Implementing Principal Component Analysis and Lloyd’s algorithm for Clustering from scratch

by Anshuman Singh

Task-1

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.
    a. 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?
    b. 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?

PCA

Principal Component Analysis (PCA) is a dimensionality reduction technique that identifies the directions (principal components) along which the variation in the data is maximal.

1. **Data Centering:**

   PCA requires the data to be centered around the mean. If X is the dataset with n samples and d features:

   $$X_{\text{centered}} = X - \mu$$

   where μ is the mean of each feature (calculated along columns)

   and X is the dataset matrix with dimensions $n * d$ containing one sample per row

2. **Covariance Matrix:**

   The covariance matrix represents how the features in the dataset vary with one another. The formula is:

   $$\text{Cov}(X) = \frac{1}{n} X_{\text{centered}}^T X_{\text{centered}}$$

3. **Eigenvalues and Eigenvectors:**
    a. The covariance matrix is symmetric, so it can be decomposed into eigenvalues (λ) and eigenvectors (v).

   This is expressed as:

   $$\text{Cov}(X)v = \lambda v$$

The eigenvalues and eigenvectors are sorted in descending order of eigenvalues, as the largest eigenvalues correspond to the most significant directions of variance.

4. **Dimensionality Reduction:**

   To reduce the dimensionality, select the top k eigenvectors corresponding to the largest k eigenvalues. This forms the projection matrix:

$$W = [v_1, v_2, \ldots, v_k]$$

   Project the data onto this lower-dimensional space:

$$X_{\text{projected}} = X_{\text{centered}} W$$

5. **Reconstruction:**

   To reconstruct the original data (approximation), project the reduced data back:

$$X_{\text{reconstructed}} = X_{\text{projected}} W^T$$

PCA has been implemented using the PCA class present in the PCA.py file.

To run the PCA code on the MNIST dataset, first the dataset was loaded using the Pandas library into a Pandas DataFrame. The data was present in the "bytes" format by default. Appropriate pre-processing was performed on the dataset to obtain each image as a vector form of shape 784. Then, a total of 1000 samples were taken from the dataset, so that each label digit has 100 samples each. This was done using the `balanced_subsample()` helper function given in the `Helpers.py` file. Sample images from the dataset are given in fig-1.

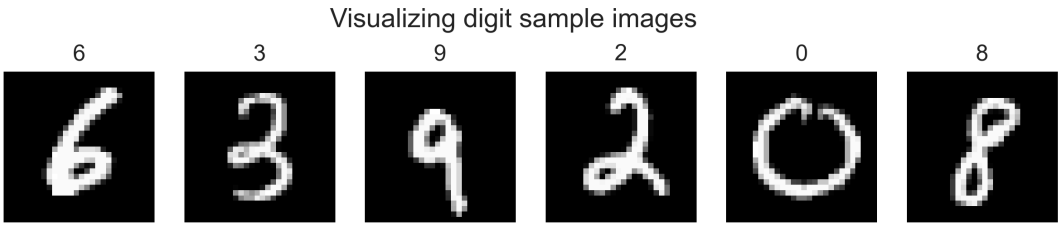


Fig-1: Visualizing some sample images from the given MNIST dataset

The images present in the dataset have a resolution of 28 × 28 pixels. After running PCA on the dataset, the principal components obtained were resized to 28 × 28 shape matrix and plotted as a heatmap using seaborn library for visualization. The results are given in fig-2.

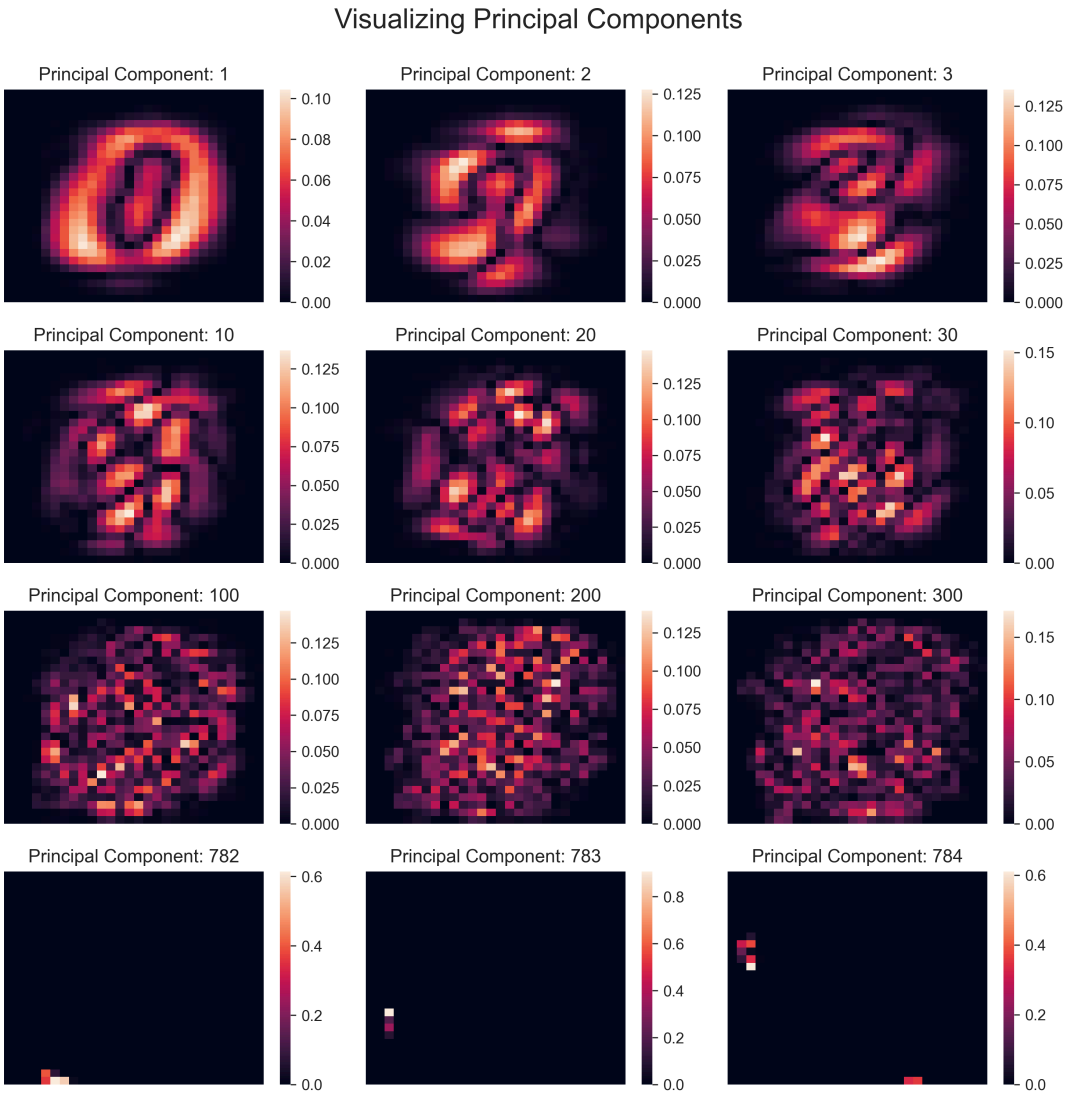


Fig-2: Visualizing the Principal Components

Explaining the visualization of the principal components:

The heatmap shows the spatial relationship between the pixels and the principal component. For example, the regions with high pixel values forming certain recognizable shapes (like zero in the first figure, 9 in the second figure) suggests that the PC is capturing features or patterns associated to those shapes. As expected, the first few PCs capture the most variance in the data, meaning they represent the most significant features that differentiate the digits. They are capturing the most general variance in the data. Later PCs are capturing noise or finer details that are less critical for distinguishing between classes. For example, the last three PC's hardly capture any useful information and are mostly noise.

The variance explained by each principal component is proportional to the eigenvalue associated with each eigenvector of the covariance matrix. The obtained eigenvalues were normalized to get the explained variance, which are shown in fig-3 and fig-4.

Principal_Component	Variance
1	9.6778
2	7.1175
3	6.3284
4	5.5730
5	4.9278

Fig-3: Variance explained by the top 5 Principal Components

Principal_Component	Variance
780	-0.0
781	-0.0
782	-0.0
783	-0.0
784	-0.0

Fig-4: Variance explained by the bottom 5 Principal Components

In-fact, the variance explained by all the principal components from 574 to 784 is zero. That is our dataset completely lies in a subspace of 573 dimensions and thus can be fully explained by a set of 573 orthogonal vectors, which are the first 573 principal components.

On plotting the % variance explained for each principal component, we get the plot as shown in fig-5.

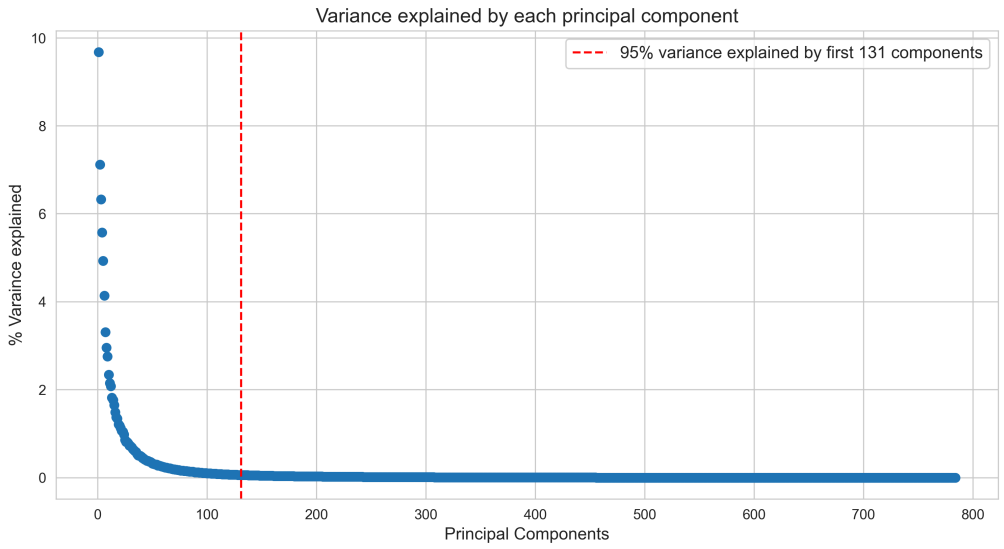
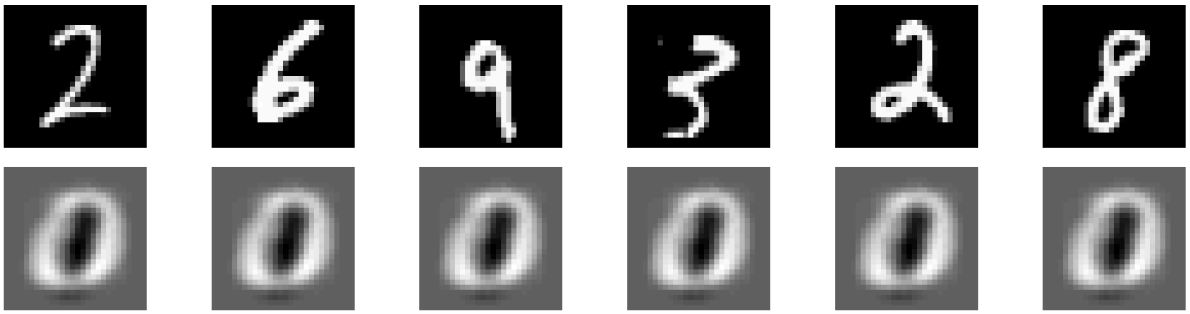


Fig-5: variance explained for each principal component. The red dotted line shows the no. of components required to explain 95% of the total variance.

The 95% variance is explained by the top 131 principal components.

For reconstructing the dataset using different dimensional representations, we first project the dataset on the first k principal components and then reconstructed the projected dataset. This was done using the `.transform()` and `.reconstruct()` methods of the PCA class. The reconstructed datasets looks like the following:

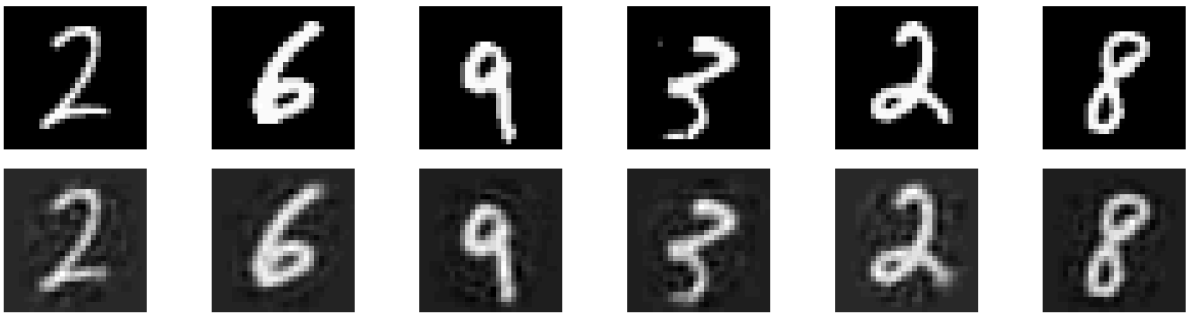
Original Images vs. Reconstructed Images using 1 dimensions



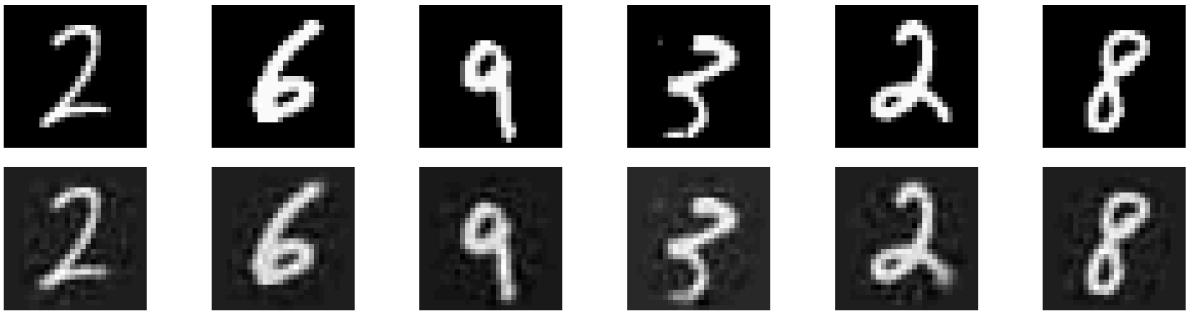
Original Images vs. Reconstructed Images using 25 dimensions

Original Images vs. Reconstructed Images using 50 dimensions

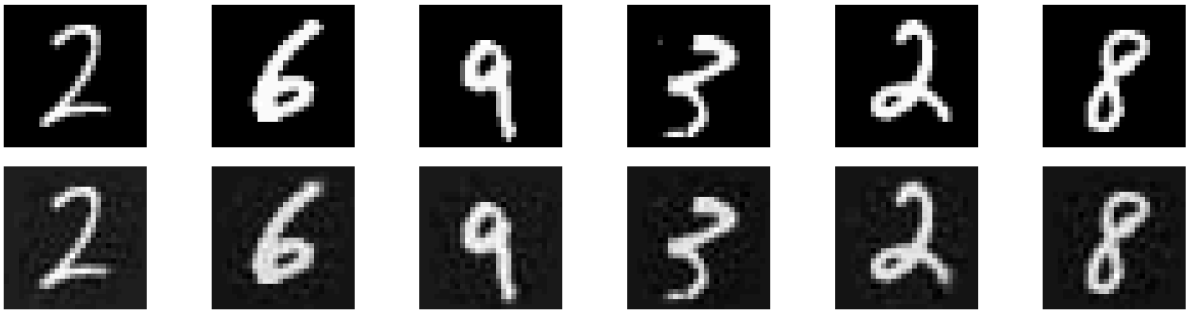
Original Images vs. Reconstructed Images using 100 dimensions



Original Images vs. Reconstructed Images using 131 dimensions



Original Images vs. Reconstructed Images using 200 dimensions



Original Images vs. Reconstructed Images using 300 dimensions

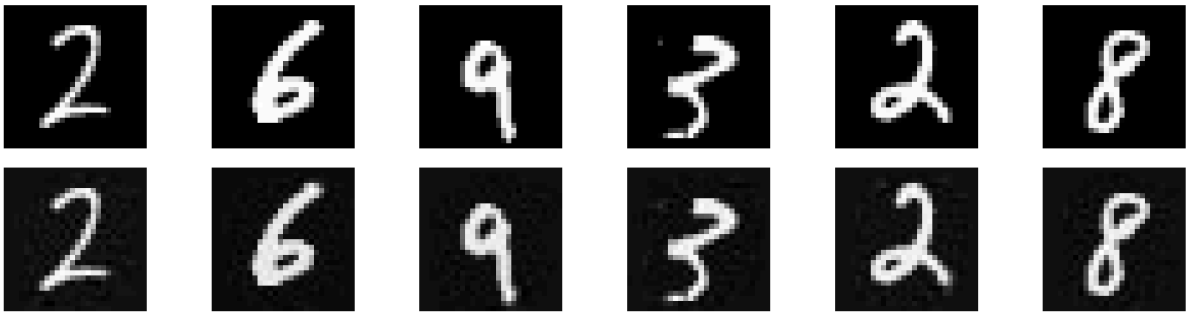


Fig-6: Visualizing reconstructed MNIST digits dataset for various dimensions

- When using only 1 dimension, all the images look very similar and it is impossible to distinguish any digits.
- When using 25 and 50 dimensions, there is some emerging shape resembling the digits, but they still are very fuzzy.
- At 100 dimension, around 93% of the variance is explained by the principal components. We get some definite shape to the digits, but their are other artefacts present which might create difficulties in downstream tasks.
- At 131 dimensions, 95% of the variance is explained by the principal components. The images are now a little bit clearer and the artefacts have reduced. The background of the digits are now slightly darker than at d=100, allowing the digits to become more easily detectable.
- At 200 dimensions, around 98% of the variance is explained by the principal components. The digits are very clearly visible now, but they are still not perfect and very minor inconsistencies are still

present.

- At 300 dimensions, around 99.3% of the variance is explained by the principal components. The images are now almost identical and indistinguishable from the original images.

If we had to pick a dimension d for downstream tasks, d = 131 is the conventional choice as it explains 95% of the variance. At d = 131, the digits are clearly visible, though there are some inconsistencies near the digits. These inconsistencies might create difficulties in downstream tasks. Therefore, due to the slight fuzziness present at d = 131, **d = 200 seems to be the best choice** as it has the best trade-off between number of dimensions and the clarity of the images.

Task-2

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

a. 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.

b. 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).

c. 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?

Lloyd's algorithm

Lloyd's Algorithm (used in K-Means clustering) is an iterative algorithm for partitioning a dataset into k clusters.

1. **Initialization:**

   We randomly select k points from the dataset X as initial cluster centers:

   $$\mu_0 = \{\mu_1, \mu_2, \ldots, \mu_k\}$$

   The cluster assignments vector z , which contains the cluster indicator for each datapoint is calculated based on the following equation:

   $$z_i = \arg\min_j ||x_i - \mu_j||_2$$

   Here, z_i is the cluster indicator of the i^{th} data point.

2. **Update Cluster Centers:**

   After assigning points to clusters, we recompute the cluster centers as the mean of all points assigned to each cluster:

   $\mu_j = \frac{1}{|C_j|} \sum_{x_i \in C_j} x_i$

   where C_j is the set of points assigned to cluster j, and $|C_j|$ is the number of points in that cluster.

When there are no points in a cluster, we re-initialize the cluster center.

3. **Objective Function (Error):**

   The algorithm minimizes the **sum of squared Euclidean distances** (error) between data points and their respective cluster centers:

$$J = \sum_{i=1}^{n} ||x_i - \mu_{z_i}||_2^2$$

   At every iteration, the algorithm seeks to minimize this error by updating z and μ .

4. **Stopping Criterion:**

   The algorithm stops when the change in cluster centers (μ) is below a specified tolerance:

$$||\mu_{\text{prev}} - \mu_{\text{cur}}||_2 \leq \text{tol}$$

To implement K-means on the given dataset, the dataset is first loaded using Pandas library in a Pandas DataFrame. Then, we run the Lloyd's algorithm on the dataset five times, using the `lloyds` function from the `Lloyds.py` file, with k=2.

The error function wrt iterations for each run is calculated by the `error()` function present in the `Lloyds.py` file. The error function value for each run against the iterations is given in fig-7.

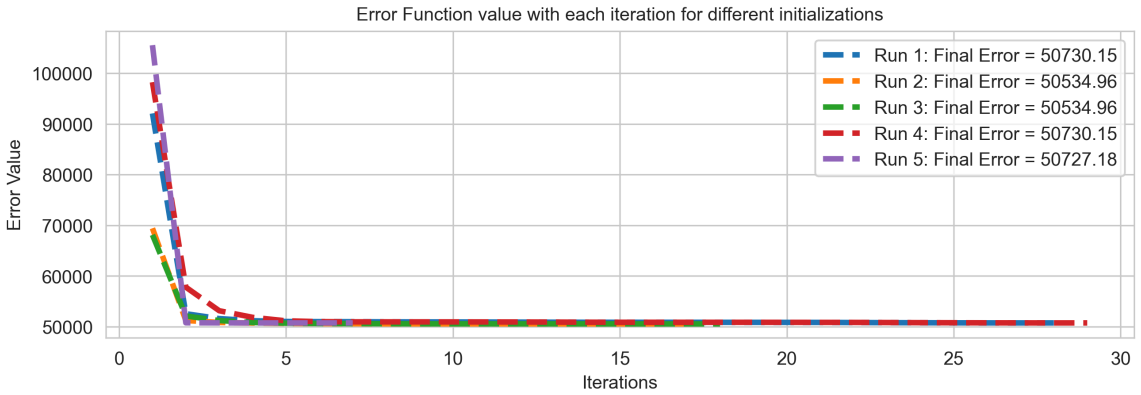


Fig-7: Error function value wrt iterations for 5 runs of Lloyd's algorithm with k=2 on the dataset

The clusters obtained at each run are plotted using the `plot_cluster` helper function present in the `Helpers.py` file. Cluster for each run is given in fig-8.

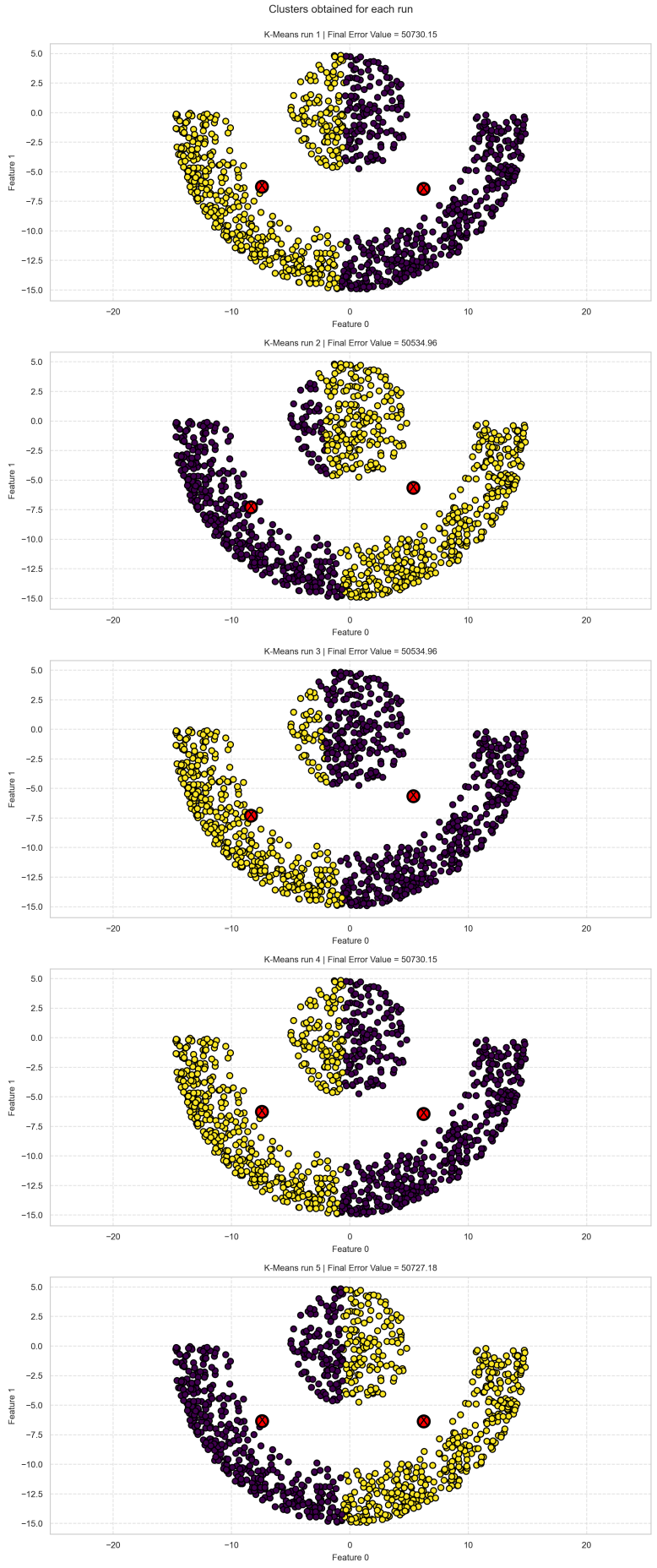

Clusters obtained for each run
K-Means run 1 | Final Error Value = 50730.15
K-Means run 2 | Final Error Value = 50534.96
K-Means run 3 | Final Error Value = 50534.96
K-Means run 4 | Final Error Value = 50730.15
K-Means run 5 | Final Error Value = 50727.18
Feature 0
Feature 1

Fig-8: Clusters obtained from 5 different runs of Lloyd's algorithm on the dataset. The cluster centers are represented by the red circles with X in them.

Now, we will run the Lloyd's algorithm on the dataset for K = {2,3,4,5} by fixing arbitrary initialization and then we will visualize the Vornoi regions. The Vornoi regions are plotted using the `plot_vornoi` helper function present in the `Helpers.py` file. The clusters obtained by running the Lloyd's algorithm are given in fig-9.

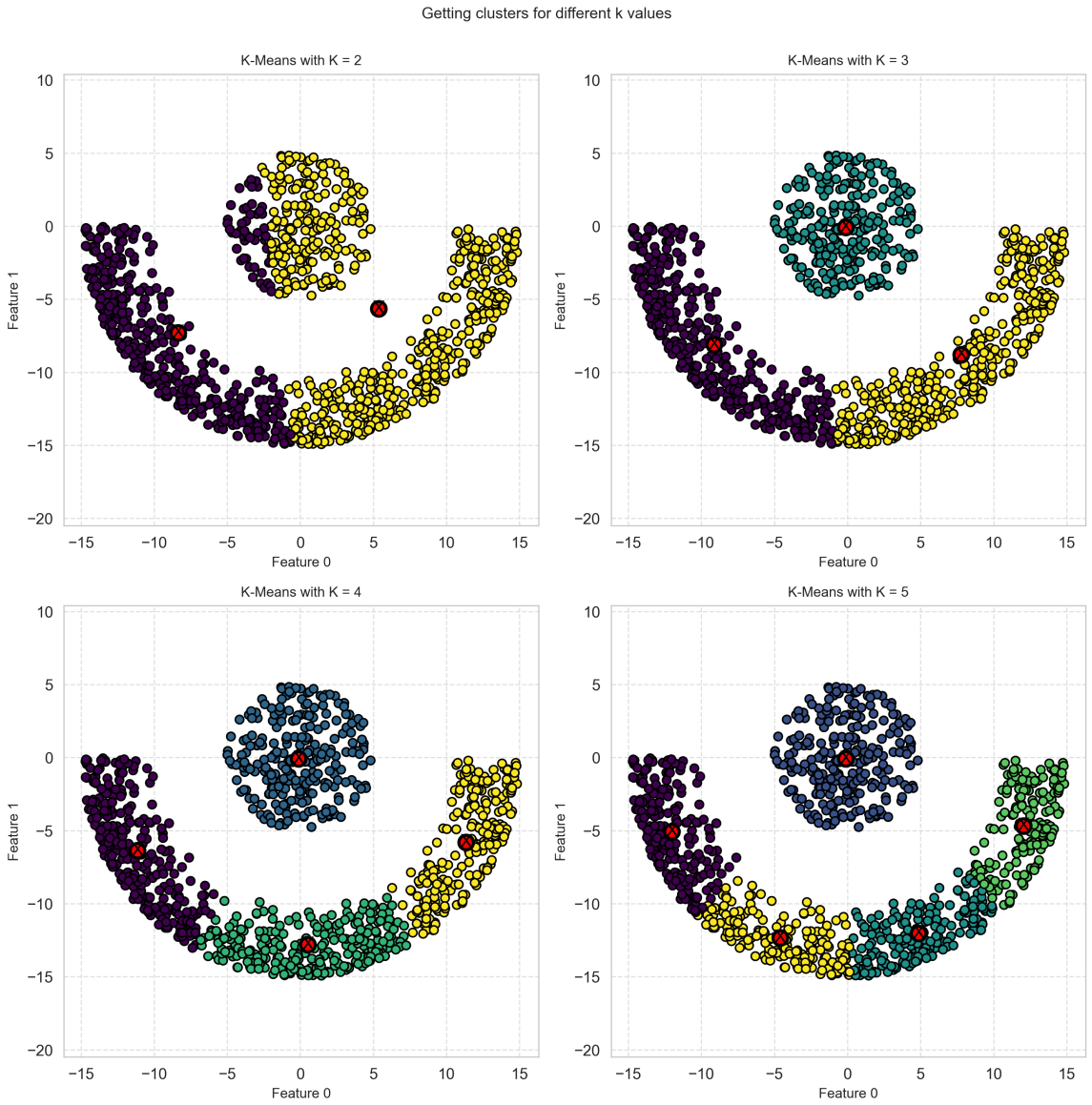


Fig-9: Running Lloyd's algorithm on the dataset for K = {2,3,4,5} and plotting the obtained clusters

The Vornoi regions associated with each of the above clusters are given in fig-10.

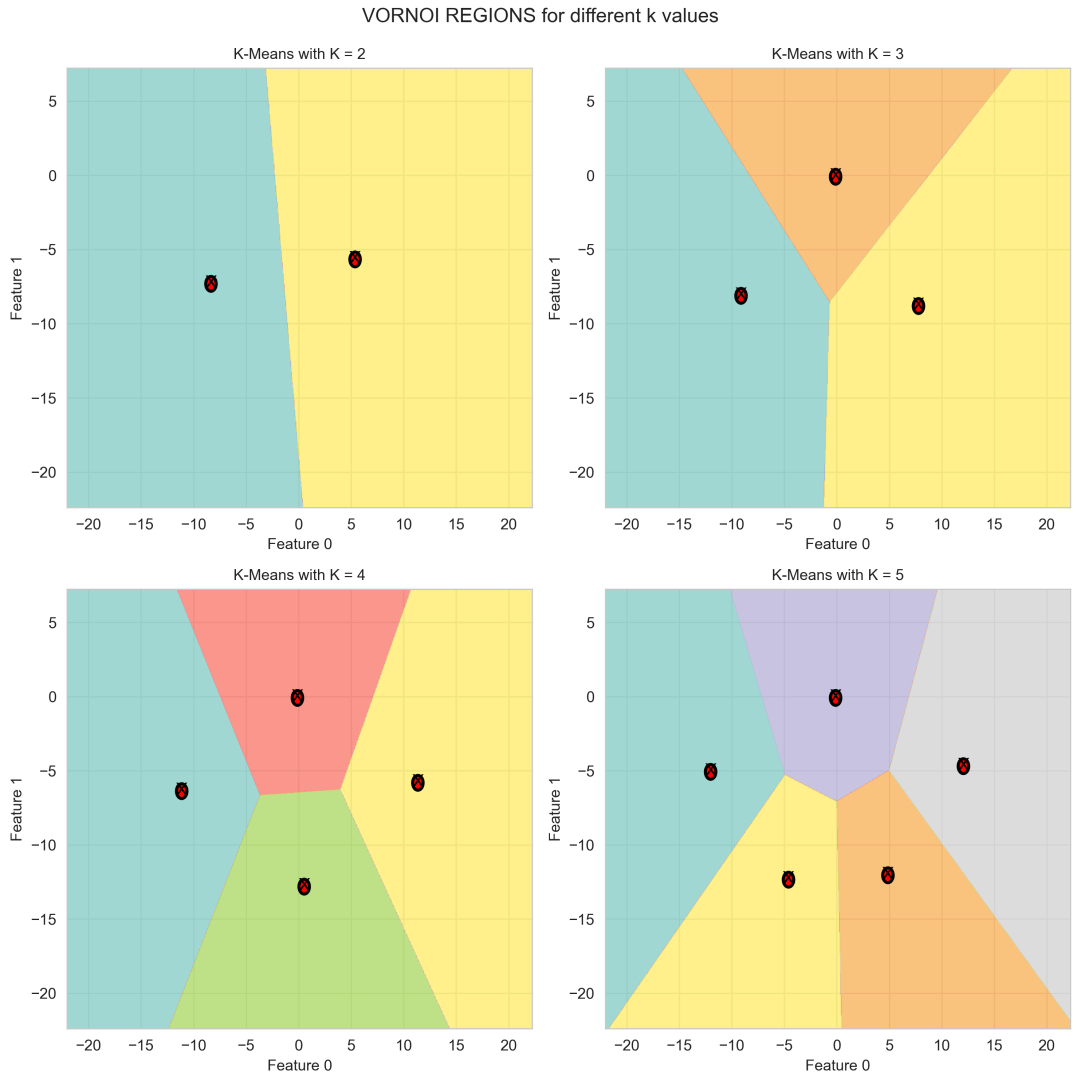


Fig-10: Vornoi regions associated with different values of k for the given dataset.

The Lloyd's algorithm is not able to find the right clusters for this dataset. The right clustering result should have 2 clusters with the centre circular cluster as one cluster and the semi-circular arc as another cluster. The Lloyd's algorithm is unable to identify these two clusters because the dataset is not convex. The dataset has non-linear relationship among the points belonging to each cluster, and Lloyd's algorithm only works on linearly separable datasets as the Vornoi regions are linearly defined regions. Hence Lloyd's algorithm is not the correct way to cluster this dataset.

Kernelized K-Means is more suitable to cluster this dataset, as the clusters are separable by linear boundaries in higher dimensions. If the datapoints are mapped to polynomial dimensions of degree 2, then these datapoints will become linearly separable and then K-Means will be able to separate these clusters by learning the correct Vornoi Regions.

Source Codes

▼ **PCA**

```
import numpy as np
```

```
class PCA():
    """
    Implement PCA with n_comp principal components on X (n x d) dataset with n samples and d fe
atures
    """
    def __init__(self, n_comp = None):
        self.n_comp = n_comp
        self.fit_done = False

    def _center(self, X):
        """
        Centers the dataset X
        """
        return X - np.mean(X, axis = 0)

    def fit(self, X):
        """
        Fits the PCA on passed X dataset. Returns eigenvalues and eigenvectors.
        All eigenvalues and eigenvectors are returned if n_comp = None (default),
        else top `n_comp` eigenvalues and eigenvectors are returned
        """
        self.X = X
        if self.n_comp is None:
            self.n_comp = self.X.shape[1]   # setting number of principal components as all component
s if not passed
        self.X = self._center(self.X) # centering X
        self.n = self.X.shape[0]    # no. of samples
        cov = 1/self.n * self.X.T @ self.X  # calculating the covariance matrix
        e, v = np.linalg.eigh(cov)   # returns eigenvalues and normalized eigenvectors of the covarian
ce matrix in increasing order
        self.fit_done = True
        self.eigenvalues = e[::-1][:self.n_comp]
        self.eigenvectors = v[:, ::-1][:, :self.n_comp]
        return self.eigenvalues, self.eigenvectors

    def transform(self, X, n_comp):
        """
        Projects the passed X (n x d) dataset with n samples and d features on the first n_comp PCA
components
        Returns the projected dataset
        """
        if not self.fit_done:
            raise ValueError("fit PCA first before transforming")
        return X @ self.eigenvectors[:, :n_comp]

    def reconstruct(self, X):
        """
        Reconstruct dataset transformed using PCA to original dimensions
        """
        n_comp = X.shape[1]
        return X @ self.eigenvectors[:, :n_comp].T
```

▼ **Lloyds**

```
import numpy as np
from copy import deepcopy

# implement Lloyds

def update_z(X, mu):
    """
    Finds the cluster centers nearest to the points of X out of the given cluster centers mu
    returns the assignment vector z
    the position of cluster center in mu is it's cluster indicator,
    i.e. cluster indicator go from 0 to len(mu) - 1
    """
    z = np.ones((len(X)))*-1   ## Initializing z to -1
    for i in range(len(X)):
        z[i] = np.argmin(np.linalg.norm(X[i, :] - mu, ord = 2, axis = 1))
    return z

def mu_calculate(X, z, k):
    """
    Calculates the mu vector for a given assignment vector z and set of Datapoints X
    k: number of clusters
    """
    mu = np.zeros((k, X.shape[1]))
    for k_cur in range(k):
        X_c = X[z == k_cur]
        if len(X_c) > 0:
            mu[k_cur, :] = np.mean(X_c, axis = 0)
        else:
            mu[k_cur, :] = np.random.uniform(X.min(axis=0), X.max(axis=0), X.shape[1])  # Reinitialize
empty cluster in case the cluster has no points
    return mu

def error(X, z, mu):
    """
    Function to calculate error value, which is the sum of squared differences between
    each point and their cluster means
    """
    return np.sum(np.linalg.norm(X - mu[z.astype("int"), :], axis=1)**2)

def lloyds(X, k = 2, init = None, tol = 10**-6):
    """
    Implement Lloyd's algorithm
    X: Data points in n x d shape
    No. of clusters: k
    returns: cluster assignment vector z, cluster centers (k x d) matrix, and errors vector
            containing error for each iteration
    Algorithm will go on until the L-2 difference between consecutive cluster centers
    is more than the tolerance value.
    """
```

```
    if init is None:
        mu_0_ind = np.random.choice(X.shape[0], k, replace=False)  # Random initialization
        mu_0 = X[mu_0_ind]
    else:
        mu_0 = init

    mu_prev = mu_0
    z = update_z(X, mu_0)
    mu_cur = np.ones(mu_0.shape)*-1
    error_list = [error(X, z, mu_prev)]

    while(np.linalg.norm(mu_prev - mu_cur, ord = 2) > tol):
        mu_prev = deepcopy(mu_cur)     # Deep copying mu_cur to mu_prev before updating mu_cur
        mu_cur = mu_calculate(X, z, k)
        z = update_z(X, mu_cur)
        error_list.append(error(X, z, mu_cur))

    return z, mu_cur, np.array(error_list)
```

▼ **main_file**

```
import pandas as pd
import os
from PCA import PCA
import matplotlib.pyplot as plt
import io
import numpy as np
import seaborn as sns
from Lloyds import lloyds
import io
from Helpers import save_figure, plot_clusters, plot_vornoi, balanced_subsample

# setting the plots style
sns.set_style("whitegrid")

# Loading the MNIST dataset into pandas dataframe
df_train = pd.read_parquet(os.path.join(os.getcwd(), "mnist_parquet", "train-00000-of-00001.par
quet"))
df_test = pd.read_parquet(os.path.join(os.getcwd(), "mnist_parquet", "test-00000-of-00001.parqu
et"))



# Since the images are stored in bytes format, converting the bytes format into matrix format
## Also flattening the 28*28 image matrix to 784 sized vector

df_train["image_matrix"] = df_train["image"].apply(lambda x: plt.imread(io.BytesIO(x["bytes"])).ra
vel())
df_test["image_matrix"] = df_test["image"].apply(lambda x: plt.imread(io.BytesIO(x["bytes"])).rave
l())

# Using the function balanced_subsample to sample 1000 images from the dataset with 100 sampl
```

```
es for each digit
df_train = balanced_subsample(df_train.loc[:, ["label", "image_matrix"]], "label", 1000)


## Visualizing some train images
inds = [2,3,4,7,9,13]

fig, ax = plt.subplots(nrows = 1, ncols=6, figsize = (10,5))

for axis, ind in zip(ax.ravel(), inds):
    axis.imshow(df_train.loc[ind, "image_matrix"].reshape(28,28), cmap = "grey")
    axis.set_title(df_train.loc[ind, "label"], size = 12)
    axis.axis("off")
plt.suptitle("Visualizing digit sample images", y = 0.7, size = 14)
plt.show()
save_img_path = os.path.join(os.getcwd(), "output_images")
save_figure(fig, "Visualizing digit sample images")

# Running PCA on the MNIST dataset
X = np.array(list(df_train["image_matrix"]))
y = np.array(list(df_train["label"]))
pca = PCA()
eigenvalues, pca_comp = pca.fit(X)

#Visualizing the principal components
pca_list = [0, 1, 2, 9, 19, 29, 99, 199, 299, 781, 782, 783]
fig, ax = plt.subplots(nrows=4, ncols=3, figsize=(10, 10))

for axis, p in zip(ax.ravel(), pca_list):
    sns.heatmap(np.abs(pca_comp[:, p].reshape(28, 28)), ax=axis)
    axis.set_title(f"Principal Component: {p+1}", size = 12)
    axis.axis("off")

plt.suptitle("Visualizing Principal Components", size = 18, y = 1)
plt.tight_layout()
plt.show()
save_figure(fig, "Visualizing Principal Components")


## Calculating the fraction of variance explained by each principal component
f_variance = eigenvalues/np.sum(eigenvalues, axis = 0)

## Finding the 95% variance explained component
f_cumsum = np.cumsum(f_variance)
x = np.argwhere(f_cumsum>0.95).ravel()[0]  ## The component upto which 95% variance is expl
ained

## Tabulating the variance explained by components
f_variance_p = np.round(f_variance*100, 4)
variance_exp = list(zip(np.arange(1, len(f_variance_p)+1), f_variance_p))
variance_exp = [{"Principal_Component": i, "Variance": j} for i,j in variance_exp]
```

```
df_variance = pd.DataFrame(columns = ["Principal_Component", "Variance"], data = variance_ex
p)
df_variance = df_variance.set_index("Principal_Component")
print("% Variance explained by top 5 Principal Components")
print(df_variance.head())
print("% Variance explained by bottom 5 Principal Components")
print(df_variance.tail())

# Plotting the explained variance as a scatter plot
fig = plt.figure(figsize = (12, 6))
plt.scatter(np.arange(1, len(f_variance)+1), f_variance*100)
plt.ylabel("% Varaince explained", size = 12)
plt.xlabel("Principal Components", size = 12)
plt.title("Variance explained by each principal component", size = 14)
plt.axvline(x, label = f"95% variance explained by first {x} components", color = "red", linestyle =
"--")
plt.legend(fontsize = 12)
plt.show()
save_figure(fig, "Variance explained by each principal component")


# Reconstructing the dataset using different dimensional representations
inds = [0,2,4,5,7,13] # samples to plot
X_ = X[:20, :]  # taking a subsample of X to prevent un-necessary transformations of samples

d = [1, 25, 50, 100, 131, 200, 300]  # dimensions used to reconstruct

for dim in d:
    fig, ax = plt.subplots(nrows = 2, ncols = len(inds), figsize = (10, 3))
    X_transformed = pca.transform(X_, dim)
    X_reconstructed = pca.reconstruct(X_transformed)
    for i, x_zip in enumerate(zip(ax.ravel(), inds*2)):
        axis, ind = x_zip
        if i < len(inds):
            axis.imshow(X[ind, :].reshape(28, 28), cmap="grey")
        else:
            axis.imshow(X_reconstructed[ind, :].reshape(28, 28), cmap="grey")
        axis.axis("off")

    plt.suptitle(f"Original Images vs. Reconstructed Images using {dim} dimensions ", size = 12, y =
1)
    plt.tight_layout()
    plt.show()
    save_figure(fig, f"Original Images vs. Reconstructed Images using {dim} dimensions ")


## Loading the dataset for K-means
df = pd.read_csv("cm_dataset_2.csv", header = None)
X = np.array(df.loc[: , :])

## Running k-means with random initialization 5 times
```

```
runs = 5
z_list = []
mu_list = []
errors_list = []

for i in range(runs):
    z, mu, errors = lloyds(X, k = 2)
    z_list.append(z)
    mu_list.append(mu)
    errors_list.append(errors)

# Plotting the error function vs iteration curves
fig, ax = plt.subplots(nrows = 1, ncols = 1, figsize = (10, 3))
for i in range(runs):
    ax.plot(np.arange(1, len(errors_list[i])+1), errors_list[i],
            label = f"Run {i+1}: Final Error = {np.round(errors_list[i][-1], 2)}",
           linestyle = "--", linewidth = 3)
    ax.set_title("Error Function value with each iteration for different initializations", size = 10)
    ax.legend()
    ax.set_xlabel("Iterations", size = 10)
    ax.set_ylabel("Error Value", size = 10)
    ax.tick_params(axis='both', which='major', labelsize=10)
plt.show()
save_figure(fig, "Error Function value with each iteration for different initializations")


# Visualizing the clusters
fig, ax = plt.subplots(nrows = 5, ncols = 1, figsize = (10, 5*runs), gridspec_kw={'hspace': 0.2})
ax = ax.ravel()[:runs]

for i, axis in zip(range(runs), ax):
    plot_clusters(X, z_list[i], mu_list[i], ax = axis)
    axis.set_title(f"K-Means run {i+1} | Final Error Value = {np.round(errors_list[i][-1], 2)}", size = 8)
    axis.set_xlabel("Feature 0", size = 8)
    axis.set_ylabel("Feature 1", size = 8)
    axis.tick_params(axis='both', which='major', labelsize=8)
plt.suptitle("Clusters obtained for each run", y = 0.9, size = 10)
# plt.tight_layout()
plt.show()
save_figure(fig, "Clusters obtained for each run")


# Running K-means for different K-values
# Taking the labels at 0, 100, 150, 200, and 400 to be the initialization points
ids = [0, 100, 150, 200, 400]
init_means = X[ids, :]
fig, ax = plt.subplots(nrows = 2, ncols = 2, figsize = (10,10))
for i, (axis, K) in enumerate(zip(ax.ravel(), [2,3,4,5])):
    z, mu, errors = lloyds(X, k = K, init = init_means[:K, :])
    plot_vornoi(X, mu, ax = axis)
    axis.set_title(f"K-Means with K = {K}", size = 10)
```

```
    axis.set_xlabel("Feature 0", size = 10)
    axis.set_ylabel("Feature 1", size = 10)
plt.suptitle("VORNOI REGIONS for different k values", y = 0.99, size = 13)
plt.tight_layout()
plt.show()
save_figure(fig, "VORNOI REGIONS for different k values")

# Plotting the clusters for the above initialization points and k values
ids = [0, 100, 150, 200, 400]
init_means = X[ids, :]
fig, ax = plt.subplots(nrows = 2, ncols = 2, figsize = (10,10))
for i, (axis, K) in enumerate(zip(ax.ravel(), [2,3,4,5])):
    z, mu, errors = lloyds(X, k = K, init = init_means[:K, :])
    plot_clusters(X, z, mu, ax = axis)
    axis.set_title(f"K-Means with K = {K}", size = 9)
    axis.set_xlabel("Feature 0", size = 9)
    axis.set_ylabel("Feature 1", size = 9)
plt.suptitle("Getting clusters for different k values", y = 1, size = 10)
plt.tight_layout()
plt.show()
save_figure(fig, "Getting clusters for different k values")
```

▼ **Helpers (helper functions)**

```
import os
import numpy as np
import matplotlib.pyplot as plt
from Lloyds import update_z
import pandas as pd


def balanced_subsample(df, label_column, total_samples):
    """
    Sub-sample x datapoints from a DataFrame such that each unique label
    in the label column has the same count in the sampled dataset.

    Parameters:
    df (pd.DataFrame): The input DataFrame.
    label_column (str): The name of the column containing labels.
    total_samples (int): Total number of samples desired in the subsample.

    Returns:
    pd.DataFrame: A DataFrame containing the balanced subsample.
    """
    unique_labels = df[label_column].unique()
    samples_per_label = 100

    subsampled_frames = []

    for label in unique_labels:
        label_subset = df[df[label_column] == label]
```

```
        subsampled_frames.append(label_subset.sample(samples_per_label, random_state=42))

    balanced_df = pd.concat(subsampled_frames).sample(frac=1, random_state=42).reset_index(dr
op=True)
    return balanced_df


save_img_path = os.path.join(os.getcwd(), "output_images")

## creating a function to save image
def save_figure(fig, filename, directory = save_img_path, dpi=300):
    """
    Save a Matplotlib figure to a specified directory with a given filename as a PNG file.

    Parameters:
    - fig: Matplotlib figure object to save.
    - directory: Path to the directory where the figure should be saved.
    - filename: Name of the file (without extension).
    - dpi: Resolution of the saved figure (default is 300).
    """
    if not os.path.exists(directory):
        os.makedirs(directory)  # Create the directory if it doesn't exist

    filepath = os.path.join(directory, f"{filename}.png")
    fig.savefig(filepath, dpi=dpi, format='png', bbox_inches='tight')


# creating a helper function to plot clusters
def plot_clusters(X, z, mu, ax=None):
    """
    Plot clusters on a given axis or create a new figure.

    Parameters:
    - X: ndarray of shape (n_samples, n_features), the data points
    - z: array-like, cluster assignments for each data point
    - mu: ndarray of shape (n_clusters, n_features), cluster centroids
    - ax: Matplotlib axis object, optional. If None, a new figure is created.

    Returns:
    - ax: The axis containing the plot
    """
    if ax is None:
        fig, ax = plt.subplots(figsize=(8, 6))

    scatter = ax.scatter(X[:, 0], X[:, 1], c=z, cmap='viridis', s=30, edgecolor='k')

    # Add circles for cluster means
    for cluster_id, mean in enumerate(mu):
        circle = plt.Circle(mean, radius=0.5, edgecolor='black', facecolor='red', lw=2)
        ax.add_artist(circle)  # Add the circle to the axis
        ax.text(mean[0], mean[1], "X", color='black', fontsize=12, ha='center', va='center')
```

```
    # Add labels, title, and other settings
    ax.set_title("Fig Title")
    ax.set_xlabel("X-axis")
    ax.set_ylabel("Y-axis")
    ax.axis("equal")  # Keep axis scales equal
    ax.grid(True, linestyle='--', alpha=0.5)

    return ax

# creating a helper function to plot vornoi regions
def plot_vornoi(X, mu, ax = None):
    """
    Plot vornoi regions for given mu and X
    """
    xlims = (np.min(X[:, 0])*1.5, np.max(X[:, 0])*1.5)  ## X-axis range
    ylims = (np.min(X[:, 1])*1.5, np.max(X[:, 1])*1.5)  ## Y-axis range

    lin_res = 1000   # variable for setting the linear resolution
    x_cords = np.linspace(xlims[0], xlims[1], num = lin_res)
    y_cords = np.linspace(ylims[0], ylims[1], num = lin_res)
    xx, yy = np.meshgrid(x_cords, y_cords)
    xy = np.c_[xx.ravel(), yy.ravel()]
    z = update_z(xy, mu).reshape(lin_res, lin_res)

    if ax is None:
        fig, ax = plt.subplots(figsize = (10, 8))
    ax.contourf(xx, yy, z, cmap='Set3', alpha = 0.8)
    for cluster_id, mean in enumerate(mu):
        circle = plt.Circle(mean, radius=0.5, edgecolor='black', facecolor='red', lw=2)
        ax.add_artist(circle)
        ax.text(mean[0], mean[1], "X", color='black', fontsize=12, ha='center', va='center')
        ax.set_xlim(xlims)
        ax.set_ylim(ylims)
    return ax
```

```
import numpy as np

def classification_metrics(y_true, y_preds):
    """
    y_true: True values
    y_preds: predicted values
    """
    y_true = np.array(y_true).ravel()
    y_preds = np.array(y_preds).ravel()

    true_pos = np.sum(np.where((y_preds == 1) & (y_true == 1), 1, 0))
    false_pos = np.sum(np.where((y_preds == 1) & (y_true == 0), 1, 0))
    true_neg = np.sum(np.where((y_preds == 0) & (y_true == 0), 1, 0))
    false_neg = np.sum(np.where((y_preds == 0) & (y_true == 1), 1, 0))
```

```
recall = true_pos/(true_pos + false_neg)
precision = true_pos/(true_pos + false_pos)
f1_score = 2*precision*recall/(precision + recall)

print(f"recall: {recall}")
print(f"precision: {precision}")
print(f"f1_score: {f1_score}")
```