Assignment-3

Data Preprocessing

This whole preprocessing is done in `Data_Preprocessing.py` file.

Libraries and Setup

1. **Imported Libraries**:
   - **Pillow (PIL)**: To handle and process images.
   - **NumPy**: For numerical operations, such as creating and manipulating arrays.
   - **Pandas**: For loading and manipulating datasets stored in parquet and CSV formats.
   - **os**: For handling file paths.
2. **Defining the Main Directory**:
   - `main_dir` sets the directory containing the script file using `os.path.dirname(__file__)`.
   - This ensures all file operations are relative to the script's location, making the code portable.

Loading Datasets

1. **Image Data for PCA**:
   - The dataset containing images is in `test-00000-of-00001.parquet`, loaded using Pandas' `read_parquet` function with the PyArrow engine.
   - This dataset contains:
     - A `label` column: The digit label (0–9) for each image.
     - An `image` column: Images stored as byte objects.
2. **Data for K-Means**:
   - A dataset `cm_dataset_2.csv` is loaded into a Pandas DataFrame using `read_csv`.
   - The data contains two numerical features: `x1` and `x2`.
   - It is converted to a NumPy array (`array_lloyd`) for further processing.

Ensuring Consistent Results

- **Random Seed**:
  - `np.random.seed(400)` ensures reproducibility. The same random numbers are generated each time the code runs, enabling consistent experiments.

Sampling 100 Images per Label for PCA

1. **Setting Up Sampling**:
   - A loop iterates through digit labels (`0` to `9`).
   - For each label:
     - It finds all the rows in `data` corresponding to that label.
     - From these rows, 100 indices are randomly selected without replacement using `np.random.choice`.
2. **Result**:
   - `indicies` becomes a list of 10 arrays, where each array contains 100 indices corresponding to a specific digit label.

Processing Images

1. **Setup**:
   - `image_data` is initialized as an empty array with shape `(1, 784)` (a placeholder for flattened 28×28 images).
   - `label_data` is initialized as an empty list to store the corresponding digit labels.
2. **Image Processing**:
   - The code loops through each label (0–9) and its 100 sampled indices.
   - For each index:
     - **Retrieve Image Bytes**: Loads the image bytes stored in the dataset's `image` column using `io.BytesIO`.
     - **Convert to Grayscale**: Converts the image to grayscale (`'L'` mode) using `Image.convert('L')`.
     - **Resize Image**: Resizes the image to 28×28 pixels using `Image.resize()` with `Image.LANCZOS` for high-quality downsampling.
     - **Flatten**: Converts the 28×28 image into a 1D array of shape `(784,)`.
     - **Update Arrays**:
       - Adds the processed image to `image_data` using `np.concatenate`.
       - Appends the label to `label_data`.
3. **Remove Placeholder Row**:
   - The first row of `image_data` is a placeholder and is removed using `image_data = image_data[1:]`.

Final Output

1. `image_data`:
    - A NumPy array of shape `(1000, 784)`, where each row is a flattened grayscale image representing a digit.
2. `label_data`:
    - A list of 1000 integers, where each element is the digit label (0–9) for the corresponding image in `image_data`.

- **PCA Task**:
    - The processed `image_data` is now ready for dimensionality reduction using PCA.
    - Labels in `label_data` can be used for downstream evaluation (e.g., clustering or classification).
- **K-Means Task**:
    - The numerical data from `cm_dataset_2.csv` is ready for clustering using K-means.

1. Question)

I have implemented code for PCA in `principal_component_analysis.py` file and the implemented code is explained below

Class Overview

1. **Initialization (`__init__`)**:
    - Accepts `components` as a parameter, which specifies the number of principal components to retain.
    - Default is set to 2 components.
2. **Fitting the Data (`fit`)**:
    - **Step 1: Mean Calculation**:
        - Computes the mean of each feature (column) in the dataset and stores it in `self.mu`.
    - **Step 2: Centering the Data**:
        - Subtracts the mean from the dataset to create a mean-centered version.
    - **Step 3: Covariance Matrix**:
        - Computes the covariance matrix of the mean-centered data.

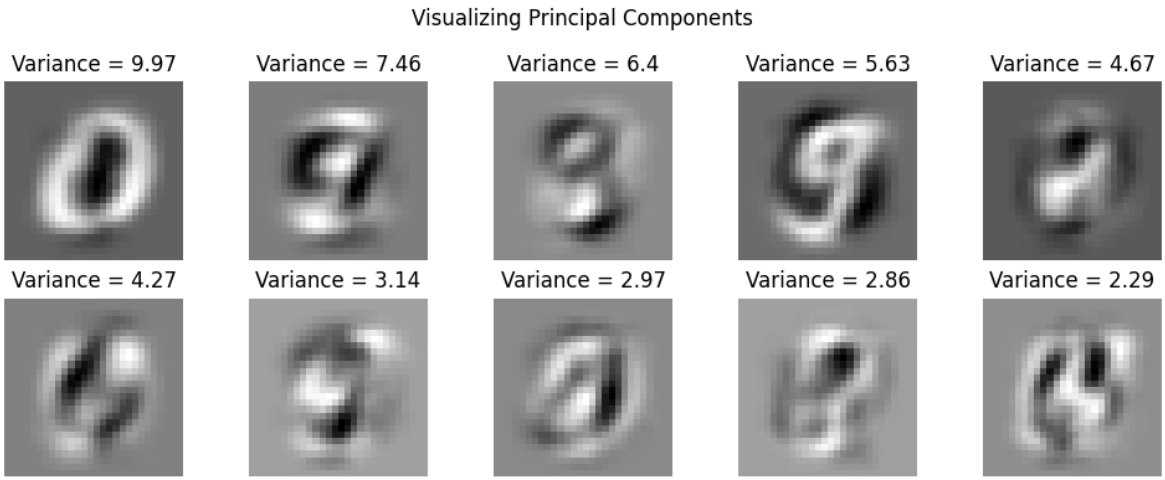
- **Step 4: Eigen Decomposition**:
  - Uses `np.linalg.eigh` to compute eigenvalues and eigenvectors of the covariance matrix.
  - `eigh` is preferred for symmetric matrices like covariance matrices.
- **Step 5: Sort and Select Components**:
  - Sorts eigenvalues and eigenvectors in descending order of eigenvalues.
  - Selects the top `components` eigenvectors (principal components) for dimensionality reduction.
- **Step 6: Transform Data**:
  - Projects the original mean-centered data onto the principal components to obtain the reduced data (`self.transformed_data`).

3. **Reconstruction (`reconstruct`)**:
   - Reconstructs the original data from the reduced representation.
   - This is done by reversing the PCA transformation:
     Where:

$$X_{reconstructed} = Z \cdot W^T + \mu$$

     - Z is the reduced data.
     - W are the principal components.
     - μ is the mean vector.
4. **Getter Methods**:
   - `get_mean()`: Returns the mean vector of the dataset.
   - `get_eigenvalues()`: Returns all eigenvalues.
   - `get_eigenvectors()`: Returns all eigenvectors.

(i) Visualizing the images of the principal components that I obtain and the variance in the data-set is explained by each of the principal components



This visualization was obtained using below code which is implemented in `main.py` file:

```
def component_visualization(components, sum_values, eigenvalues,
                            number_columns=5, image_shape=(28, 28)):
    number_of_images = components.shape[1]
    # Set up the grid for plotting
    rows = (number_of_images + number_columns - 1) // number_columns
    fig, axes = plt.subplots(rows, number_columns, figsize=(10, 2 * rows)

    for i in range(number_of_images):
        image = components[:, i].reshape(image_shape)
        variance_explained = eigenvalues[i]/sum_values
        ax = axes[i // number_columns, i % number_columns]
                  if rows > 1 else axes[i % number_columns]
        ax.imshow(image, cmap="gray")
        ax.set_title(f'Variance = {round(variance_explained*100,2)}')
        ax.axis("off")
    plt.suptitle('Visualizing Principal Components')
    plt.tight_layout()
    plt.show()
```

Code Explanation

Parameters

1. `components`:
    - A 2D NumPy array where each column represents a principal component (eigenvector).

- Shape: `(n_features, n_components)`, where `n_features` is the flattened size of the original image.

2. `sum_values`:
   - The total sum of eigenvalues, representing the total variance in the data.
   - Used to compute the percentage of variance explained by each component.
3. `eigenvalues`:
   - A 1D array of eigenvalues corresponding to the principal components.
   - Each eigenvalue represents the variance captured by its associated principal component.
4. `number_columns` **(default: 5)**:
   - Number of columns in the grid of visualized components.
5. `image_shape` **(default:** `(28, 28)`**)**:
   - The original shape of the image for reconstructing visual representations of the principal components.

Steps

1. **Calculate Rows for Grid Layout**:
   - The number of rows is calculated based on the total number of images (`components.shape[1]`) and the number of columns (`number_columns`).
2. **Create Plot Grid**:
   - Uses `plt.subplots` to create a grid of subplots with dimensions `rows x number_columns`.
3. **Iterate Over Principal Components**:
   - For each principal component:
     - Reshape the component into the original image shape using `.reshape(image_shape)`.
     - Compute the percentage of variance explained:

$$\text{Variance Explained by each principal component} = \frac{\text{Eigenvalue}}{\text{Sum of All Eigenvalues}}$$

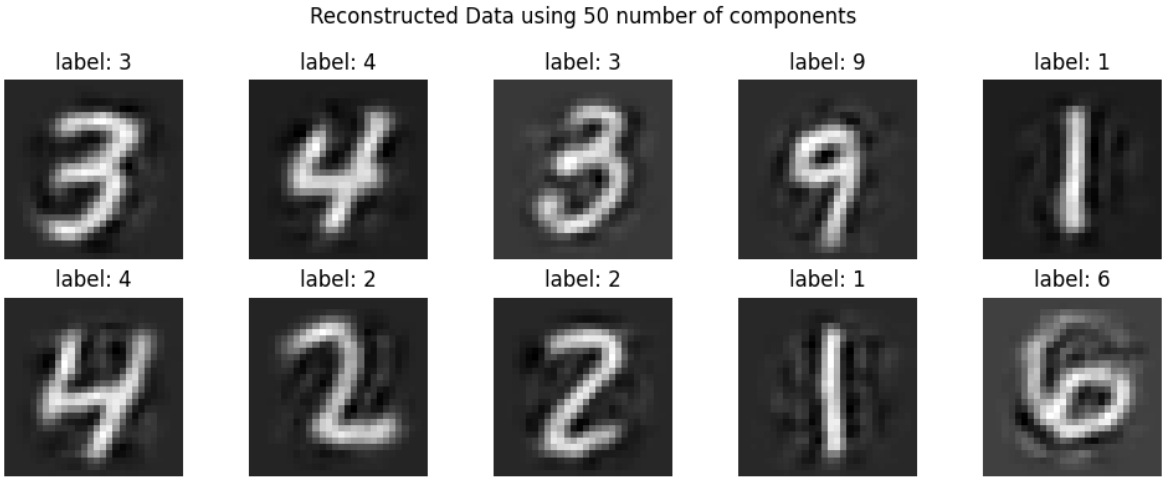
     - Display the reshaped component as an image in grayscale using `imshow`.
     - Add a title to the subplot showing the variance explained (rounded to two decimal places).
4. **Configure and Display the Plot**:

- The plot is titled "Visualizing Principal Components."
- `plt.tight_layout()` ensures that subplots do not overlap.
- `plt.show()` renders the visualization.

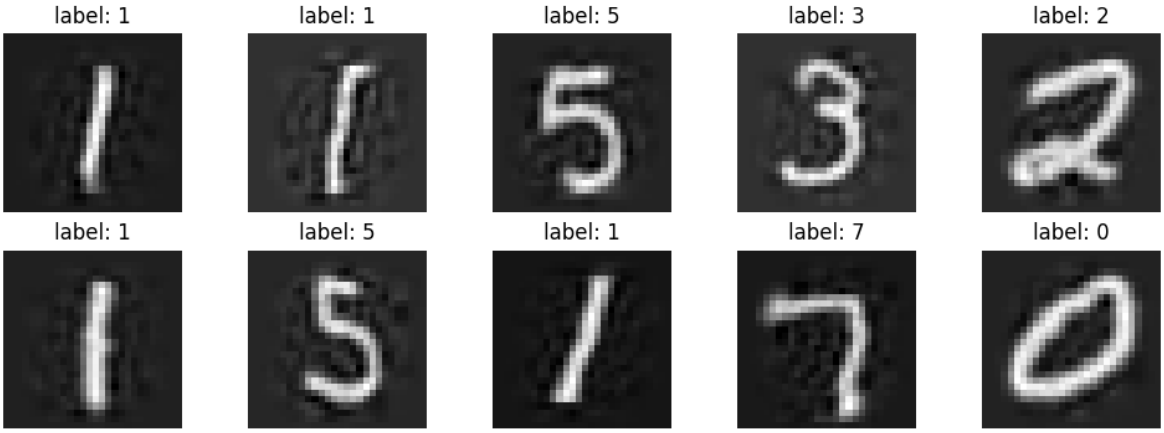
Visualization Explanation

- **Images**:
  - Each subplot corresponds to a principal component.
  - Grayscale values represent the component's effect on the original image features.
- **Variance Explained**:
  - Indicates the proportion of total variance captured by each component.
  - Useful to decide how many components to retain for tasks like dimensionality reduction.

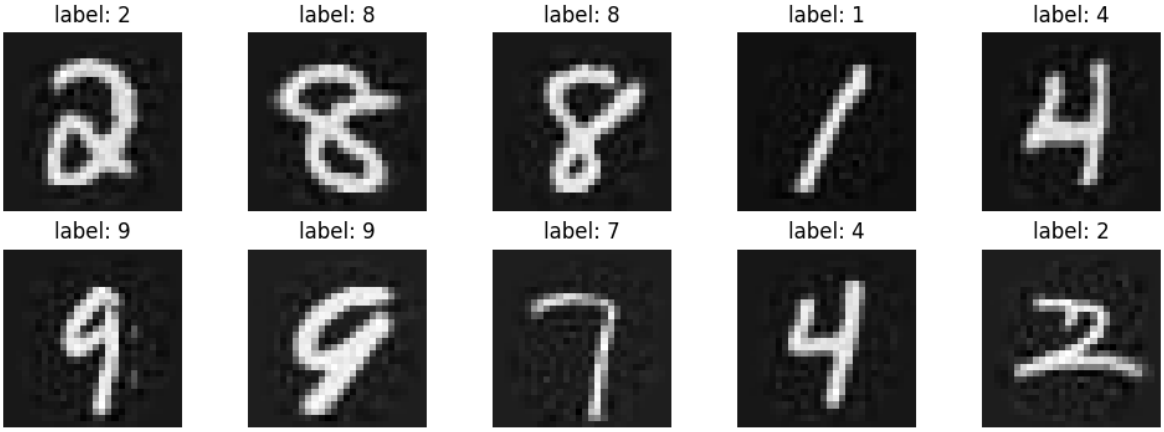
(ii) Reconstruct the dataset using different dimensional representations



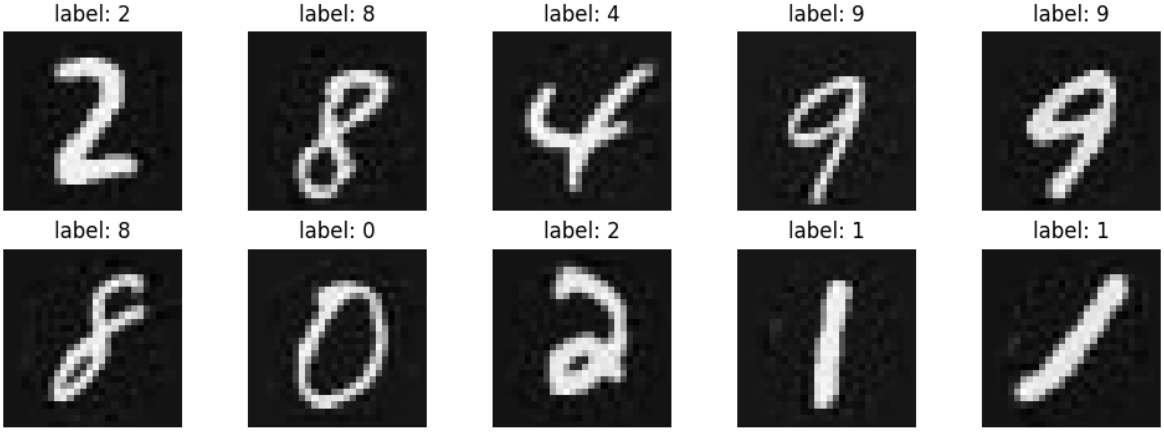
Reconstructed Data using 100 number of components

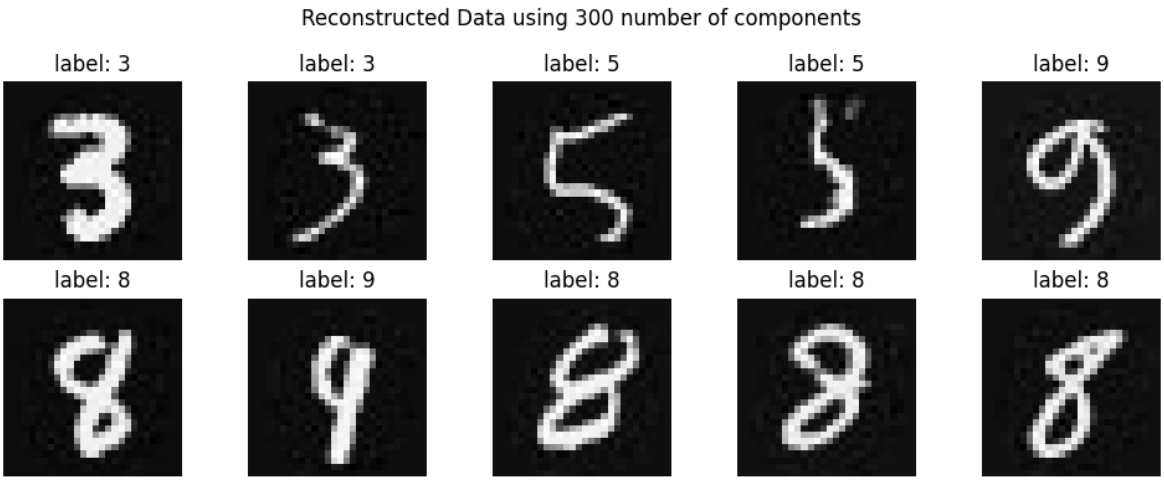


Reconstructed Data using 200 number of components



Reconstructed Data using 250 number of components





I will pick d=128 as as at this dimension variance explained by components are > 95%.

2. Question)

(i) (a) implement the Llyod's algorithm for the K-means problem with k = 2

The Llyod's algorithm was implemented in `k_means.py` file and code for the same is:

```
import numpy as np

class Llyoid_Algorithm:
    def __init__(self, number_of_clustors=2):
        self.number_of_clustors = number_of_clustors

    def fit(self, X, iterations=100, random_state=400):
        np.random.seed(random_state)
        self.error = []
        n, d = X.shape
        data_labels = np.random.randint(0, self.number_of_clustors, size=
        for _ in range(iterations):
            means = np.array([(X[data_labels == i]).mean(axis=0)
                              for i in range(self.number_of_clustors)])
            distances = np.linalg.norm(X[:, np.newaxis] - means, axis=2)
            data_labels = np.argmin(distances, axis=1)
            self.error.append(np.sum((X - means[data_labels])**2)/n)
        self.means = np.array([(X[data_labels == i]).mean(axis=0)
                               for i in range(self.number_of_clustors)])
        return data_labels
```

```
    def get_error(self):
        return self.error

    def get_mean(self):
        return self.means
```

Class Overview

The `Llyoid_Algorithm` class clusters data points into a specified number of clusters (`number_of_clustors`). It assigns each data point to the nearest cluster and iteratively adjusts the cluster centers (means) to minimize the total intra-cluster variance.

Key Components

Initialization (`__init__`)

```
def __init__(self, number_of_clustors=2):
    self.number_of_clustors = number_of_clustors
```

- `number_of_clustors`:
  - Specifies the number of clusters to divide the data into. Default is 2.
- **Purpose**:
  - Prepares the class to store cluster-related information such as means and errors.

Fit Method (`fit`)

This method performs the actual K-Means clustering on a given dataset `X`.

```
def fit(self, X, iterations=100, random_state=400):
    np.random.seed(random_state)  # Fix randomness for reproducibility
    self.error = []  # Initialize list to store errors for each iterat
ion
    n, d = X.shape  # n: number of points, d: dimensions per point
```

1. **Random Initialization**:
   - Randomly assigns each data point a cluster label (`data_labels`).
   - Ensures results are reproducible with `np.random.seed(random_state)`.
2. **Iterative Refinement**:

- Repeats the following steps for `iterations`:
  - **Step 1: Calculate Cluster Means**:

    ```
    means = np.array([(X[data_labels == i]).mean(axis=0) for i in
    range(self.number_of_clustors)])
    ```

    - Computes the mean (centroid) of points in each cluster. For cluster i, it takes the mean of all points labeled as i.
  - **Step 2: Update Labels**:

    ```
    distances = np.linalg.norm(X[:, np.newaxis] - means, axis=2)
    data_labels = np.argmin(distances, axis=1)
    ```

    - Computes the distance of each point to all centroids using the Euclidean norm.
    - Assigns each point to the nearest centroid (cluster) by finding the index of the minimum distance.
  - **Step 3: Compute Error**:

    ```
    self.error.append(np.sum((X - means[data_labels])**2)/n)
    ```

    - Computes the average intra-cluster variance (sum of squared distances from each point to its cluster center).
    - Appends this error to `self.error` for tracking.

3. **Final Means**:
   - After completing all iterations, calculates the final cluster centroids (`self.means`).
4. **Returns Cluster Assignments**:
   - Returns the final cluster assignments for each point.

Getter Methods

1. `get_error()`:

   ```
   def get_error(self):
       return self.error
   ```

   - Returns the list of errors (intra-cluster variances) computed during each iteration.
2. `get_mean()`:

```
def get_mean(self):
    return self.means
```

- Returns the final cluster centroids after the algorithm has converged.

Explanation of Outputs

1. `cluster_labels`:
   - Final cluster assignments for each data point.
2. `get_mean()`:
   - The final centroids (means) of the clusters.
3. `get_error()`:
   - A list showing how the intra-cluster variance decreases with each iteration.

(i) (b) Try 5 different random initialization and plot the error function w.r.t iterations in each case. Also plot their respective obtained clusters.

This was implemented in `main.py` file and code for that is:

```
random_initials = [400, 300, 1221, 1044, 1001]
for i in range(len(random_initials)):
    model = Llyoid_Algorithm(number_of_clustors=2)
    cluster_label = model.fit(array_lloyd, random_state=random_initials[i
    error = model.get_error()
    plt.subplot(1, 2, 1)
    plt.plot(np.arange(10), error)
    plt.title(f'Error Vs Iterations for {i+1} Random Initialization')
    plt.xlabel('Number of Iterations')
    plt.ylabel('Error')
    plt.subplot(1, 2, 2)
    plt.scatter(x = lloyd_data.x1, y = lloyd_data.x2, c=cluster_label)
    plt.title(f'Clusters for {i} Random Initialization')
    plt.show()
```

Class Overview

1. Random Initializations

```
random_initials = [400, 300, 1221, 1044, 1001]
```

- **Purpose**: Specifies five different random seeds to initialize the cluster labels.
- Each seed ensures different starting cluster assignments, which can influence the final result due to the random nature of K-Means.

2. Loop Over Random Seeds

```
for i in range(len(random_initials)):
    model = Llyoid_Algorithm(number_of_clustors=2)
    cluster_label = model.fit(array_lloyd, random_state=random_initial
s[i], iterations=10)
```

- `model = Llyoid_Algorithm(...)`:
  - Creates an instance of the `Llyoid_Algorithm` class for clustering into 2 clusters.
- `fit(...)`:
  - Fits the model to the dataset `array_lloyd` using the specified random seed (`random_state`) and runs for 10 iterations.
  - Returns `cluster_label`: The cluster assignments for each data point.

3. Retrieve Error Over Iterations

```
error = model.get_error()
```

- `get_error()`:
  - Retrieves the list of error values (intra-cluster variance) for each of the 10 iterations.

4. Plot Error vs. Iterations

```
plt.subplot(1, 2, 1)
plt.plot(np.arange(10), error)
plt.title(f'Error Vs Iterations for {i+1} Random Initialization')
plt.xlabel('Number of Iterations')
plt.ylabel('Error')
```

- **Explanation**:

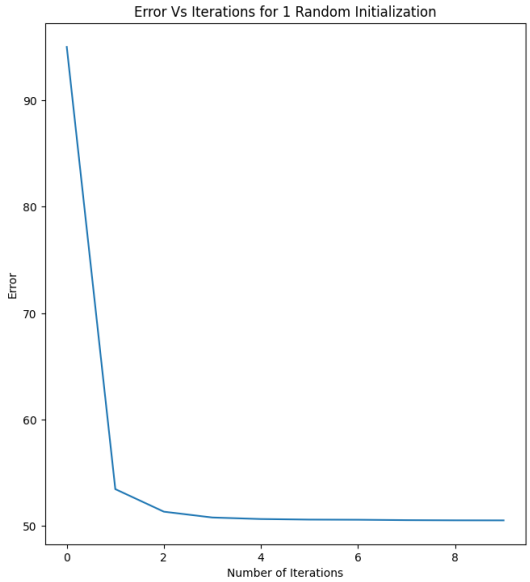
- As iterations progress, the error typically decreases and stabilizes as the clusters converge

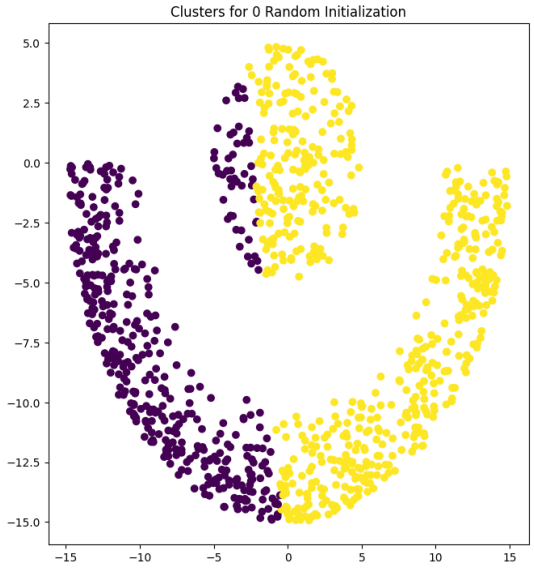
5. Plot Clusters

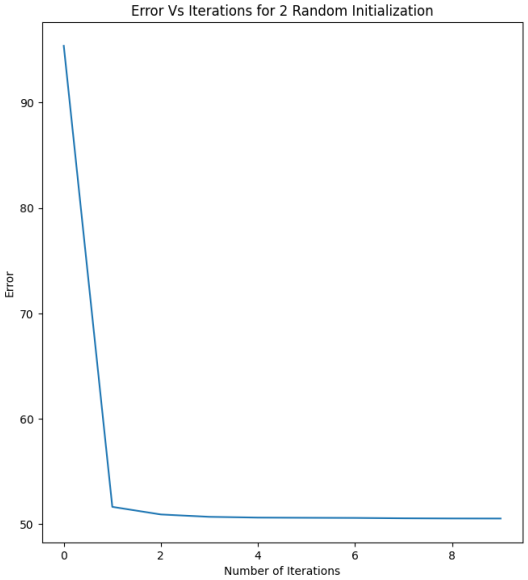
```
python
Copy code
plt.subplot(1, 2, 2)
plt.scatter(x = lloyd_data.x1, y = lloyd_data.x2, c=cluster_label)
plt.title(f'Clusters for {i} Random Initialization')
```

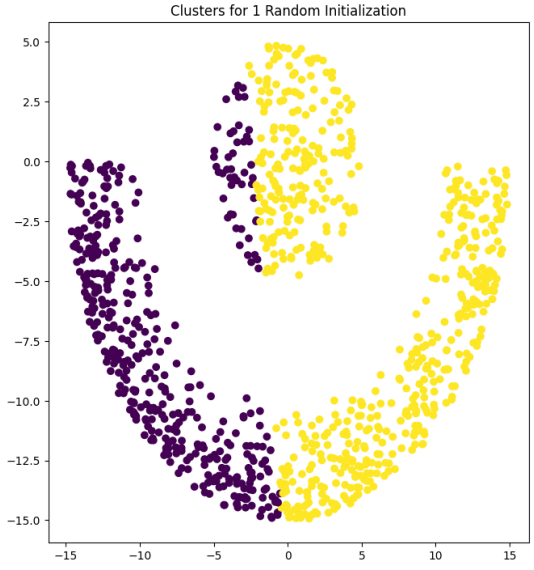
- **Purpose**:
  - Displays the clustering result for the given random initialization.
  - Each data point is colored according to its cluster label (`c=cluster_label`).

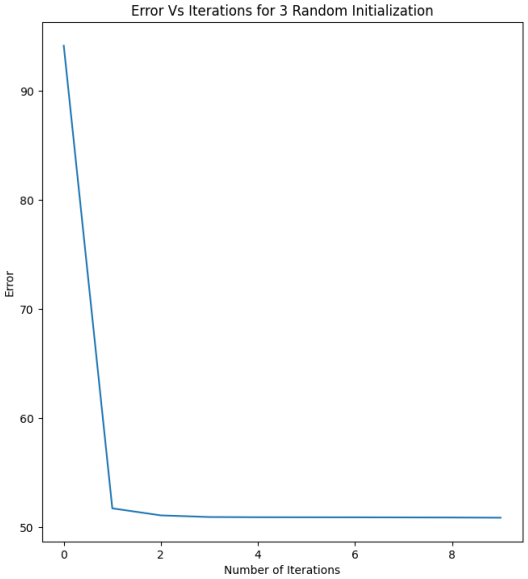
The Output of this code is:

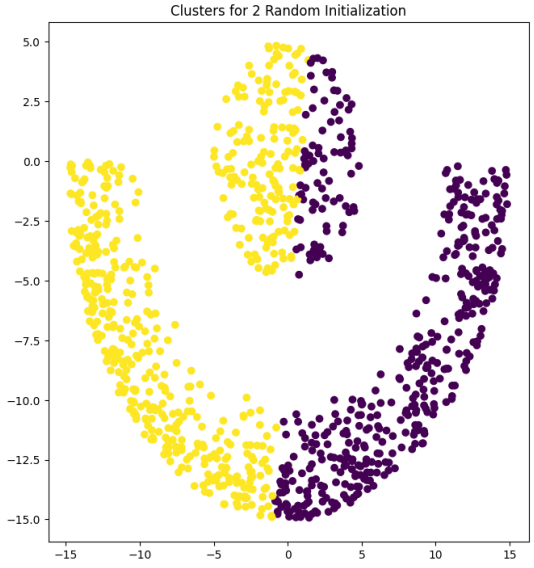


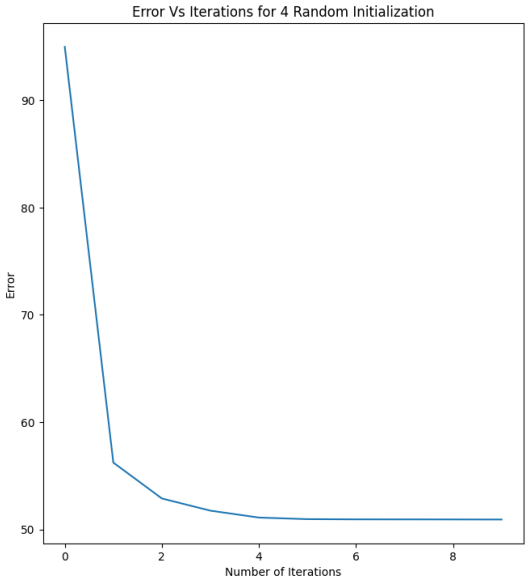


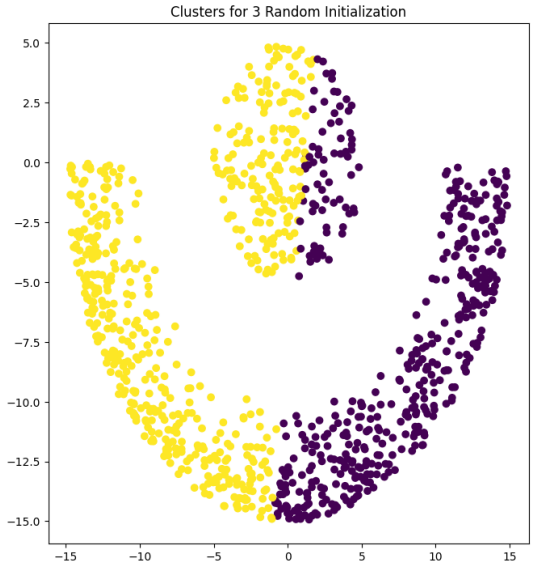

Error Vs Iterations for 2 Random Initialization
Error
Number of Iterations
90
80
70
60
50
0
2
4
6
8

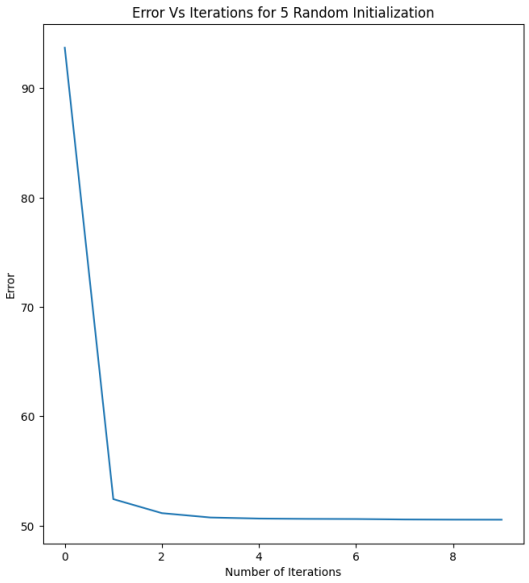

Clusters for 1 Random Initialization
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−12.5
−15.0
−15
−10
−5
0
5
10
15

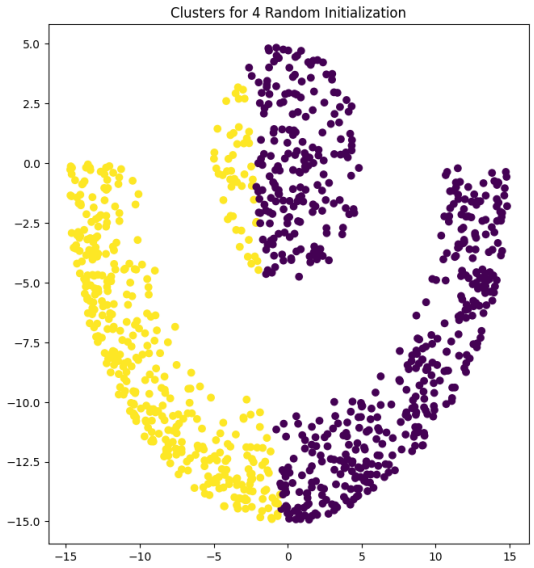

Error Vs Iterations for 3 Random Initialization
Error
Number of Iterations
90
80
70
60
50
0
2
4
6
8


Clusters for 2 Random Initialization
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−12.5
−15.0
−15
−10
−5
0
5
10
15









This 5 different plot for different random initialization and there respective cluster plots.

(ii) For each K = {2, 3, 4, 5} with fixed initialization and plot the Voronoi regions associated to each cluster center.

The plot for Voronoi regions code is implemented in `main.py` file and code for same is:

```
def plot_voronoi(centroids, data):
    if len(centroids) > 3:
        vor = Voronoi(centroids)
        voronoi_plot_2d(vor, show_vertices=False, line_colors='orange', l
        plt.scatter(data[:, 0], data[:, 1], c="blue", s=1)
        plt.scatter(centroids[:, 0], centroids[:, 1], c="red", marker="x"
        plt.show()
    else:
        x_min, x_max = data[:, 0].min() - 0.5, data[:, 0].max() + 0.5
        y_min, y_max = data[:, 1].min() - 0.5, data[:, 1].max() + 0.5
        xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02), np.arange(y_m
        mesh_points = np.c_[xx.ravel(), yy.ravel()]
        distances = np.linalg.norm(mesh_points[:, np.newaxis] - means, ax
        labels = np.argmin(distances, axis=1)
        labels = labels.reshape(xx.shape)
        plt.contour(xx, yy, labels, alpha=0.3, cmap='viridis')
        plt.scatter(data[:, 0], data[:, 1], c='black', alpha=0.7, s=10)
        plt.scatter(centroids[:, 0], centroids[:, 1], c='red', marker='o'
        plt.title(f'Voroni Region for k={len(centroids)}')
        plt.show()


for k in range(2, 6):
    model = Llyoid_Algorithm(number_of_clustors=k)
    cluster_label = model.fit(array_lloyd)
    means = model.get_mean()
    plot_voronoi(centroids=means, data=array_lloyd)
```

Class Overview

1. Function `plot_voronoi`

This function visualizes Voronoi regions for cluster centroids.

1. **Case 1: More Than 3 Centroids**:

```
vor = Voronoi(centroids)
voronoi_plot_2d(vor, show_vertices=False, line_colors='orange', lin
e_width=2)
```

```
plt.scatter(data[:, 0], data[:, 1], c="blue", s=1)
plt.scatter(centroids[:, 0], centroids[:, 1], c="red", marker="x")
plt.show()
```

- Uses the `Voronoi` class from `scipy.spatial` to compute Voronoi for the given centroids.
- Plots the Voronoi regions using `voronoi_plot_2d`.
- Displays:
  - **Voronoi boundaries** in orange.
  - **Data points** in blue.
  - **Centroids** in red.

2. **Case 2: 3 or Fewer Centroids**:

```
x_min, x_max = data[:, 0].min() - 0.5, data[:, 0].max() + 0.5
y_min, y_max = data[:, 1].min() - 0.5, data[:, 1].max() + 0.5
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02), np.arange(y_mi
n, y_max, 0.02))
```

- Creates a grid of points (`xx, yy`) spanning the data's range to calculate the decision boundaries.
- Each grid point is assigned to the nearest centroid using Euclidean distance:

```
mesh_points = np.c_[xx.ravel(), yy.ravel()]
distances = np.linalg.norm(mesh_points[:, np.newaxis] - means, a
xis=2)
labels = np.argmin(distances, axis=1).reshape(xx.shape)
```

- The `plt.contour` function visualizes these regions.
- Data points and centroids are overlaid:
  - **Data points** in black.
  - **Centroids** in red.

2. Iterating Over k Clusters

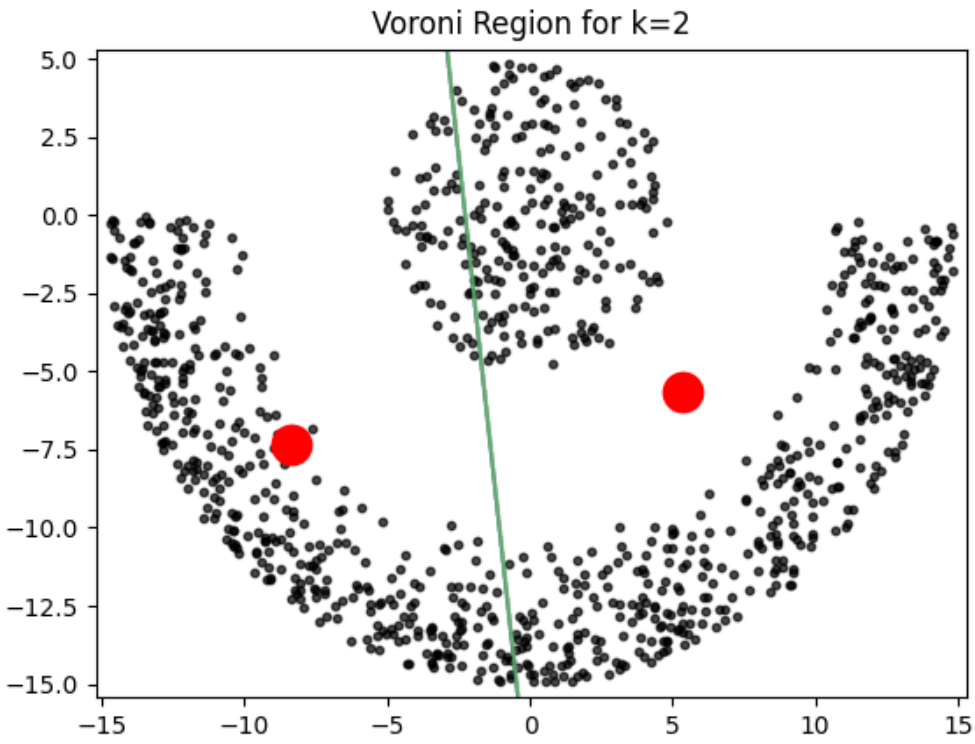
```
for k in range(2, 6):
    model = Llyoid_Algorithm(number_of_clustors=k)
    cluster_label = model.fit(array_lloyd)
```

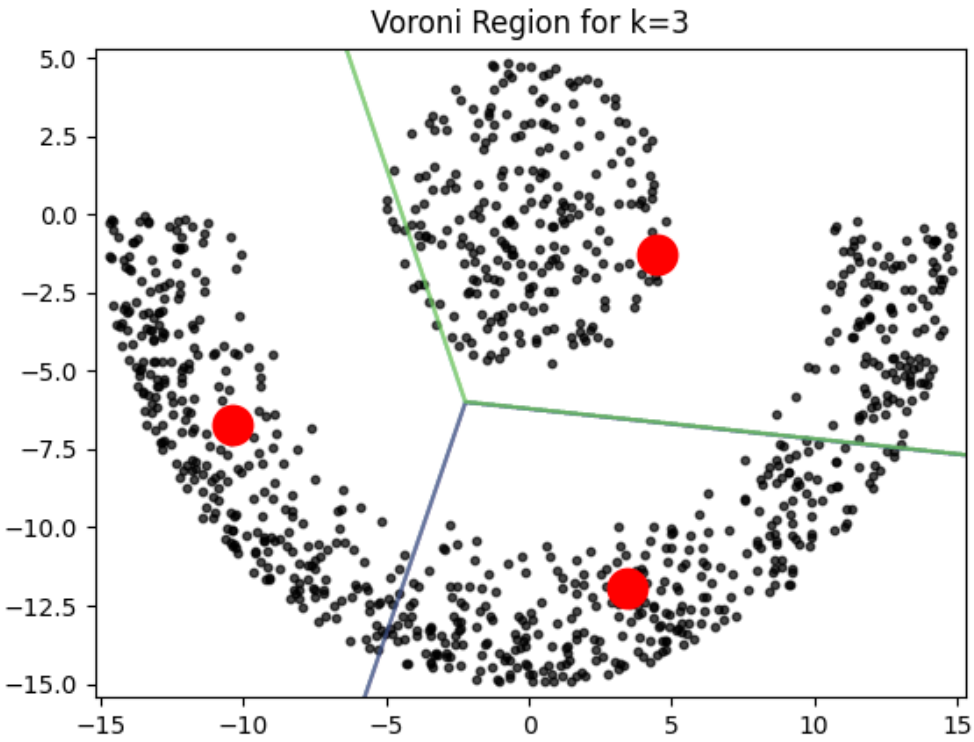
```
    means = model.get_mean()
    plot_voronoi(centroids=means, data=array_lloyd)
```

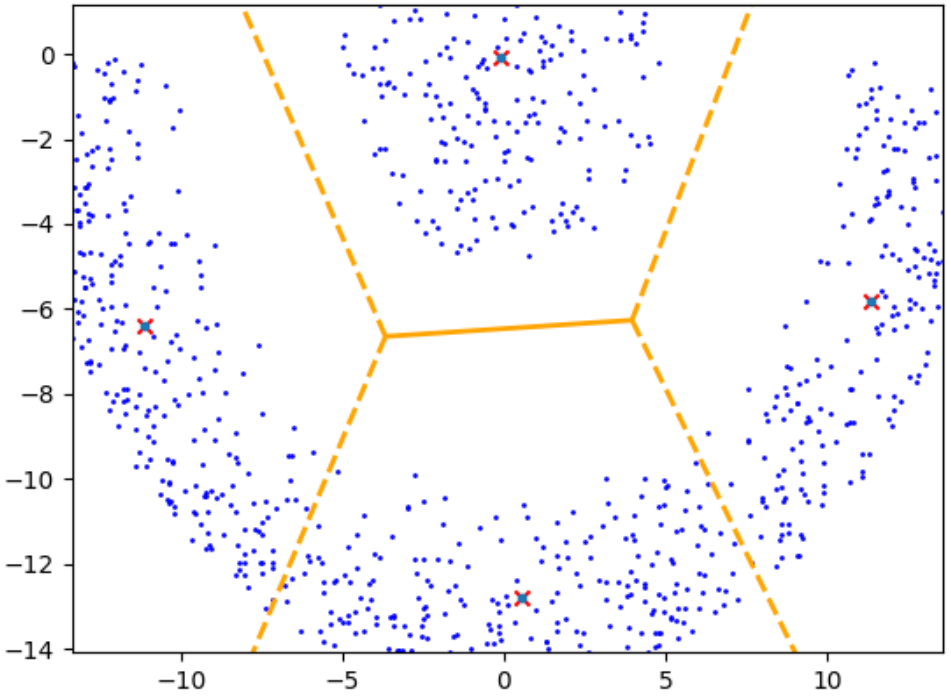
- **Loop Over k=2,3,4,5**:
  - Creates an instance of `Llyoid_Algorithm` for k clusters.

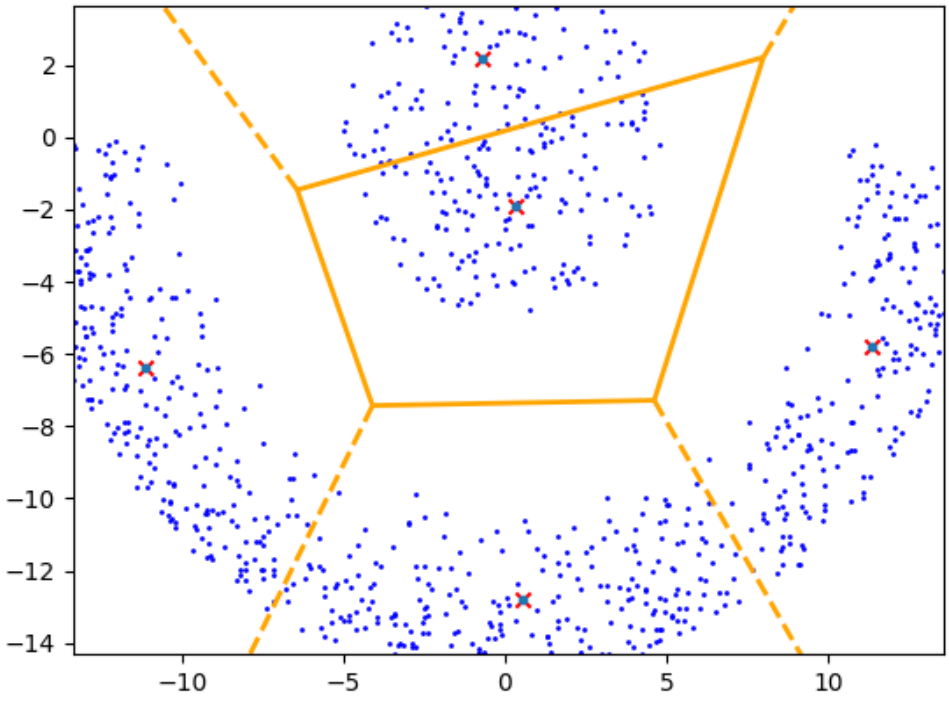
    kk
  - Fits the algorithm to the dataset `array_lloyd`.
  - Retrieves the final cluster centroids using `get_mean()`.
  - Calls `plot_voronoi` to visualize the clusters and their regions.

The output for this code will be




Voroni Region for k=3
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−12.5
−15.0
−15
−10
−5
0
5
10
15


0
−2
−4
−6
−8
−10
−12
−14
−10
−5
0
5
10



(iii) Is the Llyod's algorithm a good way to cluster this dataset?

As we can see from data visualization that given data is not a linear data but not linear so, Lloyd's algorithm is **not a good way** to cluster this dataset. Instead we can use **Kernelized K-means** for this data set to let our model understand the non-linearity in the dataset.

Running The given file

- To properly run the given file it just need the data for PCA and K-Mean same same as `test-00000-of-00001.parquet` and `cm_dataset_2.csv` respectively and all the file that will be `main.py` , `k_means.py` , `principal_component_analysis.py` and `Data_Preprocessing.py` in that folder.
- You just have to run `main.py` file to get all outputs.