Experiment No. 7

Aim : To implement clustering algorithms.

Problem Statement :

1. **Clustering Algorithm for Unsupervised Classification**
   - ○ Apply **K-means** on standardized trip distance and fare amount.
2. **Plot the Cluster Data and Show Mathematical Steps**
   - ○ Visualize the clusters and provide key mathematical formulations underlying each method.

Theory

1. K-means Clustering

- **Mathematical Steps:**
  1. Selecting K initial cluster centroids randomly.
  2. Assigning each data point to the nearest centroid.
  3. Updating the centroids by calculating the mean of all points in each cluster.
  4. Repeating steps 2 and 3 until convergence (i.e., centroids stop changing significantly).
- **Objective Function:**

$$J = \sum_{i=1}^{k} \sum_{x \in C_i} \|x - \mu_i\|^2$$

Minimizes the sum of squared distances within clusters.

Steps :

Step 1: Data Preparation

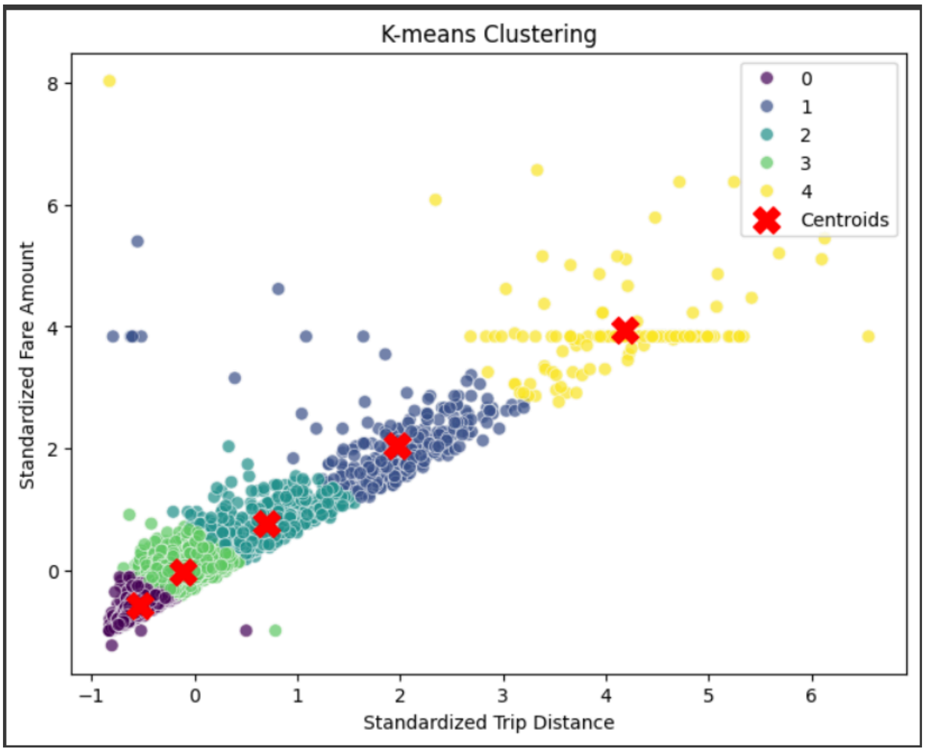
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans, DBSCAN, AgglomerativeClustering
from scipy.cluster.hierarchy import dendrogram, linkage
file_path = "/content/yellow_tripdata_2016-03.csv"
df = pd.read_csv(file_path, parse_dates=['tpep_pickup_datetime',
'tpep_dropoff_datetime'])
df.head()
features = ['trip_distance', 'fare_amount']
df_clean = df[features].dropna()
df_clean = df_clean[(df_clean['trip_distance'] > 0) & (df_clean['fare_amount'] >
0)]
df_sample = df_clean.sample(n=5000, random_state=42)
scaler = StandardScaler()
X = scaler.fit_transform(df_sample)
```

Inference

- **Data Loaded & Parsed**: The CSV is read, and date columns are converted to datetime objects.
- **Feature Selection**: Only trip_distance and fare_amount are retained for clustering.
- **Data Cleaning**: Removes missing entries and filters out any non-positive values.
- **Sampling**: Speeds up computation on large data (5,000 rows).
- **Standardization**: Ensures both features contribute equally to distance measures.

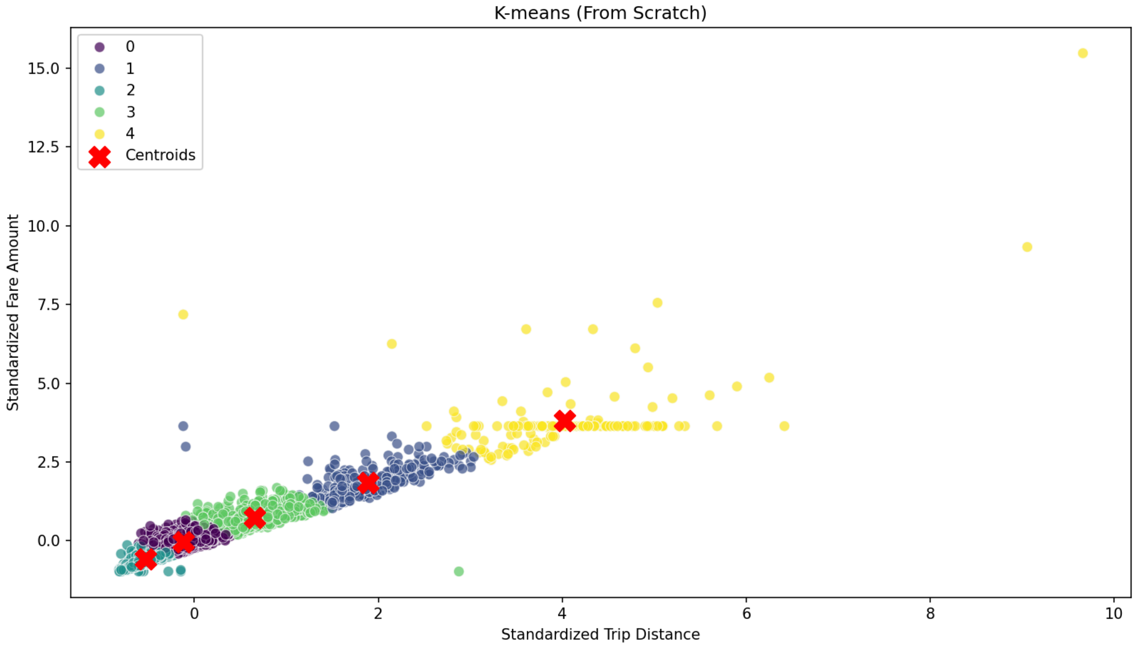
Step 2.1 : K-means Clustering

```
k = 5  # Number of clusters (tune as needed)
kmeans = KMeans(n_clusters=k, random_state=42)
labels_km = kmeans.fit_predict(X)
centroids = kmeans.cluster_centers_
# Plot K-means clusters
plt.figure(figsize=(8, 6))
sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=labels_km, palette='viridis', s=50,
alpha=0.7)
plt.scatter(centroids[:, 0], centroids[:, 1], s=200, c='red', marker='X',
label='Centroids')
plt.title("K-means Clustering")
plt.xlabel("Standardized Trip Distance")
plt.ylabel("Standardized Fare Amount")
plt.legend()
plt.show()
```



Step 2.2 : K-means Clustering (Formula)

```
def kmeans_from_scratch(X, k, max_iter=100, tol=1e-4):
    """
    K-means clustering from scratch.
    """
    n_samples, n_features = X.shape

    # Randomly choose k distinct points from X as initial centroids
    rng = np.random.default_rng(42)
    random_indices = rng.choice(n_samples, size=k, replace=False)
    centroids = X[random_indices].copy()

    for iteration in range(max_iter):
        # Compute distances to each centroid
        distances = np.empty((n_samples, k))
        for i in range(k):
            diff = X - centroids[i]
            distances[:, i] = np.sum(diff * diff, axis=1)  # squared distance

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

        # Update centroids
        old_centroids = centroids.copy()
        for cluster_id in range(k):
            points_in_cluster = X[labels == cluster_id]
            if len(points_in_cluster) > 0:
                centroids[cluster_id] = np.mean(points_in_cluster, axis=0)

        # Check convergence
        shift = np.linalg.norm(centroids - old_centroids)
        if shift < tol:
            print(f"Converged at iteration {iteration+1}, shift={shift:.5f}")
            break

    return labels, centroids

def kmeans_scratch_demo(X, k=5):
    print("=== K-means (From Scratch) ===")
    labels, centroids = kmeans_from_scratch(X, k=k, max_iter=100)
    # Plot
    plt.figure(figsize=(8, 6))
    sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=labels, palette='viridis', s=50, alpha=0.7)
    plt.scatter(centroids[:, 0], centroids[:, 1], s=200, c='red', marker='X',
label='Centroids')
    plt.title("K-means (From Scratch)")
    plt.xlabel("Standardized Trip Distance")
    plt.ylabel("Standardized Fare Amount")
    plt.legend()
    plt.show()
```

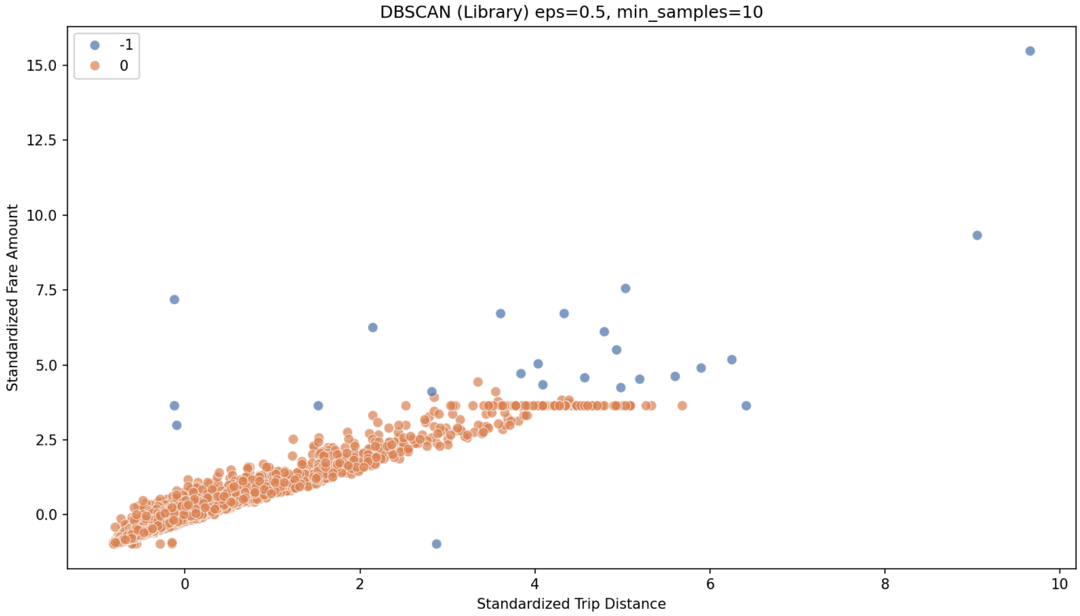


Inference

- **Positive Relationship**: The data shows a roughly linear trend: as distance increases, fare typically increases.
- **Five Clusters**: K-means partitioned the data into 5 groups, each cluster capturing a different range of distance/fare.
- **Centroids**: Red X's mark the center in standardized space for each cluster.
- **Cluster Shapes**: K-means tends to form roughly spherical clusters. The points with very high or low fare/distance appear at the extremes.

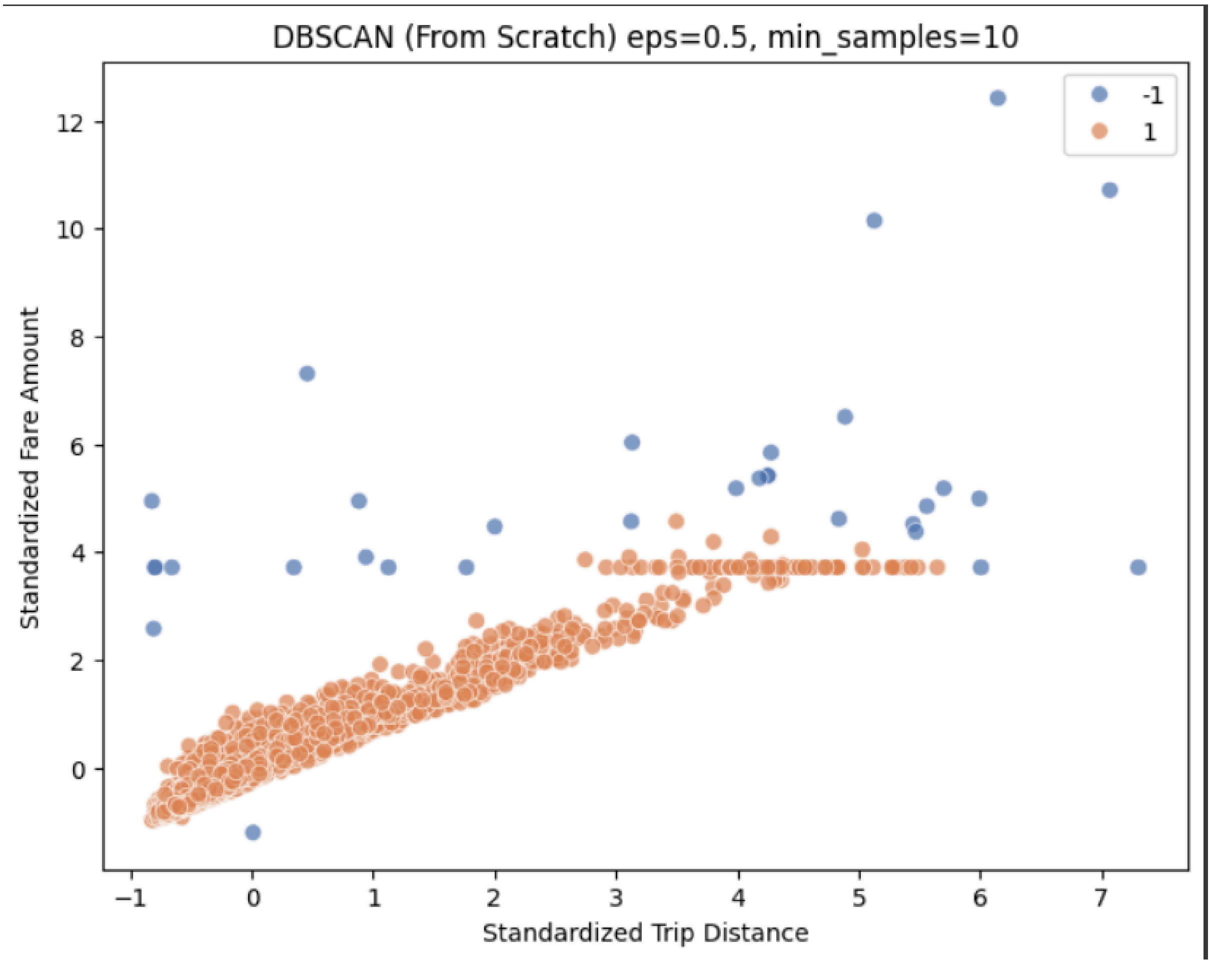
Step 3.1 : DBSCAN Clustering

```
dbscan = DBSCAN(eps=0.5, min_samples=10)
labels_db = dbscan.fit_predict(X)
# Plot DBSCAN clusters (noise points are labeled as -1)
plt.figure(figsize=(8, 6))
sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=labels_db, palette='deep', s=50,
alpha=0.7)
plt.title("DBSCAN Clustering")
plt.xlabel("Standardized Trip Distance")
plt.ylabel("Standardized Fare Amount")
plt.show()
```



Step 3.2 : DBSCAN Clustering (Formula)

```
def dbscan_from_scratch(X, eps=0.5, min_samples=10):
    """
    Basic DBSCAN from scratch:
    - RegionQuery, ExpandCluster, etc.
    """
    n_samples = X.shape[0]
    labels = np.full(n_samples, -1, dtype=int)  # -1 = noise by default
    visited = np.zeros(n_samples, dtype=bool)
    cluster_id = 0

    def euclidean_distance(a, b):
        return np.sqrt(np.sum((a - b)**2))

    def region_query(point_idx):
        neighbors = []
        for i in range(n_samples):
            if euclidean_distance(X[point_idx], X[i]) <= eps:
                neighbors.append(i)
        return neighbors

    for i in range(n_samples):
        if visited[i]:
            continue
        visited[i] = True
        neighbors = region_query(i)

        if len(neighbors) < min_samples:
            labels[i] = -1  # noise
        else:
            # create new cluster
            cluster_id += 1
            labels[i] = cluster_id
            seeds = neighbors.copy()
            seeds.remove(i)  # remove itself if present
            # Expand cluster
            while seeds:
                current_point = seeds.pop()
                if not visited[current_point]:
                    visited[current_point] = True
                    neighbors2 = region_query(current_point)
                    if len(neighbors2) >= min_samples:
                        # add new neighbors
                        for nb in neighbors2:
                            if nb not in seeds:
                                seeds.append(nb)
                if labels[current_point] == -1:
                    labels[current_point] = cluster_id
    return labels

def dbscan_scratch_demo(X, eps=0.5, min_samples=10):
    print("=== DBSCAN (From Scratch) ===")
    labels = dbscan_from_scratch(X, eps=eps, min_samples=min_samples)
    # Plot
    plt.figure(figsize=(8, 6))
    sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=labels, palette='deep', s=50,
alpha=0.7)
    plt.title(f"DBSCAN (From Scratch) eps={eps},
min_samples={min_samples}")
    plt.xlabel("Standardized Trip Distance")
    plt.ylabel("Standardized Fare Amount")
    plt.show()
```



Inference

- **Single Major Cluster**: DBSCAN lumps the majority of points into one cluster (label 0).

- **Noise/Outliers**: Points labeled “-1” deviate from the main density; these may be unusually short or long rides relative to their fares.

- **Parameter Sensitivity**: With eps=0.5 and min_samples=10, you get just one cluster + noise. Different parameters might reveal more subclusters.

- **Linear Trend**: The main cluster still follows the same linear pattern (distance vs. fare).

Conclusion :

In this experiment, clustering algorithms were applied to NYC Yellow Taxi data using standardized trip distance and fare amount. K-means effectively grouped the rides into five clusters, highlighting a clear linear relationship between distance and fare, with each cluster representing different ride patterns. The centroids indicated the average values within each group. In comparison, DBSCAN identified one dense cluster and several outliers, showcasing its ability to detect anomalies. Overall, the experiment demonstrated how unsupervised clustering can reveal patterns and outliers in real-world transportation data for deeper insights.