Cmpe 481 - Data Analysis and Visualisation

Assignment 1

Onur Dilsiz

2019400036

Contents

Introduction

To begin with, I needed to have a suitable dataset for clustering which I can generate using this line of code, which uses a function from the scikit-learn library in Python.

```
X, y = make_blobs(n_samples=1000, centers=8, n_features=2)
```

While implementing k-means algorithm, to manage the functions in a more convenient way, I needed to separate some functions.

plotclusters(X, labels, centers, k): This function provides visualization. It plots clusters and their centers.

assignment(df, centers): Used for assignment of points to the nearest cluster.

replace(df, centers): Replaces the cluster centers based on the assigned data points.

calculate_objective_function(X, labels, centers): Calculates the objective function.

compareCenters(oldCenters, newCenters): Compares old centers with new centers to check if they are closer than a specified epsilon value.

measureDistance(point1, point2): Calculates the distance between two points.

findNearestCluster(point, centers, clusterNo): Determines the nearest cluster to a given point, excluding its own cluster.

silhouetteCoefficient(X, centers, labels): Calculates the silhouette coefficient for given data, which is a indicator of the quality of clustering

K-means function:

In the beginning of K-means function, several initializations are required. In my implementation, k means function gets 2 parameters, which k,which is the wanted number of clusters, and X, which is the data itself. Then, using the length of the X, I generated a labels list, which is the same size with the X and shows the cluster of any point by giving an integer between 0 and k-1 to each point. Generation of this array is done by using a method from the numpy library.

```
labels = np.random.randint(0, k, size=n)  # Randomly assign
each data point to one of k clusters
```

Subsequently, I tried to initialize the center points' x-axis randomly between minimum point at x-axis and maximum point at x-axis. Similarly, y-axis was a random value minimum point at y-axis and maximum point at y-axis.

```
minX = min(df[0]) ## min value of X  from dataset
maxX = max(df[0]) ## max value of X from dataset
minY = min(df[1]) ## min value of Y from dataset
maxY = max(df[1]) ## max value of Y from dataset
for i in range(k):
   centres[i] = [random.uniform(minX, maxX),
random.uniform(minY, maxY)]
```

However, assigning these values in this way created some bugs. It sometimes choses a point that is further than any points to cluster centers, and since it is not a part of data at the next iteration, this point is removed from cluster centers. Then, it breaks down the functionality of k- means algorithm. Therefore, I decided to use a different approach, which assigns each center to a random data point.

```
for i in range(k):  ##randomly select k points as centers
   random_index = random.randint(0, len(df) - 1)  # Generate a
random index within the range of data points
```

```
    random_point = df[random_index]
    centres[i] = random_point
```

After this initialization process, there is a while loop for k-means to operate. In while loop, first I stored centers to a variable named oldCentres as to compare it with new centers after the operations. Subsequently, I assigned labels according to random generated centers. Then using new labels, I replaced the centers via the functions explained in the introduction. After these assignments, objective function is calculated and appended to the list generated for storing it in order to observe the progress of the my k-means algorithm.

For the finishing condition of the while loop, I first tried this line of code.

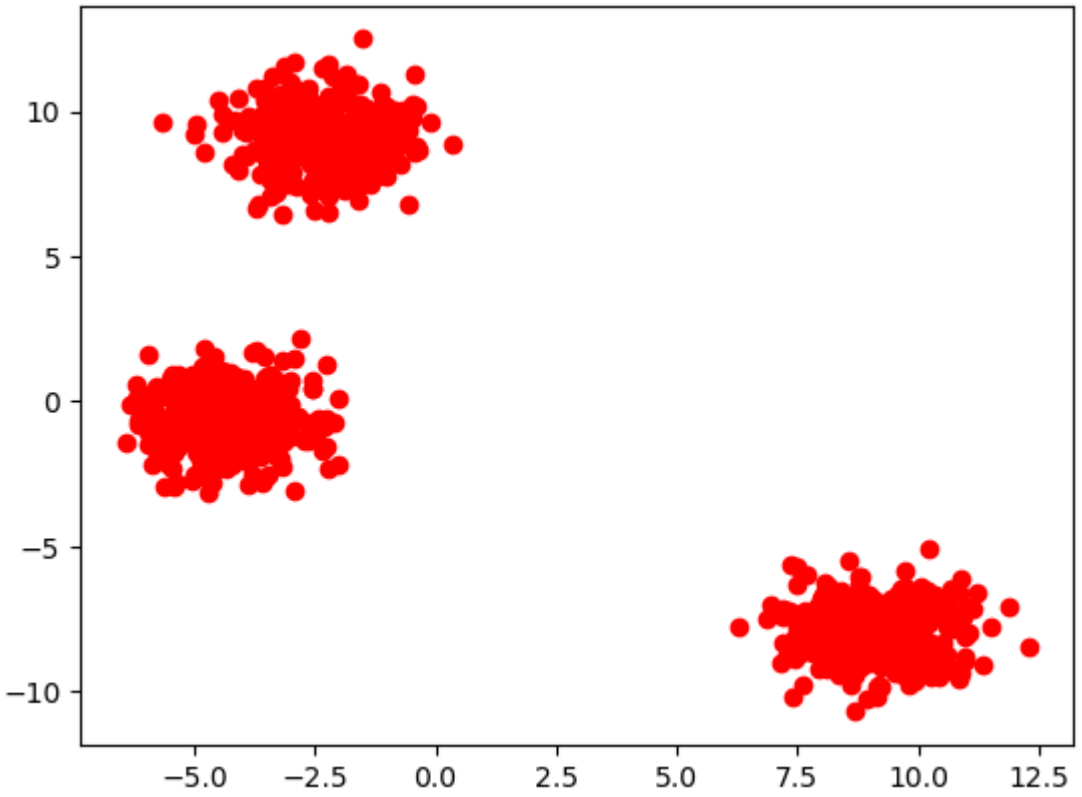
```
if np.array_equal(oldLabels, labels) and
np.array_equal(oldCentres, centres):
```

However, then I noticed that using an epsilon value to compare these centers is more sensible than this way because when the data sample is so big, this condition may fail to be true. Subsequently, I also figured out that comparison of oldLabels and labels is not that necessary by trial and error. Therefore, I used the code below, which uses an epsilon value to compare centers. While deciding the epsilon value, I tried some values like 0.1, 0.01,0.001, etc. Obviously, the bigger ones are not suitable for being an epsilon since the data centers may just change 0.1 or 0.01 which still has a huge impact on the next iteration. Therefore, I chose the value of 1e-6.

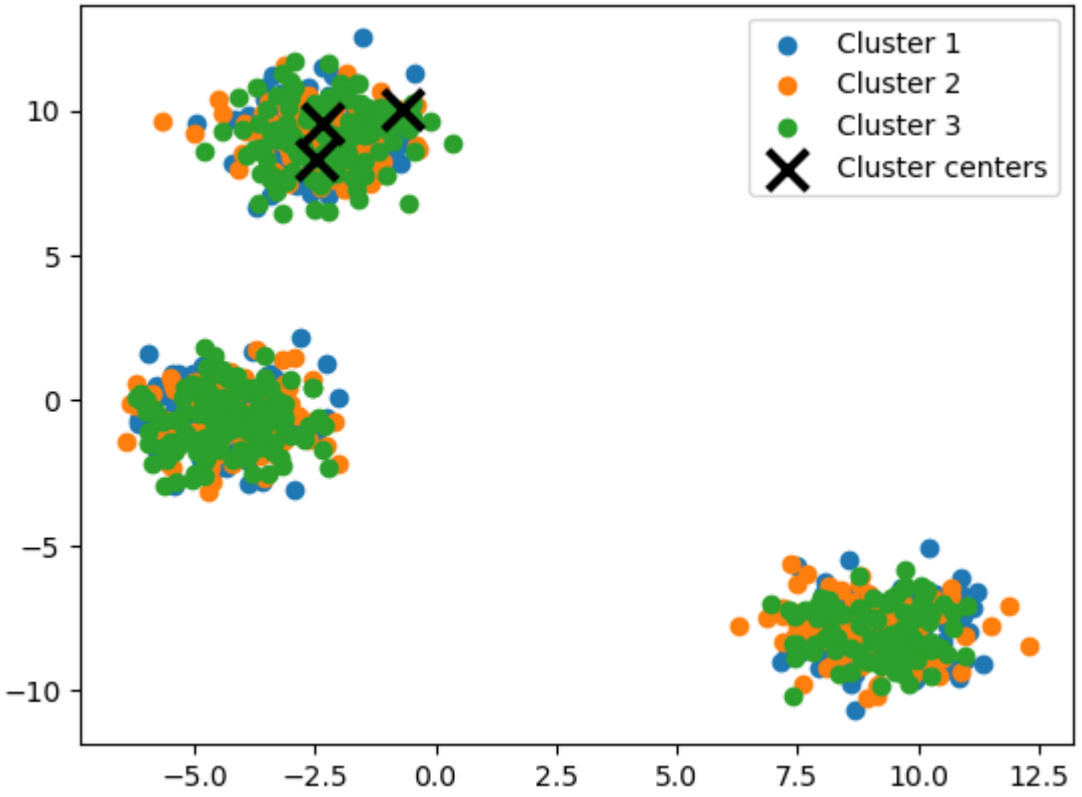
```
if (compareCentres(oldCentres, centres)):
```

After that, the final part of k-means includes plotting clusters via plotclusters function and plotting objective function. At the last, it calculates silhouette coefficient which the function returns by silhouetteCoefficient function that I wrote.

Example for k=3

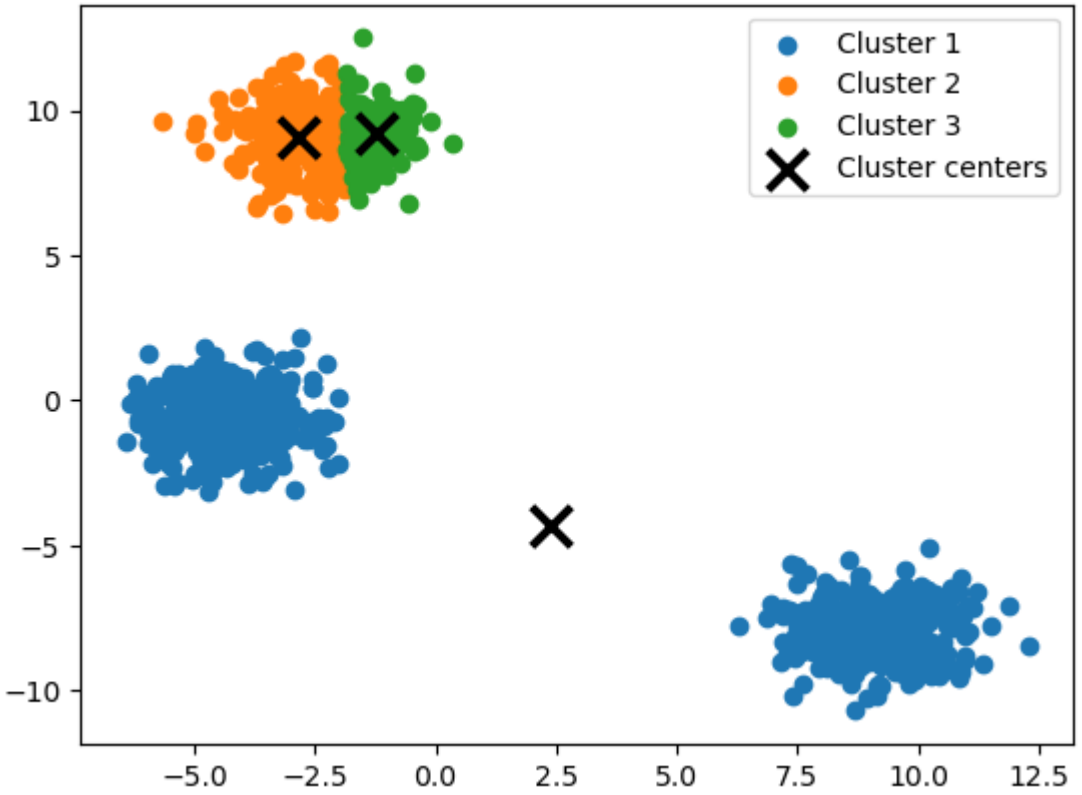


Initial data points for k=3



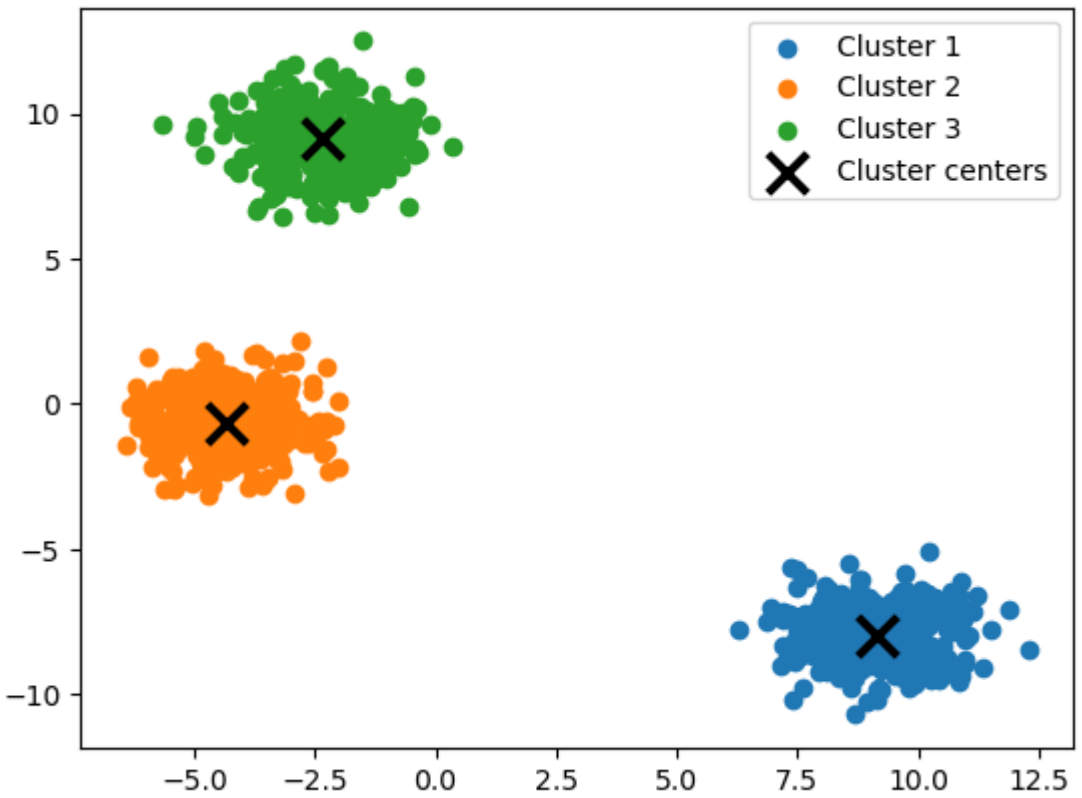
Random Initialization of Centers and Cluster Assignment

Iteration 1

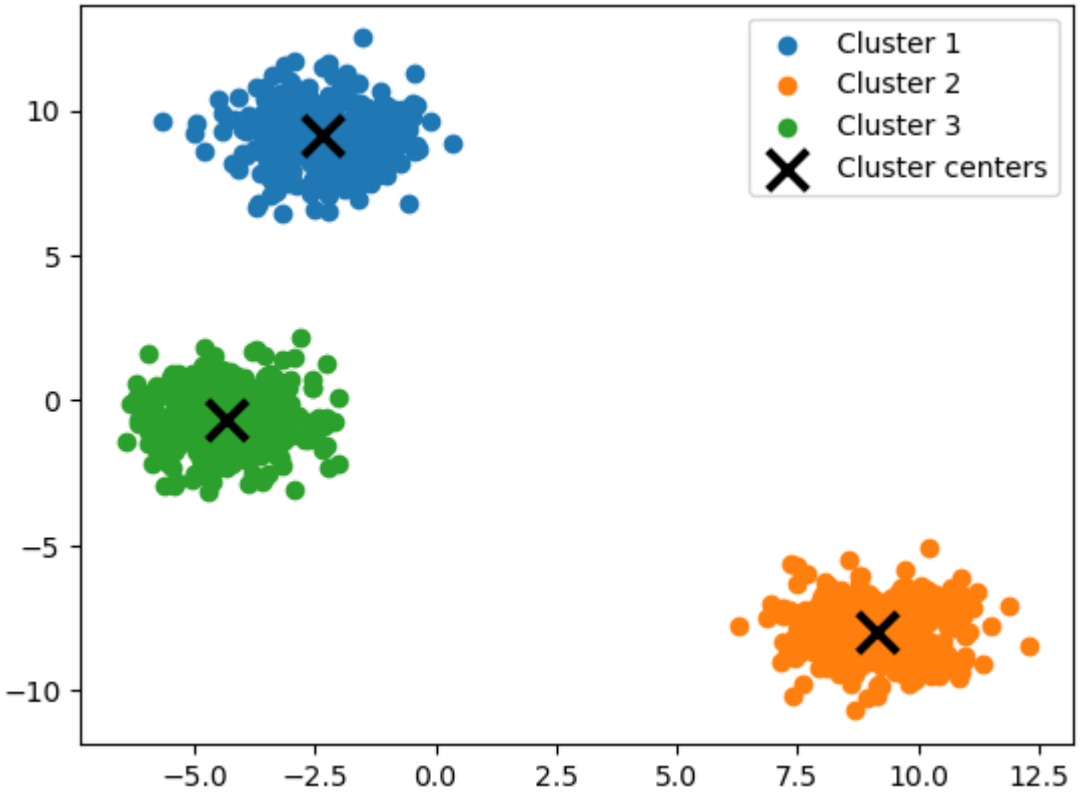

Cluster 1
Cluster 2
Cluster 3
Cluster centers
10
5
0
−5
−10
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
12.5

Iteration 2

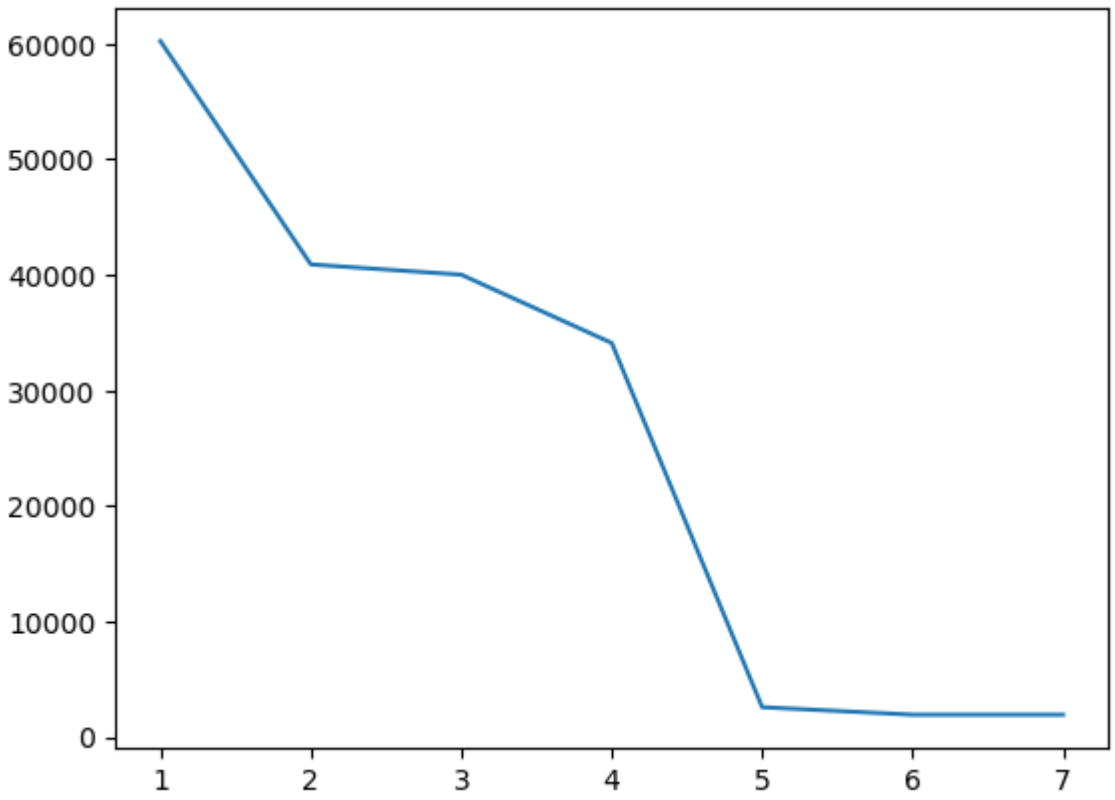
Iteration 3



Final plot

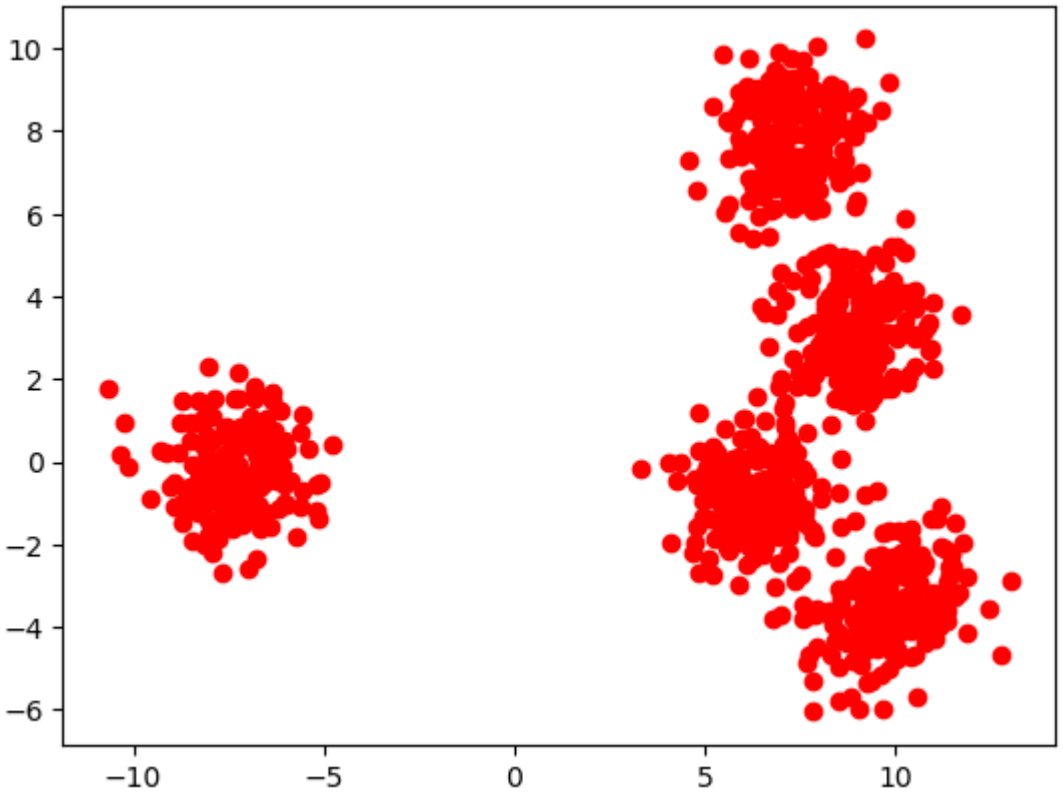


final plot for scikit-learn

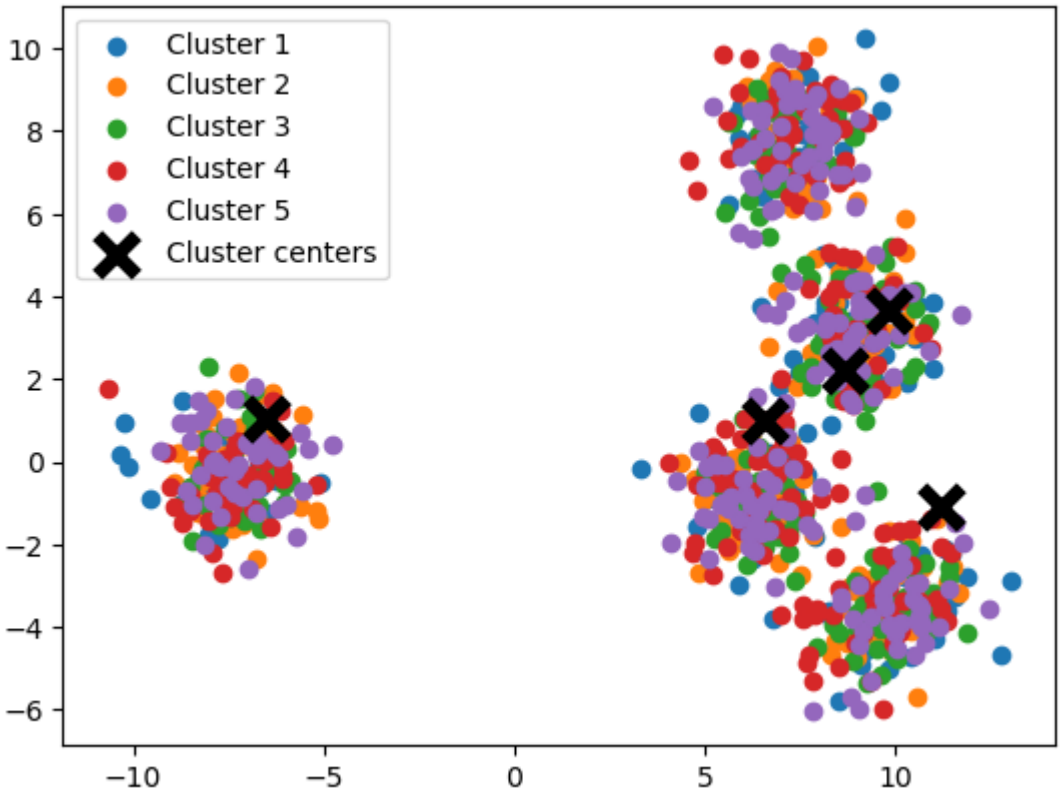


objective function plot - x-axis is the iteration count

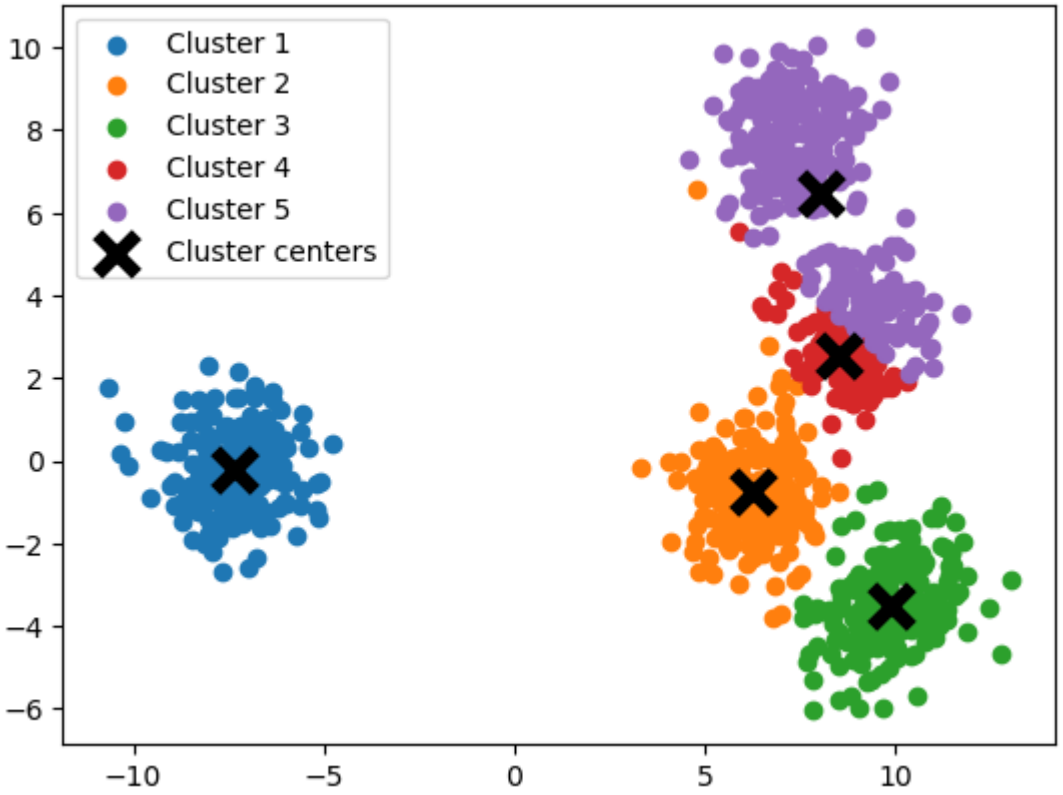
Example for k=5



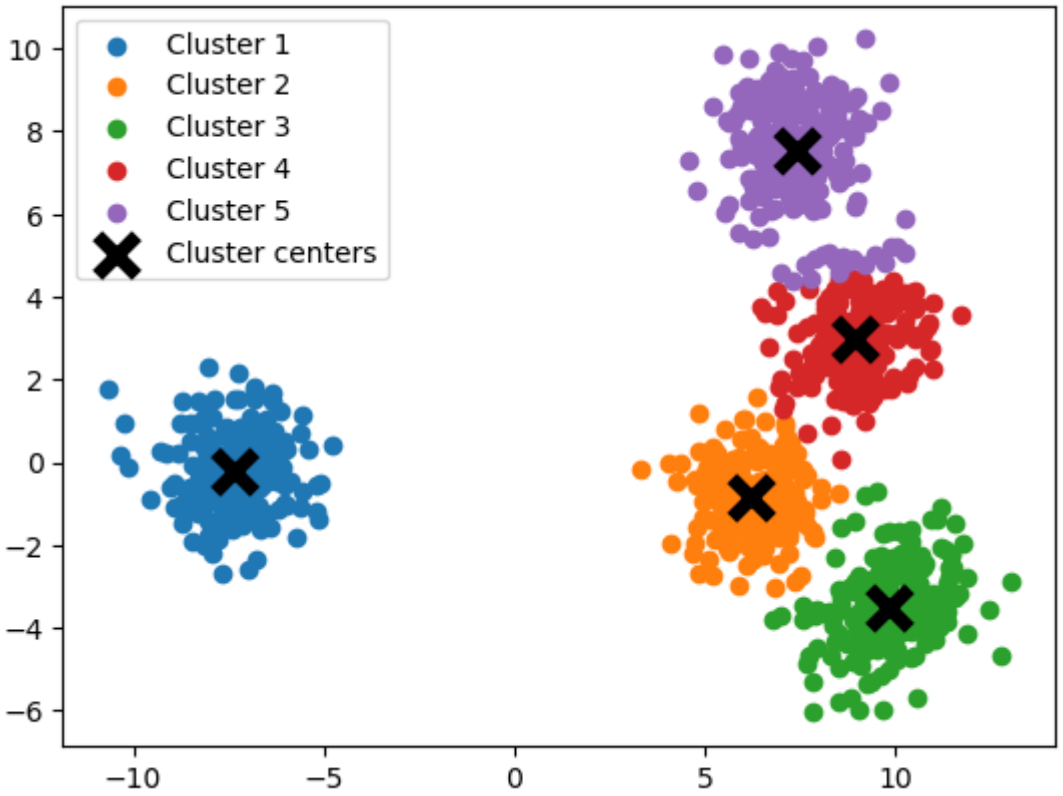
Initial data points



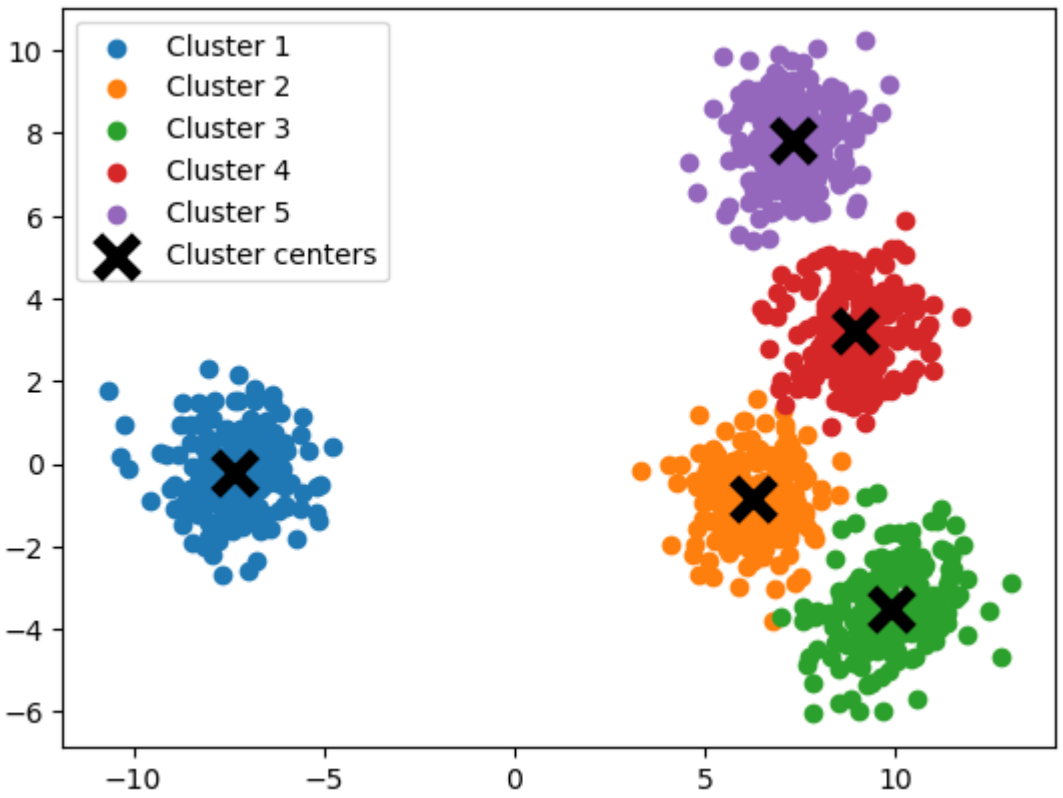
Initial data and random assign of centers and clusters



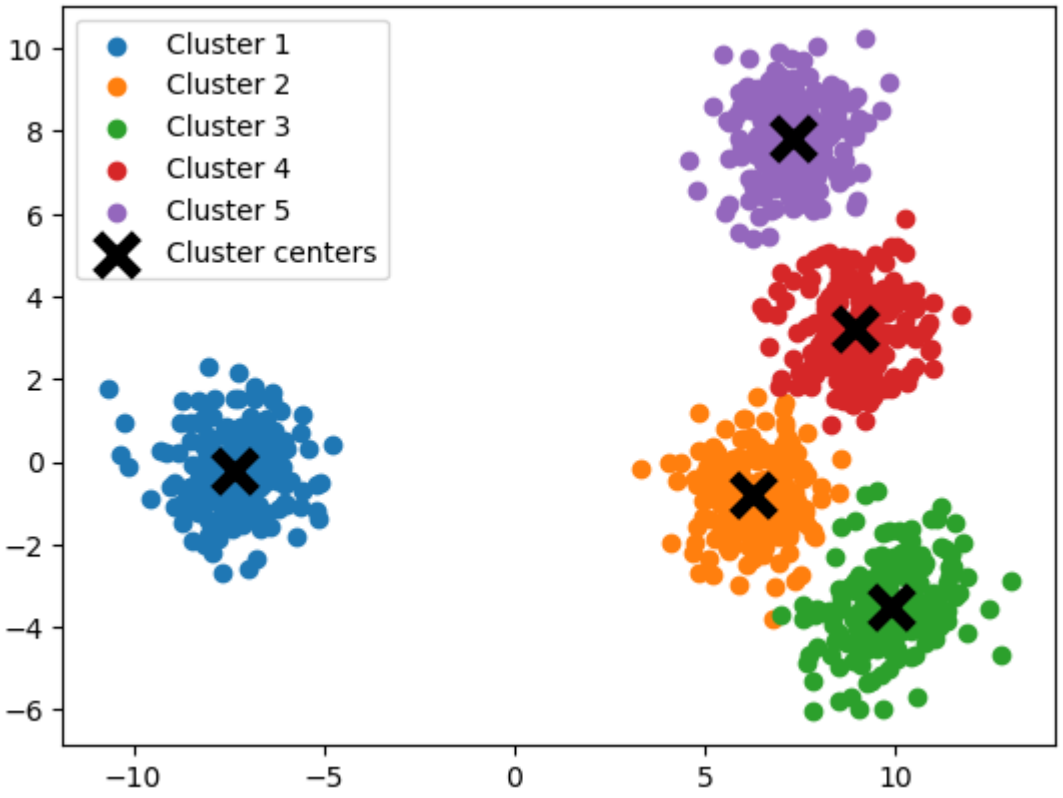
Iteration 1



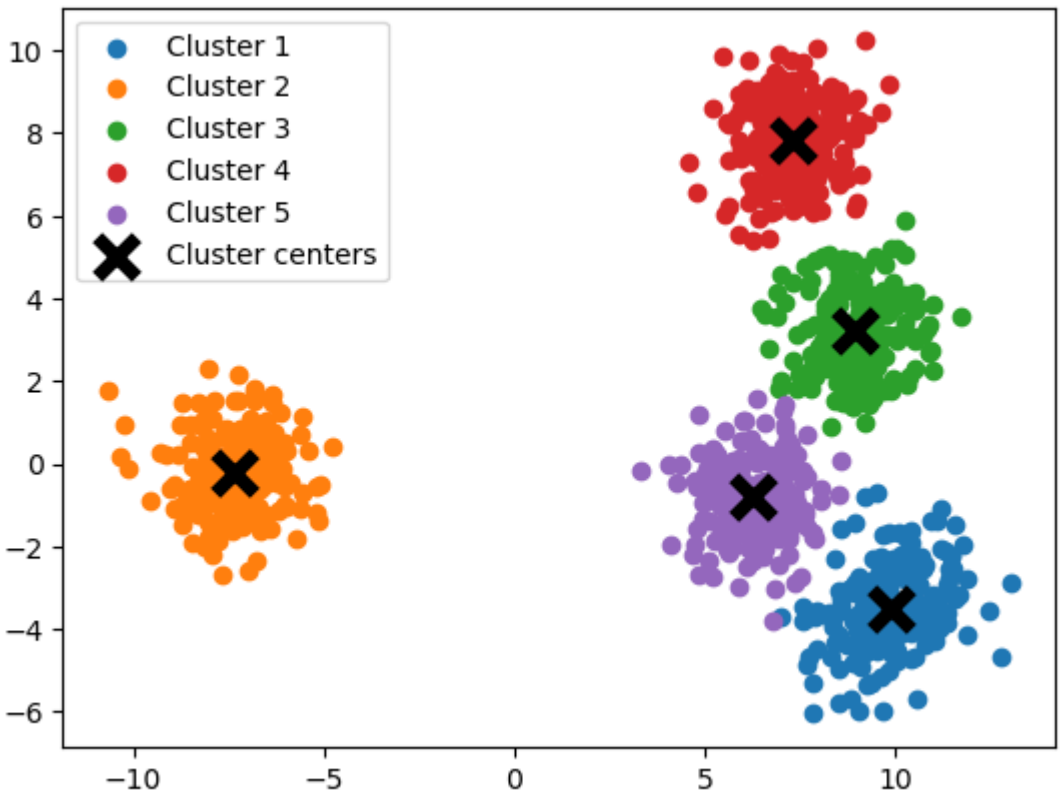
Iteration 2



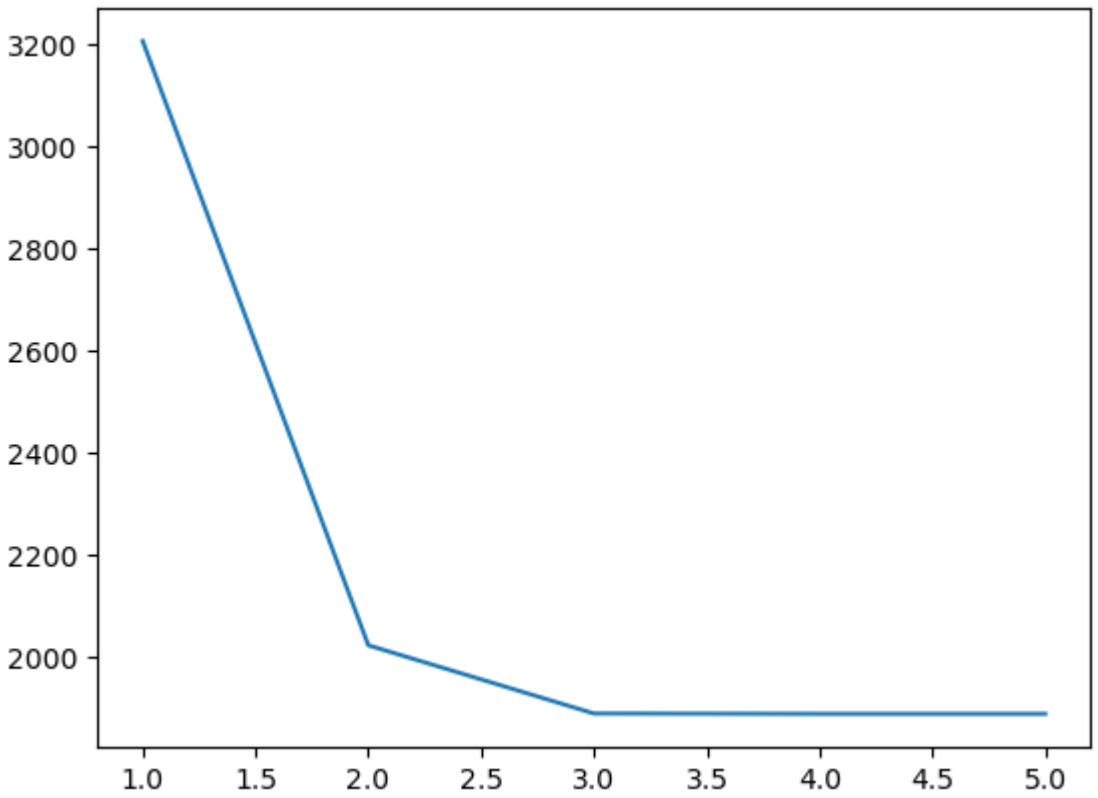
Iteration 3



Final for my implementation



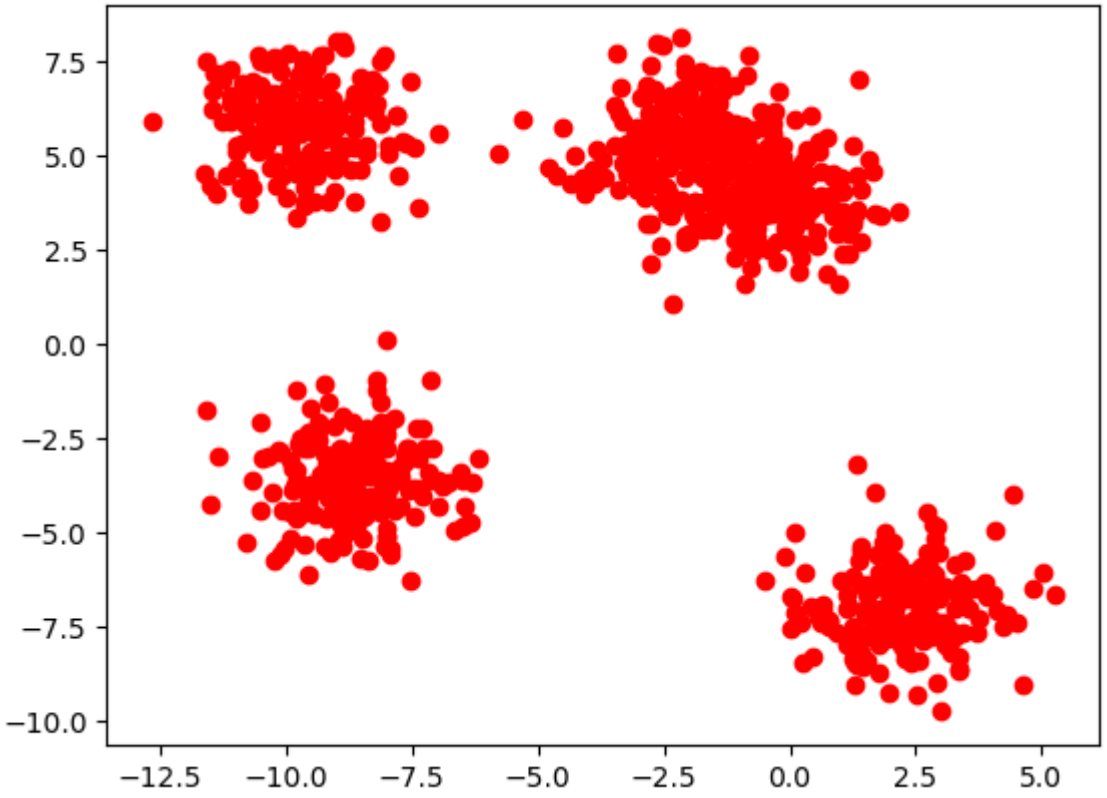
Scikit k-means final version for same k



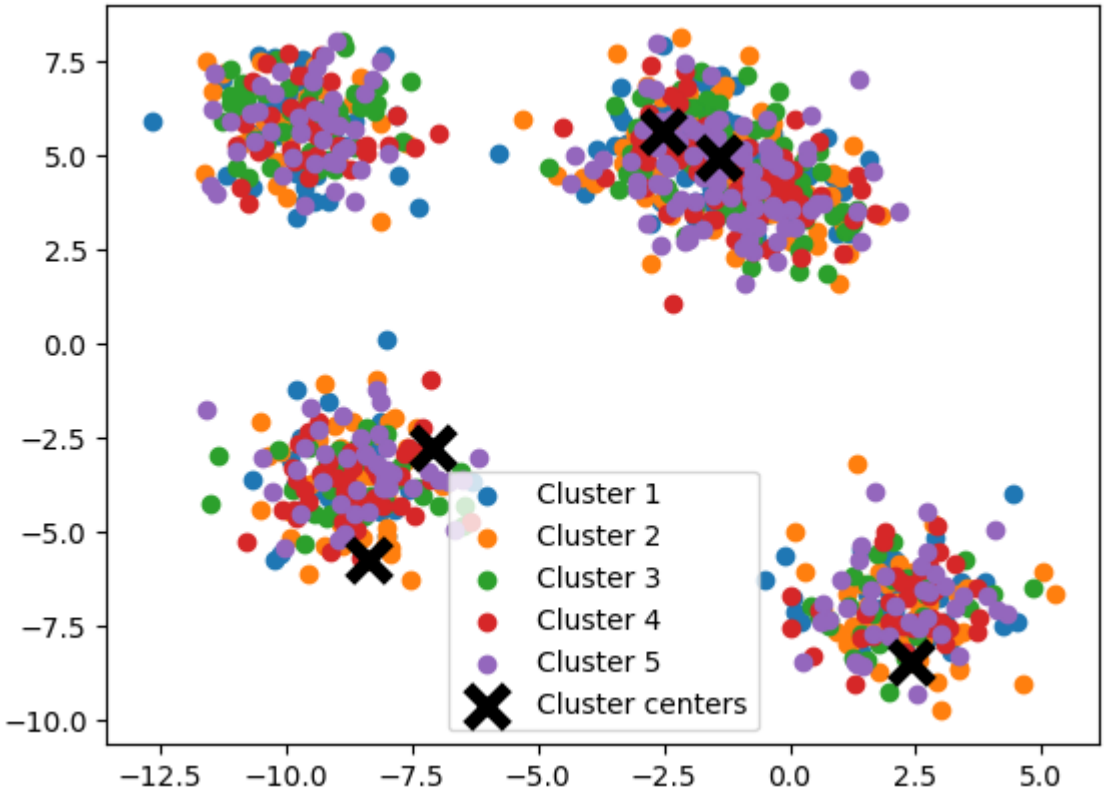
Objective function plot for k=5

Comparison

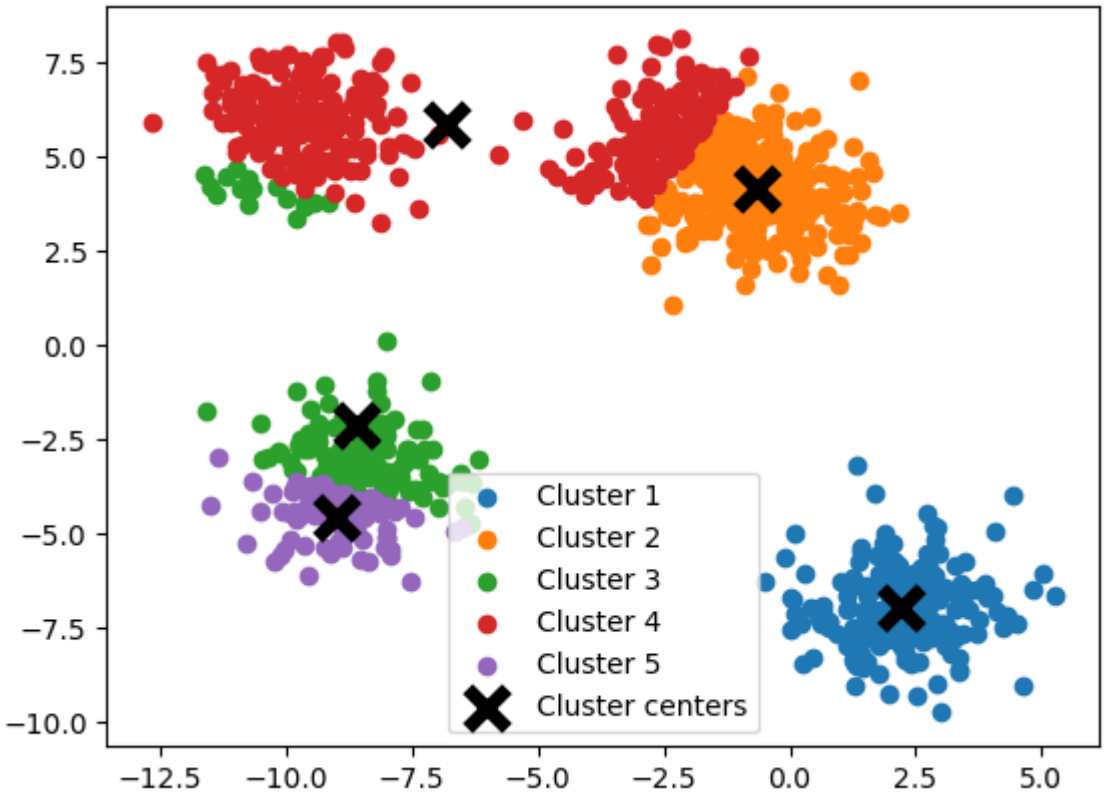
With different datasets, my implementation gives different results from the output obtained by the scikit-learn library. We can observe some differences in the example below.



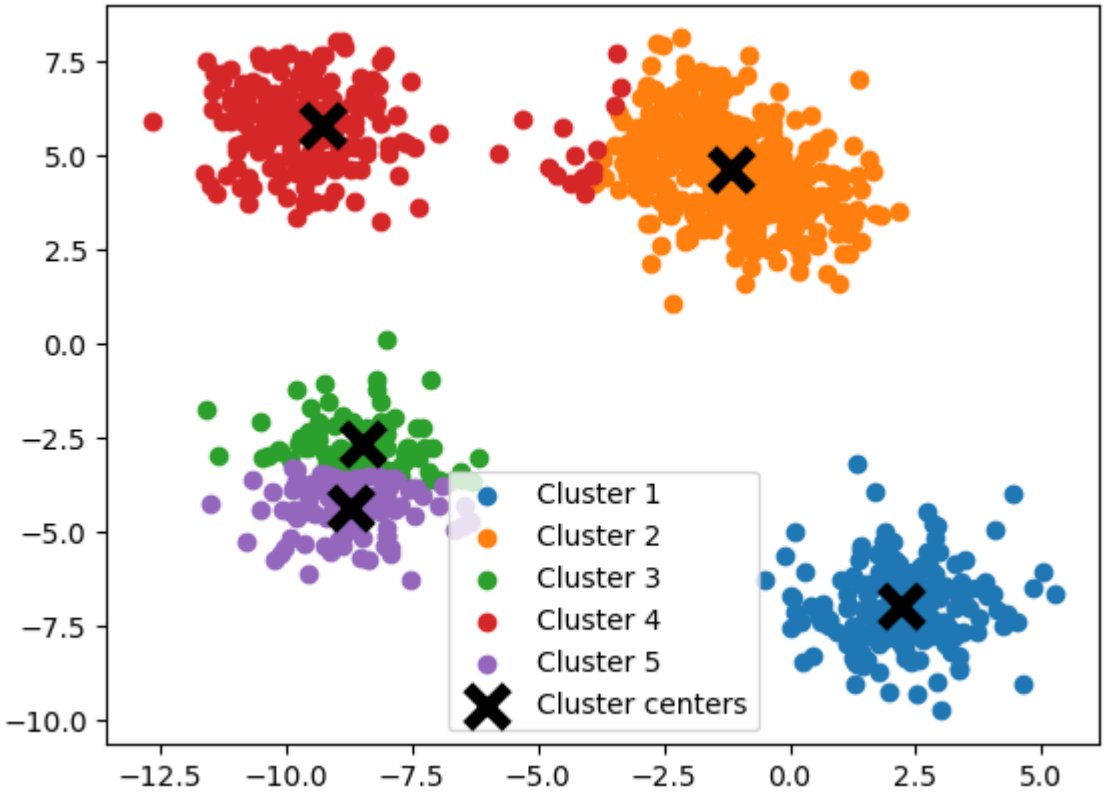
Initial data



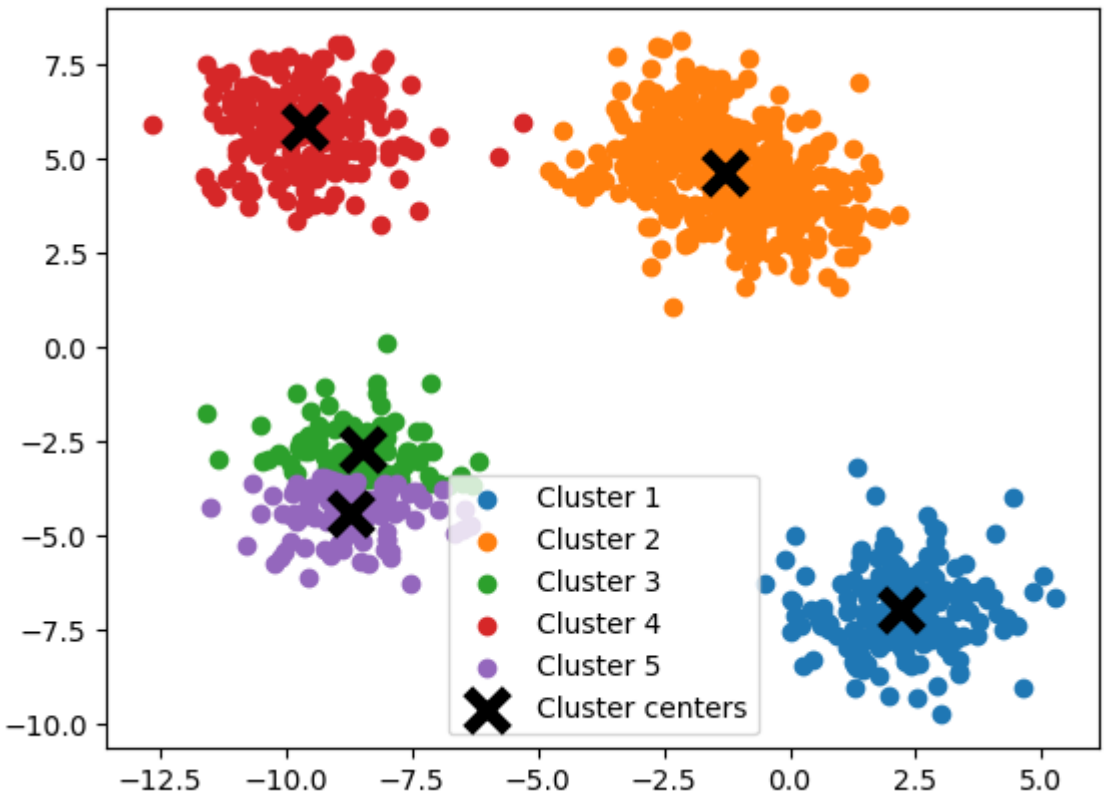
Initial assignment



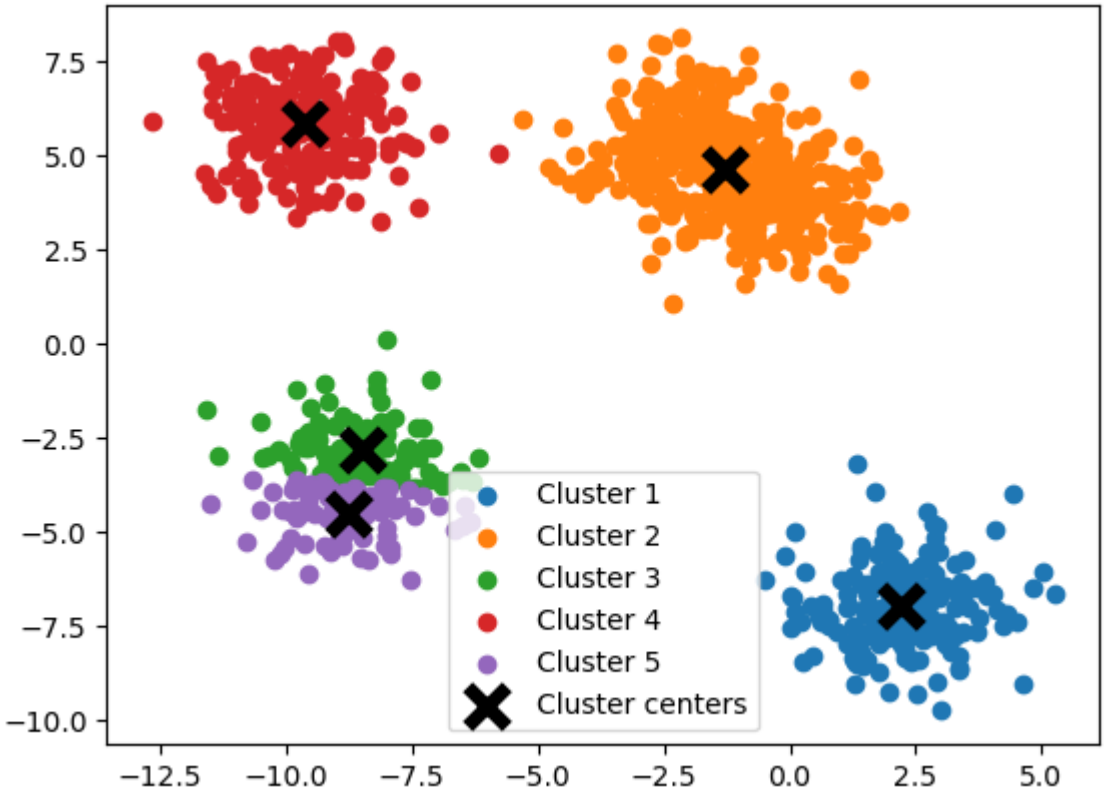
Iteration 1



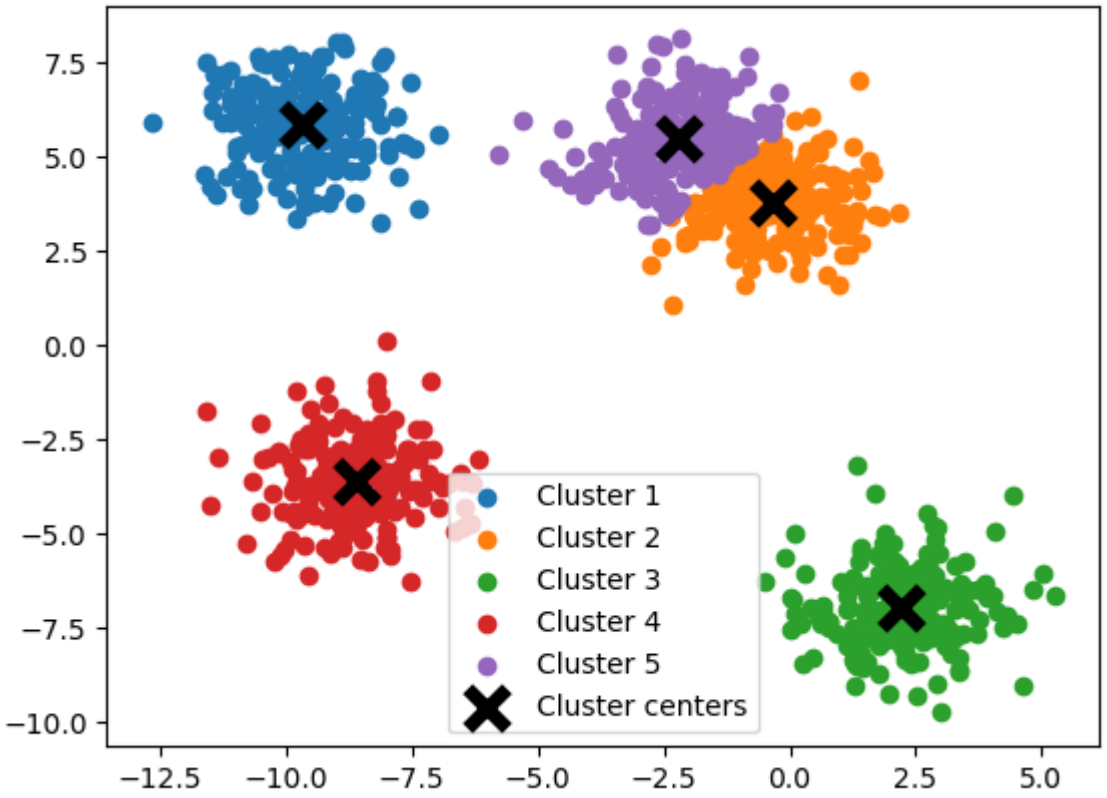
Iteration 2



Iteration 3



Final version of my clusters



final version of scikit-learn library

For this dataset, even though I used make blobs to generate a 5-clustered dataset, the generated dataset is open to interpretation. In other words, the five clusters are not so obvious, one can say that there should be 4 clusters for this dataset.

Differences of these outputs may derive from the initialization part. 2 implementation may choose different centers in the beginning which can change the output for the datasets like this. Depending on this different assignment, they assign different cluster assignments. Furthermore, since it is an iterative method, it can lead to different final cluster assignments and centers for any data.

K-means method of scikit-learn has a parameter that is named as n_init which decides the number of times the algorithm will be run. So, this k-means algorithm has a higher rate of correct results, since it runs ten times as default, and returns the best result accordingly. In my implementation, there is just one run which makes it less reliable.

Finding the best k

When I researched the ways to find the best k for a dataset on the internet, I encountered two common methods, which are The Elbow method and The Silhouette Method. When I inspected the Elbow method, I saw that I need to decide the k from a graph, which can be more interpretable.

Then, since the Silhouette method is a metric to validate the clustering, I chose it to implement. Though it is harder to implement, I decided to go with the Silhouette method since I am also familiar with this method from the class. Another reason for choosing the Silhouette method is that this method works better with the suitable datasets.

• The silhouette coefficient s for a single sample is given as:

$$s = \frac{b - a}{\max(a, b)}$$

• a: The mean distance between a sample and all other points in the same class

• b:The mean distance between a sample and all other points in the next nearest cluster

```
def silhouetteCoefficient(X, centres, labels):
        k = len(centres)
        aList = np.zeros(k)
        bList = np.zeros(k)
        clusterCount = np.zeros(k)
        for j in labels:
            clusterCount[int(j)] += 1
        for i in range(len(X)):
            sumPoint = 0
            sumPointtob = 0
            nearestCluster = findNearestCluster(X[i], centres,
labels[i])
            for k in range(len(X)):
                if (labels[i] == labels[k]):
                    dist = measureDistance(X[i], X[k])
                    sumPoint += dist
                if (labels[k] == nearestCluster):
```

```
                    disttob = measureDistance(X[i], X[k])
                    sumPointtob += disttob
            meanDist = sumPoint / (clusterCount[labels[i]] - 1)
            aList[labels[i]] += meanDist / clusterCount[labels[i]]
            meanDisttob = sumPointtob /
(clusterCount[nearestCluster])
            bList[labels[i]] += meanDisttob /
clusterCount[labels[i]]
        a = np.mean(aList)
        b = np.mean(bList)
        s = (b - a) / max(a, b)
        return s
```

- Based on the length of the centers, the number of clusters, k, is computed in the function
- The separation (b) and cohesiveness (a) values for every cluster are initialized as arrays in aList and bList.
- clusterCount is an array that uses the labels to determine how many data points are in each cluster.
- The code computes a and b for each data point iteratively over the dataset (X). The Silhouette Score depends on these parameters.
- The distances (also known as dist) between the current data point and every other data point in the same cluster are added up using sumPoint. This exemplifies cohesion (a).
- Distances between the current data point and data points in the next neighboring cluster are accumulated using sumPointtob. Separation is measured by this (b).
- By invoking the findNearestCluster function, which finds the closest cluster to the one allocated to the data point, the code finds the nearestCluster to the current data point.
- The measureDistance function is used to calculate the distances between data points.
- Cohesion is represented by a, which is computed as the mean distance within the same cluster (a).
- The mean distance to the closest neighboring cluster, or separation (b), is computed as mean of bList.

.

I have implemented a function named findBestK which takes the dataset as a parameter, and uses silhouetteCoefficient function.

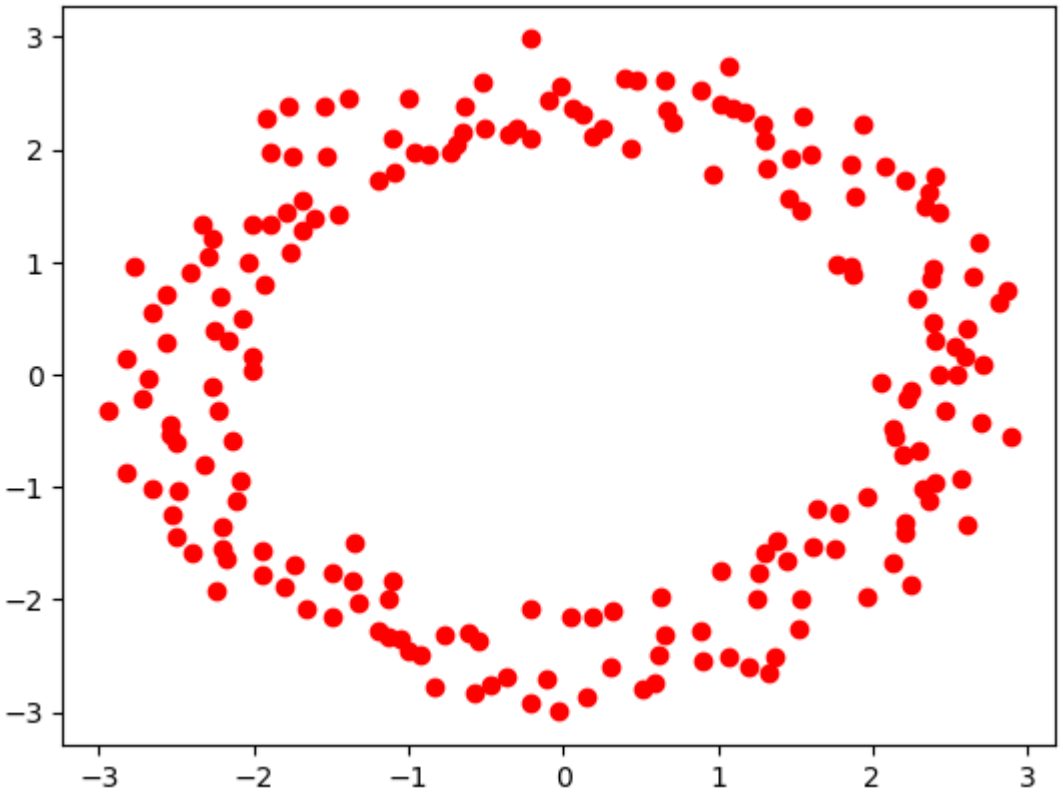
```
def findBestK(X):
    maxSilhouette = -1
    bestk = -1.0
    for i in range(3, 10):

        silhouette = kmeansWithoutPlot(i, X)
        if (maxSilhouette < silhouette):
            maxSilhouette = silhouette
            print(maxSilhouette, i)
            bestk = i
    return bestk
```

Since silhouette values are between -1 and 1, the highest values mean that it is a better cluster assignment. I have come up with this function. In this function, I tried 3 to 10 as k in a for loop, and it gives the k that gives the biggest silhouette value.

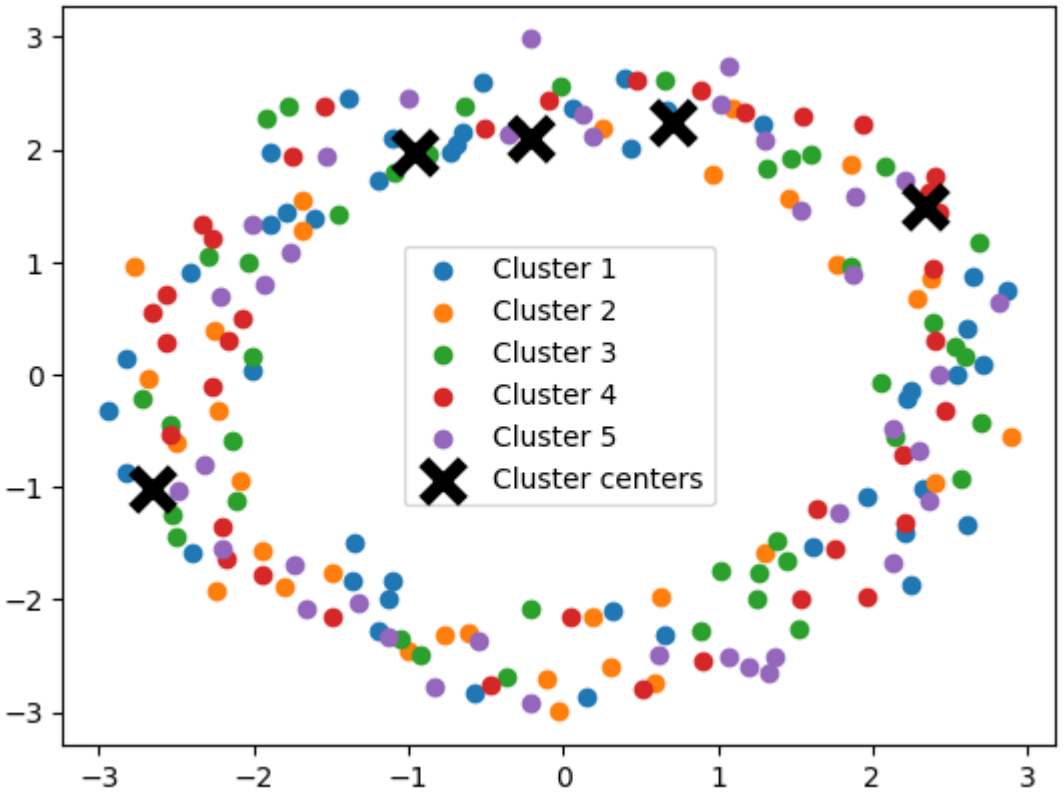
References:

- https://medium.com/analytics-vidhya/how-to-determine-the-optimal-k-for-k-means-708505d204eb
- https://www.analyticsvidhya.com/blog/2021/05/k-mean-getting-the-optimal-number-of-clusters/
- Evaluation of Clustering Quality slides from classes

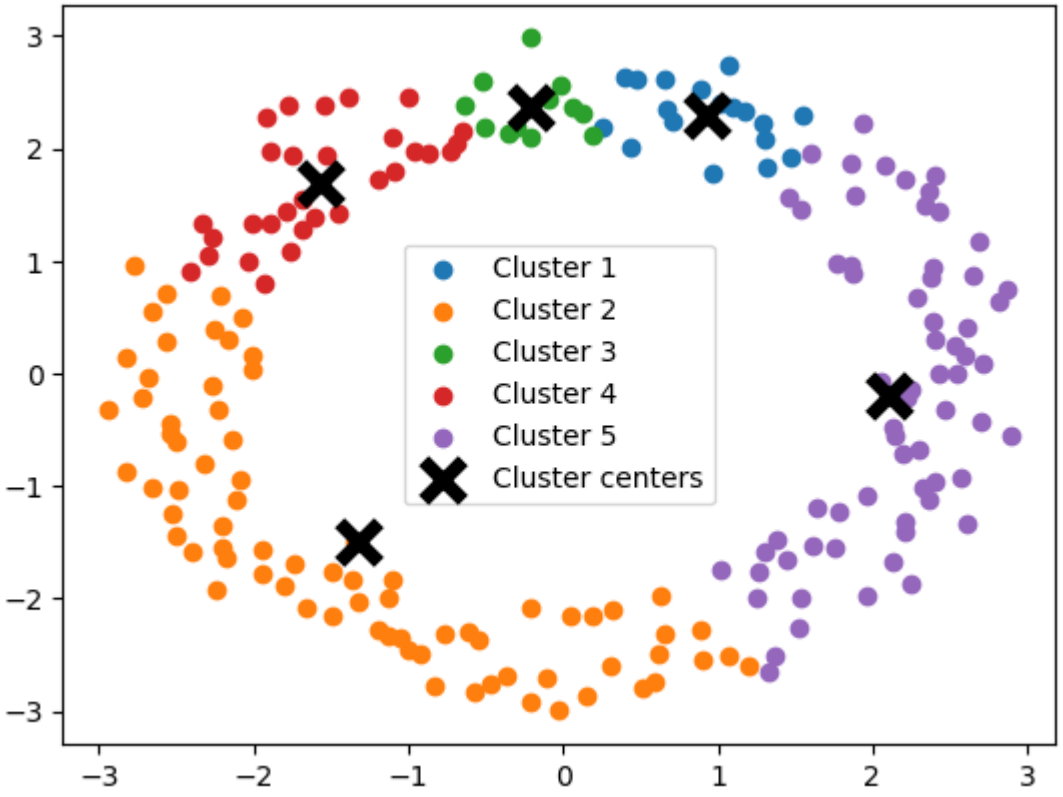
Example for inconvenient dataset



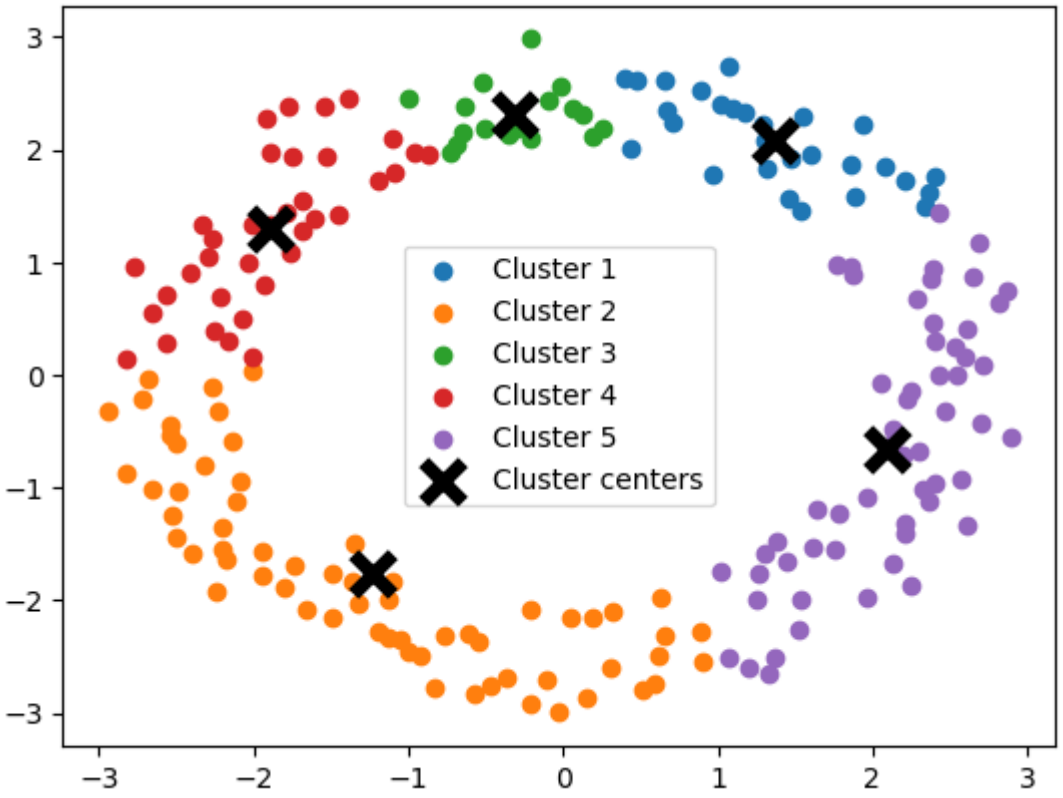
Initial data



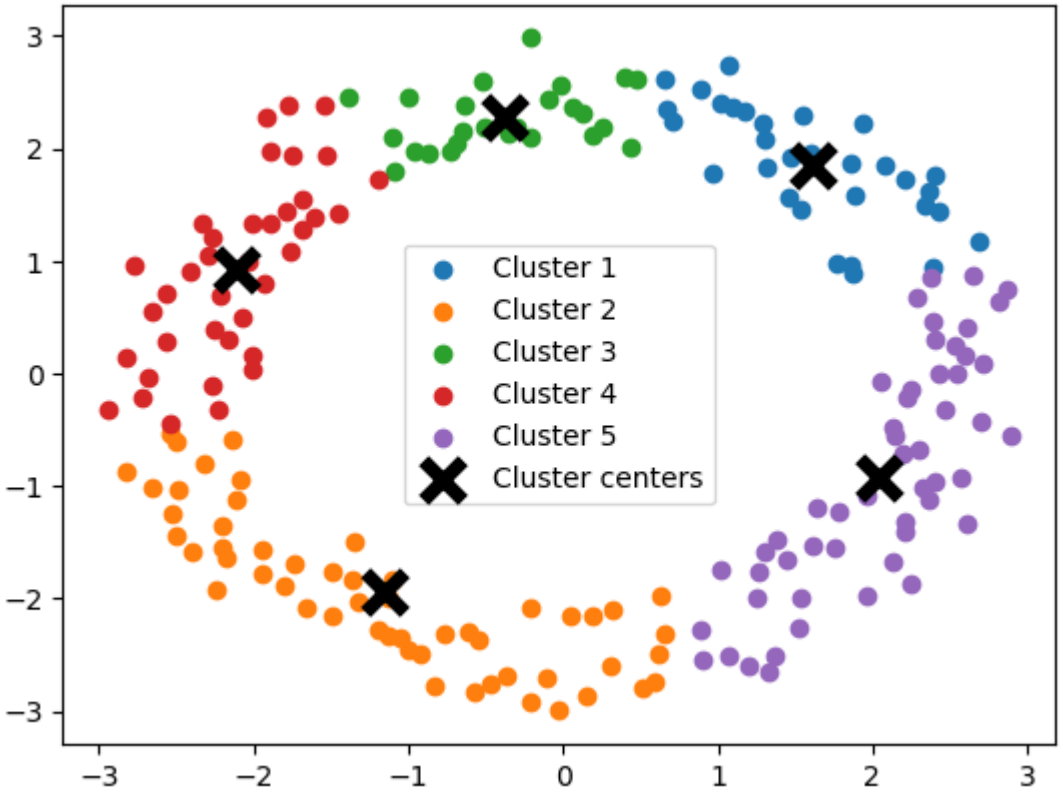
Initial assignments



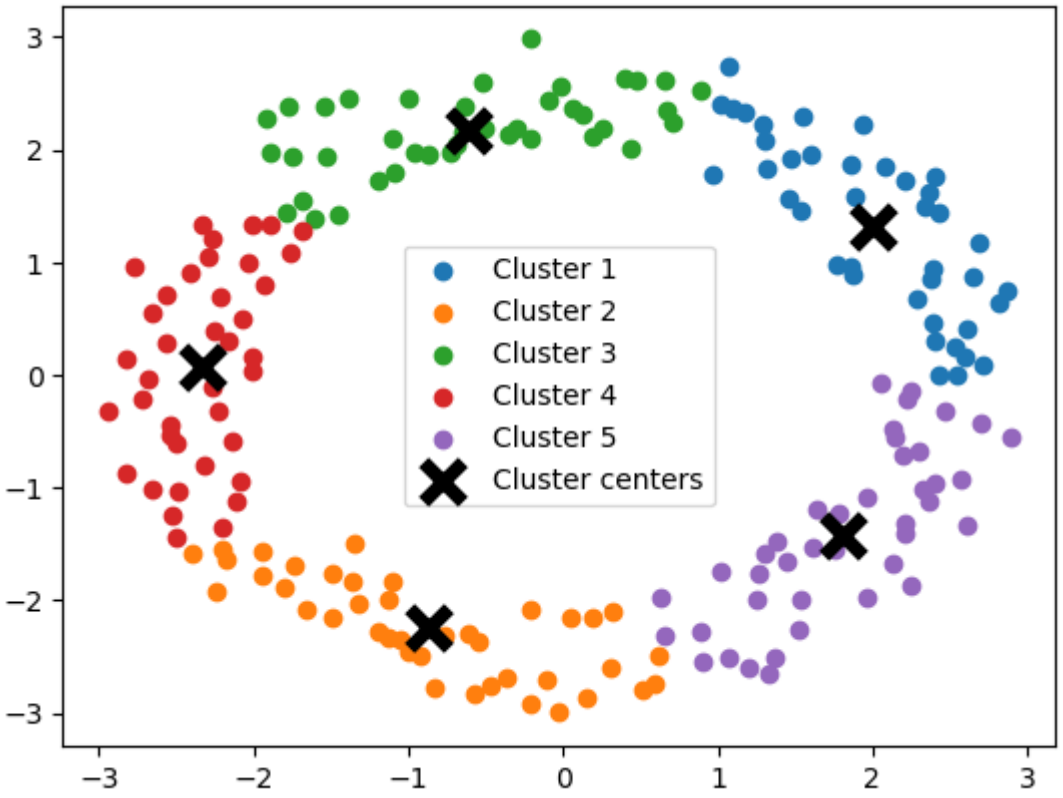
Iteration 1



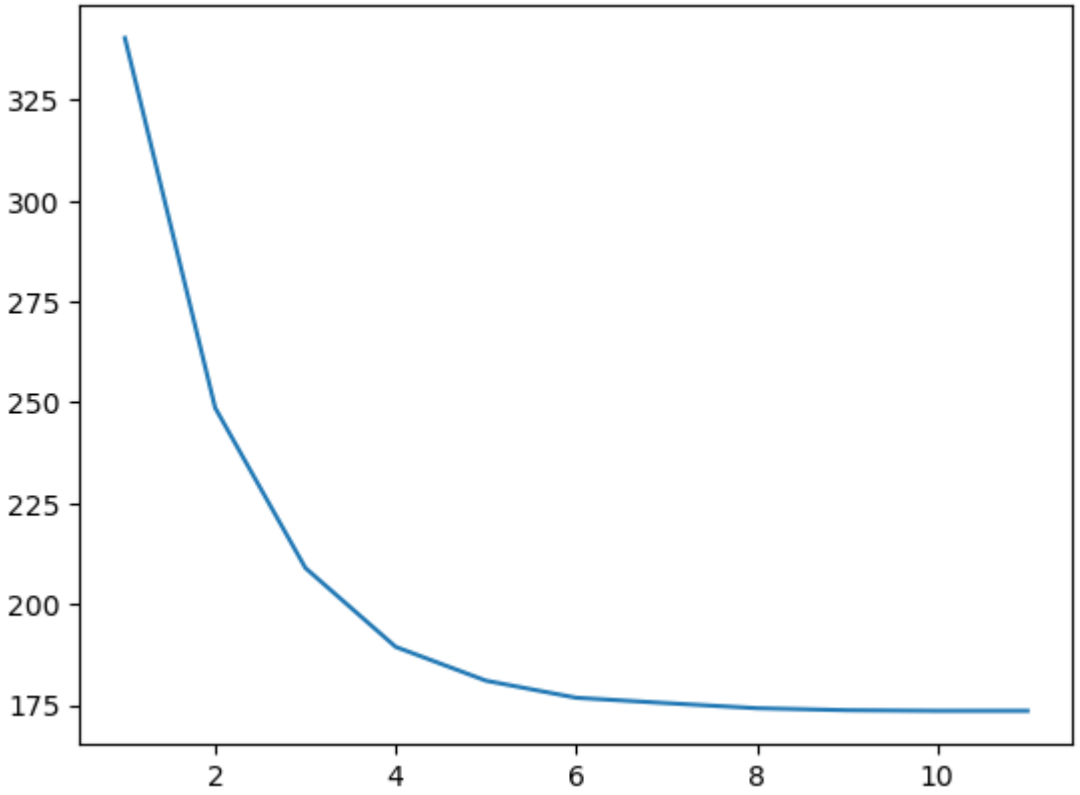
Iteration 2



Iteration 3



Final clustering



Objective function