Foundations of Machine Learning

DA5400 – Assignment III – BE21B037

Libraries Used:

```
# Libraries
import pandas as pd
from PIL import Image
import io
import matplotlib.pyplot as plt
import numpy as np
import warnings
warnings.filterwarnings('ignore')
```

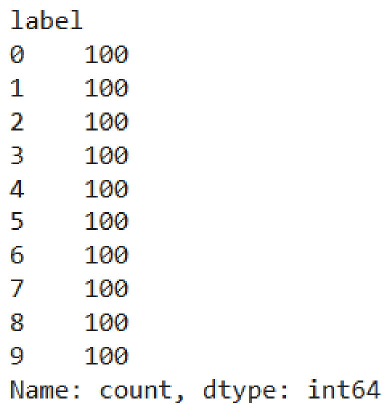
Pandas and **Numpy** for data manipulation and analysis, **PIL** and **io** to convert bytes to images and back. **Matplotlib** for graphs and plots

QUESTIONS

1) Download the MNIST dataset from https://huggingface.co/datasets/mnist . Use a random set of 1000 images (100 from each class 0-9) as your dataset.

```
np.random.seed(5400)
splits = {'train': 'mnist/train-00000-of-00001.parquet', 'test':
'mnist/test-00000-of-00001.parquet'}
df = pd.read_parquet("hf://datasets/ylecun/mnist/" + splits["train"])
data = df.groupby('label', group_keys=False).apply(lambda x: x.sample(min(len(x),
100)))
print(data['label'].value_counts())
data
```

The importing of the dataset is directly done using the **code given in hugging face** website.
The output of the above code tells us the splitting of the dataset and also what the data frame looks like



```
label
0    100
1    100
2    100
3    100
4    100
5    100
6    100
7    100
8    100
9    100
Name: count, dtype: int64
```

The dataset

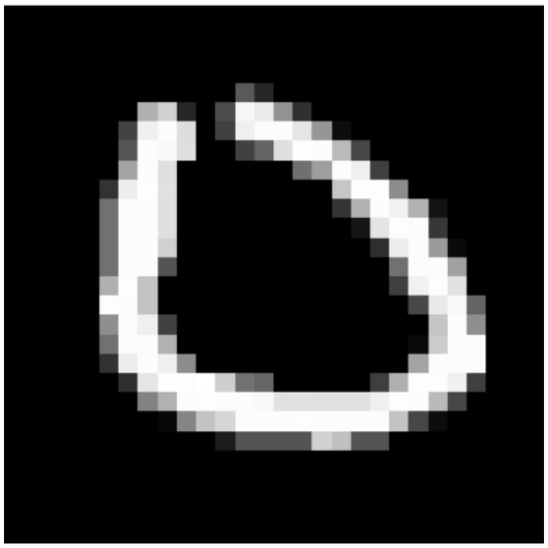
	image	label
41357	{'bytes': b"\x89PNG\r\n\x1a\n\x00\x00\x00\rIHD...	0
7875	{'bytes': b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHD...	0
42499	{'bytes': b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHD...	0
48412	{'bytes': b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHD...	0
52621	{'bytes': b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHD...	0
...	...	...
5718	{'bytes': b"\x89PNG\r\n\x1a\n\x00\x00\x00\rIHD...	9
39172	{'bytes': b"\x89PNG\r\n\x1a\n\x00\x00\x00\rIHD...	9
46490	{'bytes': b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHD...	9
29752	{'bytes': b"\x89PNG\r\n\x1a\n\x00\x00\x00\rIHD...	9
45244	{'bytes': b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHD...	9

1000 rows × 2 columns

To visualize a single image

```python
# Visualizing a single image
image = data.iloc[0]["image"]['bytes']
image = Image.open(io.BytesIO(image))
plt.imshow(image, cmap='gray')
plt.axis('off')
plt.show()
```

The output being



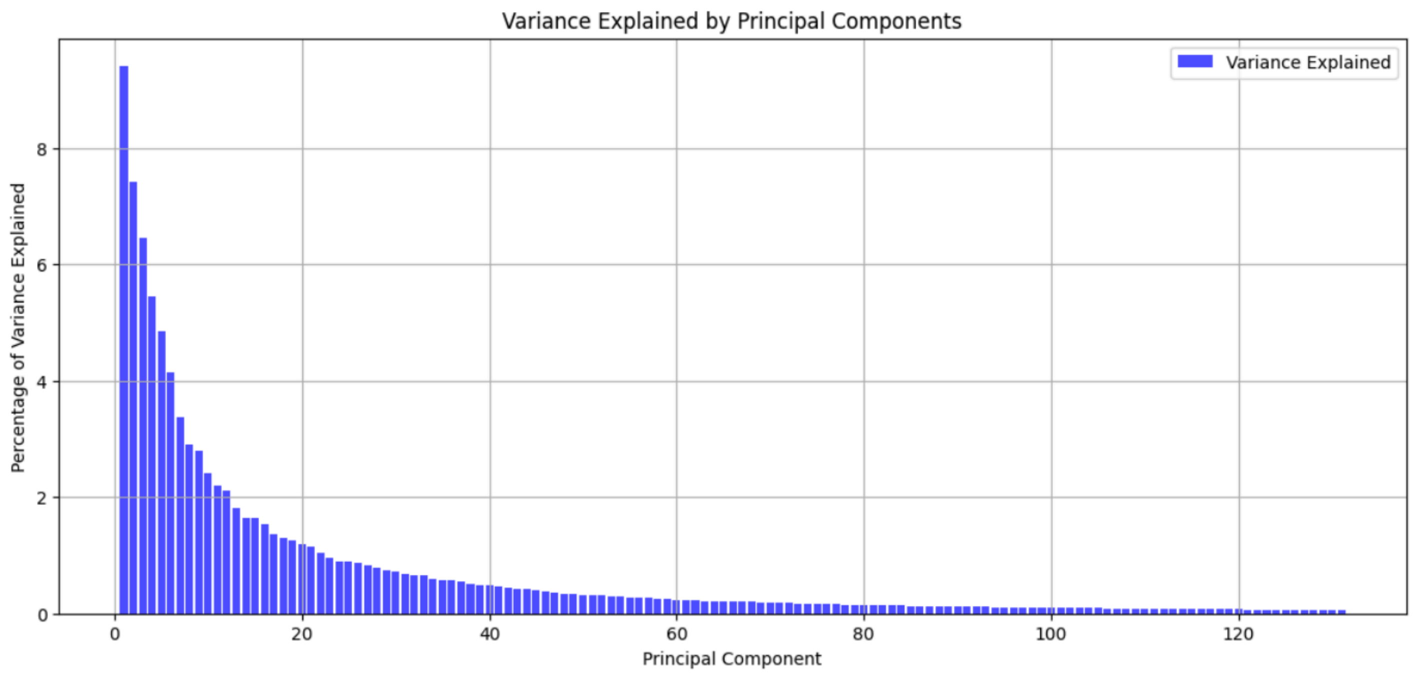
i) Write a piece of code to run the PCA algorithm on this data-set. Visualize the images of the principal components that you obtain. How much of the variance in the data-set is explained by each of the principal components?

The code to run the PCA algorithm is given below, we also find the number of principal components required to get 95% explained variance.

```python
def image_array(image_bits):
    img = Image.open(io.BytesIO(image_bits))
    image_1D = np.array(img)
    return image_1D.flatten()

data['image_numerical'] = data['image'].apply(lambda x: image_array(x['bytes']))

# Centering
X = np.stack(data['image_numerical'].values)
mean_X = np.mean(X, axis=0)
X_centered = X - mean_X

# Covariance Matrix
covariance_matrix = (X_centered.T @ X_centered) / (X_centered.shape[0] - 1)
covariance_matrix = (covariance_matrix + covariance_matrix.T) / 2
e_values, e_vectors = np.linalg.eig(covariance_matrix)
e_values = np.real(e_values)

# Top eignvalues
idx = np.argsort(e_values)[::-1]
sort_e_vectors = e_vectors[:, idx]
sort_e_values = e_values[idx]
total_variance = np.sum(sort_e_values)
variance_explained = sort_e_values / total_variance * 100
cum_variance = np.cumsum(variance_explained)
idx95 = np.argmax(cum_variance >= 95) + 1
print(f'The number of PCs required for 95 % explained variance = {idx95}')

var = variance_explained.tolist()
cumvar = cum_variance.tolist()

# Plotting
plt.figure(figsize=(14, 6))
plt.bar(range(1, idx95 + 1), variance_explained[:idx95], alpha=0.7, label='Variance
Explained', color='blue')

plt.xlabel('Principal Component')
plt.ylabel('Percentage of Variance Explained')
plt.title('Variance Explained by Principal Components')
plt.legend(loc='best')
plt.grid(True)
plt.show()
```

The output of which is the following graph

```
The number of PCs required for 95 % explained variance = 131
```



The number of PCs required to have 95% explained variance(cumulative) is 131 PCs

See the actual variance, we will check the top 25 PCs

```
pca_df = pd.DataFrame({
    'Principal Component': range(1, len(variance_explained) + 1),
    'Variance Explained (%)': variance_explained,
    'Cumulative Variance Explained (%)': cumvar })
pca_df.head(25)
```

The output of which is

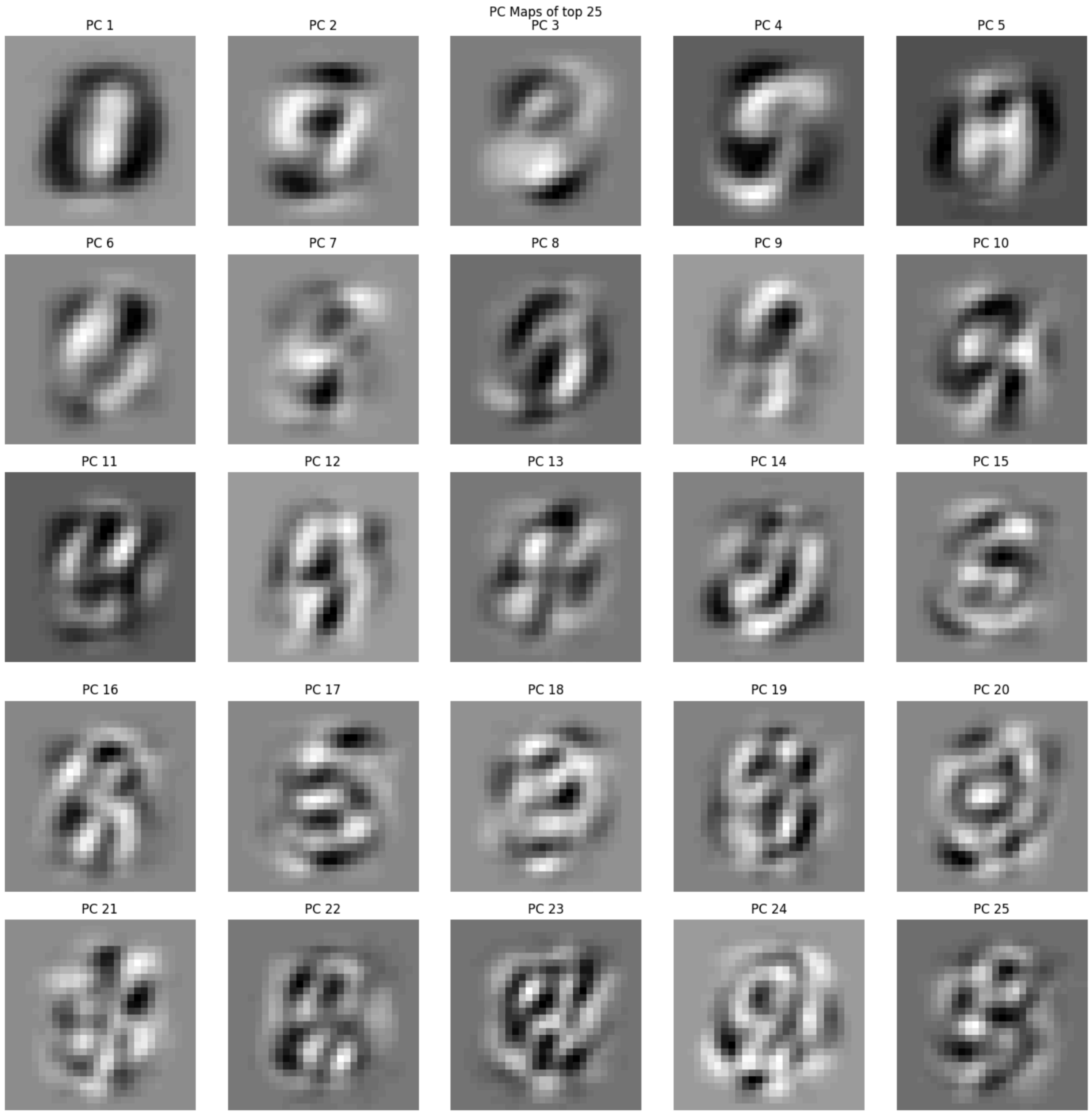
	Principal Component	Variance Explained (%)	Cumulative Variance Explained (%)
0	1	9.413416	9.413416
1	2	7.414563	16.827979
2	3	6.454818	23.282797
3	4	5.447598	28.730395
4	5	4.856141	33.586536
5	6	4.138591	37.725128
6	7	3.381233	41.106361
7	8	2.906800	44.013161
8	9	2.801987	46.815148
9	10	2.416114	49.231262
10	11	2.195657	51.426919
11	12	2.115081	53.542001
12	13	1.812081	55.354082

13	14	1.650308	57.004390
14	15	1.642837	58.647227
15	16	1.534453	60.181681
16	17	1.373428	61.555109
17	18	1.309620	62.864729
18	19	1.245764	64.110493
19	20	1.184327	65.294820
20	21	1.144698	66.439518
21	22	1.038734	67.478252
22	23	0.946590	68.424842
23	24	0.894548	69.319389
24	25	0.885174	70.204563

To visualize these 25 PC Maps the following code is used

```
k = 25
topkeigenvectors = sort_e_vectors[:, :k]
topkeigenvectors = np.real(topkeigenvectors)

plt.figure(figsize=(15, 15))
for i in range(k):
    pc_map = topkeigenvectors[:, i].reshape(28, 28)
    plt.subplot(5, 5, i + 1)
    plt.imshow(pc_map, cmap='gray')
    plt.axis('off')
    plt.title(f'PC {i + 1}')

plt.suptitle(f'PC Maps of top {k}')
plt.tight_layout()
plt.show()
```

The output plots being



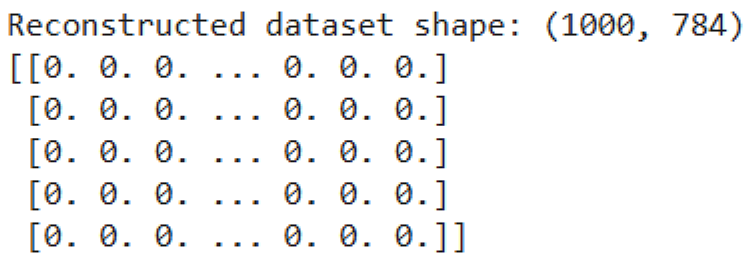
ii) Reconstruct the dataset using different dimensional representations. How do these look like? If you had to pick a dimension d that can be used for a downstream task where you need to classify the digits correctly, what would you pick and why?

The code to reconstruct the dataset using different dimensional representation is by matrix multiplying the eigenvectors, as the datapoint can be represented as a linear combination of the eigenvectors

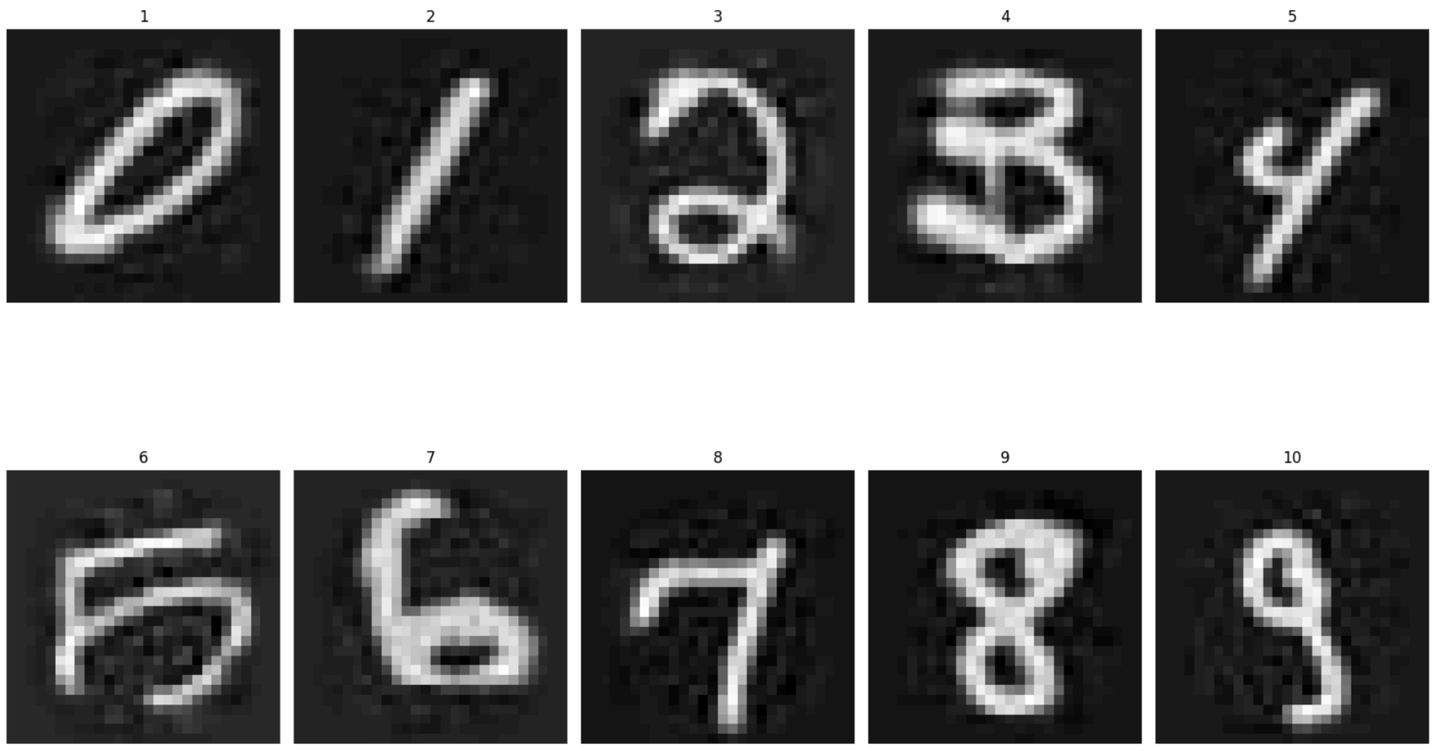
```
k = 131
top_eigenvectors = sort_e_vectors[:, :k]
projected_data = X_centered @ top_eigenvectors
reconstructed_data = projected_data @ top_eigenvectors.T
```

```python
reconstructed_data += mean_X
reconstructed_data = np.real(reconstructed_data)

print("Reconstructed dataset shape:", reconstructed_data.shape)
print(reconstructed_data[:5, :])

num = 10
plt.figure(figsize=(16, 16))
for i in range(num):
    image = reconstructed_data[i*100].reshape(28, 28)
    plt.subplot(2, 5, i + 1)
    plt.imshow(image, cmap='gray')
    plt.axis('off')
    plt.tight_layout()
    plt.title(f'{i + 1}')
```

The output of which is



```
Reconstructed dataset shape: (1000, 784)
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

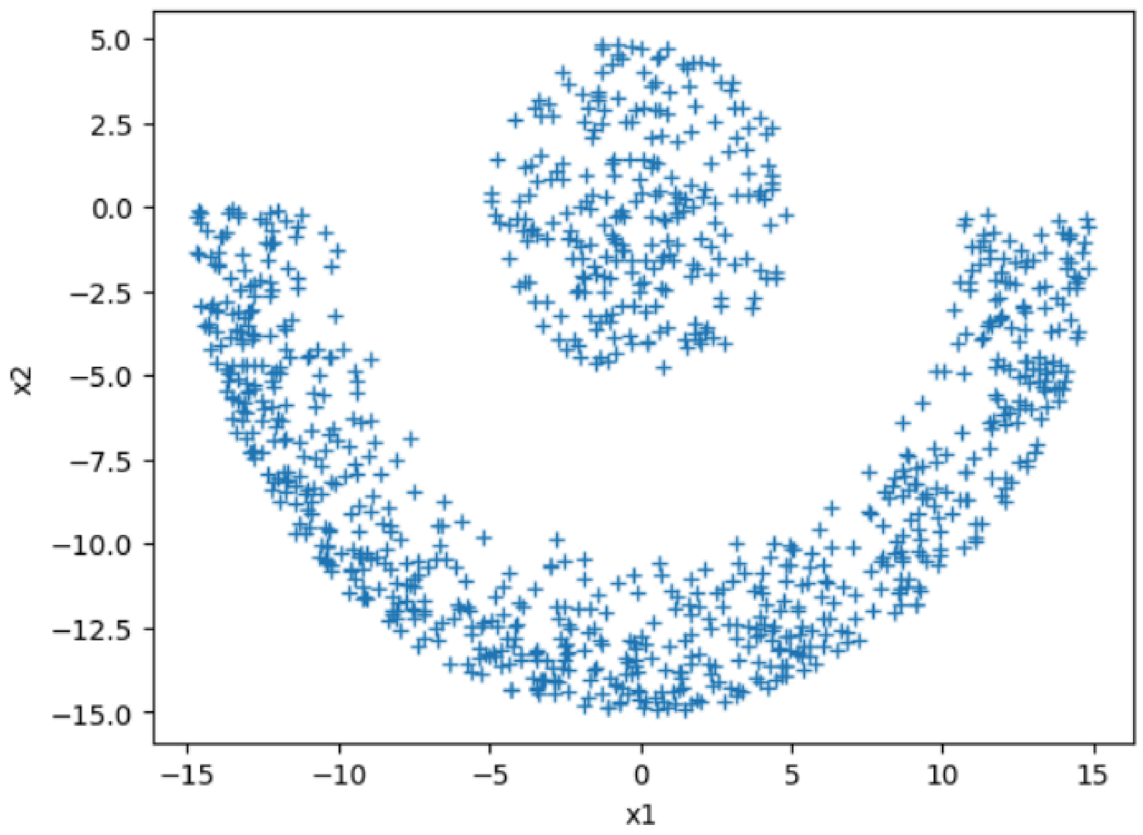


Using d = 131 principle components would be ideal for classification, since the variance explained by these 131 components would be greater than 95%, which is ideally taken as the cutoff for classification. The reconstruction showed us that the dataset was clearly reconstructed using these 131 PCs

2) You are given a data-set with 1000 data points each in R2 (cm dataset 2.csv)

```
df = pd.read_csv('/content/cm_dataset_2.csv', names=['x1', 'x2'], header=None)
plt.plot(df['x1'], df['x2'], '+')
plt.xlabel('x1')
plt.ylabel('x2')
plt.show()
```

The dataset looks like



i) Write a piece of code to implement the Llyod"s algorithm for the K-means problem with k = 2 . Try 5 different random initialization and plot the error function w.r.t iterations in each case. In each case, plot the clusters obtained in different colors.

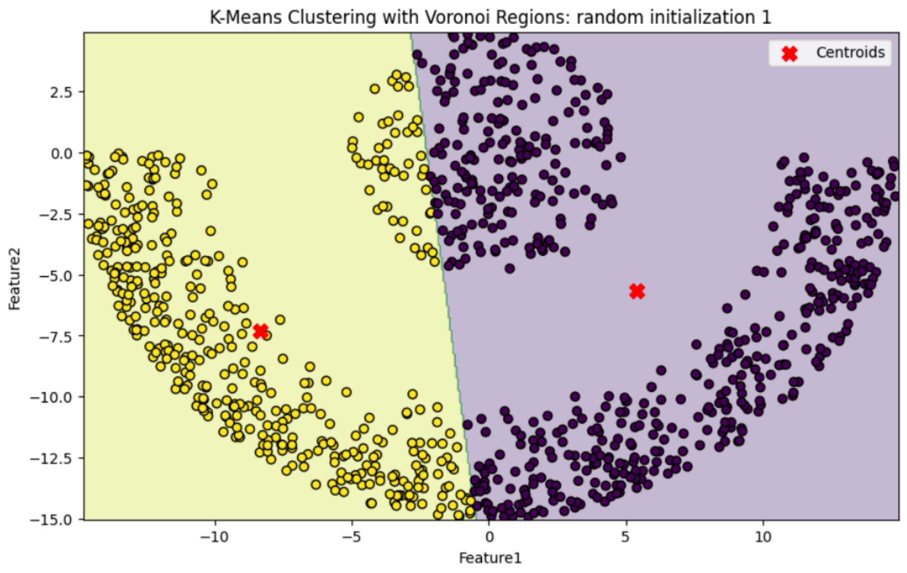
The code to implement llyod's algorithm is given below

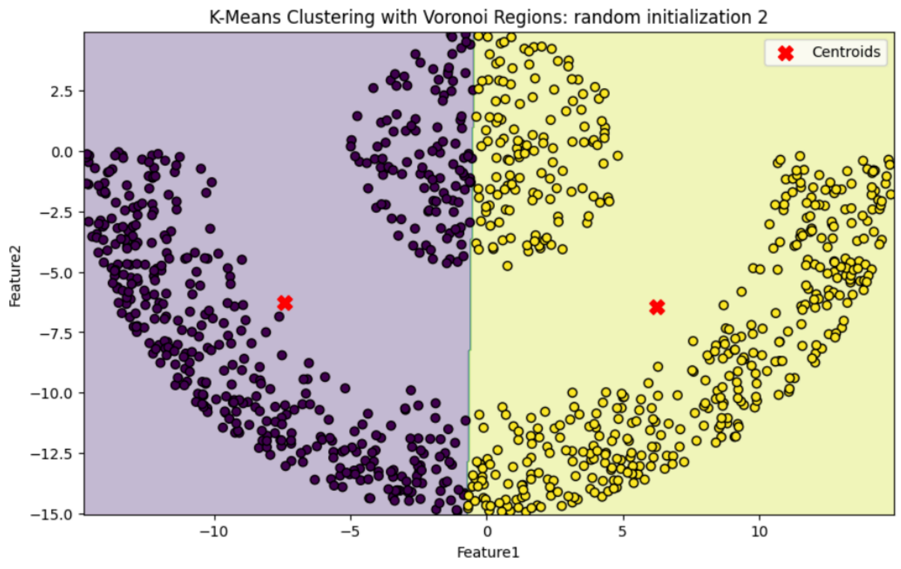
```
def llyods_algo(data, k=2, max_iters=100):
    centroids = data[np.random.choice(data.shape[0], k, replace=False)]
    err = []
    for i in range(max_iters):
        distances = np.linalg.norm(data[:, np.newaxis] - centroids, axis=2)
        cluster_assignments = np.argmin(distances, axis=1)
        error = np.sum([np.linalg.norm(data[j] -
centroids[cluster_assignments[j]])**2 for j in range(data.shape[0])])
```

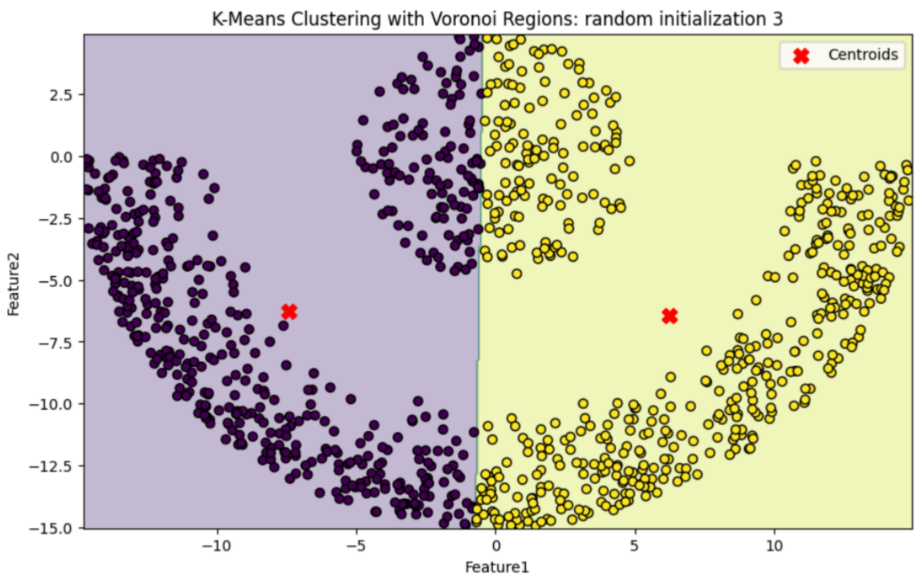
```
        err.append(error)

        new_centroids = np.array([data[cluster_assignments == j].mean(axis=0) for j
in range(k)])

        if np.allclose(centroids, new_centroids):
            break

        centroids = new_centroids
    return centroids, cluster_assignments, err

F_values = []
for i in range(5):
  np.random.seed(i)
  data_points = df.values
  centroids, cluster_assignments, F_value = llyods_algo(data_points, k=2)
  F_values.append(F_value)

  # Grid points over dataset range
  x_min, x_max = data_points[:, 0].min() - 0.1, data_points[:, 0].max() + 0.1
  y_min, y_max = data_points[:, 1].min() - 0.1, data_points[:, 1].max() + 0.1
  xx, yy = np.meshgrid(np.linspace(x_min, x_max, 500), np.linspace(y_min, y_max,
500))

  grid_points = np.c_[xx.ravel(), yy.ravel()]
  distances = np.linalg.norm(grid_points[:, np.newaxis] - centroids, axis=2)
  grid_assignments = np.argmin(distances, axis=1)

  grid_assignments = grid_assignments.reshape(xx.shape)
  plt.figure(figsize=(10, 6))
  plt.contourf(xx, yy, grid_assignments, cmap='viridis', alpha=0.3)
  plt.scatter(data_points[:, 0], data_points[:, 1], c=cluster_assignments,
cmap='viridis', edgecolor='k')
  plt.scatter(centroids[:, 0], centroids[:, 1], color='red', marker='X', s=100,
label='Centroids')

  # Labeling
  plt.title(f'K-Means Clustering with Voronoi Regions: random initialization {i+1}')
  plt.xlabel('Feature1')
  plt.ylabel('Feature2')
  plt.legend()
  plt.show()
```

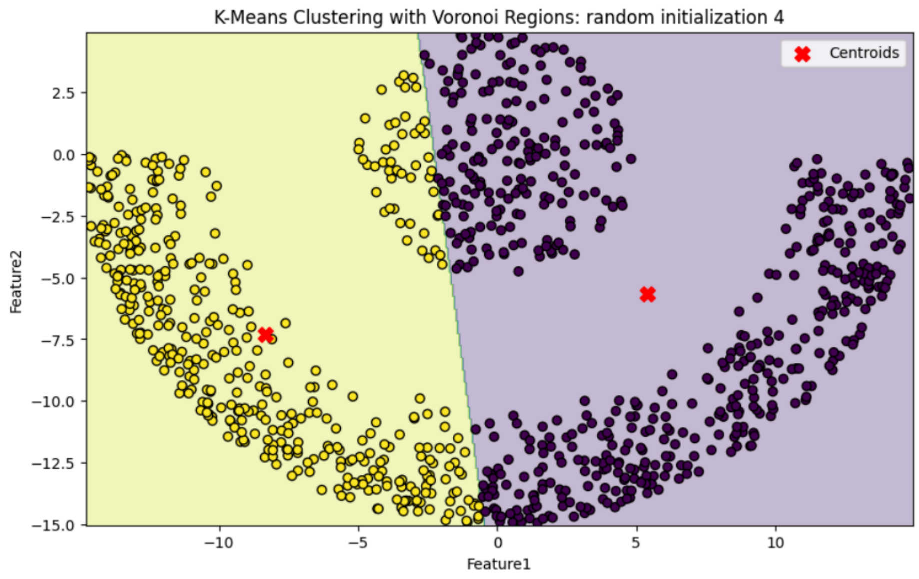
```
# Plotting the error
plt.figure(figsize=(10, 6))
for i, lst in enumerate(F_values):
    x_values = range(len(lst))
    plt.plot(x_values,lst, label=f'Initialization {i+1}')



plt.ylim(0, max(max(lst) for lst in F_values))
plt.xlabel('Iterations')
plt.ylabel('Error')
plt.title('Error vs Iterations')
plt.legend()
plt.show()
```

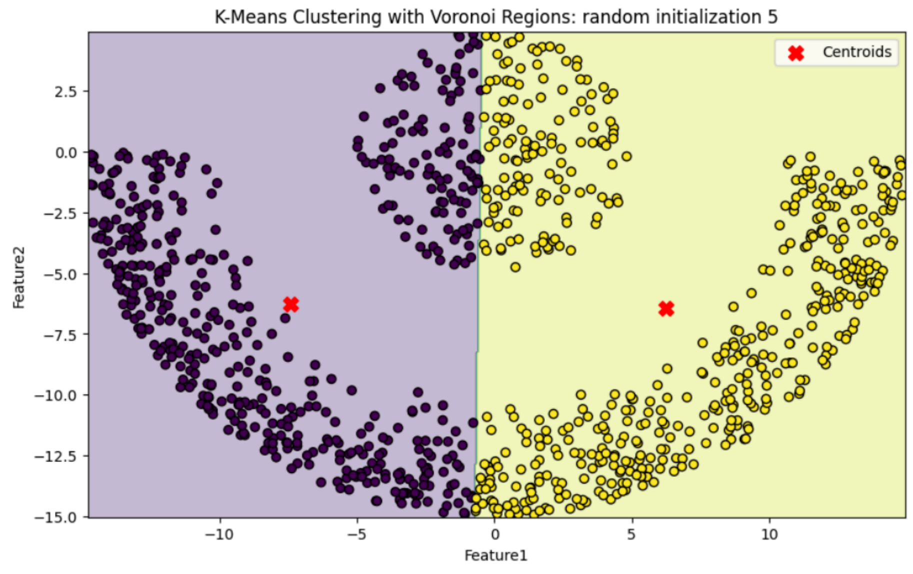
The plots for different initialization and the error vs iteration plot are the outputs



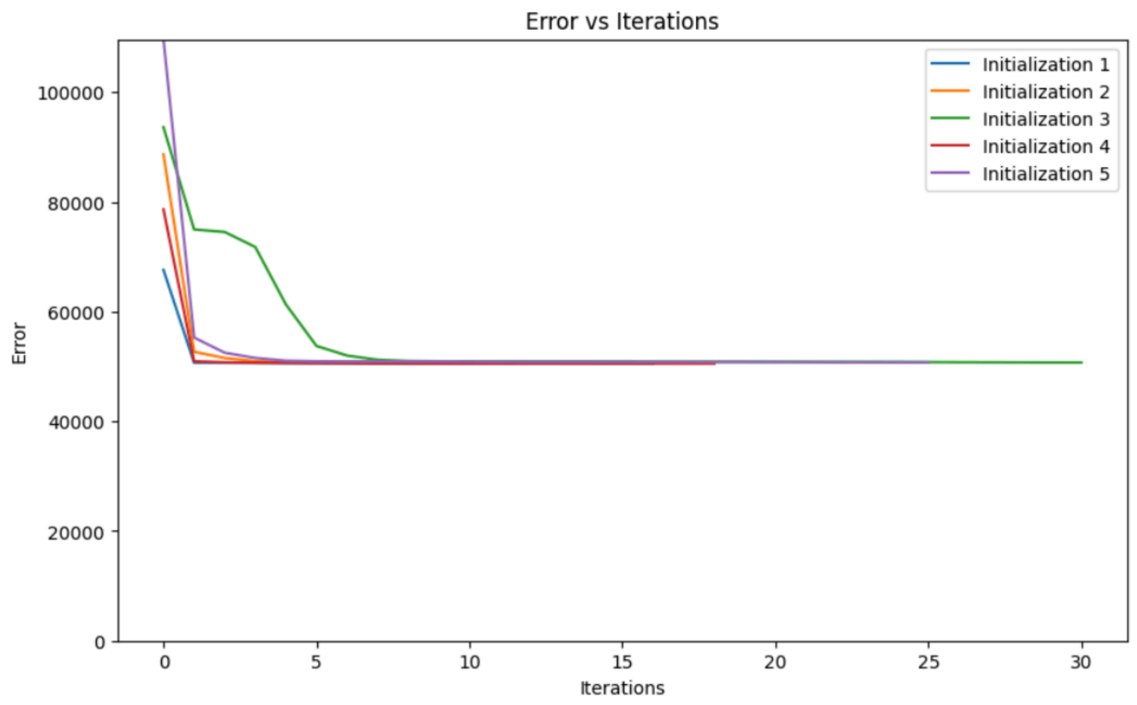








The error vs iteration graph:



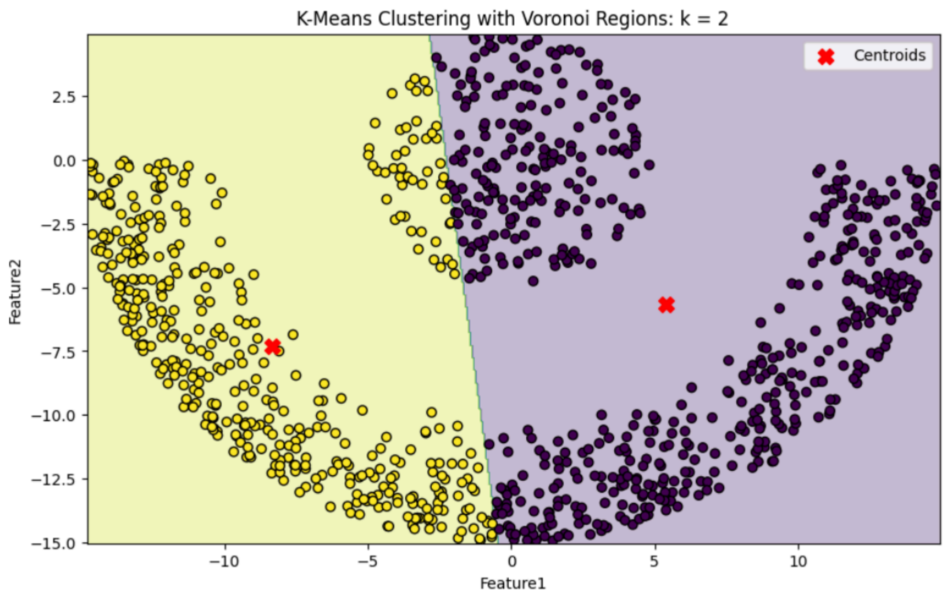
ii) For each K = {2, 3, 4, 5}, Fix an arbitrary initialization and obtain cluster centers according to K-means algorithm using the fixed initialization. For each value of K, plot the Voronoi regions associated to each cluster center. (You can assume the minimum and maximum value in the data-set to be the range for each component of R2).

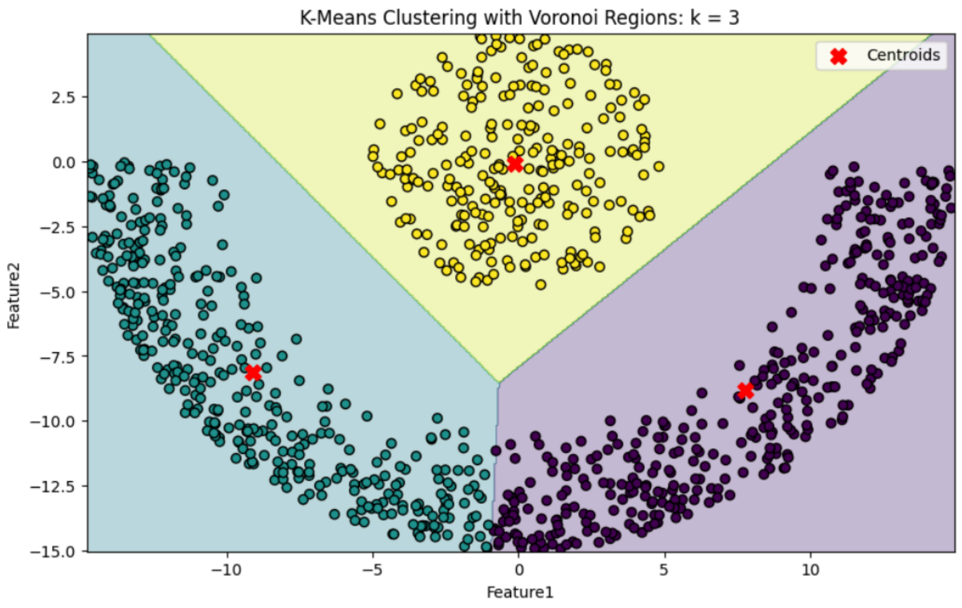
The code to implement the following is given below

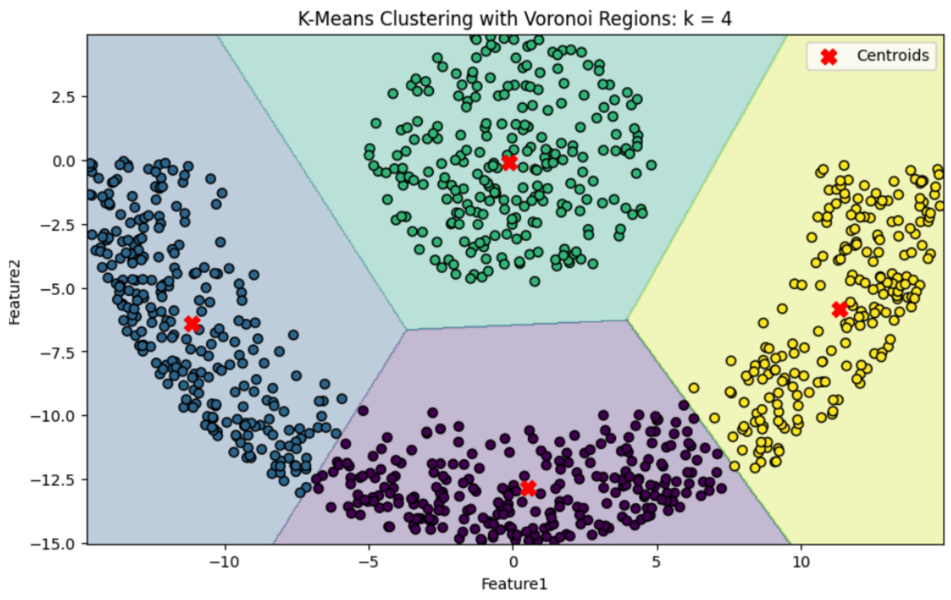
```
def llyods_algo(data, k=2, max_iters=100):
    np.random.seed(5400)
    centroids = data[np.random.choice(data.shape[0], k, replace=False)]
    err = []
    for i in range(max_iters):
        distances = np.linalg.norm(data[:, np.newaxis] - centroids, axis=2)
        cluster_assignments = np.argmin(distances, axis=1)
                                   error    =    np.sum([np.linalg.norm(data[j]    -
centroids[cluster_assignments[j]])**2 for j in range(data.shape[0])])
        err.append(error)

          new_centroids = np.array([data[cluster_assignments == j].mean(axis=0) for j
in range(k)])

         if np.allclose(centroids, new_centroids):  # comment these two lines, to be
able to zoom out the graph
             break                                 #

        centroids = new_centroids
    return centroids, cluster_assignments, err
```

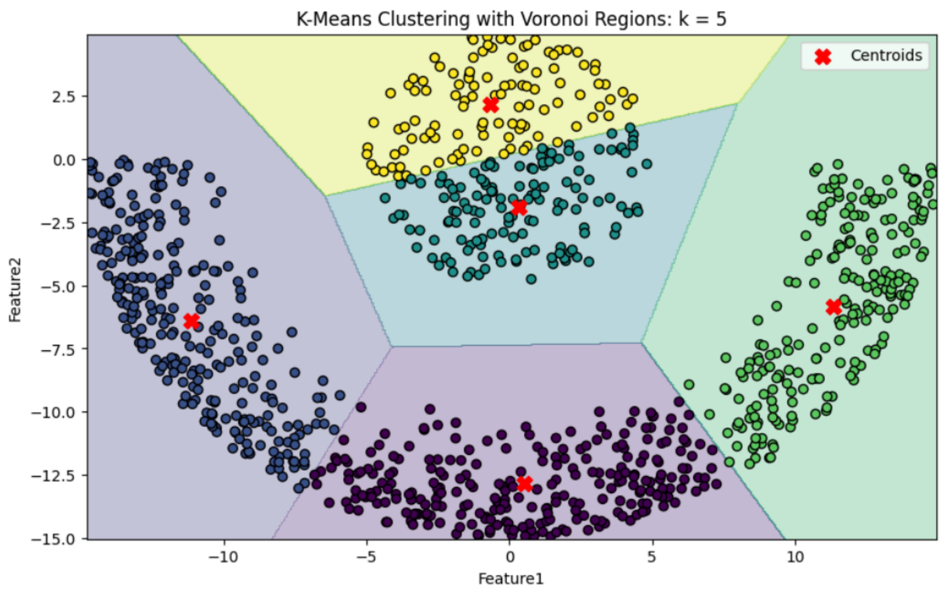
```
F_values = []
for i in range(2,6):
  data_points = df.values
  centroids, cluster_assignments, F_value = llyods_algo(data_points, k=i)
  F_values.append(F_value)
  # Generate grid points over the range of the data
  x_min, x_max = data_points[:, 0].min() - 0.1, data_points[:, 0].max() + 0.1
  y_min, y_max = data_points[:, 1].min() - 0.1, data_points[:, 1].max() + 0.1
  xx, yy = np.meshgrid(np.linspace(x_min, x_max, 500), np.linspace(y_min, y_max,
500))

  grid_points = np.c_[xx.ravel(), yy.ravel()]
  distances = np.linalg.norm(grid_points[:, np.newaxis] - centroids, axis=2)
  grid_assignments = np.argmin(distances, axis=1)
  grid_assignments = grid_assignments.reshape(xx.shape)

  # Voranoi
  plt.figure(figsize=(10, 6))
  plt.contourf(xx, yy, grid_assignments, cmap='viridis', alpha=0.3)
  plt.scatter(data_points[:, 0], data_points[:, 1], c=cluster_assignments,
cmap='viridis', edgecolor='k')
  plt.scatter(centroids[:, 0], centroids[:, 1], color='red', marker='X', s=100,
label='Centroids')

  # Labeling
  plt.title(f'K-Means Clustering with Voronoi Regions: k = {i}')
  plt.xlabel('Feature1')
  plt.ylabel('Feature2')
  plt.legend()
  plt.show()


# Plotting the error
plt.figure(figsize=(10, 6))
for i, lst in enumerate(F_values):
    x_values = range(len(lst))
    plt.plot(x_values,lst, label=f'K = {i+2}')
plt.ylim(0, max(max(lst) for lst in F_values))
plt.xlabel('Iterations')
plt.ylabel('Error')
plt.title('Error vs Iterations')
plt.legend()
plt.show()
```

The output of which is

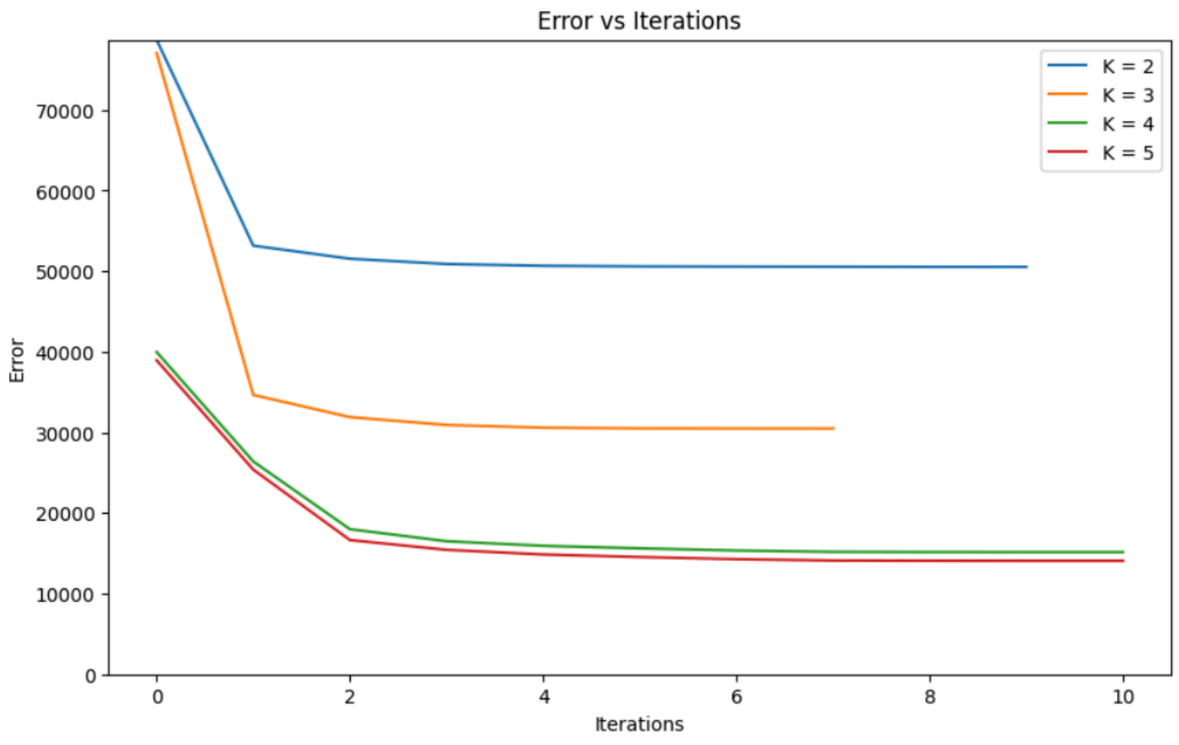








The error for each of these cases are



The error function used is

$$F\underline{(Z_1, \dots, Z_n)} = \sum_{i=1}^{n} \|x_i - \mu_{z_i}\|^2$$

$$\forall k = 1, \dots K \quad \mu_k = \frac{\sum_{i=1}^{n} x_i \underbrace{\mathbb{1}(z_i = k)}}{\sum_{i=1}^{n} \mathbb{1}(z_i = k)}$$

INDICATOR

$$\mathbb{1}(P) = \begin{cases} 1 & \text{if } P \text{ is true} \\ 0 & P \text{ is false} \end{cases}$$

iii) Is the Llyod's algorithm a good way to cluster this dataset? If yes, justify your answer. If not, give your thoughts on what other procedure would you recommend to cluster this dataset?

The llyod's algorithm creates voronoi regions which are linear in nature, the dataset provided here is not linearly separable, which is why the Llyod's algorithm is performing poorly for this case. Kernelized version of the same algorithm also called spectral clustering or kernelized llyod's algorithm would be a good algorithm to cluster this dataset.

1) Select gaussian or a polynomial kernel
2) Find apply kernel function for all possible i, j data points to create the kernel matrix
3) Find top k eigenvectors of that kernel matrix
4) Create matrix H = [h1, h2, h3 … hk]
5) Normalize the rows of H
6) Run Llyod's algorithm on H's rows to find the classification