Name : Abhik Kumar Dey
ASU Id : 1216907406
Project : Unsupervised Learning (K-Means Algorithm)

1. Introduction

In this project we will be implementing the k-means algorithm, a supervised learning method, to classify data. K-means is a Clustering Algorithm which aims to find homogeneous subgroups such that objects in the same group(cluster) are similar to each other.

We are given the data set which contains a set of 2-D points. Our goal is to implement the k-means algorithm using 2 strategies for choosing initial clusters and then calculate the objective function for them and plot the respective graphs. The objective function for k-means is given as follows:

$$\sum_{i=1}^{k} \sum_{\mathbf{x} \in D_i} ||\mathbf{x} - \boldsymbol{\mu}_i||^2 \ )$$

where X is the data set belonging to the cluster μ_i.

Strategy 1 : Select the initial Clusters randomly

Strategy 2: Select the first Cluster randomly and then choose the i^{th} cluster such that the distance between the i^{th} cluster and all the other $(i-1)^{th}$ cluster is maximal.

Goal:

a. Implement strategy 1, calculate and plot the objective function

b. Implement strategy 2, calculate and plot the objective function

Language: Python

Constraint: Use of sklearn package is not allowed

Data Source: AllSamples.mat is provided

2. Data and Methodology

We are given a set of 2D points in the file “AllSamples.mat” with which we have to implement the k-means algorithm and plot the objective function. Below explained are the two strategies in details:

2.1 Strategy 1: Choose the initial Cluster randomly

Data Extraction:

The data is extracted from the file “AllSamples.mat” and stored in a variable.

```
def extractData():
    """Extract data from Matlab"""
    ###############################################################################
    # Function - extractData()                                                    #
    # Parameters - NULL                                                           #
    # Functionality - Extract data from the matlab file                           #
    # Author - Abhik Dey                                                          #
    ###############################################################################
    mat = spio.loadmat("AllSamples.mat", squeeze_me = True)
    data = mat['AllSamples'] #Taking values in variable from mat

    return data
```

Centroid Initialization Algorithm:

The points are selected at random from the given dataset and are assigned as centroid. This depends on k. For different values of k, there will be k different clusters.

```
def initialize_centroids(k,data_set):
    """Initializing centroids at random"""
    ###############################################################################
    # Function - initialize_centroids()                                           #
    # Parameters - k - Number of clusters, data_set - data from given matlab file #
    # Functionality - Initialize centroids at random                              #
    # Author - Abhik Dey                                                          #
    ###############################################################################

    random_index = np.random.choice(a.shape[0], k, replace = False)
    centroids = data_set[random_index]
    #print (centroids)

    return centroids
```

Implement k-means algorithm:

Loop k for k = 2,3,4,5,6,7,8,9,10:

Initialize centroid by calling **initialize_centroids** function described above

Loop until the centroids have stabilized:

Loop for all the data sets:

a. Calculate the Euclidean Distance of the data set with the centroids

b. Select the minimum distance calculated and assign the point belonging to the particular centroid

c. Calculate the mean of the points falling under cluster of centroids to estimate the new Centroids

d. Compare old and new Centroid for convergence

End Loop

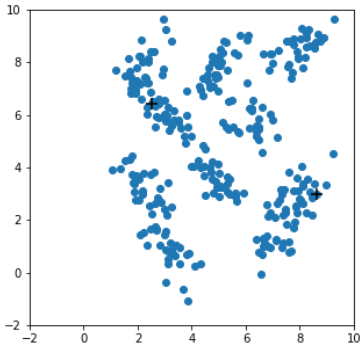
End Loop

End Loop

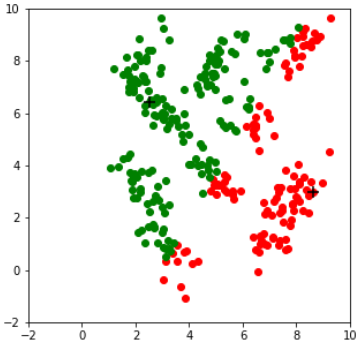
Plot the graph for each iteration and k:

This helps to understand how the centroid changes

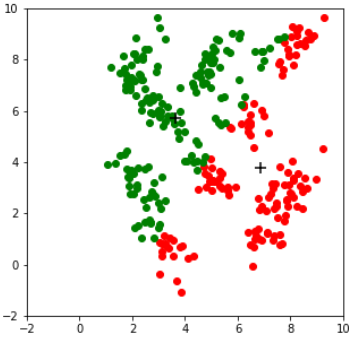
we will display plots for **k = 2**



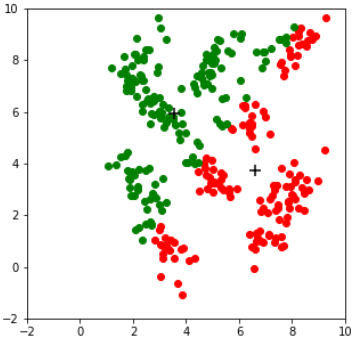
Initial Cluster



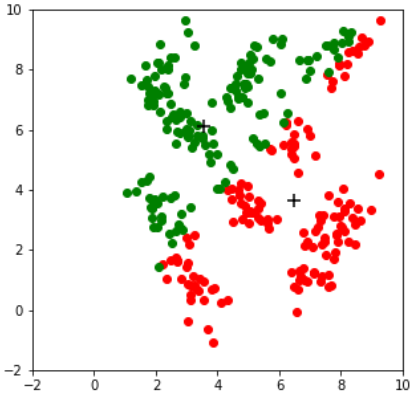
Iteration 1



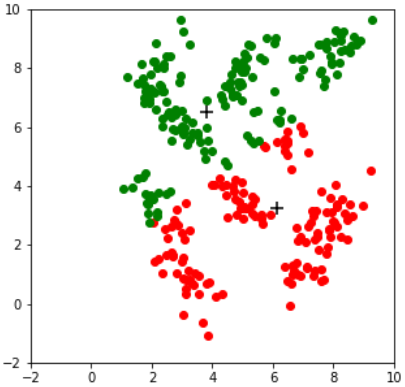
Iteration 2



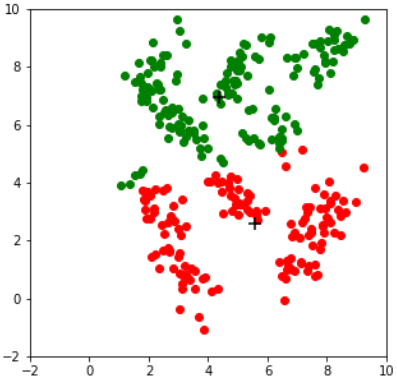
Iteration 3



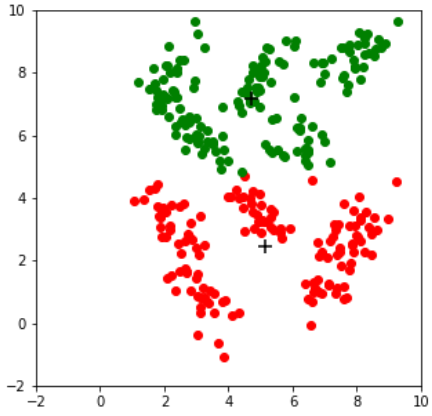
Iteration 4



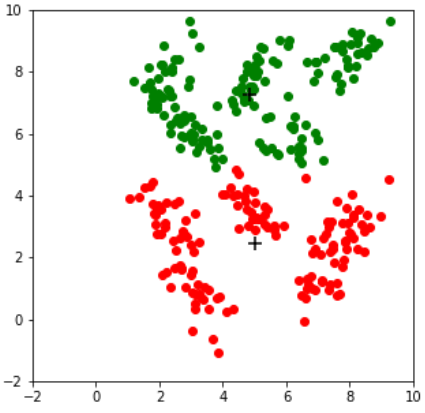
Iteration 5



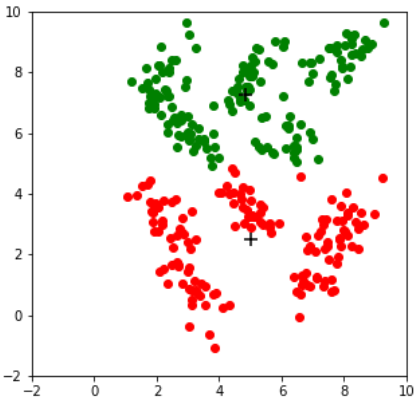
Iteration 6



Iteration 7



Iteration 8



Iteration 9

These steps are repeated for k = 3,4,5,6,7,8,9,10 and the graphs are displayed.

Calculate Objective Function(J) for all values of k:

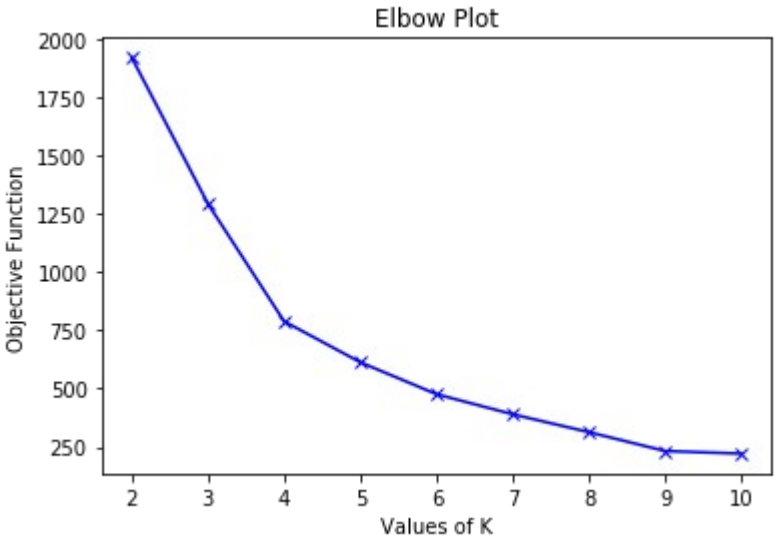
Objective function is the square of the Euclidean distance of all the points belonging to a particular cluster, summing over all data set and number of cluster. It has been implemented as below:

```
def calObjectiveFunction(data_set,centroids):
    ###############################################################################
    # Function - calEuclideanDist()                                               #
    # Parameters - data_set - data from matlab, centroids - k Clusters            #
    # Functionality - Calculate Objective Function                                #
    # Author - Abhik Dey                                                          #
    ###############################################################################
    sum = 0
    obj_val = []

    for ds in data_set:
        obj_val.append(((np.linalg.norm((ds-centroids), axis = 0)**2)))
    return np.sum(obj_val)
```

```
#Calculating the objective function and store value to obj_plot to plot against k (Elbow Plot) - @akd
sum_of_clusters = 0
for i in range(k):
    points = np.asarray([a[j] for j in range(len(a)) if C[j]==i])
    val = calObjectiveFunction(points,centroids[i])
    sum_of_clusters += val
print ("Sum of Clusters as whole",sum_of_clusters)

obj_plot.append(sum_of_clusters)
print (obj_plot)
```

Plot Objective Function with respect to k (Elbow Plot):

```
#Plot the Elbow Plot (X-> k, Y-> Objective Function) - @akd
plt.plot(k_val, obj_plot, 'bx-')
plt.xlabel('Values of K')
plt.ylabel('Objective Function')
plt.title('Elbow Plot')
plt.show()
```



2.2 Strategy 2: Random initialization of first centroid, choose i^{th} cluster such that distance between the i^{th} cluster and all the other $(i-1)^{th}$ cluster is maximal.

Data Extraction:

The data is extracted from the file “AllSamples.mat” and stored in a variable.

```
def extractData():
    """Extract data from Matlab"""
    ################################################################################
    # Function - extractData()                                                     #
    # Parameters - NULL                                                            #
    # Functionality - Extract data from the matlab file                            #
    # Author - Abhik Dey                                                           #
    ################################################################################

    mat = spio.loadmat("AllSamples.mat", squeeze_me = True)
    data = mat['AllSamples'] #Taking values in variable from mat

    return data
```

Centroid Initialization Algorithm:

a. Initialize the first centroid as below:

```
#randomly initialize 1st centroid
random_index = np.random.choice(data_set.shape[0], 1, replace = False)
centroids.append(data_set[random_index])
index_list.append(random_index[0])
```

b. Loop for range(0 to (k-1)):

i. Calculate the distance of the data sets from the centroid and store the distance values in array

ii. Calculate the mean of the distance from the centroid to the points

iii. Get the index for maximum average value

iv. Check if the index has been considered previously (To avoid repetition of centroids)

If found, get the next largest average value from the average array

v. Add the index to the index list

c. Get the centroid list from the data set using the index list

Implement k-means algorithm:

Loop k for k = 2,3,4,5,6,7,8,9,10:

Initialize centroid by calling **initialize_centroids** function described above

Loop until the centroids have stabilized:

Loop for all the data sets:

a. Calculate the Euclidean Distance of the data set with the centroids

b. Select the minimum distance calculated and assign the point belonging to the particular centroid

c. Calculate the mean of the points falling under cluster of centroids to estimate the new Centroids

d. Compare old and new Centroid for convergence

End Loop

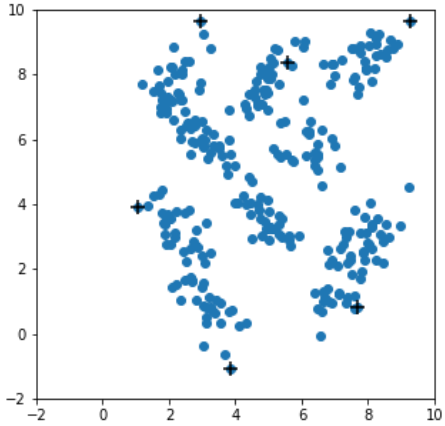
End Loop

End Loop

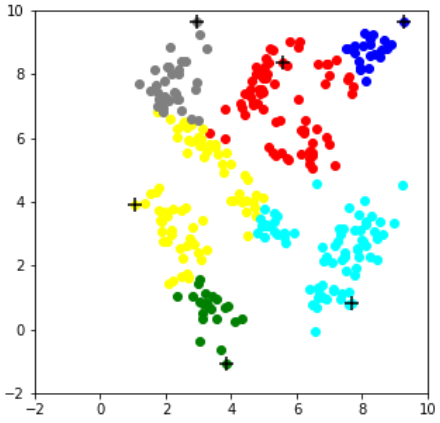
Plot the graph for each iteration and k:

This helps to understand how the centroid changes.

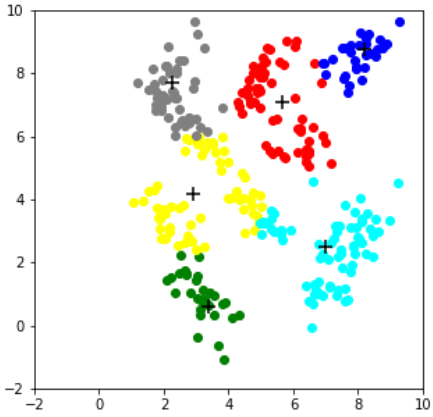
We will display the plots for **k = 6** here:



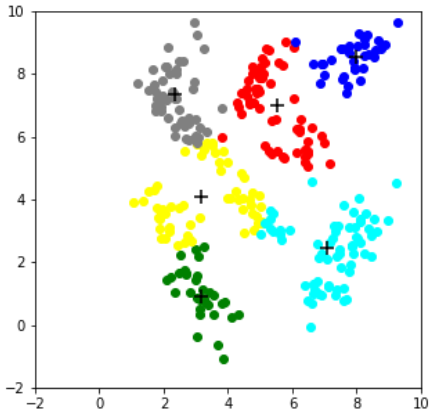
Initial Cluster



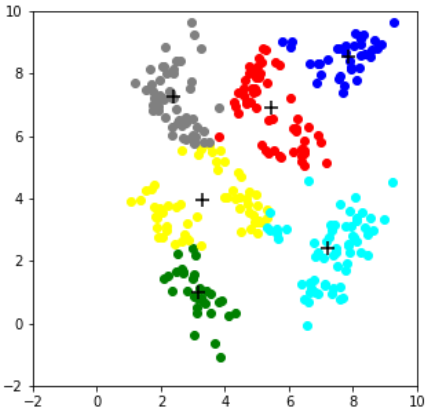
Iteration 1



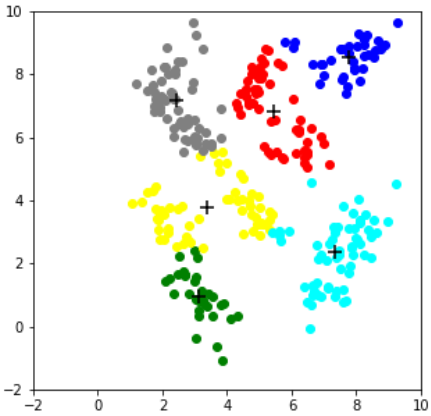
Iteration 2



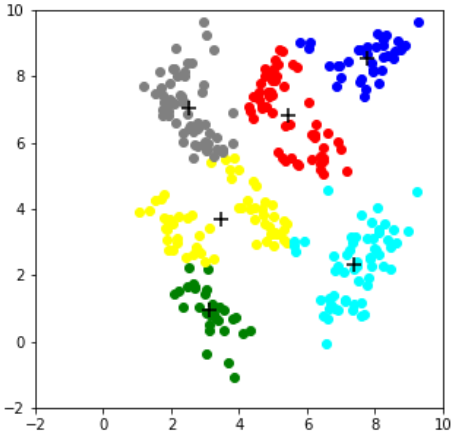
Iteration 3



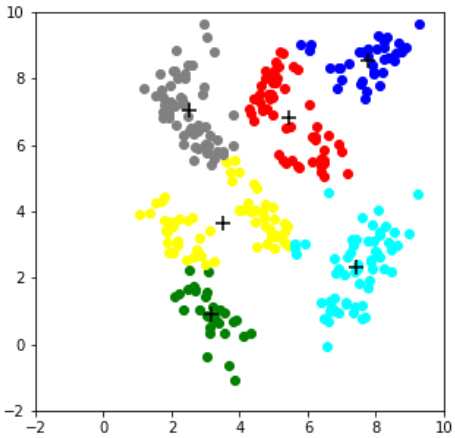
Iteration 4



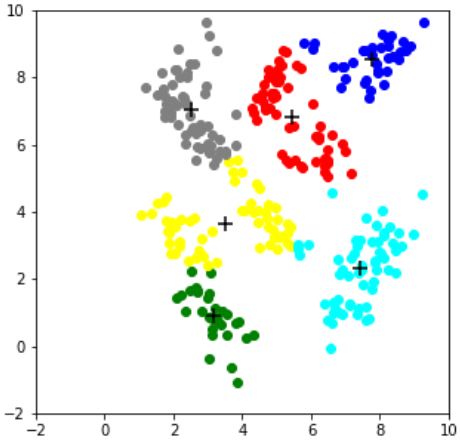
Iteration 5



Iteration 6



Iteration 7



Iteration 8

These steps are repeated for k = 2,3,4,5,7,8,9,10 and the graphs are displayed.

Calculate Objective Function(J) for all values of k:

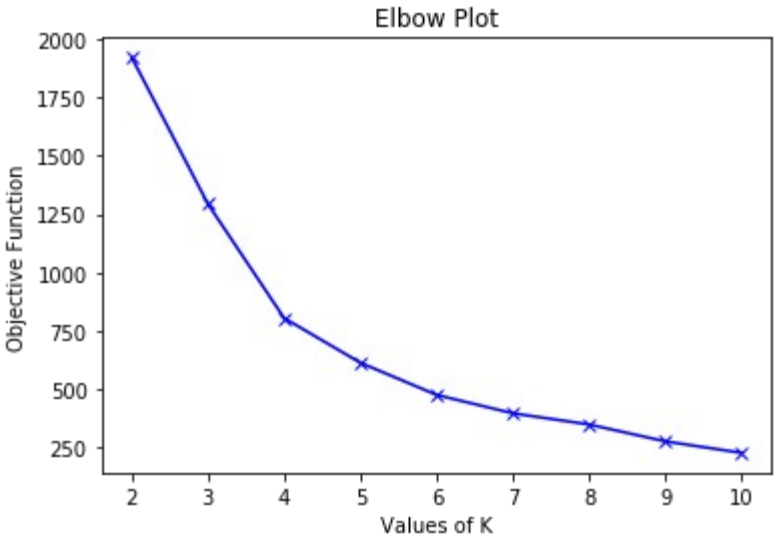
Objective function is the square of the Euclidean distance of all the points belonging to a particular cluster, summing over all data set and number of cluster. It has been implemented as below:

```
def calObjectiveFunction(data_set,centroids):
    ###############################################################################
    # Function - calEuclideanDist()                                               #
    # Parameters - data_set - data from matlab, centroids - k Clusters            #
    # Functionality - Calculate Objective Function                                #
    # Author - Abhik Dey                                                          #
    ###############################################################################
    sum = 0
    obj_val = []

    for ds in data_set:
        obj_val.append(((np.linalg.norm((ds-centroids), axis = 0)**2)))
    return np.sum(obj_val)
```

```
#Calculating the objective function and store value to obj_plot to plot against k (Elbow Plot) - @akd
sum_of_clusters = 0
for i in range(k):
    points = np.asarray([a[j] for j in range(len(a)) if C[j]==i])
    val = calObjectiveFunction(points,centroids[i])
    sum_of_clusters += val
print ("Sum of Clusters as whole",sum_of_clusters)

obj_plot.append(sum_of_clusters)
print (obj_plot)
```

Plot Objective Function with respect to k (Elbow Plot):

```
#Plot the Elbow Plot (X-> k, Y-> Objective Function) - @akd
plt.plot(k_val, obj_plot, 'bx-')
plt.xlabel('Values of K')
plt.ylabel('Objective Function')
plt.title('Elbow Plot')
plt.show()
```



3. Conclusion

From the above implementation we can say that using k-means algorithm, we can efficiently classify data (feature) according to the subgroups such that data in the same group(cluster) have similar characteristics. It is very easy to implement and has efficient performance even with large data sets. Because of its flexibility, it can be used with any type of data-type. Also it is very easy to interpret. To determine the optimal number of clusters into which a data may be clustered, we have used the Elbow Plot. From the plot, we can infer that for higher the number of k, the Objective function decreases. Initially the Objective Function decreases drastically(elbow) and then in linear fashion. We can conclude that at k=4, we get the optimal number of clusters.