Lab Sheet-9
Unsupervised Learning: Clustering Algorithms
Machine Learning
BITS F464
I Semester 2024-25

INTRODUCTION:
In this labsheet, we are going to talk about some unsupervised clustering techniques, try to observe some of their evaluation metrics and ultimately decide which one is the best.

ABOUT THE DATASET:
The **Bank Marketing dataset** is widely used for predicting whether a customer will subscribe to a term deposit after a marketing campaign. It contains a mix of **features** (socioeconomic data, previous campaign information, and contact details) and a **target variable** that represents whether a client subscribed to the term deposit.

1. Features (Independent Variables)
a. Client-related Features:

- **age**: *(Numeric)*
  The age of the customer. Age can influence financial decisions, and certain age groups may be more or less likely to subscribe to a term deposit.
- **job**: *(Categorical)*
  The type of job the customer holds (e.g., 'admin.', 'blue-collar', 'technician', 'management', etc.). Different occupations may reflect varying financial behaviors or levels of income, affecting their likelihood to subscribe.
- **marital**: *(Categorical)*
  The marital status of the customer (e.g., 'married', 'single', 'divorced'). This might impact financial stability or decision-making.
- **education**: *(Categorical)*
  The highest level of education achieved by the customer (e.g., 'primary', 'secondary', 'tertiary'). Education level is often correlated with income level and financial literacy.
- **default**: *(Binary: 'yes', 'no')*
  Indicates whether the customer has a credit in default. Customers with defaulted credit may be less likely to invest in a term deposit.
- **balance**: *(Numeric)*
  The account balance of the customer in euros. Higher balances might indicate customers who are more financially capable of subscribing to term deposits.
- **housing**: *(Binary: 'yes', 'no')*
  Whether the customer has a housing loan. Customers with housing loans may be less likely to invest in other products due to existing financial obligations.

- **loan**: *(Binary: 'yes', 'no')*
  Whether the customer has a personal loan. Like housing loans, personal loans might affect a customer's ability or interest in subscribing to a term deposit.

b. Contact-related Features:

These are details about how the customer was contacted during the marketing campaign:

- **contact**: *(Categorical)*
  Type of contact communication used for marketing (e.g., 'cellular', 'telephone'). The method of communication can influence the effectiveness of the marketing campaign.
- **day**: *(Numeric)*
  Last contact day of the month. This indicates when the customer was contacted during the campaign.
- **month**: *(Categorical)*
  Last contact month of the year (e.g., 'jan', 'feb', 'mar'). The timing of the campaign, such as seasonality, can influence customer behavior.
- **duration**: *(Numeric)*
  Duration of the last contact in seconds. Longer durations can indicate more successful conversations, and this feature has a strong correlation with the target variable. However, it is often excluded in modeling because it is a post-contact outcome (and not known before the campaign).

c. Previous Campaign-related Features:

These reflect historical campaign performance with the customer:

- **campaign**: *(Numeric)*
  Number of contacts performed during the current campaign for this customer. This shows how persistent the marketing efforts were.
- **pdays**: *(Numeric)*
  Number of days since the customer was last contacted from a previous campaign. A large number might indicate that the customer was contacted long ago or not at all (with a value of 999).
- **previous**: *(Numeric)*
  Number of contacts performed before this campaign. If a customer was frequently contacted in the past, they might be more (or less) likely to subscribe depending on their response history.
- **poutcome**: *(Categorical)*
  The outcome of the previous marketing campaign (e.g., 'success', 'failure', 'nonexistent'). Previous success or failure can be a strong predictor of the current outcome.

2. Target (Dependent Variable)

- **y**: *(Binary: 'yes', 'no')*
  This is the target variable, which indicates whether the customer subscribed to a term deposit:
    - **yes**: The customer subscribed to a term deposit.
    - **no**: The customer did not subscribe.

OBTAINING AND PRE-PROCESSING THE DATASET: (Same boring process but very very very important)(can you think of any other process that can make the data better for training?)

```
!pip install ucimlrepo

from ucimlrepo import fetch_ucirepo

import pandas as pd

from sklearn.preprocessing import StandardScaler, LabelEncoder

from sklearn.cluster import KMeans

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.decomposition import PCA



bank_marketing = fetch_ucirepo(id=222)

X = bank_marketing.data.features

y = bank_marketing.data.targets

# Step 2: Load the Bank Marketing Dataset

data = X

# Step 3: Preprocessing the data

# Encode categorical variables
```

```
label_encoder = LabelEncoder()

# Apply Label Encoding to all categorical columns

for col in ['job', 'marital', 'education', 'default', 'housing', 'loan',
'contact', 'month', 'poutcome']:

    data[col] = label_encoder.fit_transform(data[col])



# Step 4: Scale the features

# Standardize the dataset for better performance with K-Means

scaler = StandardScaler()

X_scaled = scaler.fit_transform(data)
```

Before we run into using the algorithms, we will see the evaluation metrics.

EVALUATION METRICS:

SSE (Sum of Squared Errors):

$$SSE = \sum_{i=1}^{n} (\text{distance}(x_i, \text{cluster center}))^2$$

- **Interpretation**: Lower SSE values indicate better clustering. It measures how compact the clusters are.

SILHOUTTE SCORE:

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

Where:

- a(i) is the **cohesion**, which is the average distance between point iii and all other points in the same cluster.
- b(i) is the **separation**, which is the average distance between point iii and all points in the nearest (or next closest) cluster.

DUNNS INDEX:

$$\text{Dunn's Index} = \frac{\text{min distance between clusters}}{\text{max intracluster distance}}$$

Interpretation: A higher Dunn's Index indicates well-separated clusters. It looks at the ratio of the smallest distance between clusters to the largest intra-cluster distance.

```
def dunn_index(X, labels):

    """

    Calculates the Dunn index for a given clustering result.



    Parameters:

        X (array-like): The data points.

        labels (array-like): Cluster labels for each data point.



    Returns:

        float: The Dunn index.

    """

    unique_clusters = np.unique(labels)

    num_clusters = len(unique_clusters)



    max_intracluster_dist = 0.0
```

```
    min_intercluster_dist = np.inf

    # Calculate maximum intracluster distance
    for i in unique_clusters:
        cluster_i_points = X[labels == i]

        # Handle clusters with single or no data points
        if len(cluster_i_points) <= 1:
            intracluster_dist = 0  # or any other appropriate value
        else:
            intracluster_dist = np.max(pdist(cluster_i_points))

        if intracluster_dist > max_intracluster_dist:
            max_intracluster_dist = intracluster_dist

    # Calculate minimum intercluster distance
    for i in range(num_clusters):
        for j in range(i + 1, num_clusters):
            cluster_i_points = X[labels == unique_clusters[i]]
            cluster_j_points = X[labels == unique_clusters[j]]
            intercluster_dist = np.min(cdist(cluster_i_points, cluster_j_points))
            if intercluster_dist < min_intercluster_dist:
                min_intercluster_dist = intercluster_dist

    # Calculate Dunn index
    if max_intracluster_dist == 0:  # Handle case where all clusters have single points
```

```
        return 0  # or any other appropriate value
    else:
        return min_intercluster_dist / max_intracluster_dist
```

BETA CV:

$$\beta_{cv} = \frac{\text{intra-cluster distances (cohesion)}}{\text{inter-cluster distances (separation)}}$$

Interpretation: Lower Beta CV values indicate better clustering since it shows high separation between clusters and low cohesion (compact clusters).

```
def beta_cv(X, labels):
    n_clusters = len(np.unique(labels))
    unique_clusters = np.unique(labels)
    # Cohesion: Intra-cluster distances (within-cluster compactness)
    intra_distances = []
    for cluster in unique_clusters:
        points_in_cluster = X[labels == cluster]
        intra_distances.append(np.mean(pdist(points_in_cluster)))
    # Separation: Inter-cluster distances (between-cluster separation)
    inter_distances = []
    for i in range(n_clusters):
        for j in range(i+1, n_clusters):
            points_cluster_i = X[labels == unique_clusters[i]]
            points_cluster_j = X[labels == unique_clusters[j]]
            inter_distances.append(np.mean(pairwise_distances(points_cluster_i, points_cluster_j)))
    cohesion = np.mean(intra_distances)
```

```
    separation = np.mean(inter_distances)

    return cohesion / separation  # Beta CV = Cohesion / Separation
```

ALGORITHMS:

K Means Clustering:

```
import pandas as pd

from sklearn.preprocessing import StandardScaler, LabelEncoder

from sklearn.cluster import KMeans

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.decomposition import PCA

from sklearn.metrics import silhouette_score, pairwise_distances

from scipy.spatial.distance import pdist

from sklearn.metrics import davies_bouldin_score

# Initialize K-Means with a certain number of clusters (say 5)

kmeans = KMeans(n_clusters=5, random_state=42)

kmeans.fit(X_scaled)

# Add the cluster labels to the dataset

data['Cluster'] = kmeans.labels_

# Calculate the Silhouette Score to evaluate the quality of the clusters

from sklearn.metrics import silhouette_score

silhouette_avg = silhouette_score(X_scaled, kmeans.labels_)

print(f"Silhouette Score: {silhouette_avg}")

sse = kmeans.inertia_  # Sum of Squared Errors (inertia)

print(f"SSE (Sum of Squared Errors): {sse}")

dunn = dunn_index(X_scaled, kmeans.labels_)
```

```
print(f"Dunn's Index: {dunn}")

beta_cv_value = beta_cv(X_scaled, kmeans.labels_)

print(f"Beta CV (Cohesion vs Separation Index): {beta_cv_value}")
```

K MEDOIDS:

```
#!pip install scikit-learn-extra

#!pip install ucimlrepo

#!pip install pyclustering

from ucimlrepo import fetch_ucirepo

import pandas as pd

from sklearn.preprocessing import StandardScaler, LabelEncoder

from sklearn_extra.cluster import KMedoids

import matplotlib.pyplot as plt

import pandas as pd

from sklearn.preprocessing import StandardScaler, LabelEncoder

from pyclustering.cluster.kmedoids import kmedoids

from pyclustering.cluster import cluster_visualizer

from pyclustering.utils import distance_metric, type_metric

from sklearn.metrics import silhouette_score, davies_bouldin_score

import numpy as np

#continue from where you left behind..

data_sampled = data.sample(5000, random_state=42)

# Step 5: Scale the features of the sampled data

scaler = StandardScaler()

X_sampled_scaled = scaler.fit_transform(data_sampled)

# Step 6: Initialize K-Medoids

# Choose initial medoids randomly (indices can be adjusted for testing)
```

```
initial_medoids = [0, 100]  # Random indices as initial medoids

# Step 7: Set the distance metric (Euclidean)

metric = distance_metric(type_metric.EUCLIDEAN)

# Step 8: Apply K-Medoids using PyClustering

kmedoids_instance = kmedoids(X_sampled_scaled, initial_medoids, metric=metric)

kmedoids_instance.process()

# Get clusters and medoids

clusters = kmedoids_instance.get_clusters()

medoids = kmedoids_instance.get_medoids()

# Step 9: Convert cluster labels for each point

# Create a list of labels based on clusters

labels = np.zeros(len(X_sampled_scaled), dtype=int)

for cluster_id, cluster in enumerate(clusters):

    for index in cluster:

        labels[index] = cluster_id

# Step 10: Calculate SSE (Sum of Squared Errors for K-Medoids)

sse_kmedoids = np.sum([np.linalg.norm(X_sampled_scaled[i] - X_sampled_scaled[medoids[labels[i]]])**2
for i in range(len(X_sampled_scaled))])

print(f"SSE (Sum of Squared Errors for K-Medoids): {sse_kmedoids}")

# Step 11: Calculate Silhouette Score

silhouette_avg_kmedoids = silhouette_score(X_sampled_scaled, labels)

print(f"Silhouette Score for K-Medoids: {silhouette_avg_kmedoids}")

dunn = dunn_index(X_sampled_scaled, labels)

print(f"Dunn's Index: {dunn}")

beta_cv_value = beta_cv(X_sampled_scaled, labels)

print(f"Beta CV (Cohesion vs Separation Index): {beta_cv_value}")
```

EM CLUSTERING:

```
from sklearn.mixture import GaussianMixture

gmm = GaussianMixture(n_components=5, random_state=42)

gmm.fit(X_scaled)

data['Cluster'] = gmm.predict(X_scaled)

# Step 6: Calculate SSE (Sum of Squared Errors for GMM)

# We can calculate SSE as the sum of squared distances from each point to its assigned Gaussian
component mean

means = gmm.means_

labels = gmm.predict(X_scaled)

#SSE

sse_gmm = np.sum([np.linalg.norm(X_scaled[i] - means[labels[i]])**2 for i in range(len(X_scaled))])

print(f"SSE (Sum of Squared Errors for GMM): {sse_gmm}")

# Step 7: Calculate Silhouette Score

silhouette_avg_gmm = silhouette_score(X_scaled, labels)

print(f"Silhouette Score for GMM: {silhouette_avg_gmm}")

dunn = dunn_index(X_scaled, labels)

print(f"Dunn's Index: {dunn}")

beta_cv_value = beta_cv(X_scaled, labels)

print(f"Beta CV (Cohesion vs Separation Index): {beta_cv_value}")
```

DBSCAN Clustering

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a popular clustering algorithm, especially suited for finding clusters of varying shapes and dealing with noise (outliers) in data. Here's a detailed explanation that should make you comfortable enough to teach it to someone:

1. What is Clustering?

- Clustering is the process of grouping a set of objects into clusters, where objects in the same cluster are more similar to each other than to those in other clusters.
- Unlike traditional clustering methods like K-means, DBSCAN doesn't require you to specify the number of clusters beforehand.

2. What is DBSCAN?

- DBSCAN stands for Density-Based Spatial Clustering of Applications with Noise.
- It groups points that are closely packed together, marking points that are in sparse regions as outliers or noise.
- The key idea is to use the density of points to identify clusters, where dense regions form clusters and sparse regions form boundaries between clusters or outliers.

3. Key Concepts in DBSCAN

DBSCAN has three main concepts:

- **ε (epsilon)**: The radius or neighborhood distance around a point. A point is considered a neighbor if it lies within this distance from another point.

- **MinPts**: The minimum number of points required to form a dense region (a cluster). A point must have at least MinPts within its ε radius to be considered as a core point.
- Core Point, Border Point, and Noise:
    - Core Point: A point with at least MinPts neighbors (including itself) within the ε distance.
    - Border Point: A point that is not a core point but lies within the ε radius of a core point.
    - Noise (Outlier): A point that is neither a core point nor a border point.

4. How DBSCAN Works?

DBSCAN iteratively explores the data using the following steps:

- Step 1: Pick an unvisited point randomly.
- Step 2: Check if it is a core point.
    - If it is a core point, form a cluster by finding all points that are density-reachable from this core point.
    - If it is not a core point, mark it as noise for now (it might later be part of a cluster if found as a border point).
- Step 3: Repeat the process until all points are visited and assigned to a cluster or identified as noise.
- Step 4: Merge clusters if they are density-connected through shared points.

5. Important Terms:

- **Directly Density-Reachable**: A point q is directly density-reachable from point p if p is a core point, and q is within the ε radius of p.
- **Density-Reachable**: A point q is density-reachable from point p if there is a chain of points between them where each point is directly density-reachable from the previous one.
- **Density-Connected**: Two points p and q are density-connected if there exists a third point o such that both p and q are density-reachable from o.

6. Visual Example:

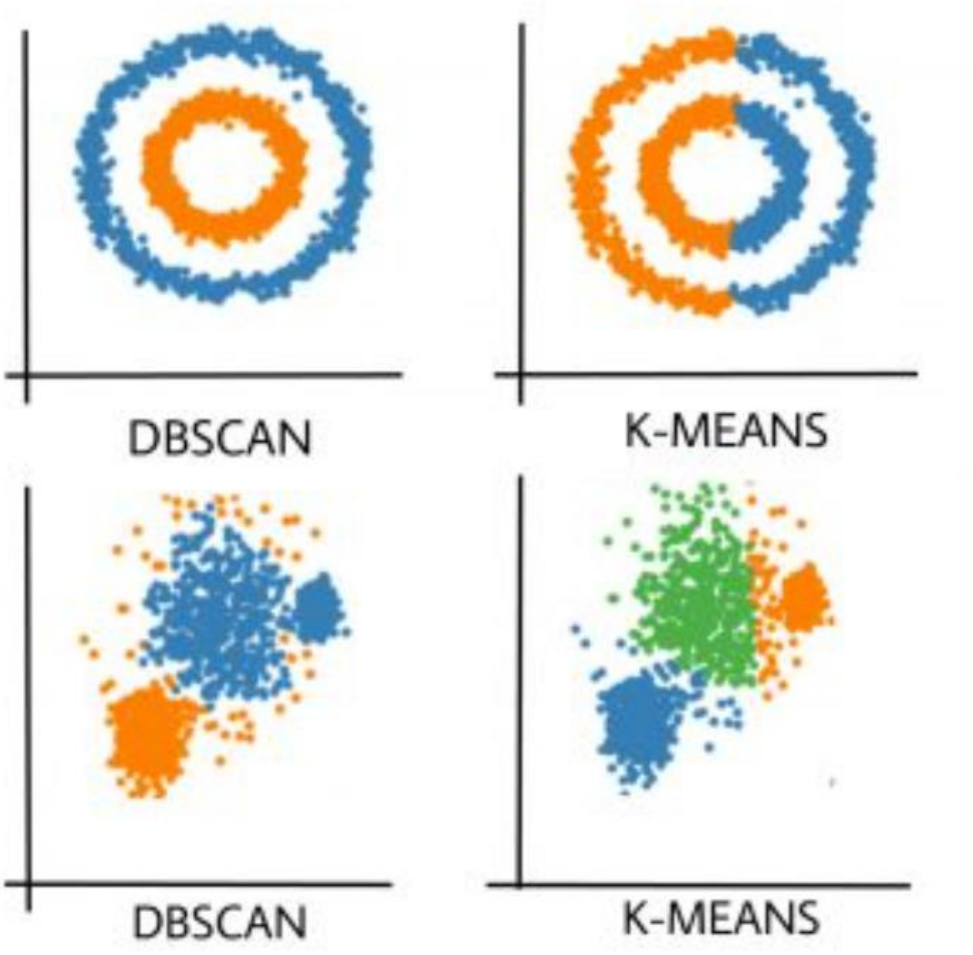
- Imagine a map of scattered data points representing towns. DBSCAN would identify densely populated areas (clusters) as cities, and sparsely populated areas as rural (noise or outliers).
- The ε radius defines the local neighborhood of a town, and MinPts is the number of neighboring towns needed to form a city.

7. Advantages of DBSCAN:

- Can Find Arbitrary Shaped Clusters: Unlike methods like K-means that find spherical clusters, DBSCAN can find clusters of any shape.
- Handles Noise and Outliers: It is effective at identifying noise in the data, making it robust for real-world data sets.
- Does Not Require Number of Clusters: Unlike K-means, DBSCAN does not need the user to specify the number of clusters in advance.

8. Limitations of DBSCAN:

- Choosing ε and MinPts is Crucial: Selecting inappropriate values for ε and MinPts can lead to poor results. Too small an ε might mark many points as noise, and too large an ε may merge distinct clusters.
- Not Ideal for Clusters with Varying Densities: If clusters have different densities, DBSCAN might fail to properly distinguish them.
- Computational Complexity: For large datasets, DBSCAN can be slow as it requires calculating distances between each pair of points.



9. Core Mathematical Concepts:

- **Distance Metric:** DBSCAN uses a distance metric to measure the similarity between points. The most common distance metric is the Euclidean distance, but other metrics (like Manhattan distance) can also be used.
- **Neighborhood:** The neighborhood of a point p is defined as the set of points within a radius ε (epsilon). Mathematically, this is represented as:

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

  where D is the set of all points, dist is the distance function (typically Euclidean distance), and N_ε(p) is the ε-neighborhood of p.

10. Defining Core, Border, and Noise Points:

- **Core Point:** A point p is a core point if the number of points in its ε-neighborhood is at least MinPts. Mathematically:

  $$| N_\epsilon(p) | \geq MinPts$$

  Here, |N_ε(p)| represents the cardinality of the ε-neighborhood of p (i.e., the number of points in the neighborhood).
- **Border Point:** A point q is a border point if it is not a core point, but it lies within the ε-neighborhood of a core point p. In other words, it satisfies:

  $$| N_\epsilon(q) | < MinPts \;\&\&\; \exists p \in N_\epsilon(q)\; s.t. | N_\epsilon(p) | \geq MinPts$$
- **Noise Point:** A point p is considered noise if it is neither a core point nor a border point. It is isolated, lying outside the ε radius of any core point.

11. Density Reachability:

- **Directly Density-Reachable:** A point q is directly density-reachable from a point p if q lies within the ε-neighborhood of p and p is a core point. Mathematically:

  $$q \in N\epsilon(p) \;\&\&\; | N\epsilon(p) | \geq MinPts$$
- **Density-Reachable:** A point q is density-reachable from a point p if there exists a chain of points (p1,p2,…,pn) such that:
  - $p_1 = p \; and \; p_n = q$
  - For each i, p_{i+1} is directly density-reachable from p_i..
- **Density-Connected:** Two points p and q are density-connected if there exists a point o such that both p and q are density-reachable from o. This means that even if p and q are not directly connected, they belong to the same cluster as long as they share a connection through other core points.

12. Mathematical Steps of the DBSCAN Algorithm:

The clustering process of DBSCAN can be described mathematically as follows:

- **Step 1:** For each point p in the dataset D, determine the ε-neighborhood N_ε(p).
- **Step 2:** Identify whether p is a core point by checking if:

$$| N\epsilon(p) | \geq MinPts$$

- **Step 3:** If p is a core point, form a cluster C by including all points that are density-reachable from p.
- **Step 4:** Iterate through each point that is density-reachable from p:
  - If a point is directly density-reachable from p, include it in the same cluster.
  - Recursively expand the cluster by visiting each directly density-reachable point.
- **Step 5:** If p is not a core point, mark it as noise or check if it is a border point (by verifying if it lies within the neighborhood of any core point).

13. Choosing Parameters ε and MinPts:

- The selection of ε affects the size of the neighborhoods, while MinPts influences the minimum density of a cluster.
- A **k-distance graph** can be used to select ε:
  - Sort distances of each point to its k-th nearest neighbor (where k = MinPts).
  - Plot these distances and look for the "elbow" point (a sudden increase in distance). This is a good estimate for ε, as it marks the transition from dense to sparse regions.

14. Euclidean Distance Calculation:

- For two points p=(p1,p2,…,pd) and q=(q1,q2,…,qd) in a d-dimensional space, the Euclidean distance between p and q is given by:

$$dist(p,q) = (\sum(pi - qi)^2)^{1/2}$$

  This formula calculates the straight-line distance between two points in space.

15. Complexity Analysis:

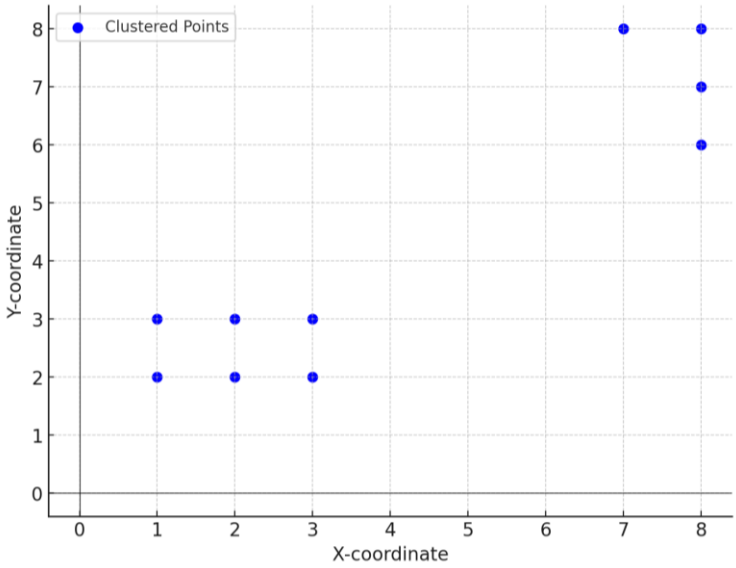
- **Time Complexity:** The computational complexity of DBSCAN is approximately O(n^2) due to the need to calculate distances between all pairs of points. However, using spatial indexing structures like **k-d trees** can reduce the complexity to O(n log n)

- **Space Complexity:** DBSCAN requires storage for the dataset, neighborhood information, and cluster labels, making its space complexity around $O(n)$.

16. Example Problem for Practice:

Suppose you have the following 2D points: $[(1, 2), (2, 2), (3, 2), (8, 7), (8, 8), (1, 3), (2, 3), (3, 3), (8, 6), (7, 8)]$. Let $\varepsilon = 1.5$ and $MinPts = 2$.

- Calculate the ε-neighborhood for each point.
- Identify core, border, and noise points.
- Form clusters based on density-reachability.
- Determine which points end up being noise.



10. Real-World Applications of DBSCAN:

- Geographical Clustering: Finding clusters of geographic data points like restaurants or stores.
- Image Segmentation: Clustering pixel data to find regions of interest in images.
- Fraud Detection: Identifying abnormal patterns or outliers in financial transactions.
- Anomaly Detection: In various domains like IoT, identifying rare events or unusual data points.

Summary:

DBSCAN is a powerful clustering algorithm that can identify clusters of varying shapes and sizes, effectively handle outliers, and does not need you to predefine the number of clusters. The core concepts of ε and MinPts help define which points form dense clusters, while points that do not fit well into clusters are marked as noise.

Implementation:

```
from sklearn.cluster import DBSCAN
import numpy as np

#Load the Bank Marketing Dataset
data = X

#Preprocessing the data
# Encode categorical variables
label_encoder = LabelEncoder()

# Apply Label Encoding to all categorical columns
for col in ['job', 'marital', 'education', 'default', 'housing', 'loan',
'contact', 'month', 'poutcome']:
    data[col] = label_encoder.fit_transform(data[col])

# Drop 'duration' column as it can bias the result (it's highly correlated
with target)
#data = data.drop(columns=['duration'])
# Scale the features
# Standardize the dataset for better performance with K-Means
scaler = StandardScaler()
X_scaled = scaler.fit_transform(data)  # Exclude target column for
clustering

# DBSCAN model
dbscan = DBSCAN(eps=3, min_samples=2)
dbscan.fit(X_scaled)

# Results
print("Labels:", dbscan.labels_)
print("Core Sample Indices:", dbscan.core_sample_indices_)
print("Number of clusters:", len(set(dbscan.labels_)) - (1 if -1 in
dbscan.labels_ else 0))
```

```
print("Number of noise points:", list(dbscan.labels_).count(-1))

labels = dbscan.labels_

# Calculate the SSE for each cluster
sse = 0.0
unique_labels = set(labels)

for label in unique_labels:
    if label == -1:
        # Ignore noise points (label -1)
        continue

    # Extract the points in the current cluster
    cluster_points = X[labels == label]

    # Calculate the centroid of the cluster
    centroid = cluster_points.mean()

    # Compute the sum of squared distances from each point to the centroid
    sse += np.sum((cluster_points - centroid) ** 2)

print("SSE:\n", sse)

#Calculate Silhouette Score
silhouette_avg_dbscan = silhouette_score(X_scaled, labels)
print(f"Silhouette Score for dbscan: {silhouette_avg_dbscan}")

dunn = dunn_index(X_scaled, dbscan.labels_)
print(f"Dunn's Index: {dunn}")

beta_cv_value = beta_cv(X_scaled, kmeans.labels_)
print(f"Beta CV (Cohesion vs Separation Index): {beta_cv_value}")
```

SLINK Clustering

SLINK (Single-Linkage Hierarchical Clustering) is an algorithm for constructing a hierarchical clustering of a dataset using the single-linkage strategy. It builds a dendrogram (a tree-like structure that represents nested cluster formations) without having to recompute the entire distance matrix at each step. This makes SLINK more efficient than a naïve single-linkage clustering algorithm, especially for large datasets.

What is Hierarchical Clustering?

Hierarchical clustering is a method to group objects into clusters based on their similarities, creating a nested structure of clusters. There are two main types of hierarchical clustering:

- **Agglomerative (Bottom-up)**: Each object starts as its own cluster, and the algorithm iteratively merges the closest clusters until only one cluster remains.
- **Divisive (Top-down)**: All objects start in one cluster, and the algorithm recursively splits clusters until each object is in its own cluster.

SLINK is an agglomerative method, meaning it builds the hierarchy from the bottom up by merging clusters.

What is Single-Linkage Clustering?

Single-linkage clustering, also known as the **nearest neighbor method**, defines the distance between two clusters as the **minimum distance** between any pair of points from the two clusters.

Mathematically, if A and B are two clusters, then the distance d(A, B) between them is:

$d(A, B) = \min(||a - b||)$ for $a \in A$ and $b \in B$

where a is in A, b is in B, and ||a - b|| denotes the Euclidean distance between points a and b.

How Does SLINK Work?

The SLINK algorithm allows for a more efficient computation of hierarchical clustering using single linkage. It processes the data in an incremental way without having to repeatedly compute distances between all pairs of points during each merge.

Step-by-Step Explanation:

1. **Initialization**:
    - Assume we have n points X = $\{x_1, x_2, \ldots, x_n\}$.
    - Create a **distance matrix** D of size n x n, where D[i][j] is the distance between points x_i and x_j.
    - Initially, each point x_i is its own cluster.
2. **Representation of Clusters**:
    - As we proceed, we track two arrays R and L:
        - R[j]: The **minimum distance** at which the point x_j will be connected to any other point (or cluster).
        - L[j]: The **index** of the point that x_j is connected to at the distance R[j].
3. **Incremental Update**:
    - Process points incrementally. For each point x_k (starting from k = 2 to n):
        - Compare x_k to every previous point $x_1, x_2, \ldots, x_{k-1}$.
        - Update the R and L arrays such that R[i] stores the distance between x_i and the closest other point that was encountered so far.
4. **Merging Clusters**:
    - At each step, R and L define the next merge in the clustering:
        - If R[i] is the smallest value, then the merge occurs between x_i and the point L[i].
        - This means that x_i is joined to the cluster that L[i] belongs to at distance R[i].
5. **Repeat** until all points are merged into a single cluster. The algorithm generates a series of merges with distances, which can be visualized as a dendrogram.

Mathematical Details of SLINK

The SLINK algorithm maintains efficient updates to distances and merges by keeping track of nearest neighbors in a way that avoids recalculating distances multiple times. Here's the math behind the updates:

1. **Updating Distances**:
    - For a new point x_k, calculate the distances to all previous points $x_1, x_2, \ldots, x_{k-1}$: $d_{\{ik\}} = ||x_i - x_k||$ for i < k where d_{ik} represents the distance between x_i and x_k.
2. **Update Arrays R and L**:
    - For each point x_i with i < k:

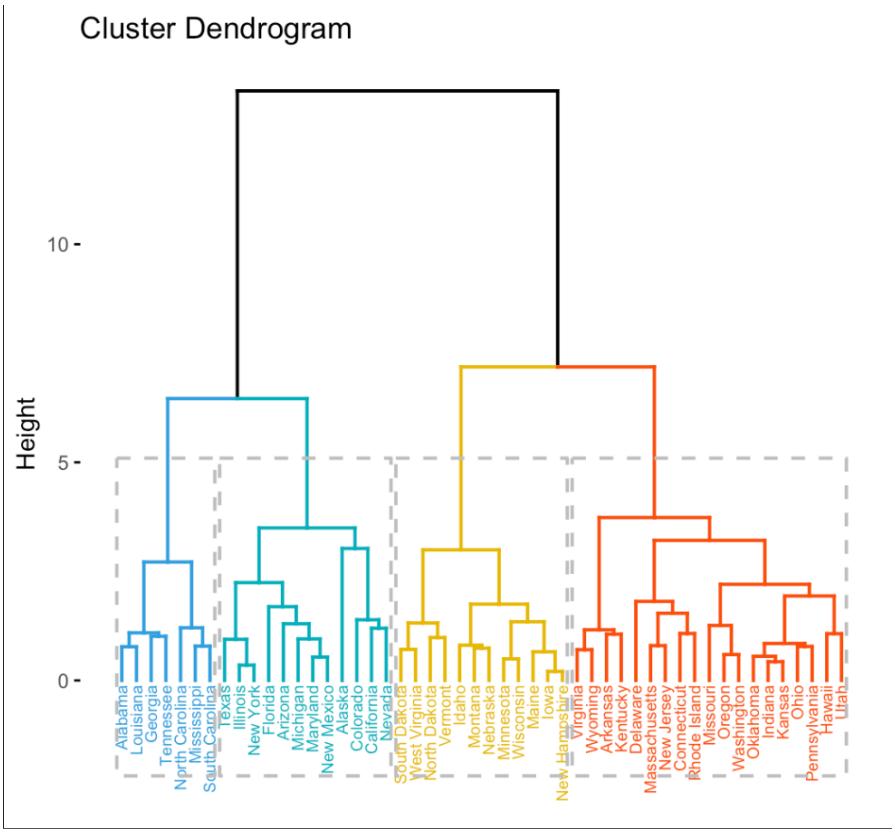
        - Update R[i] as: $R[i] = min(R[i], d_{\{ik\}})$ This ensures that R[i] always stores the minimum distance to any point connected with x_i.
        - Update L[i] to point to x_k if d_{ik} is the minimum distance: $L[i] = k$ if $d_{\{ik\}} \leq R[i]$
3. **Merge Criterion**:
    - Merge the clusters of x_i and x_k if $d_{\{ik\}} \leq R[i]$. The resulting distance becomes part of the dendrogram.

Complexity of SLINK

- **Time Complexity**: The time complexity of SLINK is O(n^2), primarily due to the distance calculations between points. Although this is the same as other naïve hierarchical algorithms, SLINK is more efficient in practice because it avoids recalculating the full distance matrix at every merge.
- **Space Complexity**: The space complexity is O(n^2) for storing the distance matrix. However, SLINK uses less space compared to building a full dendrogram.

Dendrogram Representation

- The SLINK algorithm produces a **linkage matrix** that records which clusters are merged at each step and at what distance.
- This matrix can be used to construct a **dendrogram** (a tree representation of cluster merges).
- Cutting the dendrogram at different levels (threshold distances) results in a different number of clusters.
- Here is an image showing a typical dendrogram.



Example

Consider 4 points A, B, C, and D:

- Suppose we have the following pairwise distances between these points:
  Distances: {d(A, B) = 2, d(A, C) = 4, d(A, D) = 5, d(B, C) = 3, d(B, D) = 6, d(C, D) = 1}
- SLINK will start with each point as its own cluster:
  - At k=1, A forms a singleton cluster.
  - At k=2, SLINK processes B:
    - d(A, B) = 2, so A and B merge.
  - At k=3, SLINK processes C:
    - The closest distance is between B and C, so B and C merge.
  - At k=4, SLINK processes D:
    - D is closest to C, so D is merged with the C cluster.

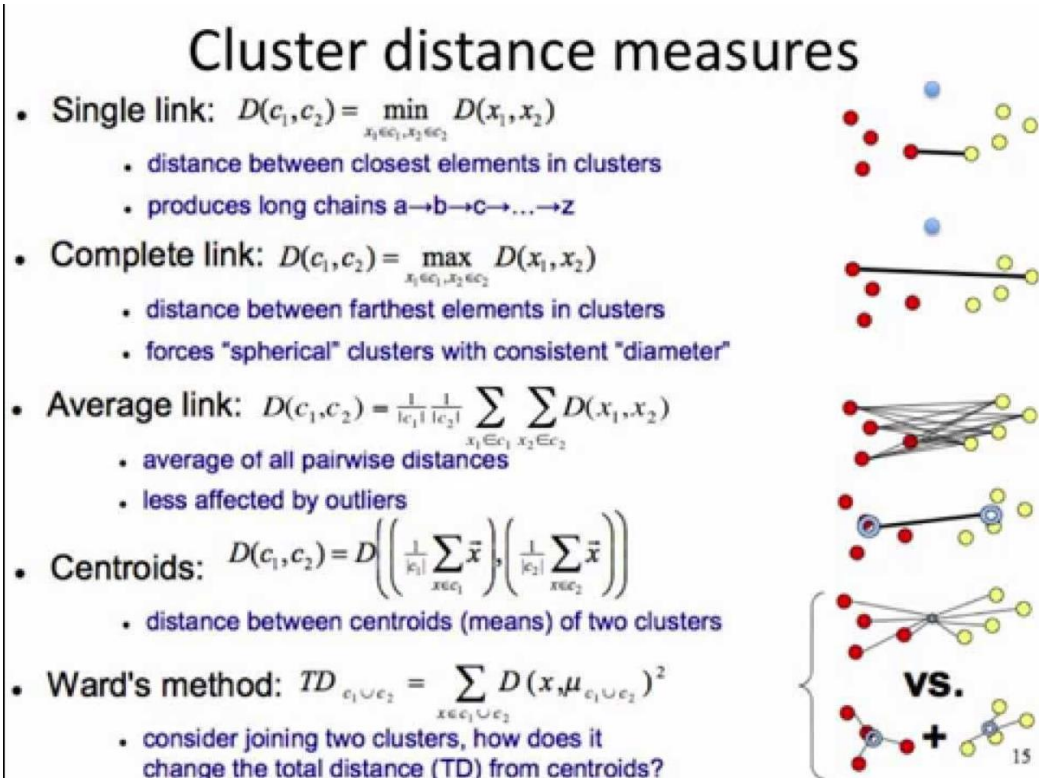
The result is a hierarchical structure where each cluster merge is recorded with the distance between the clusters.

Key Advantages of SLINK

1. **Efficient**: Avoids recalculating distances between all pairs of points, saving time and memory.
2. **Single-Linkage Properties**: Allows for identifying elongated clusters or chaining effects, which some clustering methods may struggle with.
3. **Scalability**: While not ideal for very large datasets due to its O(n^2) complexity, SLINK is still faster than other naive methods for smaller datasets.

Summary

- **SLINK** is a single-linkage hierarchical clustering algorithm that is efficient in how it tracks the nearest neighbor information.
- It incrementally builds a dendrogram by merging clusters based on the closest distance between points.
- **Mathematics** behind SLINK involves maintaining and updating nearest neighbor distances to form clusters.
- It generates a hierarchical tree that can be used to visualize cluster formation at various distance thresholds.
- More variations are depicted in the image below;



Implementation

```
from sklearn.cluster import AgglomerativeClustering

import numpy as np

from sklearn.metrics import silhouette_score



# Load the Bank Marketing Dataset

data = X



# Encode categorical variables

label_encoder = LabelEncoder()



# Apply Label Encoding to all categorical columns

for col in ['job', 'marital', 'education', 'default', 'housing', 'loan',
'contact', 'month', 'poutcome']:

    data[col] = label_encoder.fit_transform(data[col])



# Drop 'duration' column as it can bias the result (it’s highly correlated with
target)

#data = data.drop(columns=['duration'])

# Standardize the dataset for better performance with K-Means

scaler = StandardScaler()

X_scaled = scaler.fit_transform(data)  # Exclude target column for clustering



# Single-linkage clustering (SLINK-like behavior)

single_link_clustering = AgglomerativeClustering(n_clusters=20, linkage='single')

labels = single_link_clustering.fit_predict(X)
```

```
# Results
print("Labels:", single_link_clustering.labels_)

labels = single_link_clustering.labels_
sse = 0.0
unique_labels = set(labels)

for label in unique_labels:
    if label == -1:
        # Ignore noise points (label -1)
        continue

    # Extract the points in the current cluster
    cluster_points = X[labels == label]

    # Calculate the centroid of the cluster
    centroid = cluster_points.mean()

    # Compute the sum of squared distances from each point to the centroid
    sse += np.sum((cluster_points - centroid) ** 2)

print("SSE:\n", sse)
```

```
silhouette_avg_slink = silhouette_score(X_scaled, labels)

print(f"Silhouette Score for slink: {silhouette_avg_slink}")



dunn = dunn_index(X_scaled, single_link_clustering.labels_)

print(f"Dunn's Index: {dunn}")



beta_cv_value = beta_cv(X_scaled, kmeans.labels_)

print(f"Beta CV (Cohesion vs Separation Index): {beta_cv_value}")
```