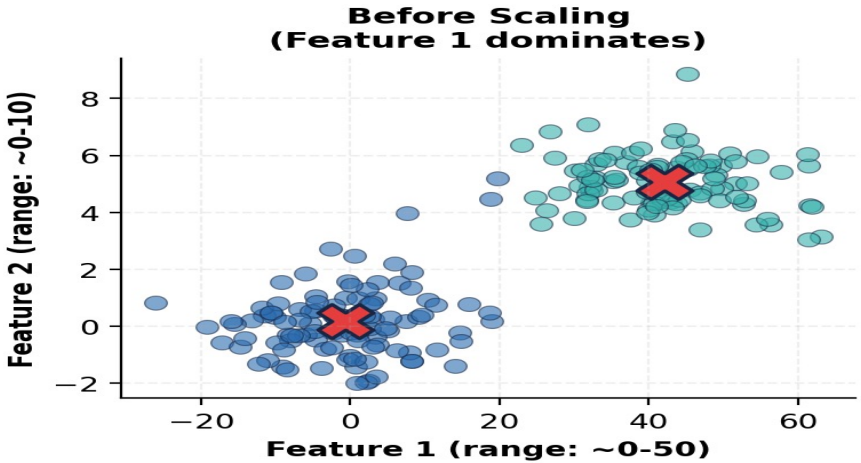
**Clustering is exploratory — there is no single "correct" answer. Iterate!**

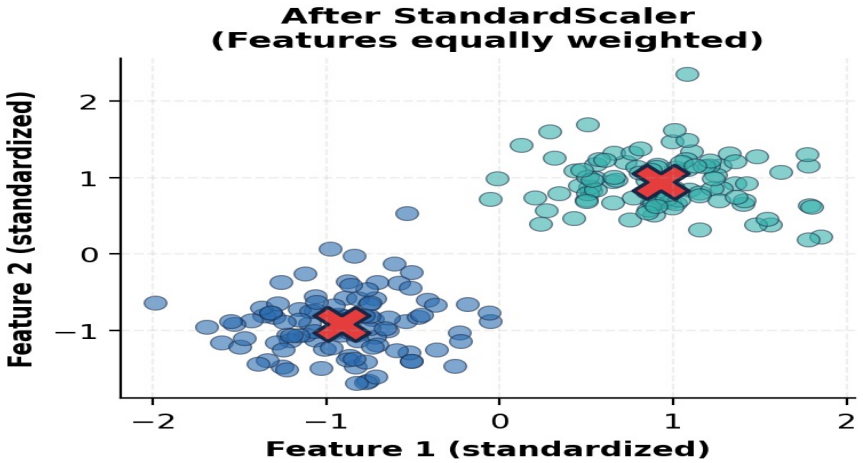
# Feature Scaling & Pipelines

Building robust clustering workflows

# Feature Scaling — Critical for Clustering!

Clustering algorithms use distance. Unscaled features bias the distance calculation.




**StandardScaler**

$z = (x - \mu) / \sigma$

Mean=0, Std=1. Default choice.

**MinMaxScaler**

$x' = (x - \min) / (\max - \min)$

Scale to [0,1]. Good for DBSCAN eps.

# Clustering Pipeline




```python
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans

# Scale -> Reduce -> Cluster
pipe = make_pipeline(
    StandardScaler(),
    PCA(n_components=10),
    KMeans(n_clusters=4, n_init=10, random_state=42)
)
pipe.fit(X)
labels = pipe[-1].labels_

# Silhouette on PCA-transformed data
X_processed = pipe[:-1].transform(X)
print(f"Silhouette: {silhouette_score(X_processed, labels):.3f}")
```

# Putting It All Together

Full example with algorithm comparison

# Customer Segmentation — Setup & Scaling (1/3)

```python
import pandas as pd, numpy as np
from sklearn.datasets import make_blobs
from sklearn.preprocessing import StandardScaler

# Generate sample customer data
X, y_true = make_blobs(n_samples=1000, n_features=5,
    centers=4, cluster_std=[1.0, 1.5, 0.8, 1.2],
    random_state=42)
feature_names = ['recency', 'frequency', 'monetary',
                 'avg_order', 'tenure']
df = pd.DataFrame(X, columns=feature_names)

# Scale features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(df)
```

## RFM Framework

**Recency:** how recently they purchased

**Frequency:** how often they buy

**Monetary:** how much they spend

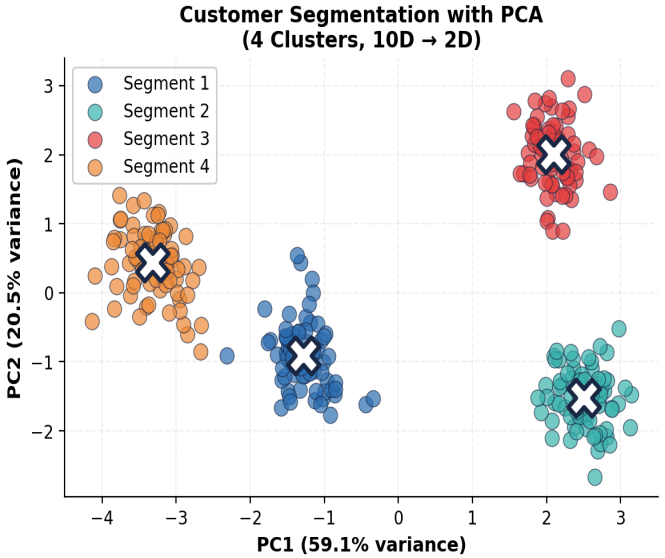
*Classic customer segmentation framework.*

# Customer Segmentation — Compare (2/3)

```python
from sklearn.cluster import KMeans, DBSCAN
from sklearn.mixture import GaussianMixture
from sklearn.metrics import silhouette_score

# Find optimal K
for k in range(2, 8):
    km = KMeans(n_clusters=k, n_init=10,
                random_state=42).fit(X_scaled)
    sc = silhouette_score(X_scaled, km.labels_)
    print(f"K={k}: silhouette={sc:.3f}")

# Compare algorithms (K=4)
models = {
    'KMeans': KMeans(n_clusters=4, n_init=10, random_state=42),
    'GMM':    GaussianMixture(n_components=4, random_state=42),
    'DBSCAN': DBSCAN(eps=1.2, min_samples=10),
}
```




*Visualize with PCA to compare cluster shapes and separation.*

# Customer Segmentation — Profile (3/3)

```
# Analyze the best model's clusters
best = KMeans(n_clusters=4, n_init=10,
              random_state=42).fit(X_scaled)
df['cluster'] = best.labels_

# Per-cluster statistics
summary = df.groupby('cluster')[feature_names].mean()
summary['count'] = df.groupby('cluster').size()
print(summary.round(2))
```

## 3-Step Cluster Profiling

### 1 Compute Means

Per-cluster feature means
to identify distinguishing traits

### 2 Name Clusters

e.g., "High-value loyal"
"At-risk churning"

### 3 Validate

Check with domain expert.
Do sizes and names make sense?

# Key Takeaways

- Clustering is unsupervised — no labels needed, discover hidden structure
- K-Means is your baseline: fast, simple, but limited to spherical clusters
- K-Means is EM with hard assignments → GMM extends to soft (probabilistic)
- DBSCAN finds arbitrary shapes + outliers; HDBSCAN handles varying density
- GMM gives soft (probabilistic) assignments and models elliptical clusters
- Hierarchical clustering: no K needed — read the dendrogram to choose
- Feature scaling is mandatory — always StandardScaler before clustering
- Profile and name your clusters — interpretation is the key to value

# Quick Reference — Clustering Cheat Sheet

```
from sklearn.cluster import KMeans, DBSCAN
from sklearn.cluster import AgglomerativeClustering
from sklearn.mixture import GaussianMixture
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import silhouette_score
from sklearn.decomposition import PCA

# 1. Scale
X_s = StandardScaler().fit_transform(X)
# 2. Cluster
km  = KMeans(n_clusters=K, n_init=10, random_state=42).fit(X_s)
db  = DBSCAN(eps=0.5, min_samples=5).fit(X_s)
gmm = GaussianMixture(n_components=K, random_state=42).fit(X_s)
# 3. Evaluate
print(silhouette_score(X_s, km.labels_))
# 4. Visualize
X_2d = PCA(n_components=2).fit_transform(X_s)
plt.scatter(X_2d[:,0], X_2d[:,1], c=km.labels_, cmap='tab10')
```

## The Workflow

1. Scale
2. Cluster
3. Evaluate
4. Visualize
5. Interpret