Aim: To Study and Implement K-Medoids algorithm

Objective: Understand the working of K-Medoids algorithm and its implementation using Python.

Theory:
K-Medoids is a clustering algorithm that is very similar to K-Means. However, instead of choosing means (centroids) as the central point for each cluster, it chooses actual data points (medoids) to minimize the sum of dissimilarities between the data points and the medoids. K-Medoids is more robust to noise and outliers compared to K-Means because it uses actual data points as cluster centers.

Input:

- K: Number of clusters
- D: Dataset containing n objects

Output:

- A set of k clusters

Given k, the K-Medoids algorithm is implemented in 5 steps:

1. Step 1: Arbitrarily choose k objects from D as the initial cluster centers (medoids).
2. Step 2: Find the dissimilarity (e.g., Euclidean distance) between each object in the dataset and the medoids.
3. Step 3: Assign each object to the cluster with the nearest medoid.
4. Step 4: Update the medoids by minimizing the sum of the dissimilarities between all objects in a cluster and the medoid. For each cluster, the object that minimizes this sum becomes the new medoid.
5. Step 5: Repeat the process until there is no change in the medoids.

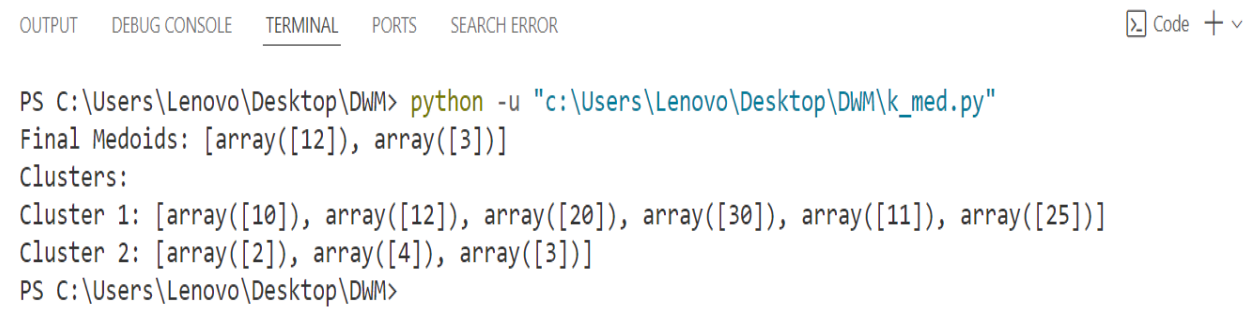
Example:
Let the dataset D = {2, 4, 10, 12, 3, 20, 30, 11, 25}, and k = 2 clusters.

1. **Randomly assign initial medoids**: Assume m1 = 3 and m2 = 20.
    - Cluster 1 (k1) = {2, 3, 4},
    - Cluster 2 (k2) = {10, 12, 20, 30, 11, 25}.
2. **Calculate distances and reassign medoids**:
    - After calculating the dissimilarities, update medoids to m1 = 4 and m2 = 25.
    - Cluster 1 (k1) = {2, 3, 4, 10, 12, 11},
    - Cluster 2 (k2) = {20, 30, 25}.
3. **Update medoids and repeat**:
    - Continue updating medoids and clusters until the medoids stop changing.
4. **Final Medoids and Clusters**:
    - Cluster 1 (k1) = {2, 3, 4, 10, 12, 11},
    - Cluster 2 (k2) = {20, 30, 25}.

CODE:

```
import numpy as np
from sklearn.metrics import pairwise_distances
import random

def k_medoids(D, k, max_iterations=100):
    medoids = random.sample(list(D), k)

    for _ in range(max_iterations):
        distances = pairwise_distances(D, medoids, metric='euclidean')

        clusters = {i: [] for i in range(k)}
        for idx, row in enumerate(distances):
            cluster_idx = np.argmin(row)
            clusters[cluster_idx].append(D[idx])

        new_medoids = []
        for cluster in clusters.values():
            intra_cluster_distances = pairwise_distances(cluster, metric='euclidean')
            total_dissimilarity = np.sum(intra_cluster_distances, axis=1)
            new_medoids.append(cluster[np.argmin(total_dissimilarity)])

        if np.all(new_medoids == medoids):
            break
        medoids = new_medoids

    return medoids, clusters

D = np.array([2, 4, 10, 12, 3, 20, 30, 11, 25]).reshape(-1, 1)
k = 2

medoids, clusters = k_medoids(D, k)

print("Final Medoids:", medoids)
print("Clusters:")
for cluster_idx, cluster in clusters.items():
    print(f"Cluster {cluster_idx+1}: {cluster}")
```

OUTPUT:



CONCLUSION:

In this practical, we studied and implemented the K-Medoids clustering algorithm using Python. We learned how K-Medoids selects actual data points as cluster centers (medoids) instead of calculating centroids, making it more robust to noise and outliers.Overall, this exercise enhanced our understanding of clustering techniques and their applications in data analysis.

What types of data preprocessing are necessary before applying the K-Medoids algorithm?

Handling Missing Values: Identify and handle missing data in your dataset. You can either remove rows with missing values, impute them using mean, median, or mode, or use more sophisticated methods depending on the context.

Normalization/Standardization: Since K-Medoids uses distance measures to form clusters, it's crucial to scale your features so they contribute equally. Normalize (scale between 0 and 1) or standardize (mean = 0, standard deviation = 1) the data, especially if your features have different units or scales.

Outlier Detection: While K-Medoids is robust to outliers, it's still a good practice to identify and potentially handle extreme values, as they can influence the choice of medoids and cluster assignments.

Data Transformation: Depending on your dataset, you may need to apply transformations (e.g., log transformation) to skewed data to meet the algorithm's assumptions better.

Categorical Encoding: If your dataset contains categorical variables, you'll need to convert them into numerical formats using techniques such as one-hot encoding or label encoding, since K-Medoids operates on numerical data.

Dimensionality Reduction: If your dataset has many features, consider using techniques like PCA (Principal Component Analysis) to reduce dimensionality while retaining variance, which can improve the efficiency and effectiveness of the clustering process.

Feature Selection: Identify and select the most relevant features that contribute significantly to clustering. This step can help reduce noise and improve the clustering performance.