

Machine Learning Project

presented by:

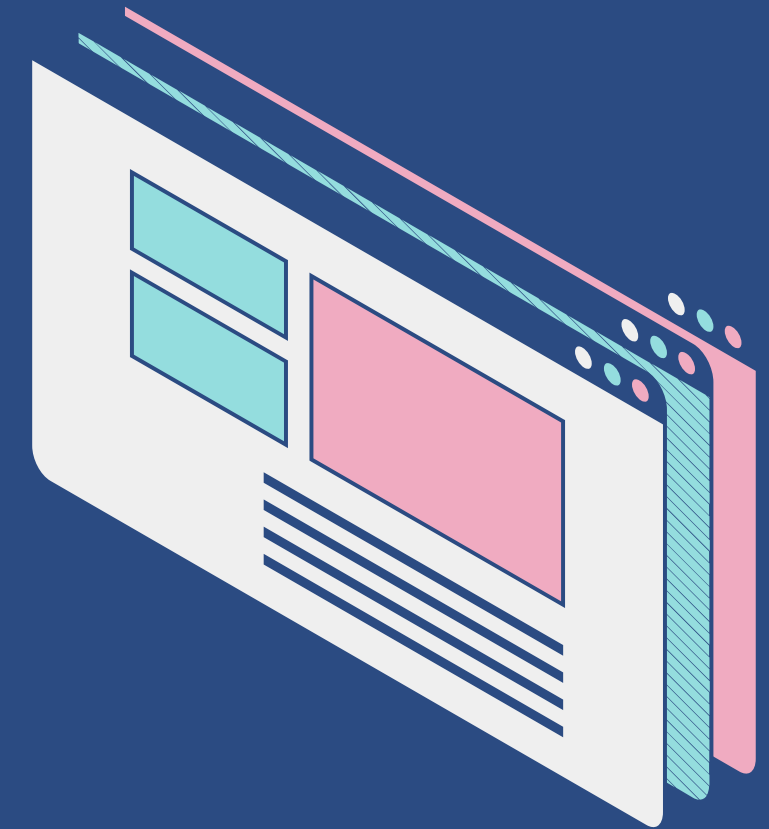
Maryam Aldahri-2110643

Rimas Alshehri-2110240

Asmaa Mahdi-2111900

Yara Alshehri-2113140

Taif Alharbi-2111458



| Contents

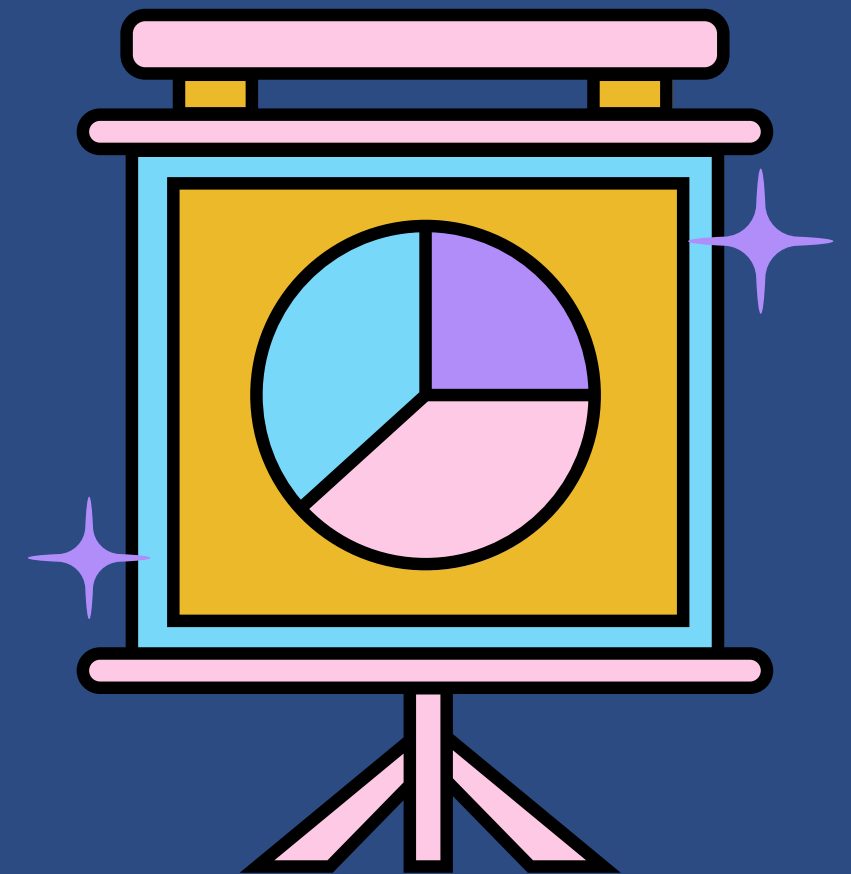
MACHINE LEARNING PROJECT

Datasets

Algorithms

Comparison

Clustering analysis



MACHINE LEARNING PROJECT



Datasets



Datasets code



code walkthrough

This code generates and plots four different synthetic datasets using scikit-learn's datasets module. Let's break down each part of the code:

`n_samples`: Specifies the number of samples in each dataset.

`random_state`: Sets the random seed for reproducibility.

Generates a synthetic dataset with blobs of points using the `make_blobs` function.

`X1`: Contains the data points.

`y1`: Contains the labels.

Generates a synthetic dataset with blobs using `make_blobs`. Applies an anisotropic transformation to the dataset by multiplying it with a 2x2 transformation matrix.

`X2`: Contains the transformed data points.

`y2`: Contains the labels.

Generates a synthetic dataset with points arranged in the shape of two interleaving half-moons using the `make_moons` function.

Introduces some noise to the dataset.

`X3`: Contains the data points.

`y3`: Contains the labels.

Generates a synthetic dataset with points arranged in the shape of two circles using the `make_circles` function.

Introduces some noise and controls the size of the inter-circular region with the `factor` parameter.

`X4`: Contains the data points.

`y4`: Contains the labels.

```
n_samples = 300
random_state = 75

# Dataset 1: Blobs dataset
X1, y1 = datasets.make_blobs(n_samples=n_samples, random_state=random_state)

# Dataset 2: Anisotropically distributed dataset
X2, y2 = datasets.make_blobs(n_samples=n_samples, random_state=random_state)
transformation = [[0.6, -0.6], [-0.4, 0.8]]
X2 = np.dot(X2, transformation)

# Dataset 3: Noisy moons dataset
X3, y3 = datasets.make_moons(n_samples=n_samples, noise=0.1, random_state=random_state)

# Dataset 4: Noisy circles dataset
X4, y4 = datasets.make_circles(n_samples=n_samples, factor=.5, noise=.05, random_state=random_state)

# Plotting the datasets
plt.figure(figsize=(12, 4))

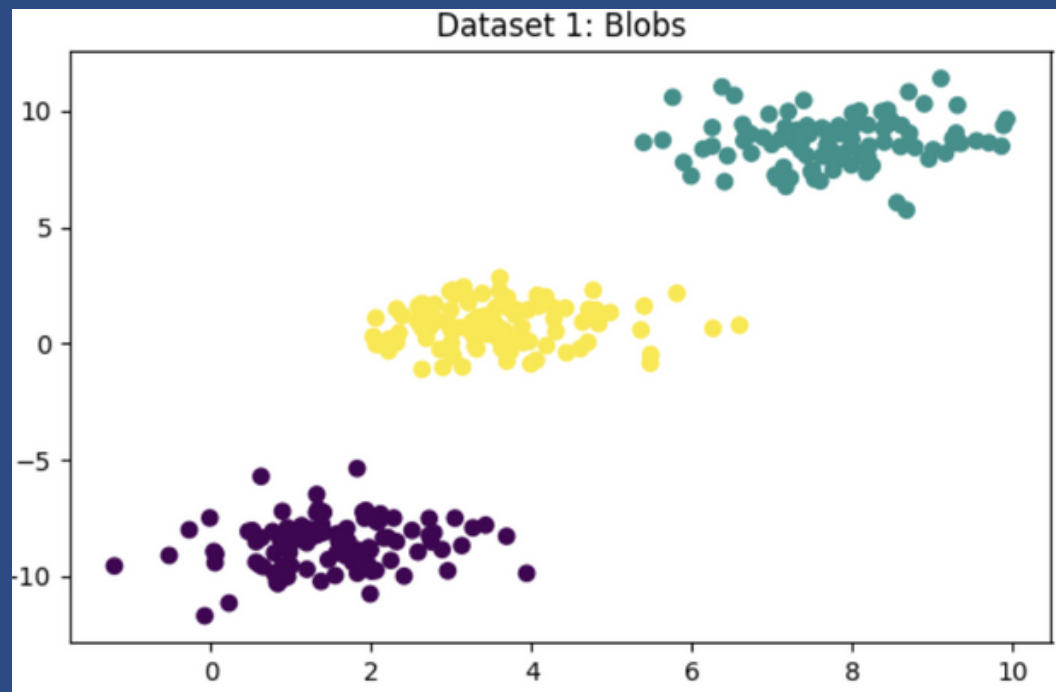
plt.subplot(141)
plt.scatter(X1[:, 0], X1[:, 1], c=y1, cmap='viridis')
plt.title('Dataset 1: Blobs')

plt.subplot(142)
plt.scatter(X2[:, 0], X2[:, 1], c=y2, cmap='viridis')
plt.title('Dataset 2: Anisotropic Distribution')

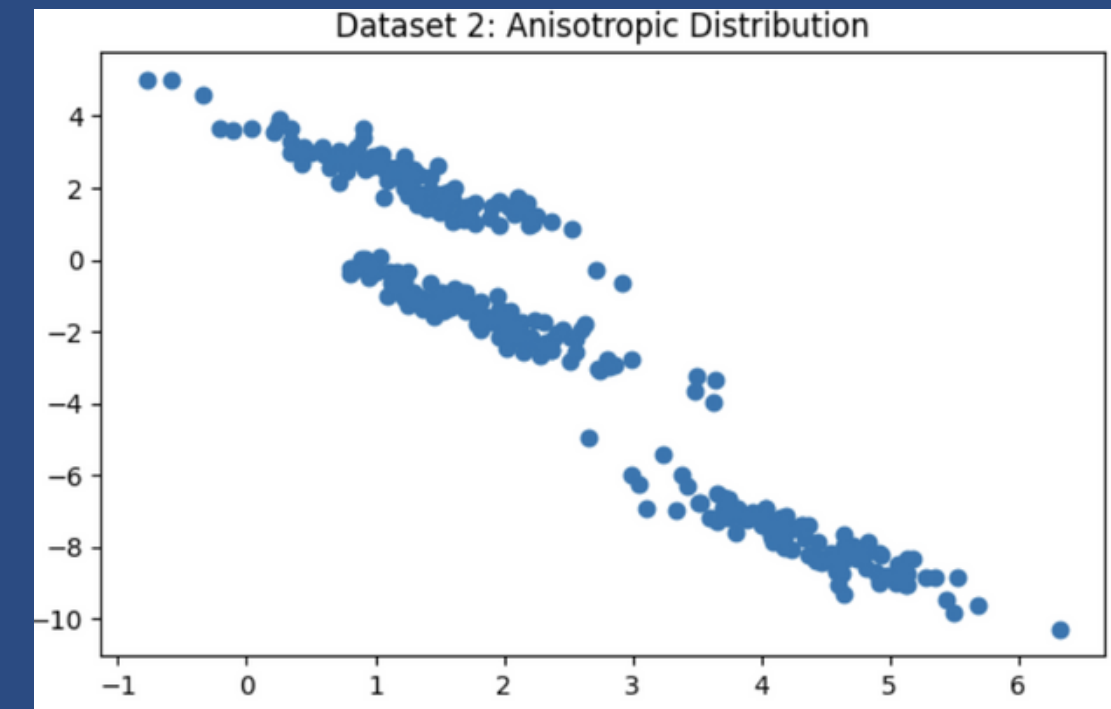
plt.subplot(143)
plt.scatter(X3[:, 0], X3[:, 1], c=y3, cmap='viridis')
plt.title('Dataset 3: Noisy Moons')

plt.subplot(144)
plt.scatter(X4[:, 0], X4[:, 1], c=y4, cmap='viridis')
plt.title('Dataset 4: Noisy Circles')
```

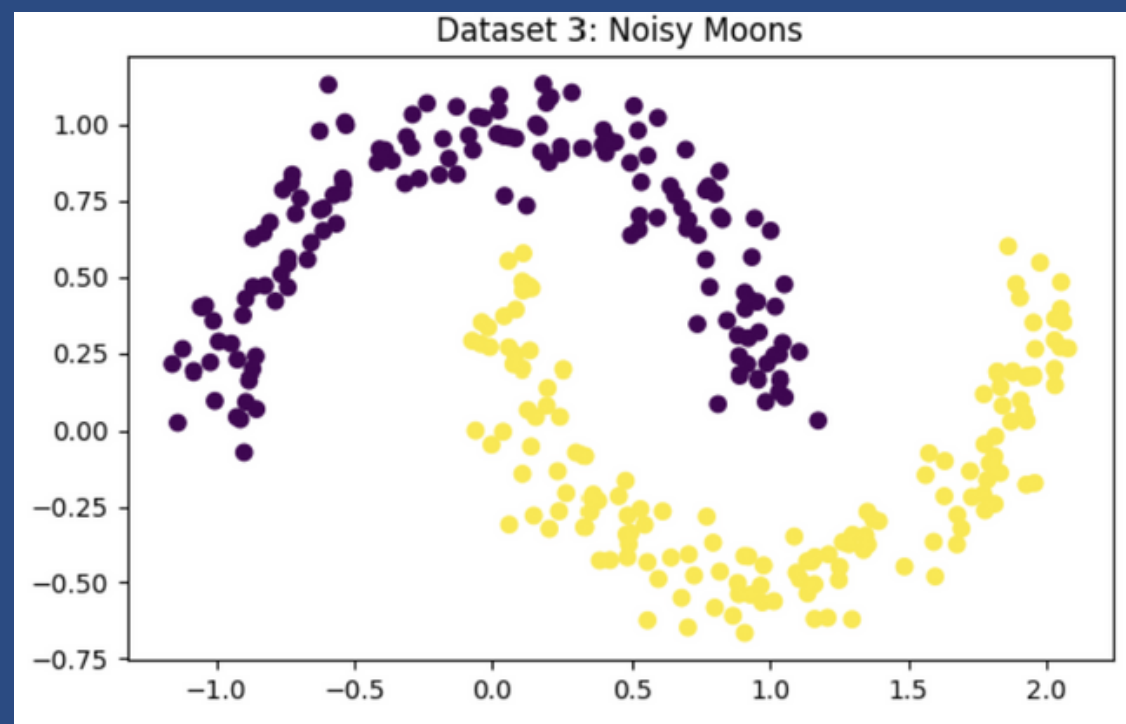
blobs dataset



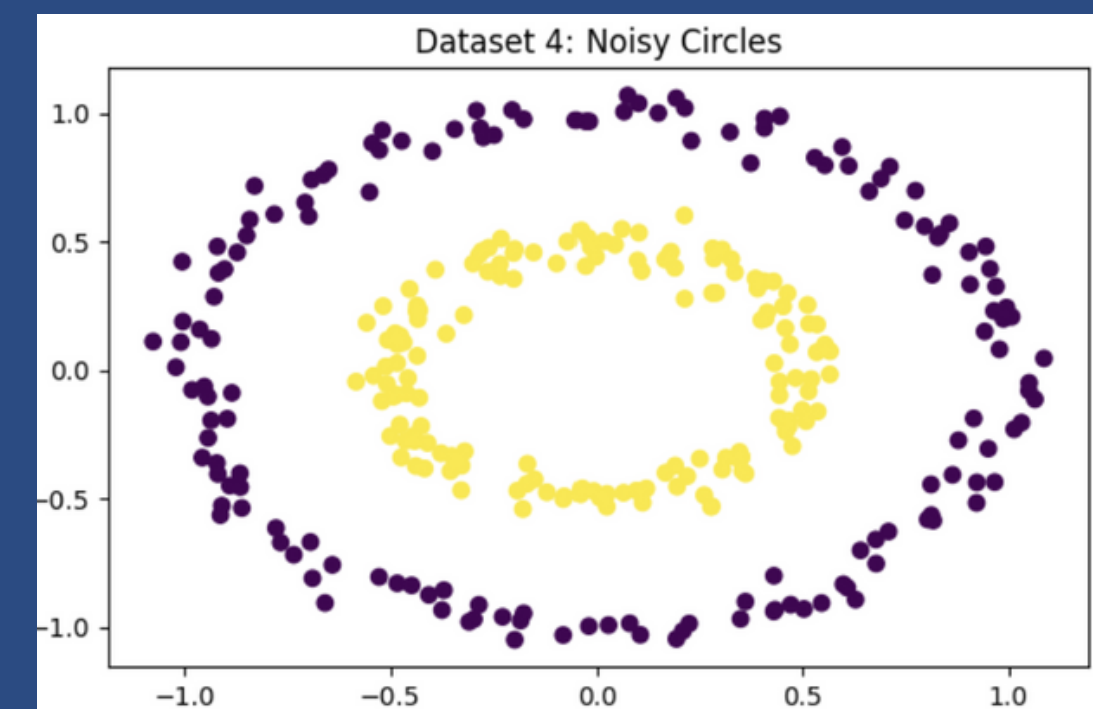
Anisotropically distributed dataset



noisy moons dataset



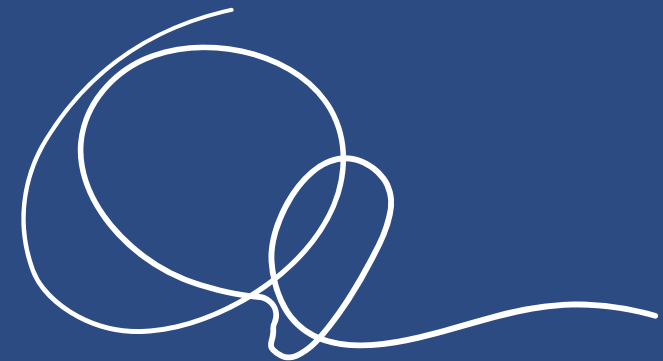
noisy circles dataset



MACHINE LEARNING PROJECT



GMM & Kmeans



Kmeans Algorithm

unsupervised machine learning algorithm used for clustering analysis.

aims to partition a given dataset into a predetermined number of clusters, where each data point belongs to the cluster with the nearest mean (centroid)

algorithm iteratively assigns data points to the nearest centroid and recalculates the centroids based on the assigned points until convergence or a predefined number of iterations

GMM Algorithm

The Gaussian Mixture Model (GMM) is an unsupervised machine learning algorithm utilized for clustering analysis.

The Gaussian Mixture Model (GMM) is an unsupervised algorithm that represents a dataset as a mix of Gaussian distributions.

Its goal is to group the data into predetermined clusters, with each cluster linked to a Gaussian component. GMM assigns data points to clusters based on the likelihood of belonging to each Gaussian distribution, offering a probabilistic and flexible way to handle complex data structures.

The iterative process involves an Expectation-Maximization (EM) algorithm, where in the E-step, the algorithm computes the probability of each data point belonging to each cluster, and in the M-step, it updates the parameters (means, covariances, and weights) based on these probabilities. This process repeats iteratively until convergence or a predefined number of iterations, akin to the iterative assignment and centroid recalculation steps in the KMeans algorithm

MACHINE LEARNING PROJECT

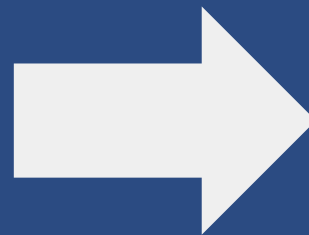


Kmeans Algorithm



Kmeans methods

```
def best_number_k(data, cluster_range, title):
    y_mean_list = []
    sse_list = []
    for k in cluster_range:
        y_mean, sse = Kmean(data, k)
        y_mean_list.append(y_mean)
        sse_list.append(sse)
    fig = plt.figure(figsize=(6, 6))
    plt.plot(cluster_range, sse_list, 'bx-')
    plt.xlabel('(k)')
    plt.ylabel('(SSE)')
    plt.title(title)
    plt.show()
    return y_mean_list, sse_list
```



```
def Kmean(X, k):
    random_centroid = random.sample(range(0, X.shape[0]), k)
    centroid = X[random_centroid]
    for _ in range(50):
        # Assign clusters
        cluster_groups = assign_point_clusters(X, centroid)

        # Move centroids
        old_centroid = centroid
        centroid = move_centroid_based_on_cluster_group(X, cluster_groups)
        # Check convergence
        check_convergence = np.all(old_centroid == centroid)
        if check_convergence: # if true break
            # Calculate SSE
            sse = calculate_sse(X, centroid, cluster_groups)
            break

    return cluster_groups, sse
```

Kmeans methods

After calling the Kmean function, the process starts by initializing the centroids randomly. Then the assign_point_clusters function is called to assign each data point in X to its nearest centroid. It takes X and current centroid locations as inputs and returns the assignment of data points to clusters.

(1)

```
def assign_point_clusters(X, centroid):  
    cluster_group = []  
    for point in X:  
        distances = []  
        for i in centroid:  
            distances.append(np.linalg.norm(point - i))  
        min_distance = min(distances)  
        index_pos = distances.index(min_distance)  
        cluster_group.append(index_pos)  
    return np.array(cluster_group)
```



the move_centroid_based_on_cluster_group function is called to update the centroid locations based on the current assignment of data points to clusters. It takes X and cluster_groups as inputs and returns the updated centroid locations.

(2)

```
def move_centroid_based_on_cluster_group(X, cluster_group):  
    new_centroid = []  
    cluster_types = np.unique(cluster_group)  
    for cluster_type in cluster_types:  
        new_centroid.append(X[cluster_group == cluster_type].mean(axis=0))  
    return np.array(new_centroid)
```



If the centroids have converged, the code breaks out of the loop and proceeds to calculate the SSE using the calculate_sse function. It takes X, final centroid locations, and cluster_groups as inputs and returns the SSE.

(3)

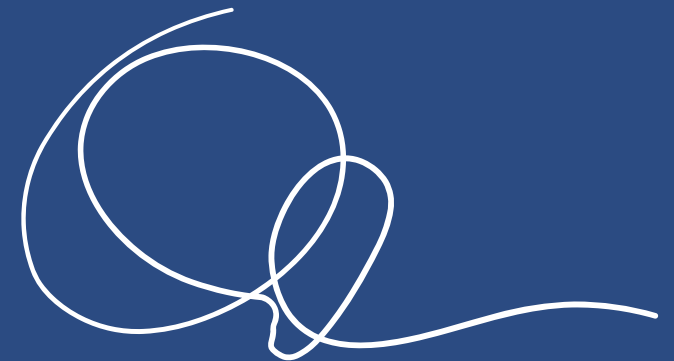
```
def calculate_sse(X, centroid, cluster_groups):  
    sse = 0  
    for i in range(len(X)):  
        cluster_type = cluster_groups[i]  
        centroid_i = centroid[cluster_type]  
        squared_distance = np.sum((X[i] - centroid_i)** 2)  
        sse += squared_distance  
    return sse
```



MACHINE LEARNING PROJECT



GMM Algorithm



GMM methods

```
class GMM:
    def __init__(self, n_clusters, max_iter=100, tol=1e-4):
        self.n_clusters = n_clusters
        self.max_iter = max_iter
        self.tol = tol
```



class GMM:: Defines a Python class named GMM.

def __init__(self, n_clusters, max_iter=100, tol=1e-4):: Defines the constructor method for the GMM class.

self.n_clusters = n_clusters: Initializes the number of clusters attribute.

self.max_iter = max_iter: Initializes the maximum number of iterations attribute for the Expectation-Maximization (EM) algorithm.

self.tol = tol: Initializes the tolerance for convergence attribute.

Purpose:

Defines a class GMM to implement the Gaussian Mixture Model.

Initializes the GMM with the number of clusters (n_clusters), maximum iterations for the EM algorithm (max_iter), and convergence tolerance (tol).

```
def initialize_parameters(self, X):
    self.n_samples, self.n_features = X.shape

    # Initialize cluster weights, means, and covariances
    self.weights = np.ones(self.n_clusters) / self.n_clusters
    self.means = np.random.choice(np.arange(self.n_samples), size=self.n_clusters, replace=False)
    self.covariances = np.array([np.cov(X.T) for _ in range(self.n_clusters)])
```



def initialize_parameters(self, X):: Defines a method to initialize the parameters of the GMM.

self.n_samples, self.n_features = X.shape: Gets the number of samples and features from the input data X.

self.weights = np.ones(self.n_clusters) / self.n_clusters: Initializes cluster weights with equal values.

self.means = np.random.choice(np.arange(self.n_samples), size=self.n_clusters, replace=False): Initializes cluster means by randomly choosing sample indices without replacement.

self.covariances = np.array([np.cov(X.T) for _ in range(self.n_clusters)]): Initializes cluster covariances using the covariance matrix of the transposed input data.

Purpose:

Defines a method initialize_parameters to set up initial parameters for the GMM.

Computes the number of samples and features in the input data X.
Initializes cluster weights, means, and covariances using random sampling and the covariance matrix.

```
def e_step(self, X):
    # Compute responsibilities
    self.responsibilities = np.zeros((self.n_samples, self.n_clusters))
    for k in range(self.n_clusters):
        self.responsibilities[:, k] = self.weights[k] * self.compute_probability(X, self.means[k], self.covariances[k])
    self.responsibilities /= np.sum(self.responsibilities, axis=1)[:, np.newaxis]
```



def e_step(self, X):: Defines a method for the Expectation (E) step of the EM algorithm.

self.responsibilities = np.zeros((self.n_samples, self.n_clusters)):
Initializes a matrix to store responsibilities.

for k in range(self.n_clusters):: Iterates over each cluster.

self.responsibilities[:, k] = self.weights[k] * self.compute_probability(X, self.means[k], self.covariances[k]): Calculates the responsibility of each data point for the current cluster.

self.responsibilities /= np.sum(self.responsibilities, axis=1)[:, np.newaxis]: Normalizes the responsibilities.



Purpose:

Defines a method e_step to perform the Expectation (E) step of the EM algorithm.

Computes the responsibilities of each data point for each cluster based on the current parameters.




```
def m_step(self, X):  
    # Update cluster weights, means, and covariances  
    self.weights = np.mean(self.responsibilities, axis=0)  
    self.means = np.dot(self.responsibilities.T, X) / np.sum(self.responsibilities, axis=0)[:, np.newaxis]  
    for k in range(self.n_clusters):  
        diff = X - self.means[k]  
        self.covariances[k] = np.dot(self.responsibilities[:, k] * diff.T, diff) / np.sum(self.responsibilities[:, k])
```



def m_step(self, X):: Defines a method for the Maximization (M) step of the EM algorithm.

self.weights = np.mean(self.responsibilities, axis=0): Updates cluster weights based on the mean of responsibilities.

self.means = np.dot(self.responsibilities.T, X) / np.sum(self.responsibilities, axis=0)[: , np.newaxis]: Updates cluster means based on the weighted average of data points.

for k in range(self.n_clusters):: Iterates over each cluster.


diff = X - self.means[k]: Calculates the difference between the data and the mean of the current cluster.

self.covariances[k] = np.dot(self.responsibilities[:, k] * diff.T, diff) / np.sum(self.responsibilities[:, k]): Updates cluster covariances based on the weighted average of outer products of differences.

Purpose:

Defines a method m_step to perform the Maximization (M) step of the EM algorithm.

Updates cluster weights, means, and covariances based on the responsibilities and input data.



```
def compute_probability(self, X, mean, covariance):  
    # Compute probability of data point given mean and covariance  
    exponent = -0.5 * np.sum(np.dot(X - mean, np.linalg.inv(covariance)) * (X - mean), axis=1)  
    return (1 / np.sqrt(np.linalg.det(covariance))) * np.exp(exponent)
```



`def compute_probability(self, X, mean, covariance):`:: Defines a method to compute the probability of a data point given a mean and covariance for a specific cluster.

`exponent = -0.5 * np.sum(np.dot(X - mean, np.linalg.inv(covariance)) * (X - mean), axis=1)`: Calculates the exponent of the multivariate Gaussian distribution.

`return (1 / np.sqrt(np.linalg.det(covariance))) * np.exp(exponent)`: Computes the probability using the determinant of the covariance matrix.

Purpose:

Defines a method `compute_probability` to calculate the probability of a data point given a mean and covariance for a specific cluster.



```
def fit(self, X):
    self.initialize_parameters(X)

    for iteration in range(self.max_iter):
        old_means = self.means.copy()

        self.e_step(X)
        self.m_step(X)

        # Check convergence
        if np.linalg.norm(self.means - old_means) < self.tol:
            print(f"Converged after {iteration + 1} iterations.")
            break

    return self.responsibilities
```

def fit(self, X):: Defines a method to fit the GMM to the input data using the EM algorithm.

self.initialize_parameters(X): Initializes GMM parameters.

for iteration in range(self.max_iter):: Iterates over a specified number of iterations.

old_means = self.means.copy(): Stores the current cluster means.

self.e_step(X): Performs the E-step.

self.m_step(X): Performs the M-step.

if np.linalg.norm(self.means - old_means) < self.tol:: Checks for convergence by comparing the change in means.

print(f"Converged after {iteration + 1} iterations."): Prints a convergence message.

break: Exits the loop if convergence is achieved.

return self.responsibilities: Returns the responsibilities of each data point for each cluster.

Purpose:

Defines a method fit to fit the GMM to the input data using the EM algorithm. Iteratively performs E-step and M-step until convergence or reaching the maximum number of iterations.

Checks for convergence using the change in cluster means.

These cells together provide a complete implementation of a Gaussian Mixture Model with the Expectation-Maximization algorithm for clustering.

MACHINE LEARNING PROJECT

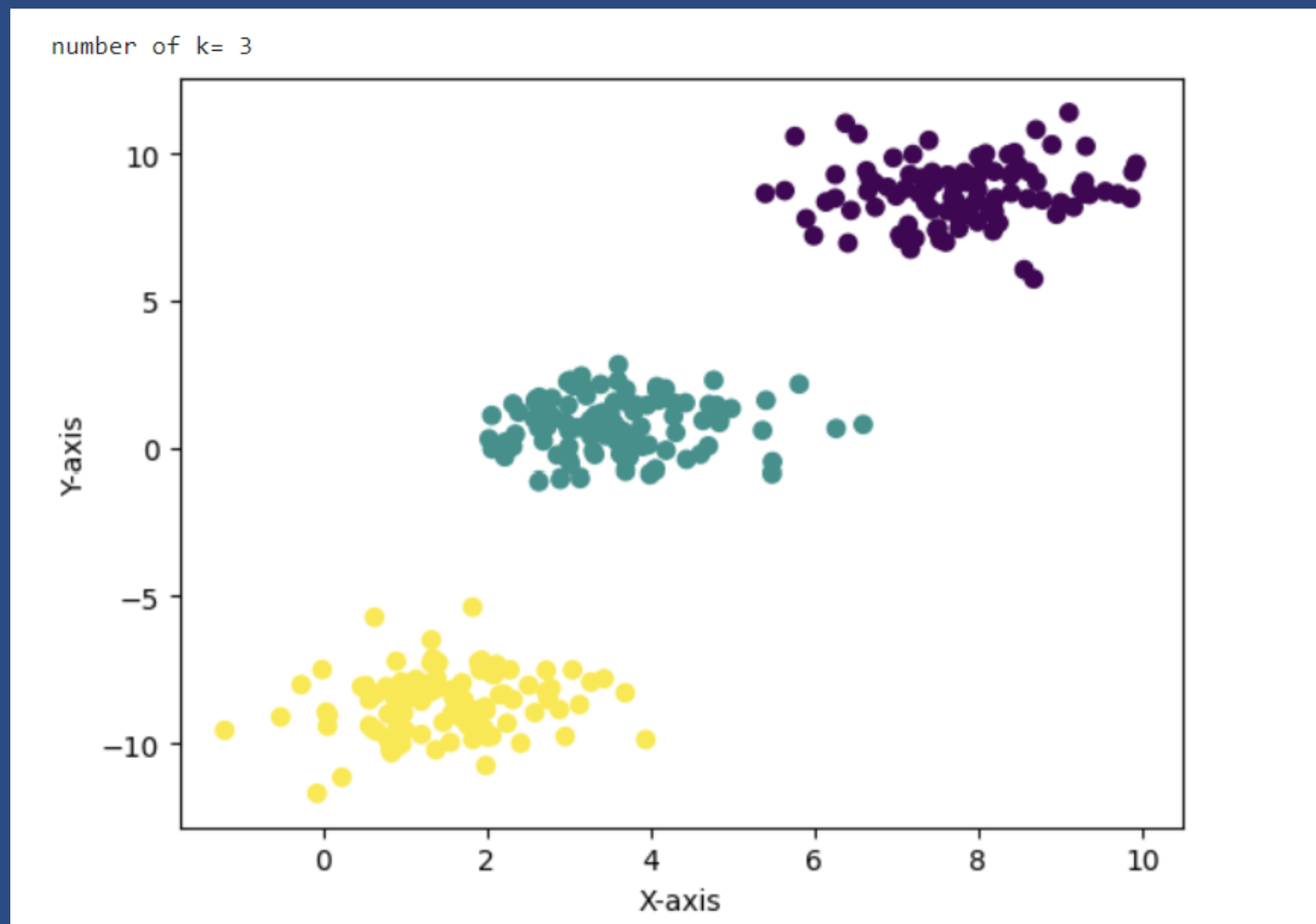


Comparison

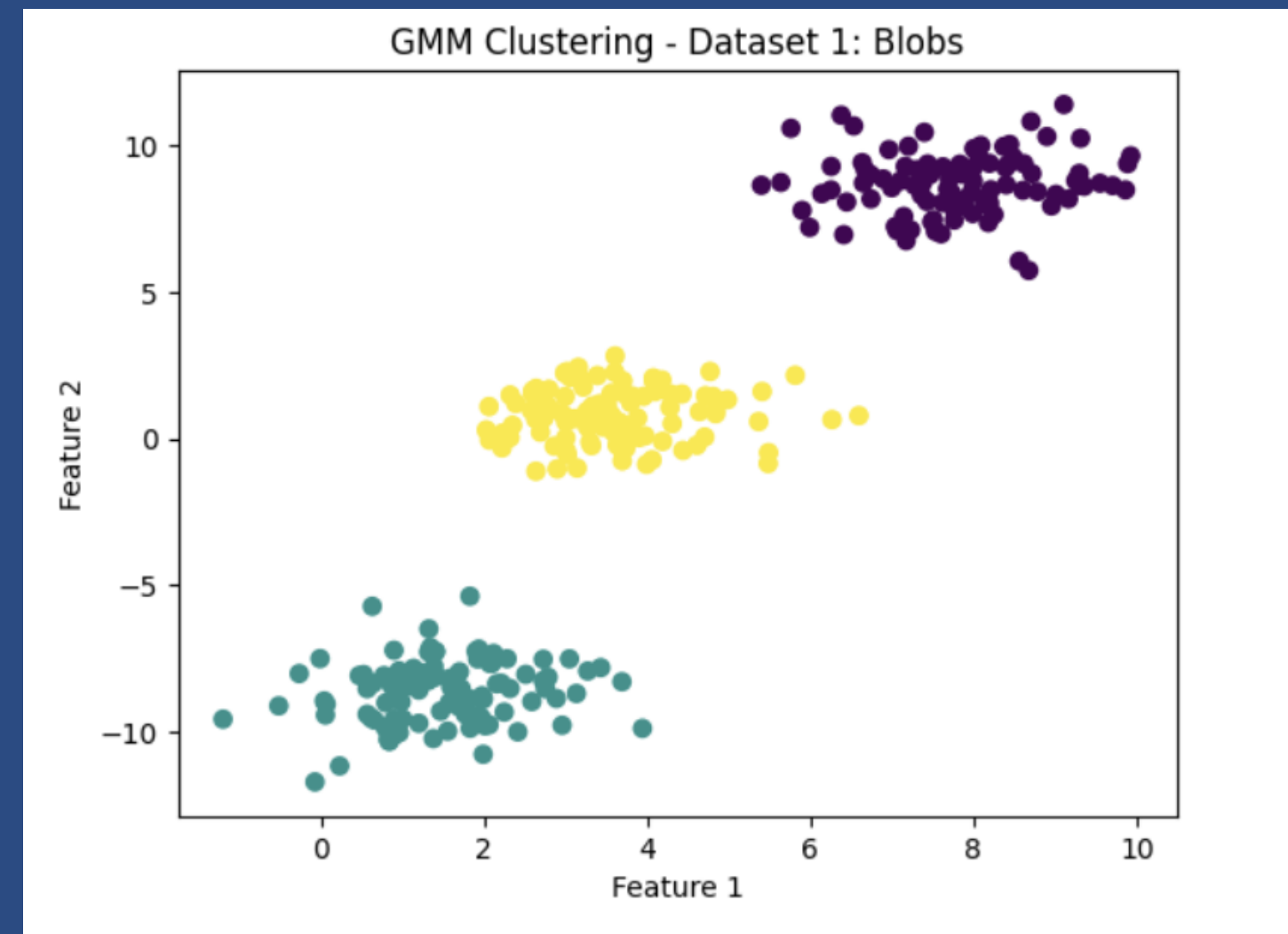




Kmean-Dataset 1

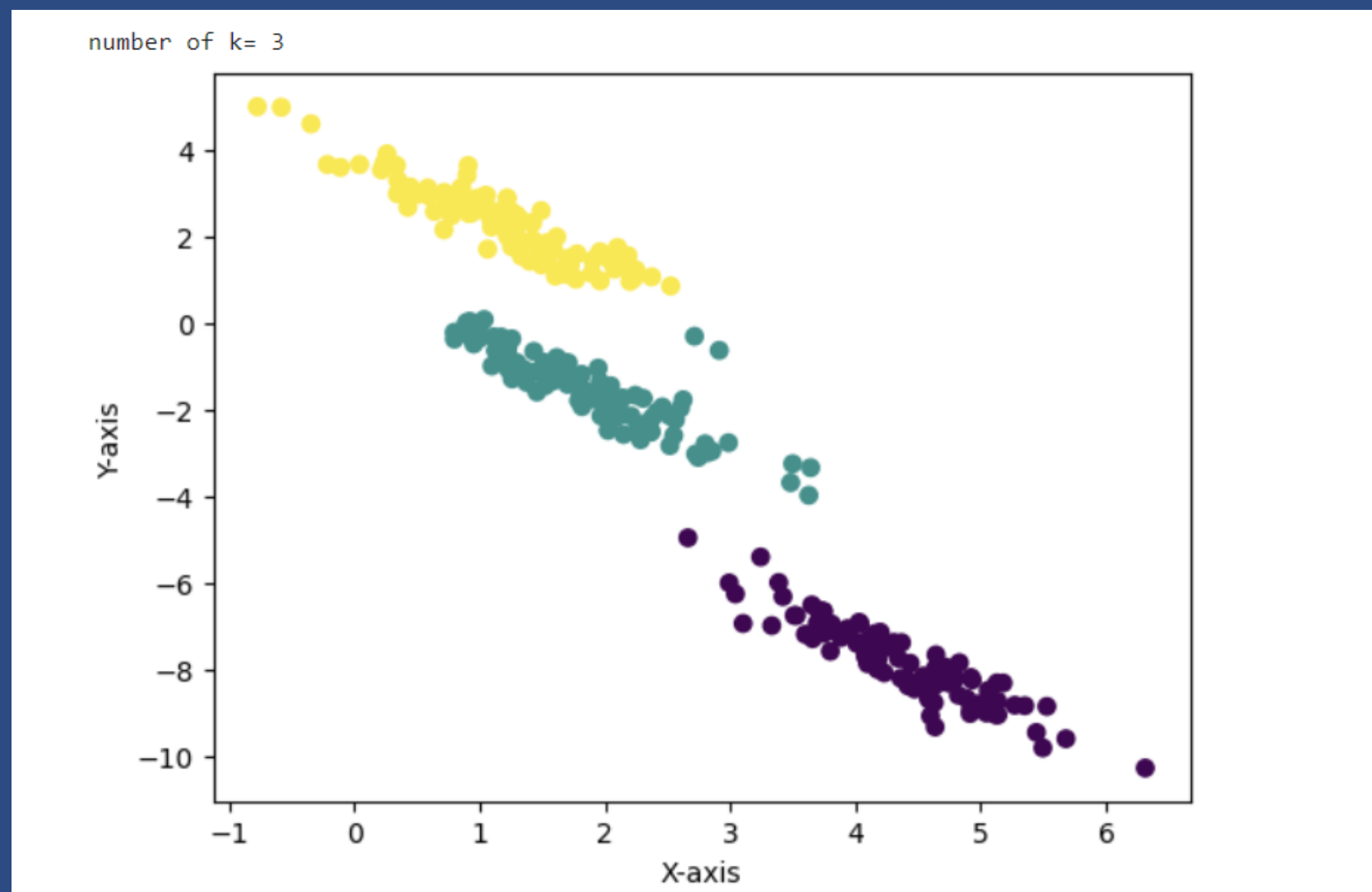


Gmm-Dataset 1

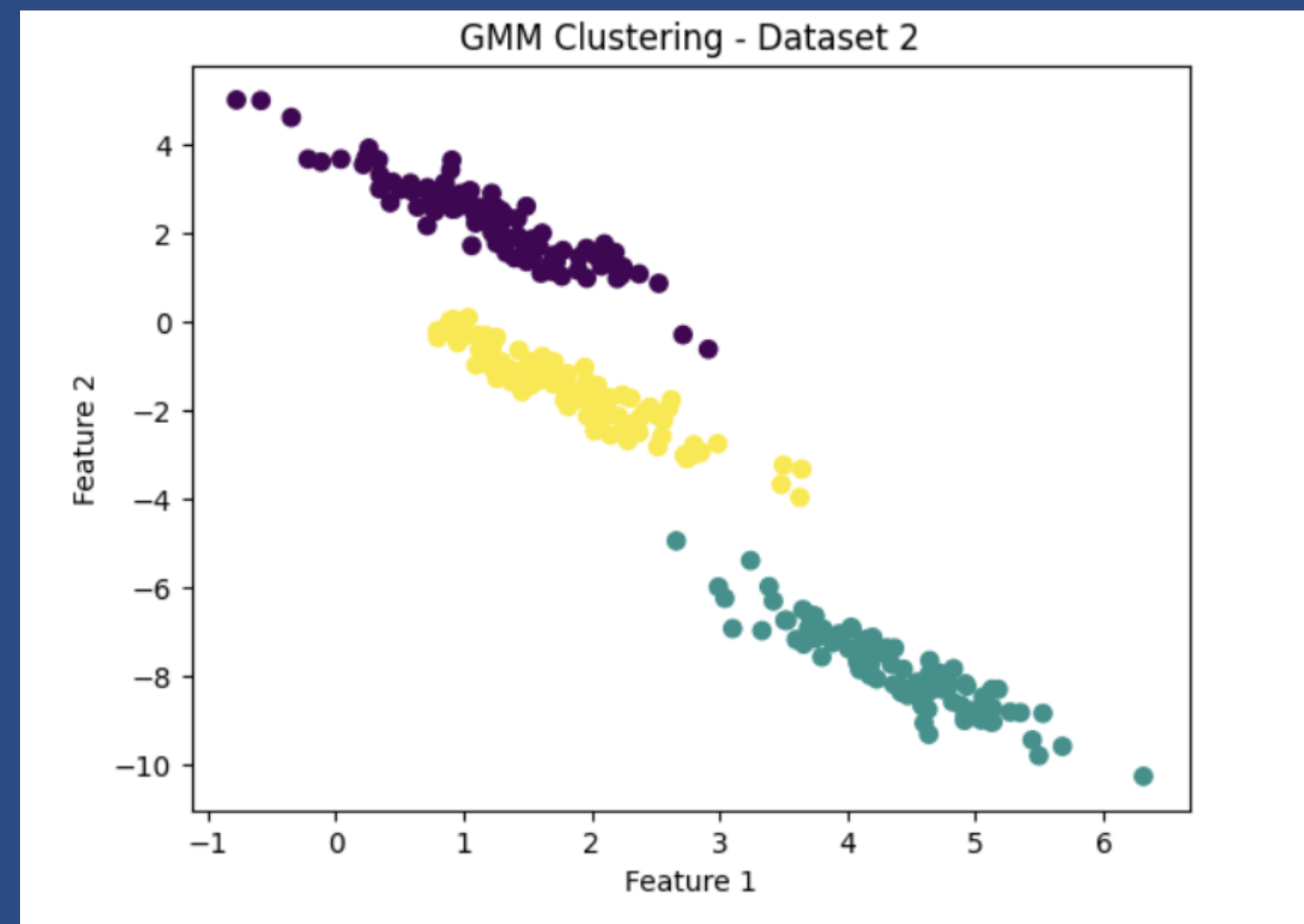




Kmean-Dataset 2

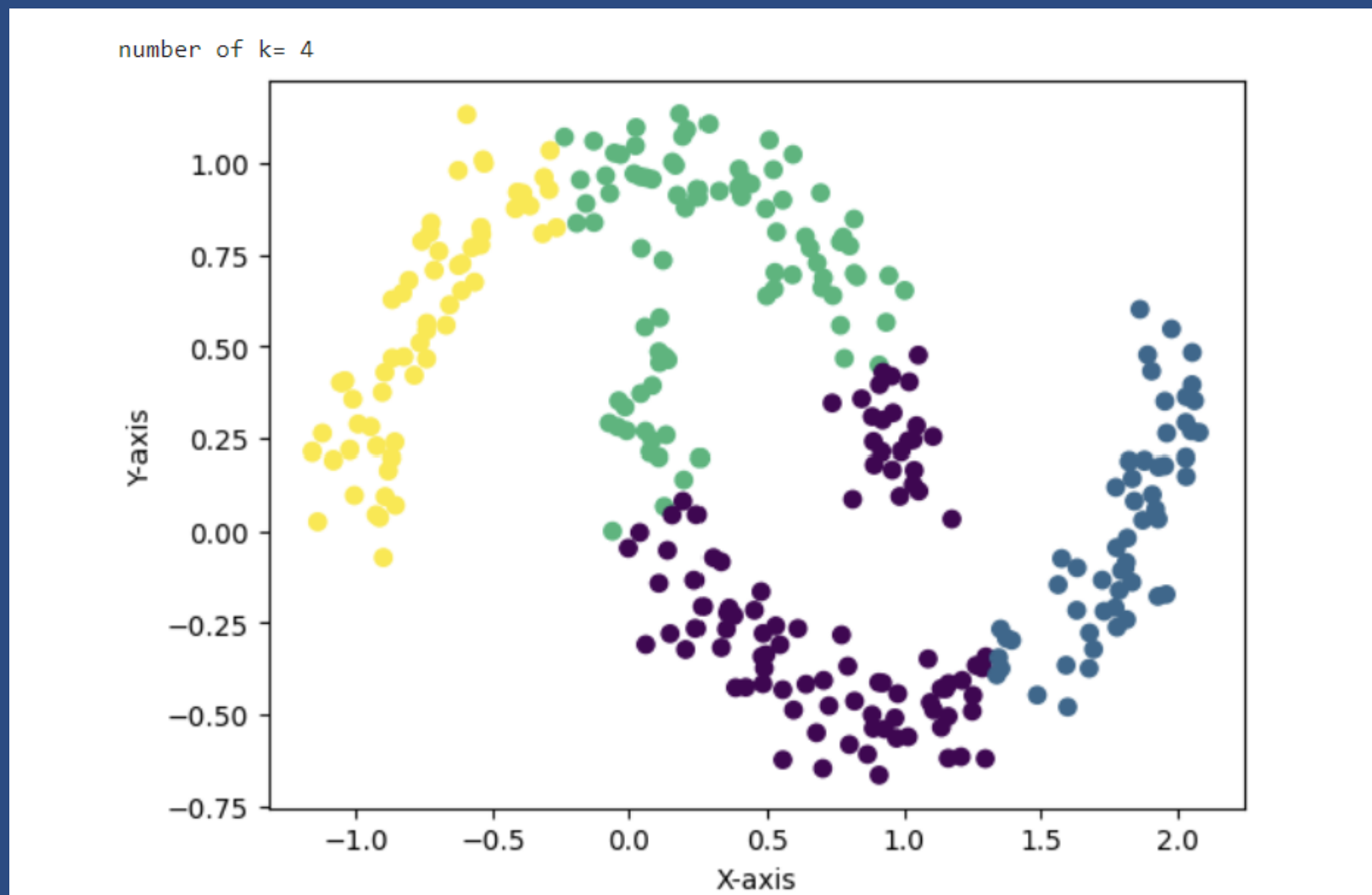


Gmm-Dataset 2

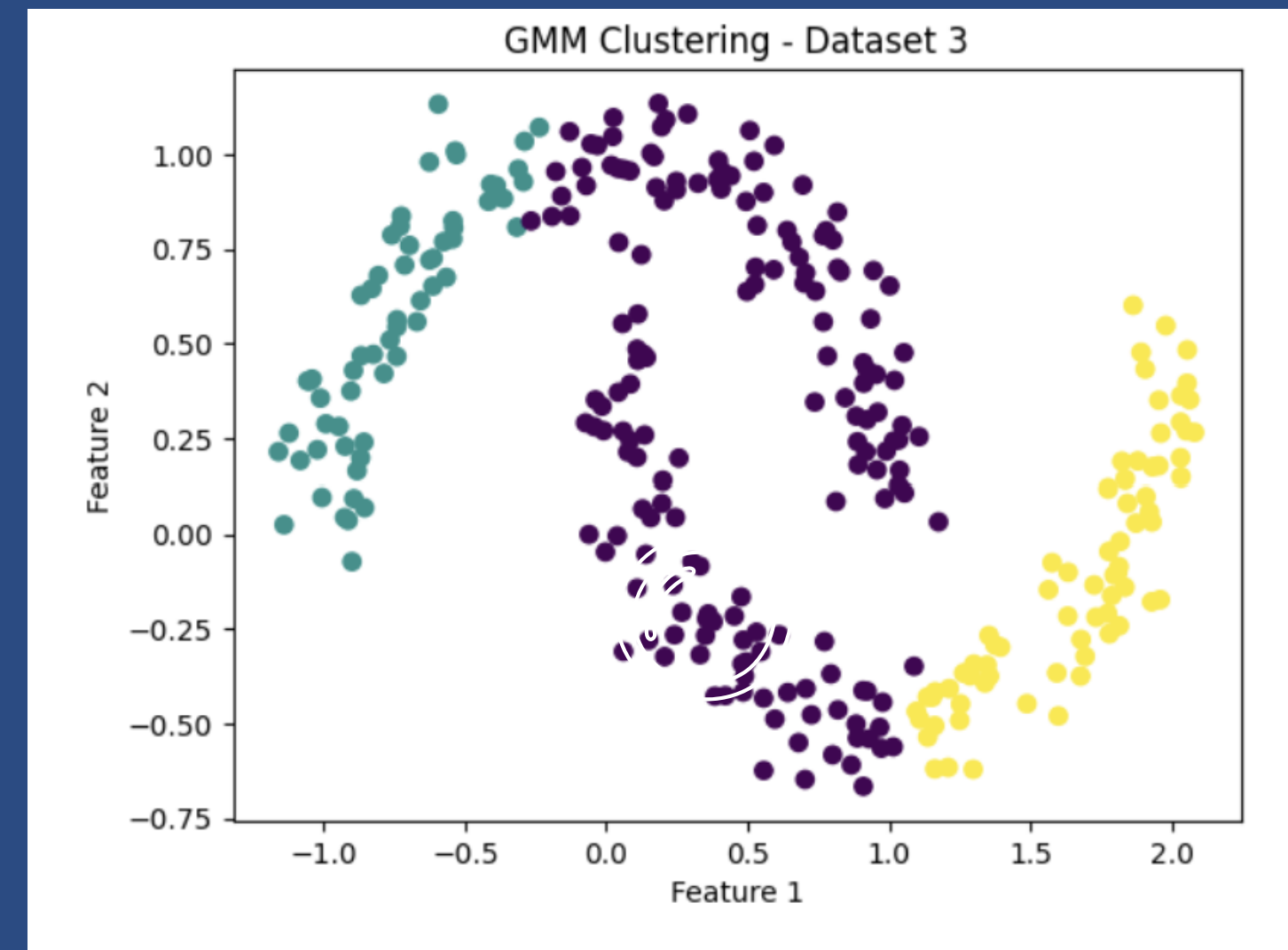




Kmean-Dataset 3

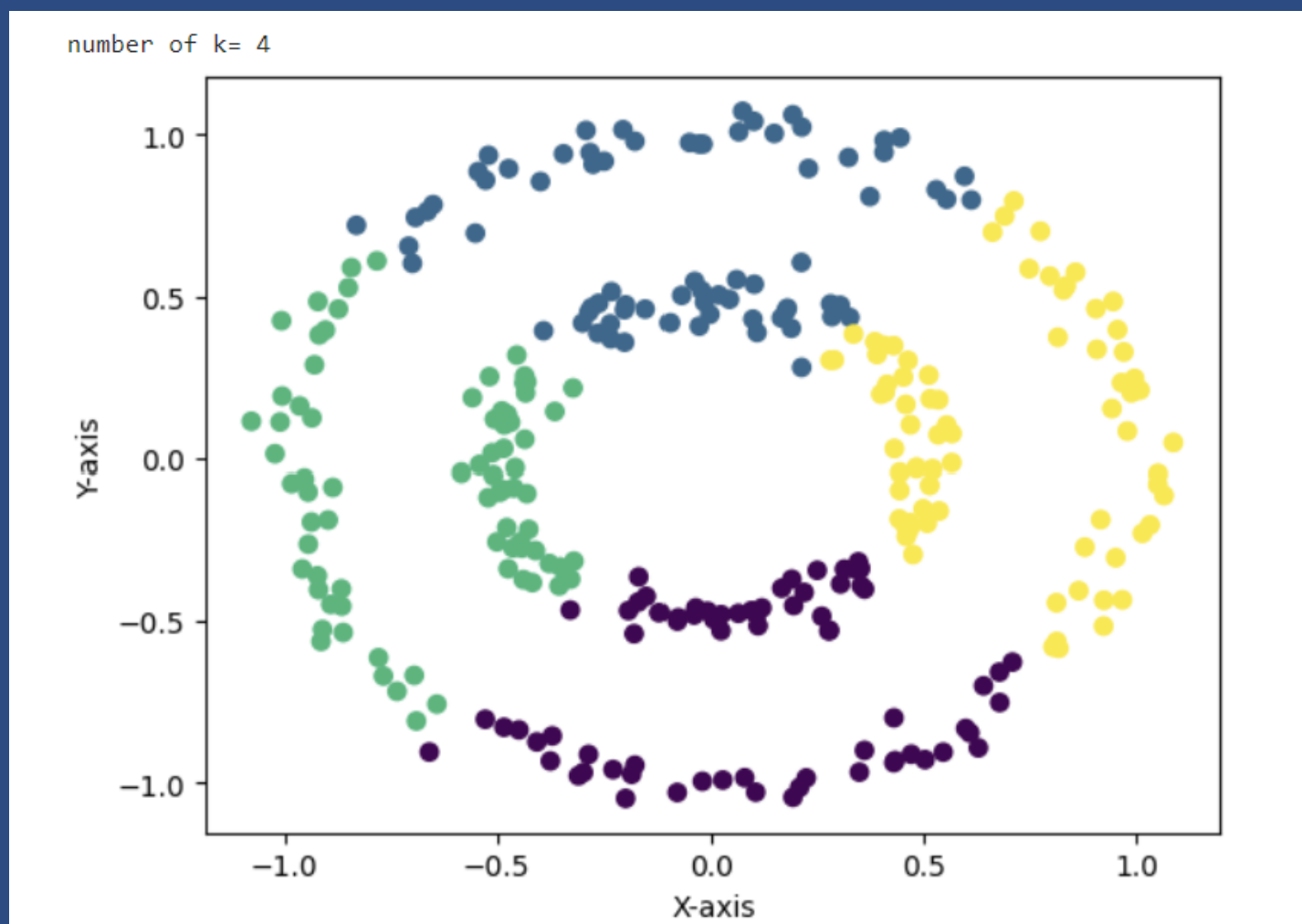


Gmm-Dataset 3

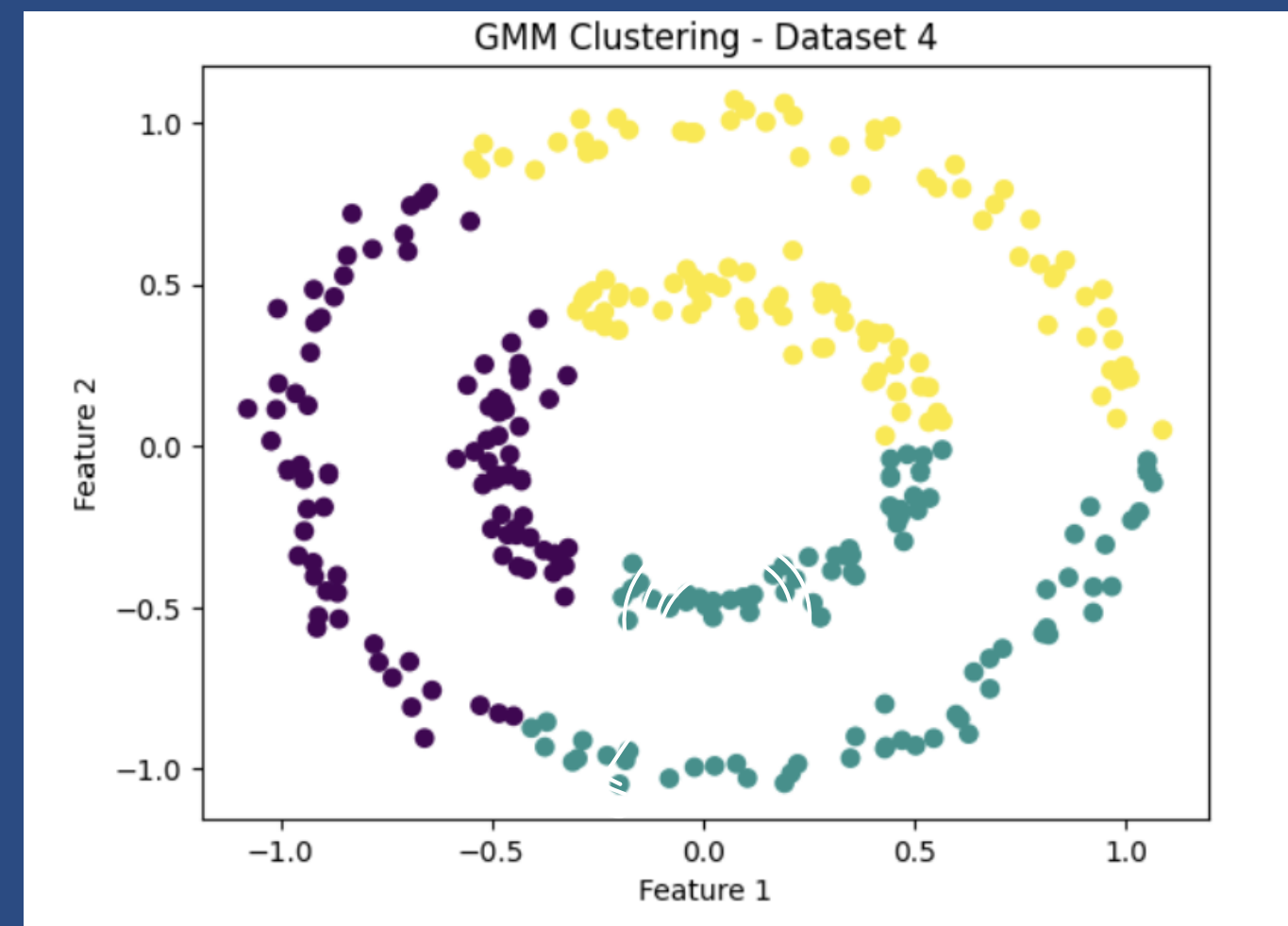




Kmean-Dataset 4



Gmm-Dataset 4



MACHINE LEARNING PROJECT



Clustering analysis





Clustering analysis



F-score

combines precision and recall to evaluate the clustering algorithms.

higher F-score values indicate better clustering results.



NMI (Normalized Mutual Information)

evaluates the similarity between two sets of labels

higher NMI value indicates a higher agreement between the ground truth and the clustering results.



Rand index

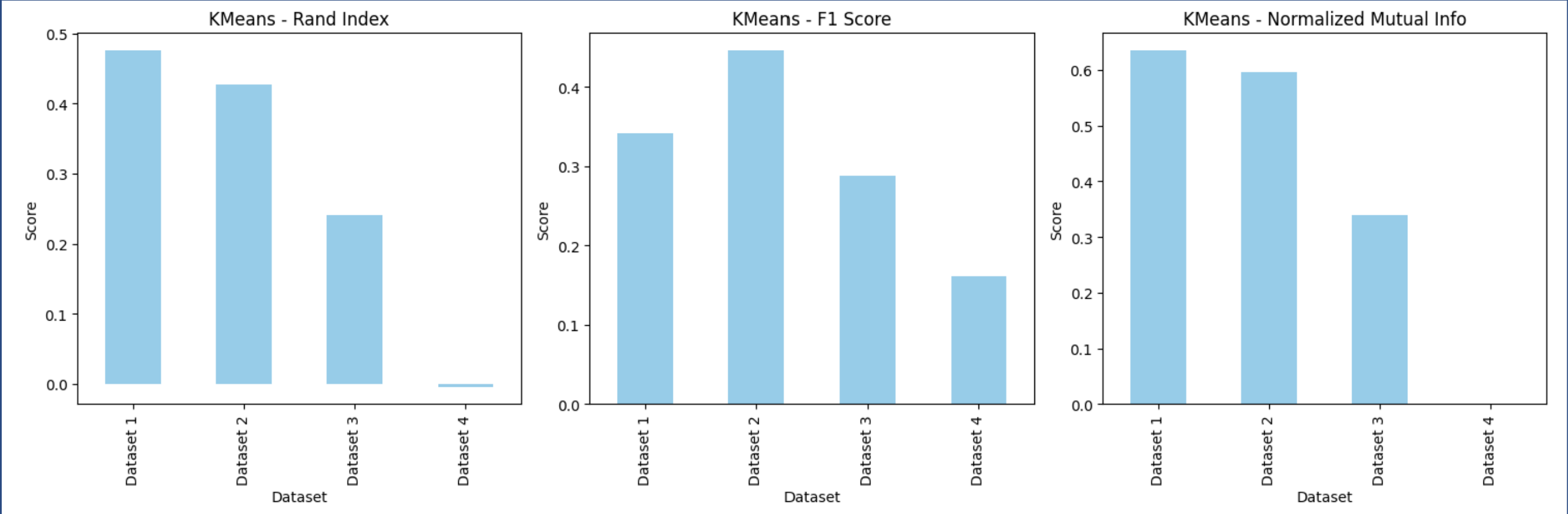
estimates the number of agreements and disagreements between pairs of data points in terms of their assignment to the same or distinct clusters

ranges from 0 to 1, where 1 means a excellent match between the partitions





Kmean results



Gmm results

