What is Clustering?

Clustering is an **unsupervised machine learning** technique used to group similar data points together based on feature similarity.

Examples:

1. 🎯 Customer Segmentation (Marketing)

Use Case: A retail company wants to group customers based on purchasing behavior.
Features Used: Age, income, purchase frequency, product categories bought.
Purpose: To create targeted marketing campaigns for each customer segment.

---

2. 🧬 Gene Expression Analysis (Biology/Genetics)

Use Case: Cluster genes with similar expression patterns across various experiments.
Features Used: Expression levels in different conditions.
Purpose: Identify functionally related genes or potential biomarkers.

---

3. 🛒 Recommendation Systems

Use Case: Group users who exhibit similar browsing or buying behavior.
Features Used: Movie ratings, clicks, viewed products, etc.
Purpose: Recommend products or content preferred by similar users.

---

4. 🛰️ Image Compression (Computer Vision)

Use Case: Reduce the number of unique colors in an image using clustering (e.g., k-means).
Features Used: RGB color values of each pixel.
Purpose: Compress image size while preserving visual appearance.

---

5. 🌆 Urban Planning / Traffic Management

Use Case: Cluster regions based on traffic flow, population density, or utility usage.
Features Used: GPS coordinates, time, usage stats.
Purpose: Optimize transportation routes or resource allocation.

K-Means Clustering

✅ Key Concepts:

1. **Centroids**: Randomly chosen K data points as the initial "means" of clusters.
2. **Inertia**: The sum of squared distances between data points and their assigned centroids.
3. **Iterations**: Updates centroids repeatedly until convergence.

🛠 Steps:

1. Choose the number of clusters `K`.
2. Initialize `K` centroids randomly.
3. Assign each point to the closest centroid.
4. Recalculate the centroids as the mean of all points in the cluster.
5. Repeat steps 3–4 until centroids don't change or max iterations reached.

⚙️ Pros:

- Fast and scalable.
- Works well when clusters are spherical.

⚠️ Cons:

- Sensitive to initial centroids.
- Requires specifying `K` in advance.
- Doesn't work well with non-convex shapes.

◇ Hierarchical Clustering

✅ Key Concepts:

1. **Agglomerative (bottom-up)**: Each point starts as its own cluster; clusters are merged step by step.
2. **Divisive (top-down)**: All data starts in one cluster, and is recursively split.
3. **Dendrogram**: A tree-like diagram that shows the hierarchy of cluster merging.

🛠 Steps (Agglomerative):

1. Compute distance (Euclidean or others) between every pair of points.
2. Merge the two closest clusters.
3. Repeat until only one cluster remains.
4. Use a dendrogram to decide the number of clusters.

⚙️ Pros:

- No need to pre-specify `K`.
- Good for nested data structures.

⚠️ Cons:

- Computationally intensive (not good for large datasets).
- Sensitive to noise and outliers.

🔄 Comparison

Feature	K-Means	Hierarchical Clustering
Cluster Shape	Spherical	Arbitrary
Scalability	Very scalable	Less scalable
Needs K?	Yes	No (can be decided via dendrogram)
Algorithm Type	Partitional	Hierarchical
Output	Flat clusters	Dendrogram/tree
Sensitive to Outliers?	Yes	Yes

Distance Measures and Data Preparation for Clustering

In clustering, **distance measures** and **data preparation** are essential components to determine the quality and relevance of the formed clusters.

1. Distance Measures in Clustering

The core idea behind most clustering algorithms (like K-Means, Hierarchical Clustering, DBSCAN) is to group similar points together based on their distance in the feature space.

Here are some common distance measures:

a) Euclidean Distance (Most Common)

The Euclidean distance is the straight-line distance between two points in the feature space. It is the most commonly used distance metric.

- $$d(p_1, p_2) = \sqrt{\sum_{i=1}^{n} (p_{1i} - p_{2i})^2}$$

- **When to Use**: Euclidean distance works well when the data points are normally distributed, and there is a meaningful scale across all features.

b) Manhattan Distance (L1 Norm)

Manhattan distance is the sum of the absolute differences between two points, and it's a good alternative when the data has many outliers or sparse data.

$$d(p_1, p_2) = \sum_{i=1}^{n} |p_{1i} - p_{2i}|$$

- **When to Use**: It is useful in high-dimensional spaces, especially when features are not normally distributed.

A **norm** is a measure of the **magnitude (length or size)** of a vector. In machine learning and linear algebra, it tells us **how far a point is from the origin** in a multi-dimensional space.

◇ L1 Norm (Manhattan Norm)

📌 Definition:

The **L1 norm** is the **sum of the absolute values** of the vector elements.

📘 Formula:

For a vector **x = [x_1, x_2, ..., x_n]**:

When you want **sparsity** (many zero coefficients).

L2 Norm (Euclidean Norm)

◇ L2 Norm (Euclidean Norm)

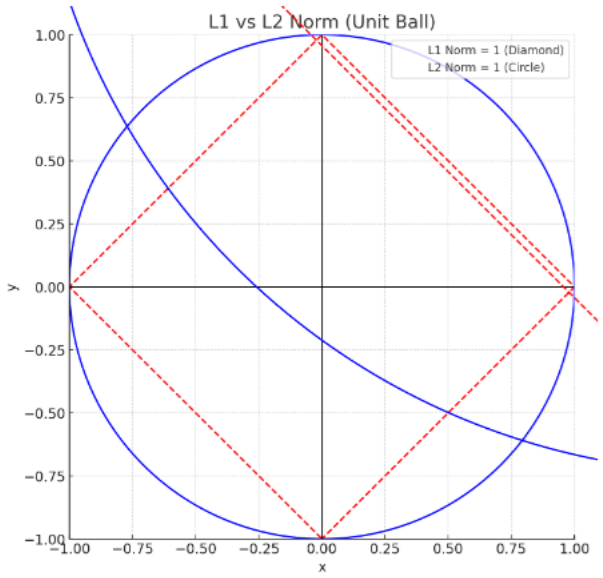
📌 Definition:

The **L2 norm** is the **square root of the sum of the squares** of the vector elements.

📘 Formula:

For a vector **x = [x_1, x_2, ..., x_n]**:

L2 norm = sqrt(x_1^2 + x_2^2 + ... + x_n^2)



Here's a visualization of **L1 vs L2 Norm**:

- 🔷 **Blue Circle**: Shows all points with **L2 norm = 1** — Euclidean distance (straight-line). It's round because all directions are treated equally.
- 🔺 **Red Diamond**: Shows all points with **L1 norm = 1** — Manhattan distance (along axes). It's shaped like a diamond because it measures absolute distances along each axis.

💡 Interpretation:

- **L1 norm** prefers solutions where many values are **zero** (sparse). The diamond edges promote this behavior.
- **L2 norm** prefers smaller but **non-zero** values (smooth), distributing the weights more evenly.

c) Minkowski Distance

Minkowski distance is a generalization of both Euclidean and Manhattan distances. It is defined as:

$$d(p_1, p_2) = \left(\sum_{i=1}^{n} |p_{1i} - p_{2i}|^p \right)^{1/p}$$

- **When to Use**: The value of pdetermines which specific distance is used:
  - p=1 gives Manhattan distance.

- p=2 gives Euclidean distance.

d) Cosine Similarity

Cosine similarity measures the cosine of the angle between two vectors. It's especially useful for text data or high-dimensional sparse data, where we are concerned about the orientation rather than the magnitude.

$$\text{Cosine similarity} = \frac{A \cdot B}{\|A\|\|B\|}$$

- **When to Use**: It's effective in text mining (document clustering) or when dealing with high-dimensional sparse data.

e) Jaccard Similarity

The Jaccard similarity measures the similarity between two sets by dividing the size of their intersection by the size of their union.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

- **When to Use**: It is often used for categorical data or binary data, especially when considering the presence or absence of certain attributes.

---

2. Data Preparation: Scaling & Weighting

In clustering, the scale and importance of features can significantly affect the results. Proper data preparation helps in improving clustering performance and ensuring meaningful clusters.

a) Feature Scaling

Feature scaling is a technique used to standardize the range of independent variables or features of the data. Features are scaled so that they all have equal importance when computing distances.

- **Why Scaling is Important**:
  - If features are on different scales (e.g., age from 0 to 100, and income from 1000 to 100,000), the clustering algorithm might give undue importance to features with higher values, causing biased clustering.
  - Algorithms like **K-Means**, **K-NN**, and **Hierarchical Clustering** depend on distances, so scaling ensures fair contribution from all features.

Common Scaling Methods:

- **Standardization (Z-score normalization)**: Subtract the mean and divide by the standard deviation for each feature.

$$x' = \frac{x - \mu}{\sigma}$$

  - **When to Use**: For algorithms sensitive to the magnitude of the data, like K-Means, K-NN, and PCA.
- **Min-Max Scaling**: Rescales the feature to a fixed range, usually [0, 1].

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

  - **When to Use**: When the features have different units or need to be within a specific range.
- **Robust Scaling**: Uses the median and interquartile range to scale the data, which makes it robust to outliers.

$$x' = \frac{x - \text{median}}{\text{IQR}}$$

  - **When to Use**: When the data contains outliers.

- b) Weighting the Features The IQR Inter Quartile Range is the difference between the third quartile (Q3) and the first quartile (Q1). It's essentially the range within which the middle half of the data falls.
- **Formula:** IQR = Q3 - Q1.

In some cases, you might want to give more importance (weight) to certain features over others. This can be done during the clustering process by adjusting the feature values or incorporating a weighting factor.

- **Why Weighting is Important**: Some features might be more important than others in defining the clusters. For example, in customer segmentation, the customer's purchasing history might be more important than their age.

Methods to Weight Features:

- **Manually Adjust Weights**: Multiply the values of the important features by a weight (e.g., 2) before applying the clustering algorithm.
- **Automated Weighting**: Some advanced clustering algorithms, like **Gaussian Mixture Models** or **DBSCAN DBSCAN (Density-Based Spatial Clustering of Applications with Noise), c**an automatically account for feature weighting based on their statistical properties.

c) Handling Missing Values

Missing data can distort the clustering process. Techniques to handle missing data include:

- **Imputation**: Replace missing values with the mean, median, or mode of the column, or use more advanced techniques such as k-NN imputation.
- **Omission**: In some cases, you can omit rows with missing values, but this may cause loss of data.

Evaluation and Profiling of Clusters:

Cluster evaluation and profiling are critical steps in clustering analysis.

After performing clustering, we want to assess the quality of the clusters and understand their characteristics.

Evaluation refers to how well the clustering algorithm performed, while profiling helps us understand the meaning and characteristics of the clusters.

1. Cluster Evaluation:

Cluster evaluation involves assessing the quality of the clusters formed by the clustering algorithm. It can be divided into two categories:

A. Internal Evaluation Metrics:

These metrics assess the quality of the clustering **without reference to external labels** (i.e., unsupervised evaluation). Common internal metrics are:

1. Silhouette Score:

- **Definition**: Measures how similar a point is to its own cluster compared to other clusters.

- **Range**: The score ranges from **-1 to +1**:
  - A value close to **+1** means that the points are well clustered.
  - A value close to **0** means that the points are on or very close to the decision boundary between clusters.
  - A value close to **-1** means that the points may have been incorrectly assigned to the wrong cluster.
- **Formula**: For a data point xi, the silhouette score is calculated as:

$$S(x_i) = \frac{b(x_i) - a(x_i)}{max(a(x_i), b(x_i))}$$

Where:

- $a(x_i)$: Average distance from x_i to all other points in the same cluster (intra-cluster distance).
- $b(x_i)$: Average distance from x_i to the points in the nearest cluster (nearest-cluster distance).

Example Code:

```
from sklearn.metrics import silhouette_score
sil_score = silhouette_score(X, y_kmeans)  # where X is your data and
y_kmeans are your cluster labels
print("Silhouette Score:", sil_score)
```

2. Davies-Bouldin Index:

- **Definition**: Measures the average similarity ratio of each cluster with the one that is most similar to it. A lower Davies-Bouldin index indicates better clustering.
- **Formula**: The Davies-Bouldin index for a cluster set is calculated as:

$$DB = \frac{1}{k}\sum_{i=1}^{k} \max_{j \neq i} \left(\frac{s_i + s_j}{d(c_i, c_j)} \right)$$

Where:

- s_i is the average distance between the points in cluster i and its centroid.
- $d(c_i, c_j)$ is the distance between the centroids of clusters i and j.

Example Code:

```
from sklearn.metrics import davies_bouldin_score
db_score = davies_bouldin_score(X, y_kmeans)
print("Davies-Bouldin Index:", db_score)
```

3. Inertia (Within-Cluster Sum of Squares):

- **Definition**: Measures how internally coherent the clusters are by calculating the sum of squared distances between each point and its assigned cluster center**. Lower inertia indicates that the points are closer to their centroids.**
- **Formula**:

$$Inertia = \sum_{i=1}^{n} \sum_{j=1}^{k} \|x_i - c_j\|^2$$

Where:

- x_i is a data point.
- c_j is the centroid of cluster j.

Example Code:

```
inertia = kmeans.inertia_  # From KMeans model fitted earlier
print("Inertia:", inertia)
```

B. External Evaluation Metrics (When true labels are available):

When we have true labels, external metrics can be used to assess the clustering algorithm's performance by comparing the predicted clusters with the ground truth.

1. Adjusted Rand Index (ARI):

- **Definition**: Measures the similarity between the predicted clusters and the true clusters, adjusted for the chance of random assignment.
- **Range**: The ARI ranges from **-1 to 1**:
  - **1**: Perfect match between predicted and true clusters.
  - **0**: No agreement, with random cluster assignments.
  - **Negative values**: Indicates worse-than-random assignment.

Example Code:

```
from sklearn.metrics import adjusted_rand_score
ari_score = adjusted_rand_score(true_labels, y_kmeans)  # true_labels are
the true cluster labels
print("Adjusted Rand Index:", ari_score)
```

2. Normalized Mutual Information (NMI):

- **Definition**: Measures the amount of information shared between the predicted clusters and the true clusters. The higher the value, the better the clustering.
- **Range**: NMI ranges from 0 to 1:
  - **1**: Perfect correlation.
  - **0**: No correlation.

Example Code:

```
from sklearn.metrics import normalized_mutual_info_score
nmi_score = normalized_mutual_info_score(true_labels, y_kmeans)
print("Normalized Mutual Information:", nmi_score)
```

---

2. Cluster Profiling:

Cluster profiling involves understanding and interpreting the characteristics of the clusters. This helps in understanding the structure and patterns in the data. Profiling is typically done by examining:

- **Cluster Centroids**: The central values or representative points of each cluster.
- **Feature Distribution**: Statistical distribution of features within each cluster.
- **Cluster Size**: Number of data points in each cluster.

Example: Profiling Using Centroids

After performing clustering, the centroids can be used to describe the clusters' key characteristics. These centroids represent the "average" data point in each cluster.

Example Code:

```
# Get the cluster centroids
centroids = kmeans.cluster_centers_
print("Cluster Centroids:")
print(centroids)
```

A. Feature Distribution:

You can also profile clusters by visualizing how individual features are distributed within each cluster.

Example Code (using boxplots or histograms):

```
import seaborn as sns

# Assuming you have the cluster labels y_kmeans and your features X
X['Cluster'] = y_kmeans
sns.boxplot(x='Cluster', y='Feature_1', data=X)  # For one feature, repeat
for others
plt.title("Feature Distribution by Cluster")
plt.show()
```

This boxplot will show how the distribution of `Feature_1` differs across the clusters, providing insights into the unique characteristics of each cluster.

B. Cluster Size:

Cluster size refers to the number of data points in each cluster. Understanding the cluster size helps you identify dominant clusters and small, sparse clusters.

Example Code:

```
# Calculate the size of each cluster
import pandas as pd
```

```
cluster_sizes = pd.Series(y_kmeans).value_counts()
print("Cluster Sizes:")
print(cluster_sizes)
```

This will show how many data points belong to each cluster.

---

3. Visual Evaluation of Clusters:

Visualizing clusters is one of the most effective ways to evaluate and profile them. You can plot the data points, color them by cluster, and overlay the cluster centroids. This helps to visually inspect whether the clusters are well-separated and coherent.

Example Code (Using Matplotlib):

```
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, cmap='viridis')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
s=300, c='red', marker='X')
plt.title("Clustering with Centroids")
plt.show()
```

☑ 1. Hierarchical Clustering

📌 Definition:

A method to group similar objects into clusters based on a hierarchy (tree structure). It **doesn't require specifying the number of clusters in advance**.

📌 Types:

- **Agglomerative (Bottom-Up)**: Each point starts as its own cluster. Pairs of clusters are merged step by step.
- **Divisive (Top-Down)**: Start with one large cluster and divide it into smaller clusters.

📌 Real-Life Example:

Imagine you're organizing books in a library:

- First, you divide them by **subject** (e.g., Science, Arts).
- Then, within Science → you divide by **Physics, Chemistry**, etc.
- Within Physics → you divide by **Quantum, Classical**, etc. That's **hierarchical clustering**!

📌 Python Example:

```
from sklearn.datasets import make_blobs
from sklearn.cluster import AgglomerativeClustering
import matplotlib.pyplot as plt
import seaborn as sns
```

```
# Create sample data
X, y = make_blobs(n_samples=50, centers=3, random_state=42)

# Apply Hierarchical clustering
model = AgglomerativeClustering(n_clusters=3)
labels = model.fit_predict(X)

# Visualize
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='rainbow')
plt.title("Hierarchical Clustering")
plt.show()
```

☑ 2. Clustering Case Study

🎯 Problem:

A **supermarket** wants to group customers into different segments for targeted promotions.

📌 Features collected:

- Amount Spent per Month
- Number of visits per week
- Membership duration

📌 What can clusters show?

- Cluster 1: High spenders
- Cluster 2: Budget customers
- Cluster 3: Rare shoppers

Clustering helps target them differently with **personalized offers**.

☑ 3. Principal Component Analysis (PCA)

📌 Definition:

PCA is a technique used to **reduce the number of features (dimensions)** in your dataset while preserving as much information as possible.

🧠 Think of it like:

- Compressing an image to fewer pixels while keeping the most detail.
- PCA finds new axes (principal components) that explain most of the data's variance.

📌 Real-Life Example:

Imagine compressing a dataset of **student scores** in 10 subjects to 2 components like:

- PC1: Academic performance
- PC2: Interest in science

📌 Python Example:

```
from sklearn.decomposition import PCA
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt

iris = load_iris()
X = iris.data

# Reduce to 2 dimensions
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

# Visualize
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=iris.target, cmap='Set1')
plt.title("PCA: Iris Dataset")
plt.xlabel("PC1")
plt.ylabel("PC2")
plt.show()
```

☑ 4. Kernel PCA

📌 Problem PCA can’t solve:

What if your data is not **linearly separable**?

📌 Solution:

Use **Kernel PCA**, which applies PCA in a higher dimension using kernel functions (like RBF -

Radial Basis Function

).

📌 Real-Life Example:

If your data forms a **circle or spiral**, kernel PCA can still reduce it meaningfully.

```
from sklearn.decomposition import KernelPCA
from sklearn.datasets import make_circles

X, y = make_circles(n_samples=100, factor=0.3, noise=0.05)
kpca = KernelPCA(kernel="rbf", gamma=15)
X_kpca = kpca.fit_transform(X)

plt.scatter(X_kpca[:, 0], X_kpca[:, 1], c=y)
plt.title("Kernel PCA with RBF")
plt.show()
```

Impact of `gamma`:

`gamma` Value	Behavior	Effect on Decision Boundary
0.001	Very low	Very smooth and general decision boundary
0.1	Moderate	Balanced complexity and generalization
10	High	Overfits, sensitive to noise
100+	Very high	Extreme overfitting, almost memorizes data

☑ 5. Random Projections

📌 Definition:

A fast and simple method to reduce dimensions by projecting data to a lower-dimensional **random subspace**.

📌 Real-Life Example:

You have a huge dataset with 10,000 features → Random Projections can reduce it to 100 features **quickly** without much error.

Types of Random Projection in `sklearn`:

Projection Type	Description
`GaussianRandomProjection`	Uses a normal distribution (mean = 0, std = 1/√n) to generate projection matrix
`SparseRandomProjection`	Uses a sparse matrix (many zeros) for faster computation, works well with sparse data

📌 Python Example:

```
from sklearn.random_projection import GaussianRandomProjection
from sklearn.datasets import load_digits

X, y = load_digits(return_X_y=True)
rp = GaussianRandomProjection(n_components=10, random_state=42)
X_rp = rp.fit_transform(X)

print("Original shape:", X.shape)
print("Reduced shape:", X_rp.shape)
```