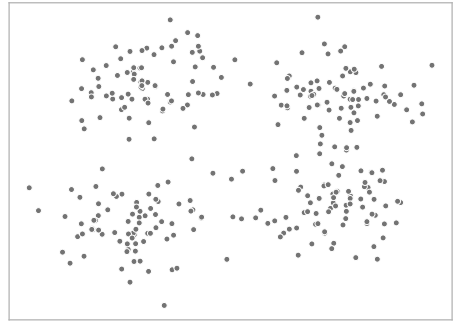
Unsupervised learning: no labels, just structure

Until now every task had a target y (rent, "bad cheese", a digit). Now we get **only the features x** — no labels — and ask:

Are there natural groups in this data?

- customer segments (marketing)
- topics in a pile of documents
- colors in an image
- cell types in gene-expression data

Unlabeled data: how many groups? which?



Clustering: partition the data into groups so points in a group are similar, and points in different groups are not.

Where clustering sits

Supervised (had y)

- regression, classification
- learn a map $\mathbf{x} \to y$

Unsupervised (no y)

- **clustering** — group the *rows* (samples)
- dimensionality reduction — compress the *columns* (features) [next deck]

You can have labels and still cluster:

- find *sub-structure* inside a labeled class;
- QA the labels / spot mislabeled points;
- build cluster-id *features* for a supervised model;
- semi-supervised setups.

(That is also when the *external* metrics later apply.)

Outline

K-means

The workhorse: pick k, alternate "assign then update" until the centers stop moving.

First: how do we measure "similar"?

Clustering needs a **distance**. The choice shapes the clusters.

- **Euclidean** $\|\mathbf{x} - \mathbf{x}'\|_2$ — the default (k-means uses it).
- **Manhattan** $\|\mathbf{x} - \mathbf{x}'\|_1$ — robust to large single-feature gaps.
- **Cosine** — angle, not magnitude (text / embeddings).

Trap — scale first. Distances are dominated by large-range features. A feature in "salary" (10,000s) crushes one in "years" (single digits). **Standardize** before any distance-based clustering, unless features already share units.

K-means: the objective

Choose k. Represent each cluster by a **centroid** $\boldsymbol{\mu}_j$. Minimize the **within-cluster sum of squares** (inertia):

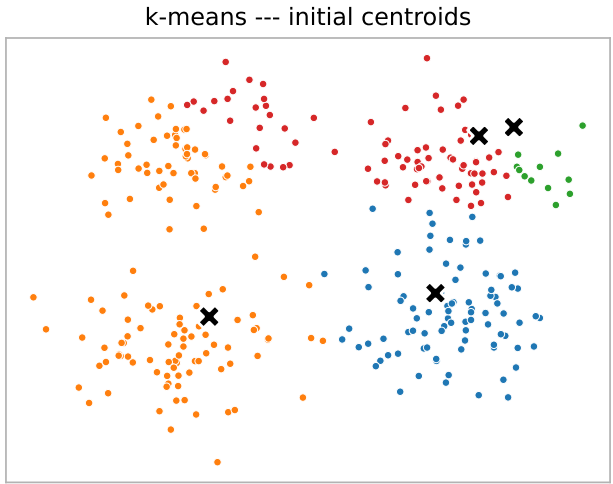
$$\text{WCSS} = \sum_{j=1}^{k} \sum_{\mathbf{x} \in C_j} \|\mathbf{x} - \boldsymbol{\mu}_j\|^2.$$

- Each point joins its nearest centroid; the centroid is the cluster's mean.
- You must **pick** k in advance (more on that soon).

Inertia is not normalized. It falls as k grows and scales with the data, so its absolute value means nothing — only useful for *relative* comparison (e.g. the elbow).

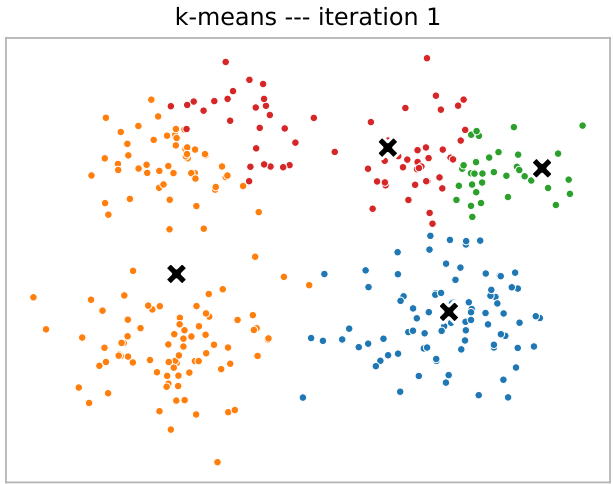
Lloyd's algorithm: assign, update, repeat

Two steps, each of which can only *lower* the WCSS (this is coordinate descent): **assign** each point to its nearest centroid, then **update** each centroid to its cluster's mean. Watch it converge:



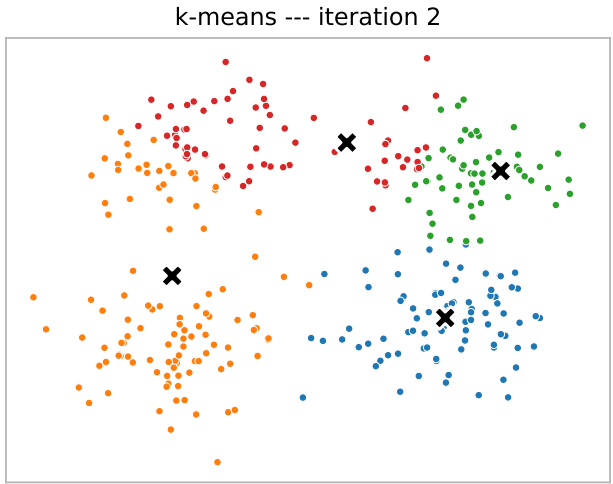
Lloyd's algorithm: assign, update, repeat

Two steps, each of which can only *lower* the WCSS (this is coordinate descent): **assign** each point to its nearest centroid, then **update** each centroid to its cluster's mean. Watch it converge:



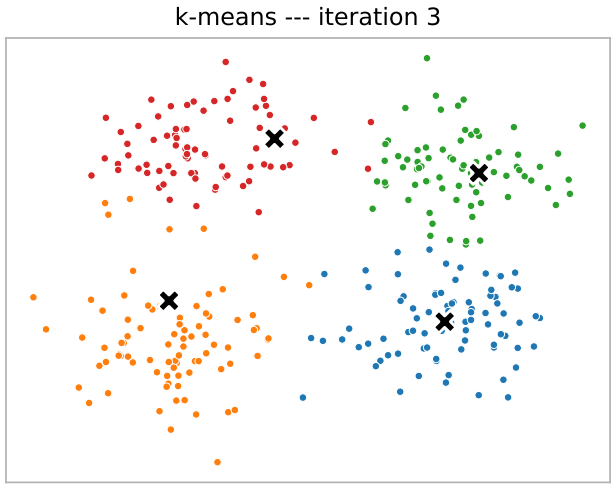
Lloyd's algorithm: assign, update, repeat

Two steps, each of which can only *lower* the WCSS (this is coordinate descent): **assign** each point to its nearest centroid, then **update** each centroid to its cluster's mean. Watch it converge:



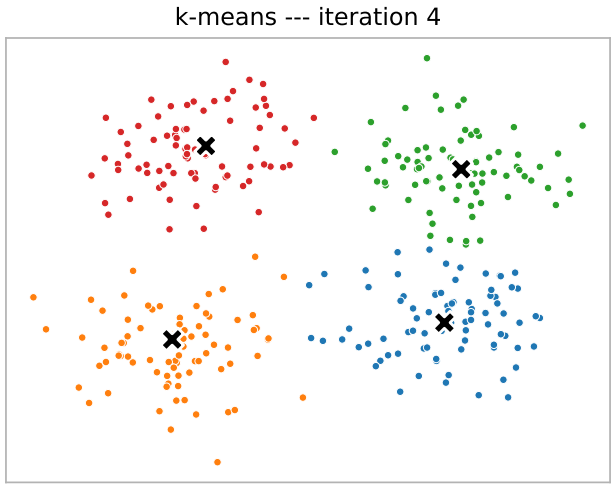
Lloyd's algorithm: assign, update, repeat

Two steps, each of which can only *lower* the WCSS (this is coordinate descent): **assign** each point to its nearest centroid, then **update** each centroid to its cluster's mean. Watch it converge:



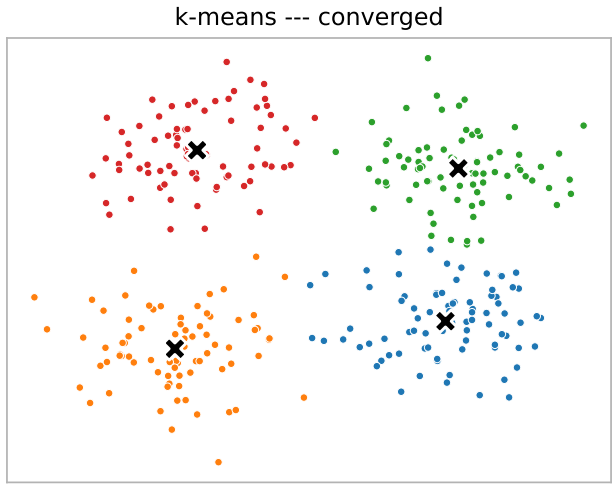
Lloyd's algorithm: assign, update, repeat

Two steps, each of which can only *lower* the WCSS (this is coordinate descent): **assign** each point to its nearest centroid, then **update** each centroid to its cluster's mean. Watch it converge:



Lloyd's algorithm: assign, update, repeat

Two steps, each of which can only *lower* the WCSS (this is coordinate descent): **assign** each point to its nearest centroid, then **update** each centroid to its cluster's mean. Watch it converge:



Worked iteration (by hand), $k = 2$

Points
$P_1(1,1), P_2(2,1), P_3(4,3), P_4(5,4), P_5(2,2)$; init centroids (placed freely, *not* on data points)
$\boldsymbol{\mu}_1 = (1.5, 1),\ \boldsymbol{\mu}_2 = (4, 4)$.

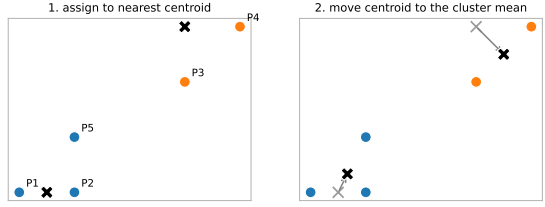
Assign (nearest centroid): $P_1, P_2, P_5 \to \mu_1$; $P_3, P_4 \to \mu_2$.

Update (cluster means):

$$\boldsymbol{\mu}_1 = \tfrac{(1,1)+(2,1)+(2,2)}{3} = (1.67, 1.33)$$

$$\boldsymbol{\mu}_2 = \tfrac{(4,3)+(5,4)}{2} = (4.5, 3.5)$$

Then repeat until nothing moves.



Predict-first: does initialization matter?

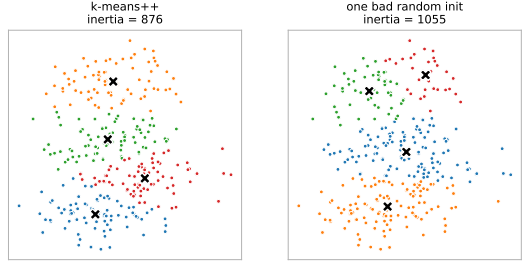
Run k-means twice from two *different* random starts. Same answer?

Predict-first: does initialization matter?

Run k-means twice from two *different* random starts. Same answer?

No. It only finds a *local* optimum — a bad start gives a worse clustering (higher inertia).

Fix: k-means++ seeds centroids spread far apart, then run several times (`n_init`) and keep the lowest-inertia result.

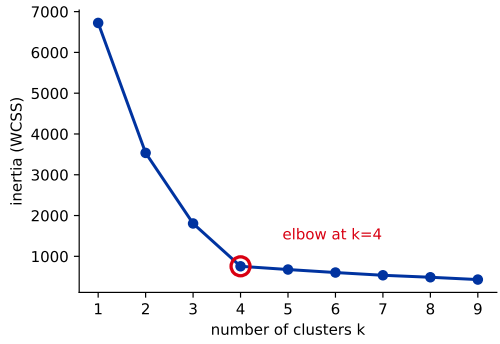


Choosing k: the elbow

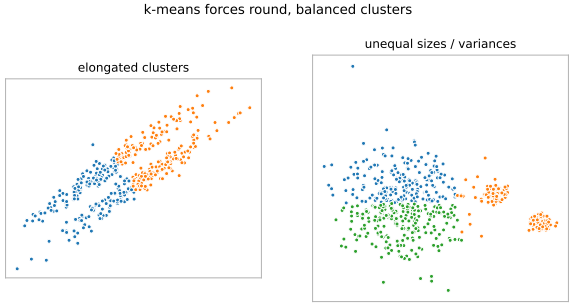
Plot inertia vs k. It always decreases; look for the **elbow** where extra clusters stop paying off.

- Below the elbow: real structure being captured.
- Past it: diminishing returns (splitting real clusters).

The elbow is a heuristic — often fuzzy. The **silhouette** (evaluation section) gives a more principled choice.



Assumptions = failure modes



K-means assumes **round, similarly-sized, convex** clusters — it breaks on elongated shapes and on clusters of very different size/variance.

Reach for k-means when: large n, round/equal clusters, you can pick k, fast baseline.

Mini-batch k-means: scaling up

For very large n, update centroids from small random **mini-batches** instead of the full data each step.

- Much faster, far less memory; streaming-friendly.
- Slightly worse clusters than full k-means — usually a fine trade.

When: same problem as k-means, but n is huge.

From hard to soft

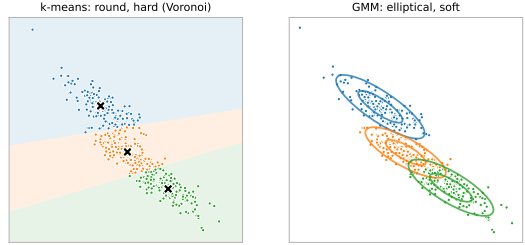
K-means says "you ARE in cluster A". A Gaussian mixture says "70% A, 30% B" — and allows elliptical clusters.

Gaussian mixtures: soft, elliptical clusters

Model each cluster as a **Gaussian** with its own mean *and* covariance, so clusters can be **elliptical** and different sizes. Each point gets a **soft** membership (responsibility) in every cluster.

K-means is the special case of a GMM with *spherical, equal-variance* covariances and *hard* (winner-take-all) assignments.

When: overlapping / elliptical clusters, soft memberships, or density estimation.



EM, step 1: the E-step (responsibilities)

Chicken-and-egg: we need assignments to fit the Gaussians, but the Gaussians to assign points. EM breaks it by **alternating**. Start from random Gaussians, then:

E-step — soft-assign every point. For point i and cluster k, the **responsibility** is the posterior probability it belongs to k:

$$\gamma_{ik} \;=\; \frac{\pi_k \,\mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \,\mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}.$$

- π_k is the cluster's weight (its prior size); $\mathcal{N}$ is the Gaussian density.
- $\gamma_{ik} \in [0, 1]$ and $\sum_k \gamma_{ik} = 1$ — a soft membership, not a hard label.

Compare k-means: its "assign" step is the *hard* version — $\gamma_{ik} \in \{0, 1\}$.

EM, step 2: the M-step, and convergence

M-step — refit each Gaussian, weighting every point by its responsibility. With $N_k = \sum_i \gamma_{ik}$ (the soft count in cluster k):

$$\boldsymbol{\mu}_k = \frac{1}{N_k} \sum_i \gamma_{ik}\, \mathbf{x}_i, \quad \boldsymbol{\Sigma}_k = \frac{1}{N_k} \sum_i \gamma_{ik} (\mathbf{x}_i - \boldsymbol{\mu}_k)(\mathbf{x}_i - \boldsymbol{\mu}_k)^\top, \quad \pi_k = \frac{N_k}{n}.$$

Each is the ordinary mean / covariance / proportion, just *responsibility-weighted*.

- **Repeat** E $\rightarrow$ M until the log-likelihood stops rising. Each round increases it $\Rightarrow$ convergence to a **local** optimum (so, like k-means, run several inits).

Pick the number of components with **BIC / AIC** — likelihood criteria (they need a probabilistic model, so they work for GMM, *not* for plain k-means). EM recurs across ML wherever there are latent variables.

Beyond k-means

Density, hierarchy, and robust centers — for clusters that are not round blobs.

K-medoids (PAM)

Like k-means, but the center is an **actual data point** — a **medoid** — chosen to minimize the total distance to its cluster (it never *averages* points).

- **Any distance / dissimilarity** works (cosine, Gower for mixed types, edit distance…) — because it never has to compute a mean.
- **Robust to outliers** — a real central point, not a mean dragged by extremes.
- **Interpretable center** — "this customer is typical of the segment".

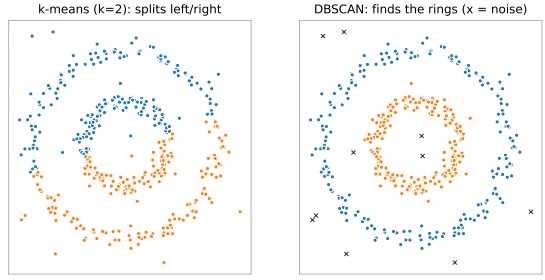
The real difference from k-means is the *medoid* and the *free choice of metric* — not the exact objective. *When:* non-Euclidean data, outliers, interpretable centers. *Not in core sklearn* — `sklearn_extra.cluster.KMedoids`.

DBSCAN: the three kinds of point

Clusters are **dense regions** separated by sparse ones. Two knobs: radius `eps` (ϵ) and `min_samples`.

- **core**: $\geq$ `min_samples` points within ϵ;
- **border**: within ϵ of a core, but not dense itself;
- **noise**: neither — an outlier.

Finds **arbitrary shapes**, needs **no** k, and **flags outliers**.

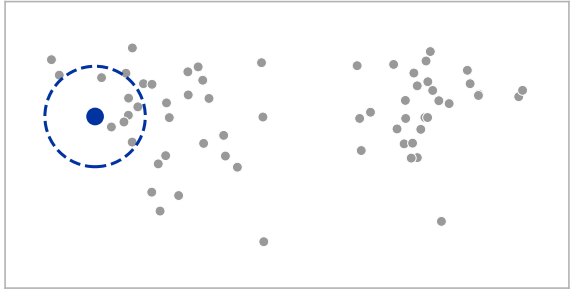


Predict-first: what can DBSCAN do that k-means/GMM cannot? *Non-convex shapes, and automatic noise/outlier flagging.*

DBSCAN, step by step

A core point "grows" a cluster by absorbing everything density-reachable from it:

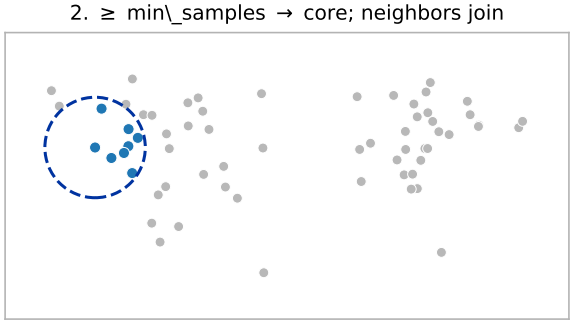
1. pick a point: count neighbors within ε



Core points chain together into a cluster; border points attach; whatever is left in sparse regions is **noise** (-1).

DBSCAN, step by step

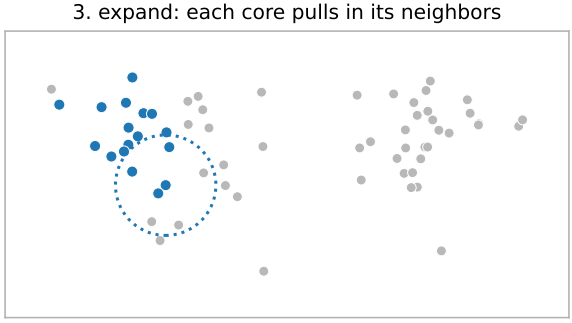
A core point "grows" a cluster by absorbing everything density-reachable from it:



Core points chain together into a cluster; border points attach; whatever is left in sparse regions is **noise** (-1).

DBSCAN, step by step

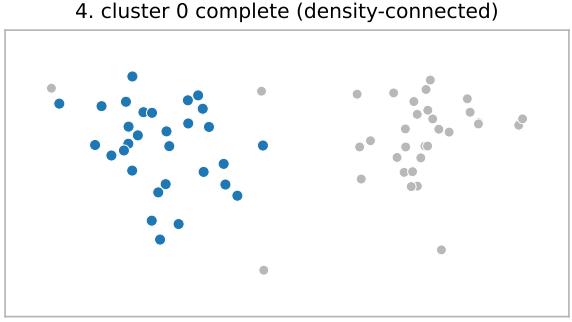
A core point "grows" a cluster by absorbing everything density-reachable from it:



Core points chain together into a cluster; border points attach; whatever is left in sparse regions is **noise** (-1).

DBSCAN, step by step

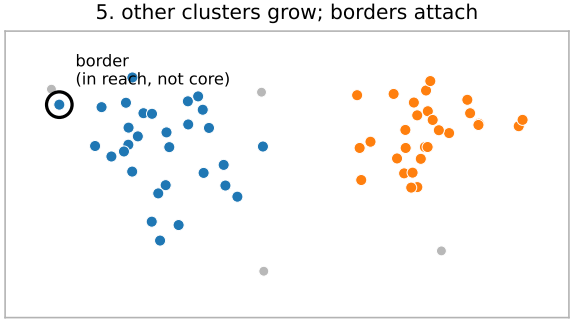
A core point "grows" a cluster by absorbing everything density-reachable from it:



Core points chain together into a cluster; border points attach; whatever is left in sparse regions is **noise** (-1).

DBSCAN, step by step

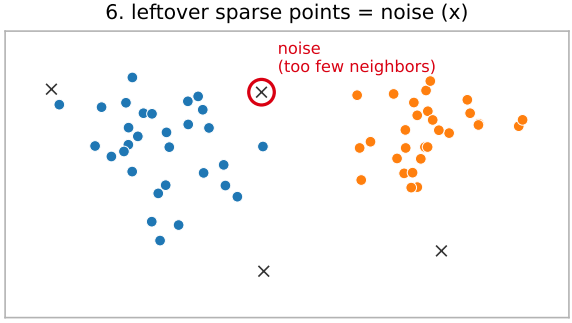
A core point "grows" a cluster by absorbing everything density-reachable from it:



Core points chain together into a cluster; border points attach; whatever is left in sparse regions is **noise** (-1).

DBSCAN, step by step

A core point "grows" a cluster by absorbing everything density-reachable from it:



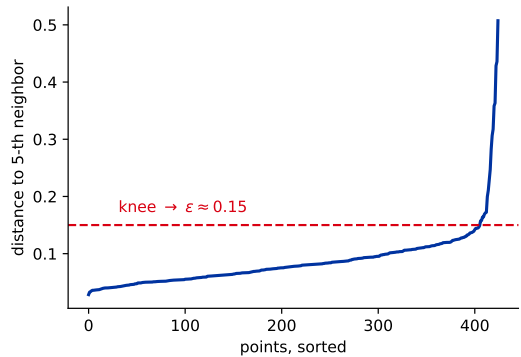
Core points chain together into a cluster; border points attach; whatever is left in sparse regions is **noise** (-1).

DBSCAN: choosing eps and min_samples

min_samples: rule of thumb $\approx 2\times$ the number of features ($\geq D+1$, and at least 3); raise it for large or noisy data.

eps: make a ***k*-distance plot** — sort every point's distance to its k-th neighbor ($k =$ min_samples) and look for the **knee**. The distance there is a good eps.

The two are coupled: change one and you usually re-tune the other.



Limits: one global eps can't fit clusters of very different density; the naive algorithm is $O(n^2)$. Heuristics: Sander et al.; see the routine in stats.stackexchange.com/q/88872.

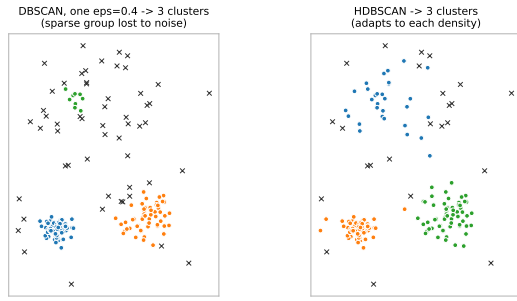
HDBSCAN: density handled automatically

DBSCAN needs one `eps`, and a *single* `eps` cannot fit clusters of **different density** — left, the sparse top group is lost to noise.

HDBSCAN fixes this — and crucially **you do not hand it a density or a range**:

- it *scans all density levels itself* and keeps the clusters that **persist** (are most stable);
- **no** `eps` — you set only `min_cluster_size` (smallest group worth calling a cluster);
- so each cluster can be dense at **its own scale**.

Core sklearn since 1.3 (`sklearn.cluster.HDBSCAN`).



HDBSCAN: how it works (sketch)

1. **Core distance** of a point = distance to its `min_samples`-th neighbor (how dense it is locally).
2. **Mutual reachability distance** between two points = the max of their two core distances and their actual distance — this pushes sparse points apart.
3. Build the **minimum spanning tree** of that distance, then sweep a threshold from low to high to get a **hierarchy** (a tree) of clusters.
4. **Condense** the tree and keep the clusters that **persist** over the widest range of densities (most "stable") — those are the output; the rest is noise.

The payoff: clusters chosen by *stability*, not by one hand-picked radius.

Hierarchical (agglomerative) clustering

Bottom-up — build clusters by repeated merging:

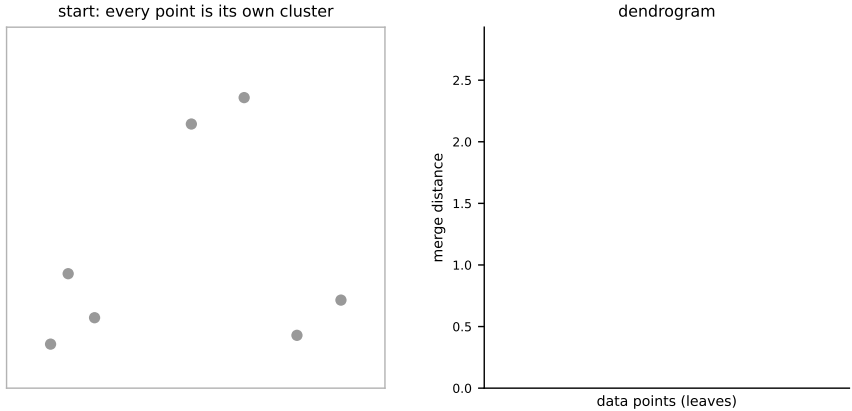
1. Start: every point is its own cluster (n of them).
2. Find the **two closest** clusters ("closest" = the *linkage*, defined shortly).
3. **Merge** them into one.
4. Repeat until a single cluster remains, recording each merge and its distance.

- No need to fix k up front — the merge history is a tree you *cut* afterwards.
- Gives **nested** structure (taxonomies).

When: nested structure / a dendrogram, exploring #clusters, connectivity constraints; smaller n.

Agglomerative, step by step

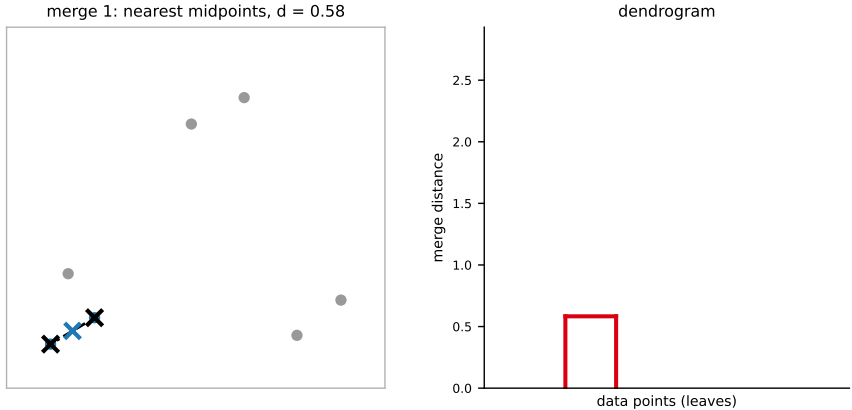
Merge the two clusters whose **midpoints** ($\times$) are closest. *Left:* points and cluster midpoints, dashed line = the current merge. *Right:* the dendrogram builds up, one bar per merge (latest in red):



Each merge distance d (between midpoints, here *centroid* linkage) becomes that bar's height in the dendrogram.

Agglomerative, step by step

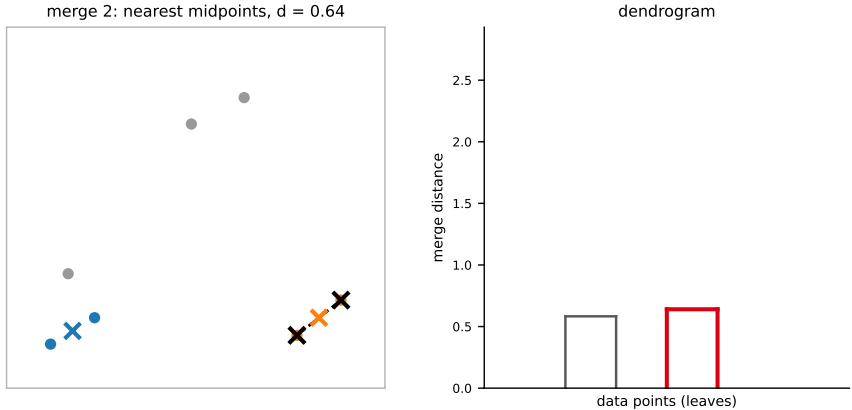
Merge the two clusters whose **midpoints** ($\times$) are closest. *Left:* points and cluster midpoints, dashed line = the current merge. *Right:* the dendrogram builds up, one bar per merge (latest in red):



Each merge distance d (between midpoints, here *centroid* linkage) becomes that bar's height in the dendrogram.

Agglomerative, step by step

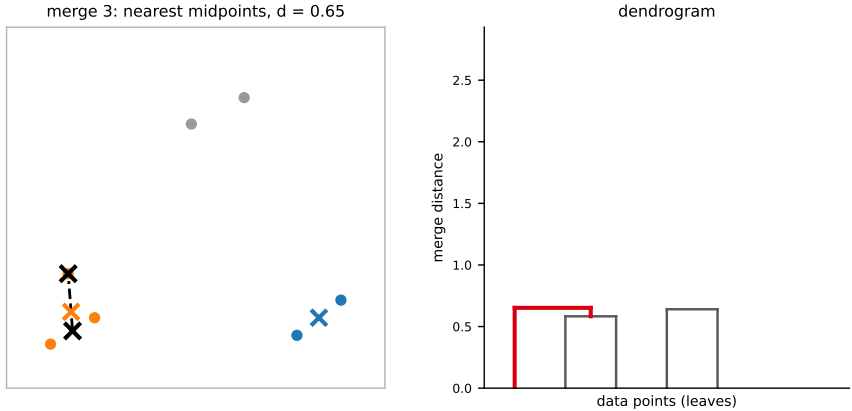
Merge the two clusters whose **midpoints** ($\times$) are closest. *Left:* points and cluster midpoints, dashed line = the current merge. *Right:* the dendrogram builds up, one bar per merge (latest in red):



Each merge distance d (between midpoints, here *centroid* linkage) becomes that bar's height in the dendrogram.

Agglomerative, step by step

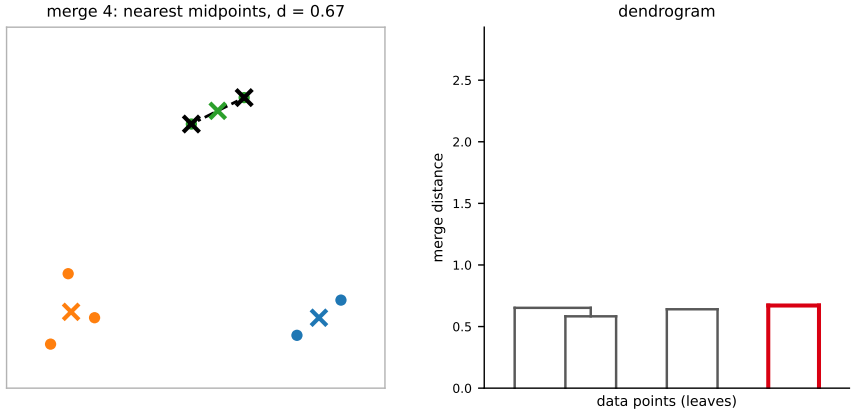
Merge the two clusters whose **midpoints** ($\times$) are closest. *Left:* points and cluster midpoints, dashed line = the current merge. *Right:* the dendrogram builds up, one bar per merge (latest in red):



Each merge distance d (between midpoints, here *centroid* linkage) becomes that bar's height in the dendrogram.

Agglomerative, step by step

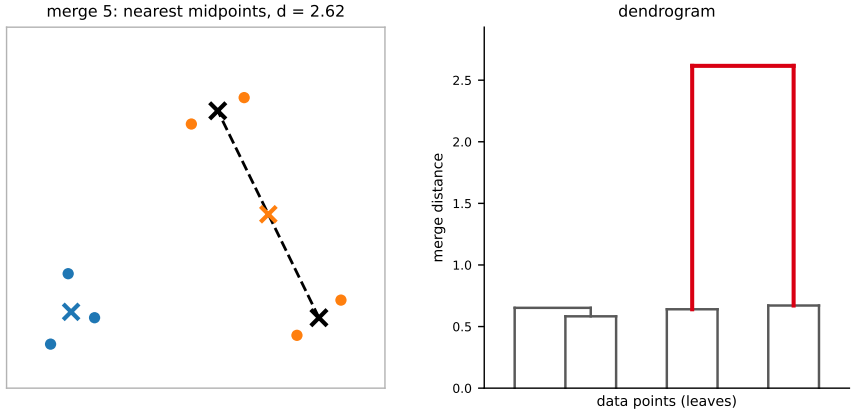
Merge the two clusters whose **midpoints** ($\times$) are closest. *Left:* points and cluster midpoints, dashed line = the current merge. *Right:* the dendrogram builds up, one bar per merge (latest in red):



Each merge distance d (between midpoints, here *centroid* linkage) becomes that bar's height in the dendrogram.

Agglomerative, step by step

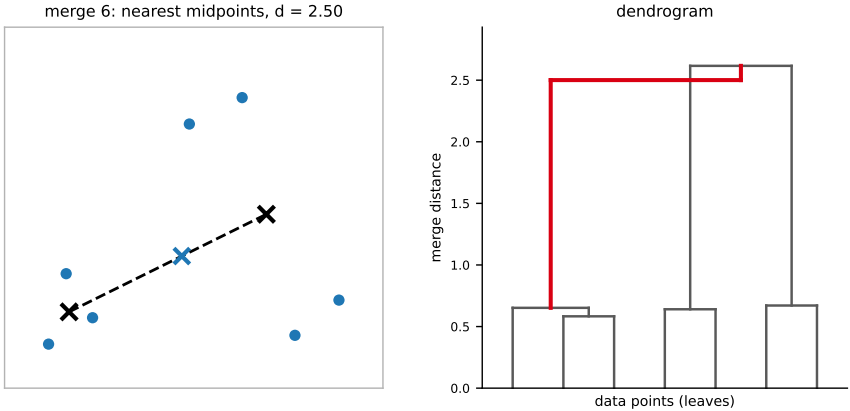
Merge the two clusters whose **midpoints** ($\times$) are closest. *Left:* points and cluster midpoints, dashed line = the current merge. *Right:* the dendrogram builds up, one bar per merge (latest in red):



Each merge distance d (between midpoints, here *centroid* linkage) becomes that bar's height in the dendrogram.

Agglomerative, step by step

Merge the two clusters whose **midpoints** ($\times$) are closest. *Left:* points and cluster midpoints, dashed line = the current merge. *Right:* the dendrogram builds up, one bar per merge (latest in red):

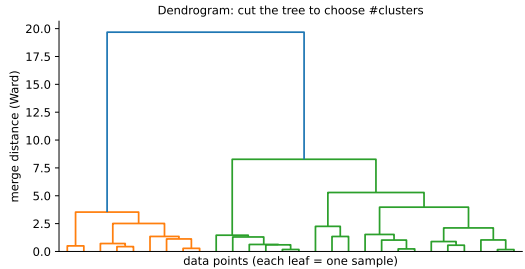


Each merge distance d (between midpoints, here *centroid* linkage) becomes that bar's height in the dendrogram.

The dendrogram: cut the tree

Every merge is a bar at its **merge distance**. **Cut** the tree at a height to read off clusters:

- low cut = many small clusters; high cut = few big ones;
- the largest vertical gap is a natural place to cut.

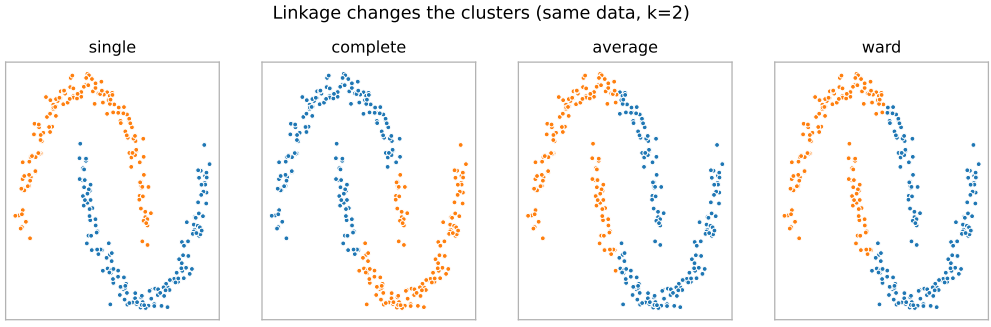


Linkage: what is the distance between two clusters?

"Merge the two closest clusters" — but *closest* can mean different things. Linkage is that definition:

- **Single** — distance between the *nearest* pair of points (one in each). Chains along shapes; can "snake".
- **Complete** — distance between the *farthest* pair. Makes compact, equal blobs.
- **Average** — mean over all cross-cluster pairs. A compromise.
- **Ward** — merge the pair that increases total within-cluster variance the least. The common default; behaves like k-means.

Linkage: same data, different answers



Two moons, $k = 2$: **single** linkage follows the crescents; **complete / average / Ward** cut them into compact halves. The linkage choice changes everything.

How good is a clustering?

The hard part: there is usually *no ground truth*. Internal metrics judge the shape; external ones need labels.

Two regimes: internal vs external

- **Internal** (no labels — the usual case): judge cohesion + separation from the data alone. Silhouette, Davies–Bouldin, Calinski–Harabasz, inertia/elbow.
- **External** (labels exist — the "labels but still cluster" cases): compare the clustering to known labels. ARI, AMI, V-measure.

There is no single "accuracy" for clustering — the cluster numbers are arbitrary, and usually there is nothing to be right *about*.

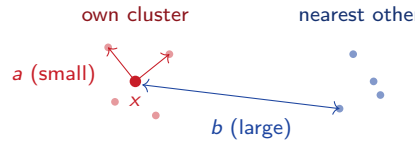
Silhouette: the idea (a vs b)

For one point: a = mean distance to its *own* cluster; b = mean distance to the *nearest other* cluster.

$$s = \frac{b - a}{\max(a, b)} \in [-1, 1]$$

- $s \approx 1$: snug in its cluster, far from others (ideal).
- $s \approx 0$: on a boundary; $s < 0$: probably misassigned.

Ideal: small, tight clusters that sit far apart $\Rightarrow$ small a, large b.

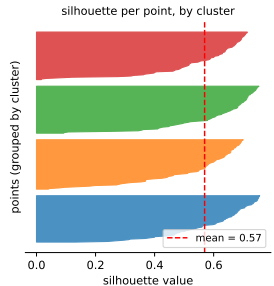


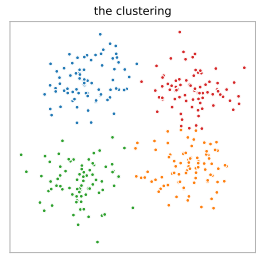
Silhouette: reading the plot

Each bar is one point's s, **grouped by cluster** and sorted; the dashed line is the **mean** silhouette.

- Wide, tall, mostly-right bars = a good cluster.
- A cluster dipping below 0 = likely merged/over-split.

Pick the k with the highest mean silhouette. (Also: Davies–Bouldin, lower better; Calinski–Harabasz, higher better.)





External metrics: when you do have labels

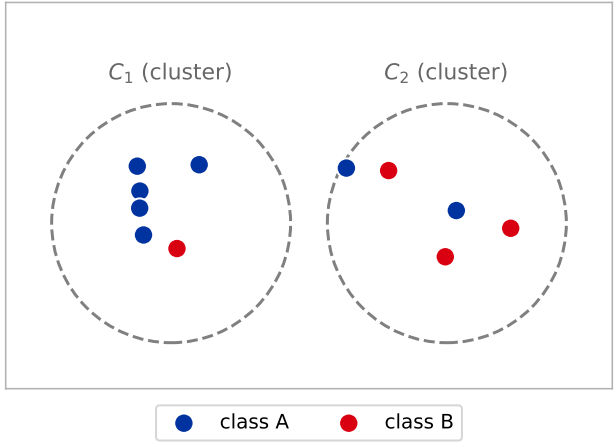
Compare the clustering to a known labeling (which cluster = which label does *not* matter):

- **ARI** (adjusted Rand index) and **AMI** (adjusted mutual information): **chance-adjusted** — 0 means "no better than random", 1 perfect.
- **V-measure** (homogeneity + completeness); also Fowlkes–Mallows.

Prefer the **adjusted** versions (ARI, AMI). Raw mutual information / NMI is not corrected for chance, so it rewards simply using more clusters. Plain accuracy fails — the labels are a *permutation* of the clusters.

The data behind the Rand index

Eleven points, each with a true **class** (color) and a **cluster** (the dashed groups). The next slide just *counts* these into a table:



Cluster C_1: 5 class-A + 1 class-B; cluster C_2: 2 class-A + 3 class-B.

Worked example: Rand index and ARI

11 points, each with a known **class** (A/B) and a **clustering** (C_1/C_2). The **contingency table** counts every combination — e.g. the **5** means five points are *both* class A *and* in cluster C_1:

	C_1	C_2	$\sum$
A	5	2	7
B	1	3	4
$\sum$	6	5	11

Rand counts **pairs** of points the two labelings *agree* on, out of all $\binom{11}{2} = 55$ pairs:

- same class & same cluster $= \binom{5}{2}+\binom{2}{2}+\binom{1}{2}+\binom{3}{2} = 10+1+0+3 = 14$
- same class $= \binom{7}{2}+\binom{4}{2} = 27$; same cluster $= \binom{6}{2}+\binom{5}{2} = 25$
- different in both $= 55 - 27 - 25 + 14 = 17$

Rand $= \dfrac{\text{agree}}{\text{all}} = \dfrac{14+17}{55} = \mathbf{0.56}$.

Adjust for chance: expected "same-both" under random labels $= \frac{27\cdot 25}{55} = 12.3$; max $= \frac{27+25}{2} = 26$. So ARI $= \dfrac{14-12.3}{26-12.3} = \mathbf{0.13}$. Rand looks okay (0.56), but ARI shows the clustering is *barely* above chance — which is why we prefer the **adjusted** index.

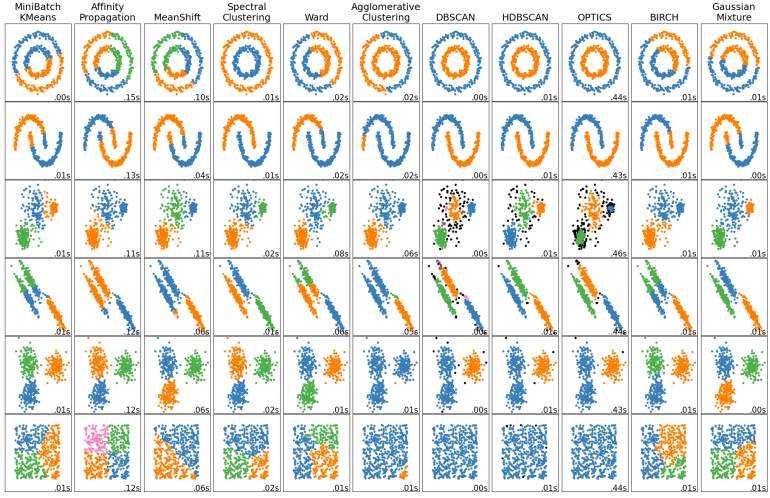
Evaluation in practice

- **Track more than one metric.** No single number captures a clustering — report, say, silhouette *and* Davies–Bouldin (and ARI/AMI if labels exist); trust the result only when they agree.
- Pick k by the **silhouette** (or the elbow), not by eye alone.
- Check **stability**: do clusters survive a re-run / a resample / a different seed?
- **Scale first** — every internal metric is distance-based too.
- Sanity-check by *looking* at the clusters.

Choosing, and a wider view

Many more algorithms exist — here is how they all behave, and the traps to avoid.

The wider zoo



Source: scikit-learn (BSD-3), clustering-algorithm comparison.

Comparison: which algorithm when?

Algorithm	**Needs k?**	**Cluster shape**	**Noise?**	**Scales?**
k-means / MiniBatch	yes	round, equal	no	very well
k-medoids	yes	any metric	somewhat	poorly
GMM (EM)	yes (+BIC)	elliptical	no	medium
DBSCAN	no	arbitrary	**yes**	medium
HDBSCAN	no	arbitrary, varying density	**yes**	medium
Hierarchical	no (cut tree)	flexible (linkage)	no	poorly

Use when: fast round blobs → k-means; odd shapes + outliers → DBSCAN/HDBSCAN; soft/elliptical → GMM; nested structure → hierarchical; robust/any-metric centers → k-medoids.

Bonus use: anomaly detection

Clustering doubles as outlier detection:

- **DBSCAN / HDBSCAN** label low-density points as **noise** (-1) — those are your anomalies, for free.
- **GMM** gives each point a **likelihood**; flag the lowest-probability points.

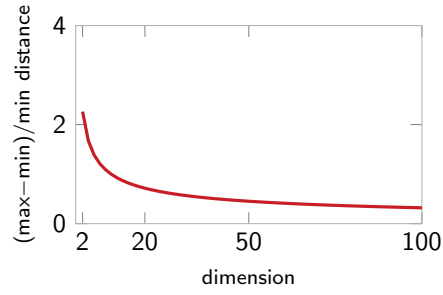
Useful when "normal" has structure but "weird" does not — fraud, faults, intrusions.

The curse of dimensionality

As dimensions grow, points spread out and **pairwise distances concentrate** — the nearest and farthest neighbors become almost equally far.

Distance-*based* clustering (all of today's methods) degrades: "near" stops meaning much.

Fix: **reduce dimensions first** (PCA), then cluster in the low-dimensional space.



Contrast between nearest and farthest collapses toward 0.

Cluster after PCA

A standard pipeline for high-dimensional data:

$$\text{standardize} \;\rightarrow\; \text{PCA (e.g. to 95\% variance)} \;\rightarrow\; \text{cluster.}$$

- Distances behave better; clustering is faster and often cleaner.
- You can finally *plot* the result in 2D.

Dimensionality reduction (PCA, t-SNE, UMAP) is the **next deck** — it pairs naturally with clustering.

All of it in sklearn

```
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans, DBSCAN, AgglomerativeClustering

X = StandardScaler().fit_transform(X_raw)      # scale first!

labels = KMeans(n_clusters=4, n_init=10, random_state=509).
    fit_predict(X)
labels = DBSCAN(eps=0.3, min_samples=5).fit_predict(X)        # -1 =
     noise
labels = AgglomerativeClustering(n_clusters=4).fit_predict(X)
```

Same `fit_predict` pattern for every algorithm. Set `random_state` for reproducibility; evaluate with `silhouette_score` / `adjusted_rand_score`.

Easy to get wrong — watch out

- Results depend on **scaling, k, the metric, and the algorithm** — change any one and the clusters change.
- There is **no ground truth** to check against.
- It is easy to **see groups that are not real** — k-means will happily split pure noise into k tidy "clusters".

Always validate (silhouette, stability, a hard look) before trusting a clustering.

Next time: shrinking image files with k-means

Color quantization. An image has up to ∼16 million colors. Run k-means on the *pixels* (in RGB space); repaint each pixel with its centroid color.

Result: only k **colors** — a much smaller palette / file, for a tiny quality cost. We build this end-to-end in the **practical**.

Martiros Saryan, *Mountains* (1923).

Recap

- **Clustering** groups unlabeled data; you choose the distance and (often) k.
- **k-means** — fast, round, equal clusters; Lloyd's alternation, k-means++, pick k by elbow/silhouette. **GMM** generalizes it (soft, elliptical) via EM.
- **DBSCAN / HDBSCAN** — density-based: arbitrary shapes, no k, free outliers. **Hierarchical** — a dendrogram you cut. **k-medoids** — robust, any metric.
- **Evaluation** has no ground truth: silhouette (internal); ARI/AMI (external); track more than one metric, always scale, check stability.

Next: dimensionality reduction (PCA, t-SNE, UMAP) — compress the features, visualize the data, and cluster better in high dimensions.