Foundations of Machine Learning DA5400
ASSIGNMENT 3

SUBMITTED BY

GLEN PHILIP SEQUEIRA
DA24C005

Contents

1. Folder Structure and Instructions

The folder named `solutions_da24c005` contains:

- `pca.py`
- `kmeans.py`

1.1. Code.

- `pca.py`:It contains the code for implementation of the PCA algorithm, visualization of principal components and reconstruction of the dataset using different dimensional representations.

- `kmeans.py`: It contains the code to implement the Lloyd's algorithm and to plot the cluster regions for the K-means problem. It also plots the Voronoi regions associated to each cluster center for different values of k

1.2. Datasets.

- For the PCA problem, *MNIST* dataset has been used which has obtained using python's `datasets` library.

- For the kmeans problem, `cm_dataset_2.csv` dataset has been used which was provided with the problem statement.

1.3. Instructions.

- All the `.py` files are saved in `solutions_da24c005` zip folder. After extracting the files, run python from the same directory/folder where all these files are located, otherwise it will raise error.

- Kindly ensure that the following python libraries have been installed on your device:
  (1) `pandas` for data handling
  (2) `matplotlib.pyplot` for visualisation
  (3) `load_dataset` from `dataset` to obtain the dataset for PCA problem
  (4) `numpy` for numerical operations

2. Principal Component Analysis

Question 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.

2.1. Implementation of PCA algorithm. :

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

Code:

I have implemented the PCA algorithm using a function called `pca(images)`:

- The parameter of the function, `images` is a matrix, $X \in \mathbb{R}^{n \times d}$. It is the numerical representation of the image dataset with n images and d features.
- The variable `mean` stores the vector, $\mu \in \mathbb{R}^d$, which is calculated for each feature i as:

$$\mu[i] = \frac{1}{n} \sum_{k=1}^{n} X[k, i], \quad \forall i \in \{1, 2, \ldots, d\}$$

- The data matrix, `images` is centered by subtracting the mean from each sample, this corresponds to the operation:

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

- The covariance matrix $C \in \mathbb{R}^{d \times d}$, `cov_matrix` is calculated as:

$$C[i, j] = \frac{1}{n} \sum_{k=1}^{n} X_{\text{centered}}[k, i] \cdot X_{\text{centered}}[k, j], \quad \forall i, j \in \{1, 2, \ldots, d\}$$

- The eigenvalues λ and eigenvectors v are obtained by solving the eigenvalue equation:

$$Cv = \lambda v$$

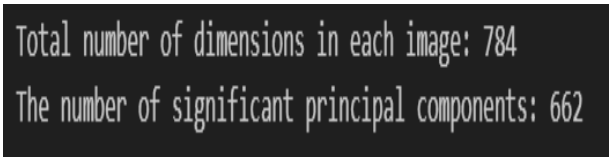
  In the code, these values are stored in the variables, `eigen_values` and `eigen_vectors`

- We sort the eigenvalues in descending order and their corresponding eigenvectors v are the principal components in the descending order of the variance they capture:

$$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d$$

- The function returns:
  (1) `eigen_values`, representing the variance explained by each principal component $\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$.
  (2) `eigen_vectors`, representing the directions of the principal components $\{v_1, v_2, \ldots, v_n\}$.
  (3) `mean`, which is used to center the data.

```python
def pca(images):
    m, n = images.shape
    mean = np.zeros(n)
    for i in range(n):
        mean[i] = np.sum(images[:, i]) / m
    images = images - mean
    cov_matrix = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            cov_matrix[i, j] = np.sum(images[:, i] * images[:, j]) / (m)
    eigen_values, eigen_vectors = np.linalg.eigh(cov_matrix)
    indices = np.argsort(eigen_values)[::-1]
    eigen_values = eigen_values[indices]
    eigen_vectors = eigen_vectors[:, indices]
    return eigen_values, eigen_vectors, mean
```



Observations:

From this above image, we can say that originally each image is represented in a 784-dimensional space (flattened 28×28 pixel images). But after performing PCA, we observe that it actually lies in a 662-dimensional subspace. We say this because there are 662 significant principal components corresponding to non-zero eigen values. The remaining 122 principal components correpond to zero eigen values and do not contribute to any variance or information of the dataset.

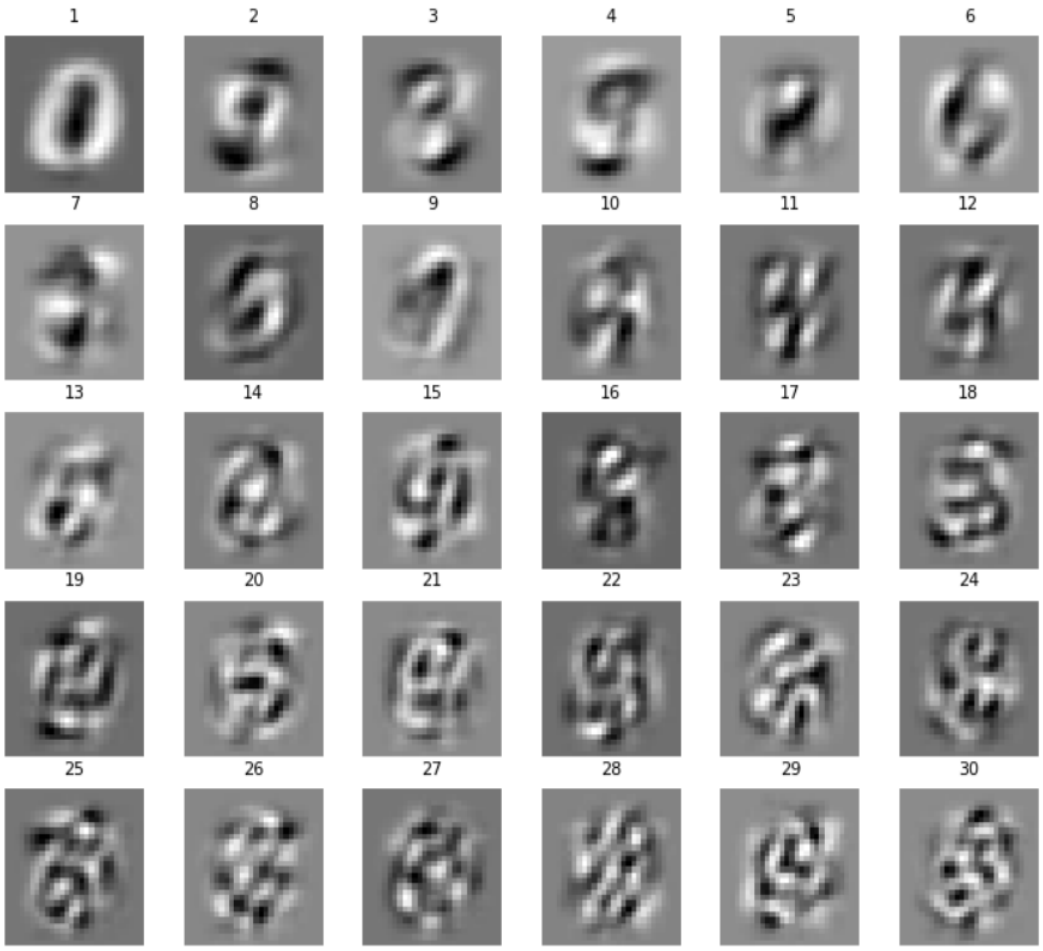


FIGURE 1.

- If the dataset is mean centered, the covariance matrix C is defined as:

$$C = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{x}_i)(\mathbf{x}_i)^\top$$

- The eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_d$ and eigenvectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_d$ of C satisfy the equation:

$$C\mathbf{v}_j = \lambda_j \mathbf{v}_j$$

  where λ_j represents the variance along the direction defined by $\mathbf{v}_j$

- The variance explained by the j-th principal component,$\mathbf{v}_j$ is equal to its eigenvalue λ_j:

$$\lambda_j = \frac{1}{n}\sum_{i=1}^{n}\left((\mathbf{x}_i^\top \mathbf{v}_j)^2\right)$$

- The proportion of variance explained by the j-th principal component is given by:

$$\text{Variance Explained (\%)} = \frac{\lambda_j}{\sum_{k=1}^{d}\lambda_k} \times 100$$

Principal Component No	Variance Explained (%)
1	9.703544
2	6.805271
3	6.351726
4	5.549296
5	4.759953
6	4.276089
7	3.336327
8	2.927904
9	2.779123
10	2.386621
11	2.234308
12	1.996646
13	1.766235
14	1.689030
15	1.608003

Principal Components	Variance Explained (%)
3	22.860541
5	33.169790
10	48.875855
25	69.846469
50	83.384692
75	89.238257
100	92.488252
125	94.498648
134	95.047427
150	95.869302
175	96.850779
200	97.587079
400	99.822632
662	100.000000
784	100.000000

FIGURE 2.

The above tables tell us how much of the variance in the data-set is explained by the principal components.

- The first principal component is the most important direction, capturing 9.7% variance of the dataset. The subsequent principal components are in the descending order of importance (variance explained).

- The first 10 principal components explain roughly half of the variance in the data. Roughly 134 principal components explain about 95% or most of the variance in the dataset.

- The dataset originally has 784 dimensions but we observe that 662 principal components are enough to explain the entire information of the entire dataset. We can reconstruct the entire dataset as before, even if we drop the 122 directions because they don't contribute any information.

2.2. Reconstruction of the Dataset. :

1.(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?

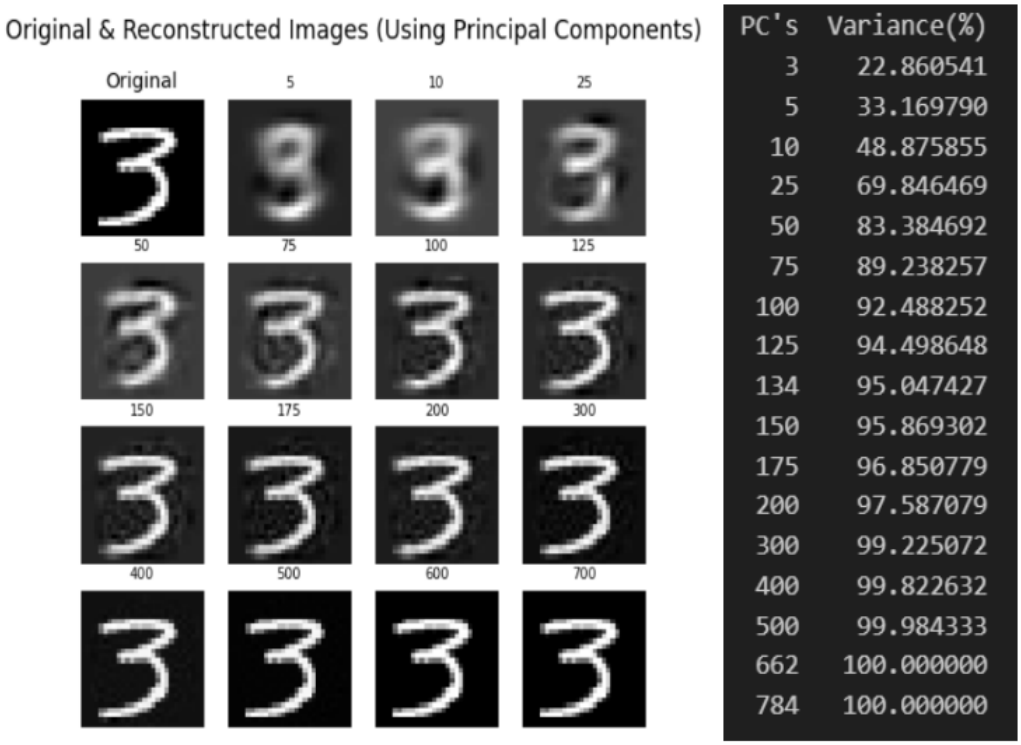
The `reconstruct_images(images, dim)` is used to reconstruct `images` using the first `dim` principal components:

- We apply the `pca(images)` function on the `images` dataset. Using `mean`, we center the dataset.
- Using `pca` function, we get `eigenvectors` which contain the principal components $\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_d$ in the descending order of importance.
- `dim` is a list which contains values for reconstruction of the dataset using different dimensionality representations. We perform reconstruction for different values of d, using only the top d principal components.

$$\hat{\mathbf{x}}_i = \mu + \sum_{j=1}^{d} (\mathbf{x}_i^\top \mathbf{w}_j)\mathbf{w}_j$$

- The function returns `imgs_recon` which contains the reconstruction of the images using different values of d.

```python
def reconstruct_images(images, dim):
    _, eigenvectors, mean = pca(images)
    images = images -mean
    imgs_recon = []
    for d in dim:
        d_eigenvectors = eigenvectors[:, :d]
        img_recon = ((images @ d_eigenvectors) @ d_eigenvectors.T)+mean
        imgs_recon.append(img_recon)
    return imgs_recon
```



PC's	Variance(%)
3	22.860541
5	33.169790
10	48.875855
25	69.846469
50	83.384692
75	89.238257
100	92.488252
125	94.498648
134	95.047427
150	95.869302
175	96.850779
200	97.587079
300	99.225072
400	99.822632
500	99.984333
662	100.000000
784	100.000000

FIGURE 3.

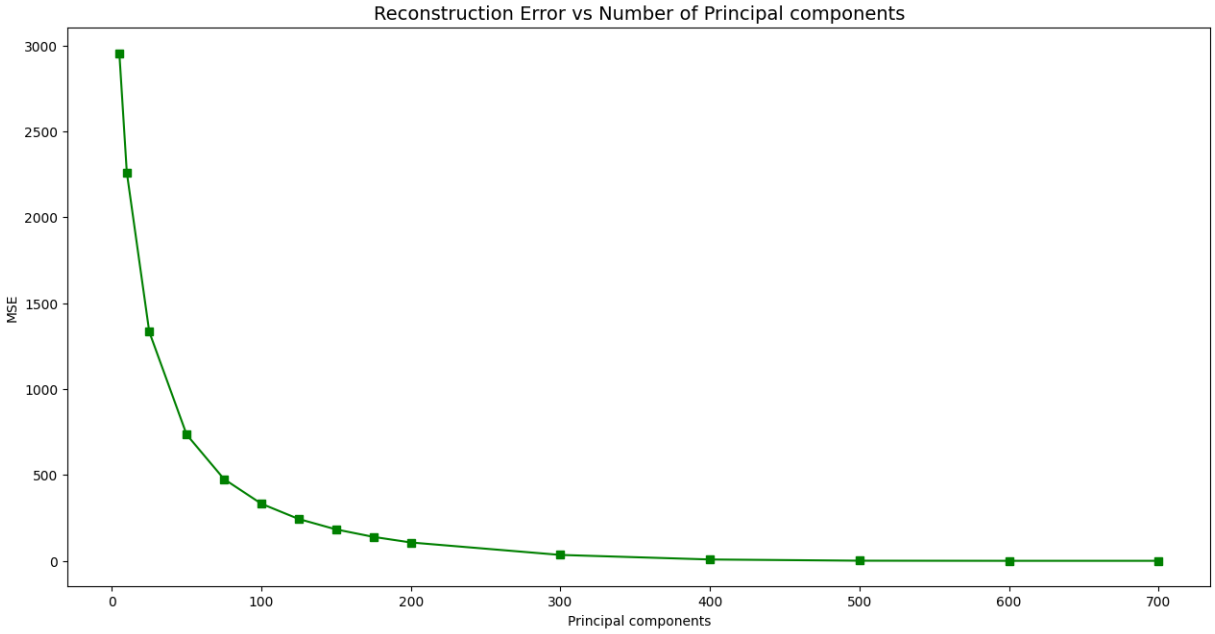


FIGURE 4.

For reconstruction purpose, we use only a single image of digit 3. In Figure 3, we visualize the reconstructions of a image sample of digit 3 using different number of principal components (dimensionality representations).

Figure 4 visualizes the reconstruction error (MSE) as a function of the number of principal components. As we increase the number of principal components, the error between the reconstructed image and the actual image decreases.

Reconstructing the dataset involves a trade-off between reconstruction error and the number of principal components used. Higher the number of principal components we use, lower is the reconstruction error and higher is the clarity of the reconstructed images. But, using more principal components can lead to memory or computational issues.

Usually, 95% of the variance captures most of the variance or information in the dataset. The remaining variance usually corresponds to noise or minor details which are irrelevant. We observe in figure 3 (above), 95% of the variance is captured by 134 principal components. Also, between 125 and 150 components, the clarity of the image is good enough to distinguish that it is the digit 3.

Therefore,for downstream tasks like digit classification $d = \mathbf{134}$ is a suitable choice. This helps us to reconstruct the data with decent clarity, while reducing dimensionality of the images from 784.

3. K-means Clustering

3.1. Implementation of Lloyd's Algorithm. :

Code

Initialization:

z_i^t are the cluster indicators of the i-th data point at iteration t, where $i \in \{1, 2, \ldots, n\}$ and $z_i^t \in \{1, 2, \ldots, k\}$. We randomly initialize the cluster means μ_k^0, for $k \in \{1, 2, \ldots, k\}$.

Calculating Means:

At iteration t, the mean of each cluster k is calculated using:

$$\mu_k^t = \frac{\sum_{i=1}^{n} x_i \cdot \mathbb{I}(z_i^t = k)}{\sum_{i=1}^{n} \mathbb{I}(z_i^t = k)}$$

- $\mathbb{I}(z_i^t = k)$ is an indicator function, equal to 1 if $z_i^t = k$, and 0 otherwise.
- μ_k^t is the mean of the data points assigned to cluster k at iteration t.

Reassignment:

For each $i \in \{1, 2, \ldots, n\}$, update the cluster assignment as:

$$z_i^{t+1} = \underset{k \in \{1,2,\ldots,k\}}{\arg\min} \|x_i - \mu_k^t\|^2$$

Convergence condition:

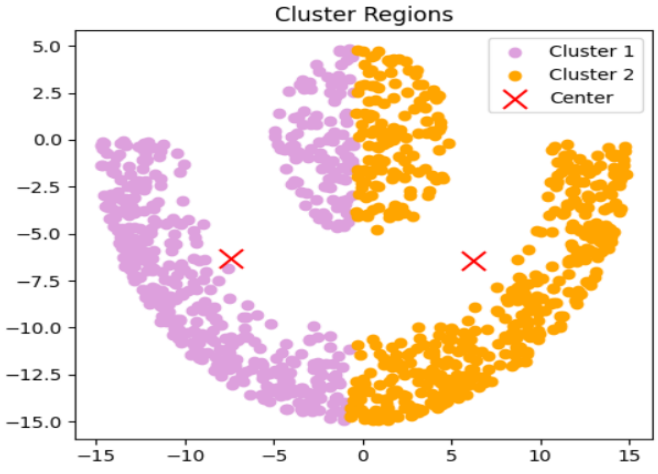
The convergence condition is met when the change in the cluster centers between two consecutive iterations is negligible:

$$\|\mu_k^{(t+1)} - \mu_k^t\| < \epsilon \quad \text{for all} \quad k \in \{1, 2, \ldots, k\}$$

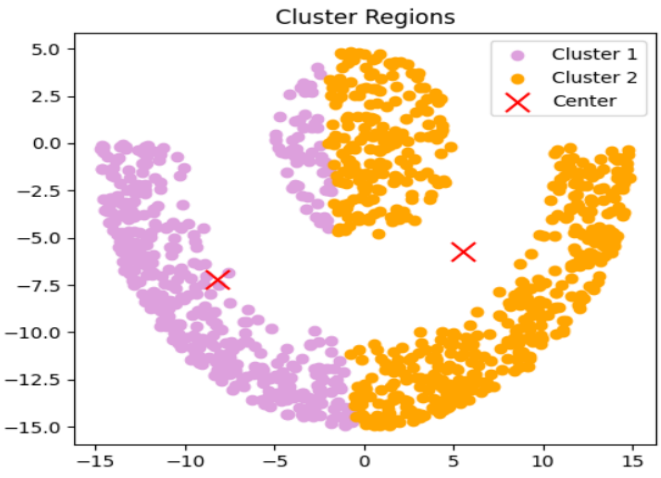
- ϵ is a small positive value. I have taken ϵ as 10^{-6}

```python
def Llyods(X, k,rs):
    np.random.seed(rs)
    n, d = X.shape
    means = X[np.random.choice(n, k, replace=False)]
    indicators = np.zeros(n)
    errors = []
    iteration = 0
    while True:
        iteration += 1
        for i in range(n):
            pt = X[i]
            distances = np.linalg.norm(pt - means, axis=1)
            indicators[i] = np.argmin(distances)
        new_means = np.empty((k, d))
        for j in range(k):
            pts = X[indicators== j]
            if len(pts) > 0:
                new_means[j] = np.mean(pts, axis=0)
        error = calculate_error(X, means, indicators, k)
        errors.append(error)
        if np.linalg.norm(new_means - means) < 1e-6:
            break
        means = new_means
    return means, indicators, errors, iteration
```

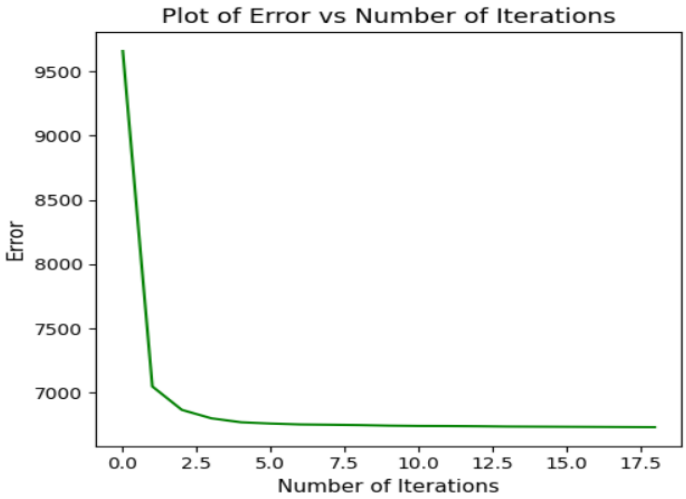
Initialization 1

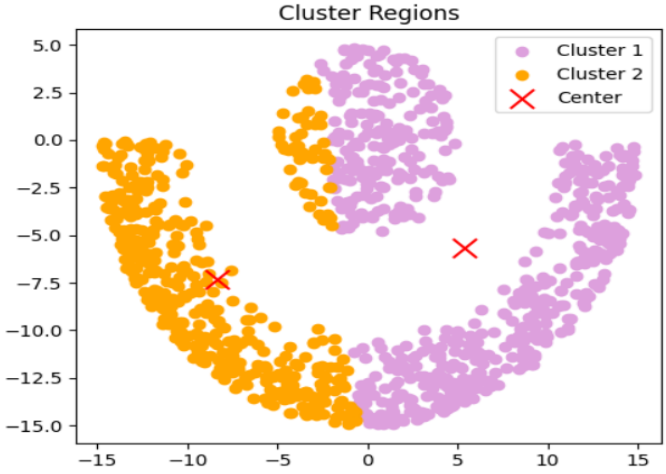


Initialization 2

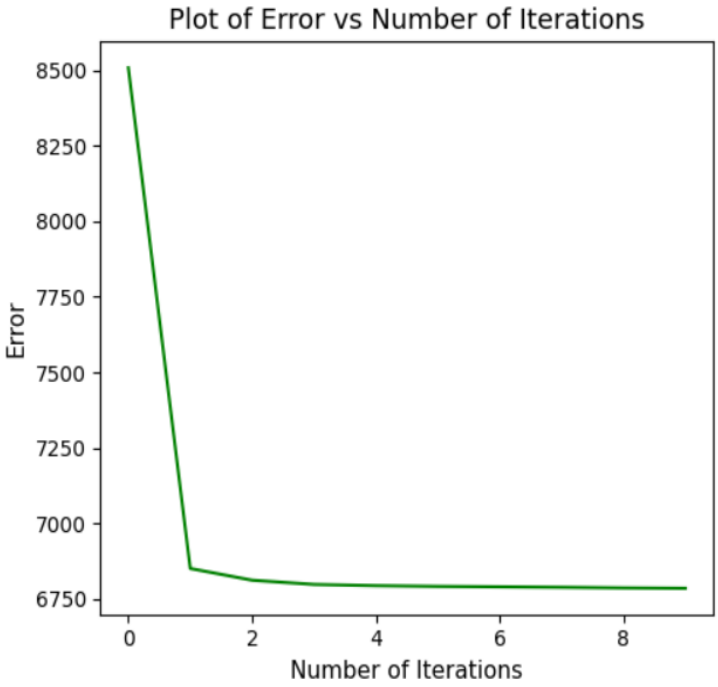


Initialization 3

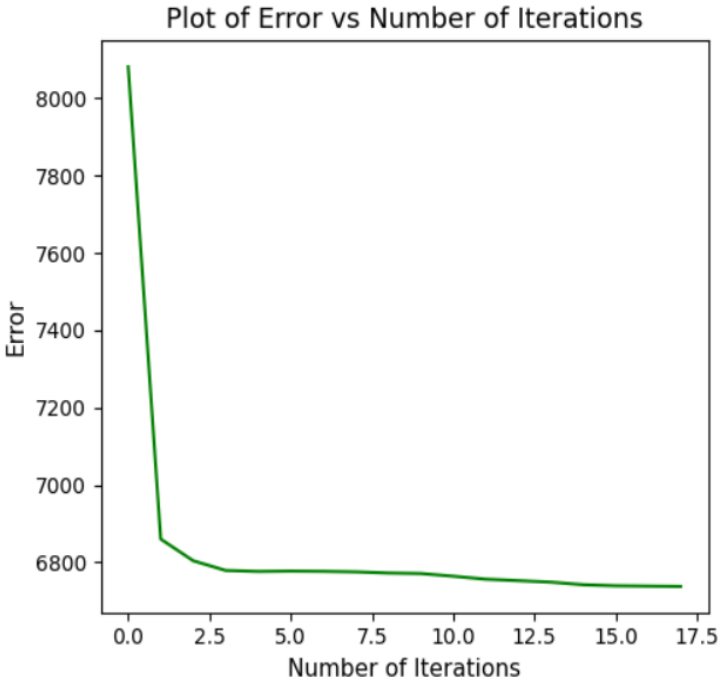


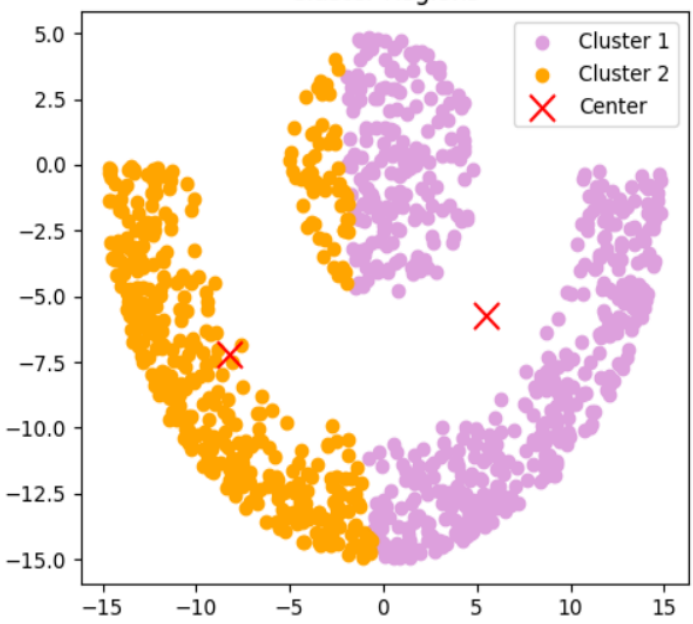


Initialization 4



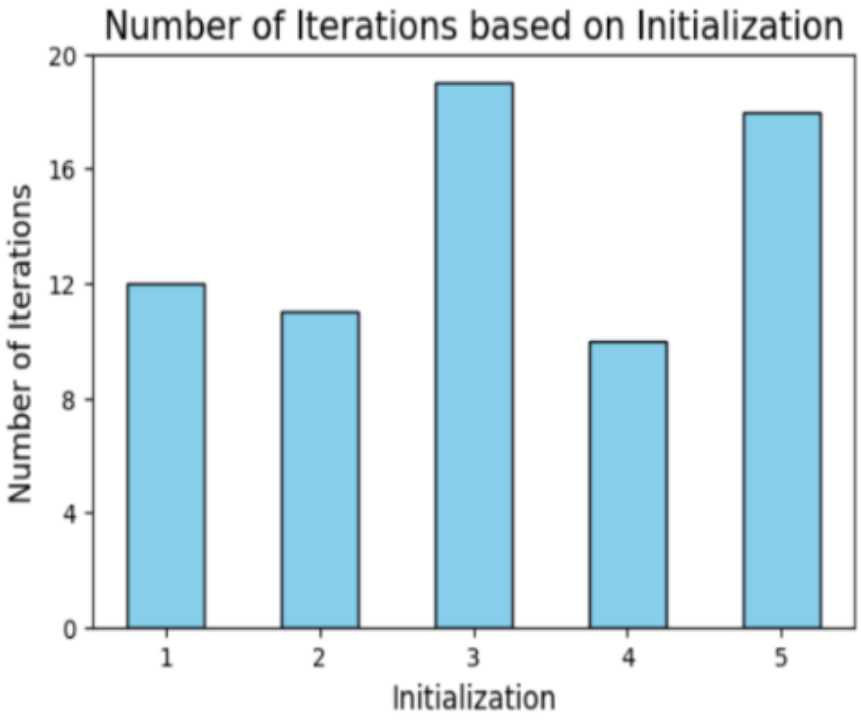
Initialization 5





Observations:
From the plots of the cluster regions, we observe that k-means clustering with k=2 is not a suitable way to cluster these datapoints. We clearly see that the datapoints lie in two groups, one in a circular cluster and another in a semi-circular cluster. k-means clustering is unable to detect these clusters. We will discuss an alternative solution in the subsection, *Alternative Solution*

Figure 5 (below) shows that in all the 5 initializations, the Llyod's algorithm converges but different initializations result in different number of iterations and different final errors. This is because we are initializing the means randomly. Random initialization could result in Lylod's algorithm converging towards 'bad' clusters with high error or taking lot of iterations. To avoid this, we can do the initialization of the means in a more principled approach using the k-means++ algorithm where the means are probabilistically chosen to be as far apart as possible



Initialization	Final Error	Iterations
1	6784.154136	12
2	6737.265394	11
3	6730.698641	19
4	6784.154136	10
5	6737.265394	18

FIGURE 5.

3.2. Vornoi Regions. :

(ii) For each $K = \{2, 3, 4, 5\}$, fix an arbitrary initialization and obtain cluster centers according to the K-means algorithm using the fixed initialization. For each value of K, plot the Voronoi regions associated with each cluster center. (You can assume the minimum and maximum values in the dataset to be the range for each component of $\mathbb{R}^2$).

Code

My implementation visualizes the Voronoi regions formed by the cluster centers obtained through the Lyod's algorithm for $K = \{2, 3, 4, 5\}$:

- For each $K \in \{2, 3, 4, 5\}$, the Lloyd's algorithm (`Llyods(X,k,rs)`) is used to compute K cluster centers/means $(\mu_1, \mu_2, \ldots, \mu_K)$ based on the given dataset.

- A uniform grid of points $\mathbf{g}_i$ is created over the range of the dataset using `np.meshgrid`. Each grid point is represented as:
$$\mathbf{g}_i = (x, y)$$
These grid points are combined into a 2D array `grid`

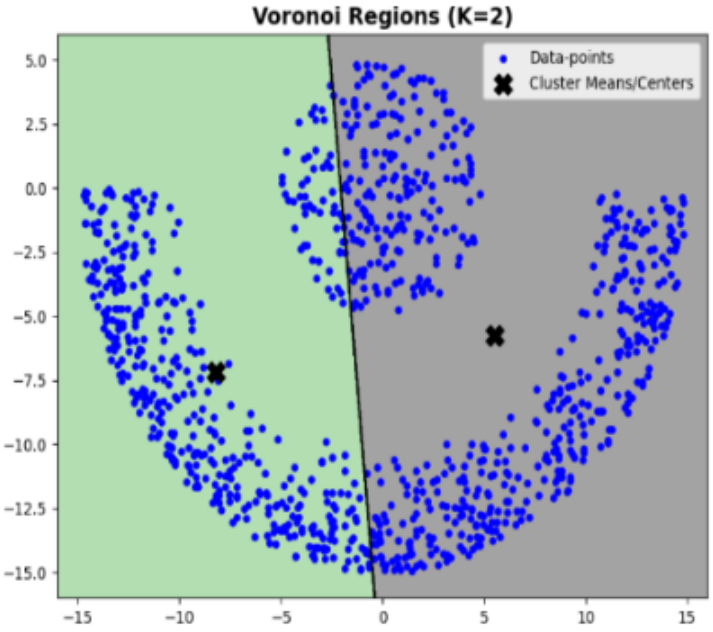
- For each grid point $\mathbf{g}_i$, its distance to each cluster center is calculated as:
$$d_{ij} = \|\mathbf{g}_i - \mu_j\|_2, \quad j = 1, 2, \ldots, K$$
These distances are stored in a list `distances`.

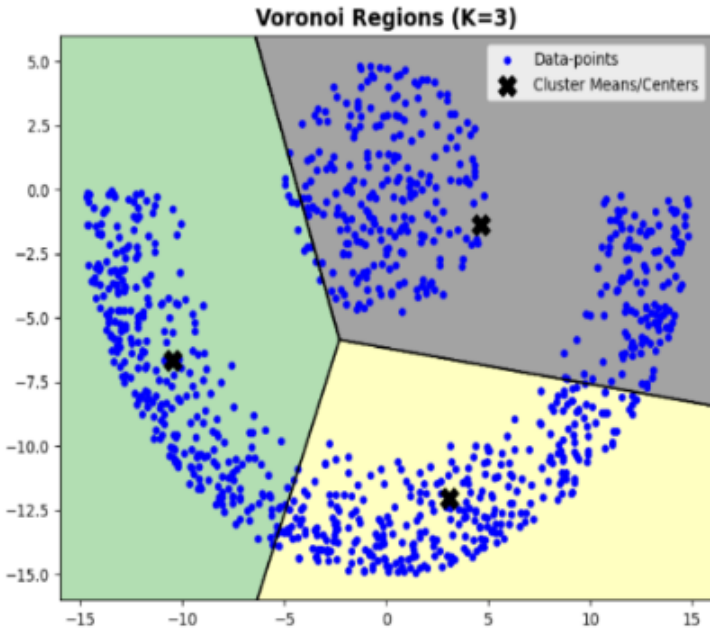
- Each grid point is assigned to the nearest cluster center:
$$\text{label}(\mathbf{g}_i) = \arg\min_j \|\mathbf{g}_i - \mu_j\|_2$$
`pt_labels` indicates the Voronoi region each grid point belongs to.

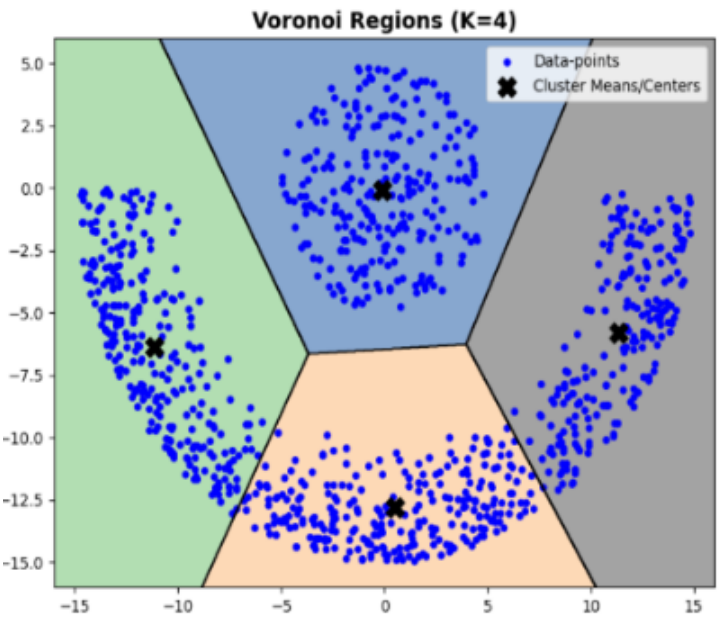
- The Voronoi regions are visualized using `plt.contourf` and boundaries between the regions are highlighted using `plt.contour`. The Voronoi region for cluster center μ_j consists of all points $\mathbf{g}_i$ closer to μ_j than to any other center:
$$\text{Region}(\mu_j) = \{\mathbf{g}_i : \|\mathbf{g}_i - \mu_j\|_2 \leq \|\mathbf{g}_i - \mu_k\|_2, \forall k \neq j\}$$

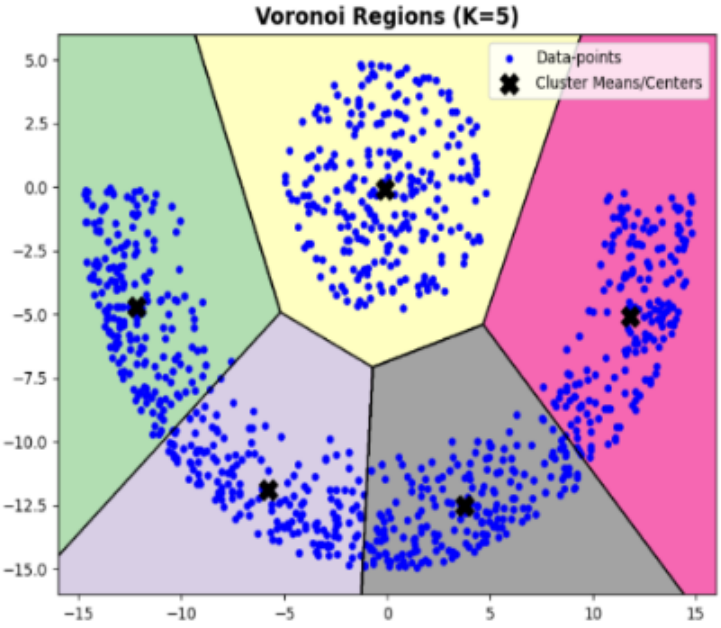
```
for k in [2, 3, 4, 5]:
    means, _, _, _ = Llyods(dataset,k,42)
    xx, yy = np.meshgrid(np.linspace(-16, 16, 1000), np.linspace(-16, 6, 1000))
    grid = np.column_stack((xx.ravel(), yy.ravel()))
    distances = []
    for pt in grid:
        dist = np.linalg.norm(pt - means, axis=1)
        distances.append(dist)
    pt_labels = np.argmin(distances, axis=1)
    plt.figure(figsize=(8, 6))
    plt.contourf(xx, yy, pt_labels.reshape(xx.shape), cmap="Accent", alpha=0.6)
    plt.contour(xx, yy, pt_labels.reshape(xx.shape), colors='black', linewidths=1)
    plt.scatter(dataset[:, 0], dataset[:, 1], color="blue", s=15, label="Data-points")
    plt.scatter(means[:, 0], means[:, 1], color="black", marker="X", s=150, label="Cluster Means/Centers")
    plt.title(f"Voronoi Regions (K={k})",fontsize=14,fontweight='bold')
    plt.legend()
    plt.show()
```

Observations









3.3. Alternative Solution. :

iii. Is the Llyods 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?

Issue

Llyod's algorithm (standard k-means) is not effective for clustering this dataset. Lloyd's algorithm assumes that clusters are convex. But the given dataset has a semi-circular cluster which is non-convex in nature. Also, Vornoi regions make use of linear boundaries but this datasaet has non-linear clusters. Applying the Llyod's algorithm to this problem will give incorrect clusters which is evident in the Cluster regions and Voronoi regions plots.

Fix To handle non-convex clusters, **Kernel K-Means** algorithm is a suitable alternative. Using a kernel function, the dataset is transformed into a higher dimensional feature space where the non-convex clusters are linearly separable.

- First, we use a kernel function, $k(x_i, x_j)$ to calculate the kernel matrix K, where:

$$K_{ij} = k(x_i, x_j)$$

  We choose Radial Basis Function (RBF) kernel because it transforms the features into an infinite dimensional space. Using this we can plot smooth decision boundaries even for curved clusters without overfitting.RBF:

$$k(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right)$$

- We then perform eigen-decomposition of the kernel matrix K to calculate the top K eigenvectors $\{v_1, v_2, \ldots, v_K\}$

- Then, we normalize each row of the eigenvector matrix to unit length and the normalized rows are used as the feature representation for the data points and we use them as initialization for Lloyd's algorithm.

To get an ideal solution for this dataset, we can implement this algorithm and use the output obtained as the initialization for the Lloyds algorithm. The Llyod's algorithm will then group the datapoints into two clusters: a circular cluster and a semi-circular cluster. This approach handles both the non-convexity and non-linearity of the dataset giving required clusters. Also, this way of initialization improves clustering performance compared to random initialization done in standard k-means algorithm.