Clustering

What is Clustering?

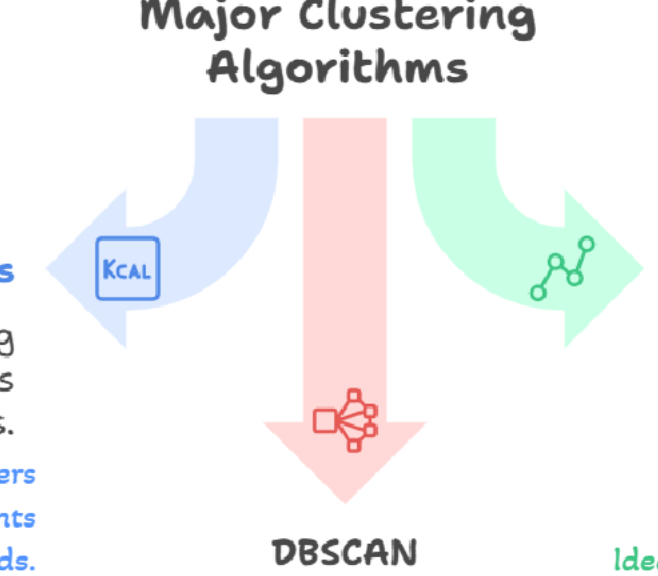
Clustering is the process of grouping a set of data points into clusters such that:

high intra-cluster similarity
Points within the same cluster are similar

low inter-cluster similarity
Points in different clusters are dissimilar

It's like organizing messy data into natural groups without prior labels.

Major Clustering Algorithms



K-means
Suitable for partitioning data into distinct clusters with clear boundaries.
Idea: Partition data into k clusters by minimizing the distance of points from cluster centroids.

DBSCAN
Ideal for identifying clusters of varying shapes and densities, robust against noise.
Idea: Groups together points that are closely packed, and marks outliers as noise.

Hierarchical Clustering
Effective for creating a hierarchy of clusters, useful for exploring data at different levels of granularity.
Idea: Build a tree of clusters(dendrogram).

K-Means algorithm

Mathematical Intuition:

K-Means is an unsupervised learning algorithm used to group data points into K clusters.
The idea is simple:

Each cluster is represented by its centroid (mean point), and each data point belongs to the cluster with the closest centroid.

Step-by-Step Process

1. Choose number of clusters (K) : Decide how many groups you want to partition the data into.

2. Initialize centroids: Pick K random points from the dataset as the initial cluster centroids.

   Better method: K-Means++ initializes more intelligently.

3. Assign points to nearest centroid : For each data point xi, find the closest centroid (using Euclidean distance, usually):

   Assign $x_i \rightarrow C_k$ if $\| x_i - \mu_k \|^2$ is minimal.

4. Update centroids : After assignment, recalculate the centroid of each cluster:

   $\mu_k = (1 / |C_k|) \sum_{x \in C_k} x$

4. Repeat : Go back to step 3 and reassign points → update centroids until convergence.
   Convergence means: a) Centroids stop changing significantly, or
   b) Assignments of points do not change.

Objective Function (minimized): $J = \sum_k \sum_{x \in C_k} \| x - \mu_k \|^2$

Geometrically:
Each cluster is represented by its mean (centroid)
Data space is partitioned into Voronoi regions
Algorithm finds centroids that minimize within-cluster variance

K-Means Clustering – Toy Example

Data points: A(2, 3), B(3, 3), C(6, 7), D(7, 8)

Number of clusters: k = 2

Step 1: Initialize centroids
C1 = (2, 3)
C2 = (6, 7)

Step 2: Assign points to nearest centroid
Distance formula: ‖ (x1, y1) − (x2, y2) ‖ = √((x1−x2)2 + (y1−y2)2)

dist(A, C1) = √((2−2)2 + (3−3)2) = 0
dist(A, C2) = √((2−6)2 + (3−7)2) = √32 ≈ 5.66 → A → C1

dist(B, C1) = √((3−2)2 + (3−3)2) = 1
dist(B, C2) = √((3−6)2 + (3−7)2) = √25 = 5 → B → C1

dist(C, C1) = √((6−2)2 + (7−3)2) = √32 ≈ 5.66
dist(C, C2) = √((6−6)2 + (7−7)2) = 0 → C → C2

dist(D, C1) = √((7−2)2 + (8−3)2) = √50 ≈ 7.07
dist(D, C2) = √((7−6)2 + (8−7)2) = √2 ≈ 1.41 → D → C2

Clusters after step 2:
Cluster 1 = {A, B}
Cluster 2 = {C, D}

Step 3: Update centroids
C1 = mean(A, B) = ((2+3)/2, (3+3)/2) = (2.5, 3)
C2 = mean(C, D) = ((6+7)/2, (7+8)/2) = (6.5, 7.5)

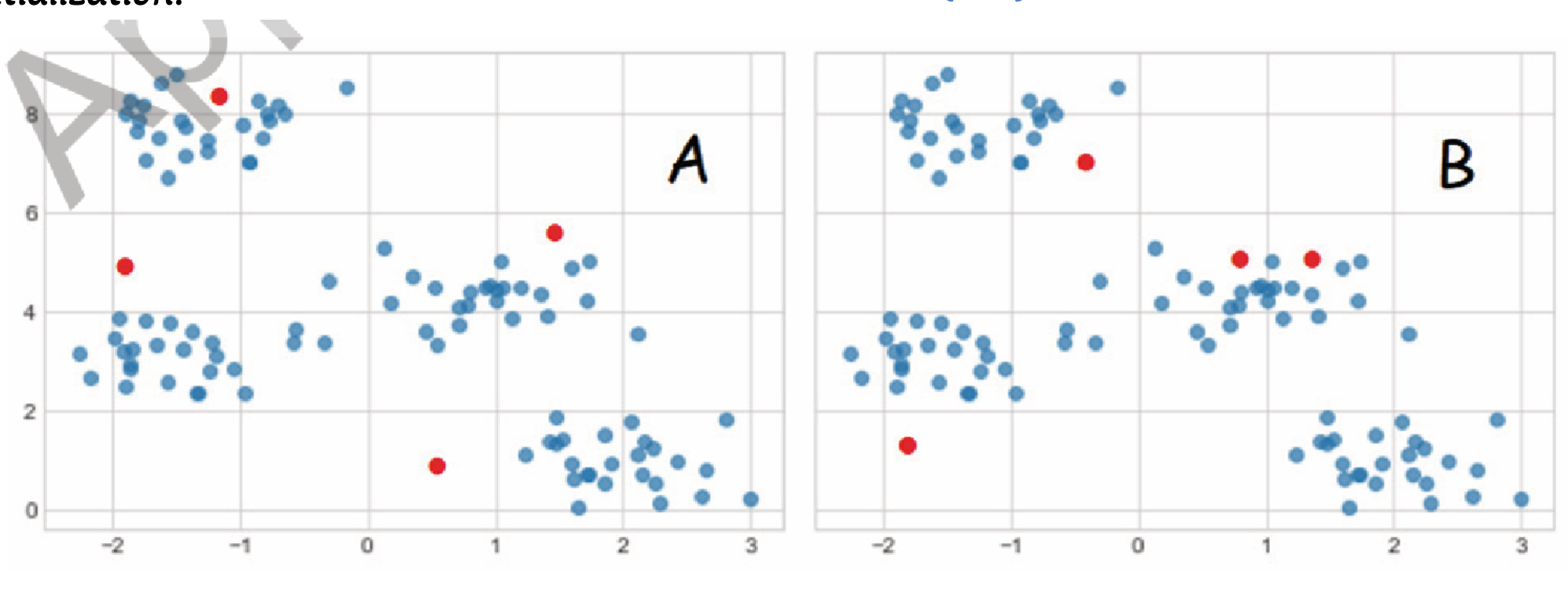
Step 4: Reassign points
All points remain in same clusters → convergence reached.

Final Result
Cluster 1 = {A(2,3), B(3,3)}, centroid = (2.5, 3)
Cluster 2 = {C(6,7), D(7,8)}, centroid = (6.5, 7.5)

Problems due to random initialization:

The effectiveness of Llyod's algorithm mainly depends on the first K randomly selected points.

If the randomly chosen points are such that they are located as shown in image A below, the Llyod's algorithm will generally provide good clustering. But if the initially chosen points are located as shown in image B (Some centroids too close to each other), then the algorithm might result in suboptimal clustering.



K Means ++ initialization technique:

The K Means ++ algorithm is an initialization technique for Lloyd's algorithm. It aims at overcoming the random initialization problem just discussed, by choosing K initial centroids such that they are appropriately far apart from each other. It involves the following steps:
1. Randomly choose the first centroid.
2. Choose the next centroid using the following procedure:
   a. For each data point compute the distance(s) with respect to all the previously chosen centroids and choose the smallest distance to represent it (di = min(distances from centroids) - in case of the first step there will be only one centroid).

   b. Create a probability distribution where each data point's probability is proportional to the normalized square of distance d associated with it, as shown below.

3. Randomly choose a point from the above distribution and use it as the next centroid.
4. Repeat steps 2 and 3 till K centroids are got
5. Use these centroids as the initial centroids in Llyod's algorithm.
Using squared distances increases the separation between points far away from current centroid and points close to it and thus using normalized squared distance as probability, increases the chance of randomly selecting a "faraway" point.

Sample Implementation:

```python
import numpy as np
import pandas as pd
from sklearn.datasets import load_breast_cancer

# Load dataset
data = load_breast_cancer()
df_cancer = pd.DataFrame(data['data'])
df_cancer.columns = ['f' + str(i) for i in range(len(df_cancer.columns))]
df_cancer = df_cancer.assign(labels=data['target'])
print(df_cancer.shape)
df_cancer.head()
```

(569, 31)

	f0	f1	f2	f3	f4	f5	f6	f7	f8	f9	...	f21	f22	f23	f24	f25	f26	f27	f28	f29	labels
0	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419	0.07871	...	17.33	184.60	2019.0	0.1622	0.6656	0.7119	0.2654	0.4601	0.11890	0
1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812	0.05667	...	23.41	158.80	1956.0	0.1238	0.1866	0.2416	0.1860	0.2750	0.08902	0
2	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069	0.05999	...	25.53	152.50	1709.0	0.1444	0.4245	0.4504	0.2430	0.3613	0.08758	0
3	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	0.2597	0.09744	...	26.50	98.87	567.7	0.2098	0.8663	0.6869	0.2575	0.6638	0.17300	0
4	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	0.1809	0.05883	...	16.67	152.20	1575.0	0.1374	0.2050	0.4000	0.1625	0.2364	0.07678	0

5 rows × 31 columns

The dataset contains thirty features/dimensions representing different biological measurements and has labels indicating whether the observations (rows) correspond to a positive case of cancer or not (1 & 0).
The scikit learn library provides two classes for K means clustering:
1. The KMeans class
2. The MiniBatchKMeans class

The KMeans class:

The KMeans class is scikit learn's main tool for K Means Clustering.
Once an instance of this class is instantiated, by providing the required hyperparameter values, it can be fit to the data we want to cluster by using the fit method (ie: The fit method applies the K means algorithm on the data). Note that the data needs to be standardized before fitting the KMeans instance on to it.

```python
[3]:
from sklearn.preprocessing import StandardScaler
import pandas as pd

df_X = df_cancer.iloc[:, :-1]
std_X = StandardScaler().fit_transform(df_X.values)
df_X_std = pd.DataFrame(std_X)
print(df_X_std.head())
```

The "trained"/fitted KMeans instance is now capable of predicting the cluster of any other new data point of the same kind. We use the predict method for doing this. To know the cluster that each point in the original data belongs to, we use the predict method on it, this returns an array containing cluster labels corresponding to each data point. This is demonstrated in the code shown below.

We use the cluster_centers_ attribute of the fitted KMeans instance to access the cluster centers of the clusters detected by the K Means clustering. This returns an array of cluster centers indexed as per their labels.(ie: kmeans_model.cluster_centers_[0] will give centroid of cluster 0 and so on).

In the code shown above, we can see three parameters that were defined while instantiating the KMeans instance named "kmeans". These three parameters /kwargs are the most relevant specifications of the instance. The K Means algorithm requires that we specify the number of clusters expected within the data and it clusters the data based on this information. The n_cluster kwarg lets us specify this. It is set to three in the above example.

```python
from sklearn.cluster import KMeans

# Create KMeans model with 3 clusters
kmeans = KMeans(n_clusters=3, init='k-means++', n_init=5, random_state=42)
kmeans_model = kmeans.fit(std_X)
cluster_per_pt = kmeans_model.predict(std_X)
print(cluster_per_pt[:20])
print(kmeans_model.cluster_centers_.shape)
```

```
[0 0 0 2 0 2 0 2 2 2 1 2 0 1 2 2 1 2 0 1]
(3, 30)
```

The init and n_init parameters specify the random selection procedure to be used for selecting the first set of centroids. The init parameter specifies the method to use to initialize the first set of centroids. We have two choices for this parameter: "k-means++" and "random". Scikit learn by default specifies this as the former. The n_init parameter is used to specify how many times the initiation method specified by the init parameter has to be repeated. In the example above this is specified as five. This means that the K Means algorithm will be run five times for five different kmeans++ initiations and the initiation that yielded the least WCSS/inertia value will be used.

Scikit learn specifies a default value of ten for this parameter.
As can be seen from the output of the code above, the predict method returns an array of cluster labels (0, 1, 2) corresponding to each data point and the cluster_centers_ attributes gives us three thirty dimensional cluster centers.

The MiniBatchKMeans class:

```python
[17]:
def fn_inertia(data_points, n_clusters, batch_size):
    # MINI BATCH K_MEANS CLUSTERING:
    kmeans = MiniBatchKMeans(n_clusters=n_clusters, batch_size=batch_size)
    kmeans = kmeans.fit(data_points)

    # Array containing cluster centers corresponding to each data point:
    cluster_centers = np.array([kmeans.cluster_centers_[idx] for idx in kmeans.labels_])

    # Calculate mean distance of each point to its cluster center:
    inertia = paired_distances(data_points, cluster_centers, metric='euclidean')

    return inertia.mean()
```

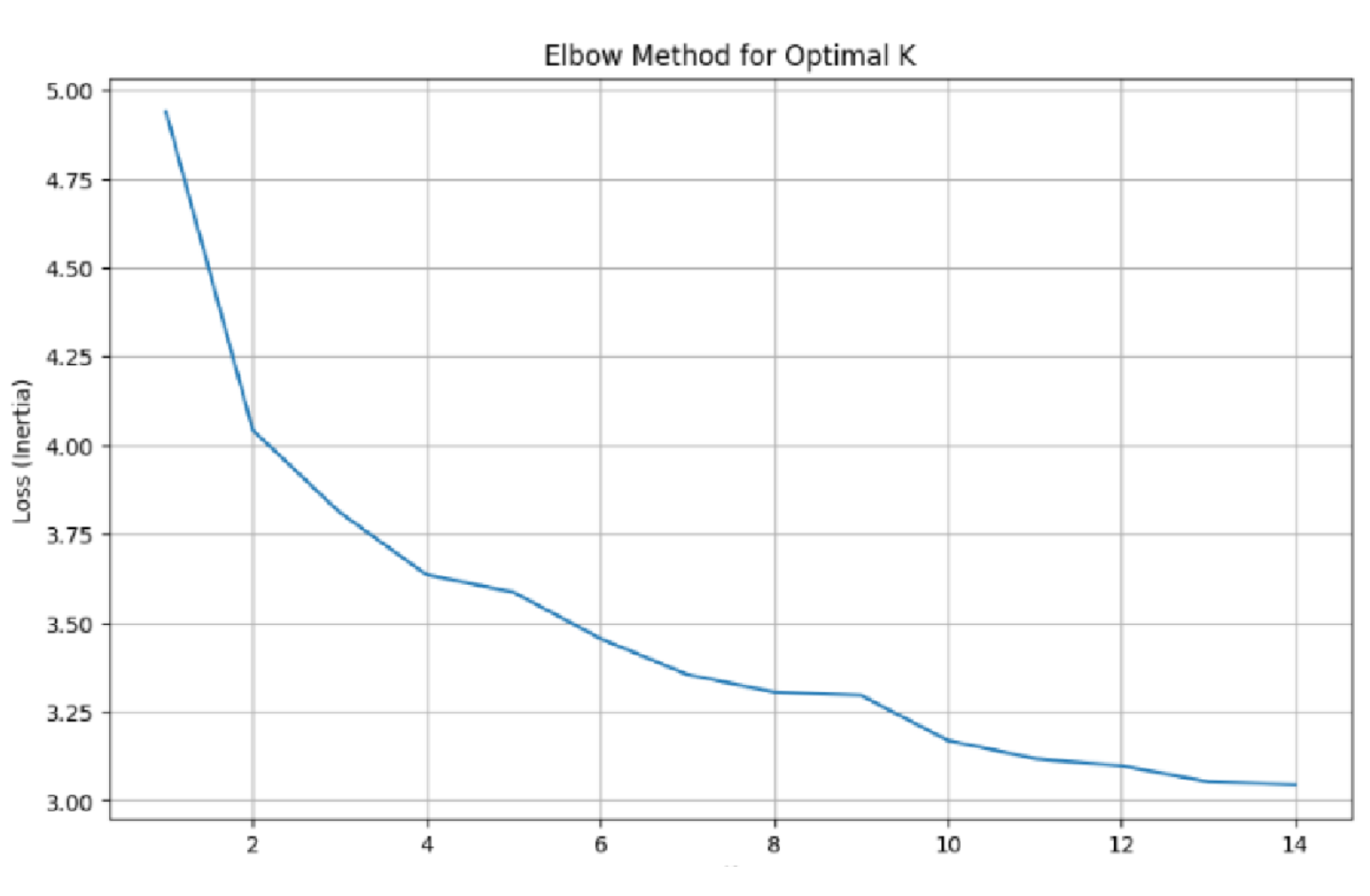
```python
[15]:
def fn_elbow_plot(data_points, n_clusters_choices, batch_size):
    results = []
    # Using tqdm for progress bar (more common than ProgressBar)
    for n_clusters in tqdm(n_clusters_choices):
        avg_var = fn_inertia(data_points, n_clusters, batch_size)
        results.append((n_clusters, avg_var))

    df_sorted_kmeans = pd.DataFrame(results, columns=['K', 'loss'])
    df_sorted_kmeans = df_sorted_kmeans.sort_values(by=['K'])  # Sort by K for proper plotting

    plt.figure(figsize=(10, 6))
    sns.lineplot(x=df_sorted_kmeans['K'].values, y=df_sorted_kmeans['loss'].values)
    plt.xlabel('K')
    plt.ylabel('Loss (Inertia)')
    plt.title('Elbow Method for Optimal K')
    plt.grid(True)
    plt.show()

    return df_sorted_kmeans
```

In the example above we randomly chose n_cluster to be three. In reality one needs to use the "elbow method" to determine the optimal number of clusters. This is where the MiniBatchKmeans class comes in. The MiniBatchKMeans is a variant of the KMeans algorithm which uses a slightly modified version of Lloyd's algorithm, where instead of using the entire dataset at each iteration, to update the centroids, it uses smaller random samples of fixed size.
The mini-batch k-means produces results that are generally quite comparable to that of the standard K-Means algorithm, but not as good. MiniBatchKmeans saves a lot of compute time and makes it possible to cluster large sized datasets without taking up much memory. We choose a range of n_clusters values and create a MiniBatchKmeans instance for each of these n_clusters values. We then fit each of these instances to the data and compute the resulting WCSS/inertia values. The n_clusters value corresponding to the elbow of the n_clusters Vs inertia plot gives us the optimal n_clusters.

The two functions shown above implement the elbow method described.
The function fn_inertia computes the inertia resulting from choosing a particular value for n_clusters.

The fn_elbow_plot uses fn_inertia to compute the inertia values for a range of n_clusters values and plots the "elbow" curve described earlier.

The "elbow" of the plot shown above is at n_clusters = 2 and hence we will use this value to finally cluster the data points using the function fn_kmeans_model shown below.
The function shown above performs KMeans clustering on the given data points using the
102
specified n_clusters and returns the k-means model/instance that was fitted to the data. This can be used for prediction purposes. Also, the function displays the clusters and the number of clusters information. This is demonstrated in the code.



K-Means Clustering: Limitations and Enhancements

Key Limitations of K-Means:

Assumption of spherical clusters: K-Means assumes clusters are spherical and equally sized, which isn't always true in real data.

Sensitivity to outliers: Outliers can significantly distort cluster centroids.

Dependence on initial centroid selection: Different initializations can lead to different results.

Requires specifying k: The number of clusters must be specified in advance.

Struggles with non-linear shapes: Performs poorly on data with complex geometries.

When to Consider Alternative Algorithms:

DBSCAN: Excellent for data with arbitrary shapes and outlier detection

Gaussian Mixture Models: For elliptical clusters of different sizes

Spectral Clustering: For non-convex clusters and complex geometries

Hierarchical Clustering: When you need a hierarchy of clusters

Hierarchical Clustering

Hierarchical clustering is an unsupervised machine learning algorithm that builds a hierarchy of clusters. Unlike K-Means which requires specifying the number of clusters upfront, hierarchical clustering creates a tree-like structure (dendrogram) that shows how clusters are merged or split at different levels.

Two Main Approaches

Agglomerative (Bottom-Up) Clustering

Starts with: Each data point as its own cluster

Process: Iteratively merges the most similar clusters

Ends with: A single cluster containing all data points

Most common approach

Divisive (Top-Down) Clustering

Starts with: All data points in one cluster

Process: Iteratively splits clusters into smaller ones

Ends with: Each data point as its own cluster

Less commonly used

Key Concepts

Distance Metrics

How we measure similarity between points:

Euclidean: Straight-line distance
Manhattan: Sum of absolute differences
Cosine: Angle between vectors

Dendrogram

The dendrogram is the visual representation of hierarchical clustering:
- Y-axis: Shows distance/similarity between clusters
- X-axis: Shows individual data points
- Horizontal lines: Represent cluster mergers
- Height of merger: Indicates distance between clusters

How to read a dendrogram:
- Cut the tree at a certain height to get clusters
- Longer vertical lines indicate better-defined clusters
- The optimal number of clusters is where the longest vertical lines appear

Linkage Methods

How we measure similarity between clusters:

Single Linkage
- Distance between closest points of two clusters
- Can create "chaining" effect (long, elongated clusters)

Complete Linkage
- Distance between farthest points of two clusters
- Creates compact, spherical clusters

Average Linkage
- Average distance between all pairs of points
- Balanced approach

Ward's Method
- Minimizes variance when merging clusters
- Tends to create clusters of similar size

Advantages Over K-Means

- No need to specify number of clusters upfront
- Provides hierarchical relationship between clusters
- Works well with non-globular cluster shapes
- Visual output (dendrogram) is intuitive to interpret

Limitations

- Computationally expensive - O(n3) time complexity
- Sensitive to noise and outliers
- Different linkage methods can give different results
- Once a decision is made to merge clusters, it cannot be undone

When to Choose Hierarchical Clustering

- When you want to explore data structure at multiple levels
- When you don't know the optimal number of clusters
- When hierarchical relationships are important for your analysis
- With smaller datasets (due to computational constraints)

Applications

- Biology: Phylogenetic trees, gene expression analysis
- Social Sciences: Document clustering, market segmentation
- Image Processing: Image segmentation and compression
- Any domain where understanding hierarchical relationships is valuable

Toy Dataset:

Points: A(1, 2), B(2, 1), C(5, 4), D(6, 5), E(7, 8), F(8, 7)

1. INITIAL CLUSTERS (Distance = 0):

   {A}, {B}, {C}, {D}, {E}, {F}

2. CALCULATE ALL PAIRWISE DISTANCES (Euclidean):
   AB: √((1-2)2+(2-1)2) = √2 ≈ 1.41
   AC: √((1-5)2+(2-4)2) = √20 ≈ 4.47
   AD: √((1-6)2+(2-5)2) = √34 ≈ 5.83
   AE: √((1-7)2+(2-8)2) = √72 ≈ 8.49
   AF: √((1-8)2+(2-7)2) = √74 ≈ 8.60
   BC: √((2-5)2+(1-4)2) = √18 ≈ 4.24
   BD: √((2-6)2+(1-5)2) = √32 ≈ 5.66
   BE: √((2-7)2+(1-8)2) = √74 ≈ 8.60
   BF: √((2-8)2+(1-7)2) = √72 ≈ 8.49
   CD: √((5-6)2+(4-5)2) = √2 ≈ 1.41
   CE: √((5-7)2+(4-8)2) = √20 ≈ 4.47
   CF: √((5-8)2+(4-7)2) = √18 ≈ 4.24
   DE: √((6-7)2+(5-8)2) = √10 ≈ 3.16
   DF: √((6-8)2+(5-7)2) = √8 ≈ 2.83
   EF: √((7-8)2+(8-7)2) = √2 ≈ 1.41

3. MERGE CLOSEST CLUSTERS (Distance ≈ 1.41):
   Multiple pairs at this distance: AB, CD, EF
   Let's merge AB first → {A,B}, {C}, {D}, {E}, {F}

4. Distances Between These Clusters (Complete Linkage)

   d({A,B}, {C,D}) = max(AC, AD, BC, BD) = 5.83
   d({A,B}, {E,F}) = max(AE, AF, BE, BF) = 8.60
   d({C,D}, {E,F}) = max(CE, CF, DE, DF) = 4.47

Step 5: Next Merge
Smallest is 4.47 → merge {C,D} and {E,F}.
Now clusters:
{A,B}, {C,D,E,F}

Now merge EF →{A,B}, {C,D}, {E,F}

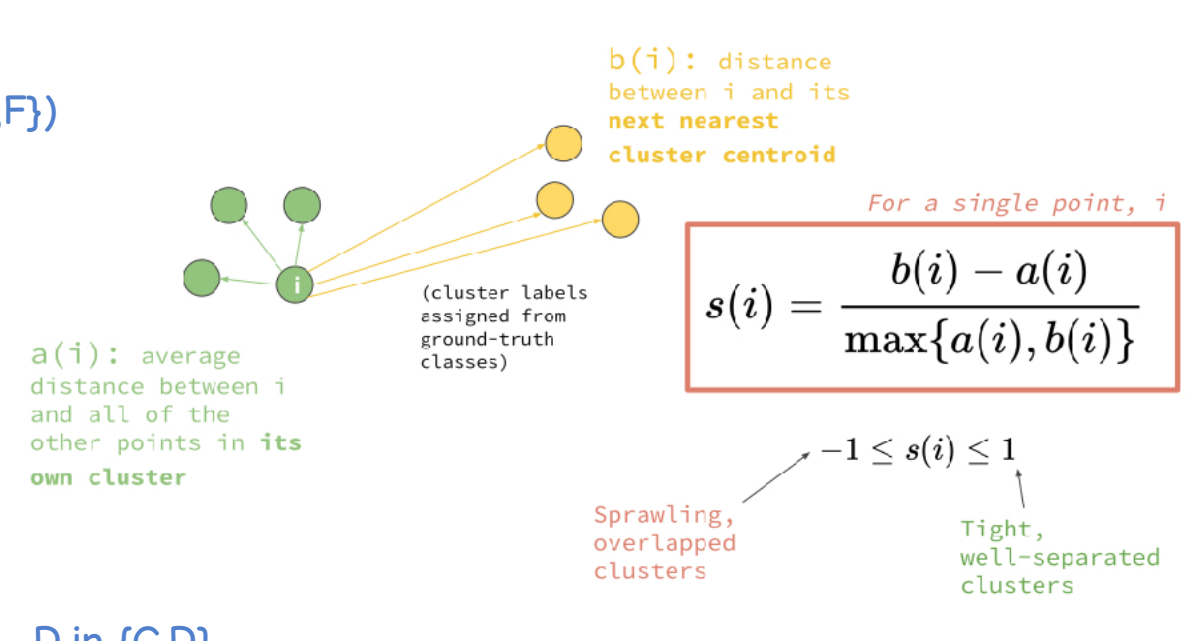
Step 6: Final Merge:

d({A,B}, {C,D,E,F}) = max(all cross distances) ≈ 5.83
At 5.83 everything merges into one cluster.

Silhouette Scoring

Silhouette Score Calculation (3 clusters = {A,B}, {C,D}, {E,F})

Formula
a(i) = average distance from i to its own cluster
b(i) = minimum (over other clusters) of average distance from i to that cluster
s(i) = (b(i) − a(i)) / max(a(i), b(i))



Step 1: Pairwise distances (Euclidean)
AB=CD=EF=1.414
AC=4.472, AD=5.831, BC=4.243, BD=5.657
CE=4.472, CF=4.243, DE=3.162, DF=2.828
AE=8.485, AF=8.602, BE=8.602, BF=8.485

Step 2: Point calculations

A in {A,B}
a(A)=1.414
b(A)=min(5.152, 8.544)=5.152
s(A)=(5.152−1.414)/5.152=0.725

B in {A,B}
a(B)=1.414
b(B)=4.950
s(B)=(4.950−1.414)/4.950=0.714

C in {C,D}
a(C)=1.414
b(C)=4.358
s(C)=(4.358−1.414)/4.358=0.675

D in {C,D}
a(D)=1.414
b(D)=2.995
s(D)=(2.995−1.414)/2.995=0.528

E in {E,F}
a(E)=1.414
b(E)=3.817
s(E)=(3.817−1.414)/3.817=0.630

F in {E,F}
a(F)=1.414
b(F)=3.536
s(F)=(3.536−1.414)/3.536=0.600

Step 3: Overall average silhouette

$\bar{s}$ = (0.725 + 0.714 + 0.675 + 0.528 + 0.630 + 0.600) / 6
$\bar{s}$ ≈ 0.645

☑ Result: Average silhouette ≈ 0.645.
This confirms 3 clusters ({A,B}, {C,D}, {E,F}) give the best structure.

☑ Result: Average silhouette ≈ 0.645.
This confirms 3 clusters ({A,B}, {C,D}, {E,F}) give the best structure.

DBSCAN
(Density-Based Spatial Clustering of Applications with Noise)

Unlike K-Means, DBSCAN does not assume spherical clusters or require the number of clusters beforehand.
It groups points based on density: areas where many points are close together form clusters, while sparse areas are treated as noise (outliers).

Two Main Approaches

Parameters

ε (epsilon): neighborhood radius

minPts: minimum number of points to form a dense region

Definitions

Core Point: $|N_\varepsilon(p)| \geq minPts$
Border Point: $|N_\varepsilon(p)| < minPts$ but $\exists\, q \in N_\varepsilon(p)$ that is a Core Point
Noise Point: not Core, not Border

Where
$N_\varepsilon(p) = \{ q \in D \mid dist(p, q) \leq \varepsilon \}$

Algorithm:

For each unvisited point p:
a. If p is a Core Point → create new cluster C
b. Add all density-reachable points from p into C
c. If p is Border → assign to nearby cluster
d. Else → mark as Noise

Repeat until all points are visited

Sample Example:

Points: A(0,0), B(0.1,0), C(0.2,0), D(5,5), E(5.1,5), F(10,0)
Choose ε = 0.3, minPts = 2

Nε(A) = {A, B, C} → |Nε(A)| = 3 ≥ 2 → A = Core
Nε(B) = {A, B, C} → |Nε(B)| = 3 ≥ 2 → B = Core
Nε(C) = {A, B, C} → |Nε(C)| = 3 ≥ 2 → C = Core
→ Cluster1 = {A, B, C}

Nε(D) = {D, E} → |Nε(D)| = 2 ≥ 2 → D = Core
Nε(E) = {D, E} → |Nε(E)| = 2 ≥ 2 → E = Core
→ Cluster2 = {D, E}

Nε(F) = {F} → |Nε(F)| = 1 < 2 → Noise

Result
Cluster1 = {A,B,C}
Cluster2 = {D,E}
Noise = {F}

Limitation:

Choice of Parameters (ε, minPts)
- Sensitive to the selection of ε (epsilon).
- Too small ε → many points become Noise.
- Too large ε → merges distinct clusters.

Varying Density Problem
- Struggles if clusters have different densities.
- One ε may work well for dense regions but fail for sparse ones.

High Dimensional Data
- Distance measures (like Euclidean) lose meaning in high-dimensions (curse of dimensionality).
- DBSCAN may group incorrectly or label many as noise.

Scalability
- Complexity ≈ O(n log n) with spatial indexing, but O(n2) without it.
- Not efficient for very large datasets unless optimized.

Limitation:

☑ When DBSCAN works best
- Clusters are well-separated by low-density regions.
- Data has similar density levels.
- Need to detect noise/outliers naturally.

Clustering Algorithm Comparison

Feature	DBSCAN	K-Means	Hierarchical
Cluster Number	No	Yes	Optional
Cluster Shape	Arbitrary	Spherical	Any
Outlier Handling	Yes	No	No
Density Variance	Struggles	Not handled	Partially
Parameters	ε, minPts	k	Linkage, Distance
Scalability	O(n log n)	O(n × k × iter)	O(n2)
Interpretability	Moderate	High	Moderate
Best For	Arbitrary, Noise	Well-separated, Spherical	Small/Medium, Hierarchy