| **Name:** | **James Lewis** |
| --- | --- |
| **Roll No:** | **32** |
| **Class/Sem:** | TE/V |
| **Experiment No.:** | 8 |
| **Title:** | Implementation of any one clustering algorithm using languages like JAVA/ python. |
| **Date of Performance:** |  |
| **Date of Submission:** |  |
| **Marks:** |  |
| **Sign of Faculty:** |  |

**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

class KMedoids:

def \_\_init\_\_(self, n\_clusters=2, max\_iter=300, random\_state=None):

self.n\_clusters = n\_clusters

self.max\_iter = max\_iter

self.random\_state = random\_state

self.medoids\_ = None

self.labels\_ = None

def fit(self, X):

np.random.seed(self.random\_state)

m = X.shape[0]

medoid\_indices = np.random.choice(m, self.n\_clusters, replace=False)

medoids = X[medoid\_indices]

for \_ in range(self.max\_iter):

distances = pairwise\_distances(X, medoids)

labels = np.argmin(distances, axis=1)

new\_medoids = np.copy(medoids)

for i in range(self.n\_clusters):

cluster\_points = X[labels == i]

if len(cluster\_points) == 0:

continue

intra\_distances = pairwise\_distances(cluster\_points, cluster\_points)

total\_distances = np.sum(intra\_distances, axis=1)

new\_medoids[i] = cluster\_points[np.argmin(total\_distances)]

if np.all(new\_medoids == medoids):

break

medoids = new\_medoids

self.medoids\_ = medoids

self.labels\_ = labels

return self

def fit\_predict(self, X):

self.fit(X)

return self.labels\_

X = np.array([[1, 2], [1, 4], [1, 0],

[4, 2], [4, 4], [4, 0]])

kmedoids = KMedoids(n\_clusters=2, random\_state=0)

labels = kmedoids.fit\_predict(X)

print("Cluster Labels:", labels)

print("Final Medoids:\n", kmedoids.medoids\_)

**OUTPUT:**

Cluster Labels: [1 1 1 0 0 0]

Final Medoids:

[[4 2]

[1 2]]

**CONCLUSION:**

**The K-Medoids algorithm is a robust clustering method, especially in the presence of noise and outliers, because it uses actual points from the dataset as the medoids. It effectively partitions data into k clusters based on minimizing the sum of dissimilarities between points and their assigned medoids.**

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

Before applying K-Medoids, proper data preprocessing is essential for accurate and meaningful clustering, as the algorithm relies on distances between points. Key steps include:

1. Handling Missing Values: Impute missing values (mean, median, KNN) or remove incomplete rows/columns.
2. Feature Scaling / Normalization: Scale features using Min-Max or Standardization to prevent dominance of large-range features.
3. Encoding Categorical Variables: Use One-Hot or Ordinal encoding if a numeric distance metric is required.
4. Outlier Handling (Optional): Remove or limit extreme values using Z-score, IQR, or domain-specific rules.
5. Dimensionality Reduction (Optional): Use PCA, t-SNE, or UMAP for high-dimensional data to improve distance calculations.
6. Distance Metric Selection: Choose Euclidean or Manhattan for numeric data, Hamming for categorical data.