```
"""
***********************************************************************************
*******
** COMP572 Data Mining & Visualization
**
** NAME: RAHUL NAWALE
**
** STUDENT ID - 201669264
**
** TASK - CA Assignment 2 - Data Clustering - Implementing Clustering algorithms
**
**
**
***********************************************************************************
*******
       %%%%       %%%%%     %%%        %%%  %%% %%%%     %%%%%%%%%  %%%%%%%%  %%%%%
%
     %%%  %%%   %%%    %%%  %%% %%  %% %%%  %%%     %%%  %%             %%%%
%%%
     %%%        %%%    %%%  %%%   %%   %%%  %%%     %%%  %%%%%          %%%
%%%
     %%%        %%%    %%%  %%%        %%%  %%% %%%%          %%%     %%%      %%%
     %%%  %%%   %%%    %%%  %%%        %%%  %%%               %%%    %%%    %%
      %%%%        %%%%%     %%%        %%%  %%%          %%%%%      %%%     %%%%%%%
%%%
===================================================================================
========
"""
import random
import numpy as np
import matplotlib.pyplot as plt


# Set the random seed for reproducibility
random.seed(1234)
np.random.seed(1234)

# Load the dataset from file
with open("dataset", "r") as f:
    # # Create a list to store the dataset without string values
    # data_without_strings = []
    # # Remove string values from each line of the dataset and append the cleaned
data to the list
    # for line in f:
    #     data_without_strings.append(' '.join(line.split()[1:]))
    # Create a list to store the dataset as a list of float values
    data = []
    # Convert each cleaned data point to a list of floats and append it to the list
    for line in f:
        row = line.strip().split()
        data.append([float(x) for x in row[1:]])
    # Print the first 10 converted data points and a separator line
    # print(data[:10])

# Convert the data to a numpy array for efficient computation
data = np.array(data)
# print(data)

# Define a function to compute the Euclidean distance between two points
```

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


# Question no.4 - KMeans algorithm implementation

# Define a function to initialize the k cluster representatives randomly
def initialize_centroids(k, data):
    # Get the number of data points
    n = data.shape[0]
    # Choose k random indices from the range of data points
    indices = random.sample(range(n), k)
    # Select the data points at the chosen indices as the initial centroids
    centroids = data[indices]
    return centroids

# Define a function to assign each data point to the closest centroid
def assign_clusters(data, centroids):
    # Get the number of data points and the number of centroids
    n = data.shape[0]
    k = centroids.shape[0]
    # Create an array to store the cluster assignments for each data point
    clusters = np.zeros(n)
    # Iterate over all data points
    for i in range(n):
        # Compute the distance between the data point and each centroid
        distances = [euclidean_distance(data[i], centroids[j]) for j in range(k)]
        # Assign the data point to the cluster with the closest centroid
        cluster = np.argmin(distances)
        clusters[i] = cluster
    return clusters

# Define a function to update the centroids based on the mean of the assigned data
points
def update_centroids(data, clusters, k):
    # Get the number of data points, the number of features, and the number of
centroids
    n = data.shape[0]
    d = data.shape[1]
    centroids = np.zeros((k, d))
    # Iterate over all centroids
    for i in range(k):
        # Get the indices of the data points assigned to the current centroid
        indices = np.where(clusters == i)
        # If there are no data points assigned to the centroid, choose a random
data point as the new centroid
        if len(indices[0]) > 0:
            centroids[i] = np.mean(data[indices], axis=0)
        else:
            centroids[i] = data[random.randint(0, n-1)]
    return centroids

# Define a function to compute the Silhouette coefficient
def silhouette_coefficient(data, clusters):
    # Get the number of data points
    n = data.shape[0]
    # Create arrays to
    a = np.zeros(n)
    b = np.zeros(n)
    for i in range(n):
        cluster = int(clusters[i])
        indices = np.where(clusters == cluster)[0]
        if len(indices) == 1:
            a[i] = 0
        else:
            distances = [euclidean_distance(data[i], data[j]) for j in indices if j
!= i]
            a[i] = np.mean(distances)
        other_clusters = list(set(range(k)) - set([cluster]))
        min_distances = []
        for j in other_clusters:
            indices = np.where(clusters == j)[0]
            distances = [euclidean_distance(data[i], data[k]) for k in indices]
            if len(distances) > 0:
                min_distances.append(np.mean(distances))
        if len(min_distances) > 0:
            b[i] = np.min(min_distances)
        else:
            b[i] = 0
    s = (b - a) / np.maximum(a, b)
    return np.mean(s)

# Run the k-means algorithm for k = 2 to 9
silhouette_scores = []
for k in range(2, 10):
    centroids = initialize_centroids(k, data)
    for i in range(100):
        clusters = assign_clusters(data, centroids)
        new_centroids = update_centroids(data, clusters, k)
        if np.allclose(centroids, new_centroids):
            break
        centroids = new_centroids
    silhouette = silhouette_coefficient(data, clusters)
    silhouette_scores.append(silhouette)

# Plot the Silhouette coefficient for each value of k
# plt.plot(range(
plt.plot(range(2, 10), silhouette_scores)
plt.xlabel("Number of clusters (k)")
plt.ylabel("Silhouette coefficient")
plt.suptitle("KMeans")
plt.show()

# Find the maximum Silhouette coefficient and its corresponding k value
# max_silhouette = max(silhouette_scores)
# best_k = silhouette_scores.index(max_silhouette) + 2

#print values of Silhouette coefficients corresponding to K
print("KMeans algorithm:")
for k, score in enumerate(silhouette_scores):
    print(f"Silhouette coefficient for k = {k+2}: {score:.4f}")




# Question no.5 - KMeans++ algorithm implementation

# Define a function to initialize the k cluster representatives using the k-means++
algorithm
def initialize_centroids_plus(k, data):
    # Get the number of data points
    n = data.shape[0]
    # Initialize the first centroid randomly
    centroids = [data[random.randint(0, n-1)]]
    # Compute the remaining centroids using the k-means++ algorithm
    for i in range(1, k):
        # Compute the distances between each data point and the nearest centroid
        distances = [min([euclidean_distance(data[j], c) for c in centroids]) for j
in range(n)]
        # Compute the probability distribution for selecting the next centroid
        probabilities = distances / np.sum(distances)
        # Choose the next centroid randomly based on the probability distribution
        index = np.random.choice(range(n), p=probabilities)
        centroids.append(data[index])
    # Convert the centroids list to a numpy array
    centroids = np.array(centroids)
    return centroids

# Define a function to assign each data point to the closest centroid
def assign_clusters_plus(data, centroids):
    # Get the number of data points and the number of centroids
    n = data.shape[0]
    k = centroids.shape[0]
    # Create an array to store the cluster assignments for each data point
    clusters = np.zeros(n)
    # Iterate over all data points
    for i in range(n):
        # Compute the distance between the data point and each centroid
        distances = [euclidean_distance(data[i], centroids[j]) for j in range(k)]
        # Assign the data point to the cluster with the closest centroid
        cluster = np.argmin(distances)
        clusters[i] = cluster
    return clusters

# Define a function to update the centroids based on the mean of the assigned data
points
def update_centroids_plus(data, clusters, k):
    # Get the number of data points, the number of features, and the number of
centroids
    n = data.shape[0]
    d = data.shape[1]
    centroids = np.zeros((k, d))
    # Iterate over all centroids
    for i in range(k):
        # Get the indices of the data points assigned
        # Get the indices of the data points assigned to the current centroid
        indices = np.where(clusters == i)
        # If there are no data points assigned to the centroid, choose a random
data point as the new centroid
        if len(indices[0]) > 0:
            centroids[i] = np.mean(data[indices], axis=0)
        else:
            centroids[i] = data[random.randint(0, n - 1)]
    return centroids

# Define a function to compute the Silhouette coefficient
def compute_silhouette(data, clusters):
    # Get the number of data points
    n = data.shape[0]
    # Compute the pairwise distances between all data points
    distances = np.zeros((n, n))
    for i in range(n):
        for j in range(i+1, n):
            distances[i][j] = euclidean_distance(data[i], data[j])
            distances[j][i] = distances[i][j]
    # Compute the mean distance to other points in the same cluster and the nearest
other cluster for each data point
    a = np.zeros(n)
    b = np.full(n, np.inf)
    for i in range(n):
        # Get the index of the current cluster
        current_cluster = int(clusters[i])
        # Compute the mean distance to other points in the same cluster
        indices = np.where(clusters == current_cluster)[0]
        if len(indices) > 1:
            a[i] = np.mean(distances[i][indices])
        # Compute the nearest mean distance to points in another cluster
        for j in range(k):
            if j != current_cluster:
                indices = np.where(clusters == j)[0]
                if len(indices) > 0:
                    distance = np.mean(distances[i][indices])
                    if distance < b[i]:
                        b[i] = distance
    # Compute the Silhouette coefficient for each data point
    s = np.zeros(n)
    for i in range(n):
        if a[i] == 0 and b[i] == np.inf:
            s[i] = 0
        else:
            s[i] = (b[i] - a[i]) / max(a[i], b[i])
    # Compute the mean Silhouette coefficient for all data points
    silhouette_coefficient = np.mean(s)
    return silhouette_coefficient

# Initialize variables for storing the Silhouette coefficients and the values of k
silhouette_coefficients = []
ks = range(2, 10)

# Iterate over all values of k
for k in ks:
    # Initialize the centroids using the k-means++ algorithm
    centroids = initialize_centroids_plus(k, data)
    # Assign each data point to the closest centroid
    clusters = assign_clusters_plus(data, centroids)
    # Update the centroids based on the mean of the assigned data points
    centroids = update_centroids_plus(data, clusters, k)
    # Compute the Silhouette coefficient for the current set of clusters
    silhouette_coefficient = compute_silhouette(data, clusters)
    # Append the Silhouette coefficient to the list
    silhouette_coefficients.append(silhouette_coefficient)

# Plot the Silhouette coefficients as a function of k
plt.plot(ks, silhouette_coefficients)
plt.xlabel('k')
plt.ylabel('Silhouette coefficient')
plt.suptitle("KMeans++")
plt.show()

#print values of Silhouette coefficients corresponding to K
print("\n")
print("KMeans++ algorithm:")
for k, score in enumerate(silhouette_coefficients):
    print(f"Silhouette coefficient for k = {k+2}: {score:.4f}")





# Question no.6 - Bisecting KMeans algorithm implementation

def kMeans(X, k):
    n = X.shape[0]  # number of data points
    if n == 0:
        return None, None
    if X.ndim == 1:
        X = X.reshape(-1, 1)
    d = X.shape[1]  # number of features

    # Initialize centroids randomly
    centroids = X[np.random.choice(n, size=k, replace=False), :]
    old_centroids = np.zeros((k, d))

    # Initialize labels
    labels = np.zeros(n)

    # Loop until convergence
    while not np.array_equal(centroids, old_centroids):
        old_centroids = centroids.copy()

        # Compute distances to centroids and assign labels
        for i in range(n):
            distances = np.linalg.norm(X[i] - centroids, axis=1)
            labels[i] = np.argmin(distances)

        # Update centroids
        for j in range(k):
            centroids[j] = np.mean(X[labels == j], axis=0)

    return labels, centroids


def bisectingKMeansClustering(k, dataset):
    # Start with all data points in a single cluster C1
    clusters = [dataset]
    while len(clusters) < k:
        # Choose the largest cluster Cj
        largest_cluster_index = max(range(len(clusters)), key=lambda x: 
np.sum(np.square(clusters[x])))
        largest_cluster = clusters[largest_cluster_index]
        # Apply standard k-means to Cj to obtain two new clusters Cj1 and Cj2
        labels, _ = kMeans(largest_cluster, 2)
        cluster1 = largest_cluster[labels == 0]
        cluster2 = largest_cluster[labels == 1]
        # Replace Cj with Cj1 and Cj2
        del clusters[largest_cluster_index]
        clusters.append(cluster1)
        clusters.append(cluster2)
    # Output the resulting clusters C1,...,Ck
    return clusters

# Compute the Bisecting k-Means clustering with Max clusters =9
hierarchy = [data]
for k in range(2, 10):
    clusters = bisectingKMeansClustering(k, hierarchy[0])
    hierarchy.append(clusters)

silhouette_scores_bis = [0] * 9  # Initialize with zeros
for s in range(1, 10):
    labels, _ = kMeans(hierarchy[s-1][0], 2)
    s_scores = []
    for c in hierarchy[s-1]:
        if len(c) > 1:
            labels, _ = kMeans(c, 2)
            s = np.sum(np.square(c))
            a = np.zeros(len(c))
            b = np.zeros(len(c))
            for j in range(len(c)):
                if labels[j] == 0:
                    a[j] = np.sum(np.square(c[labels == 0] - c[j]))
                    b[j] = np.min(np.sum(np.square(c[labels == 1] - c[j])))
                else:
                    a[j] = np.sum(np.square(c[labels == 1] - c[j]))
                    b[j] = np.min(np.sum(np.square(c[labels == 0] - c[j])))
            score = np.mean((b - a) / np.maximum(a, b))
            s_scores.append(score)
        else:
            s_scores.append(0)
    silhouette_scores_bis.append(np.mean(s_scores))

del silhouette_scores_bis[:9]
# Plot the Silhouette coefficients for each clustering
plt.plot(range(1, 10), silhouette_scores_bis, 'o-')
plt.xlabel('s')
plt.ylabel('Silhouette coefficient')
plt.suptitle("Bisecting KMeans")
plt.show()

#print values of Silhouette coefficients corresponding to each cluster s
print("\n")
print("Bisecting KMeans algorithm:")
for k, score in enumerate(silhouette_scores_bis):
    print(f"Silhouette coefficient for k = {k+2}: {score:.4f}")
```