DA5400 Assignment-3

Preethi - MM21B051

Task 1

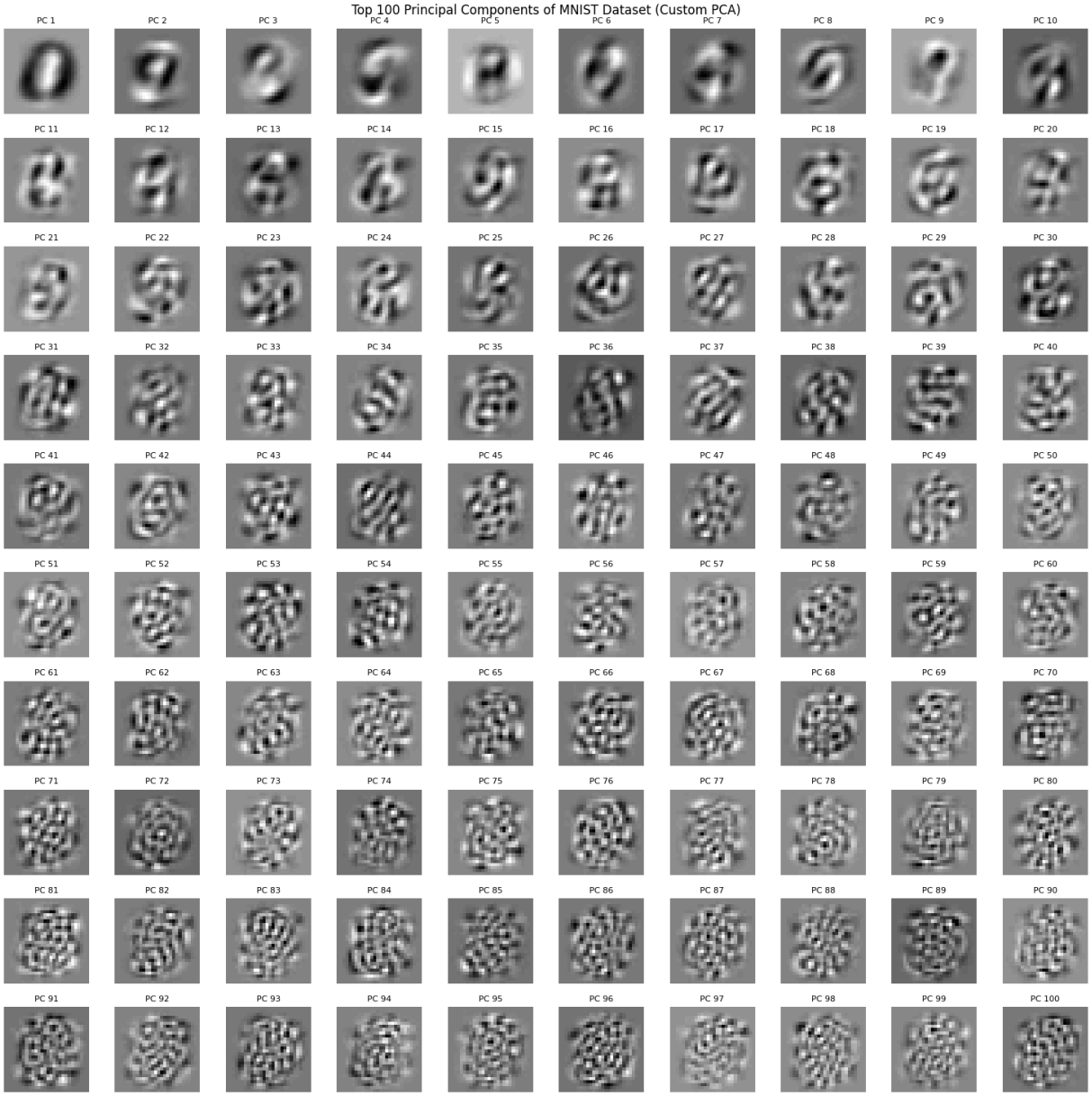
We load the dataset, and collect 100 images of each digit, for the PCA:

1. mean centre the data set X
2. compute the cov matrix C, of the data set X
3. compute the eigen values and eigen vectors of C
4. sort the eigen vectors in decending order of eigen values
5. find projection of X on each vector and reshape to image size

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from datasets import load_dataset

# load the MNIST dataset
mnist = load_dataset("mnist", split="train")
sampled_data = {"image": [], "label": []}
class_counts = {i: 0 for i in range(10)}
target_samples_per_class = 100
mnist = mnist.shuffle(seed=42)

# we collect 100 images from each class
for sample in mnist:
    label = sample["label"]
    if class_counts[label] < target_samples_per_class:
        sampled_data["image"].append(np.array(sample["image"]).flatten())
        sampled_data["label"].append(label)
        class_counts[label] += 1
    if sum(class_counts.values()) >= 1000:
        break

X = np.array(sampled_data["image"])
y = np.array(sampled_data["label"])
n_samples, n_features = X.shape

# PCA function
def custom_pca(X, n_components):
    X_centered = X - np.mean(X, axis=0)     # mean center the data
    covariance_matrix = np.cov(X_centered, rowvar=False) # covariance matrix
    eigenvalues, eigenvectors = np.linalg.eigh(covariance_matrix) # compute eigenva

    sorted_indices = np.argsort(eigenvalues)[::-1]
    sorted_eigenvectors = eigenvectors[:, sorted_indices]
```

```
    # top n eigenvectors
    selected_eigenvectors = sorted_eigenvectors[:, :n_components]
    X_reduced = np.dot(X_centered, selected_eigenvectors) # Project the data onto t
    return X_reduced, selected_eigenvectors, np.mean(X, axis=0)

# function to reconstruct the images
def reconstruct_images(X_reduced, eigenvectors, mean_image):
    return np.dot(X_reduced, eigenvectors.T) + mean_image

X_reduced, selected_eigenvectors, mean_image = custom_pca(X, n_components=100)

# Visualize the top 100 principal components
fig, axes = plt.subplots(10, 10, figsize=(15, 15))
axes = axes.ravel()

for i in range(100):
    component_image = selected_eigenvectors[:, i].reshape(28, 28)  # reshape to ori
    axes[i].imshow(component_image, cmap='gray')
    axes[i].set_title(f"PC {i+1}", fontsize=8)
    axes[i].axis('off')

plt.suptitle("Top 100 Principal Components of MNIST Dataset (Custom PCA)")
plt.tight_layout()
plt.show()
```

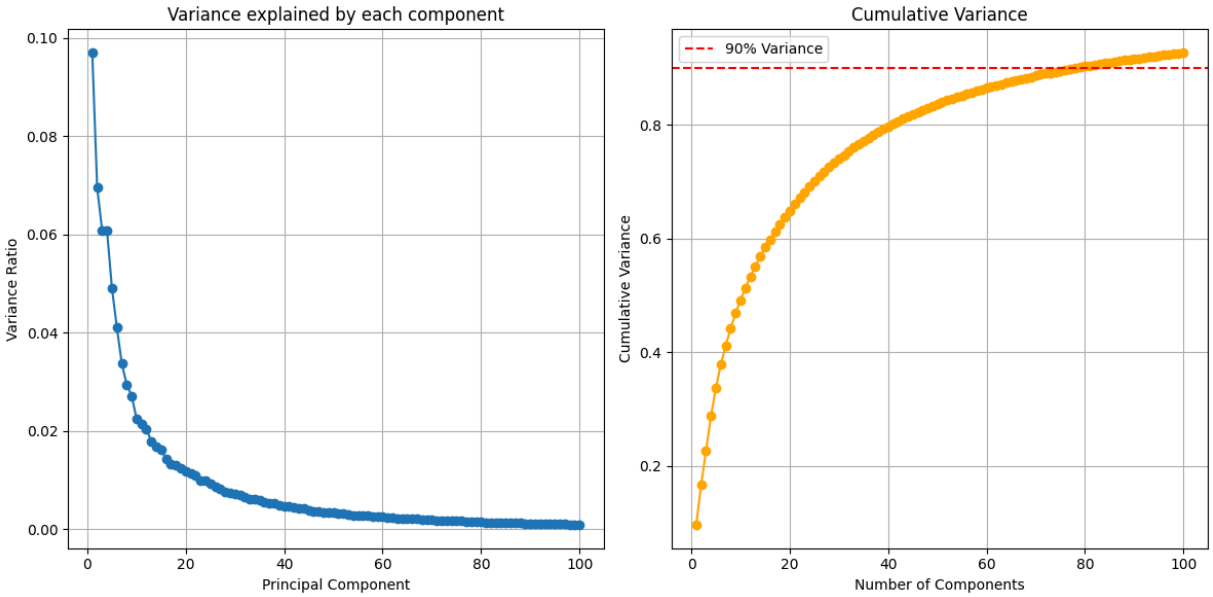
```
c:\Users\preet\AppData\Local\Programs\Python\Python312\Lib\site-packages\tqdm\auto.p
y:21: TqdmWarning: IProgress not found. Please update jupyter and ipywidgets. See ht
tps://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```



The variance ratio is given by, eigen value/ sum of eigen values, cummulative ratio is the sum of top n eigen values/ sum of all eigen values.

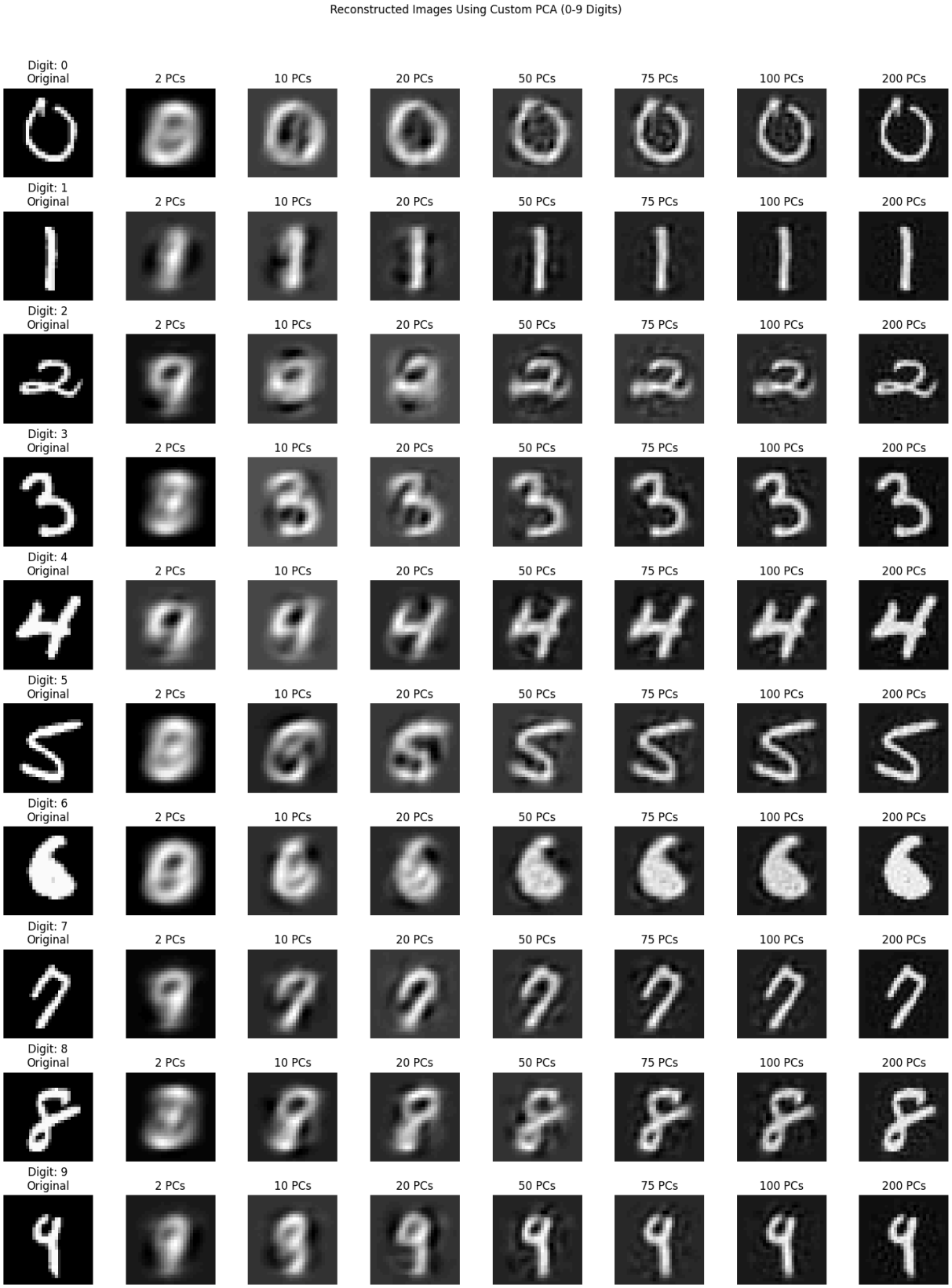
In [2]:
```
# Compute variance ratio for the first 100 components
eigenvalues = np.linalg.eigh(np.cov(X - np.mean(X, axis=0), rowvar=False))[0]  # Us
sorted_eigenvalues = np.sort(eigenvalues)[::-1]  # Sort in descending order
explained_variance = sorted_eigenvalues[:100]  # Select the first 100 eigenvalues
explained_variance_ratio = explained_variance / np.sum(sorted_eigenvalues)

plt.figure(figsize=(12, 6))

# Scree plot: Variance explained by each of the first 100 components
plt.subplot(1, 2, 1)
plt.plot(range(1, 101), explained_variance_ratio, marker='o')
plt.title("Variance explained by each component")
plt.xlabel("Principal Component")
plt.ylabel("Variance Ratio")
plt.grid()
```

```
# Cumulative explained variance plot
cumulative_variance = np.cumsum(explained_variance_ratio)
plt.subplot(1, 2, 2)
plt.plot(range(1, 101), cumulative_variance, marker='o', color='orange')
plt.title("Cumulative Variance")
plt.xlabel("Number of Components")
plt.ylabel("Cumulative Variance")
plt.grid()
plt.axhline(y=0.90, color='r', linestyle='--', label="90% Variance")
plt.legend()

plt.tight_layout()
plt.show()
```



In [3]:
```
def reconstruct_images(X_reduced, eigenvectors, mean, n_components):
    return np.dot(X_reduced, eigenvectors.T) + mean

# number of components for reconstruction
n_components_list = [2, 10, 20, 50, 75, 100, 200]

fig, axes = plt.subplots(10, len(n_components_list) + 1, figsize=(15, 20))
fig.suptitle("Reconstructed Images Using Custom PCA (0-9 Digits)")

for digit in range(10):
    # chosing a random image of the current digit
    indices = np.where(y == digit)[0]
    random_index = np.random.choice(indices)
    original_image = X[random_index].reshape(28, 28)

    # plot the original image
    axes[digit, 0].imshow(original_image, cmap='gray')
    axes[digit, 0].set_title(f"Digit: {digit}\nOriginal")
    axes[digit, 0].axis('off')

    # apply PCA and reconstruct images with different numbers of components
    for j, n_components in enumerate(n_components_list):
```

```
        X_reduced, eigenvectors, mean = custom_pca(X, n_components)
        reconstructed_image = reconstruct_images(X_reduced[random_index], eigenvect

        axes[digit, j + 1].imshow(reconstructed_image, cmap='gray')
        axes[digit, j + 1].set_title(f"{n_components} PCs")
        axes[digit, j + 1].axis('off')

plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()
```



We can see that for each of these digits, we are able to recognize it well at 50PCs, however 75 is more clear with lesser variance and can be used for downstream tasks.

Task 2

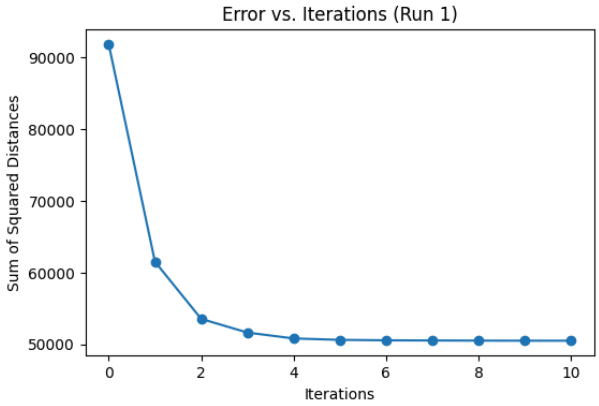
for the Lloyd's algorithm:

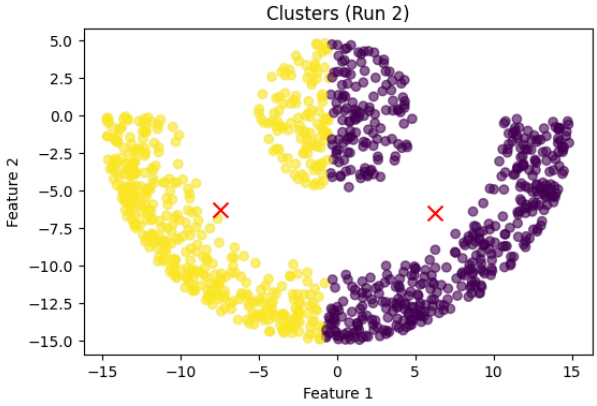
1. randomly initialize k centroids
2. assign each point to its nearest centroid
3. update the centroid of each cluster
4. repeat until convergence

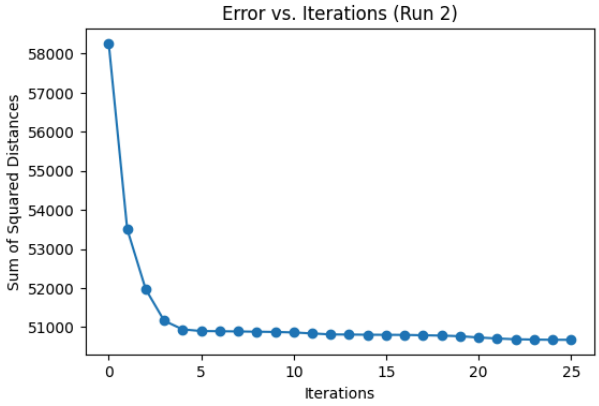
In [ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Load the dataset
data = pd.read_csv('cm_dataset_2.csv')
X = data.values

# Sum squared error
def compute_error(X, centroids, labels):
    error = 0
    for i in range(len(centroids)):
        cluster_points = X[labels == i]
        error += np.sum((cluster_points - centroids[i]) ** 2)
    return error

# Lloyd's algorithm
def lloyds_kmeans(X, k, max_iter=100):
    n_samples, n_features = X.shape
    errors = []
    initial_centroids=X[np.random.choice(n_samples, k, replace=False)]
    k = len(initial_centroids)
    # random initialization of centroids
    centroids = initial_centroids.copy()
    errors = []

    for iteration in range(max_iter):
        # Assign each point to the nearest centroid
        distances = np.linalg.norm(X[:, np.newaxis] - centroids, axis=2)
        labels = np.argmin(distances, axis=1)
        new_centroids = np.array([X[labels == i].mean(axis=0) for i in range(k)])
        error = compute_error(X, new_centroids, labels)  # compute SSE
        errors.append(error)
        if np.all(centroids == new_centroids):   # Check for convergence
            break
        centroids = new_centroids
    return centroids, labels, errors

def plot_results(X, labels, centroids, errors, run_idx, ax1, ax2):
    # Cluster plot
    ax1.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', marker='o', alpha=0.6)
    ax1.scatter(centroids[:, 0], centroids[:, 1], c='red', marker='x', s=100)
    ax1.set_title(f'Clusters (Run {run_idx})')
    ax1.set_xlabel('Feature 1')
    ax1.set_ylabel('Feature 2')

    # Error function plot
```

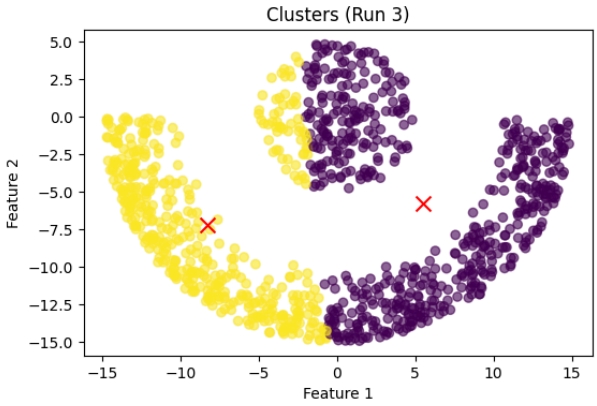
```
    ax2.plot(errors, marker='o')
    ax2.set_title(f'Error vs. Iterations (Run {run_idx})')
    ax2.set_xlabel('Iterations')
    ax2.set_ylabel('Sum of Squared Distances')

fig, axes = plt.subplots(nrows=5, ncols=2, figsize=(12, 20))
fig.tight_layout(pad=4.0)

for i in range(5):
    centroids, labels, errors = lloyds_kmeans(X, k=2)
    plot_results(X, labels, centroids, errors, run_idx=i + 1, ax1=axes[i, 0], ax2=a

plt.show()
```

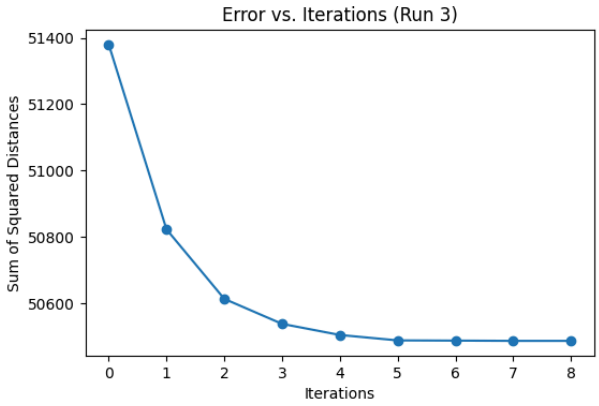

Clusters (Run 1)
Feature 1
Feature 2


Error vs. Iterations (Run 1)
Iterations
Sum of Squared Distances

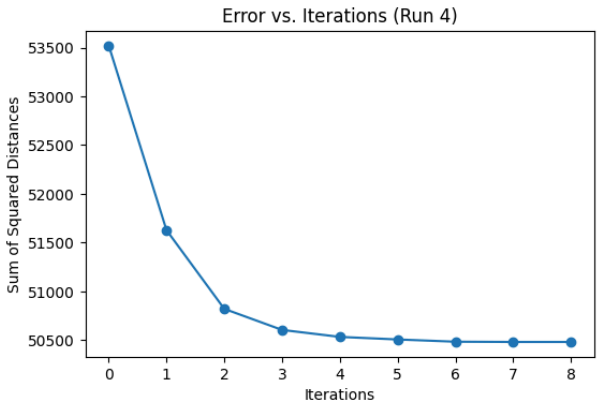

Clusters (Run 2)
Feature 1
Feature 2

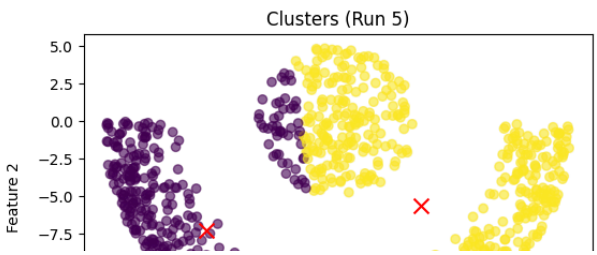

Error vs. Iterations (Run 2)
Iterations
Sum of Squared Distances

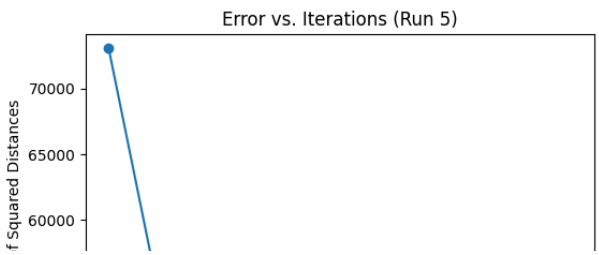

Clusters (Run 3)
Feature 1
Feature 2

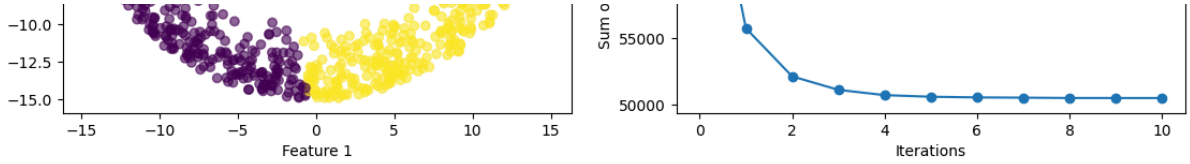

Error vs. Iterations (Run 3)
Iterations
Sum of Squared Distances


Clusters (Run 4)
Feature 1
Feature 2


Error vs. Iterations (Run 4)
Iterations
Sum of Squared Distances

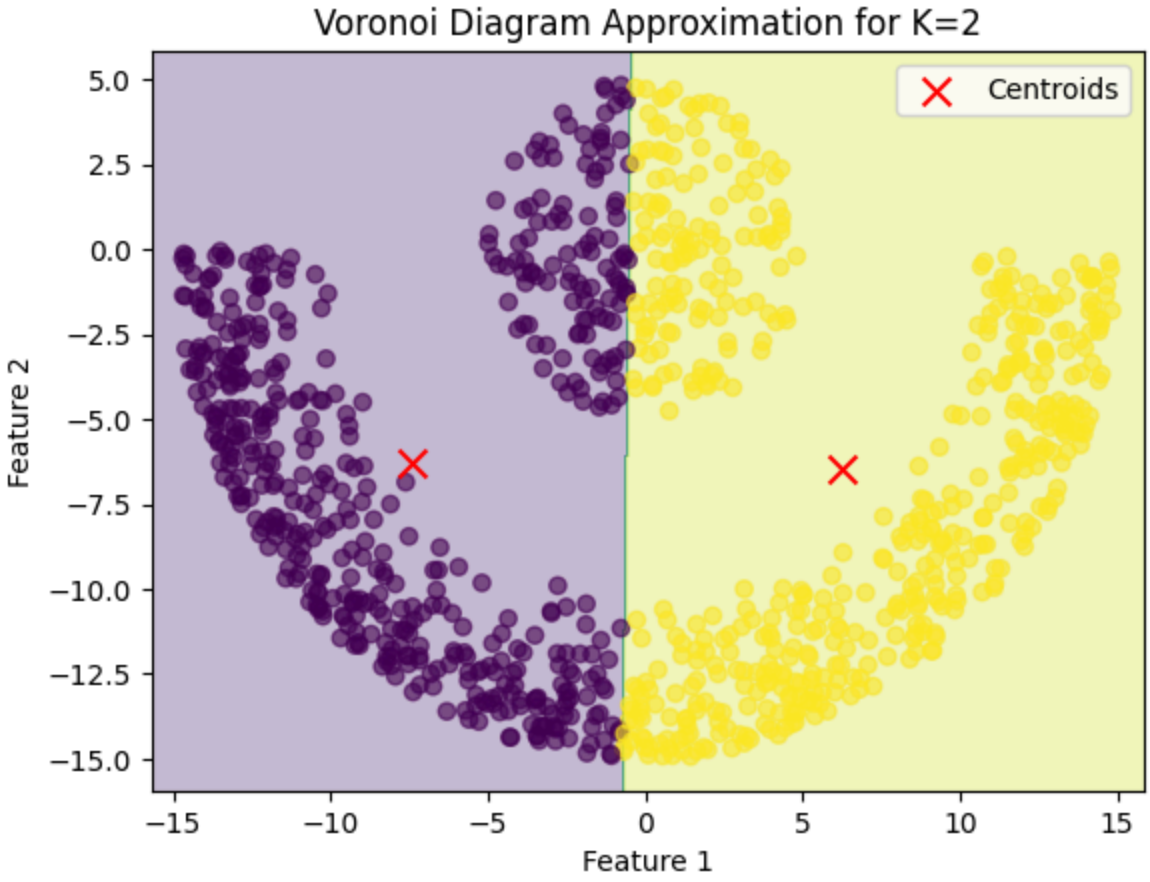

Clusters (Run 5)
Feature 2


Error vs. Iterations (Run 5)

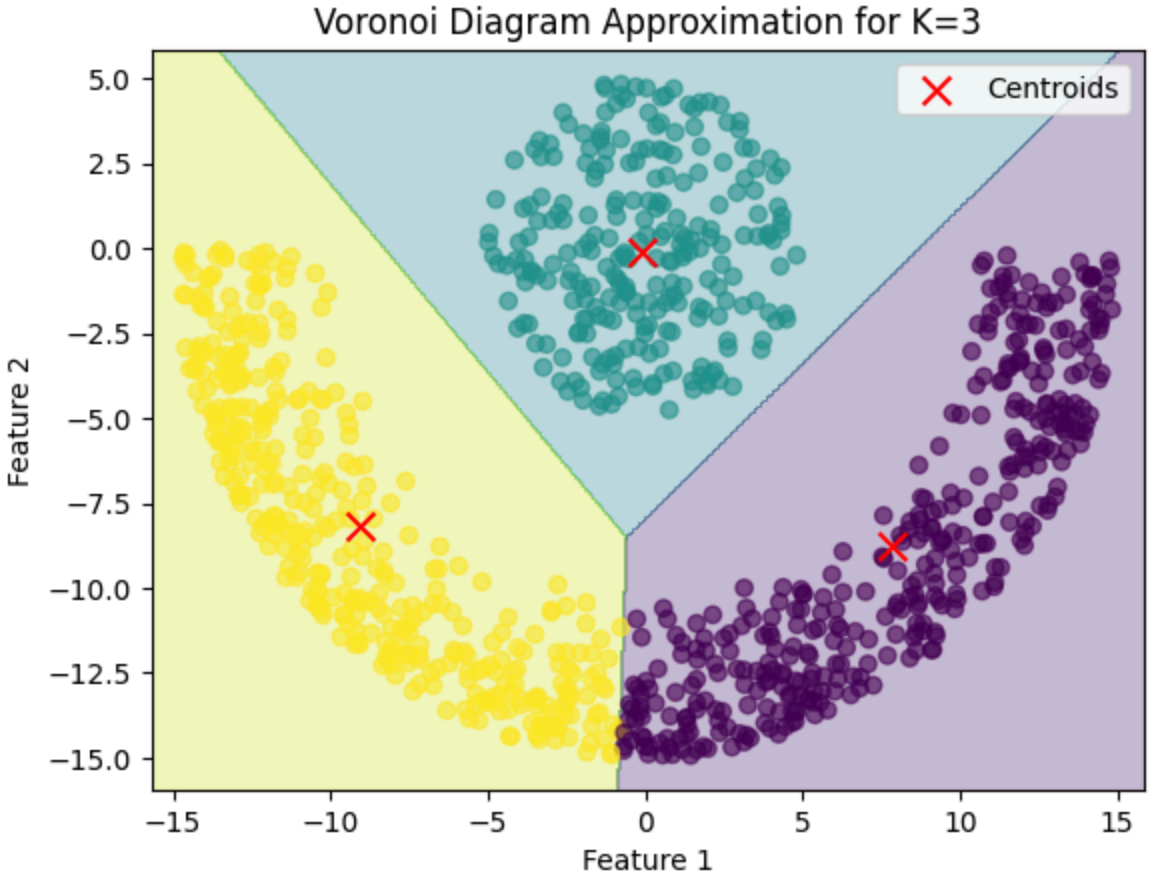


In [ ]:
```python
# Function to plot Voronoi regions
def plot_voronoi_approximation(X, centroids, labels, k):
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 500), np.linspace(y_min, y_max,
    grid_points = np.c_[xx.ravel(), yy.ravel()]

    # assigning each point in the mesh grid to the nearest centroid
    distances = np.linalg.norm(grid_points[:, np.newaxis] - centroids, axis=2)
    grid_labels = np.argmin(distances, axis=1)
    grid_labels = grid_labels.reshape(xx.shape)

    plt.contourf(xx, yy, grid_labels, cmap='viridis', alpha=0.3)
    plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', marker='o', alpha=0.6)
    plt.scatter(centroids[:, 0], centroids[:, 1], c='red', marker='x', s=100, label
    plt.title(f'Voronoi Diagram Approximation for K={k}')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.legend()
    plt.show()

K_values = [2, 3, 4, 5]
n_samples, n_features = X.shape

# Iterate over each K value, run K-means, and plot Voronoi regions
for k in K_values:
    print(f'Running K-means for K={k}...')
    # initial_centroids = initializations[k]
    centroids, labels, errors = lloyds_kmeans(X, k)
    plot_voronoi_approximation(X, centroids, labels, k)
```

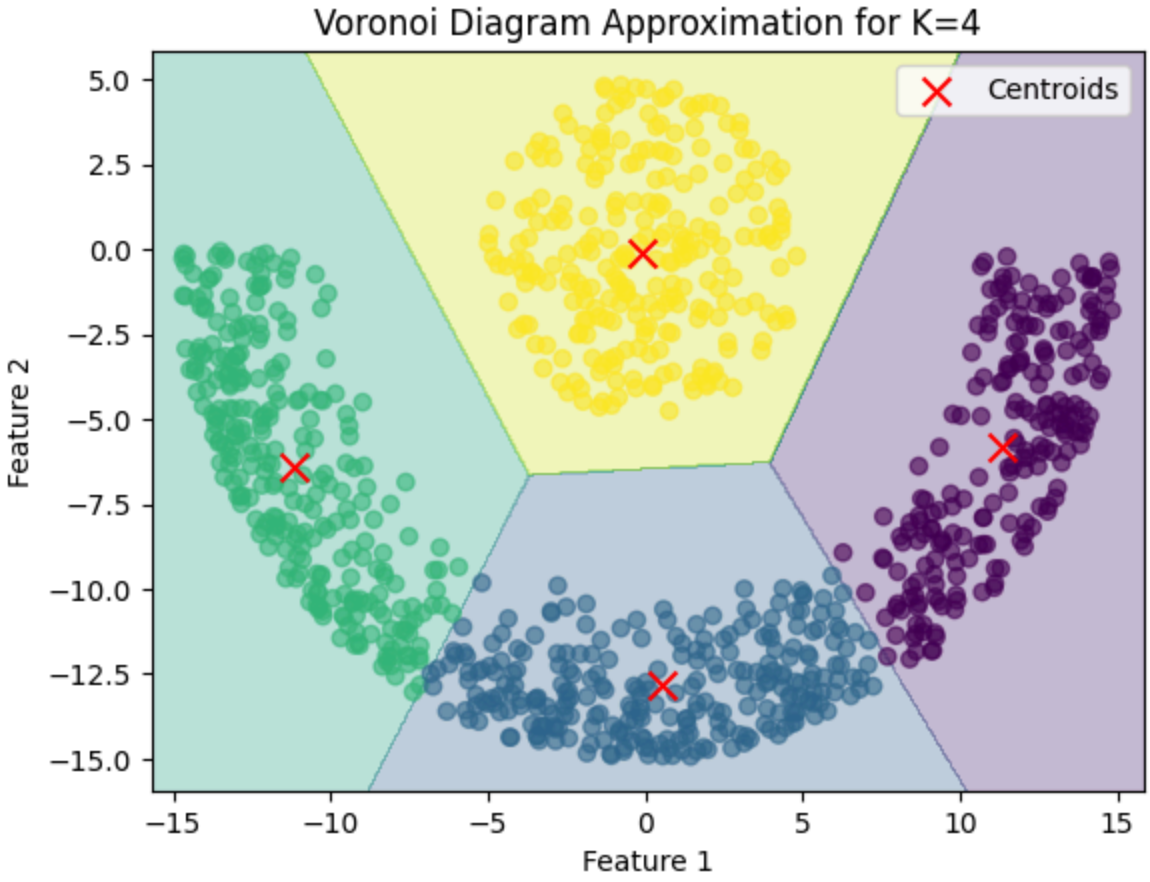
```
Running K-means for K=2...
```



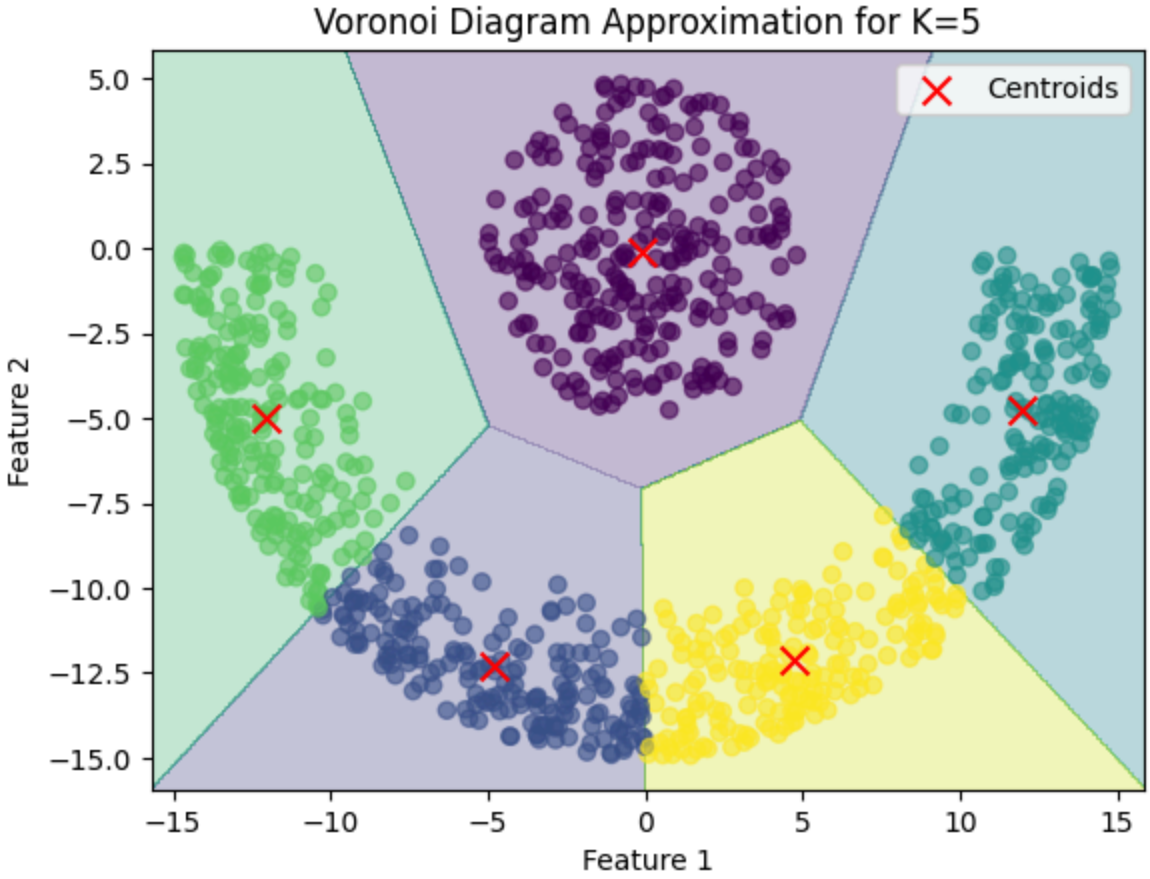

```
Running K-means for K=3...
```




```
Running K-means for K=4...
```




```
Running K-means for K=5...
```



We see that none of the k values can separate the data efficiently with a low loss as the data is not linearly separable. We need to kernalize the data into a higher dimension and then separate it using a hyper plane.