Experiment No. 7

Aim : To implement different clustering algorithms.

Problem Statement :

1. **Clustering Algorithm for Unsupervised Classification**
    - Apply **K-means**, **DBSCAN**, and **Hierarchical Clustering** 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. **Initialization:** Select k centroids randomly.
    2. **Assignment:** Assign each point x to the nearest centroid μi\mu_iμi (using Euclidean distance).
    3. **Update:** Recompute each centroid as the mean of assigned points.
    4. **Iteration:** Repeat assignment and update until convergence.
- **Objective Function:**

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

Minimizes the sum of squared distances within clusters.

2. DBSCAN (Density-Based Spatial Clustering of Applications with Noise)

- **Key Parameters:**
    1. **eps:** Neighborhood radius.
    2. **min_samples:** Minimum points required to form a dense region.
- **Mathematical Steps:**
    1. Identify **core points** (those with at least min_samples points in their eps-radius).
    2. All points within eps of a core point belong to the same cluster.
    3. Points not reachable from any core point are labeled as noise (cluster = −1-1−1).

3. Hierarchical Clustering (Ward Linkage)

- **Approach:**
    - Start with each point as its own cluster.
    - Iteratively merge the two “closest” clusters.
    - Use **Ward’s method** to minimize within-cluster variance at each merge step.
- **Dendrogram:**
    - Visualizes how clusters merge at different distance thresholds.
    - A horizontal “cut” in the dendrogram determines the final number of clusters.

Steps :

Step 1: Data Preparation

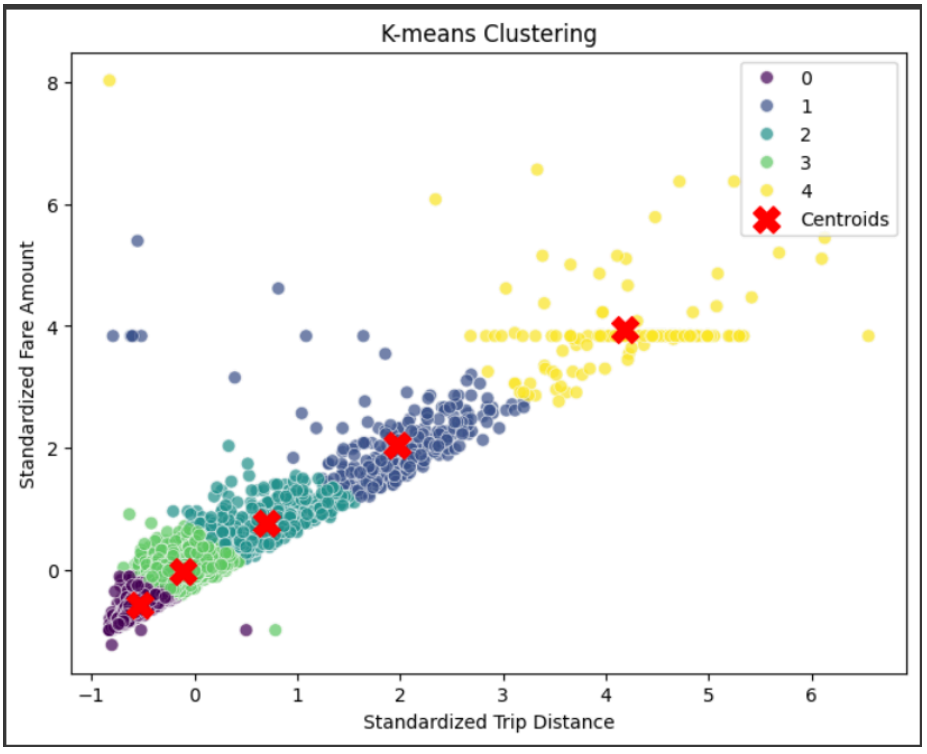
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans, DBSCAN, AgglomerativeClustering
from scipy.cluster.hierarchy import dendrogram, linkage
file_path = "/content/yellow_tripdata_2016-03.csv"
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

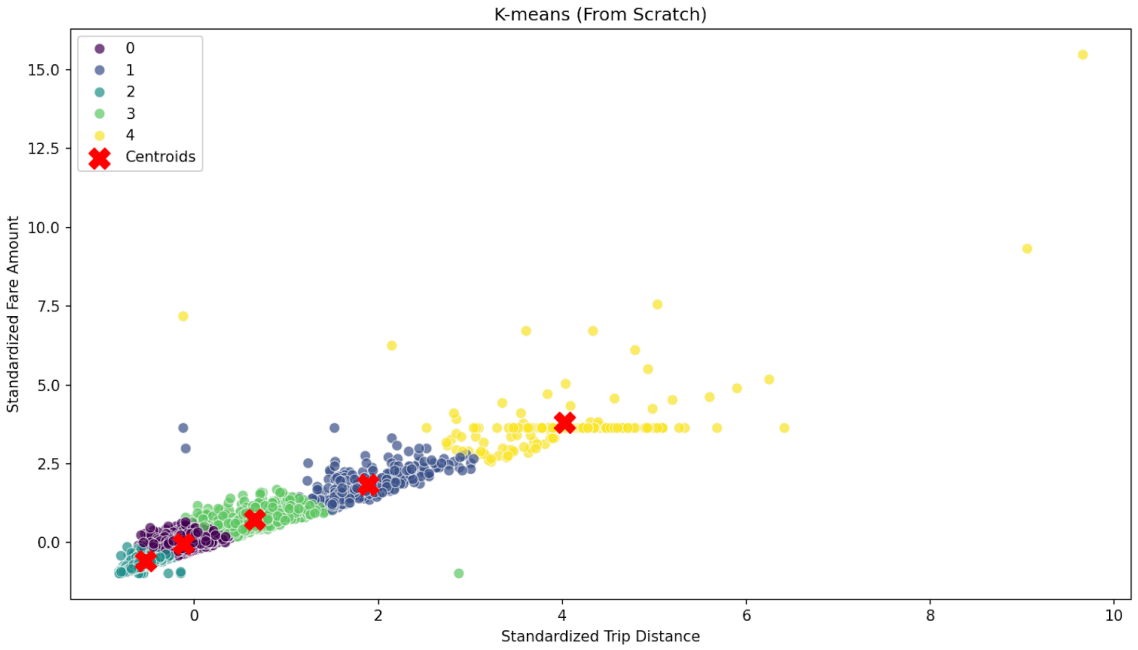
```
k = 5  # Number of clusters (tune as needed)
kmeans = KMeans(n_clusters=k, random_state=42)
labels_km = kmeans.fit_predict(X)
centroids = kmeans.cluster_centers_
# Plot K-means clusters
plt.figure(figsize=(8, 6))
sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=labels_km, palette='viridis', s=50, alpha=0.7)
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()
```

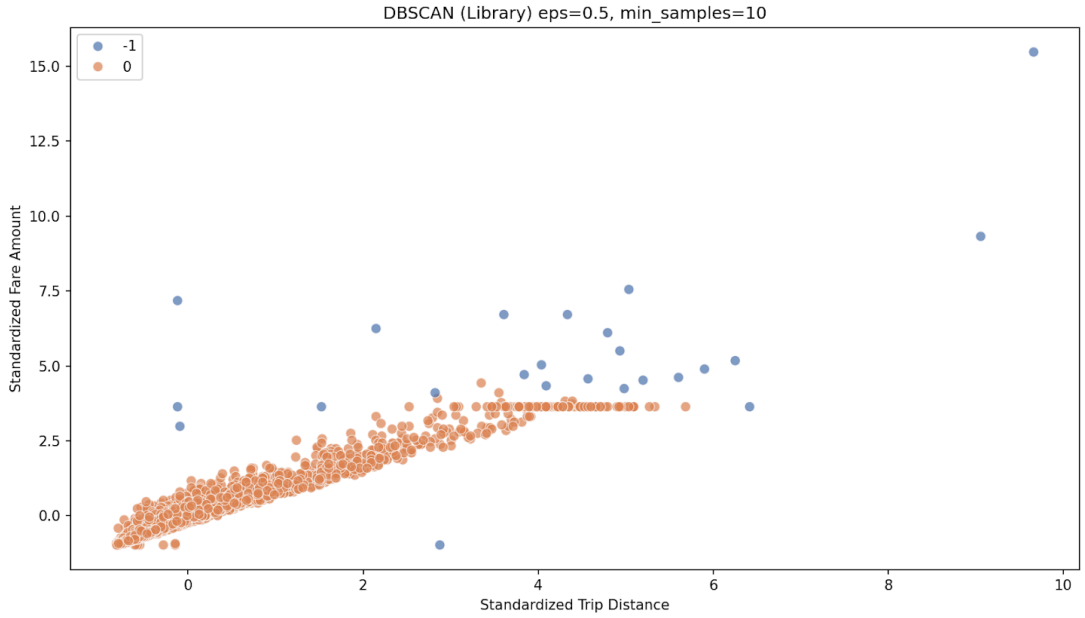


Inference

- **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

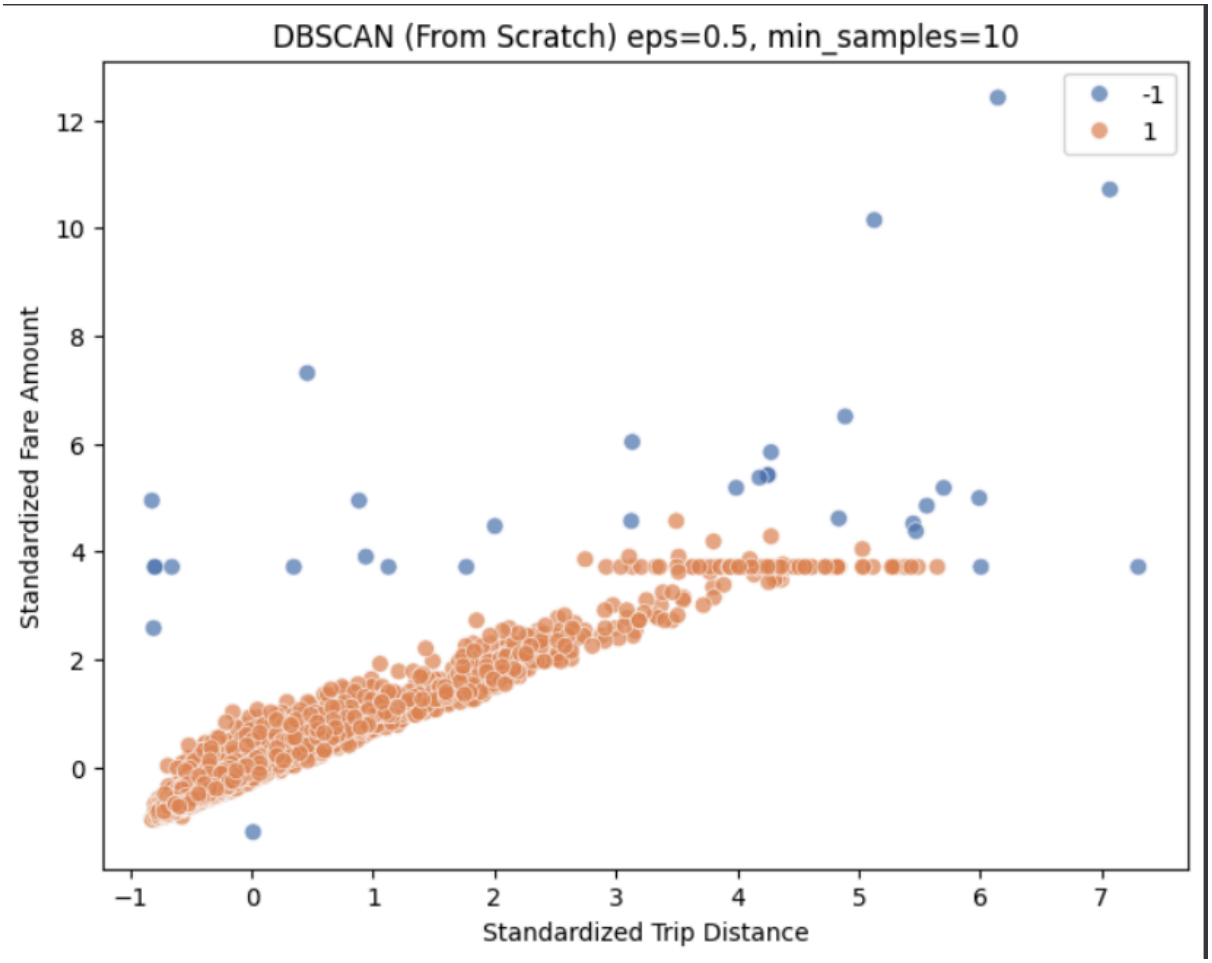
```
dbscan = DBSCAN(eps=0.5, min_samples=10)
labels_db = dbscan.fit_predict(X)
# Plot DBSCAN clusters (noise points are labeled as -1)
plt.figure(figsize=(8, 6))
sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=labels_db, palette='deep', s=50, alpha=0.7)
plt.title("DBSCAN Clustering")
plt.xlabel("Standardized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

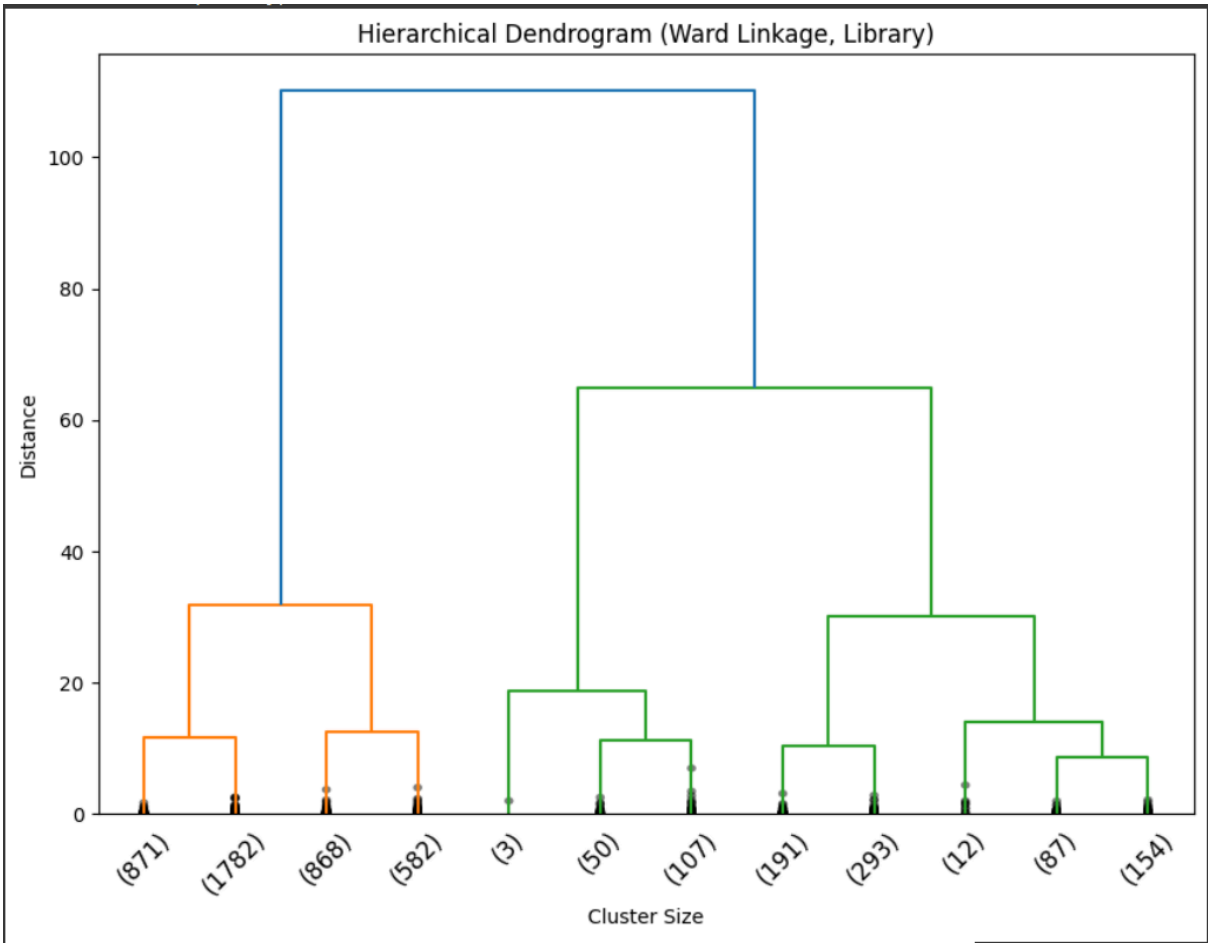


Inference

- **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).

Step 4.1 : Hierarchical Clustering (Ward Linkage)

```
linked = linkage(X, method='ward')
# Plot dendrogram to visualize the hierarchical clustering process
plt.figure(figsize=(10, 7))
dendrogram(linked, truncate_mode='lastp', p=12, leaf_rotation=45.,
leaf_font_size=12., show_contracted=True)
plt.title("Hierarchical Clustering Dendrogram (Ward Linkage)")
plt.xlabel("Cluster Size")
plt.ylabel("Distance")
plt.show()
```

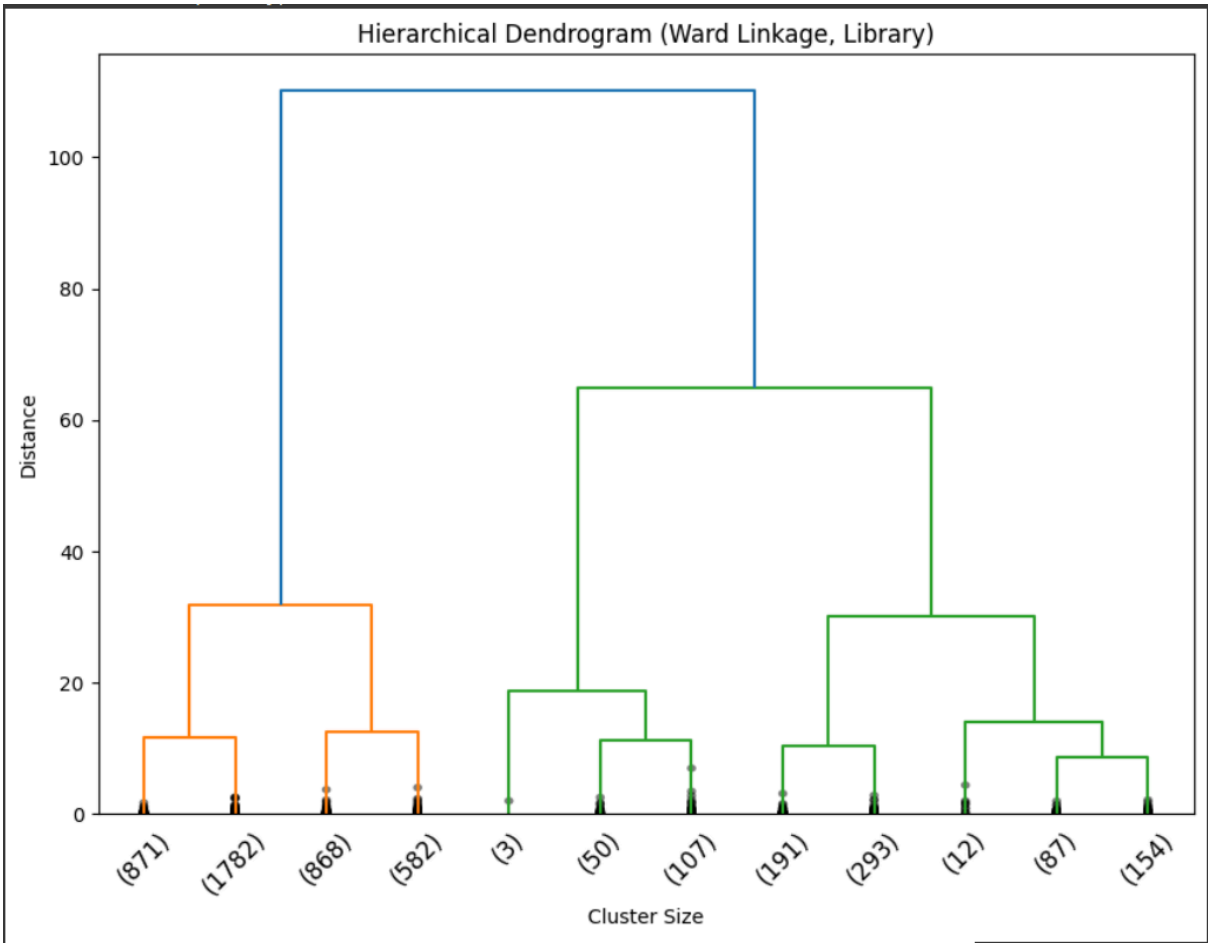


Step 4.1 : Hierarchical Clustering (Ward Linkage)

```
def hierarchical_from_scratch(X, n_clusters=5):
    """
    Simple Agglomerative Clustering from scratch with Ward's method.
    NOTE: This naive approach is O(n^3) for large data.
          It's purely for demonstration.
    """
    print("=== Hierarchical (From Scratch) ===")
    n_samples = X.shape[0]
```

```
# Each sample starts in its own cluster
clusters = [{i} for i in range(n_samples)]

# Helper function: SSE of a cluster
def sse_of_cluster(indices):
    points = X[list(indices)]
    centroid = np.mean(points, axis=0)
    diff = points - centroid
    return np.sum(diff * diff)

# Maintain SSE for each cluster
cluster_sse = [sse_of_cluster(c) for c in clusters]

merges = []  # track merges if needed

# Ward distance = SSE(merged) - SSE(A) - SSE(B)
def ward_distance(idxA, idxB):
    new_cluster = clusters[idxA].union(clusters[idxB])
    return sse_of_cluster(new_cluster) - cluster_sse[idxA] - cluster_sse[idxB]

while len(clusters) > n_clusters:
    min_dist = float('inf')
    pair_to_merge = (None, None)

    for i in range(len(clusters)):
        for j in range(i+1, len(clusters)):
            dist_ij = ward_distance(i, j)
            if dist_ij < min_dist:
                min_dist = dist_ij
                pair_to_merge = (i, j)

    i, j = pair_to_merge
    new_cluster = clusters[i].union(clusters[j])
    merges.append((i, j, min_dist, len(new_cluster)))

    new_sse = sse_of_cluster(new_cluster)
    clusters[i] = new_cluster
    cluster_sse[i] = new_sse

    clusters.pop(j)
    cluster_sse.pop(j)

# Build final labels
labels = np.zeros(n_samples, dtype=int)
for cluster_id, cset in enumerate(clusters):
    for idx in cset:
        labels[idx] = cluster_id

# Plot
plt.figure(figsize=(8, 6))
```

```
sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=labels, palette='Set1', s=50, alpha=0.7)
plt.title(f"Hierarchical (From Scratch) n_clusters={n_clusters}")
plt.xlabel("Standardized Trip Distance")
plt.ylabel("Standardized Fare Amount")
plt.show()
```



Inference

- **Ward Linkage**: Minimizes within-cluster variance at each merge.
- **Major Branches**: Two large branches at a higher distance threshold, suggesting a broad split in the data.
- **Fine-Grained Clusters**: Further down the dendrogram, each branch splits into smaller clusters.

- **Cutting the Dendrogram**: You can “cut” at a certain distance to get your desired number of clusters (e.g., 2, 3, 5, etc.).

- **Interpretation**: Possibly indicates short/medium vs. long/high fare rides as the biggest partition, with subpartitions for more nuanced differences.

Conclusion :

In summary, **K-means** identified five distinct clusters along the distance fare axis, effectively grouping rides into different tiers of length and cost. **DBSCAN**, using the specified parameters, yielded one large cluster plus several outliers, reflecting a dense region of typical rides and highlighting anomalies. **Hierarchical clustering** demonstrated a clear top-level division into two main groups, which further split into smaller clusters when viewed at lower distance thresholds in the dendrogram. Overall, these results confirm a strong positive correlation between trip distance and fare amount. Each method offers a unique perspective: K-means enforces a fixed number of clusters, DBSCAN distinguishes dense regions from noise, and hierarchical clustering provides flexibility in choosing cluster counts by adjusting the dendrogram cut.