Clustering Part 2

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

DBSCAN Algorithm

DBSCAN(Density-Based Spatial Clustering of Applications with Noise) is a commonly used unsupervised clustering algorithm. DBSCAN does not need to specify the number of clusters. It can automatically detect the number of clusters based on your input data and parameters. More importantly, DBSCAN can find arbitrary shape clusters that k-means are not able to find.

Algorithm:

a. The algorithm proceeds by arbitrarily picking up a point in the dataset (until all points have been visited).

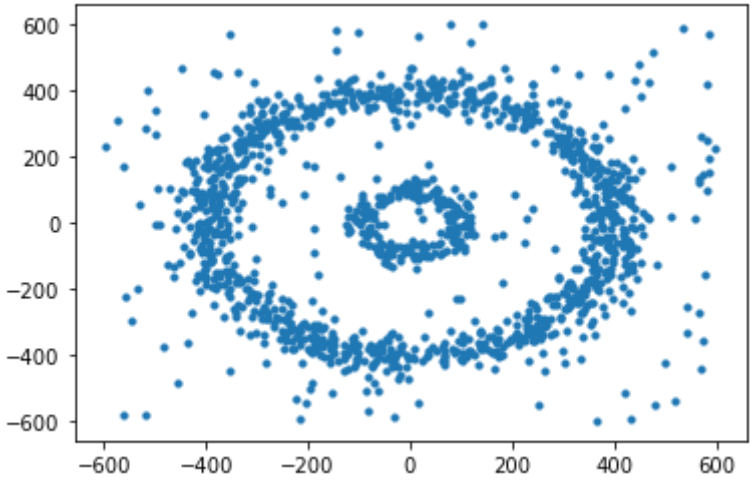
b. If there are at least ‘minPoint’ points within a radius of ‘ε’ to the point then we consider all these points to be part of the same cluster.

c. The clusters are then expanded by recursively repeating the neighborhood calculation for each neighboring point

A. **Generate "N" spherical training data points.**

In [ ]:
```
from random import random


def gen_circular_data(n, radius, noise):
        thetas = np.random.uniform(0, 2 * np.pi, n)
        data = (radius*np.array([np.cos(thetas), np.sin(thetas)])).T + np.ra
        return data

# Generate circular data
cir1 = gen_circular_data(1000, 400, 30)
cir2 = gen_circular_data(200, 100, 20)
outliers = np.random.uniform(-600, 600, (200, 2))

points = np.concatenate((cir1, cir2, outliers), axis=0)

# Plot the data
plt.scatter(points[:,0], points[:,1], s=10)
```

Out[ ]:
```
<matplotlib.collections.PathCollection at 0x7f0ca0f7ed70>
```

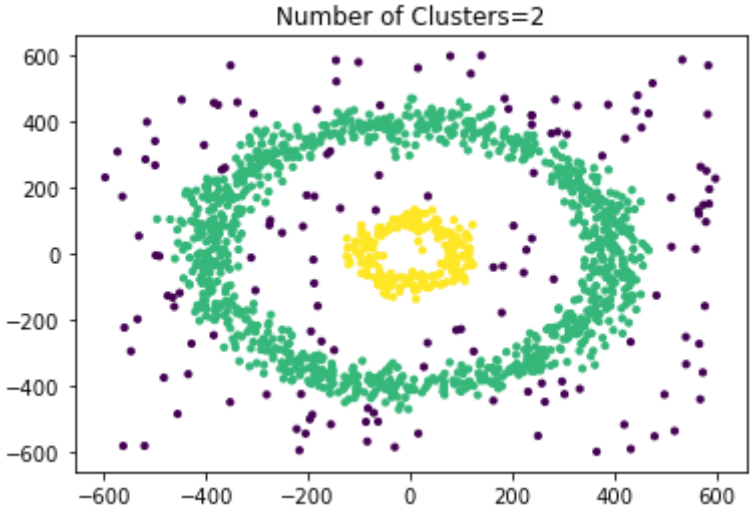


B. **Perform DBSCAN Algorithm on the above generated data to obtain clusters**

In [ ]:

```
from cProfile import label
from tqdm import tqdm

# DBSCAN
class DBSCAN:
        def __init__(self, eps, min_pts):
                self.eps = eps
                self.min_pts = min_pts

        def fit(self, X):
                self.X = X
                self.n = X.shape[0]
                # Compute distance matrix
                self.dist_matrix = np.zeros((self.n, self.n))
                for i in range(self.n):
                        for j in range(self.n):
                                self.dist_matrix[i, j] = np.linalg.norm(self
                # Find neighbors
                self.neighbors = []
                for i in range(self.n):
                        self.neighbors.append(self.find_neighbors(i))

                # Compute labels
                self.labels = np.zeros(self.n)
                cluster_id = 0
                for i in range(self.n):
                        if self.labels[i] == 0:
                                neighbors = self.neighbors[i]
                                if len(neighbors) < self.min_pts:
                                        self.labels[i] = -1
                                else:
                                        cluster_id += 1
                                        self.labels[i] = cluster_id
                                        self.expand_cluster(i, cluster_id)

                self.num_clusters = cluster_id

                return cluster_id

        def expand_cluster(self, i, cluster_id):
                opened = set([i])
                while len(opened) > 0:
                        j = opened.pop()
                        neighbors = self.neighbors[j]
                        for k in neighbors:
```

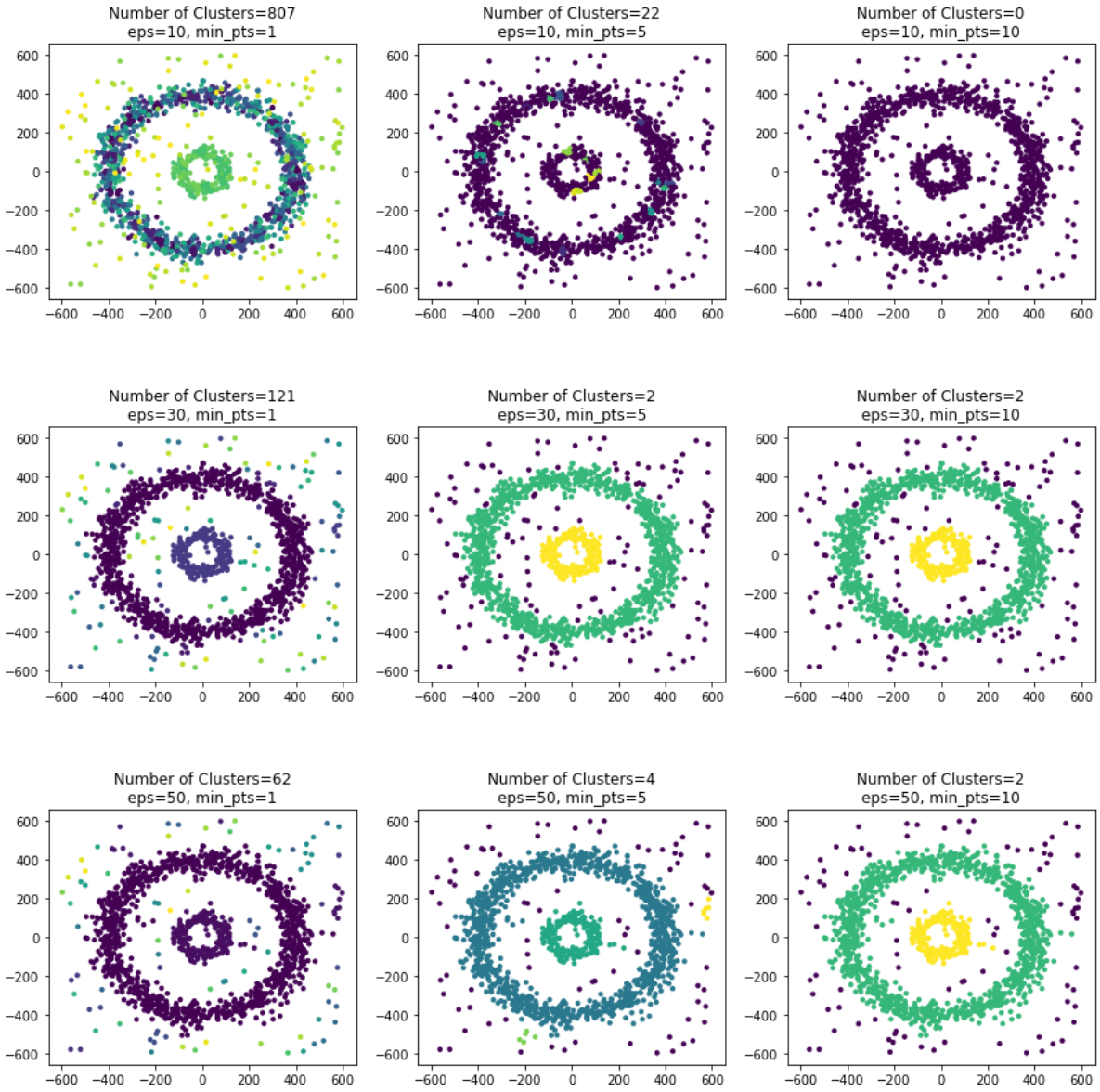
```
                                if self.labels[k] == 0:
                                        self.labels[k] = cluster_id
                                        opened.add(k)
                                elif self.labels[k] == -1:
                                        self.labels[k] = cluster_id


        def find_neighbors(self, i):
                neighbors = []
                for j in range(self.n):
                        if self.dist_matrix[i, j] <= self.eps:
                                neighbors.append(j)
                return neighbors

        def plot(self):
                plt.title(f"Number of Clusters={self.num_clusters}")
                plt.scatter(self.X[:,0], self.X[:,1], c=self.labels, s=10)
                plt.show()

# Run DBSCAN
dbscan = DBSCAN(eps=30, min_pts=5)
dbscan.fit(points)
dbscan.plot()
```



C. **Experiment by varying the number of min points and epsilon radius and plot your observations**

In [ ]:

```
eps = [10, 30, 50]
min_pts = [1, 5, 10]

fig, axs = plt.subplots(3, 3, figsize=(15, 15))
fig.subplots_adjust(hspace=0.5)
for i in range(3):
        for j in range(3):
                dbscan = DBSCAN(eps=eps[i], min_pts=min_pts[j])
                num_clusters = dbscan.fit(points)
                axs[i, j].scatter(dbscan.X[:,0], dbscan.X[:,1], c=dbscan.lab
                axs[i, j].set_title(f"Number of Clusters={num_clusters}\neps

plt.show()
```

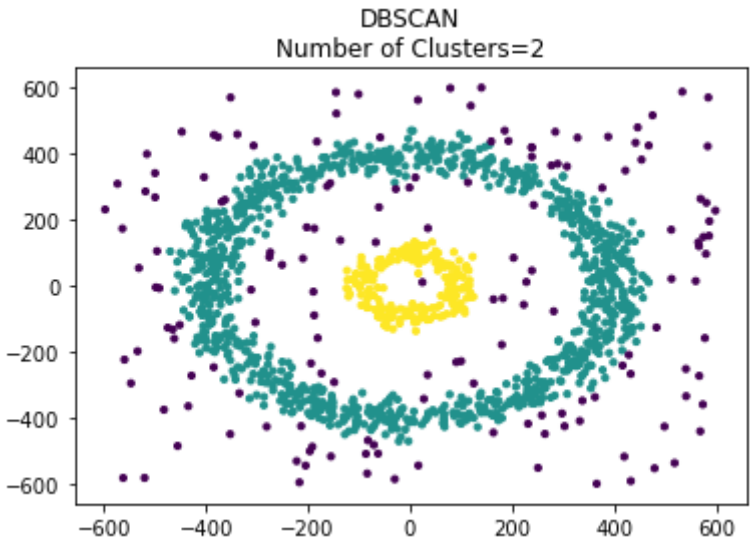


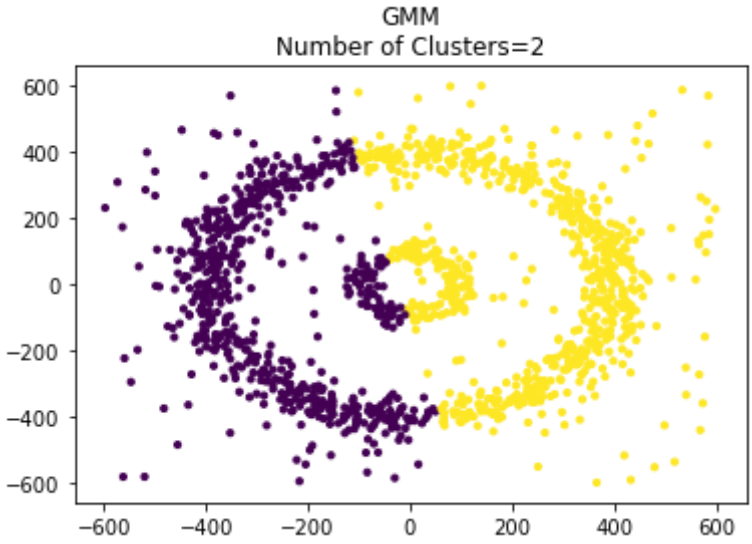
D. **Compare your model with the built in DBSCAN in Sci-kit Learn. Also compare you results with GMM and the K-means Algorithm**

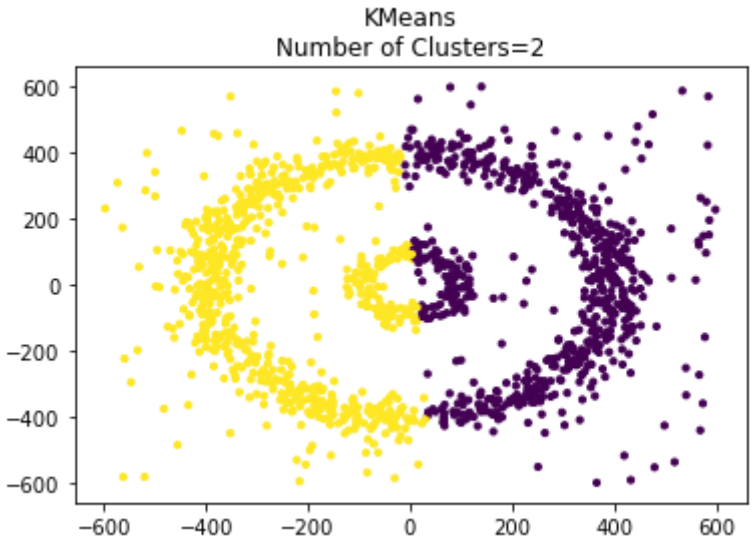
In [ ]:

```
from sklearn.cluster import DBSCAN
# Use DBSCAN from sklearn
dbscan = DBSCAN(eps=30, min_samples=5)
dbscan.fit(points)
plt.scatter(points[:,0], points[:,1], c=dbscan.labels_, s=10)
plt.title(f"DBSCAN\nNumber of Clusters={len(set(dbscan.labels_))-1}")
plt.show()

##########################################################################
from sklearn.mixture import GaussianMixture
# Use GMM from sklearn
gmm = GaussianMixture(n_components=2)
gmm.fit(points)
plt.scatter(points[:,0], points[:,1], c=gmm.predict(points), s=10)
plt.title(f"GMM\nNumber of Clusters={gmm.n_components}")
plt.show()


##########################################################################
from sklearn.cluster import KMeans
# Use KMeans from sklearn
kmeans = KMeans(n_clusters=2)
kmeans.fit(points)
```

```
plt.scatter(points[:,0], points[:,1], c=kmeans.labels_, s=10)
plt.title(f"KMeans\nNumber of Clusters={kmeans.n_clusters}")
plt.show()
```







Fuzzy C-Means Based clustering

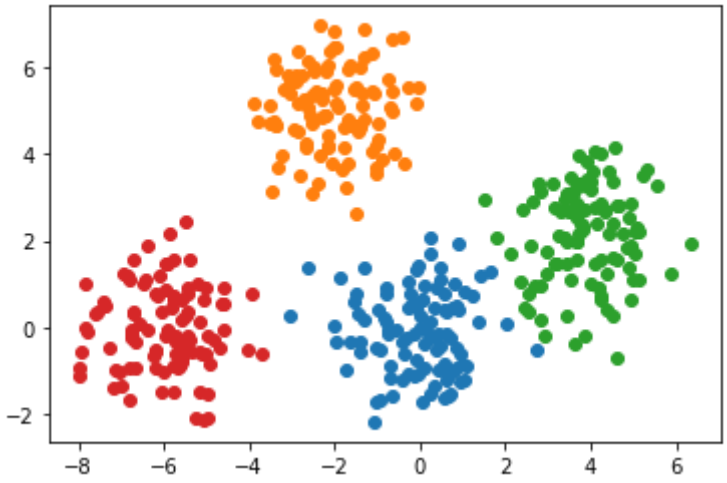
1. Randomly initialize the centroids and clusters K, and compute the probability that each data point xi is a member of a given cluster k, P(point xi has label k|xi, k).

2. Iteration: Recalculate the centroids of the clusters as the weighted centroid given the probabilities of membership of all data points xi:

$$\mu_k(n+1) = \frac{\sum_{x_i \in k} x_i * P(\mu_k \mid x_i)^b}{\sum_{x_i \in k} P(\mu_k \mid x_i)^b}$$

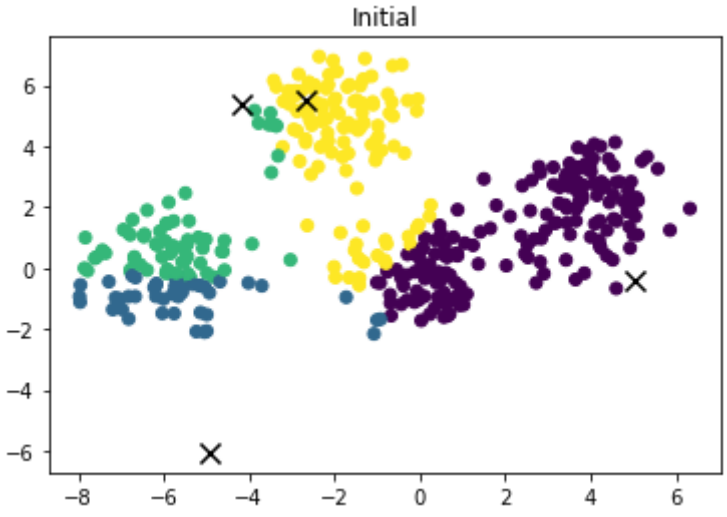
1. Implement it on the data for which Kmeans was implemented.

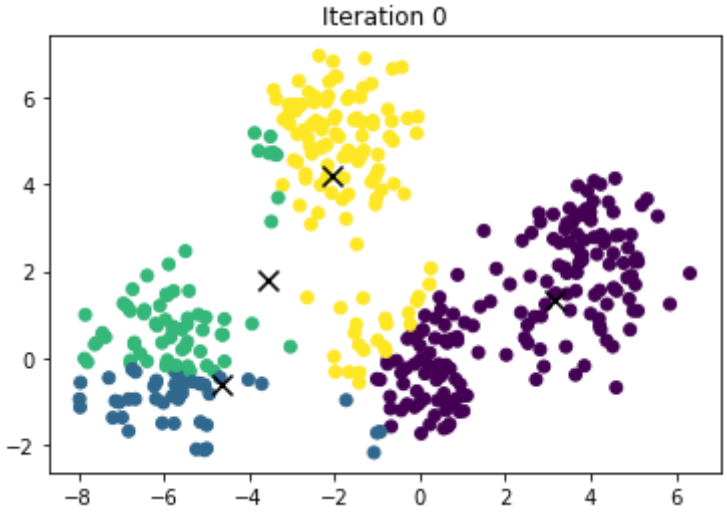
In [ ]:
```
# Means and variances
means = np.array([[0, 0], [-2, 5], [4, 2], [-6,0]])
variance = np.identity(2)


# Sample 100 points from each
samples = np.array([np.random.multivariate_normal(mean, variance, size=100)
for sample in samples:
    plt.scatter(sample[:, 0], sample[:, 1])
plt.show()

data = np.concatenate(samples, axis=0)
```

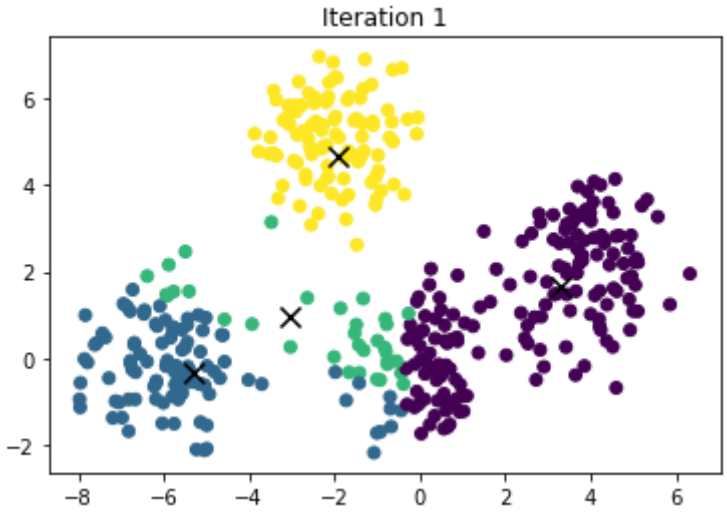


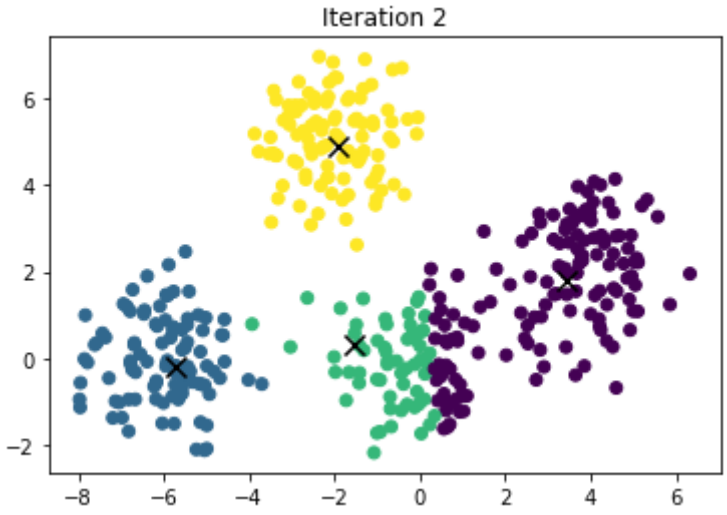
In [ ]:
```
class FuzzyC:
        # works for only 2D
        """ TODO :
                1 ) first find c centers randomly and calc dist matr and mer
                2 ) find new centers
                3 ) find dist matrix
                4 ) find new membership matrix
                5) Do it till convergence
        """
        def __init__(self, n_clusters, data):
                self.n = data.shape[0]
                self.num_clusters = n_clusters
                self.data = data
                self.mu = np.random.uniform(np.min(data), np.max(data), (n_c
                self.distMatrix = np.zeros((self.n, n_clusters))
                self.c = np.zeros((self.n, n_clusters))
                self.loss = []

        def updateDistMatrix(self):
                for i in range(self.n):
                        for j in range(self.num_clusters):
                                self.distMatrix[i][j] = np.linalg.norm(self.

        def updateC(self):
```

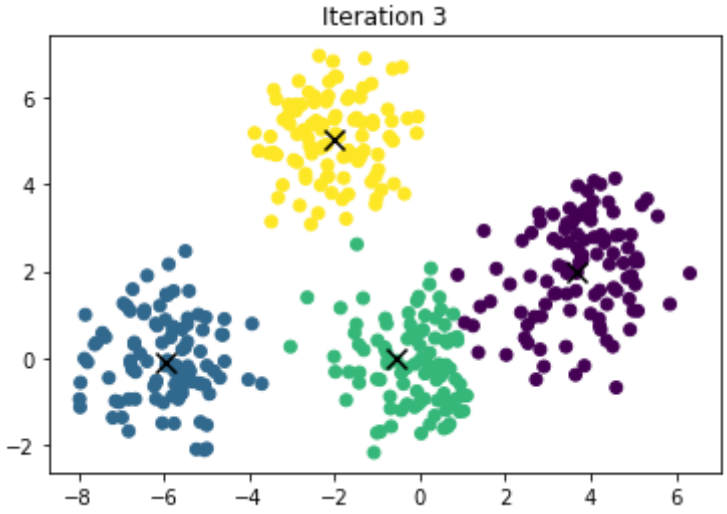
```
                for i in range(self.n):
                        for j in range(self.num_clusters):
                                sum = 0
                                for k in range(self.num_clusters):
                                        sum += self.distMatrix[i][j]**2 / se
                                self.c[i][j] = 1 / sum

        def updateMu(self):
                for i in range(self.num_clusters):
                        sum = [0,0]
                        for j in range(self.n):
                                sum += self.c[j][i]**2 * self.data[j]
                        self.mu[i] = sum / np.sum(self.c[:, i]**2)

        def plotClusters(self, title):
                plt.scatter(self.data[:, 0], self.data[:, 1], c=np.argmax(se
                plt.scatter(self.mu[:, 0], self.mu[:, 1], c='black', s=100,
                plt.title(title)
                plt.show()

        def calculateLoss(self):
                loss = 0
                for i in range(self.n):
                        for j in range(self.num_clusters):
                                loss += self.c[i][j]**2 * self.distMatrix[i]
                self.loss.append(loss)

        def plotLoss(self):
                plt.plot(self.loss)
                plt.title("Loss")
                plt.xlabel("Iterations")
                plt.show()

        def doFCM(self):
                self.updateDistMatrix()
                self.updateC()
                self.calculateLoss()
                self.plotClusters("Initial")
                i = 0
                while True:
                        self.updateDistMatrix()
                        self.updateC()
                        self.updateMu()
                        self.calculateLoss()
                        self.plotClusters(f"Iteration {i}")
                        i += 1
                        if self.loss[-2] - self.loss[-1] < 1e-5 and i > 5:
                                break

fcm = FuzzyC(4, data)
fcm.doFCM()
fcm.plotClusters("Final Clustering")
fcm.plotLoss()
```

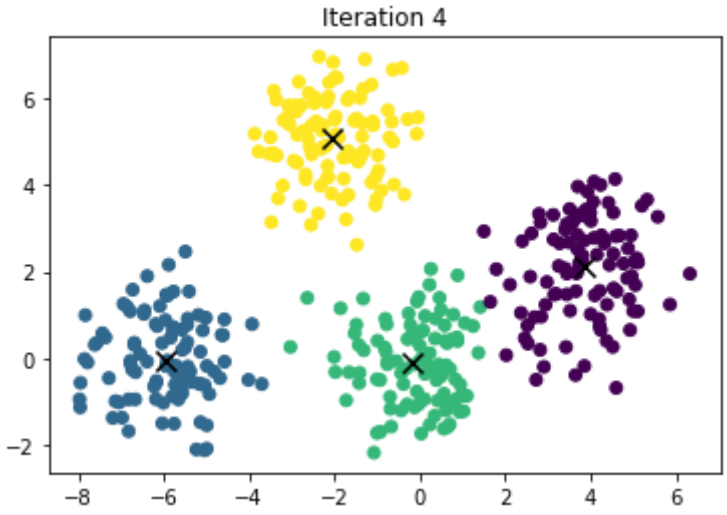

Initial
6
4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6

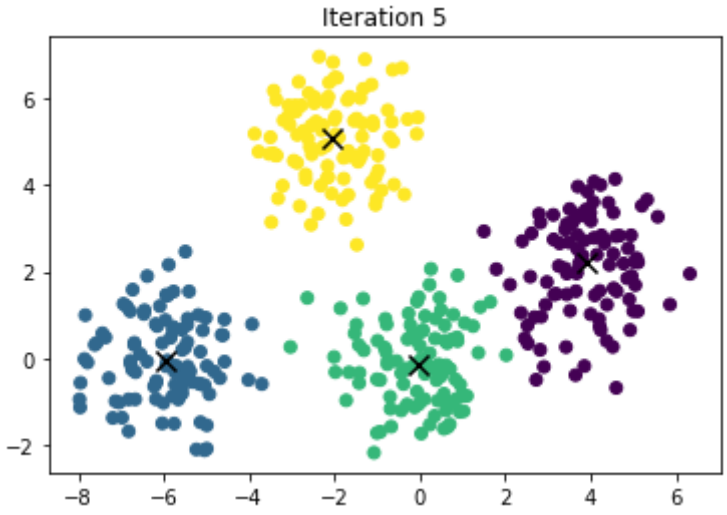

Iteration 0
6
4
2
0
−2
−8
−6
−4
−2
0
2
4
6

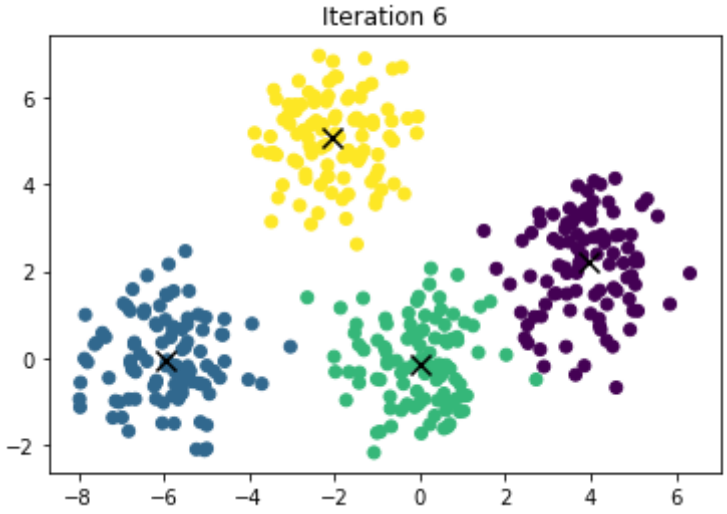

Iteration 1
6
4
2
0
−2
−8
−6
−4
−2
0
2
4
6

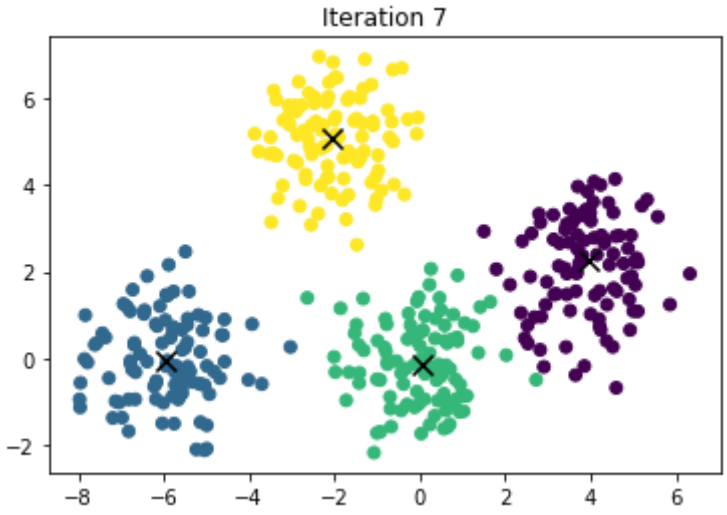

Iteration 2
6
4
2
0
−2
−8
−6
−4
−2
0
2
4
6

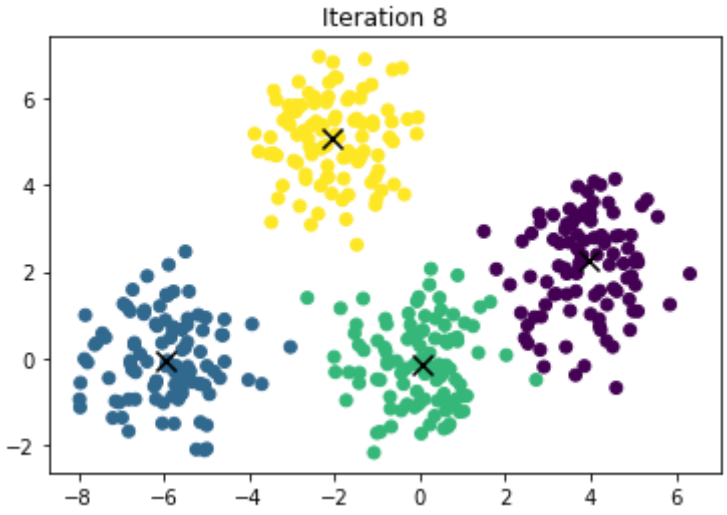

Iteration 3
6
4
2
0
−2
−8
−6
−4
−2
0
2
4
6

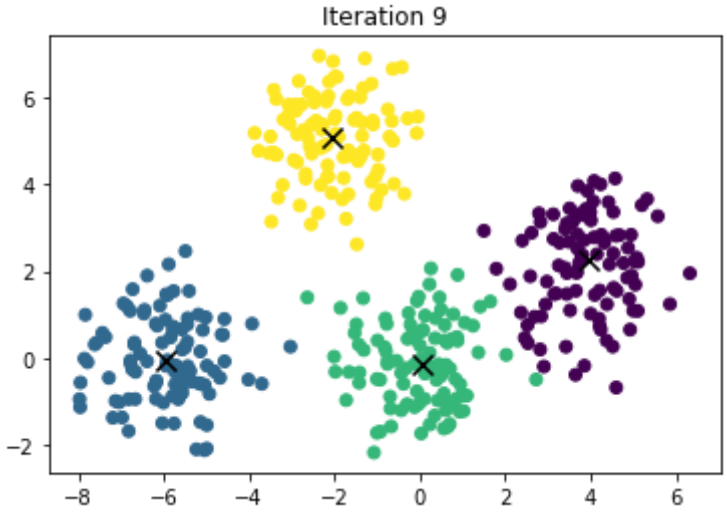

Iteration 4
6
4
2
0
−2
−8
−6
−4
−2
0
2
4
6

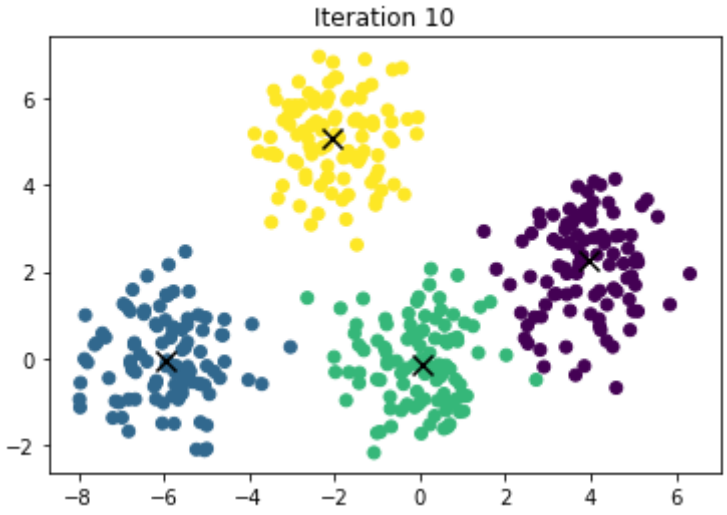

Iteration 5
6
4
2
0
−2
−8
−6
−4
−2
0
2
4
6

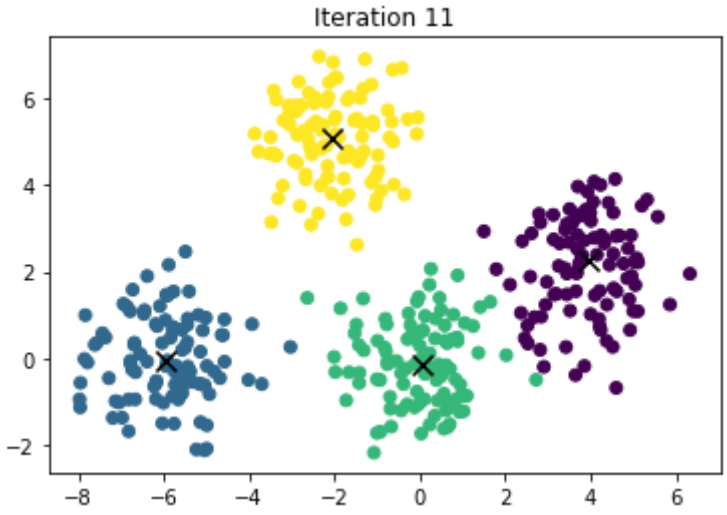

Iteration 6
6
4
2
0
−2
−8
−6
−4
−2
0
2
4
6

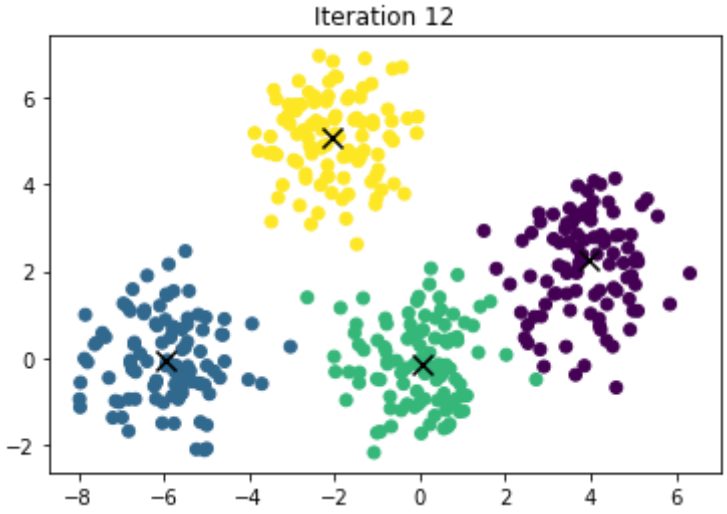

Iteration 7
6
4
2
0
−2
−8
−6
−4
−2
0
2
4
6

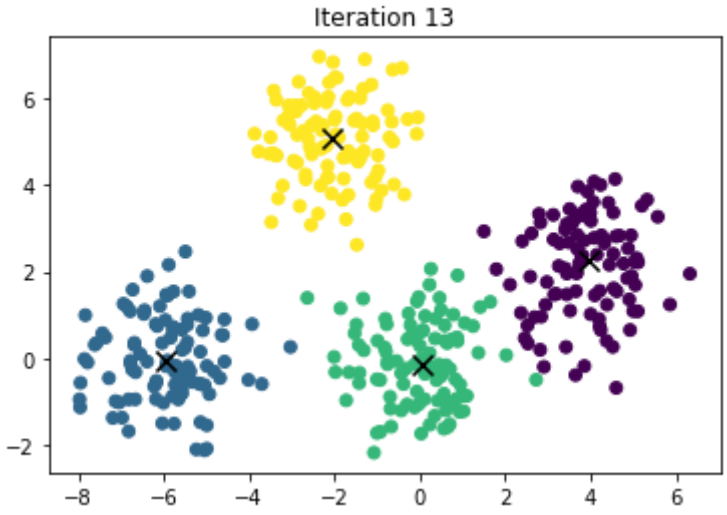

Iteration 8
6
4
2
0
−2
−8
−6
−4
−2
0
2
4
6

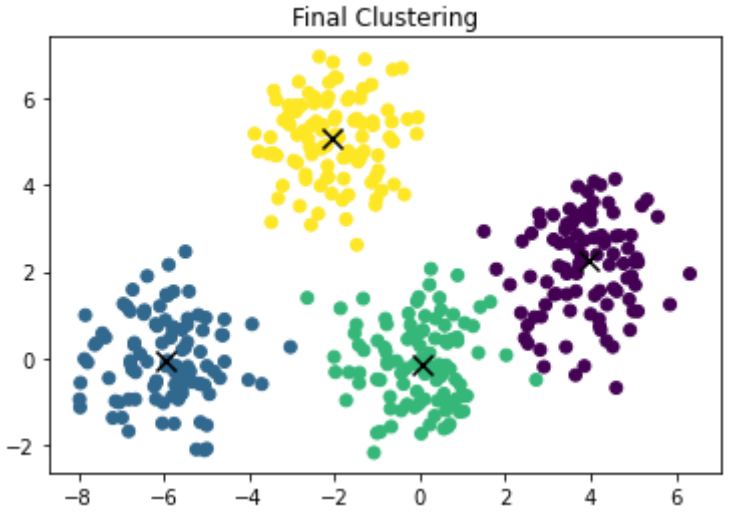

Iteration 9
6
4
2
0
−2
−8
−6
−4
−2
0
2
4
6

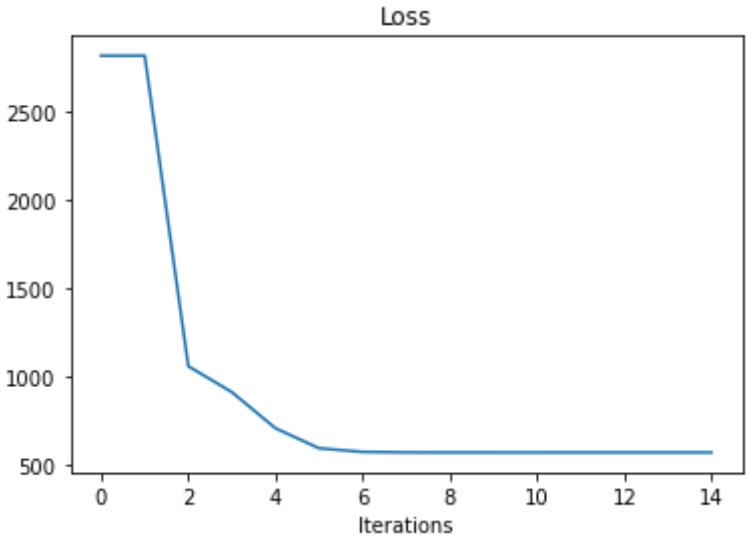

Iteration 10
6
4
2
0
−2
−8
−6
−4
−2
0
2
4
6


Iteration 11
6
4
2
0
−2
−8
−6
−4
−2
0
2
4
6


Iteration 12
6
4
2
0
−2
−8
−6
−4
−2
0
2
4
6


Iteration 13
6
4
2
0
−2
−8
−6
−4
−2
0
2
4
6





Hierarchical Clustering

Hierarchical clustering is an unsupervised clustering technique which groups together the unlabelled data of similar characteristics.

There are two types of hierarchical clustering:

- Agglomerative Clustering
- Divisive Clustering

Agglomerative Clustering:

In this type of hierarchical clustering all data set are considered as indivisual cluster and at every iterations clusters with similar characteristics are merged to give bigger clusters. This is repeated untill one single cluster is reached. It is also called bottem-top approach.

Agglomerative Clustering:

Lets start with some dummy example :

X=$[x_1, x_2, \ldots, x_5]$, with

$$x_1 = \begin{bmatrix}1\\1\end{bmatrix}, x_2 = \begin{bmatrix}2\\1\end{bmatrix}, x_3 = \begin{bmatrix}5\\4\end{bmatrix}, x_4 = \begin{bmatrix}6\\5\end{bmatrix}, x_5 = \begin{bmatrix}6.5\\6\end{bmatrix}$$

Steps to perform Agglomerative Clustering:

1. Compute Distance matrix ($N \times N$ matrix, where N number of vectors present in the dataset): $D(a,b) = ||x_a - x_b||_2$
2. Replace the diagonal elements with inf and find the index of the minimum element present in the distance matrix (suppose we get the location (l,k)).
3. Replace $x_{min(l,k)} = .5 \times [x_l + x_m]$ and delete $x_{max(l,m)}$ vector from X(i.e now $(N = N-1)$),

repeat from step 1 again untill all the vectors combined to a single cluster.

In [ ]:
```
from math import inf

class AggClustering:
        def __init__(self, data):
                self.data = data
                self.history = []

        def euclideanDist(self,x,y):
                dist = np.linalg.norm(x-y)
                if dist == 0:
                        return inf
                return dist

        def calculateDistMatrix(self):
                self.distMatrix = np.zeros((self.data.shape[0], self.data.sh
                for i in range(self.data.shape[0]):
                        for j in range(self.data.shape[0]):
                                self.distMatrix[i][j] = self.euclideanDist(s

        def iterate(self):
                self.calculateDistMatrix()
                i, j = np.unravel_index(np.argmin(self.distMatrix), self.dis
                a = self.data[i]
                b = self.data[j]
                self.data = np.delete(self.data, max(i,j), axis=0)
                self.data[min(i,j)] = (a+b)/2
                self.history.append([i+1,j+1])

        def doAggClustering(self):
                while len(self.data) > 1:
                        self.iterate()
                print(f"Merges are done in the following order:\n {self.hist
```

In [ ]:
```
X=np.array([[1,1],[2,1],[5,4],[6,5],[6.5,6]])
X=X.transpose()

aggClustering = AggClustering(X.T)
aggClustering.doAggClustering()

## validate from inbuilt Dendogram
import plotly.figure_factory as ff

lab=np.linspace(1,X.shape[1],X.shape[1])
fig = ff.create_dendrogram(X.T, labels=lab)
fig.update_layout(width=800, height=300)
fig.show()
```

```
Merges are done in the following order:
 [[1, 2], [3, 4], [2, 3], [1, 2]]
```

Clustering Algorithms on MNIST Digit dataset

Perform Kmeans and gmm clustering on MNIST dataset

1. Load MNIST data from the given images and labels
2. Consider any 2 classes

In [ ]:
```
%pip install idx2numpy
```

```
Defaulting to user installation because normal site-packages is not write
able
Requirement already satisfied: idx2numpy in /home/omp/.local/lib/python3.
10/site-packages (1.2.3)
Requirement already satisfied: numpy in /usr/lib/python3.10/site-packages
(from idx2numpy) (1.23.1)
Requirement already satisfied: six in /usr/lib/python3.10/site-packages
(from idx2numpy) (1.16.0)
Note: you may need to restart the kernel to use updated packages.
```

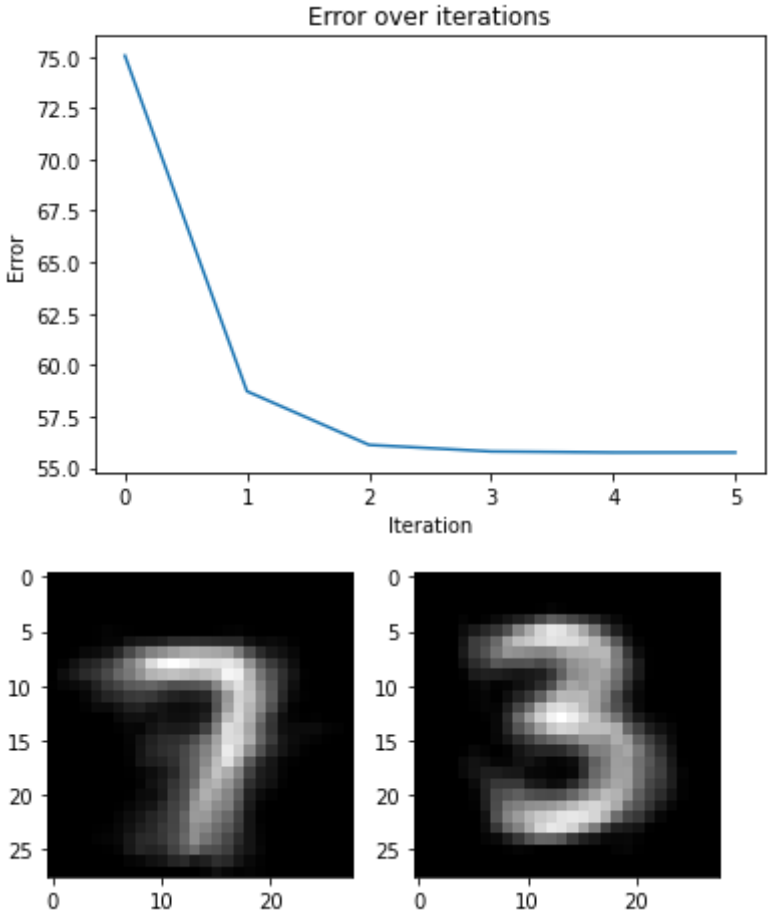
In [ ]:
```
import idx2numpy
from keras.utils import np_utils
img_path = "t10k-images-idx3-ubyte"
label_path = "t10k-labels-idx1-ubyte"

Images = idx2numpy.convert_from_file(img_path)
Labels = idx2numpy.convert_from_file(label_path)

# Let us choose 3s and 7s
three_idx = np.where(Labels == 3)
seven_idx = np.where(Labels == 7)

# Flatten the images
three_img = Images[three_idx].reshape(-1, 28*28)
seven_img = Images[seven_idx].reshape(-1, 28*28)

# Concatenate the images
data = np.concatenate([three_img[:50], seven_img[:50]], axis=0)
```

Use the K-means clustering algorithm from the last lab to form the clusters

In [ ]:
```
def kmeans(X, k, centroids):
        # Assign each sample to the closest centroid
        def assign_clusters(X, centroids):
                # Initialize empty list of clusters
                clusters = [[] for i in range(k)]
                # For each sample
                label_pred = []
                for sample in X:
                        # Find the centroid closest to the sample
                        closest_centroid = np.argmin(np.linalg.norm(sample -
                        # Add the sample to the closest centroid
                        clusters[closest_centroid].append(sample)
                        label_pred.append(closest_centroid)
```

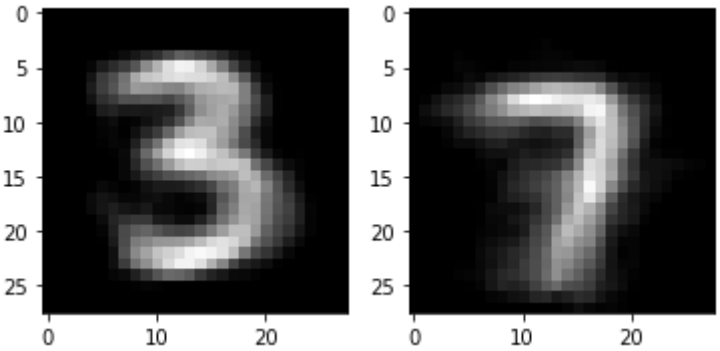
```
                return clusters, label_pred

        # Recompute centroids based on new assignments
        def recompute_centroids(clusters):
                # Initialize empty list of new centroids
                new_centroids = []
                # For each cluster
                for cluster in clusters:
                        # Compute the mean of the cluster
                        mean = np.mean(cluster, axis=0)
                        # Add the new centroid to the list of new centroids
                        new_centroids.append(mean)
                return new_centroids

        # Error function
        def compute_error(clusters, centroids):
                # Initialize error as 0
                error = 0
                # For each cluster
                for idx, cluster in enumerate(clusters):
                        error += np.sum(np.linalg.norm(cluster - centroids[i
                error /= 400
                return error

        def plot_clusters(clusters, centroids, iteration, error):
                # Plot the clusters
                for cluster in clusters:
                        cluster = np.array(cluster)
                        plt.scatter(cluster[:, 0], cluster[:, 1])
                # Plot the centroids
                centroids = np.array(centroids)
                plt.title(f"Iteration {iteration} | Error: {error}")
                plt.scatter(centroids[:, 0], centroids[:, 1], marker='X', c=
                plt.show()

        # K-means algorithm
        error = []
        for i in range(1000):
                # Assign each sample to the closest centroid
                clusters, label_pred = assign_clusters(X, centroids)
                # Recompute centroids based on new assignments
                if not i == 0:
                        centroids = recompute_centroids(clusters)
                # Compute error
                error.append(compute_error(clusters, centroids))
                # Plot the clusters
                # plot_clusters(clusters, centroids, i, error[-1])
                # If error is low break
                if i > 1:
                        if abs(error[-2] - error[-1]) < 1e-10:
                                break

        plt.plot(error)
        plt.title("Error over iterations")
        plt.xlabel("Iteration")
        plt.ylabel("Error")
        plt.show()

        return centroids

centroids = kmeans(data, 2, data[np.random.choice(data.shape[0], 2, replace=

im1 = centroids[0].reshape(28, 28)
```

```
im2 = centroids[1].reshape(28, 28)

plt.subplot(1, 2, 1)
plt.imshow(im1, cmap='gray')
plt.subplot(1, 2, 2)
plt.imshow(im2, cmap='gray')
plt.show()
```



Use the GMM clustering algorithm from the last lab to form the clusters

In [ ]:
```
from sklearn.mixture import GaussianMixture

gmm = GaussianMixture(n_components=2, init_params='kmeans')
gmm.fit(data)

im1 = gmm.means_[0].reshape(28, 28)
im2 = gmm.means_[1].reshape(28, 28)

plt.subplot(1, 2, 1)
plt.imshow(im1, cmap='gray')
plt.subplot(1, 2, 2)
plt.imshow(im2, cmap='gray')
plt.show()
```


0
5
10
15
20
25
0
10
20
0
5
10
15
20
25
0
10
20