Project 2. Machine Learning. Reyhane Esmailizadeh. 810800004

1.

A) In this part we have:

Importing Libraries:

numpy (as np): For numerical operations.
pandas (as pd): For data manipulation and analysis.
matplotlib.pyplot (as plt): For data visualization.
rbf_kernel from sklearn.metrics.pairwise: For calculating the radial basis function kernel.
StandardScaler from sklearn.preprocessing: For standardizing features.

Utility Functions:

euclidean_distance: Computes the Euclidean distance between two points.
initialize_centroids: Initializes cluster centroids randomly from the dataset.
assign_clusters: Assigns each data point to its nearest centroid.
update_centroids: Updates the centroids based on the mean of the assigned data points for each cluster.
kmeans: Performs the K-means clustering algorithm.

Loading Dataset:

Reads a CSV file named "driver-data.csv" containing the dataset. You need to adjust the file path according to your file location.
Extracts the features 'mean_dist_day' and 'mean_over_speed_perc' from the dataset.
Data Standardization:

Standardizes the features using Z-score normalization.

Applying KMeans Clustering:

Defines the number of clusters k as 4.
Calls the kmeans function with the standardized features and k.
Obtains the clusters and centroids.

Data Visualization:

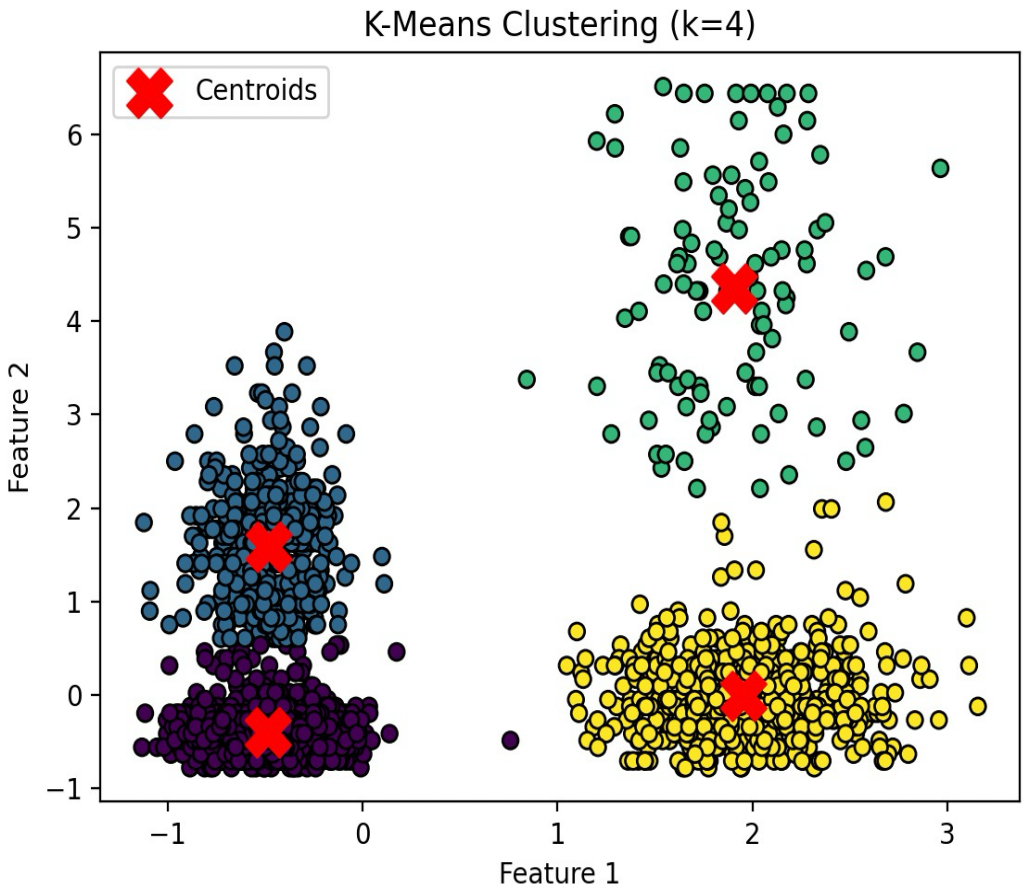
Plots the clusters using plt.scatter.
Centroids are plotted separately using a different marker and color.
Adds labels and a legend for better understanding.

The code:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics.pairwise import rbf_kernel
from sklearn.preprocessing import StandardScaler
#part a

#Function to calculate Euclidean distance between two points
def euclidean_distance(point1, point2):
    return np.sqrt(np.sum((point1 - point2)**2))

# Function to initialize centroids randomly
def initialize_centroids(data, k):
    indices = np.random.choice(len(data), k, replace=False)
    return data[indices]

# Function to assign data points to clusters
def assign_clusters(data, centroids):
    distances = np.array([np.array([euclidean_distance(point, centroid) for centroid
in centroids]) for point in data])
    return np.argmin(distances, axis=1)

# Function to update centroids based on mean of assigned data points
def update_centroids(data, clusters, k):
    new_centroids = np.zeros((k, data.shape[1]))
    for i in range(k):
        cluster_points = data[clusters == i]
        if len(cluster_points) > 0:
            new_centroids[i] = np.mean(cluster_points, axis=0)
        else:
            new_centroids[i] = data[np.random.choice(len(data))]
    return new_centroids

# Function to perform k-means clustering
def kmeans(data, k, max_iterations=100):
    centroids = initialize_centroids(data, k)
    for _ in range(max_iterations):
        clusters = assign_clusters(data, centroids)
        new_centroids = update_centroids(data, clusters, k)
        if np.array_equal(centroids, new_centroids):
            break
        centroids = new_centroids
    return clusters, centroids

# Load the dataset
data = pd.read_csv('/Users/reyhane/Downloads/driver-data.csv')

# Assuming your dataset has features like 'feature1', 'feature2', etc.
# Adjust the feature names according to your actual dataset
features = data[['mean_dist_day', 'mean_over_speed_perc']].values

# Standardize the features (optional, but can be helpful)
features = (features - features.mean(axis=0)) / features.std(axis=0)
```

```
# Apply KMeans clustering
k = 4
clusters, centroids = kmeans(features, k)

# Plot the clusters
plt.scatter(features[:, 0], features[:, 1], c=clusters, cmap='viridis', marker='o',
edgecolor='black')
plt.scatter(centroids[:, 0], centroids[:, 1], s=300, c='red', marker='X',
label='Centroids')
plt.title('K-Means Clustering (k=4)')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.show()
```

The result is :



1.

B) In this part:

Defining Gaussian Kernel Function (gaussian_kernel):

This function computes the Gaussian kernel matrix for the given data.
It uses the Radial Basis Function (RBF) kernel (rbf_kernel) from scikit-learn with a specified gamma parameter.
The gamma parameter controls the width of the Gaussian distribution.
Standardizing Features:

The features are standardized using StandardScaler() from scikit-learn.
Standardization ensures that each feature has a mean of 0 and a standard deviation of 1, which can be helpful for certain algorithms.

Applying Gaussian Kernel to Standardized Features:

The Gaussian kernel is applied to the standardized feature space.
This step transforms the original feature space into a higher-dimensional space where non-linear relationships between features are captured more effectively.
Applying KMeans Clustering on the Kernel Feature Space:

K-means clustering is applied to the feature space obtained from the Gaussian kernel.
The number of clusters (k) is set to 4.

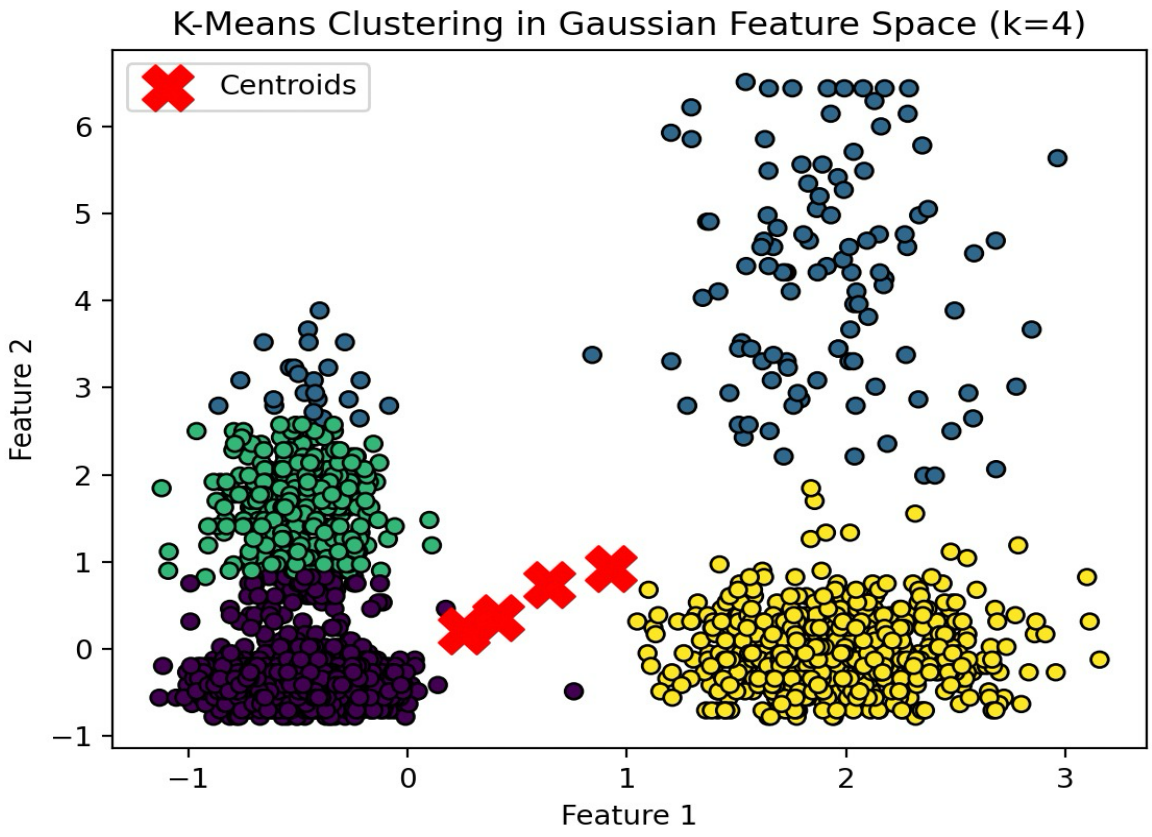
Data Visualization:

The clusters in the original feature space are plotted using plt.scatter.
Cluster centroids are plotted separately using a different marker and color.
Labels and a legend are added for better interpretation of the plot.

The code:

```
def gaussian_kernel(data, gamma=1):
    return rbf_kernel(data, gamma=gamma)

# Standardize the features (optional, but can be helpful)
features_standardized = StandardScaler().fit_transform(features)

# Apply Gaussian kernel to the standardized features
gamma = 0.15  # You can adjust the gamma parameter based on your data
feature_space = gaussian_kernel(features_standardized, gamma)

# Apply KMeans clustering on the feature space
k = 4
kernel_clusters, kernel_centroids = kmeans(feature_space, k)

# Plot the clusters in the original feature space
plt.scatter(features_standardized[:, 0], features_standardized[:, 1],
c=kernel_clusters, cmap='viridis', marker='o', edgecolor='black')
plt.scatter(kernel_centroids[:, 0], kernel_centroids[:, 1], s=300, c='red',
marker='X', label='Centroids')
plt.title('K-Means Clustering in Gaussian Feature Space (k=4)')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.show()
```

The result is:



1.

C) In this part we have:

Data Preprocessing:

The code starts by normalizing the input data using MinMaxScaler from scikit-learn.
Normalization scales each feature to a range between 0 and 1.
Initialization of Parameters:

The function initialize_parameters initializes the parameters needed for the EM algorithm:
variances: Variance matrices for each cluster, initialized as identity matrices.
average: Initial cluster centroids, randomly initialized.
cluster_prob: Initial probabilities of each cluster, initialized uniformly.

Expectation Step:

In the expectation_step function, the algorithm computes the posterior probabilities of each data point belonging to each cluster.
It calculates the probability density function (PDF) for each data point using a multivariate normal distribution.

Maximization Step:

In the Maximization_step function, the algorithm updates the parameters based on the posterior probabilities calculated in the expectation step.
It computes new cluster centroids (newaverage), variances (newvariance), and cluster probabilities (newprob_cluster).

Convergence Check:

The algorithm iterates between the expectation and maximization steps until a convergence criterion is met.
The convergence criterion in this code is based on the change in the centroids. If the change falls below a specified threshold (eps), the algorithm terminates.

Iteration Count:

The variable count keeps track of the number of iterations required for convergence.

The code is:

```
#part c
from scipy.stats import multivariate_normal
import numpy as np



# Extract only the desired features (columns [1] and [2])
mydataa = data.iloc[:, [1, 2]].values
print(mydataa)
# Assuming 'data' is your input data
from sklearn.preprocessing import MinMaxScaler

# Create MinMaxScaler
scaler = MinMaxScaler()

# Fit and transform the data
mydata = scaler.fit_transform(mydataa)

# Print the normalized data
print(mydata)
myk = 4
eps = 0.001
n_features = mydata.shape[1]
n_samples = mydata.shape[0]
print(n_features)
print(n_samples)
def initialize_parameters(data, k):
    n_features = data.shape[1]
    cluster_prob = np.ones(k) / k
    # Initialize variances as identity matrices
    variances = np.tile(np.eye(n_features), (k, 1, 1))
    average = np.random.randn(k, n_features)
    return variances, average, cluster_prob


def sumofdistances(averageold, averagecurrent, k):
    sum = 0
    for i in range(k):
        distance = np.linalg.norm(averagecurrent[i] - averageold[i])
        #print(distance)
        sum += distance
    return sum

def cal_prob_data(datapoint, variance, average, cluster_prob, k):
    total_sum = 0
    for i in range(k):
        multivariate_dist = multivariate_normal(mean=average[i], cov=variance[i])
        pdf_value = multivariate_dist.pdf(datapoint)
        total_sum += pdf_value * cluster_prob[i]
    return total_sum

def expectation_step(data, k, n_samples, average, cluster_prob, variance):
    post_prob = np.zeros((k, n_samples))
    for i in range(k):
        # Create a multivariate normal distribution
```

```
        multivariate_dist = multivariate_normal(mean=average[i], cov=variance[i])
        print("average for cur_cluster:",average[i])
        for j in range(n_samples):
            # Extract the data point for the j-th sample
            datapoint = data[j, :]  # Assuming data is a 2D array
            #print("datapoint:", datapoint)
            # Calculate the probability density function (PDF) for the data point
            pdf_value = multivariate_dist.pdf(datapoint)
            #print(f"PDF for cur_datapoint: {pdf_value}")
            post_prob[i][j] = (pdf_value * cluster_prob[i]) / cal_prob_data(datapoint,
variance, average, cluster_prob, k)
    return post_prob

def Maximization_step(data, post_prob, n_samples,k):
    newvariance = np.zeros_like(old_variance)
    newaverage = np.zeros_like(old_average)
    newprob_cluster = np.zeros_like(old_cluster_prob)
    for i in range(k):
        sum1 = np.sum(post_prob[i][:, np.newaxis] * data, axis=0)
        sum2 = np.sum(post_prob[i])
        newaverage[i] = sum1 / sum2

    for i in range(k):
        sum2 = np.sum(post_prob[i])
        sum3 = np.sum(post_prob[i] * np.linalg.norm(data - newaverage[i], axis=1))
        newvariance[i] = sum3 / sum2
        newprob_cluster[i] = sum2 / n_samples

    print("new_prob_cluster:",newprob_cluster)
    return newvariance, newaverage, newprob_cluster

old_variance, old_average, old_cluster_prob = initialize_parameters(mydata, myk)
print("old_variance :", old_variance)
print("old_average:", old_average)
print("old_cluster_prob:", old_cluster_prob)
post_prob = expectation_step(mydata, myk, n_samples, old_average, old_cluster_prob,
old_variance)
print("post_prob:",post_prob)
cur_variance, cur_average, cur_cluster_prob = Maximization_step(mydata, post_prob,
n_samples,myk)
print("cur_variance:",cur_variance)

sumofdist = sumofdistances(old_average, cur_average, myk)
count = 0
print("cur_average:",cur_average)
print("sumofdist:",sumofdist)
print("cur_cluster_prob:", cur_cluster_prob)
while sumofdist >= eps:
    count = count + 1
    old_variance, old_average, old_cluster_prob = cur_variance, cur_average,
cur_cluster_prob
    post_prob = expectation_step(mydata, myk, n_samples, old_average,
old_cluster_prob, old_variance)
    cur_variance, cur_average, cur_cluster_prob = Maximization_step(old_average,
mydata, post_prob, n_samples,myk)
    sumofdist = sumofdistances(old_average, cur_average, myk)
    print(count)
```

1.

D) In this part we have:

Euclidean Distance Calculation:
The function euclidean_distance calculates the Euclidean distance between two points.

Betacv Calculation:

The function betacv computes the betacv index for a given clustering result.
It takes three parameters:
features: The feature matrix of the data.
clusters: The cluster assignments for each data point.
centroids: The centroids of each cluster.

Within this function:
For each cluster, it calculates the average distance of each point in the cluster to its centroid.
Then, it computes the betacv index as the average of the maximum similarity between clusters. The similarity between two clusters is defined as the ratio of the sum of their average intra-cluster distances to their inter-cluster distance (Euclidean distance between centroids).

Betacv Calculation for KMeans:

The betacv index for KMeans clustering is computed using the betacv function with the parameters features, clusters (the cluster assignments from KMeans), and centroids (the centroids of the clusters).

Betacv Calculation for Kernel KMeans:

Assuming feature_space contains the data transformed by a Gaussian kernel, the betacv index for Kernel KMeans is computed similarly using the betacv function with feature_space, kernel_clusters (the cluster assignments from Kernel KMeans), and kernel_centroids (the centroids of the kernel clusters).

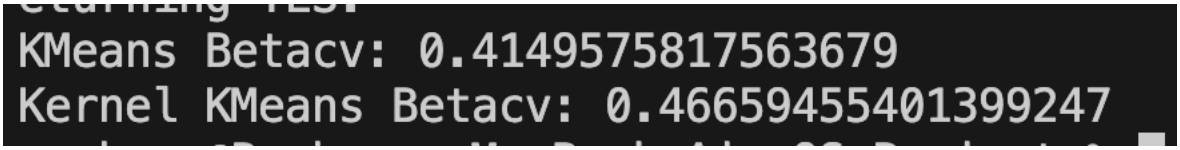
Printing Betacv Values:

Finally, the betacv values for both KMeans and Kernel KMeans are printed.

The code:

```
#q1 part d

from sklearn.metrics.pairwise import euclidean_distances

# Function to calculate Euclidean distance between two points
def euclidean_distance(point1, point2):
    return np.sqrt(np.sum((point1 - point2) ** 2))

# Function to compute the betacv index
def betacv(features, clusters, centroids):
    k = len(centroids)
    cluster_distances = np.zeros(k)
    for i in range(k):
        cluster_points = features[clusters == i]
        cluster_center = centroids[i]
        intra_cluster_distances = euclidean_distances(cluster_points,
[cluster_center])
        avg_intra_cluster_distance = np.mean(intra_cluster_distances)
        cluster_distances[i] = avg_intra_cluster_distance

    betacv_index = 0
    for i in range(k):
        max_similarity = 0
        for j in range(k):
            if i != j:
                similarity = (cluster_distances[i] + cluster_distances[j]) /
euclidean_distance(centroids[i], centroids[j])
                max_similarity = max(max_similarity, similarity)

        betacv_index += max_similarity

    betacv_index /= k
    return betacv_index


# Calculate betacv for KMeans
kmeans_betacv = betacv(features, clusters, centroids)
print(f'KMeans Betacv: {kmeans_betacv}')

#betacv for kernel kmeans

# Assuming 'feature_space' is the data after applying the Gaussian kernel
kernel_kmeans_betacv = betacv(feature_space, kernel_clusters, kernel_centroids)
print(f'Kernel KMeans Betacv: {kernel_kmeans_betacv}')
```

The result is:



As is obvious, Kernel Kmeans Betacv is higher which indicates that the Kenrel Kmeans has better results in clustering these data points.

2.

A)

Functions:

euclidean_distance(point1, point2): Computes the Euclidean distance between two points using NumPy's linalg.norm function.
find_neighbors(data, target_point, eps): Finds all points in the dataset within a given epsilon distance (eps) from a target point.
dbscan(data, eps, min_samples): Implements the DBSCAN algorithm. It assigns cluster labels to each point in the dataset based on epsilon (eps) and minimum number of samples (min_samples) parameters.

DBSCAN Algorithm Steps:

Initialize labels for all points as -1 (representing noise).
Iterate over each point in the dataset:
If the point is not visited yet:
Find its neighbors within the epsilon distance.
If the number of neighbors is less than min_samples, mark the point as noise (-1).
Otherwise, assign a new cluster ID to the point and expand the cluster by recursively adding reachable points.
Return the cluster labels.

Usage:

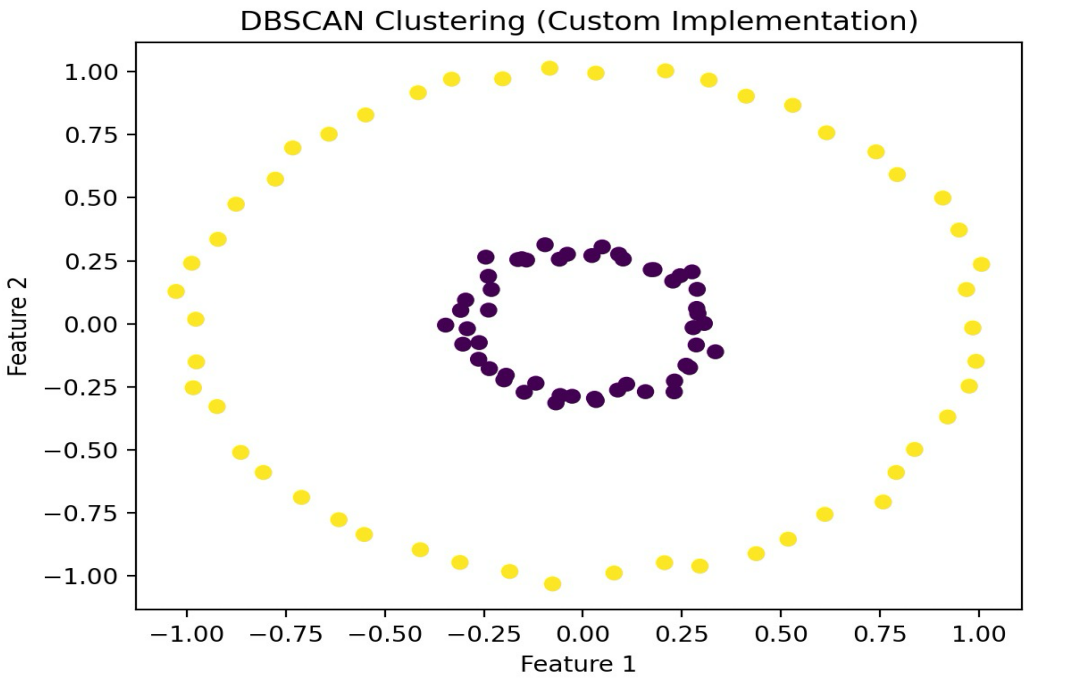
It generates a dataset of two circles using make_circles from sklearn.datasets.
Applies the custom DBSCAN implementation (dbscan) to the generated dataset.
Plots the clustered data points using Matplotlib.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_circles
def euclidean_distance(point1, point2):
    return np.linalg.norm(point1 - point2)

def find_neighbors(data, target_point, eps):
    neighbors = []
    for i, point in enumerate(data):
        if euclidean_distance(target_point, point) < eps:
            neighbors.append(i)
    return neighbors

def dbscan(data, eps, min_samples):
    labels = np.full(len(data), -1)  # -1 represents noise
    cluster_id = 0

    for i, point in enumerate(data):
        if labels[i] != -1:  # Skip if already visited
            continue
```

```
        neighbors = find_neighbors(data, point, eps)

        if len(neighbors) < min_samples:
            labels[i] = -1  # Mark as noise
        else:
            cluster_id += 1
            labels[i] = cluster_id

            for neighbor in neighbors:
                if labels[neighbor] == -1:
                    labels[neighbor] = cluster_id
                    new_neighbors = find_neighbors(data, data[neighbor], eps)
                    if len(new_neighbors) >= min_samples:
                        neighbors.extend(new_neighbors)

    return labels

# Generate a dataset with two circles
n_samples = 100
circles_data, _ = make_circles(n_samples=n_samples, factor=0.3, noise=0.02,
random_state=42)

# Apply our DBSCAN implementation
eps = 0.3
min_samples = 5
dbscan_labels = dbscan(circles_data, eps, min_samples)

# Plot the results
plt.scatter(circles_data[:, 0], circles_data[:, 1], c=dbscan_labels, cmap='viridis')
plt.title('DBSCAN Clustering (Custom Implementation)')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```

The result is:



2.

B)

Functions:

initialize_centers(data, k): Randomly initializes k cluster centers from the data points.
assign_to_clusters(data, centers): Assigns each data point to the nearest cluster center.
update_centers(data, labels, k): Updates the cluster centers based on the mean of the points assigned to each cluster.
kmeans(data, k, max_iterations, tol): Implements the K-means algorithm. It iteratively assigns points to clusters and updates cluster centers until convergence.

K-means Algorithm Steps:

Initialize k cluster centers randomly.
Repeat until convergence (or a maximum number of iterations):
Assign each point to the nearest cluster center.
Update each cluster center to the mean of the points assigned to it.
Check for convergence based on a tolerance threshold.

Usage:

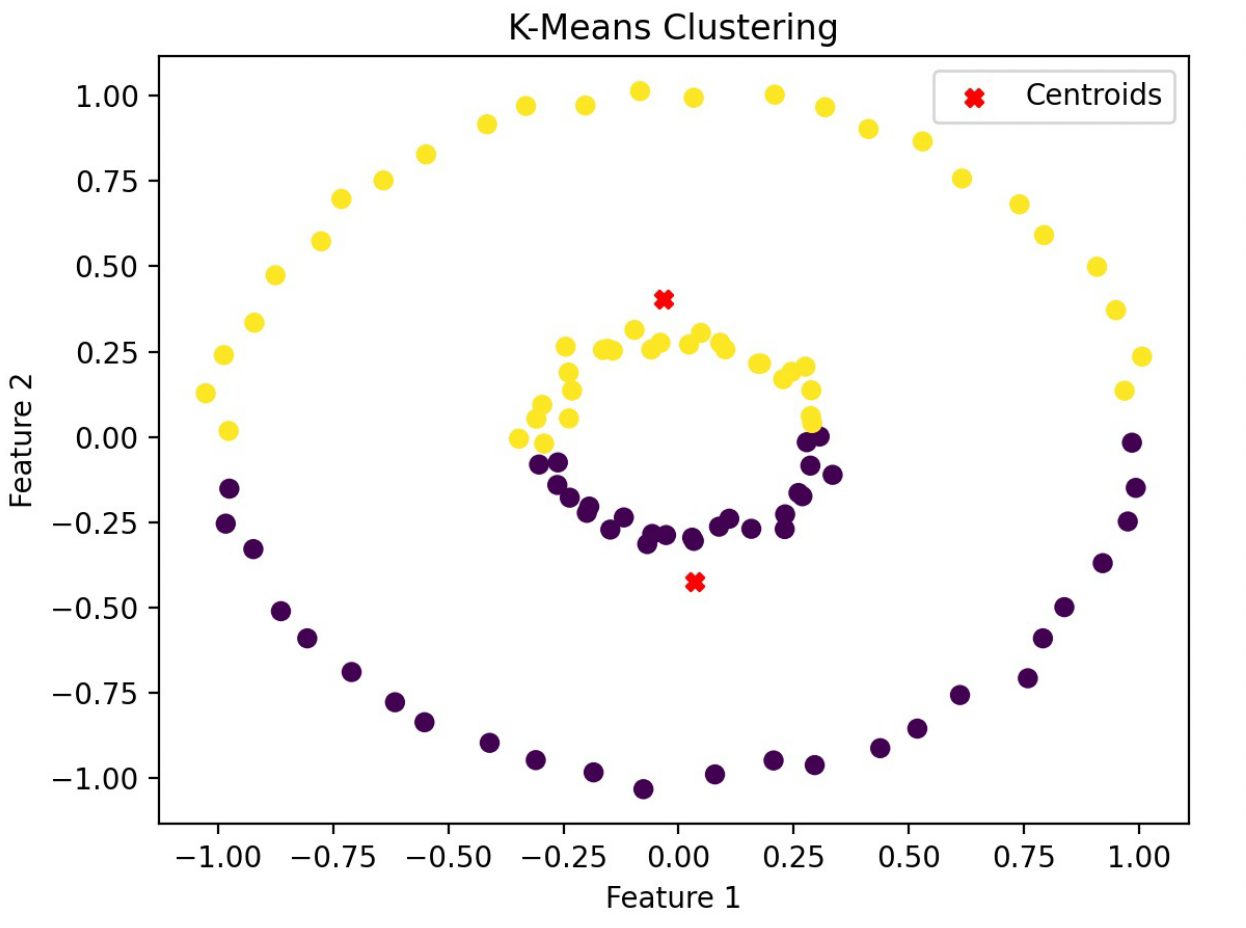
It applies the custom K-means implementation (kmeans) to the same dataset of two circles.
Plots the clustered data points and centroids using Matplotlib.

Code:

```
import numpy as np
import matplotlib.pyplot as plt

def initialize_centers(data, k):
    indices = np.random.choice(len(data), k, replace=False)
    return data[indices]

def assign_to_clusters(data, centers):
    distances = np.linalg.norm(data[:, np.newaxis, :] - centers, axis=2)
    return np.argmin(distances, axis=1)

def update_centers(data, labels, k):
    new_centers = np.zeros((k, data.shape[1]))
    for i in range(k):
        cluster_points = data[labels == i]
        if len(cluster_points) > 0:
            new_centers[i] = np.mean(cluster_points, axis=0)
    return new_centers

def kmeans(data, k, max_iterations=100, tol=1e-6):
    centers = initialize_centers(data, k)

    for _ in range(max_iterations):
        # Assign points to clusters
        labels = assign_to_clusters(data, centers)

        # Update cluster centers
        new_centers = update_centers(data, labels, k)
```

```
        # Check for convergence
        if np.all(np.abs(new_centers - centers) < tol):
            break

        centers = new_centers

    return labels, centers

# Apply k-means clustering
k = 2
kmeans_labels, kmeans_centers = kmeans(circles_data, k)

# Plot the results
plt.scatter(circles_data[:, 0], circles_data[:, 1], c=kmeans_labels, cmap='viridis')
plt.scatter(kmeans_centers[:, 0], kmeans_centers[:, 1], c='red', marker='X',
label='Centroids')
plt.title('K-Means Clustering')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.show()
```

The result is:



DBSCAN might be considered more suitable because it can handle non-convex shapes and automatically find clusters based on density. The synthetic dataset generated using make_circles has a circular structure, and DBSCAN is able to capture this structure without assuming a specific cluster shape.

3.

Kernel Functions:

I(x): Indicator function that returns 1 if the absolute value of x is less than or equal to 1, and 0 otherwise.
kernel1(x): Defines a kernel function as half of the indicator function.
kernel2(x): Defines a kernel function using the Gaussian kernel (normal distribution) formula.
kernel3(x): Defines a kernel function based on the Epanechnikov kernel formula, which is a quadratic function within the range [-1, 1].

Estimated Density Calculation Function:

estimateddensity(data, kernelfunction, datapoint, n, h): Computes the estimated density of a given datapoint using the specified kernelfunction with bandwidth h on the dataset data containing n samples.
It iterates over each data point in the dataset, computes the kernel function value for the difference between the datapoint and the current data point scaled by h, and accumulates the sum.
The estimated density is calculated as the sum divided by the product of the number of samples n and the bandwidth h.

Density Estimation:

The code reads a dataset datanerve from a CSV file.
It selects a specific datapoint from the dataset.
It iterates over different bandwidth values (h ranging from 0.1 to 0.5 with a step of 0.1).
For each bandwidth value, it computes the estimated density of the datapoint using three different kernel functions (kernel1, kernel2, kernel3) and prints the results.

The code:

```
def I(x):
    if abs(x) <= 1:
        result = 1
    else:
        result = 0
    return result

def kernel1(x):
    result = 0.5 * I(x)
    return result

def kernel2(x):
    result = (1 / np.sqrt(2 * np.pi)) * np.exp((-1 / (x**2)))
    return result

def kernel3(x):
    result = (3/4) * (1 - x**2) * I(x)
    return result

def estimateddensity(data,kernelfunction,datapoint,n,h):
```

```
    sum = 0
    for i in range(n):
        curdata = data.iloc[i,1]
        z = (datapoint - curdata)/h
        sum = sum + kernelfunction(z)
    result = (1/(n * h)) * sum
    return result


datanerve = pd.read_csv('/Users/reyhane/Downloads/nerve.csv')
print(datanerve)
datapoint = datanerve.iloc[1,1]
print( " datapoint:",datapoint)
n_samples3 = datanerve.shape[0]

for h in np.arange(0.1,0.6,0.1):
    print("current h ",h)
    estimatedk1 = estimateddensity(datanerve,kernel1,datapoint,n_samples3,h)
    estimatedk2 = estimateddensity(datanerve,kernel2,datapoint,n_samples3,h)
    estimatedk3 = estimateddensity(datanerve,kernel3,datapoint,n_samples3,h)
    print("estimated density kernel 1", estimatedk1)
    print("estimated density kernel 2", estimatedk2)
    print("estimated density kernel 3", estimatedk3)
```

The result:

```
current h  0.1
/Users/reyhane/MLp2.py:371: RuntimeWarning: divi
  result = (1 / np.sqrt(2 * np.pi)) * np.exp((-1
estimated density kernel 1 2.2590738423028784
estimated density kernel 2 1.851690153684504
estimated density kernel 3 2.567459324155197
current h  0.2
estimated density kernel 1 1.6520650813516895
estimated density kernel 2 0.552328377219351
estimated density kernel 3 1.936178035043804
current h  0.30000000000000004
estimated density kernel 1 1.2995410930329576
estimated density kernel 2 0.23480396041556106
estimated density kernel 3 1.5826032540675865
current h  0.4
estimated density kernel 1 1.0700876095118899
estimated density kernel 2 0.11603207933622224
estimated density kernel 3 1.3405032658010017
current h  0.5
estimated density kernel 1 0.9198998748435545
estimated density kernel 2 0.062423737960049616
estimated density kernel 3 1.1654665832290383
```

When the bandwidth (h) is small, each data point has a narrow influence range, resulting in a more localized density estimate. As h increases, the influence range of each data point widens, causing more distant points to contribute to the density estimation. This broader influence leads to a smoothing effect on the density estimate.

However, as the bandwidth becomes too large, distant data points start to influence the density estimate significantly, potentially diluting the density in regions where the data is sparse. This can cause the estimated density to decrease because the contribution from distant points outweighs the contribution from nearby points, resulting in an overly generalized estimation of density.