CMPE 462 – PROJECT 3

Implementing K-Means & PCA

- **Student ID1:** 2014400051
- **Student ID2:** 2018400291
- **Student ID3:** 2019705072

Requirements and How to Run:

- The project uses some python libraries that need to be installed before starting the project. Namely, scipy.io, numpy, matplotlib.
- Also, the data needs to be in the folder data under data which is in the same workspace as the .ipynb file.

If you would like to load data from another folder. Since the data for PCA is matlab file, need to use loadmat function.

Specify it in the data loading cell. Dot means current working directory.

```
#Import required libraries

import scipy.io as spio
import numpy as np
import matplotlib.pyplot as plt

#K-Means Data
data = np.load('./kmeans_data/data.npy')
labels = np.load('./kmeans_data/label.npy')

#PCA Data
dic = spio.loadmat("USPS.mat")
X = dic['A']
```

INTRODUCTION:

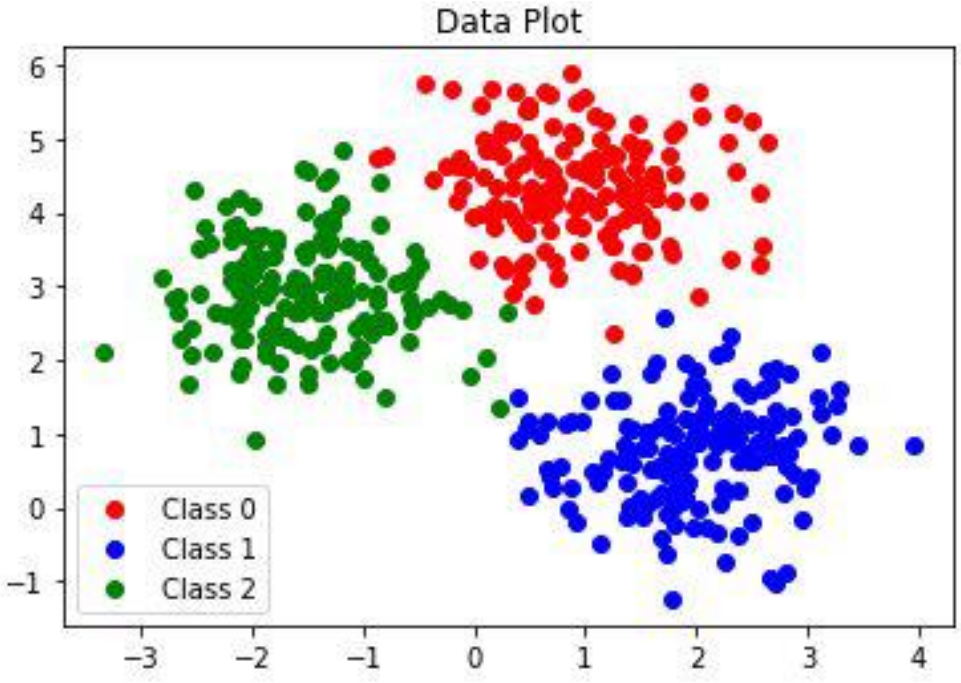
In this project, we implemented k-means clustering and principal component analysis (PCA). After the implementation, for k-means we tried to observe how number of iteration changes the clustering results and for PCA we tried to observe how dimension of the principal component set affects the image of one handwritten digit (between 0 and 9).

TASK 1:

In task 1, we will discover k-means algorithm which is an unsupervised learning algorithm, but it can also be used for supervised learning problems. In our case we know the labels, and we will see how k-means gets closer to the given labels.

Task 1.1: Plot Clusters

We are asked to plot the given labels, the following plot shows the real clusters of our data. As it is seen on the legend, red points for label 0, blues for label 1 and greens for label 2. In the rest of this task, we will stick to the colors.



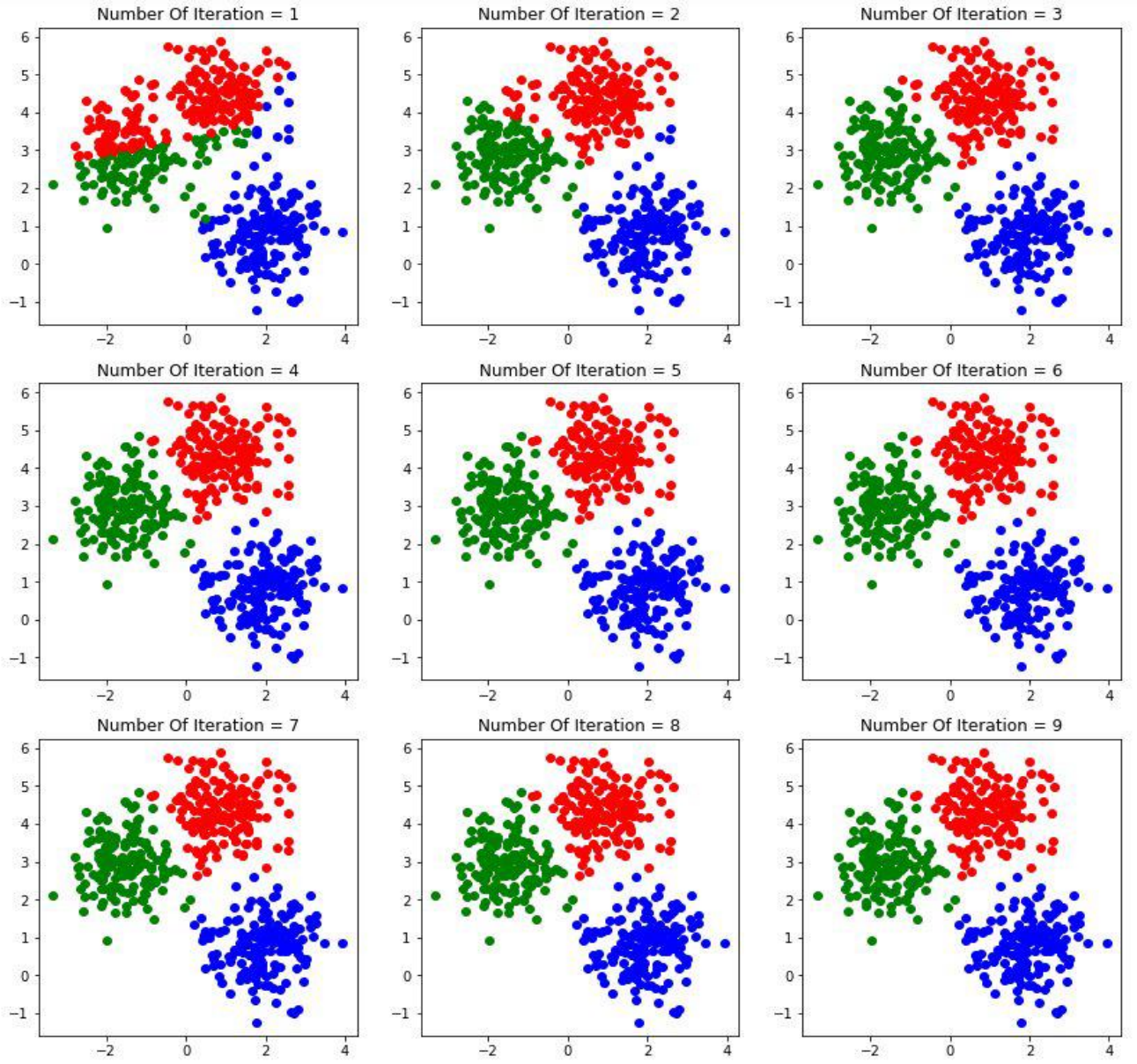
Task 1.2: K-Means Implementation

We construct a class is named as k_means for k-means clustering algorithm. Constructor of the class takes data which is data points, k which is number of clusters and max_iter which is number of iterations as parameters. We have one class method which is fit() method that finds clusters and centroids of these

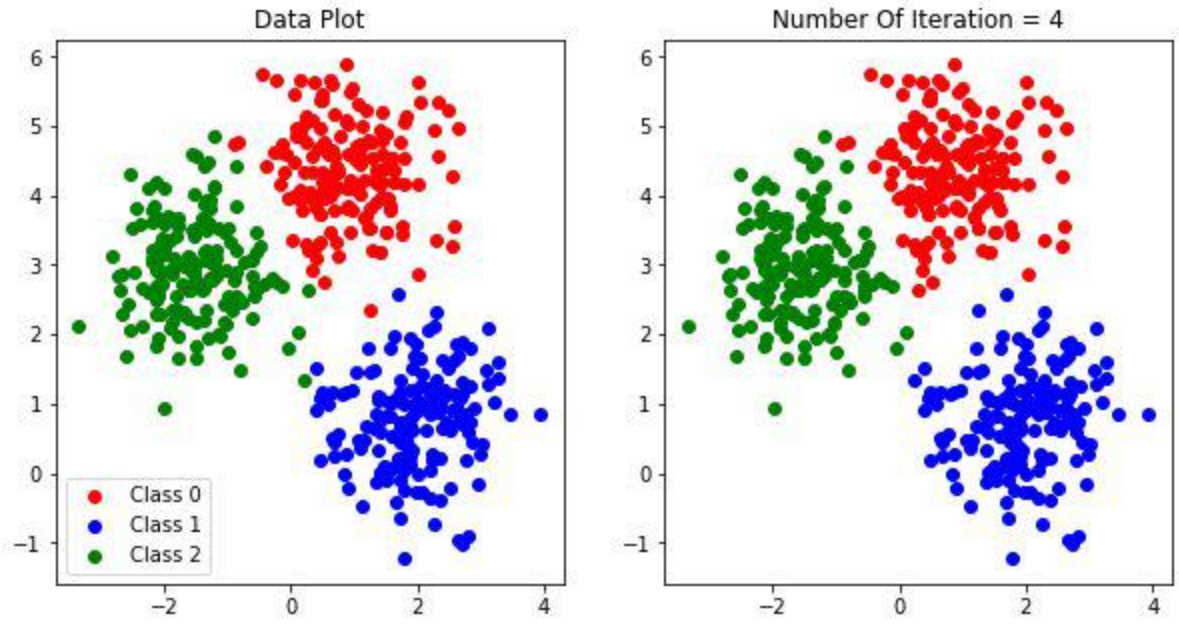
clusters. Firstly, in the fit() method, We set seed for random state as 1 which is mentioned in the guideline then create k which is equal 3 random integers between 0 and number of rows of data set . We use these integers as row indexes to get k points from data set. We initialize centroids randomly via this procedure. We have for loop which iterates max_iter times. Firstly, we initialize empty dictionary in order to keep clusters then initialize k elements of dictionary as empty lists to keep points in the each cluster. For each data points in data set, we calculate Euclidean distance between data point and centroids then assign data point to closest centroid. After that, we recomputed centroid of each cluster via calculating mean of data points in the each cluster. We can access centroids and points in the each clusters by object.centroids and object.classifications, when we construct an object from k_means class and call the fit() function to apply k-means algorithm.

Task 1.3: Evaluation

We are asked to run our k-means implementation 9 times with the number of iterations N = {1,2,3,4,5,6,7,8,9}. Following plot shows the changes of the clusters while we are increasing the number of iterations. Clusters are improved and getting closer to the real one that we plot in task 1.1 until N=4. There is no improvement between N=4 and N=9. When we compare them with the real one, we can say that algorithm has done its job very well. Also, we can easily say that algorithm converged at 4 iterations.



When number of iteration is 4, clusters look like very similar to the real one. To be sure about that, we plot the n=4 and real one again. From the following plot, we see only 3 misclassified data points which is an acceptable result.



In the first iteration, due to randomly initialized centroids, we cannot get optimal clustering. In the second iteration, there are lots of exchange of memberships between green and red clusters. In the third iteration, it is getting closer to optimal clustering. In the fourth iteration, we get optimal clustering with 3 misclassified data points. There is no change on the memberships of clusters and centroids of clusters between N=4 and N=9.

TASK 2:

In task 2, we will discover principal component analysis which aims to reduce the dimensionality of the data set by finding a new set of variables. We will work on image of one handwritten digit between 0 and 9.

Task 2.1: PCA Implementation

Standardization:

In this part, we start with standardizing the input X. Standardizing is making data centered and making each feature has zero mean. It is done simply by subtracting

sample mean from each input row. Sample mean is found using np.mean(X, axis=0) function of numpy. Then, we call standardized array as X.

Covariance:

Covariance matrix is a matrix of size (256, 256) and it is found with formula

S = 1/n * transpose(X) * X where X is standardized input matrix.

Construct PCs:

In this part, we first create eigenvectors and eigenvalues of covariance matrix.

To calculate eigenvectors of X, we use a code piece like this:

eigenValues, eigenVectors = np.linalg.eig(S)

This function calculates eigenvalues and eigenvectors of a given matrix S

Then, we sort eigenvector – eigenvalue pairs with respect to eigenvalues.

idx = eigenValues.argsort()[::-1]

eigenValues = eigenValues[idx]

eigenVectors = eigenVectors[:,idx]

Then, we select d of the eigenvectors and put into matrix G such that columns of G represent eigenvectors.

At the end, G matrix which is linear transformation will be of size (256, d) where d is number of PCs.

PCs matrix which represents principal components will be of size (3000, d)

Task 2.2: Image Reconstruction

In this part, we create reconstructed images from principal components and linear transformation matrix. The formula to create reconstructed images is:

Reconstructed Image = G(G T X T)

Calculating it is pretty plain : G * transpose(XG)

def reconstruction(G, PCs):

XGt = np.transpose(PCs) # XGt is of (d, 3000) size

GGtXt = np.matmul(G, XGt) # GGtXt is of (256, 3000) size

return np.transpose(GGtXt) # (3000, 256)

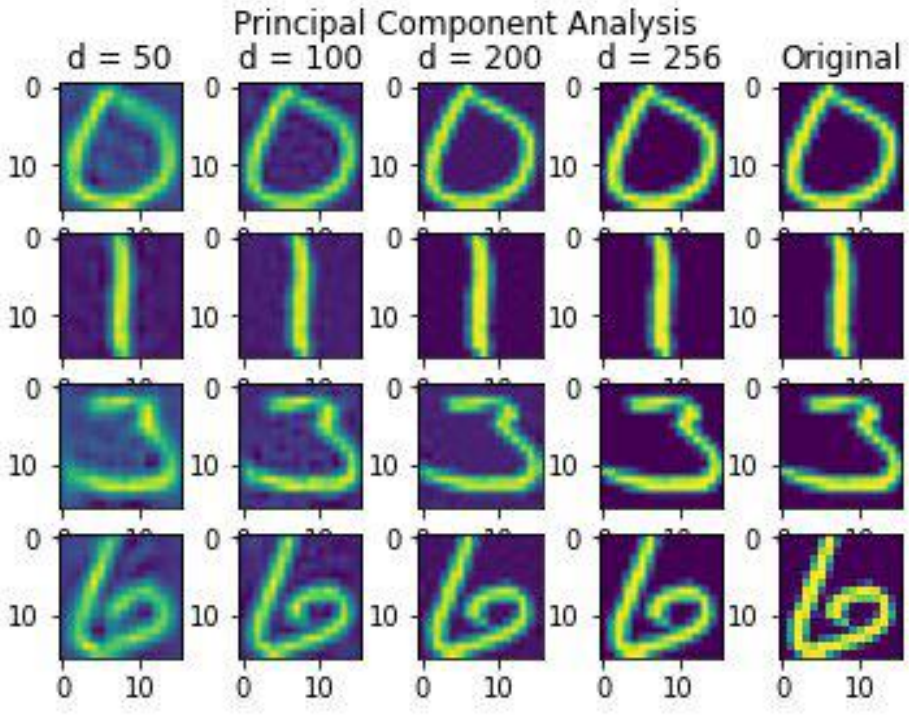
Task 2.3: Evaluation

In this part, we create a 4 by 5 subplot and show the results there. The rows of the subplot represent certain numbers in input with indices (0, 500, 1000, 2000)

One can simply change the indices to view different results of numbers.

for i, index in enumerate([0, 500, 1000, 2000]): # Images at certain indices

Also, the columns of subplot represent results with certain principal component numbers (50, 100, 200, 256 and original image)



From the resulting plot above, we can see that the analysis is pretty successful and as the number of principal components increase, the quality of the images gets better and images gets closer to originals. Also, when number of principal components decrease, reconstruction error increases. If some valuable information is lost due to lack of certain principal components, then, a differentiator may have a hard time classifying the input images. From d=100 to d=200, images are getting closer to originals. In d=50 and d=100, images are looking blurred. When number of principal components is equal to 50, handwritten 1 is closer to its original than others. Maybe, we can say that intrinsic dimension of handwritten 1 is smaller than others by looking these pictures.