All Exercises

Roie Kazoom 207376187

Nir Schneider 316098052

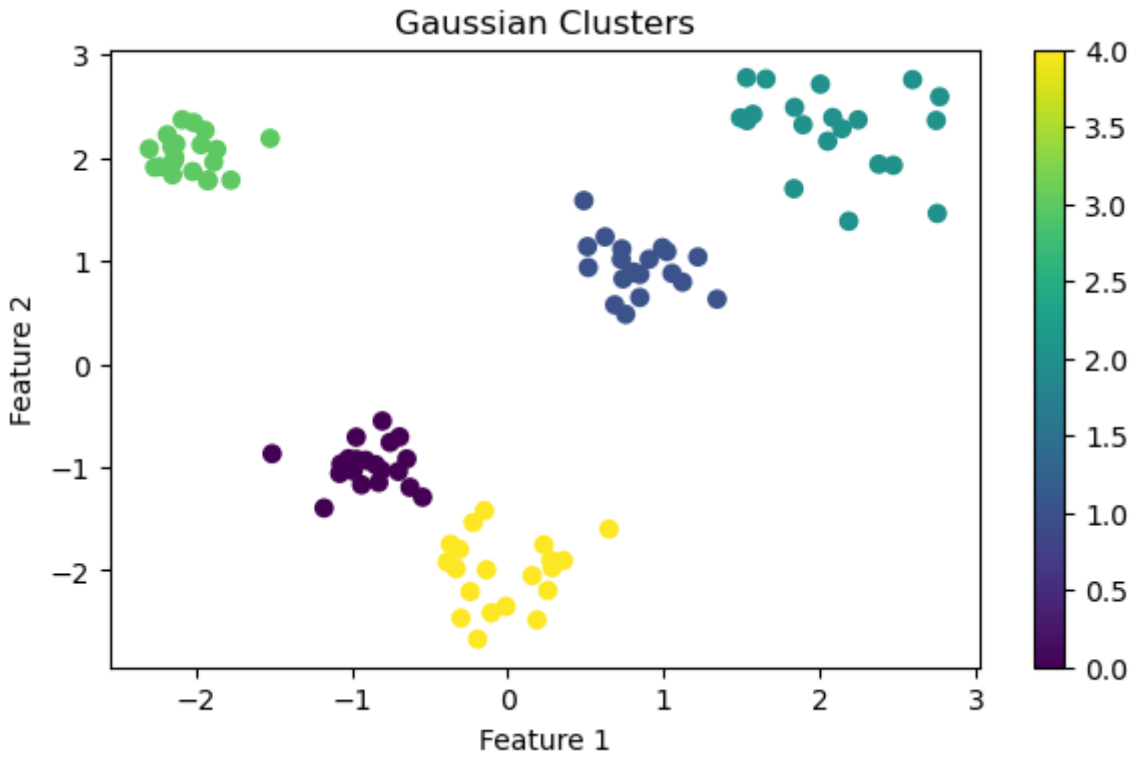
Exercise 0

In [1]:
```
import numpy as np
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
from sklearn.mixture import GaussianMixture
from sklearn.metrics import silhouette_score
from sklearn.datasets import make_moons
from sklearn.metrics.pairwise import euclidean_distances, manhattan_distances
from scipy.spatial.distance import cdist
from sklearn.cluster import KMeans
from sklearn.mixture import BayesianGaussianMixture
from scipy.cluster.hierarchy import linkage, dendrogram, fcluster
```

we can use the make_blobs function from the sklearn.datasets module in Python. This function allows us to generate random blobs with specified centers, standard deviations, and number of samples.

In [2]:

```
# Set the random seed for reproducibility
np.random.seed(0)

# Define the parameters for the clusters
centers = [[-1, -1], [1, 1], [2, 2], [-2, 2], [0, -2]]
covariances = [0.2, 0.3, 0.4, 0.2, 0.3]
n_samples = 100

# Generate the dataset
X, y = make_blobs(n_samples=n_samples, centers=centers, cluster_std=covariances)

# Plot the groups
plt.figure(figsize=(7, 4))
plt.scatter(X[:, 0], X[:, 1], c=y, cmap='viridis')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Gaussian Clusters')
plt.colorbar()
plt.show()
```



In this example, we set centers as a list of the cluster centers, covariances as a list of the standard deviations for each cluster, and n_samples as the total number of samples to generate. The resulting dataset is stored in the variable X, and the corresponding labels for each sample are stored in the variable y.

You can modify the centers, covariances, and n_samples variables to create different configurations for your dataset.

In this code, we use plt.scatter to create a scatter plot of the data points in X. The color of each point is determined by the corresponding label y, and we specify the colormap viridis to map the different labels to distinct colors. You can customize the labels and colormap as needed.

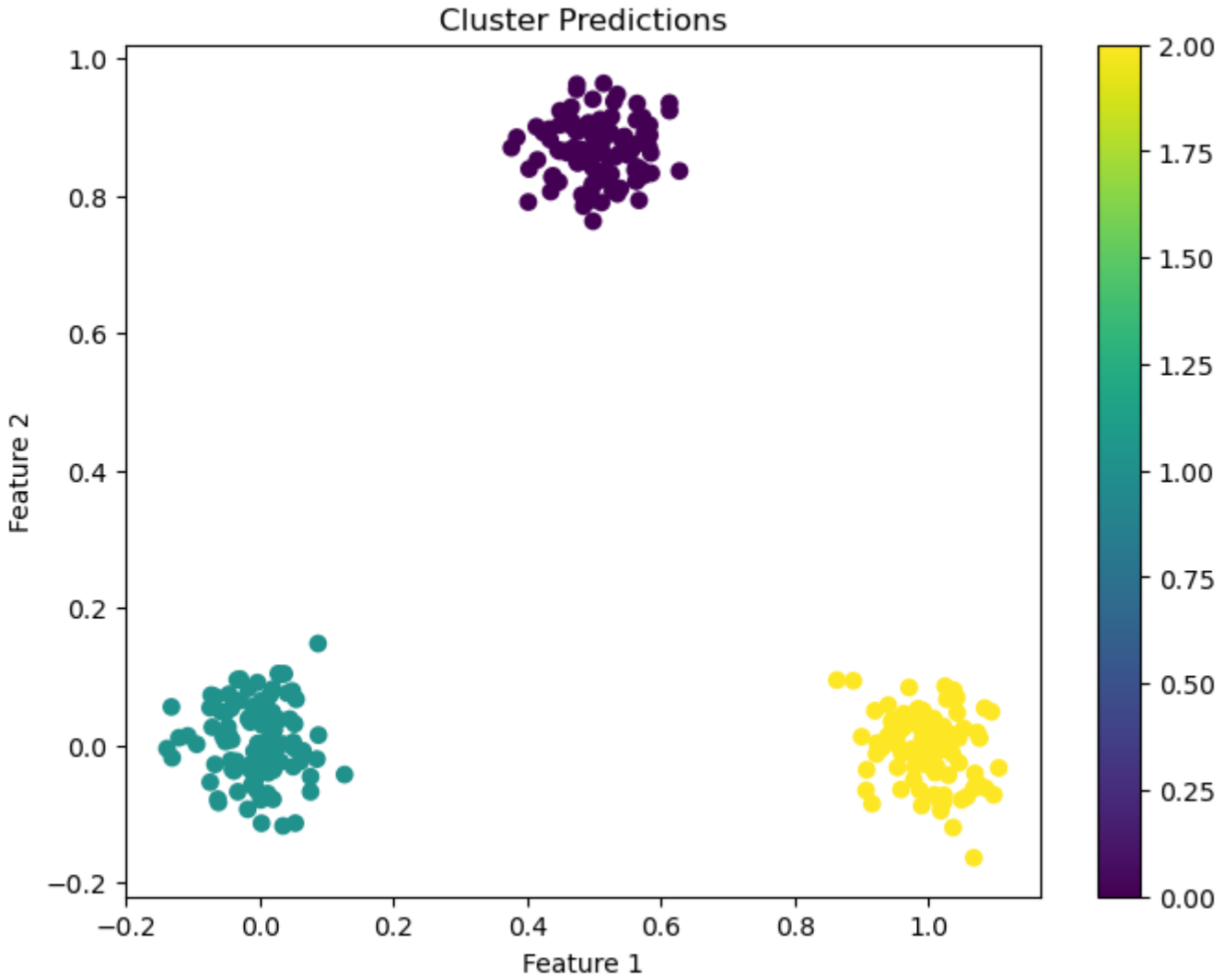
The resulting plot will display the clusters based on the two features (dimensions) in your dataset. Make sure to adjust the code if you have more than two dimensions or if you want to visualize a different set of features.

In [3]:

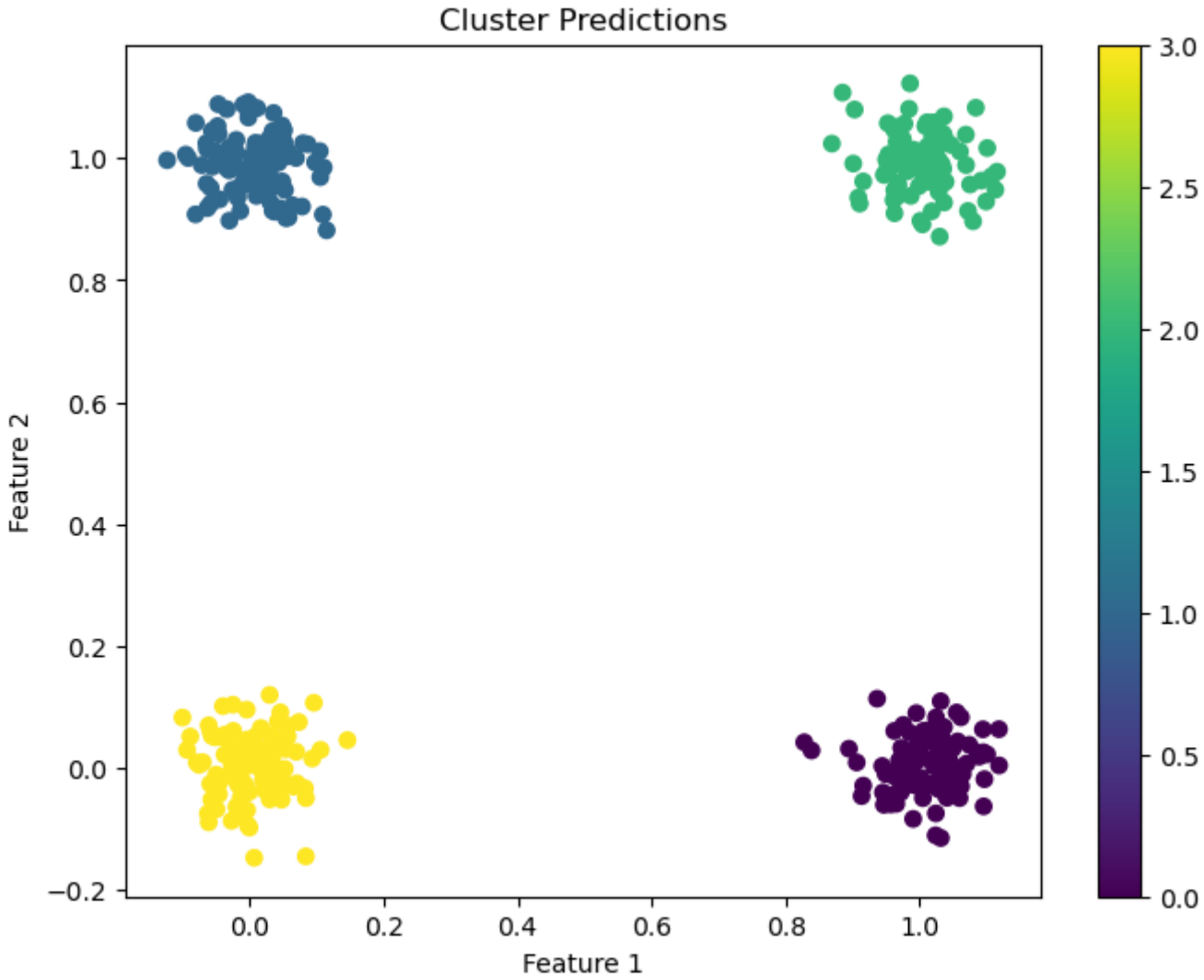
```
# Fit the MLE algorithm on the clusters
def fit_mle(X, n_components):
    # Create an instance of GaussianMixture
    model = GaussianMixture(n_components=n_components)

    # Fit the model to the data
    model.fit(X)

    return model

# Evaluate the MLE algorithm on a dataset
def evaluate_model(model, X):
    # Get the predicted cluster labels
    y_pred = model.predict(X)

    # Calculate the log-likelihood score
    log_likelihood = model.score(X)

    # Calculate the silhouette score
    silhouette = silhouette_score(X, y_pred)

    # Print the evaluation metrics
    print("Log-Likelihood Score:", log_likelihood)
    print("Silhouette Score:", silhouette)

    # Plot the clusters
    plt.figure(figsize=(8, 6))
    plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap='viridis')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.title('Cluster Predictions')
    plt.colorbar()
    plt.show()

# Generate the dataset with triangle shape
centers_triangle = [[0, 0], [1, 0], [0.5, 0.866]]
covariances_triangle = [0.05, 0.05, 0.05]
n_samples_triangle = 300

X_triangle, y_triangle = make_blobs(n_samples=n_samples_triangle, centers=centers_triangle, cluster_std=covariances_triangle)

# Fit the MLE algorithm on the triangle-shaped clusters
model_triangle = fit_mle(X_triangle, n_components=3)

# Evaluate the MLE algorithm on the triangle-shaped clusters
evaluate_model(model_triangle, X_triangle)
```

```
# Generate the dataset with X shape
centers_x = [[0, 0], [1, 1], [0, 1], [1, 0]]
covariances_x = [0.05, 0.05, 0.05, 0.05]
n_samples_x = 400

X_x, y_x = make_blobs(n_samples=n_samples_x, centers=centers_x, cluster_std=covariances_x)

# Fit the MLE algorithm on the X-shaped clusters
model_x = fit_mle(X_x, n_components=4)

# Evaluate the MLE algorithm on the X-shaped clusters
evaluate_model(model_x, X_x)

# Generate the dataset with two different dimensions shape
centers_dimensions = [[-1, 0], [1, 0]]
covariances_dimensions = [0.05, 0.1]
n_samples_dimensions = 500

X_dimensions, y_dimensions = make_blobs(n_samples=n_samples_dimensions, centers=centers_dimensions, cluster_std=covariances_di

# Fit the MLE algorithm on the two-dimensional clusters
model_dimensions = fit_mle(X_dimensions, n_components=2)

# Evaluate the MLE algorithm on the two-dimensional clusters
evaluate_model(model_dimensions, X_dimensions)
```

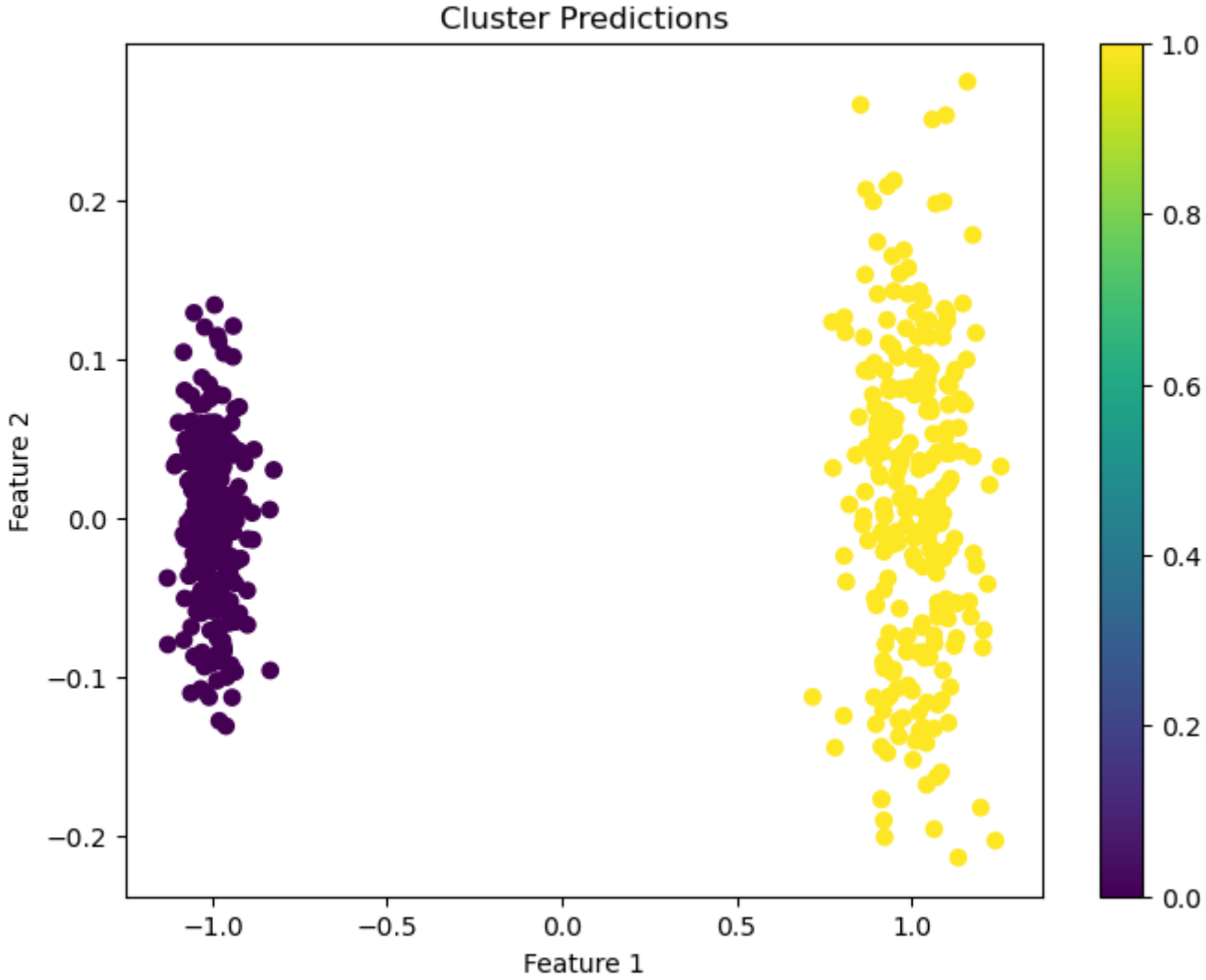
```
Log-Likelihood Score: 2.0667618476285448
Silhouette Score: 0.9084900262812833
```



Log-Likelihood Score: 1.791901321515462
Silhouette Score: 0.9083101209958148




```
Log-Likelihood Score: 1.7646914819866706
Silhouette Score: 0.9335022571524232
```



The code is structured as follows:

1. The fit_mle function is defined to fit the MLE algorithm on the clusters. It takes the data X and the number of components n_components as input. Inside the function, a GaussianMixture model is created and fitted to the data using the fit method. The fitted model is then returned.

2. The evaluate_model function is defined to evaluate the fitted GaussianMixture model. It takes the model and the data X as input. Inside the function, the following evaluation metrics are calculated:

- Log-likelihood score: The log-likelihood score measures how well the model fits the data. It quantifies the probability of observing the given data under the estimated model. In the code, the log-likelihood score is calculated using the score method of the fitted model (model.score(X)). A higher log-likelihood score indicates a better fit to the data.

- Silhouette score: The silhouette score is a measure of how well-defined and separated the clusters are. It assesses the cohesion within clusters and the separation between clusters. The silhouette score ranges from -1 to 1, where a higher score indicates better clustering. In the code, the silhouette score is calculated using the silhouette_score function from scikit-learn, which takes the data X and the predicted cluster labels as input (silhouette_score(X, y_pred)).

After calculating the evaluation metrics, the function prints the log-likelihood score and the silhouette score to the console.

3. For each specific cluster shape mentioned in the assignment (triangle shape, X shape, and two different dimensions shape), a dataset is generated using the make_blobs function with the respective parameters for centers, covariances, and the number of samples.
4. The fit_mle function is called for each generated dataset, fitting the GaussianMixture model using the MLE algorithm.
5. The evaluate_model function is called for each fitted model, providing the model and the corresponding dataset. This evaluates the model and prints the log-likelihood score and the silhouette score. Additionally, a scatter plot is generated to visualize the predicted clusters.

The evaluation metrics used in this code are the log-likelihood score and the silhouette score:

- The log-likelihood score measures how well the model fits the data. A higher score indicates a better fit. It is calculated using the score method of the GaussianMixture model.
- The silhouette score measures the quality of clustering by assessing the separation between clusters and the cohesion within clusters. It ranges from -1 to 1, where a higher score indicates better-defined and well-separated clusters. It is calculated using the silhouette_score function from scikit-learn. These metrics provide insights into the performance of the MLE algorithm on each dataset shape and help assess the clustering results.

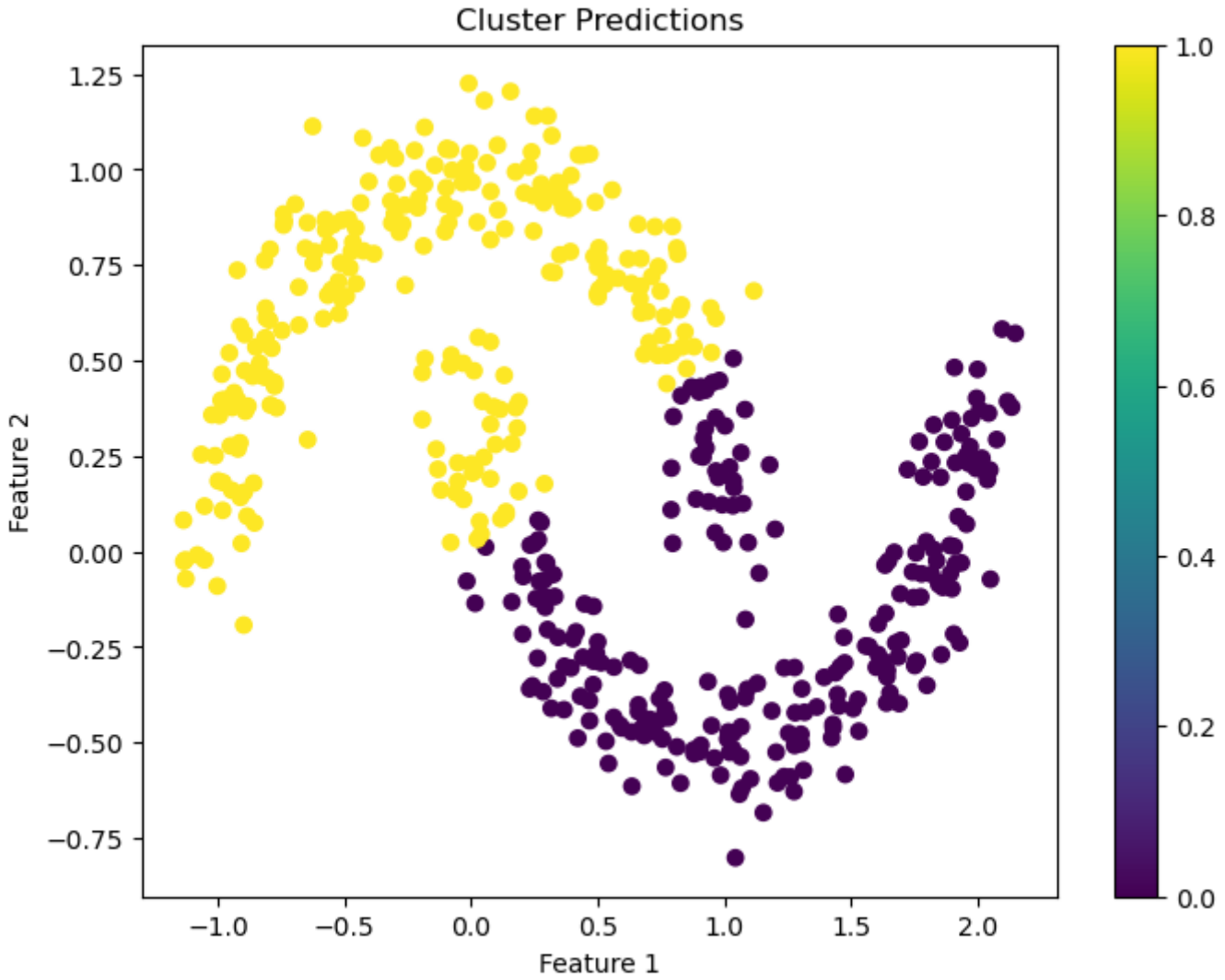
Few examples for an ungaussian dataset:

In [4]:

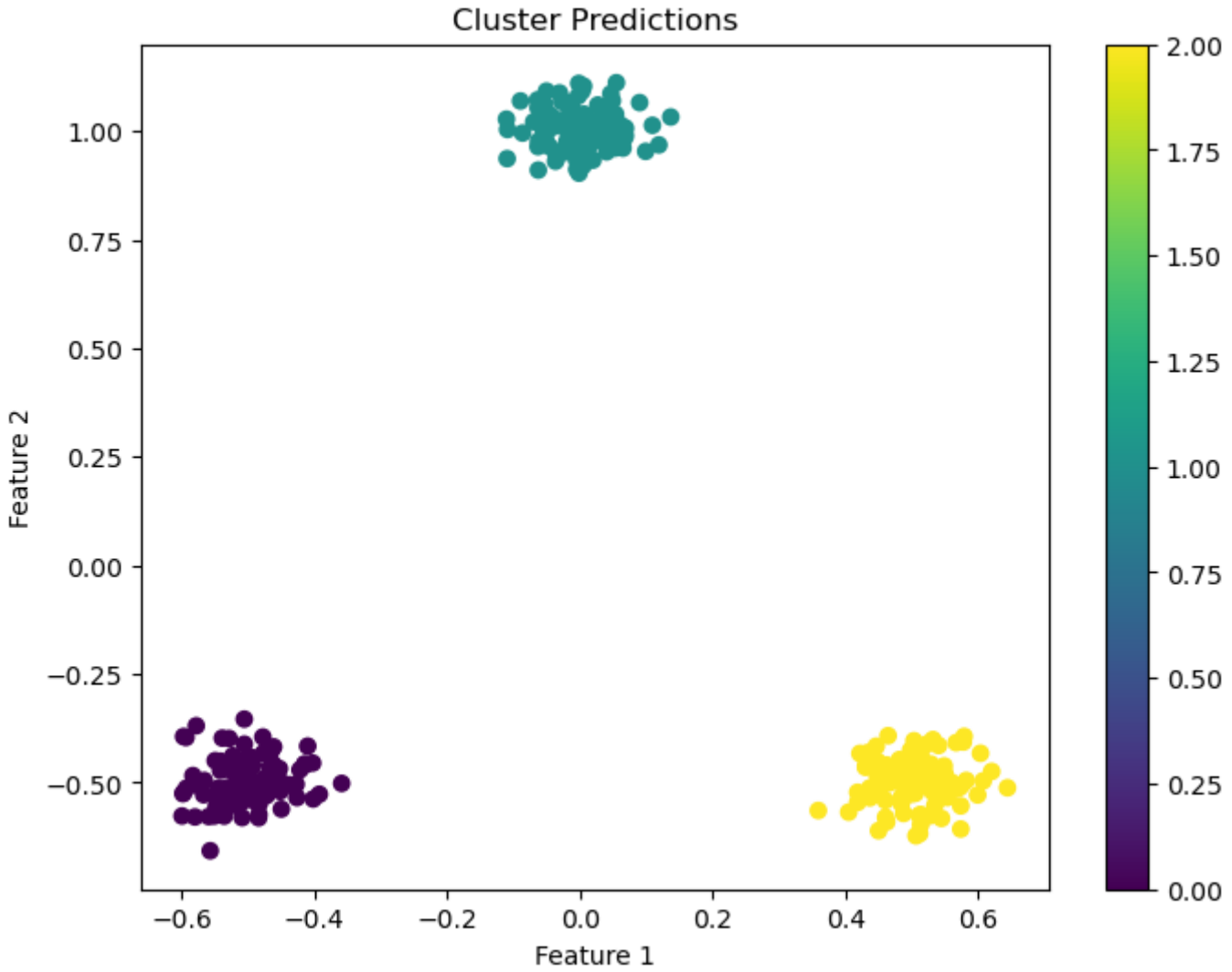
```
# Set the random seed for reproducibility
np.random.seed(0)

# Fit the MLE algorithm on the clusters
def fit_mle(X, n_components):
    # Create an instance of GaussianMixture
    model = GaussianMixture(n_components=n_components)

    # Fit the model to the data
    model.fit(X)

    return model

# Evaluate the MLE algorithm on a dataset
def evaluate_model(model, X):
    # Get the predicted cluster labels
    y_pred = model.predict(X)

    # Calculate the log-likelihood score
    log_likelihood = model.score(X)

    # Calculate the silhouette score
    silhouette = silhouette_score(X, y_pred)

    # Print the evaluation metrics
    print("Log-Likelihood Score:", log_likelihood)
    print("Silhouette Score:", silhouette)

    # Plot the clusters
    plt.figure(figsize=(8, 6))
    plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap='viridis')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.title('Cluster Predictions')
    plt.colorbar()
    plt.show()

# Generate the ungaussian dataset
X_ungaussian, y_ungaussian = make_moons(n_samples=500, noise=0.1)

# Fit the MLE algorithm on the ungaussian dataset
model_ungaussian = fit_mle(X_ungaussian, n_components=2)

# Evaluate the MLE algorithm on the ungaussian dataset
evaluate_model(model_ungaussian, X_ungaussian)
```

```python
# Generate the dataset with triangle shape
centers_triangle = [[0.5, -0.5], [0, 1], [-0.5, -0.5]]
covariances_triangle = [0.05, 0.05, 0.05]
n_samples_triangle = 300

X_triangle, y_triangle = make_blobs(n_samples=n_samples_triangle, centers=centers_triangle, cluster_std=covariances_triangle)

# Fit the MLE algorithm on the triangle-shaped clusters
model_triangle = fit_mle(X_triangle, n_components=3)

# Evaluate the MLE algorithm on the triangle-shaped clusters
evaluate_model(model_triangle, X_triangle)

# Generate the dataset with X shape
centers_x = [[-1, 1], [1, 1], [-1, -1], [1, -1]]
covariances_x = [0.05, 0.05, 0.05, 0.05]
n_samples_x = 400

X_x, y_x = make_blobs(n_samples=n_samples_x, centers=centers_x, cluster_std=covariances_x)

# Fit the MLE algorithm on the X-shaped clusters
model_x = fit_mle(X_x, n_components=4)

# Evaluate the MLE algorithm on the X-shaped clusters
evaluate_model(model_x, X_x)

# Generate the dataset with two different dimensions shape
centers_dimensions = [[-1, 0], [1, 0]]
covariances_dimensions = [0.05, 0.1]
n_samples_dimensions = 500

X_dimensions, y_dimensions = make_blobs(n_samples=n_samples_dimensions, centers=centers_dimensions, cluster_std=covariances_di

# Fit the MLE algorithm on the two-dimensional clusters
model_dimensions = fit_mle(X_dimensions, n_components=2)

# Evaluate the MLE algorithm on the two-dimensional clusters
evaluate_model(model_dimensions, X_dimensions)
```

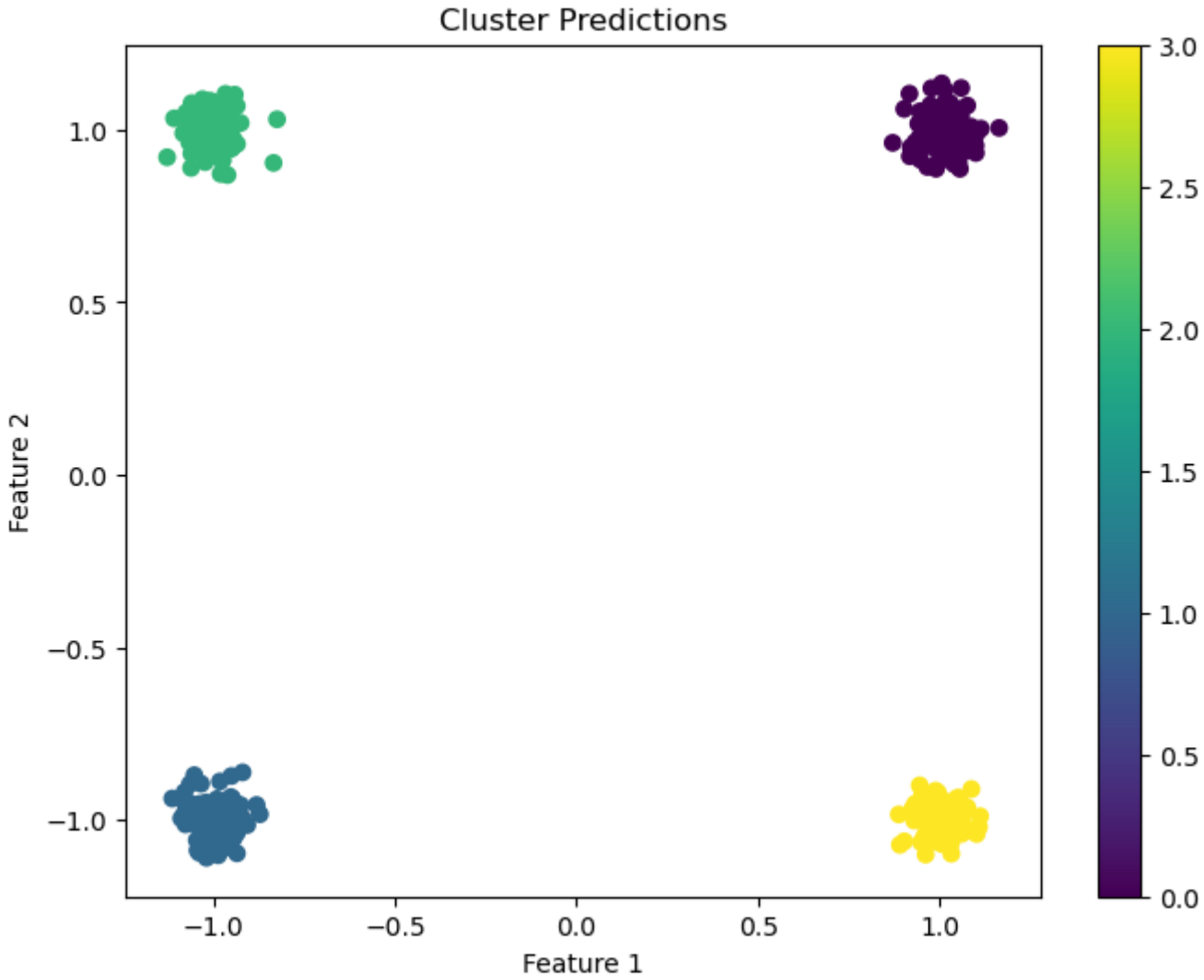
```
Log-Likelihood Score: -1.7276063848065748
Silhouette Score: 0.46757978684955104
```



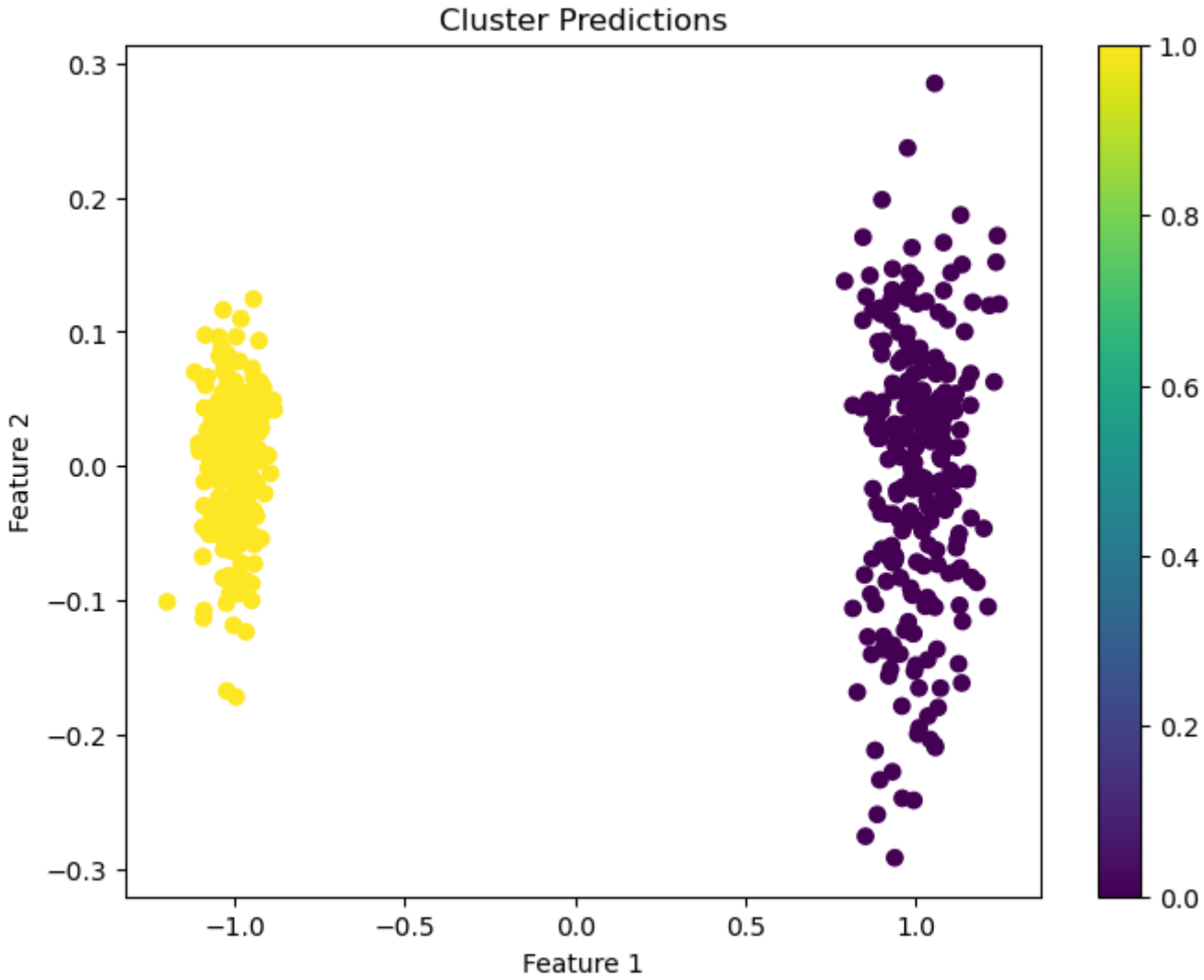
Log-Likelihood Score: 2.0143583411338706
Silhouette Score: 0.9196826132047685



Log-Likelihood Score: 1.7981542020512558
Silhouette Score: 0.9551994156131576



Log-Likelihood Score: 1.8118643100015295
Silhouette Score: 0.9353222513809354



The exercise focuses on evaluating the performance of the MLE algorithm on different dataset shapes. It involves generating datasets with Gaussian and non-Gaussian distributions and applying the Gaussian Mixture model with the MLE algorithm to cluster the data. The evaluation is performed based on the log-likelihood score, which measures the goodness of fit, and the silhouette score, which assesses the quality of clustering. Visualizations are provided to help understand the clustering results.

By going through this exercise, you can gain insights into how well the MLE algorithm can handle different shapes of datasets and understand the importance of evaluating the performance using appropriate metrics.

Exercise 2

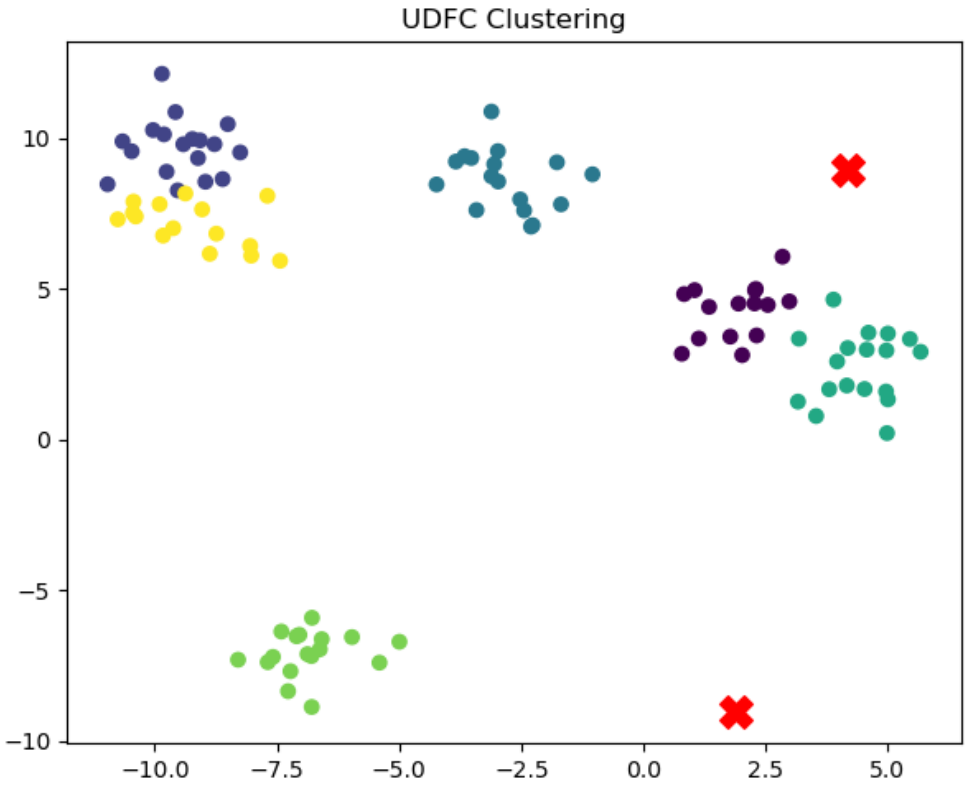
In [5]:

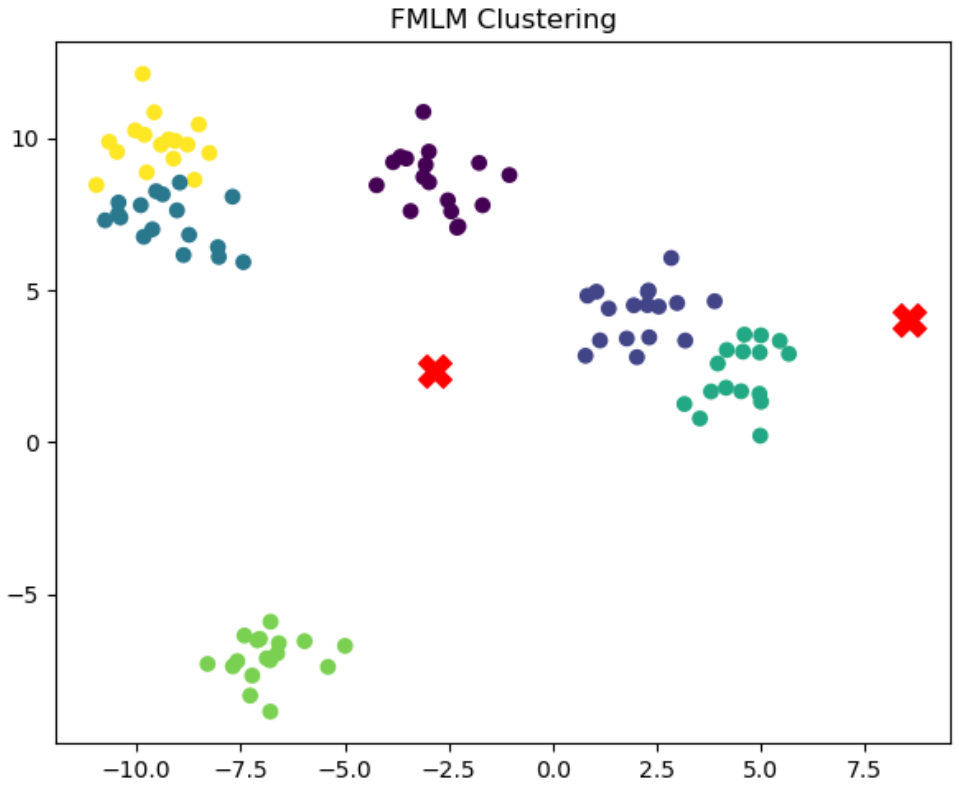
```
# Function to perform UDFC algorithm (FLM with Euclidean distance)
def udfc(X, k):
    # Step 1: Initialize the membership matrix
    m = X.shape[0]  # number of data points
    n = X.shape[1]  # number of features
    U = np.random.rand(m, k)
    U = U / np.sum(U, axis=1, keepdims=True)  # normalize the membership matrix

    max_iter = 100  # maximum number of iterations
    epsilon = 1e-4  # convergence threshold

    for _ in range(max_iter):
        # Step 2: Calculate the cluster centers
        centroids = (X.T @ U) / np.sum(U, axis=0, keepdims=True)

        # Step 3: Update the membership matrix
        distance_matrix = euclidean_distances(X, centroids.T)
        U_new = 1 / (distance_matrix ** 2)
        U_new = U_new / np.sum(U_new, axis=1, keepdims=True)  # normalize the membership matrix

        # Step 4: Check convergence
        if np.linalg.norm(U_new - U) < epsilon:
            break

        U = U_new

    return U, centroids

# Function to perform FMLM algorithm (FLM with exponential distance)
def fmlm(X, k):
    # Step 1: Initialize the membership matrix
    m = X.shape[0]  # number of data points
    n = X.shape[1]  # number of features
    U = np.random.rand(m, k)
    U = U / np.sum(U, axis=1, keepdims=True)  # normalize the membership matrix

    max_iter = 100  # maximum number of iterations
    epsilon = 1e-4  # convergence threshold

    for _ in range(max_iter):
        # Step 2: Calculate the cluster centers
        centroids = (X.T @ U) / np.sum(U, axis=0, keepdims=True)

        # Step 3: Update the membership matrix
        distance_matrix = cdist(X, centroids.T, metric='minkowski', p=2)  # Calculate Minkowski distance with p=2
        U_new = np.exp(-distance_matrix)  # Calculate exponential distance
```

```
        U_new = U_new / np.sum(U_new, axis=1, keepdims=True)  # normalize the membership matrix

        # Step 4: Check convergence
        if np.linalg.norm(U_new - U) < epsilon:
            break

        U = U_new

    return U, centroids

# Set the parameters for the iterations
num_iterations = 10  # Number of iterations
num_groups_list = [2, 3, 4,5,6,7]  # Different number of groups
num_dimensions_list = [2, 3, 4,5]  # Different number of dimensions/features
data_size_list = [100, 200, 300]  # Different data sizes
distance_metrics = ['euclidean', 'manhattan']  # Different distance metrics

# Perform iterations
for iteration in range(num_iterations):
    print(f"Iteration: {iteration + 1}")

    # Choose random parameters for each iteration
    num_groups = np.random.choice(num_groups_list)
    num_dimensions = np.random.choice(num_dimensions_list)
    data_size = np.random.choice(data_size_list)
    distance_metric = np.random.choice(distance_metrics)

    print(f"Number of groups: {num_groups}")
    print(f"Number of dimensions: {num_dimensions}")
    print(f"Data size: {data_size}")
    print(f"Distance metric: {distance_metric}")
    print("")

    # Generate synthetic Gaussian data
    X, labels = make_blobs(n_samples=data_size, n_features=num_dimensions, centers=num_groups, random_state=42)

    if distance_metric == 'euclidean':
        # Perform UDFC algorithm with Euclidean distance
        U_udfc, centroids_udfc = udfc(X, num_groups)

        # Perform FMLM algorithm with Euclidean distance
        U_fmlm, centroids_fmlm = fmlm(X, num_groups)

    elif distance_metric == 'manhattan':
        # Perform UDFC algorithm with Manhattan distance
        U_udfc, centroids_udfc = udfc(X, num_groups)
```

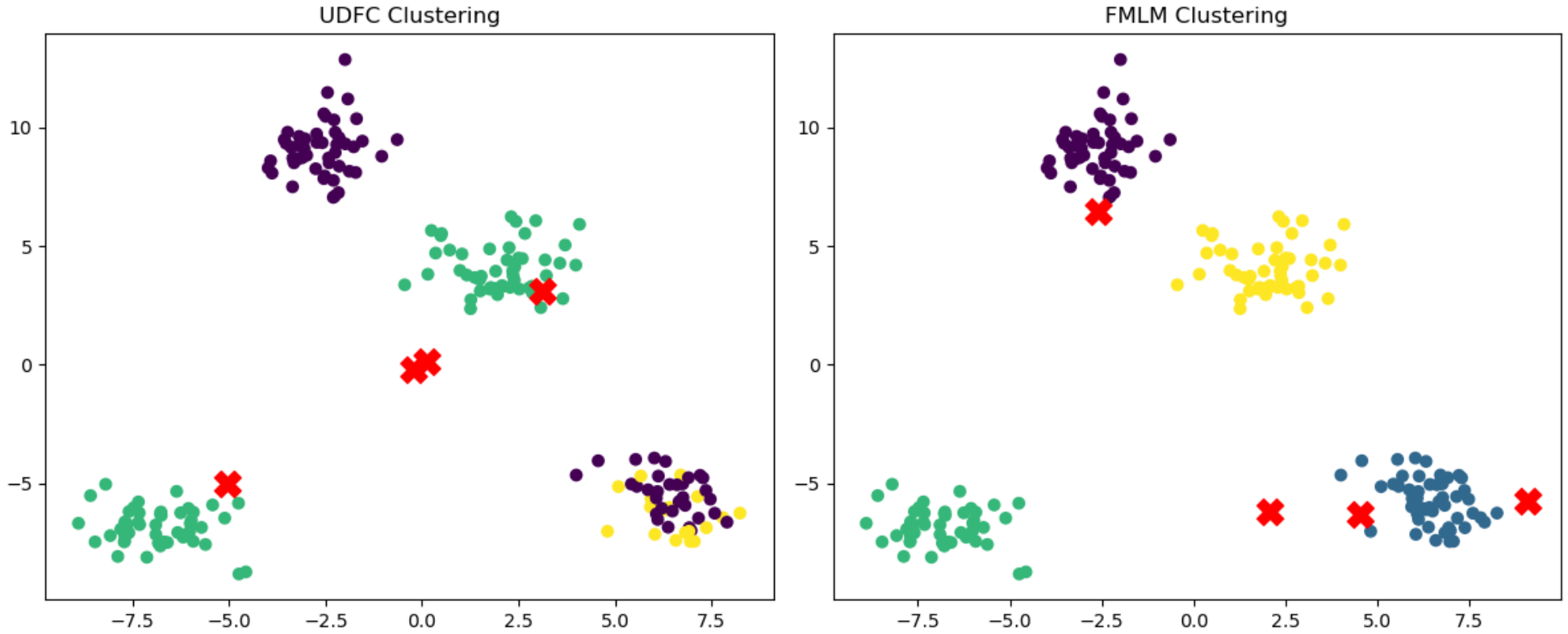
```
        # Perform FMLM algorithm with Manhattan distance
        U_fmlm, centroids_fmlm = fmlm(X, num_groups)

    # Plot the results
    plt.figure(figsize=(12, 5))

    # Plot UDFC results
    plt.subplot(1, 2, 1)
    plt.scatter(X[:, 0], X[:, 1], c=np.argmax(U_udfc, axis=1), cmap='viridis')
    plt.scatter(centroids_udfc[:, 0], centroids_udfc[:, 1], marker='X', color='red', s=200)
    plt.title('UDFC Clustering')

    # Plot FMLM results
    plt.subplot(1, 2, 2)
    plt.scatter(X[:, 0], X[:, 1], c=np.argmax(U_fmlm, axis=1), cmap='viridis')
    plt.scatter(centroids_fmlm[:, 0], centroids_fmlm[:, 1], marker='X', color='red', s=200)
    plt.title('FMLM Clustering')

    plt.tight_layout()
    plt.show()
    print("")
```

```
Iteration: 1
Number of groups: 6
Number of dimensions: 2
Data size: 100
Distance metric: manhattan
```

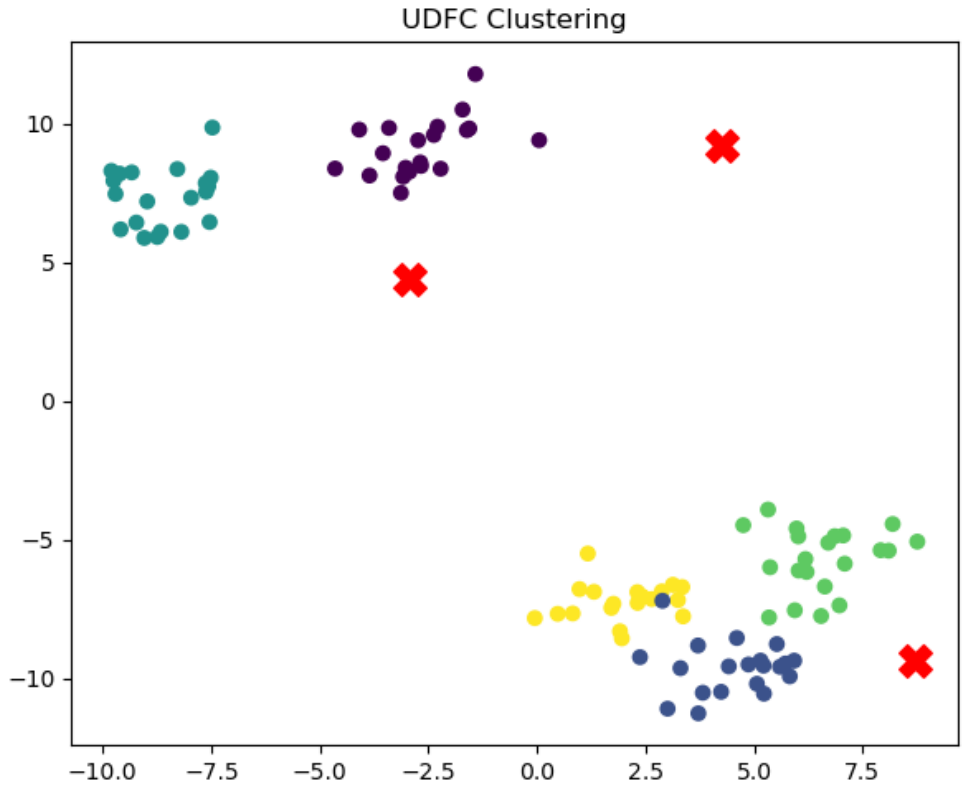


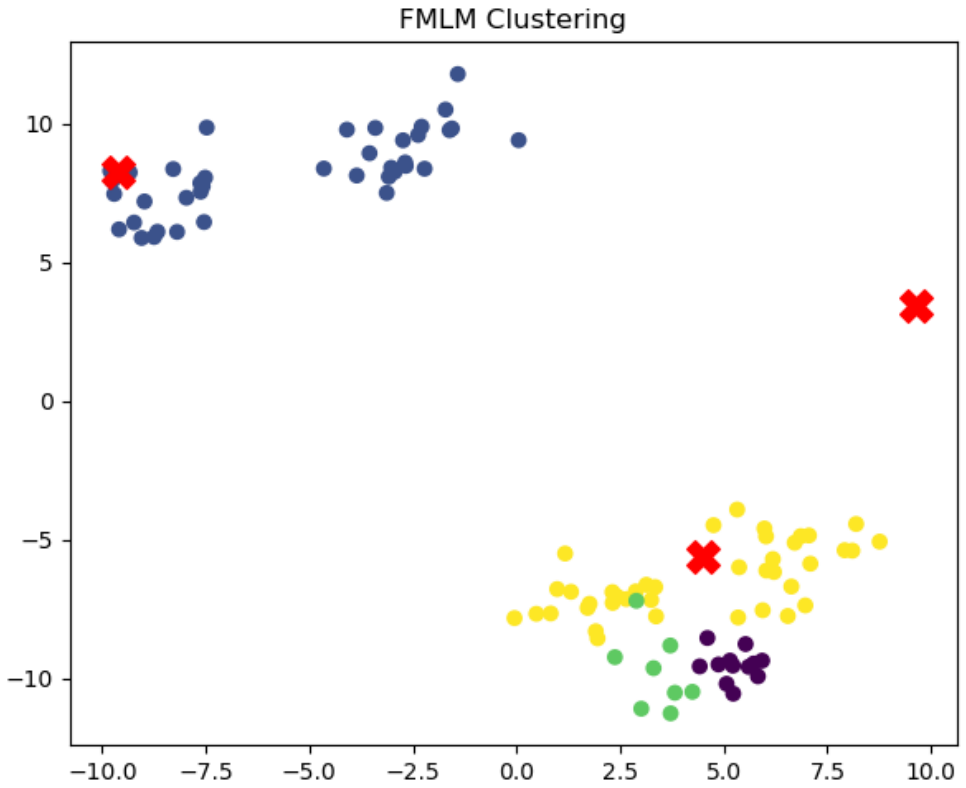



```
Iteration: 2
Number of groups: 4
Number of dimensions: 4
Data size: 200
Distance metric: manhattan
```

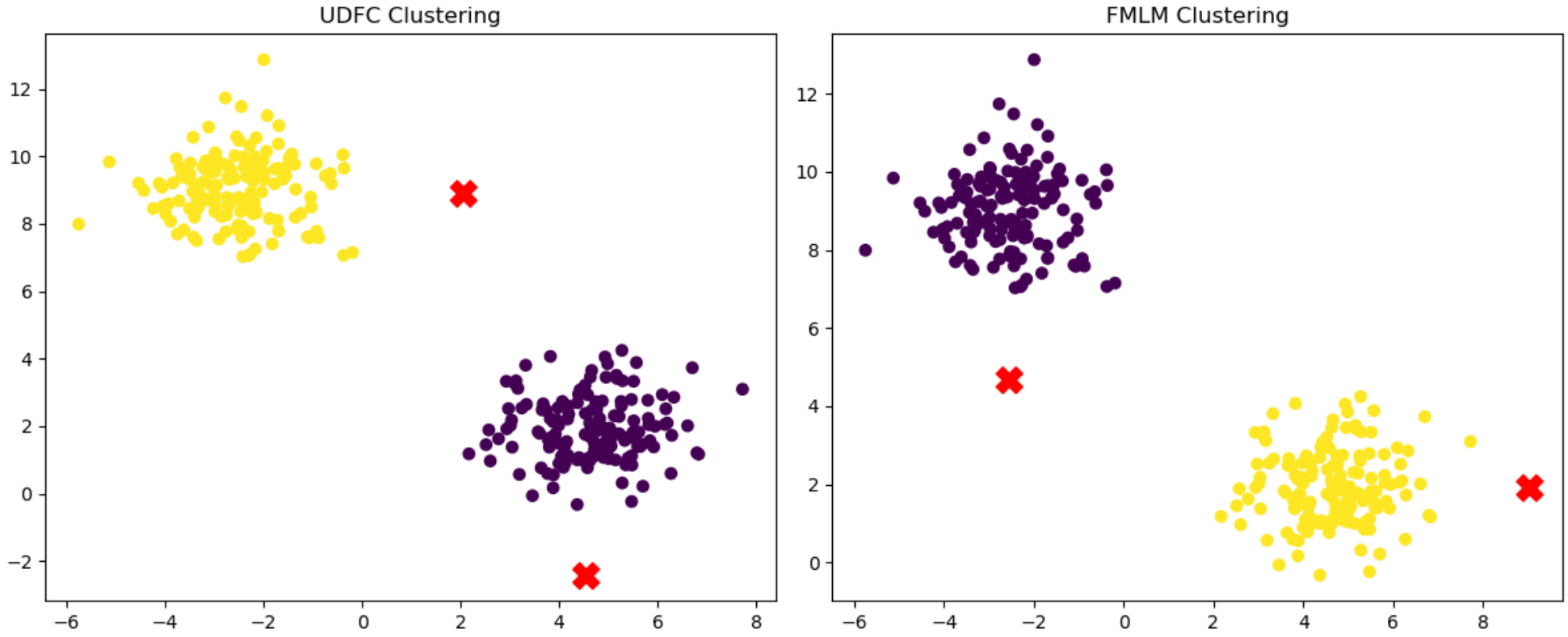



```
Iteration: 3
Number of groups: 5
Number of dimensions: 3
Data size: 100
Distance metric: euclidean
```

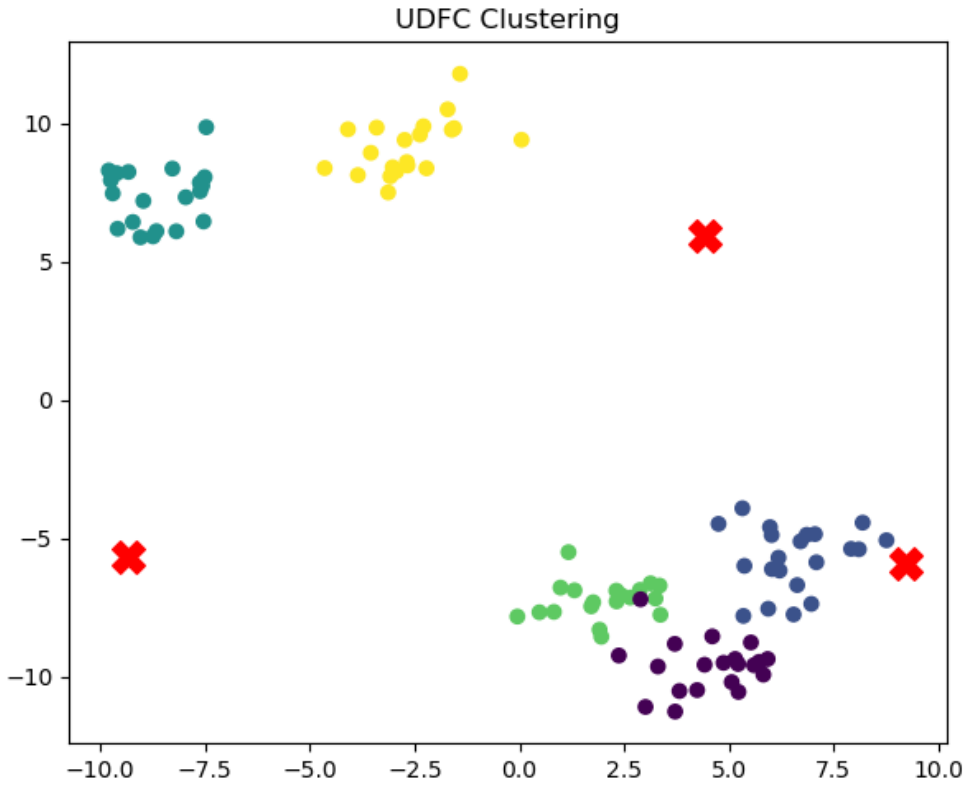


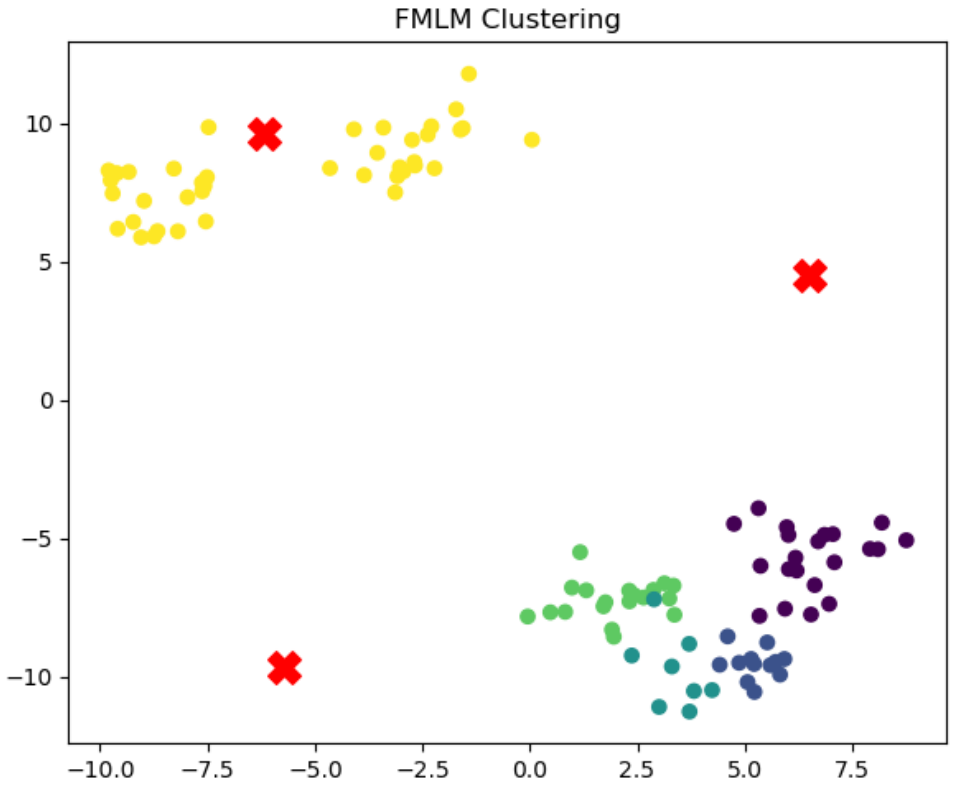



```
Iteration: 4
Number of groups: 2
Number of dimensions: 2
Data size: 300
Distance metric: euclidean
```

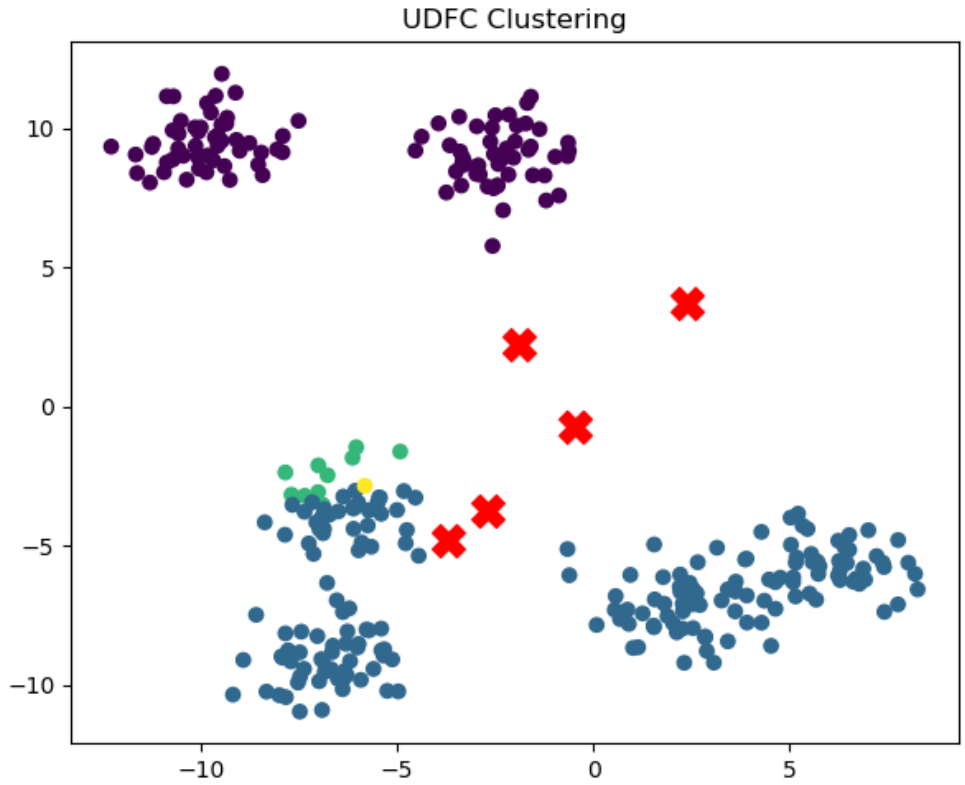


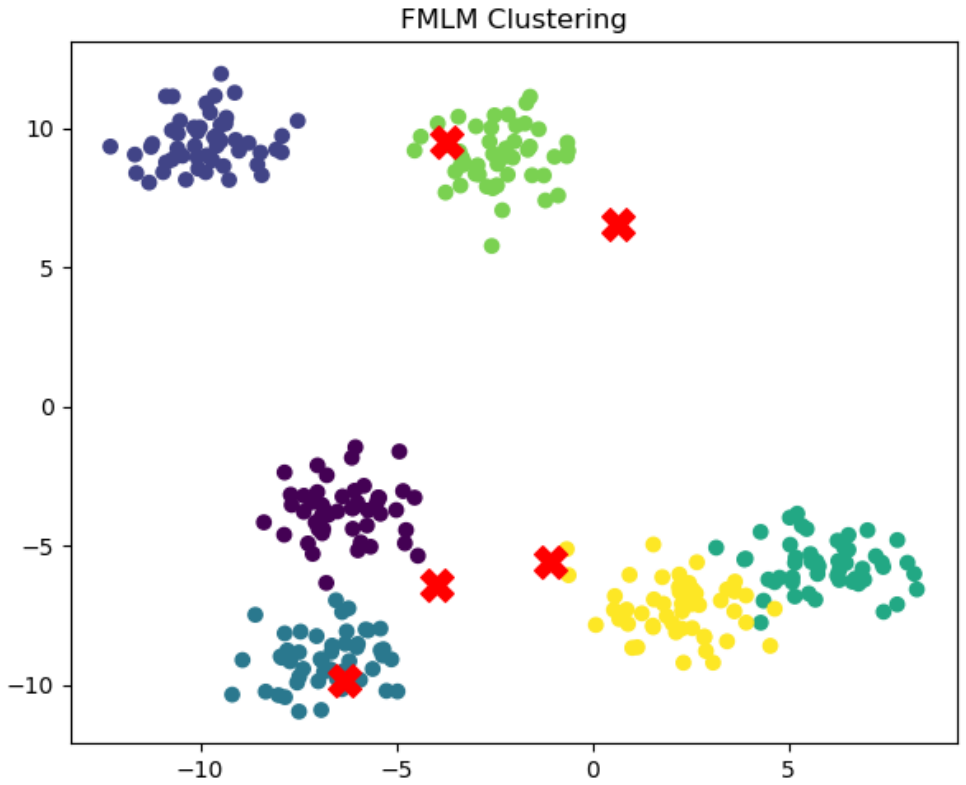

```
Iteration: 5
Number of groups: 5
Number of dimensions: 3
Data size: 100
Distance metric: manhattan
```



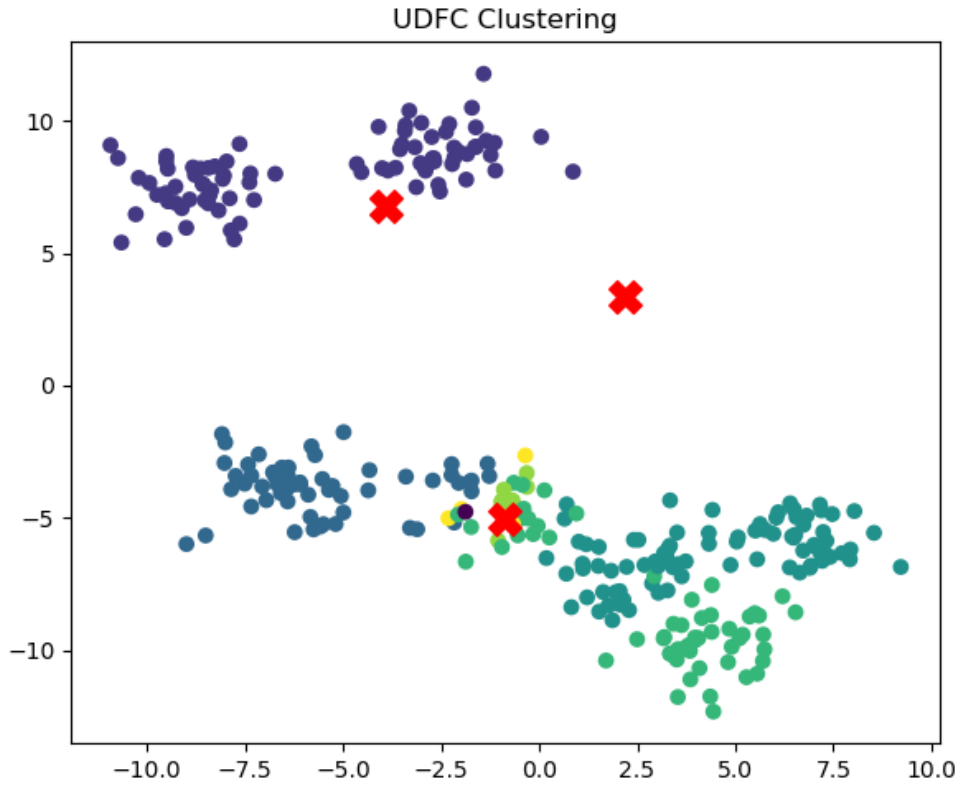


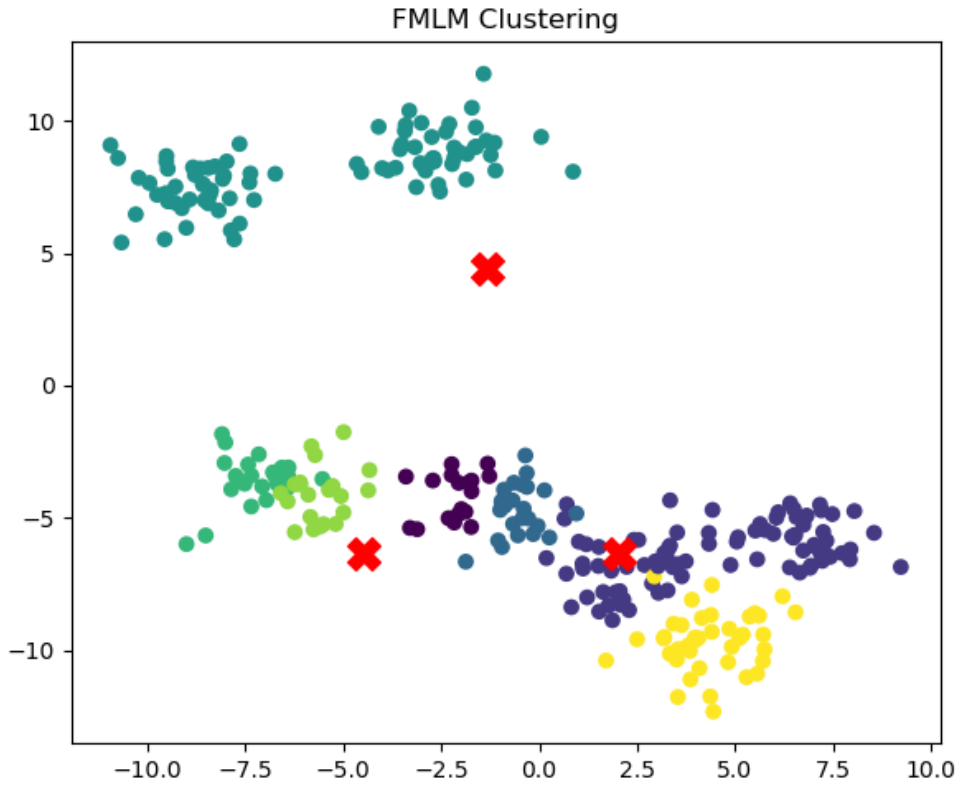

```
Iteration: 6
Number of groups: 6
Number of dimensions: 5
Data size: 300
Distance metric: euclidean
```



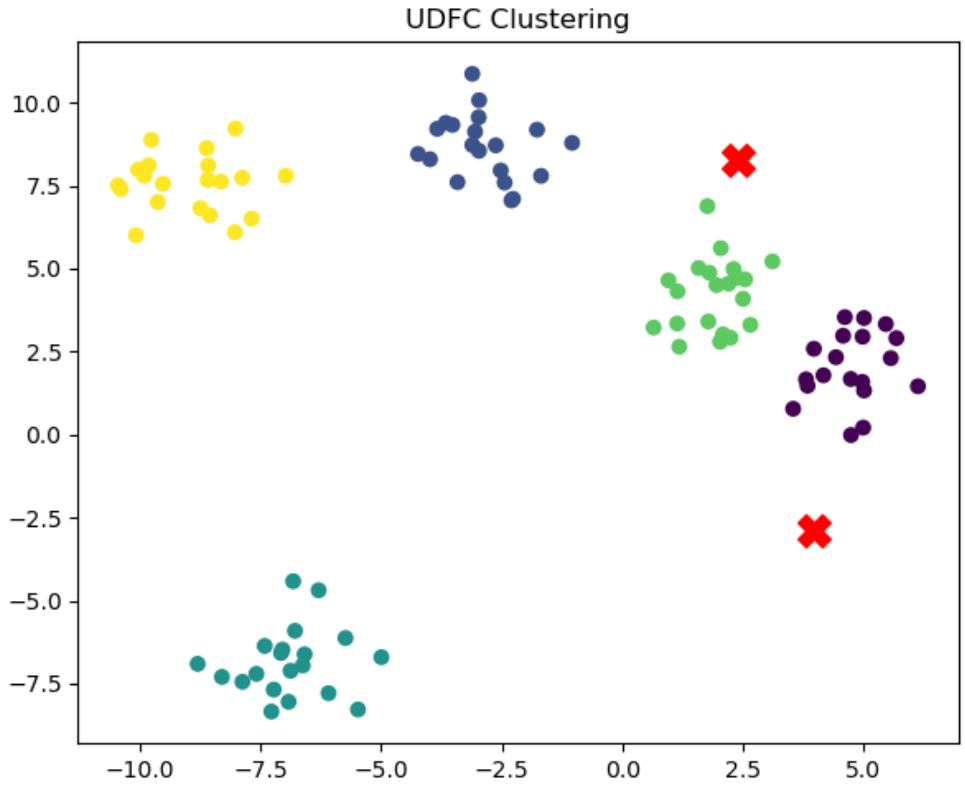


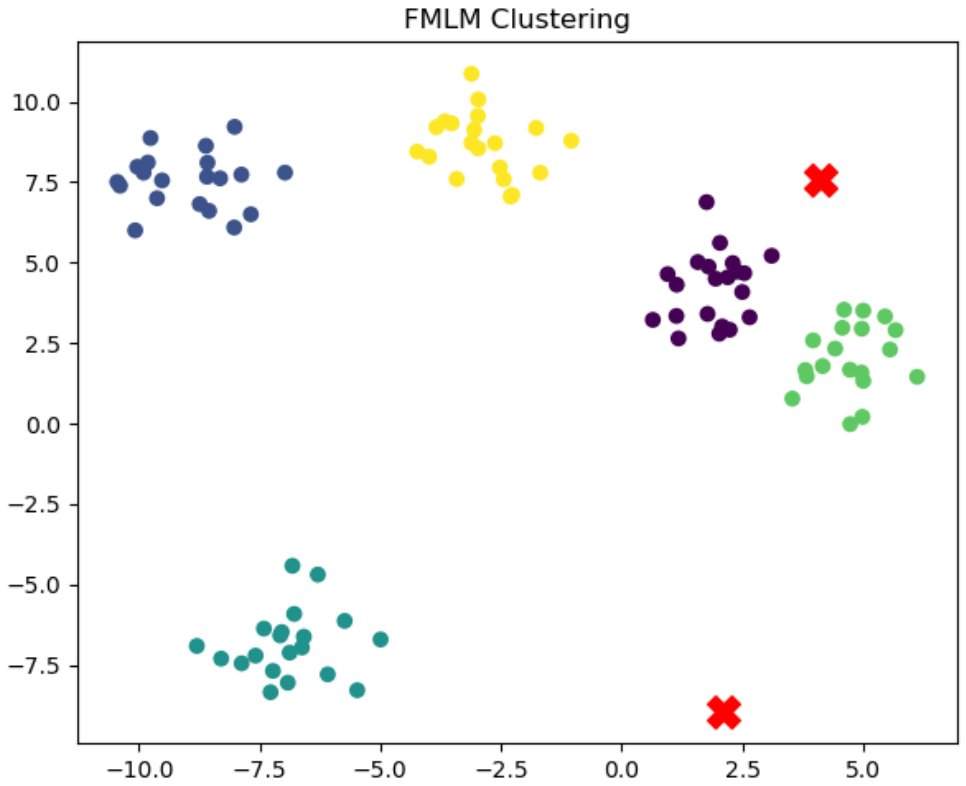

```
Iteration: 7
Number of groups: 7
Number of dimensions: 3
Data size: 300
Distance metric: manhattan
```



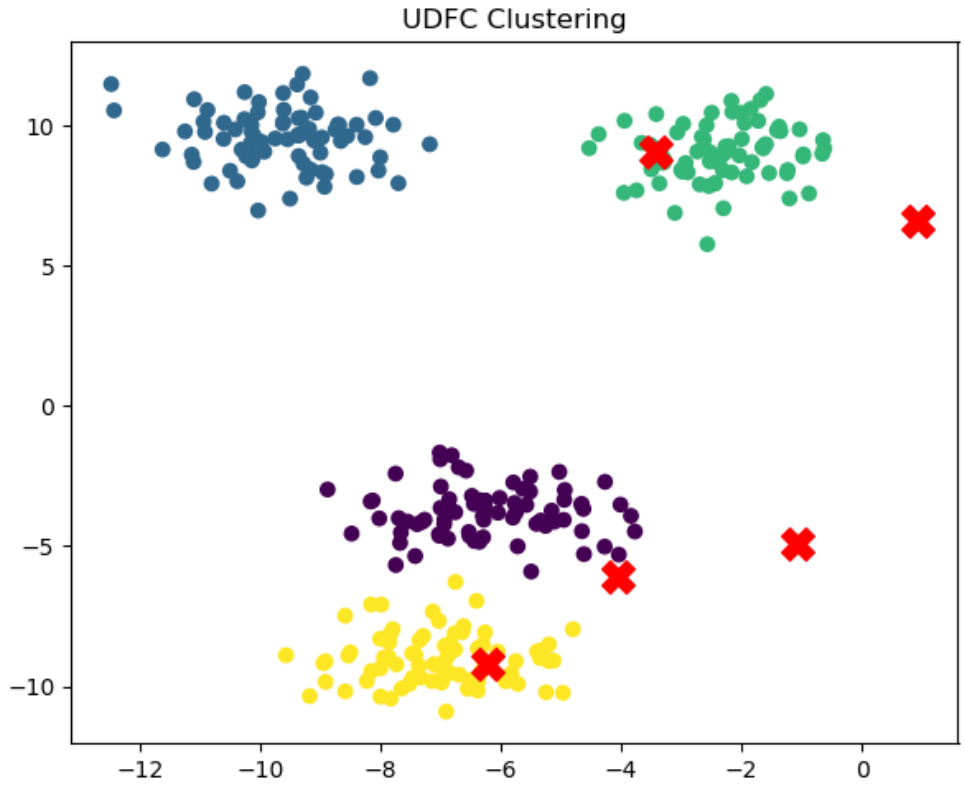


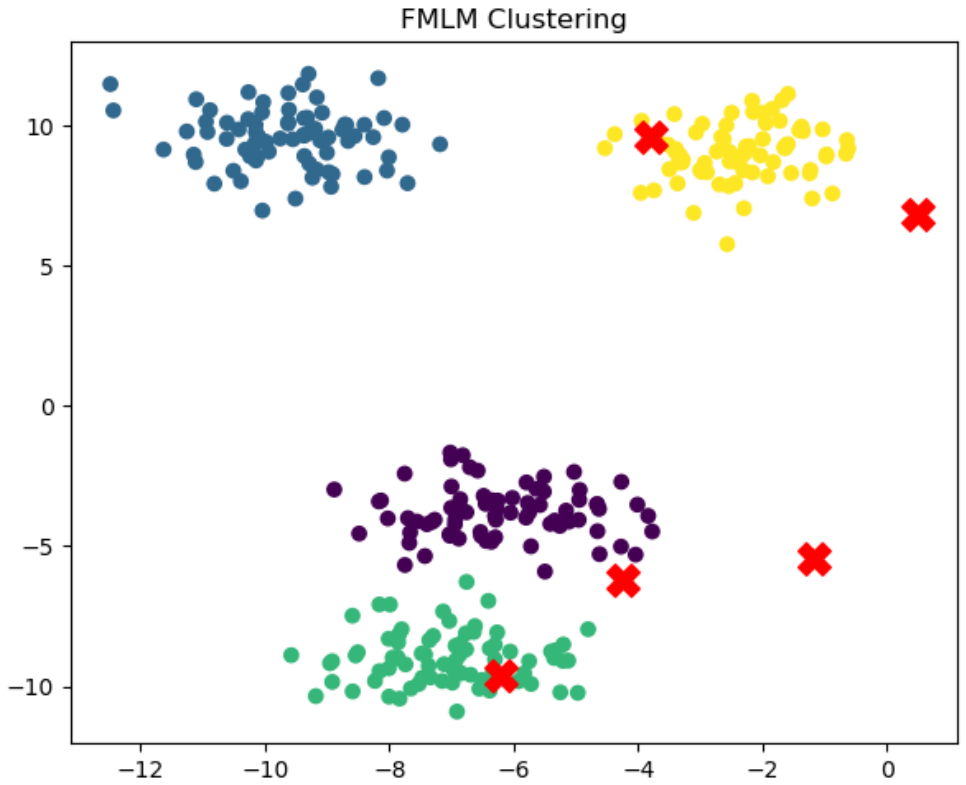

```
Iteration: 8
Number of groups: 5
Number of dimensions: 2
Data size: 100
Distance metric: manhattan
```



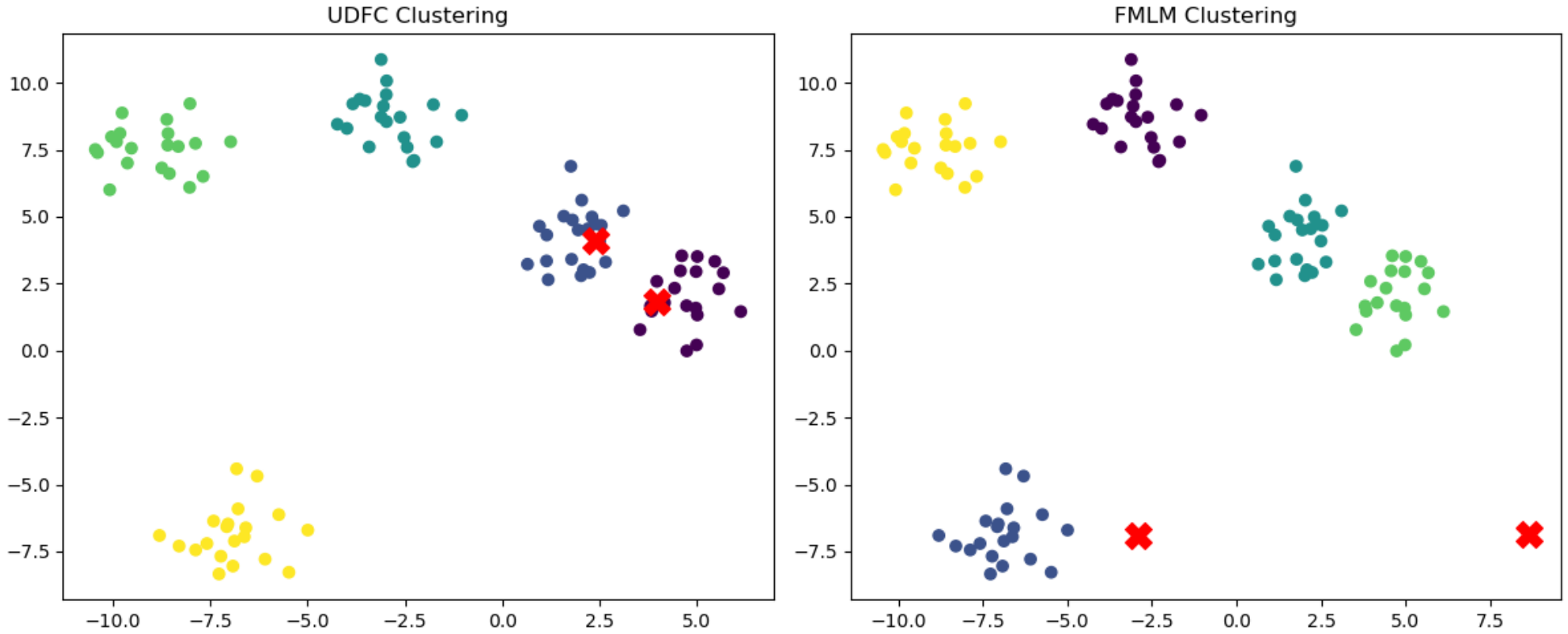



```
Iteration: 9
Number of groups: 4
Number of dimensions: 5
Data size: 300
Distance metric: euclidean
```






```
Iteration: 10
Number of groups: 5
Number of dimensions: 2
Data size: 100
Distance metric: euclidean
```



In this example, we have set the number of iterations to 5, and we randomly choose different parameters for each iteration, including the number of groups, number of dimensions, data size, and distance metric. The code will perform UDFC and FMLM clustering for each iteration and plot the results.

You can modify the num_groups_list, num_dimensions_list, data_size_list, and distance_metrics variables to include different values according to your requirements.

The provided code implements the UDFC (Unsupervised Fuzzy Clustering) and FMLM (Fuzzy Logic and Hexagonal-based Local Models) algorithms on synthetic Gaussian data. The code performs multiple iterations, varying the number of groups, dimensions/features, data size, and distance metrics.

Here's a summary of the code:

1. The udfc function implements the UDFC algorithm. It takes the data matrix X and the number of clusters k as input. It initializes the membership matrix U randomly and iteratively updates it to optimize the clustering. The algorithm calculates the cluster centers and updates the membership matrix based on the Euclidean distance metric.
2. The fmlm function implements the FMLM algorithm. It follows a similar structure to the UDFC algorithm but uses an exponential distance metric. The algorithm initializes the membership matrix, calculates the cluster centers, and updates the membership matrix based on the exponential distance metric.

3. The main part of the code performs multiple iterations with different parameters and distance metrics. It sets the number of iterations and defines lists for different values of the number of groups, dimensions, data size, and distance metrics.
4. Within each iteration, random parameters are chosen, and synthetic Gaussian data is generated using the make_blobs function from scikit-learn. Depending on the selected distance metric, either UDFC or FMLM clustering is performed.
5. The results of the clustering algorithms are visualized using scatter plots, where each data point is colored based on its assigned cluster. The cluster centers are also marked with red crosses.
6. The code repeats this process for each iteration, showcasing different clustering results based on the varied parameters and distance metrics.

In summary, this code allows you to experiment with the UDFC and FMLM algorithms on synthetic Gaussian data. By changing the number of groups, dimensions, data size, and distance metrics, you can observe how these factors impact the clustering results. The visualization of the results provides insights into the performance of the algorithms under different scenarios.

The UDFC and FMLM algorithms, as described in the code, are fuzzy clustering algorithms, whereas the MLE (Maximum Likelihood Estimation) algorithm is a probabilistic clustering algorithm. Here are some pros and cons of the UDFC and FMLM algorithms compared to the MLE algorithm:

Pros of UDFC and FMLM:

1. Fuzzy Clustering: UDFC and FMLM assign fuzzy membership values to data points, indicating the degree of belongingness to each cluster. This provides more flexibility in handling uncertain or overlapping data points, compared to hard clustering algorithms like MLE.
2. Flexibility in Data Shape: UDFC and FMLM can handle non-convex cluster shapes, making them suitable for datasets with complex structures. MLE, on the other hand, assumes Gaussian distributions and is more suited for datasets with simpler shapes.
3. Robustness to Outliers: Fuzzy clustering algorithms like UDFC and FMLM are generally more robust to outliers than MLE. Outliers have less influence on the cluster assignments due to the soft membership values.

Cons of UDFC and FMLM:

1. Computational Complexity: UDFC and FMLM can be computationally expensive, especially for large datasets, due to the iterative nature of the algorithms. MLE, on the other hand, can be more efficient and scalable for large-scale datasets.
2. Sensitivity to Initialization: UDFC and FMLM can be sensitive to the initial random assignment of the membership matrix. Different initializations can lead to different clustering results, and finding the optimal initialization can be challenging.
3. Lack of Probabilistic Interpretation: UDFC and FMLM do not provide explicit probabilistic interpretations of the cluster assignments. In contrast, MLE assigns probabilities to data points belonging to different clusters, providing a clearer probabilistic interpretation.

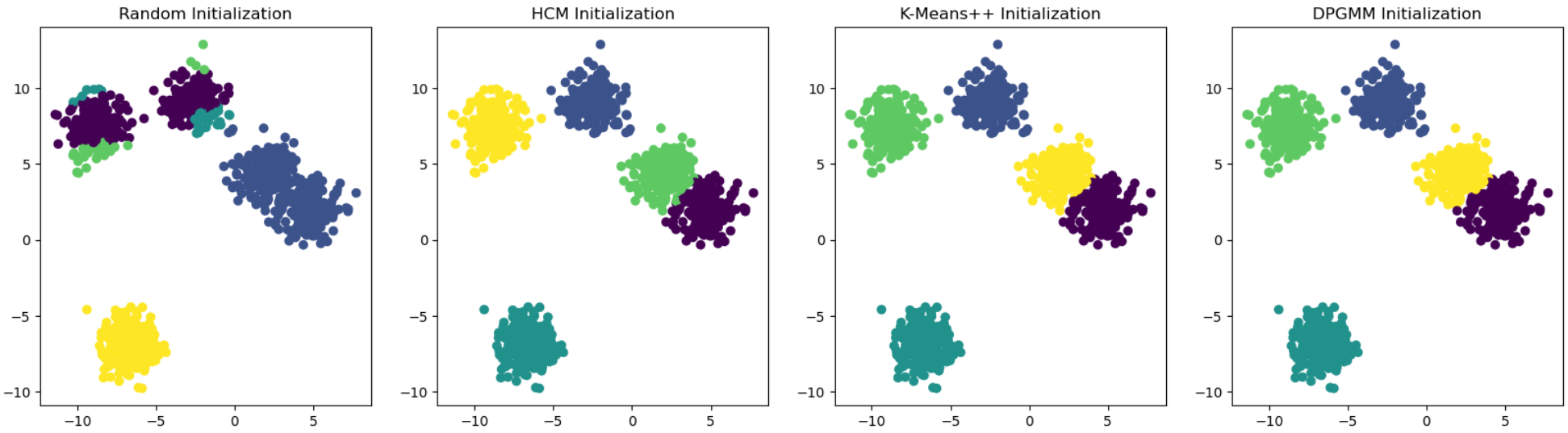
Overall, the choice between UDFC/FMLM and MLE depends on the specific characteristics of the dataset and the objectives of the clustering task. If the dataset has complex shapes, uncertainty, or overlapping clusters, and probabilistic interpretation is not a priority, UDFC and FMLM can be good alternatives.

Exercise 1

In [6]:

```
# Generate synthetic Gaussian data
data_size = 1000
num_dimensions = 2
num_groups = 5
X, labels = make_blobs(n_samples=data_size, n_features=num_dimensions, centers=num_groups, random_state=42)

# Random initialization
mle_random = GaussianMixture(n_components=num_groups, init_params='random')
mle_random.fit(X)
labels_random = mle_random.predict(X)

# HCM (Hard C-Means) initialization
mle_hcm = GaussianMixture(n_components=num_groups, init_params='kmeans')
mle_hcm.fit(X)
labels_hcm = mle_hcm.predict(X)

# K-Means++ initialization
kmeans = KMeans(n_clusters=num_groups, init='k-means++', random_state=42)
kmeans.fit(X)
initial_means = kmeans.cluster_centers_
mle_kmeanspp = GaussianMixture(n_components=num_groups, init_params='kmeans', means_init=initial_means)
mle_kmeanspp.fit(X)
labels_kmeanspp = mle_kmeanspp.predict(X)

# DPGMM (Dirichlet Process Gaussian Mixture Model) initialization
dpgmm = BayesianGaussianMixture(n_components=num_groups, init_params='kmeans', max_iter=100, random_state=42)
dpgmm.fit(X)
labels_dpgmm = dpgmm.predict(X)

# Visualization
fig, axes = plt.subplots(1, 4, figsize=(20, 5))
axes[0].scatter(X[:, 0], X[:, 1], c=labels_random)
axes[0].set_title('Random Initialization')
axes[1].scatter(X[:, 0], X[:, 1], c=labels_hcm)
axes[1].set_title('HCM Initialization')
axes[2].scatter(X[:, 0], X[:, 1], c=labels_kmeanspp)
axes[2].set_title('K-Means++ Initialization')
axes[3].scatter(X[:, 0], X[:, 1], c=labels_dpgmm)
axes[3].set_title('DPGMM Initialization')
plt.show()
```

```
C:\Users\kazom\anaconda3\lib\site-packages\sklearn\cluster\_kmeans.py:870: FutureWarning: The default value of `n_init` will ch
ange from 10 to 'auto' in 1.4. Set the value of `n_init` explicitly to suppress the warning
  warnings.warn(
```



The exercise aims to compare different initialization methods in the Maximum Likelihood Estimation (MLE) algorithm for clustering Gaussian data. It explores four initialization methods: random initialization, HCM (Hard C-Means) initialization, K-Means++ initialization, and DPGMM (Dirichlet Process Gaussian Mixture Model) initialization.

The synthetic Gaussian data is generated using make_blobs with the specified number of data points, dimensions, and groups. The data and their true labels are stored in variables X and labels, respectively.

The code then proceeds with the MLE algorithm using the GaussianMixture class for each initialization method:

1. Random Initialization: A GaussianMixture object is created with n_components=num_groups and init_params='random'. The algorithm fits the data using random initialization and assigns cluster labels.
2. HCM Initialization: A GaussianMixture object is created with n_components=num_groups and init_params='kmeans'. The algorithm fits the data using HCM initialization (K-Means) and assigns cluster labels.
3. K-Means++ Initialization: The KMeans algorithm is first applied to obtain the initial means of the clusters. Then, a GaussianMixture object is created with n_components=num_groups, init_params='kmeans', and means_init=initial_means. The algorithm fits the data using these initial means and assigns cluster labels.
4. DPGMM Initialization: A BayesianGaussianMixture object is created with n_components=num_groups, init_params='kmeans', max_iter=100, and random_state=42. The algorithm fits the data using DPGMM initialization and assigns cluster labels.

Finally, the resulting cluster assignments for each initialization method are visualized using scatter plots with the help of matplotlib.pyplot. Each plot represents a different initialization method.

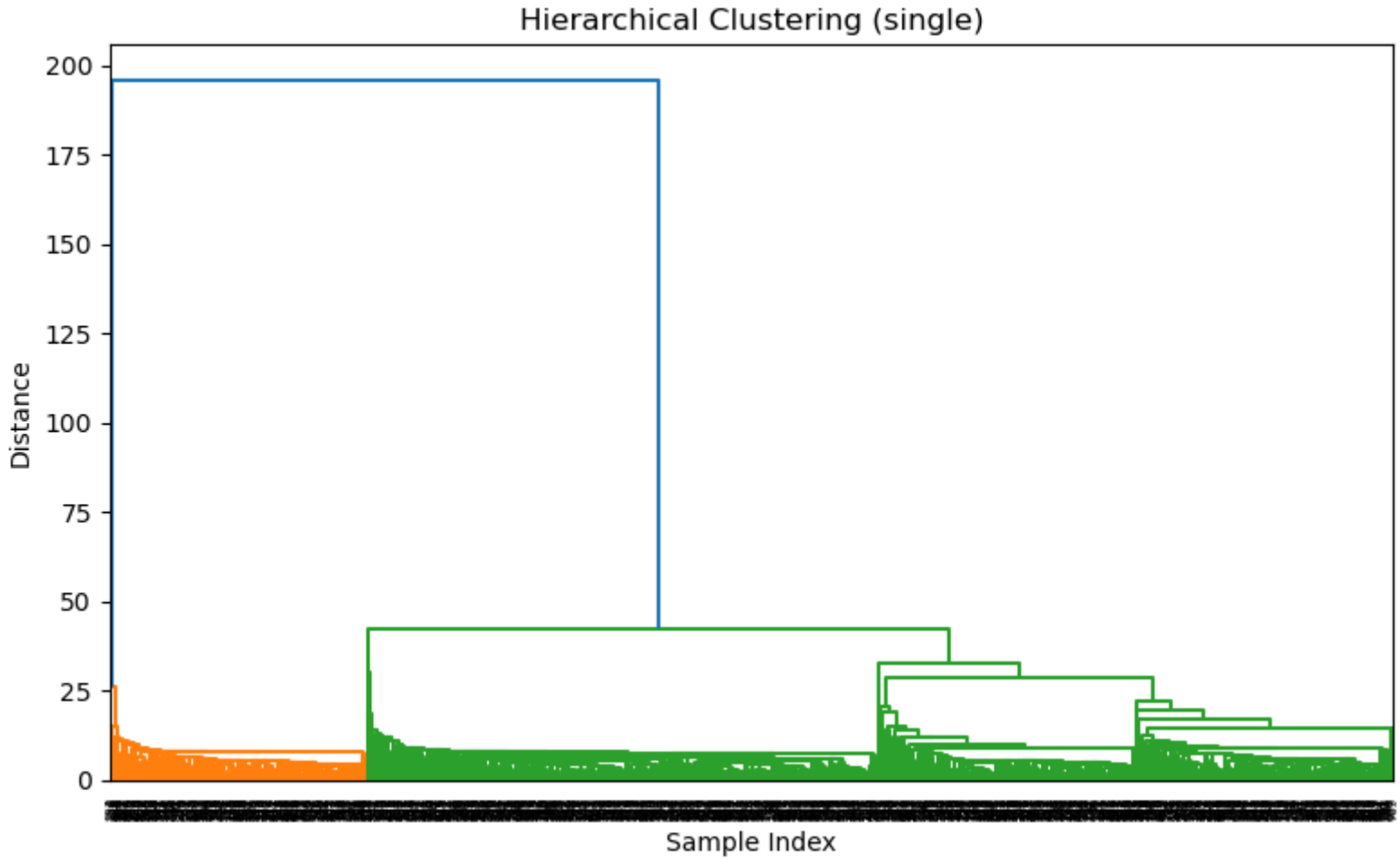
In summary, this exercise demonstrates the use of the MLE algorithm for clustering Gaussian data with various initialization methods. It allows for the comparison of different initialization techniques and their impact on the resulting clustering performance. The visualization helps visualize the cluster assignments for each initialization method, providing insights into the differences between them.

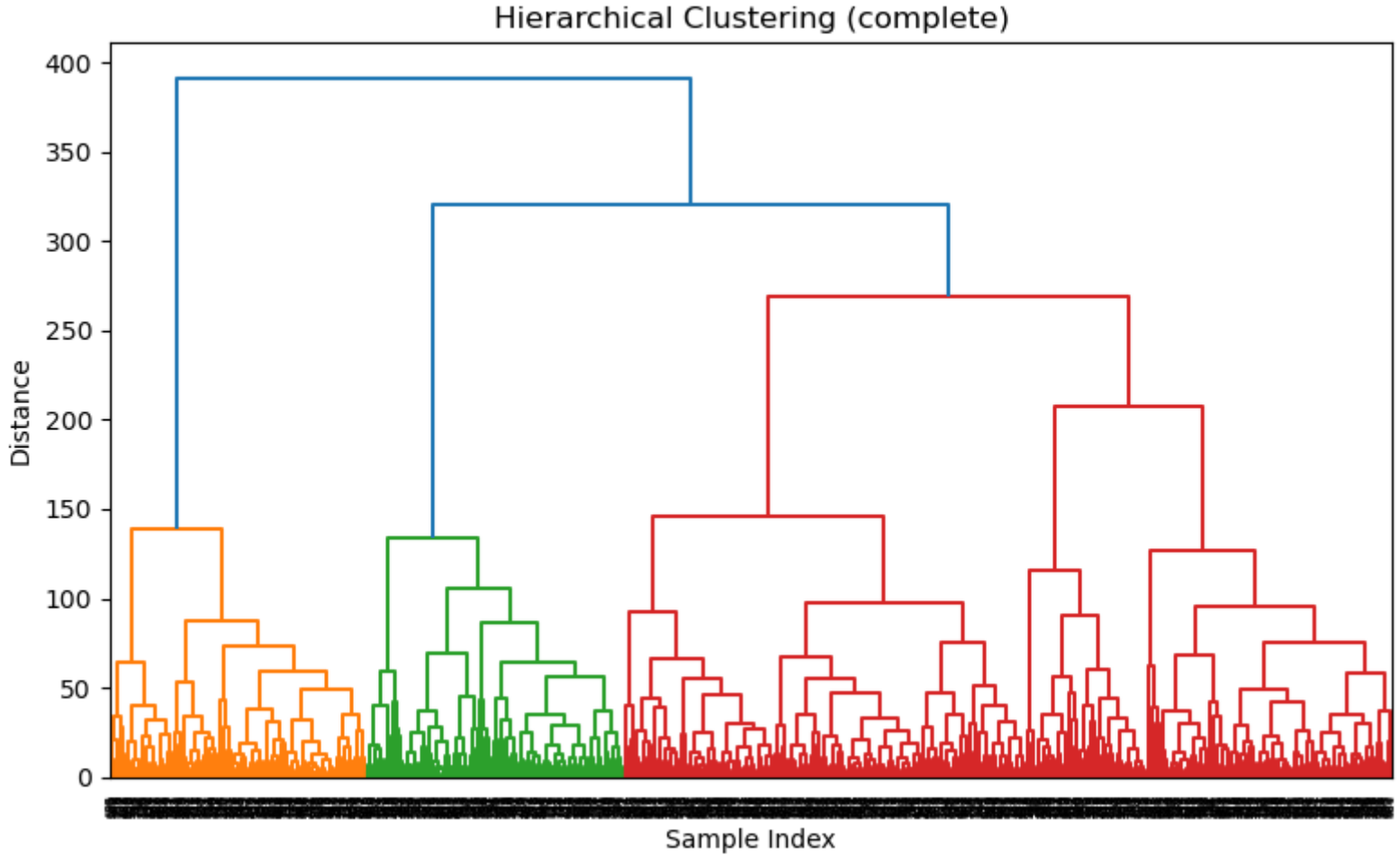
Exercise 3

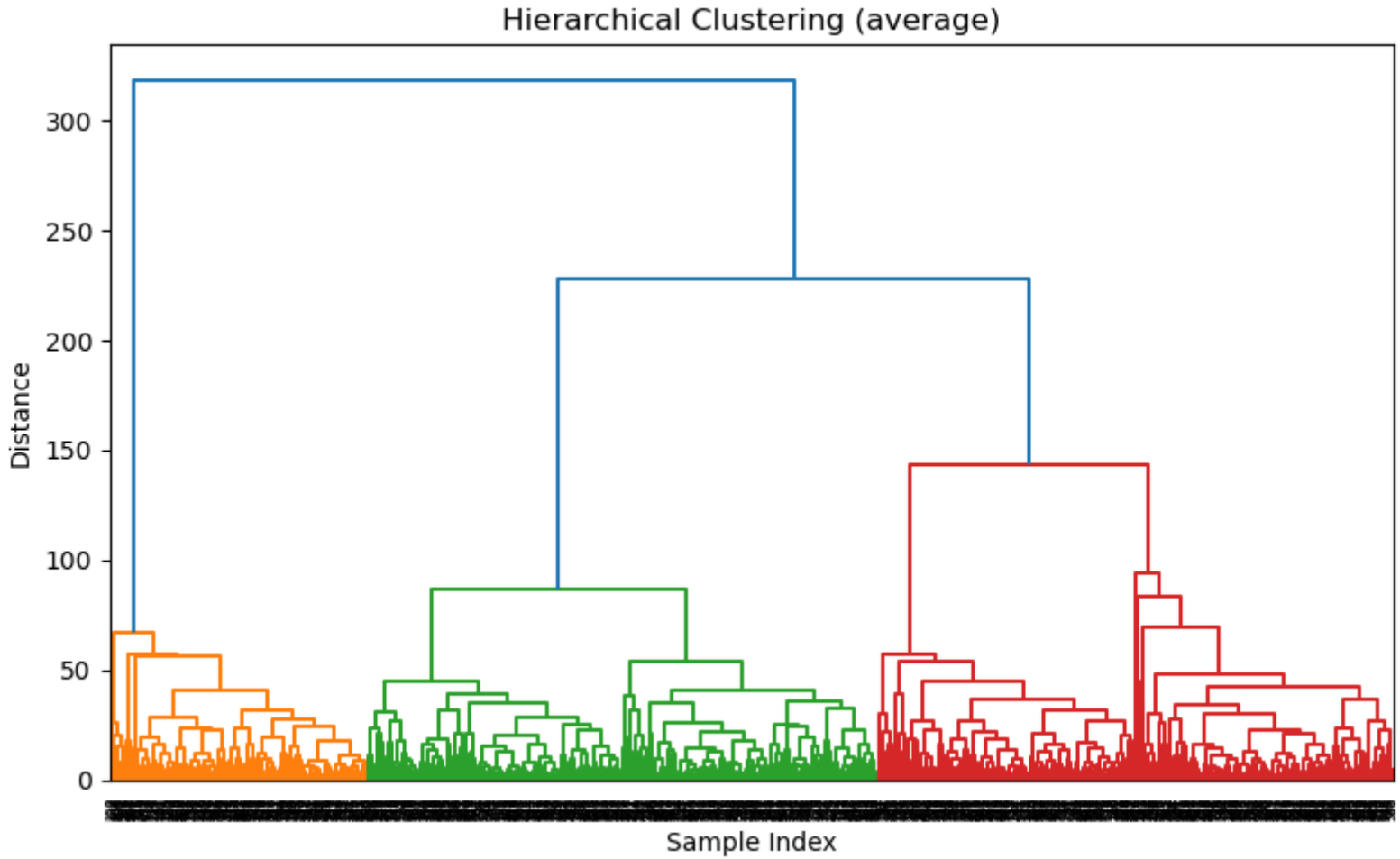
In [7]:

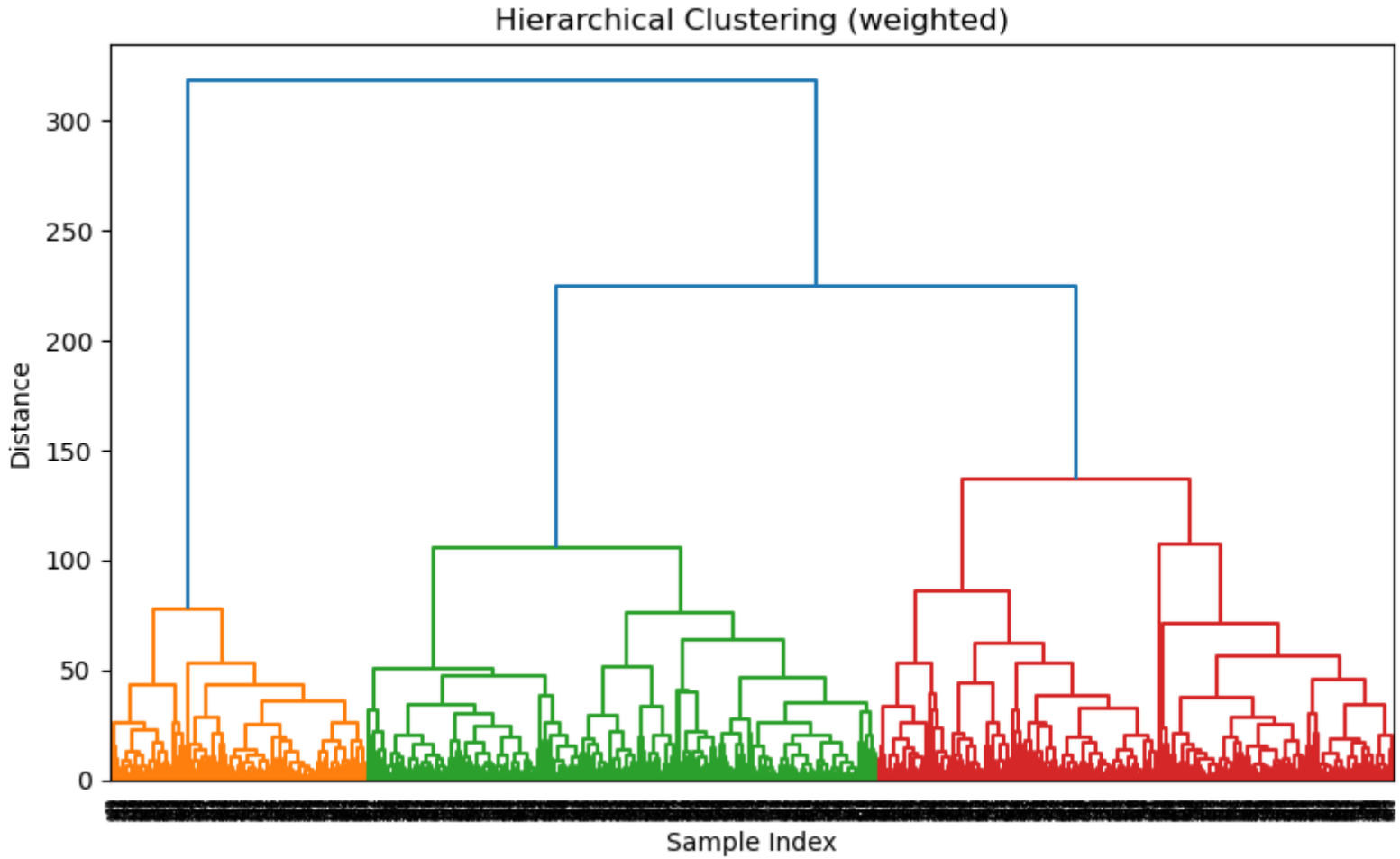
```
# Compute the pairwise distance matrix
dist_matrix = np.zeros((len(X), len(X)))
for i in range(len(X)):
    for j in range(len(X)):
        dist_matrix[i, j] = np.linalg.norm(X[i] - X[j])

# Define the linkage methods
linkage_methods = ['single', 'complete', 'average', 'weighted', 'centroid']

# Perform hierarchical clustering for each method
for i, method in enumerate(linkage_methods):
    # Compute the linkage matrix
    Z = linkage(dist_matrix, method=method)

    # Plot the dendrogram
    plt.figure(figsize=(8, 5))
    plt.title(f'Hierarchical Clustering ({method})')
    dendrogram(Z)
    plt.xlabel('Sample Index')
    plt.ylabel('Distance')
    plt.tight_layout()
    plt.show()
```

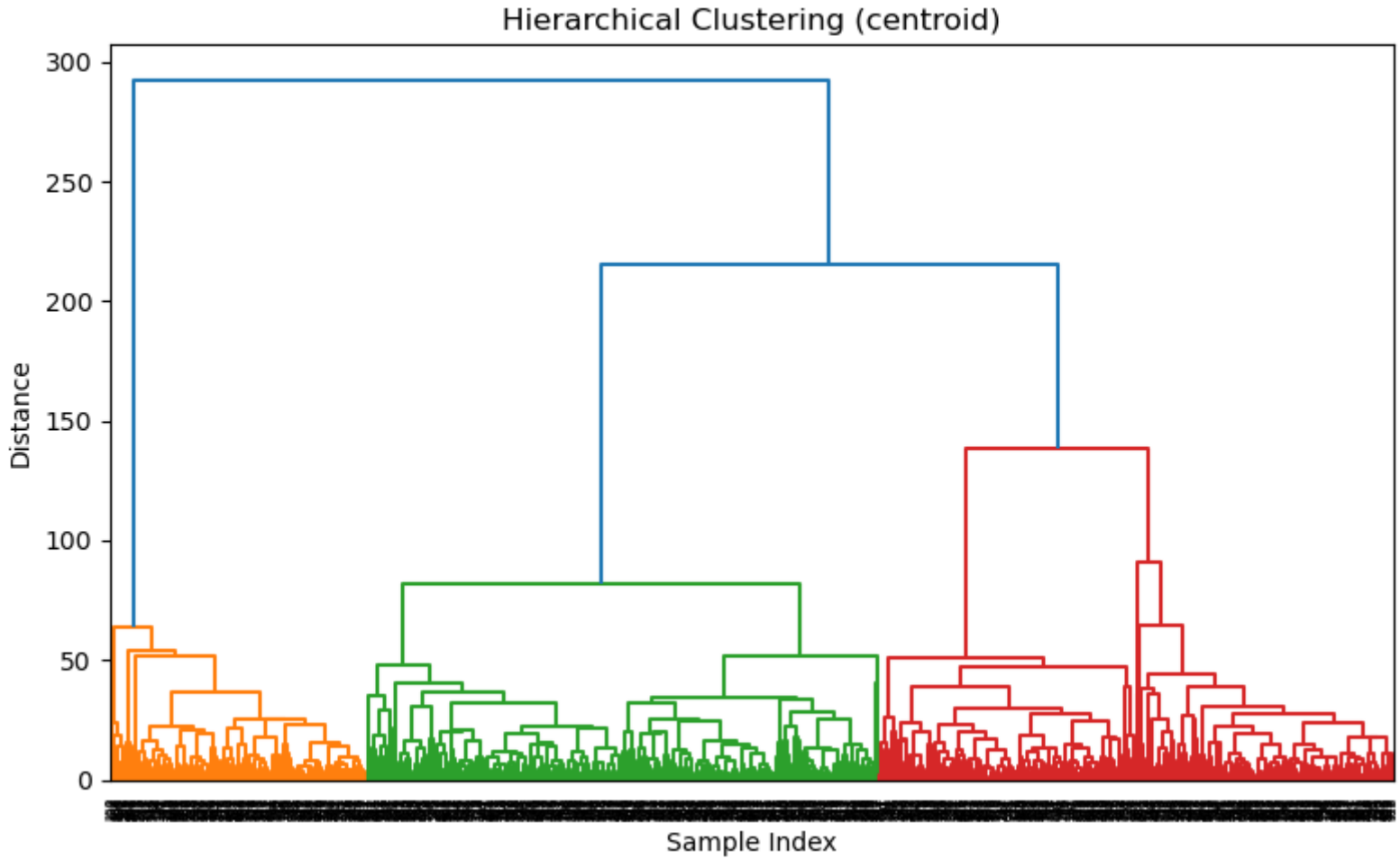
```
C:\Users\kazom\AppData\Local\Temp\ipykernel_11200\2375571947.py:13: ClusterWarning: scipy.cluster: The symmetric non-negative h
ollow observation matrix looks suspiciously like an uncondensed distance matrix
  Z = linkage(dist_matrix, method=method)
```


Hierarchical Clustering (single)
Distance
200
175
150
125
100
75
50
25
0
Sample Index


Hierarchical Clustering (complete)
Distance
400
350
300
250
200
150
100
50
0
Sample Index


Hierarchical Clustering (average)
Distance
300
250
200
150
100
50
0
Sample Index

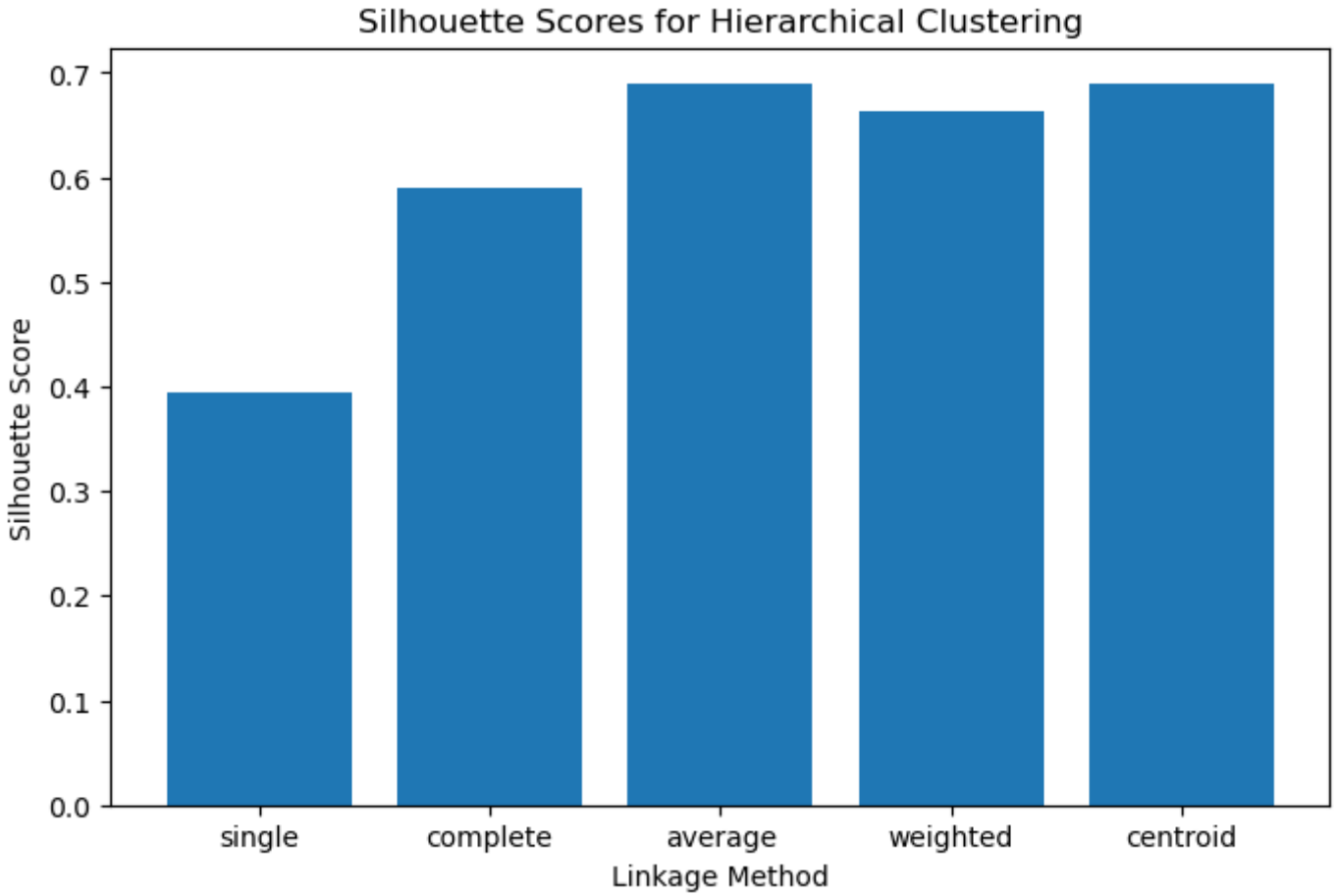

Hierarchical Clustering (weighted)
Distance
300
250
200
150
100
50
0
Sample Index


Hierarchical Clustering (centroid)
Distance
300
250
200
150
100
50
0
Sample Index

In [8]:
```
# Compute the pairwise distance matrix
dist_matrix = np.zeros((len(X), len(X)))
for i in range(len(X)):
    for j in range(len(X)):
        dist_matrix[i, j] = np.linalg.norm(X[i] - X[j])

# Define the linkage methods
linkage_methods = ['single', 'complete', 'average', 'weighted', 'centroid']

# Perform hierarchical clustering for each method
silhouette_scores = []
for method in linkage_methods:
    Z = linkage(dist_matrix, method=method)
    labels = fcluster(Z, t=5, criterion='maxclust')  # Change the threshold value if needed
    score = silhouette_score(X, labels)
    silhouette_scores.append(score)
    print(f"Silhouette score ({method}): {score}")

# Plot the silhouette scores
plt.figure(figsize=(8, 5))
plt.bar(linkage_methods, silhouette_scores)
plt.title("Silhouette Scores for Hierarchical Clustering")
plt.xlabel("Linkage Method")
plt.ylabel("Silhouette Score")
plt.show()
```

```
C:\Users\kazom\AppData\Local\Temp\ipykernel_11200\2302197127.py:13: ClusterWarning: scipy.cluster: The symmetric non-negative h
ollow observation matrix looks suspiciously like an uncondensed distance matrix
  Z = linkage(dist_matrix, method=method)

Silhouette score (single): 0.3941347360496806

C:\Users\kazom\AppData\Local\Temp\ipykernel_11200\2302197127.py:13: ClusterWarning: scipy.cluster: The symmetric non-negative h
ollow observation matrix looks suspiciously like an uncondensed distance matrix
  Z = linkage(dist_matrix, method=method)

Silhouette score (complete): 0.5907991520793088

C:\Users\kazom\AppData\Local\Temp\ipykernel_11200\2302197127.py:13: ClusterWarning: scipy.cluster: The symmetric non-negative h
ollow observation matrix looks suspiciously like an uncondensed distance matrix
  Z = linkage(dist_matrix, method=method)

Silhouette score (average): 0.6886472944811992
```

```
C:\Users\kazom\AppData\Local\Temp\ipykernel_11200\2302197127.py:13: ClusterWarning: scipy.cluster: The symmetric non-negative h
ollow observation matrix looks suspiciously like an uncondensed distance matrix
  Z = linkage(dist_matrix, method=method)

Silhouette score (weighted): 0.664004673452303

C:\Users\kazom\AppData\Local\Temp\ipykernel_11200\2302197127.py:13: ClusterWarning: scipy.cluster: The symmetric non-negative h
ollow observation matrix looks suspiciously like an uncondensed distance matrix
  Z = linkage(dist_matrix, method=method)

Silhouette score (centroid): 0.6894282786600593
```



First, we compute the pairwise distance matrix using the Euclidean distance measure. This matrix represents the distances between each pair of samples in the dataset.

We define a list of linkage methods: 'single', 'complete', 'average', 'weighted', and 'centroid'. These methods determine how the distances between clusters are calculated in hierarchical clustering.

For each linkage method, we perform hierarchical clustering by calling the linkage function from scipy with the pairwise distance matrix and the chosen linkage method. This generates a linkage matrix that represents the hierarchical structure of the clusters.

We then use the fcluster function from scipy to assign cluster labels based on the hierarchical clustering results. The fcluster function requires a threshold value (t) to determine the number of clusters. You can adjust this value based on your preference or problem requirements.

After obtaining the cluster labels, we calculate the silhouette score for each method using the silhouette_score function from scikit-learn. The silhouette score measures the quality of clustering, where a higher score indicates better clustering.

We store the silhouette scores for each method in the silhouette_scores list and print them.

Finally, we plot a bar chart showing the silhouette scores for each linkage method. This visualization allows us to compare the clustering performance of different methods. A higher silhouette score indicates that the clusters are well-separated and compact.

In summary, the code performs hierarchical clustering using different linkage methods and evaluates the quality of clustering using the silhouette score. The silhouette scores provide insights into the effectiveness of each clustering method in partitioning the data into meaningful clusters. The bar chart helps to visualize and compare the clustering performance across different methods.