DA5400 - Foundation of Machine Learning

Assignment 3

Gokulakrishnan B

DA24M007

(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.

First, we download the MNIST dataset from hugging face datasets name “ylecun/mnist" , then for each label (0 to 9), we store 100 images in list. Then convert every image to numpy arrays, and store the final result in array called imagesArray. We will use this imagesArray as input to our PCA algorithm.

Write a piece of code to run the PCA algorithm on this data-set.

The steps to compute principal components are as follows:

1. Find the mean of data points.
2. Center the dataset by subtracting the mean from every data point in the dataset.
3. Compute the covariance matrix of the centred dataset.
4. Find the Eigen values and Eigen vectors of the covariance matrix.
5. Sort the Eigen values and the corresponding Eigen vectors in decreasing order.
6. Now these top Eigen vectors are the principal components.
7. If we want to choose top k principal components to reduce dimensionality, choose top k Eigen vectors.

8. To transform a given input, centre the input by subtracting with mean and compute its dot product with the transpose of top k Eigen vectors. This gives the dimensionally reduced representation of input.

9. To reconstruct the image, again compute dot product of top k principal components and reduced data. We may not the get the exact original representation as this is not a lossless reconstruction.

The above steps are implemented as follows:

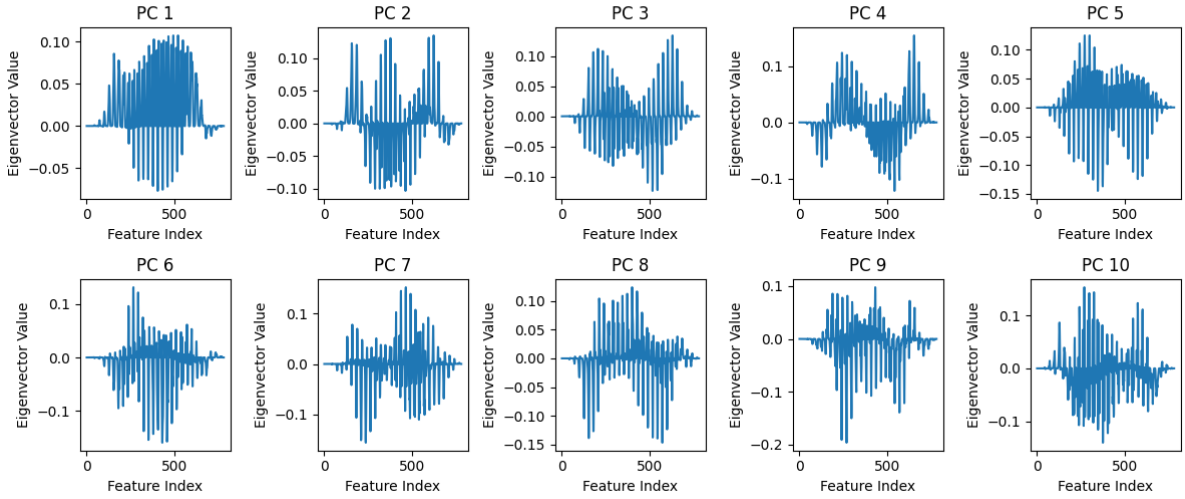
```
class PCA:
    def __init__(self):
        self.mean = None
        self.eigenVectors = None
        self.eigenValues = None

    def fit(self, X):
        # Center the dataset
        self.mean = X.mean(axis=0)
        X = X - self.mean

        # Compute Covariance matrix
        cov = np.cov(X.T)

        # find eigen values and vectors
        values,vectors = np.linalg.eigh(cov)
        vectors = vectors.T

        # sort them
        idxs = np.argsort(values)[::-1]
        values = values[idxs]
        vectors = vectors[idxs]
```

```
        self.eigenValues = values
        self.eigenVectors = vectors


    def transform(self,X,n_components):

        # center the dataset
        X = X - self.mean

        # compute projected X

        projectedX = X @ self.eigenVectors[:n_components].T

        # return projected X
        return projectedX


    def variance_ratio(self,pc):
        var = self.eigenValues[pc] / sum(self.eigenValues) * 100
        return var.round(2)

    def inverse_transform(self,X):
        # get the reduced dimension of data
        dimension = X.shape[0]

        # Compute eigenvector.T @ transformed data
        reconstructed = self.eigenVectors[:dimension].T @ X

        # Add mean to it
        reconstructed += self.mean
```

```
    # Return reconstructed image

    return reconstructed
```

Visualize the images of the principal components that you obtain

1.a) First 10 Principal Components



Here, the X axis indicates the features(totally 784 features from 28 x 28 image). The y axis represents the eigenvector values. Higher the absolute value, more the variance it captures about the data. The plots indicate that first few principal components (especially PC1) captures a significant amount of variance in the data.

How much of the variance in the data-set is explained by each of the principal components?

```
Explained variance ratio
Principal component 1 - 9.69 %
Principal component 2 - 7.44 %
Principal component 3 - 6.92 %
Principal component 4 - 5.44 %
Principal component 5 - 4.88 %
Principal component 6 - 4.57 %
Principal component 7 - 3.49 %
Principal component 8 - 3.03 %
Principal component 9 - 2.8 %
Principal component 10 - 2.15 %
```

Here, we can see that PC1 captures 9.69 percentage variance, followed by other top 10 PC's. The difference between the explained ratio is not large.

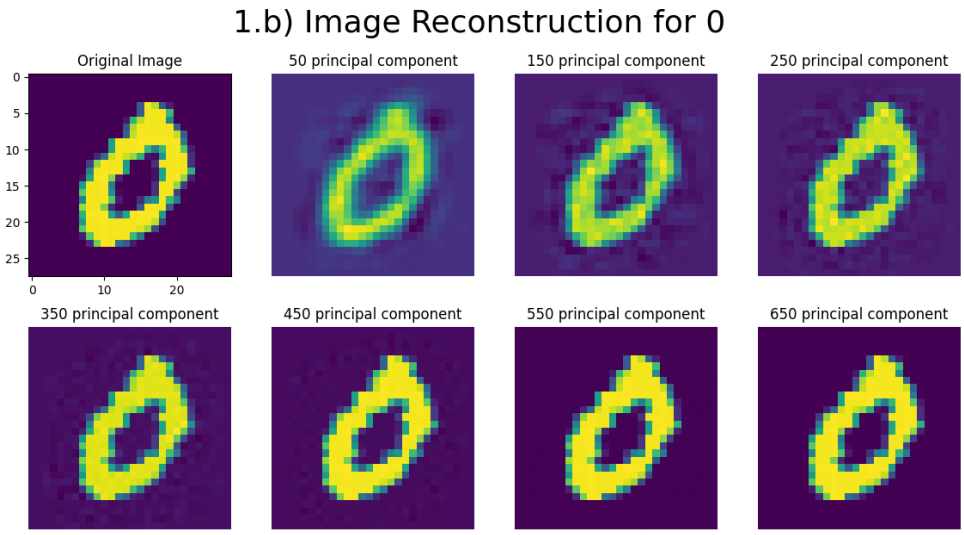
Percentage of explained variance is calculated by

$$\text{Explained Variance \%} = \frac{\lambda_i}{\sum_{i=1}^{n} \lambda_i} \times 100$$

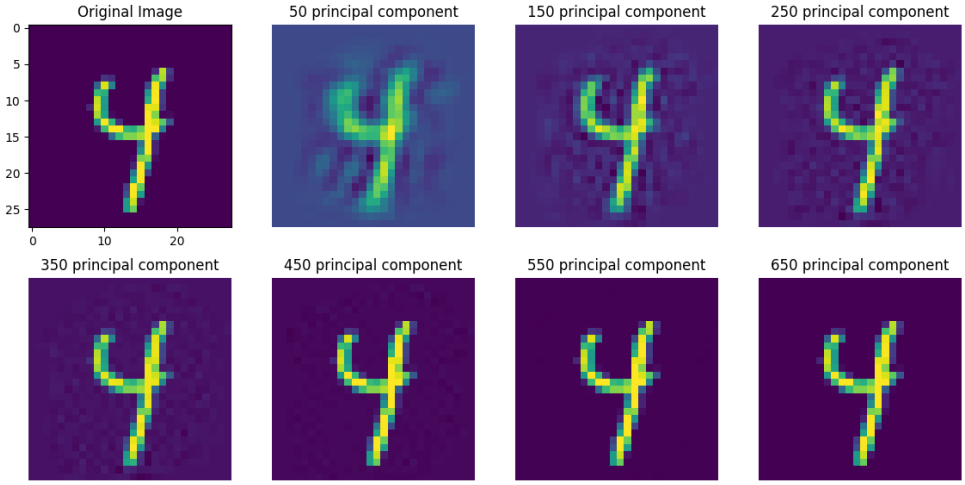
where λ_i is the i-th eigenvalue and n is the total number of eigenvalues.

Reconstruct the dataset using different dimensional representations. How do these look like?

Reconstructed image with label 0 for different principal components looks like



1.b) Image Reconstruction for 4



Reconstructed image with label 4 for different principal components looks like

In the above images, we can see that, as we increase the number of principal components, the image become less blurry and more clear. This is because, lesser the principal component, lesser the dimension, and less information is retained. We are able to recognise the label with 150 principal components, but with 550 principal components, we could barely notice a difference with the original image.

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?

In both the images, with 50 principal components, we are able to recognise the number displayed (with lower confidence) although the number looks blurred. With 150 principal components, we are able to recognise the image with higher confidence. Thus I would go with the value of d as 150, where we reduce the dimension of data, still able to identify the label of the image comfortably.

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

Write a piece of code to implement the Llyod"s algorithm for the K-means problem with k = 2 .

Llyod's Algorithm takes a bunch of points as input and number of clusters k, and output label of cluster for each data point.

The Llyod's algorithm is implemented with below algorithm:

1. Let k be the number of clusters that we want to segment the datapoints into.
2. Given the datapoints, we randomly pick k data points as cluster centroids.
3. Now for every datapoint, compute its distance with all the k cluster centroids.
4. Assign each datapoint to a cluster to which it has least distance (I have used euclidian distance here).
5. Now, recompute the cluster centroids by computing mean of all points in that particular centroid.
6. Now, we get the new cluster centroids, again calculate the distance of every datapoint to new cluster centroids and choose the cluster with least distance.
7. Keep on repeating this step until convergence. Here convergence mean, the data points remain in the old cluster and new cluster centroids is same as the old cluster centroids. Doesn't matter how much time we iterate the algo again after convergence, the points won't change its cluster.
8. Now return the cluster labels for all the datapoints.

The code for above algorithm is given below

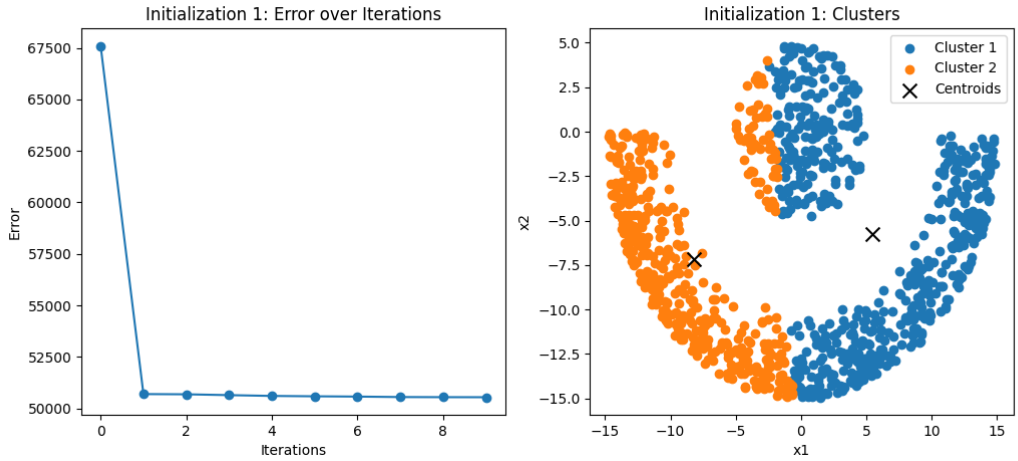
```
class KmeansClustering:

    def __init__(self,k=2,max_iter = 10, seed = 42):

        # number of clusters is represented by k

        self.k = k

        # max_iter -  maximum number of iterations to stop, if the algo
did not converge

        self.max_iter = max_iter

        # initilaising centroids

        self.centroids = [None] * k
```

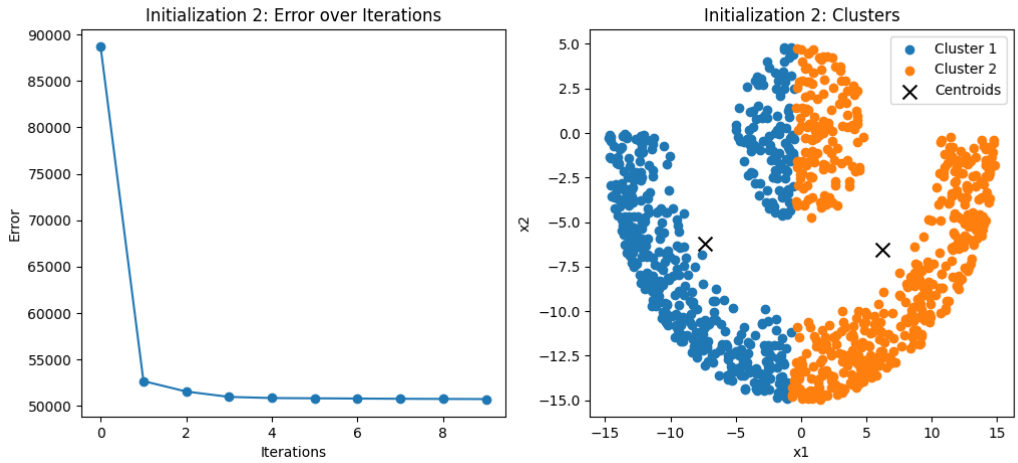
```
        # setting random seed to determine initlisation
        np.random.seed(seed)
        # errors, to store the error history across iterations
        self.errors=[]

    def fit(self,X):
        # choosing random points as centroids
        random_idx = np.random.choice(len(X), self.k,replace=False)
        self.centroids = X[random_idx]

        for _ in range(self.max_iter):
            # compute distance for every point to every cluster
            distances = np.linalg.norm(X[:,np.newaxis] - self.centroids
, axis = 2)
            # choose the centroid with minimum distance
            labels = np.argmin(distances,axis=1)

            # update centroids by calculating the mean of cluster
            new_centroids = []
            for i in range(self.k):
                new_centroids.append(X[labels == i].mean(axis=0))

            new_centroids = np.array(new_centroids)

            # compute error as the squared distance between data points
and its cluster centroid
            error = np.sum((X - self.centroids[labels]) ** 2)
            self.errors.append(error)

            # If the centroids are not updating (reached convergence),
stop the algorithm
            if np.allclose(self.centroids, new_centroids):
```

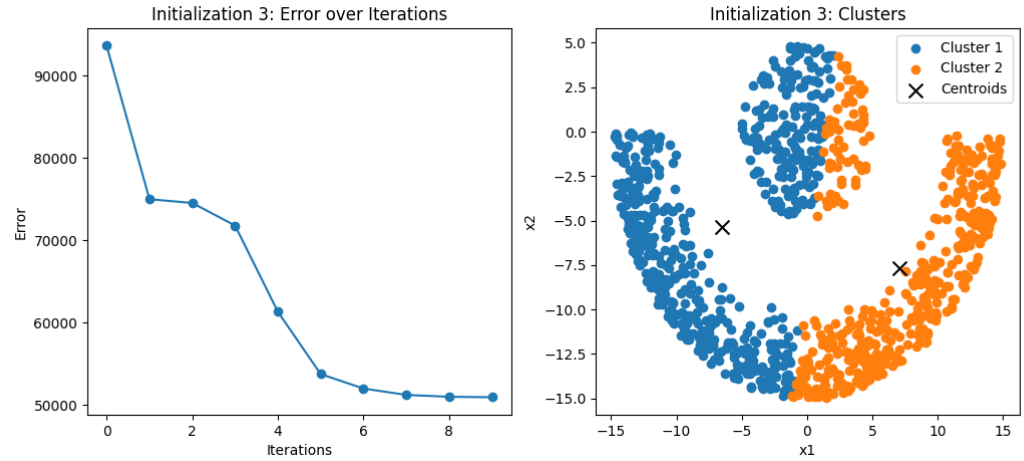
```
            break
        self.centroids = new_centroids
    return labels
```

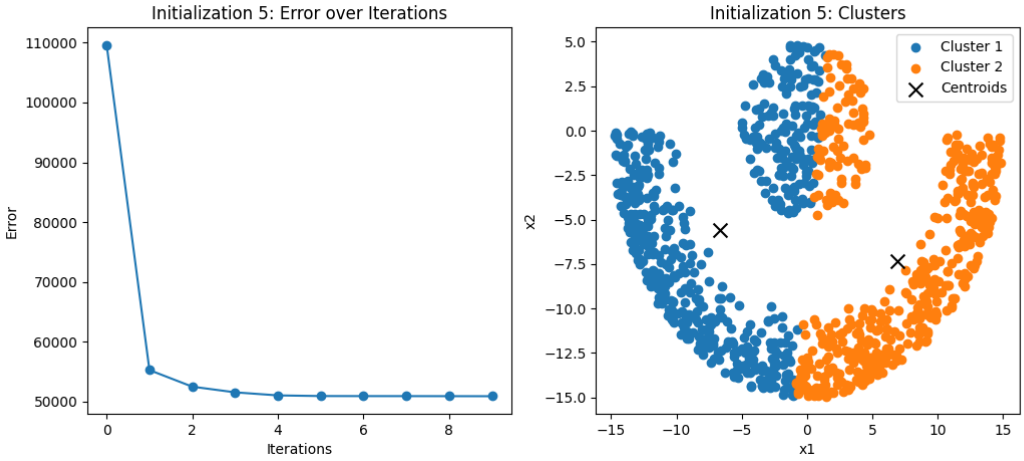
Try 5 different random initialization and plot the error function w.r.t iterations in each case. In each case, plot the clusters obtained in different colors.

Here, we tweak the random seed of numpy, to get different initialisation in each trial.





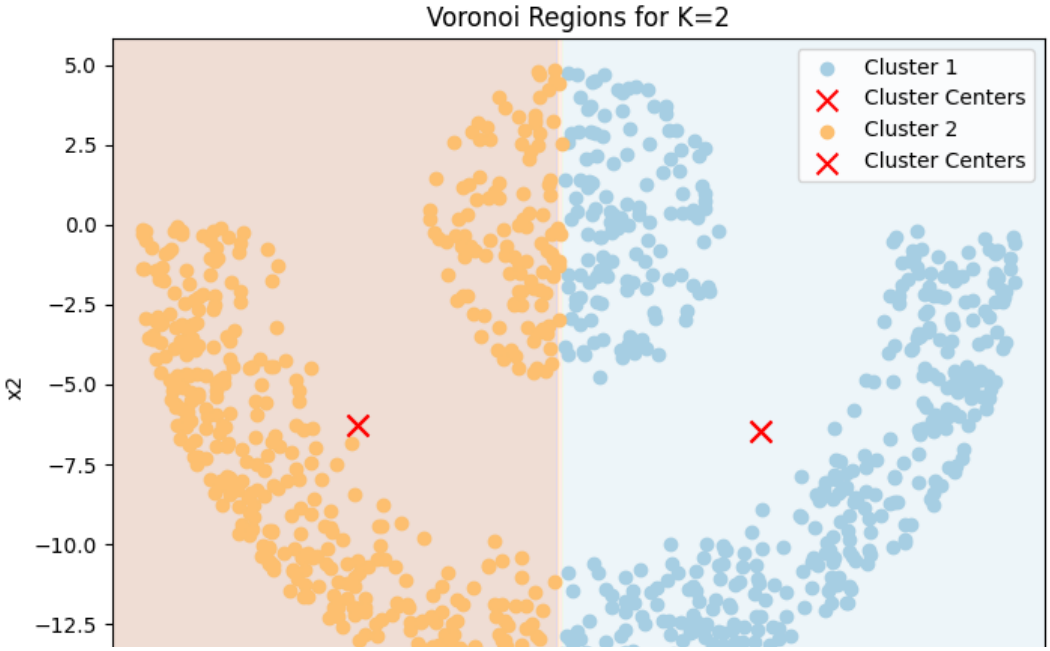


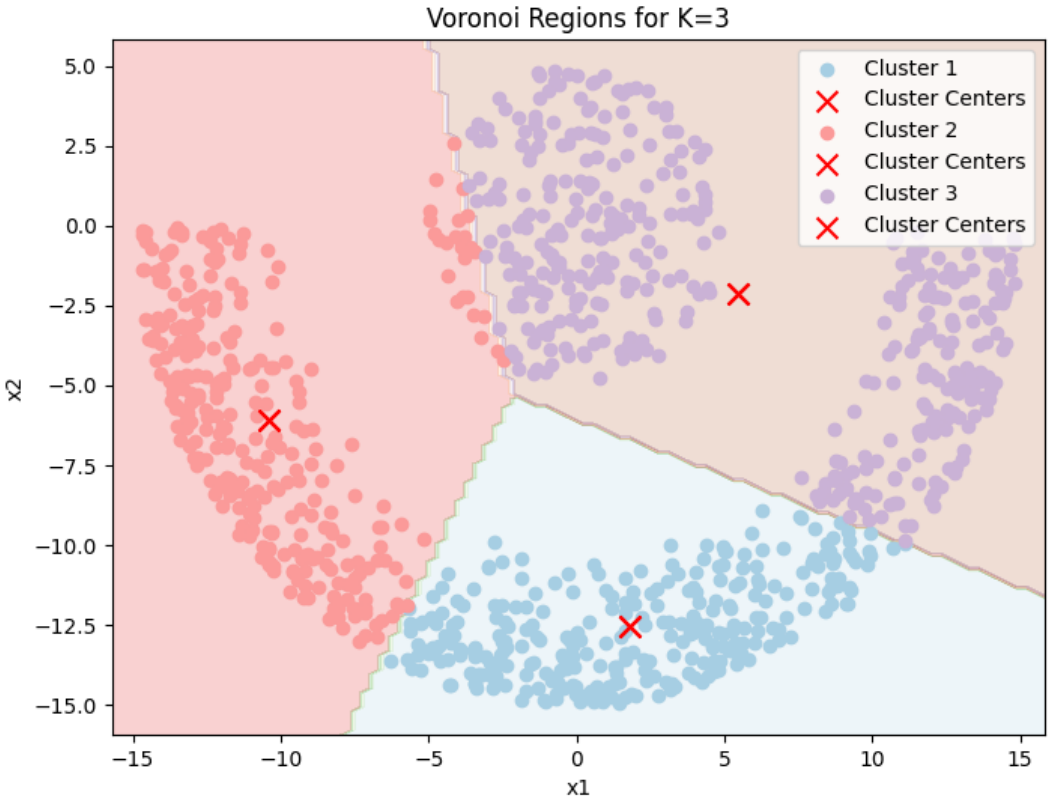


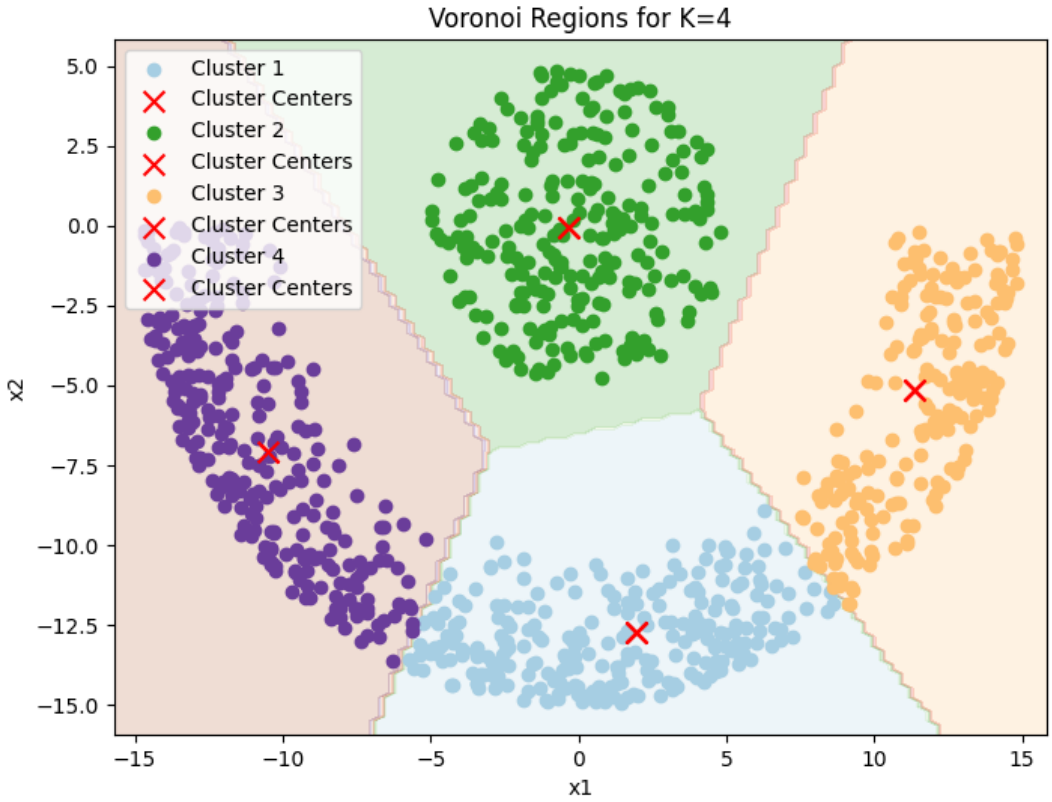
In the above images we can see that, we get different final clusters centroids, with different cluster initialisation. We have used distance between datapoints and its cluster centroids as error function. We can see how the error function value over various iterations is not same for every initialisation. Thus the error function and final clusters that we get in Lloyd's algorithm is prone to initialisation.

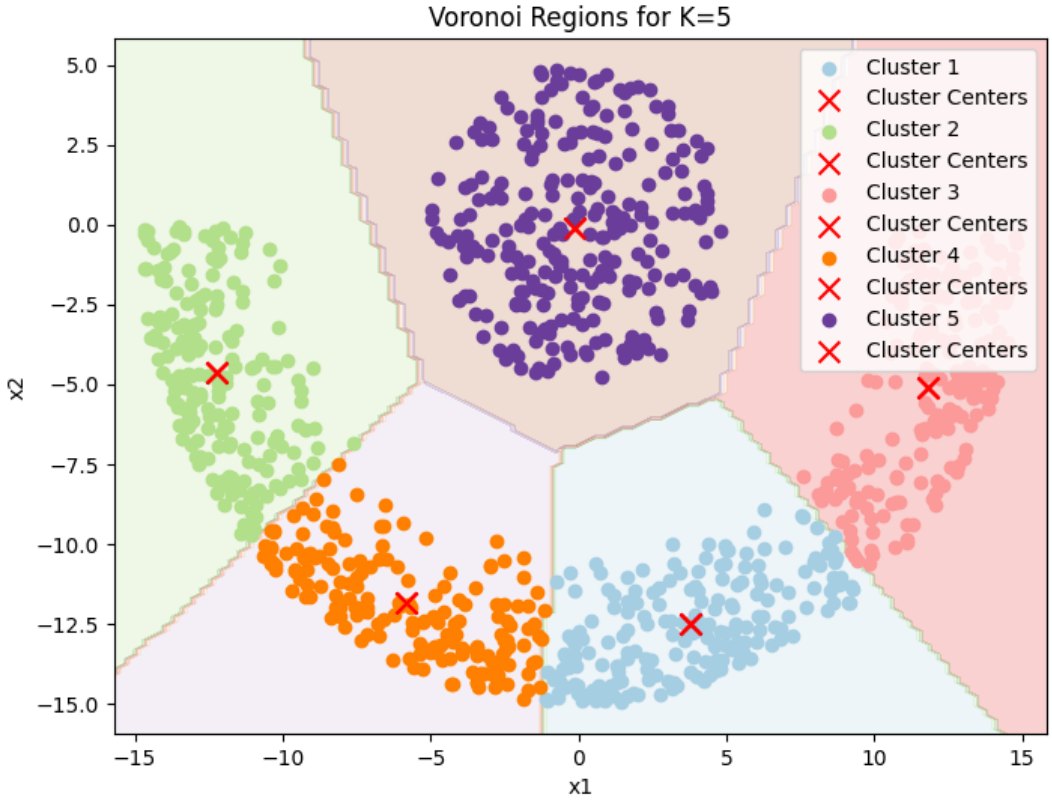
For each K = {2, 3, 4, 5}, Fix an arbitrary initialization and obtain cluster centers according to K-means algorithm using the fixed initialization. For each value of K, plot the Voronoi regions associated to each cluster center.

For the following images, random seed is set as 42. So we will get the same initialisation for every k value.









In the above images, we have plotted the voronoi regions for Lloyd's algorithm for k values 2,3,4 and 5. For k = 4 and 5, we are able to capture points in top circular region as separate cluster, which we cannot do with k = 2 and 3.

If we ask a human kid, that baby would have clustered the given data points into 2 clusters with points in top circular region as one cluster, and points in curved banana shaped region as second cluster, which makes more sense. But our current algorithm is not able to do it.

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?

As said earlier, I don't think Lloyd's algorithm is good in capturing non linear patterns. In the given problem, the data points have non linear relationship, to which our model struggle to capture.

One solution would be to kernalise Lloyd's algorithm. As kernel Kmeans have the capability to capture non linear patterns and can cluster them properly.