FML Assignment 3 Report

Sakthe Balan G K ME21B714

Introduction

This report presents solutions involving Principal Component Analysis (PCA) and K-means clustering on two different datasets. The tasks are divided into two main parts: applying PCA on the MNIST dataset and performing K-means clustering on a synthetic 2D dataset.

Task 1: PCA on the MNIST Dataset

For this task, we downloaded the MNIST dataset from the Hugging Face repository and used a random subset of 1000 images, with 100 images from each digit class (0–9).

Part (i): Applying PCA and Visualizing Variance Explained by Principal Components

The PCA was implemented using a custom class. The data was first centered, and the covariance matrix was computed. The eigenvalues and eigenvectors of the covariance matrix were then calculated and sorted to identify the principal components.

The following code loads the MNIST dataset, preprocesses it, and applies PCA to compute the principal components and the explained variance.

```
# Loading necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from datasets import load_dataset

# Load MNIST dataset and sample 100 images per class
ds = load_dataset("ylecun/mnist")
df_train = pd.DataFrame(ds['train'])
df = pd.DataFrame()

for i in range(10):
    df_i = df_train[df_train['label'] == i].sample(n=100, random_state=42)
    df = pd.concat([df, df_i], axis=0)

X = df['image'].to_numpy()
y = df['label'].to_numpy()

# Flattening and normalizing images
X_array = np.array([np.array(image).flatten() for image in X]) / 255
```

Listing 1: Loading the Dataset and Preprocessing

This code loads and preprocesses the MNIST dataset. It selects 100 samples from each class, flattens each image, and normalizes pixel values to prepare the data for PCA.

```
class PCA:
    def __init__(self, n_components):
        self.n_components = n_components
        self.mean = None
        self.components = None
        self.explained_variance = None

    def fit(self, X):
        # Centering the data
        self.mean = np.mean(X, axis=0)
        X_centered = X - self.mean

        # Computing covariance matrix and eigendecomposition
        covariance_matrix = np.cov(X_centered, rowvar=False)
        eigenvalues, eigenvectors = np.linalg.eigh(covariance_matrix)

        # Sorting eigenvalues and selecting top components
        sorted_indices = np.argsort(eigenvalues)[::-1]
        eigenvalues = eigenvalues[sorted_indices]
        eigenvectors = eigenvectors[:, sorted_indices]

        # Storing principal components and explained variance
        self.components = eigenvectors[:, :self.n_components]
        self.explained_variance = eigenvalues[:self.n_components]
        return self.explained_variance

    def transform(self, X):
        X_centered = X - self.mean
        return np.dot(X_centered, self.components)

    def inverse_transform(self, X_transformed):
        return np.dot(X_transformed, self.components.T) + self.mean
```

Listing 2: Custom PCA Class Implementation

The `PCA` class implements PCA. The `fit` method centers the data, computes the covariance matrix, performs eigen decomposition, and selects the top components based on eigenvalues.

```
# Applying PCA with full dimensionality
pca = PCA(n_components=784)
explained_variance = pca.fit(X_array)

# Plotting cumulative explained variance
cumulative_variance = np.cumsum(pca.explained_variance / np.sum(pca.
    explained_variance))
plt.plot(range(1, len(cumulative_variance) + 1), cumulative_variance)
plt.xlabel("Number of Principal Components")
plt.ylabel("Cumulative Explained Variance")
plt.show()
```

Listing 3: Calculating and Plotting Cumulative Explained Variance

This code applies PCA, calculates cumulative explained variance, and plots it to show how much variance each principal component explains, aiding in dimensionality selection.

The variance explained by each components is give as the output in the code as "variances" which is nothing but the eigen values. The variance obtained corresponding to first 10 principal components are listed below

```
    Variances:[5.07821134 3.73474107 3.32068165 2.92429288 2.58572318 2.16902598
 1.73423302 1.54885368 1.44697988 1.22955789]
```

The cumulative explained variance was plotted as a function of the number of principal components. This plot (Figure 1) shows that a large portion of the variance in the data can be explained by the first few principal components. This indicates that dimensionality reduction can capture significant information with fewer components.

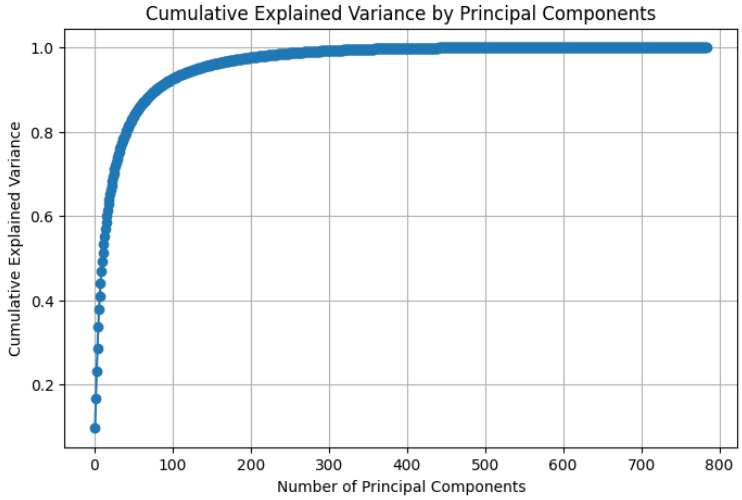

Figure 1: Cumulative Explained Variance by Principal Components

The first ten principal components were visualized to show how each component captures specific patterns in the digit images (Figure 2). These principal components are directions in the dataset with the highest variance and can be interpreted as characteristic features of the digit images. The following code implements the above mentioned.

```
fig, axes = plt.subplots(2, 5, figsize=(12, 5))
axes = axes.ravel()
for i in range(10):
    pc_image = pca.components[:, i].reshape(28, 28)
    axes[i].imshow(pc_image, cmap='gray')
    axes[i].set_title(f"PC {i+1}")
    axes[i].axis('off')
plt.tight_layout()
plt.show()
```

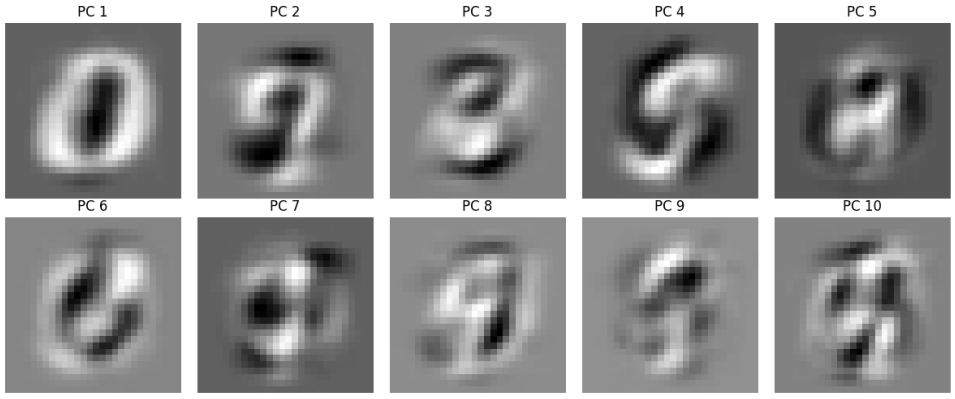

Figure 2: Visualization of the First 10 Principal Components

Part (ii): Reconstructing Images Using Different Dimensions

The code below reconstructs images with varying principal component dimensions to show the trade-off between dimensionality and information retention.

The dataset was reconstructed using varying dimensions, specifically 2, 10, 25, 50, 100, 200, and 300. As shown in Figure 3, the quality of reconstruction improves with increasing dimensions. For classification tasks, a balance between dimensionality and reconstruction quality is essential. Based on the results, we recommend a dimension of around **50**, which retains sufficient information while reducing computational complexity.

The below code transforms and reconstructs one sample image using different numbers of principal components to visualize the effects of dimensionality on image quality.

```
dimensions = [2, 10, 25, 50, 100, 200, 300]
fig, axes = plt.subplots(1, len(dimensions), figsize=(15, 5))

for i, d in enumerate(dimensions):
    pca = PCA(n_components=d)
    pca.fit(X_array)
    transformed = pca.transform(X_array)
    reconstructed = pca.inverse_transform(transformed)

    # Plot reconstructed image for each dimension
    axes[i].imshow(reconstructed[800].reshape(28, 28), cmap='gray')
    axes[i].set_title(f"d = {d}")
    axes[i].axis('off')
plt.show()
```

Listing 4: Reconstruction with Different Dimensions

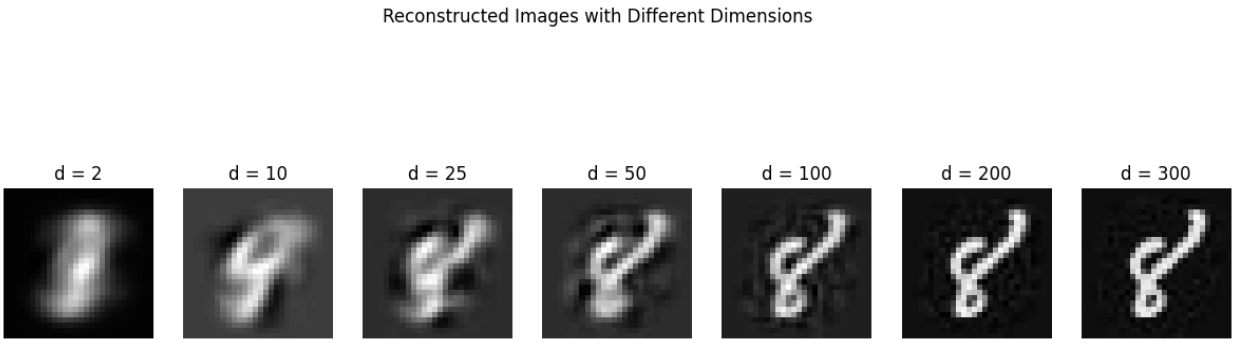

Figure 3: Reconstructed Images with Different Dimensions

Task 2: K-Means Clustering on a 2D Dataset

Part (i): Implementing Lloyd's Algorithm for K-Means with $k = 2$

The following code defines a custom K-means class and applies Lloyd's algorithm with $k = 2$ clusters, using five different random initializations. I have created seperate functions for initializing the centroids, computing euclidean distance, assigning data to the clusters, update the centroids after each iteration.

```
class KMeans:
    def __init__(self, k=2, max_iters=100, tol=1e-4, random_state=42):
        self.k = k
        self.max_iters = max_iters
        self.tol = tol
        self.random_state = random_state
        self.centroids = None
        self.labels_ = None
        self.wcss_ = []

    def initialize_centroids(self, data): # Random initialization
        np.random.seed(self.random_state)
        random_indices = np.random.choice(data.shape[0], self.k, replace=False)
        self.centroids = data[random_indices]
        self.initial_centroids = self.centroids.copy()  # Store the initial
            centroids


    def compute_distances(self, data): #Compute the Euclidean distance from each
        point to each centroid.
        distances = np.zeros((data.shape[0], self.k))
        for i, centroid in enumerate(self.centroids):
            distances[:, i] = np.linalg.norm(data - centroid, axis=1)
        return distances

    def assign_clusters(self, distances):#Assign each point to the nearest
        centroid based on computed distances
        return np.argmin(distances, axis=1)

    def update_centroids(self, data, labels):#Update centroids by calculating the
        mean of points in each cluster
        new_centroids = np.zeros((self.k, data.shape[1]))
        for i in range(self.k):
            cluster_points = data[labels == i]
            new_centroids[i] = cluster_points.mean(axis=0) if len(cluster_points)
                > 0 else self.centroids[i]
        self.centroids = new_centroids

    def calculate_wcss(self, data, labels): #Calculate within-cluster sum of
        squares (WCSS).
        wcss = 0
        for i, centroid in enumerate(self.centroids):
            cluster_points = data[labels == i]
            wcss += np.sum((cluster_points - centroid) ** 2)
        return wcss

    def fit(self, data): #Run the K-means algorithm on the data.
        self.initialize_centroids(data)

        for iteration in range(self.max_iters):
            distances = self.compute_distances(data)
```

```
            labels = self.assign_clusters(distances)
            self.update_centroids(data, labels)
            wcss = self.calculate_wcss(data, labels)
            self.wcss_.append(wcss)
            if iteration > 0 and abs(self.wcss_[-2] - self.wcss_[-1]) < self.tol:
                break

            self.labels_ = labels  # Store labels for final clusters

    def predict(self, data): #Predict the nearest cluster for each data point.
        distances = self.compute_distances(data)
        return self.assign_clusters(distances)
```

Listing 5: K-Means Clustering with Lloyd's Algorithm

```
    def plot_voronoi_regions(self, data):
        # Plot data points
        plt.scatter(data[:, 0], data[:, 1], c=self.labels_, cmap='viridis', alpha
            =0.5)

        # Get plot range based on data
        x_min, x_max = data[:, 0].min() - 1, data[:, 0].max() + 1
        y_min, y_max = data[:, 1].min() - 1, data[:, 1].max() + 1
        plt.xlim(x_min, x_max)
        plt.ylim(y_min, y_max)

        # Plot each centroid and bisector between centroid pairs
        for i, centroid in enumerate(self.centroids):
            plt.scatter(centroid[0], centroid[1], c='red', marker='x', s=100)

        # Draw bisectors between each pair of centroids
        for i in range(self.k):
            for j in range(i + 1, self.k):
                # Midpoint between two centroids
                midpoint = (self.centroids[i] + self.centroids[j]) / 2

                # Slope of the line connecting the two centroids
                delta_y = self.centroids[j][1] - self.centroids[i][1]
                delta_x = self.centroids[j][0] - self.centroids[i][0]

                if delta_x == 0:  # Vertical line
                    plt.axvline(x=midpoint[0], color='orange', linestyle='--')
                else:
                    # Perpendicular bisector slope
                    slope = -delta_x / delta_y

                    # y = mx + c form of the bisector
                    intercept = midpoint[1] - slope * midpoint[0]

                    # Generate line points for plotting within the plot range
                    x_vals = np.linspace(x_min, x_max, 500)
                    y_vals = slope * x_vals + intercept

                    # Plot the line (Voronoi boundary)
                    plt.plot(x_vals, y_vals, 'orange', linestyle='--')

        plt.title(f"Voronoi Regions for K = {self.k}")
        plt.show()
```

The former code manually implements the lloyd algorithm and the latter code is for manually plotting the voronoi regions for different clusters. I have referred class lectures and notes for the logics for these codes.

Lloyd's algorithm was implemented in a custom K-means class. The algorithm was run with $k = 2$ clusters and five different random initializations. The Within-Cluster Sum of Squares (WCSS) was plotted with respect to the number of iterations for each initialization. The error function converged after a few iterations, demonstrating the stability of the clustering.

The final clusters for each initialization were visualized below. Different colors represent different clusters, and each run produced slightly different results due to the random initialization.

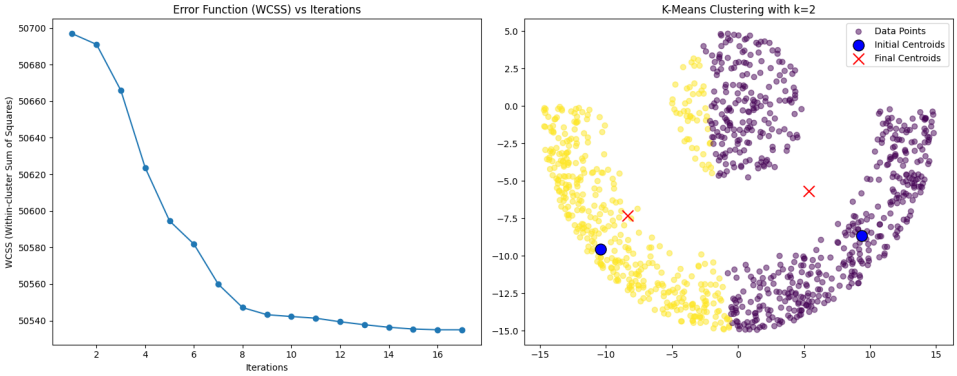

Figure 4: Error Function plot and clusters (k=2) obtained from random initialization 1

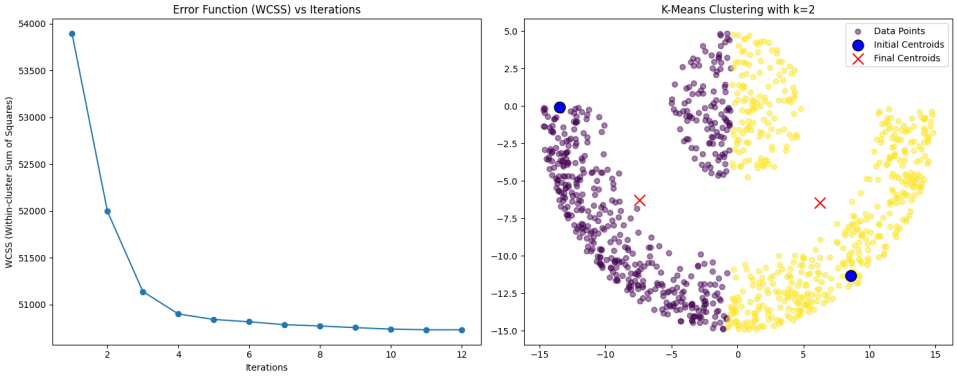

Figure 5: Error Function plot and clusters (k=2) obtained from random initialization 1

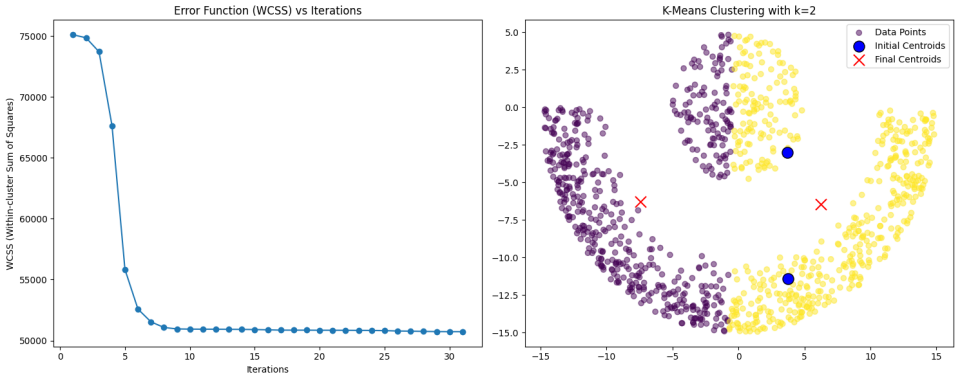

Figure 6: Error Function plot and clusters (k=2) obtained from random initialization 1

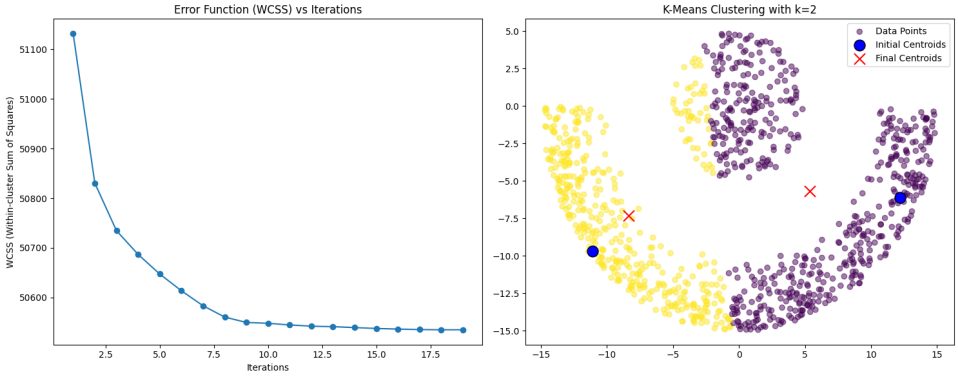

Figure 7: Error Function plot and clusters (k=2) obtained from random initialization 1

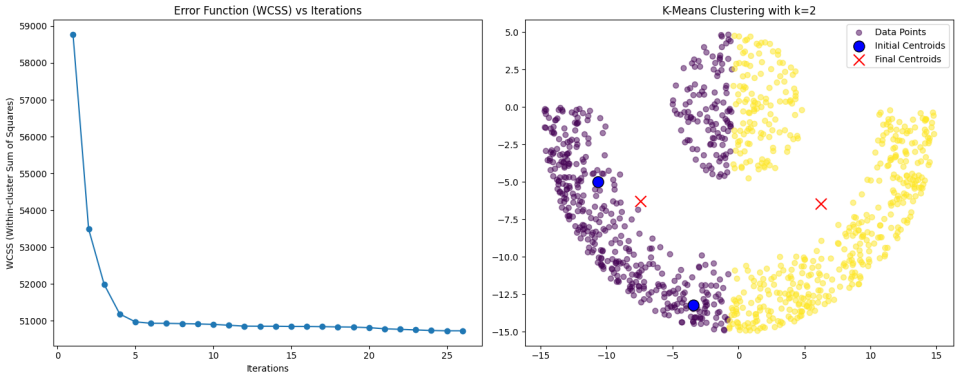

Figure 8: Error Function plot and clusters (k=2) obtained from random initialization 1

Part (ii): K-Means Clustering with Varying K Values and Voronoi Regions

The K-means algorithm was applied with different values of $K = \{2, 3, 4, 5\}$. For each K, the Voronoi regions were plotted. Each region represents points closer to a particular cluster center than to any other center.

You can see that for different initializations, lloyd algorithm converges to same optimum but the number of iterations varies depends according to the initialization.

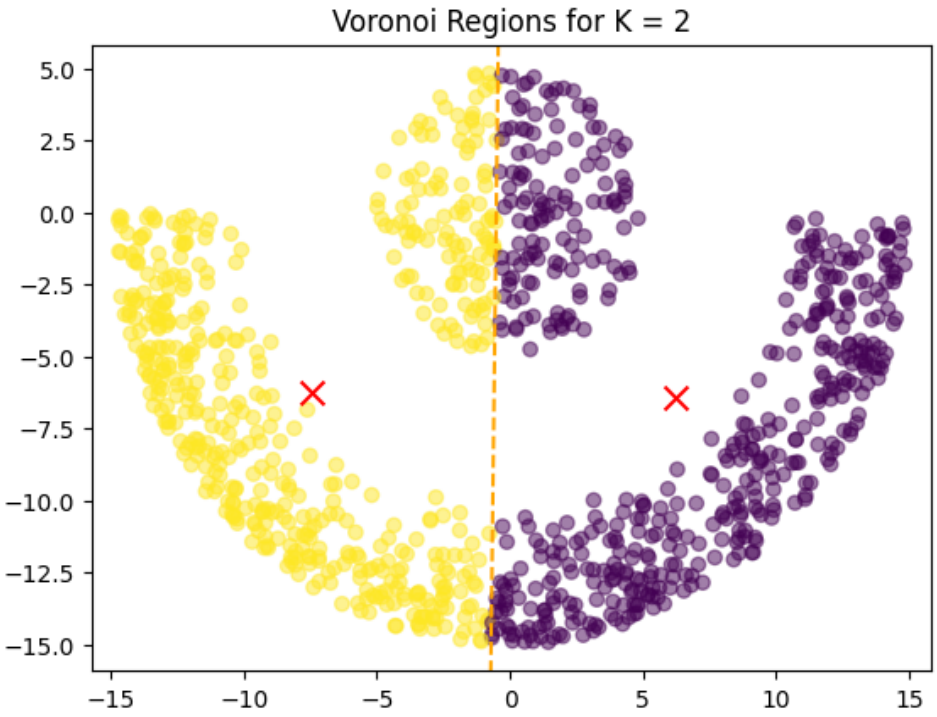

Figure 9: Voronoi Regions for Different Values of K

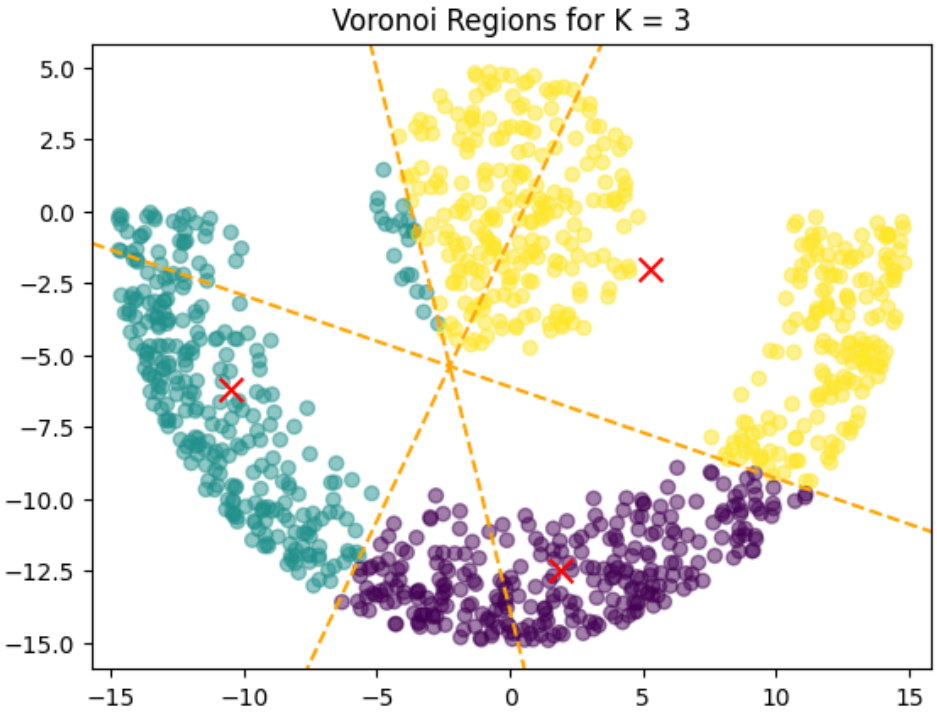

Figure 10: Voronoi Regions for Different Values of K

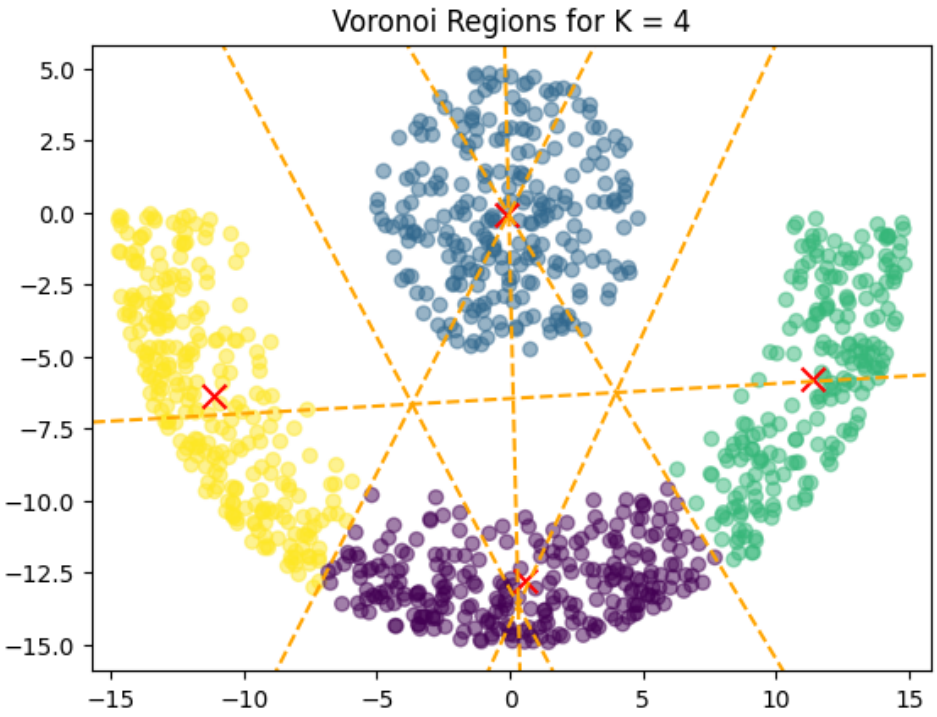

Figure 11: Voronoi Regions for Different Values of K

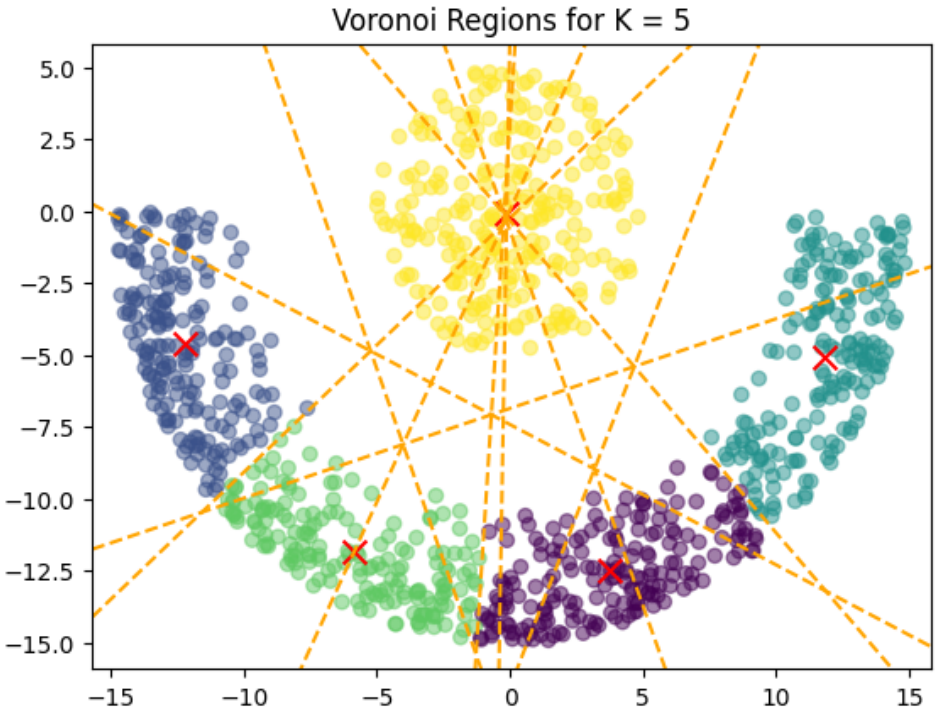

Figure 12: Voronoi Regions for Different Values of K

Part (iii): Evaluation of Lloyd's Algorithm for Clustering

Lloyd's algorithm, as used in the K-means clustering, generally performed well in separating the data into clusters based on Euclidean distance. However, K-means clustering is limited by its linear decision boundaries, making it less suitable for datasets where the optimal boundaries between clusters are non-linear.

From the plots, we can observe that the clusters do not align perfectly with a clear separation due to the non-linear nature of the data. For datasets like this, where the underlying structure of clusters may be better represented with non-linear boundaries, algorithms such as spectral clustering(Kernalized kNN), DBSCAN, or Gaussian Mixture Models (GMMs) would be more appropriate. These methods can capture more complex, non-linear relationships within the data, which K-means is inherently unable to do due to its linear separation limitations.

In conclusion, while Lloyd's algorithm provides a simple and effective clustering approach for many datasets, it is not ideal for non-linear datasets where clusters require non-linear decision boundaries.