MachineLearning_PA#3_Assignment#1_K-Means

November 30, 2021

Dataset:Cluster Dataset In this assignment we have to implement the standard version of the K-Means algorithm.

Importing the necessary libraries

[1]:

```
import numpy as np
import matplotlib.pyplot as plot
import matplotlib.animation as animation
import random
import pickle
from math import sqrt
```

ggplot style sheet

[2]:

```
plot.style.use('ggplot')
```

K-Means

[3]:

```
class KMeans(object):
    centroids = []
    assgns = []
    clusterMeans = []
    bestCentroids = []
    bestAssgns = []
    allSquareErrors = []
    bestSquareErrors = 10000
    bestFound = 0
    current_squareError = 0
    epsilon = .001
    plotcolors = ['r', 'g', 'b', 'm', 'c', 'k', 'y','b',
                  'g', 'r', 'c', 'm', 'y', 'k','b', 'g',
                  'r', 'c', 'm', 'y', 'k','b', 'g', 'r',
                  'c', 'y', 'm', 'k','b', 'g', 'r', 'c',
                  'm', 'y', 'k','b', 'g', 'r', 'c', 'm',
                  'y', 'b','k', 'g', 'r', 'c', 'm', 'y',
                  'k','g']
    def printAssignments(self, vectors):
        for index in range(len(vectors)):
            print(vectors[index], " is assigned to centroid ", self.
↪centroids[self.assgns[index]])

    def squareErrors(self):
        print("All squareErrors sums:")
        print(self.allSquareErrors)
        print("Best squareError sum:")
        print(self.bestSquareErrors)

    #calculating the Kmeans
    def calculateMeans(self,vectors,clusters):
        stop = False
        sums = []
        lens = []
        means = []
        meanX = []
        meanY = []
        squareErrorSum = 0
        for i in range(clusters):
          sums.append([0,0])
          lens.append(0)

        for j in range(len(vectors)):
          index = self.assgns[j]
          sums[index][0] += vectors[j][0]
          sums[index][1] += vectors[j][1]
          lens[index] += 1

        for k in range(clusters):
          meanX = sums[k][0]/lens[k]
          meanY = sums[k][1]/lens[k]
          means.append([meanX,meanY])
        self.centroids = means

        for i in range(len(vectors)):
          index = self.assgns[i]
          cost = self.calculateDistance(vectors[i],self.centroids[index])
          squareErrorSum += self.calculateDistance(vectors[i],self.
↪centroids[index])

        if squareErrorSum < self.bestSquareErrors:
          self.bestFound += 1
          self.bestCentroids = means
          self.bestAssgns = self.assgns
          self.bestSquareErrors = squareErrorSum
        self.allSquareErrors.append(squareErrorSum)
        if(self.current_squareError - squareErrorSum <= self.epsilon):
            stop = True
            print("Epsilon reached. Early halting")
        self.current_squareError = squareErrorSum
        squareErrorSum = 0
        return stop

    #best Results
    def bestResults(self):
       self.centroids = self.bestCentroids
       self.assgns = self.bestAssgns

    #graph
    def showGraph(self, vectors, computation, iteration):
        vectors = np.array(vectors)
        localCentroids = np.array(self.centroids)
        x,y = vectors.T
        figure = plot.figure()
        fig = figure.add_subplot(1,1,1)
        count = 0
        for num in range(len(vectors)):
            x,y = vectors[num].T
            color = self.assgns[num]
            fig.scatter(x,y, s=10, c=self.plotcolors[color])
        for i in localCentroids:
            x,y = i.T
            fig.scatter(x,y, s=200, c=self.plotcolors[count], marker='X',␣
 ↪edgecolors='k')
            count = count+1
        plot.
 ↪title("KMeans_graph"+str(iteration)+"_step"+str(computation)+"\n"+"SquareError:
 ↪ " + str(self.current_squareError))
        plot.savefig("KMeans_graph"+str(iteration)+"_step"+str(computation)+".
 ↪png")
        plot.show()
        fig.clear()
        return figure

    #select the centroids
    def selectCentroids(self, centroids, upperBound, vectors):
        self.centroids = []
        indexes = [] # make an array of random indices
        indexes = random.sample(range(0,upperBound), centroids)
        for i in indexes:
            self.centroids.append(vectors[i])

    def assignPoints(self, vectors):
        self.assgns = []
```

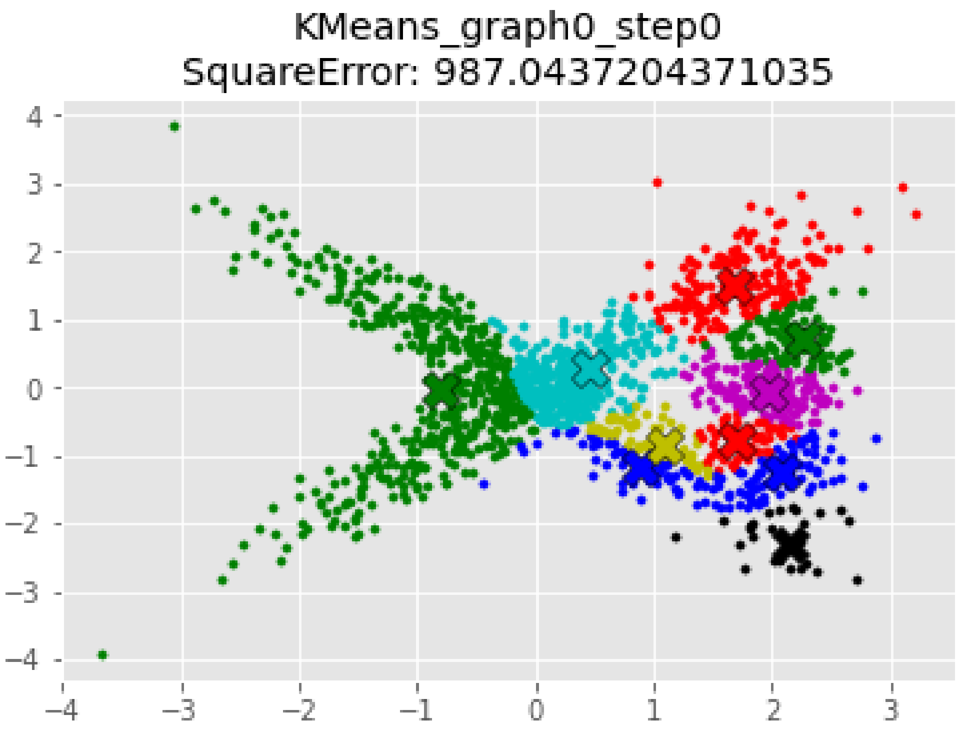
```
        assigned_centroid = 0
        cost = 100
        nextCost = 0
        wcss_sum = 0
        for i in range(len(vectors)):
            for j in range(len(self.centroids)):
                nextCost = self.calculateDistance(vectors[i], self.centroids[j])
                if cost > nextCost:
                    cost = nextCost
                    assigned_centroid = j
            self.assgns.append(assigned_centroid)
            cost = 100

    #calculate the distance
    def calculateDistance(self, point, centroid):
        return sqrt((point[0]-centroid[0])**2 + (point[1]-centroid[1])**2)

    #Kmeans Classifier
    def KMeansClassifier(self, iters, clusters, vectors, steps):
        for i in range(iters):
            stop = False
            self.selectCentroids(clusters, len(vectors), vectors)
            self.assignPoints(vectors)
            for j in range(steps):
                stop = self.calculateMeans(vectors, clusters)
                self.assignPoints(vectors)
                graph = self.showGraph(vectors, i, j)
                if(stop == True):
                    break
        self.bestResults()
        self.squareErrors()
        print("The best graph is . . .")
        graph = self.showGraph(vectors, 0, "best")
        return graph
```

[4]:

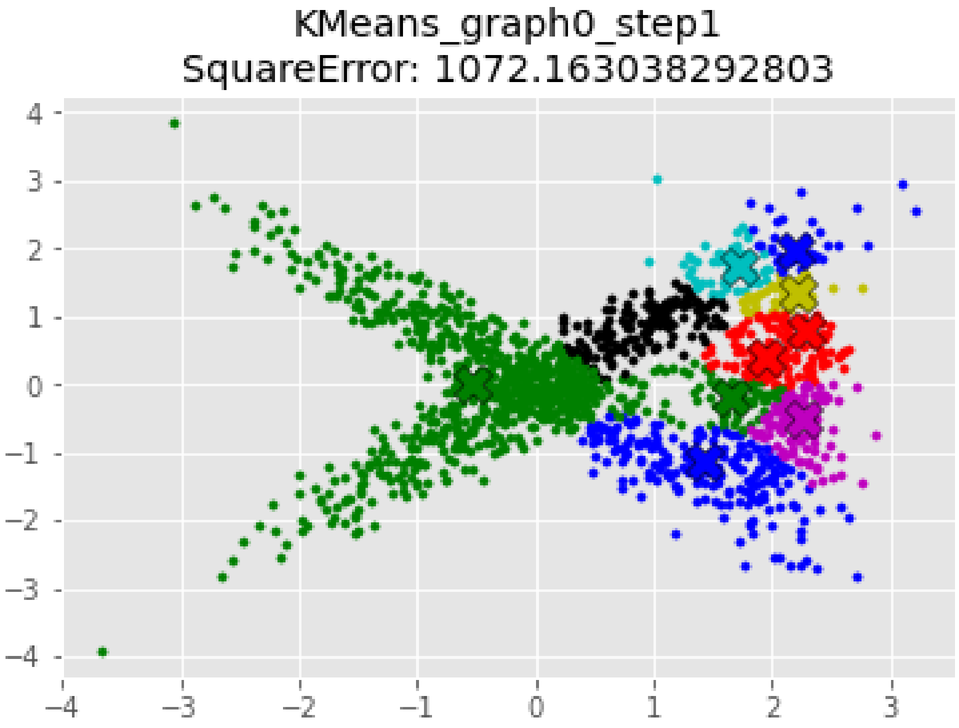
```
#defining the main
def main():
    classifier = KMeans()
    data = open("/Users/sriharshithaayyalasomayajula/Desktop/Machine␣
 ↪Learning_PSU/Program_3/Dataset/cluster_dataset.txt",'r')
    data = data.readlines()
    graphs = []
    vectors = []
    for i in data:
        vectors.append(i.split())
        for i in range(len(vectors)):
            for j in range(len(vectors[i])):
                vectors[i][j] = float(vectors[i][j])

    clusters = input("How many clusters do you want to generate? (Choose␣
↪between 1-50?)")
    iterations = input("How many iterations do you want to run?")

    steps = 100
    graph = classifier.
↪KMeansClassifier(int(iterations),int(clusters),vectors,steps)
```

[5]:
```
#Calling the main function
main()
```

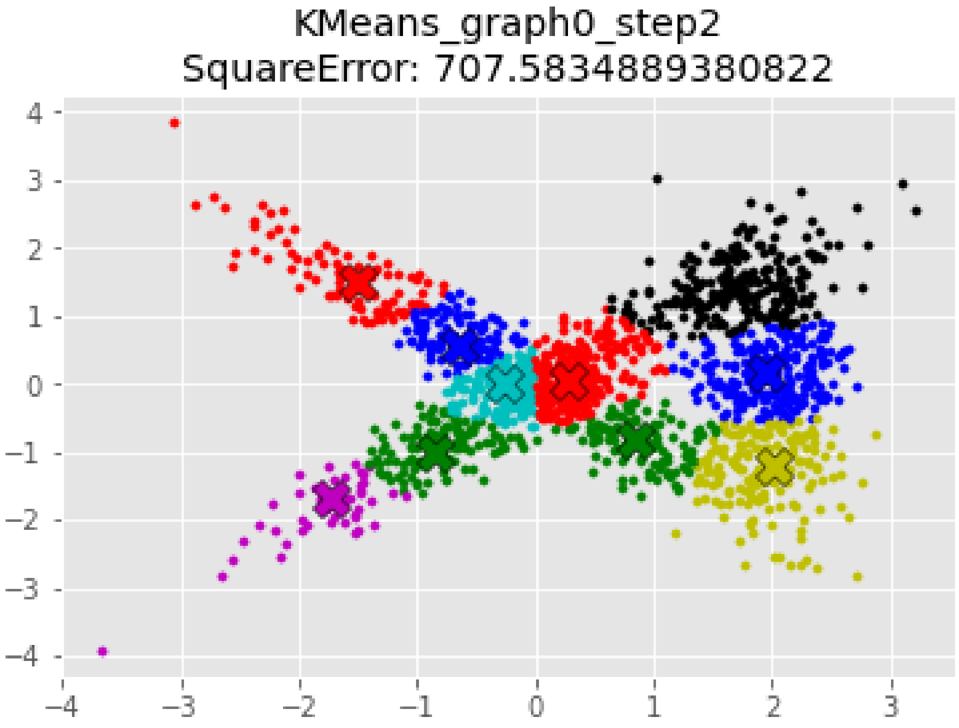
```
How many clusters do you want to generate? (Choose between 1-50?)10
How many iterations do you want to run?3
Epsilon reached. Early halting
```



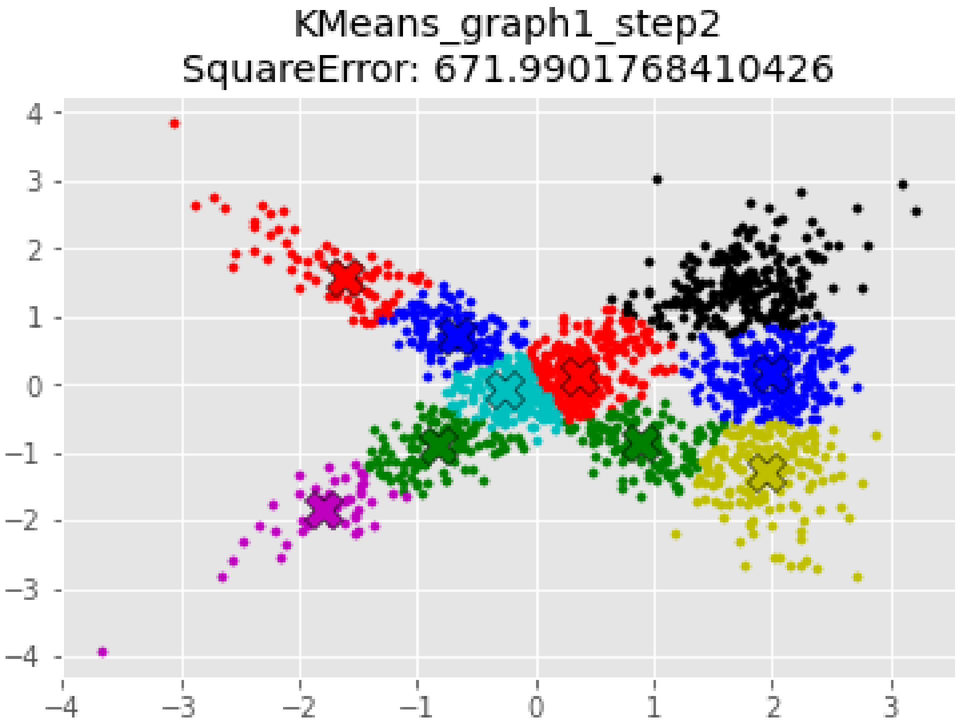
```
Epsilon reached. Early halting
```

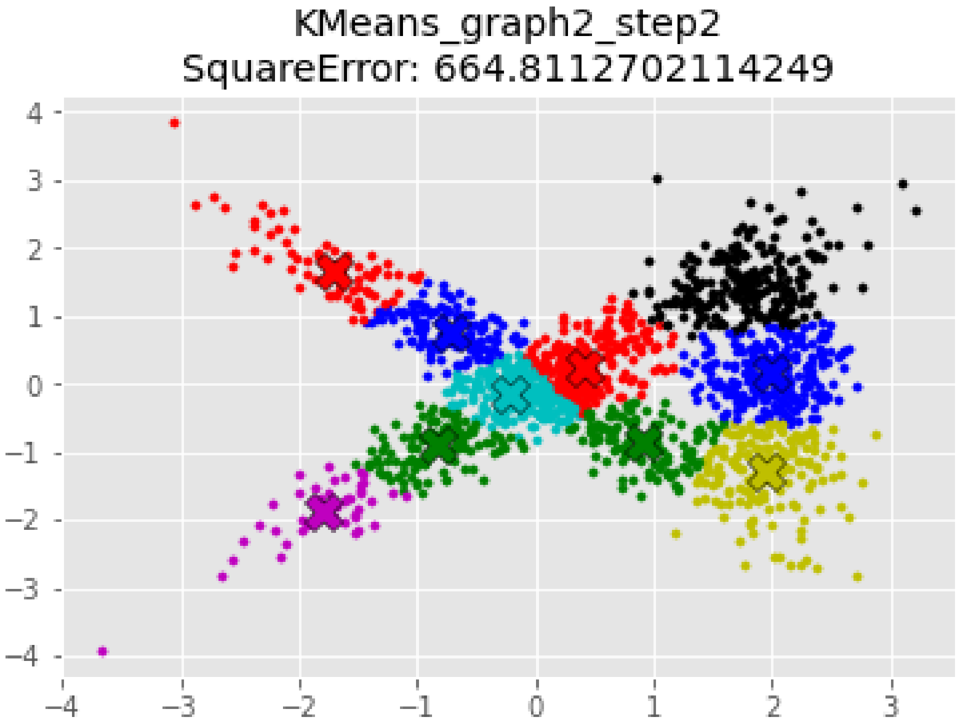


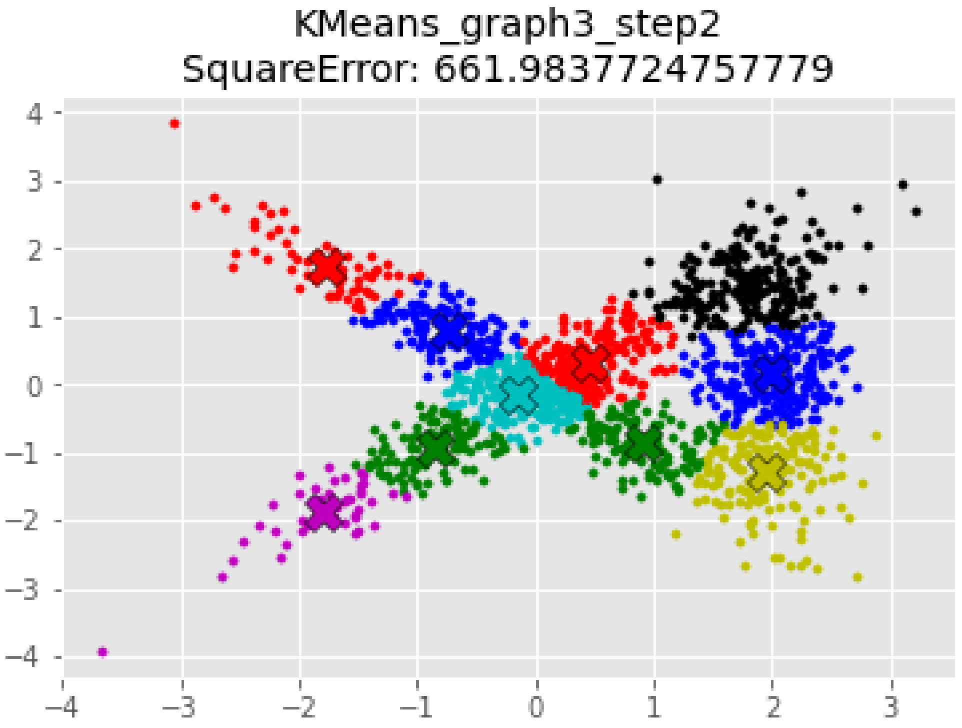
KMeans_graph0_step1
SquareError: 1072.163038292803



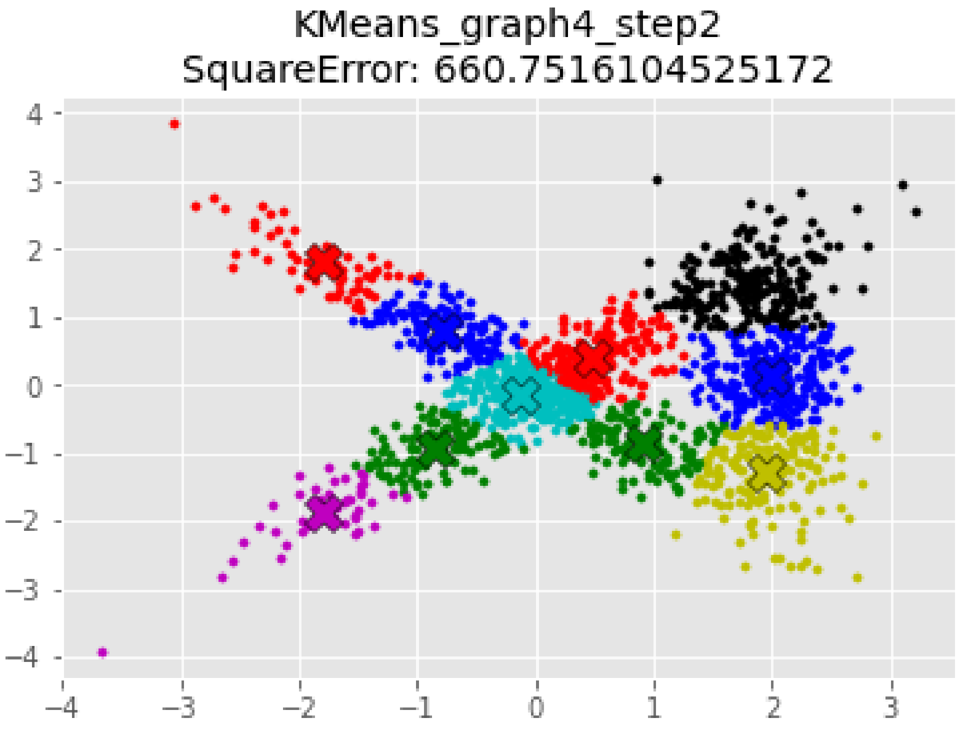
KMeans_graph0_step2
SquareError: 707.5834889380822


KMeans_graph1_step2
SquareError: 671.9901768410426

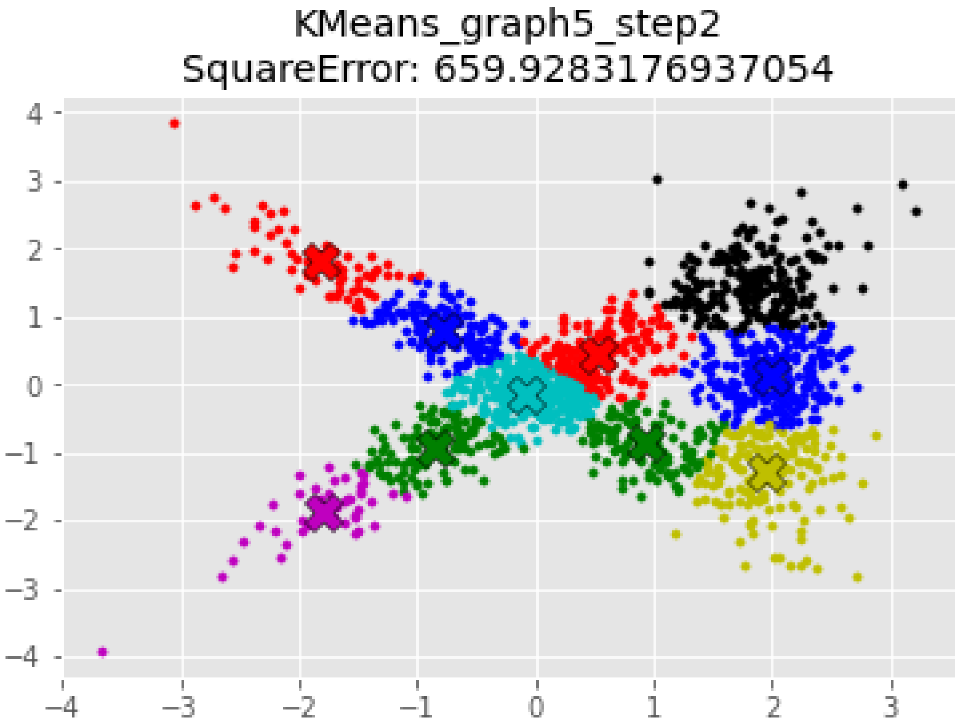

KMeans_graph2_step2
SquareError: 664.8112702114249

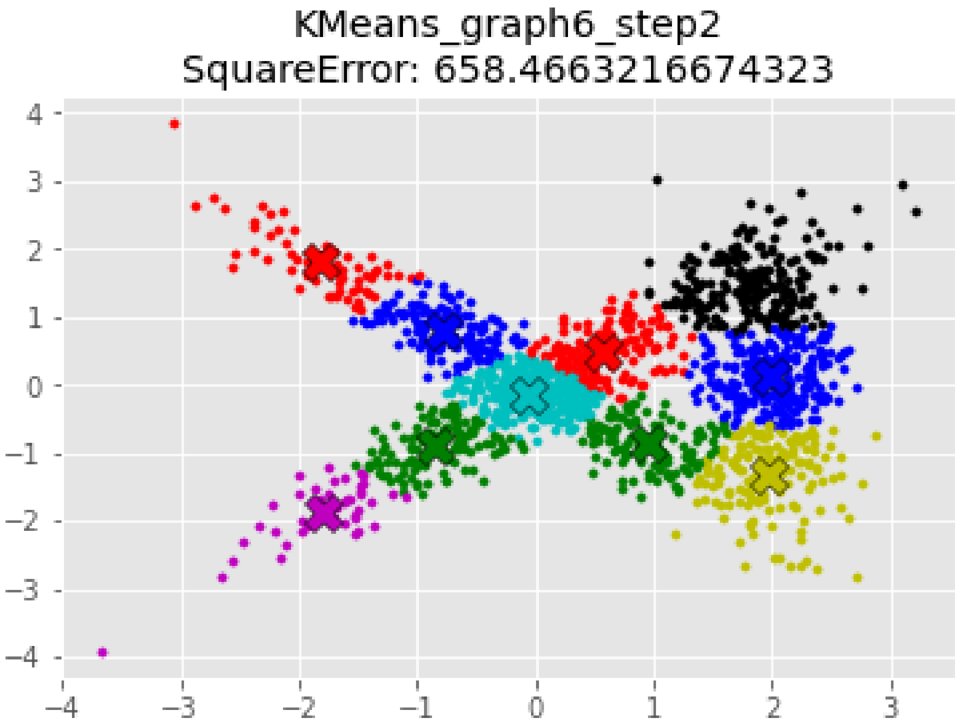


KMeans_graph3_step2
SquareError: 661.9837724757779

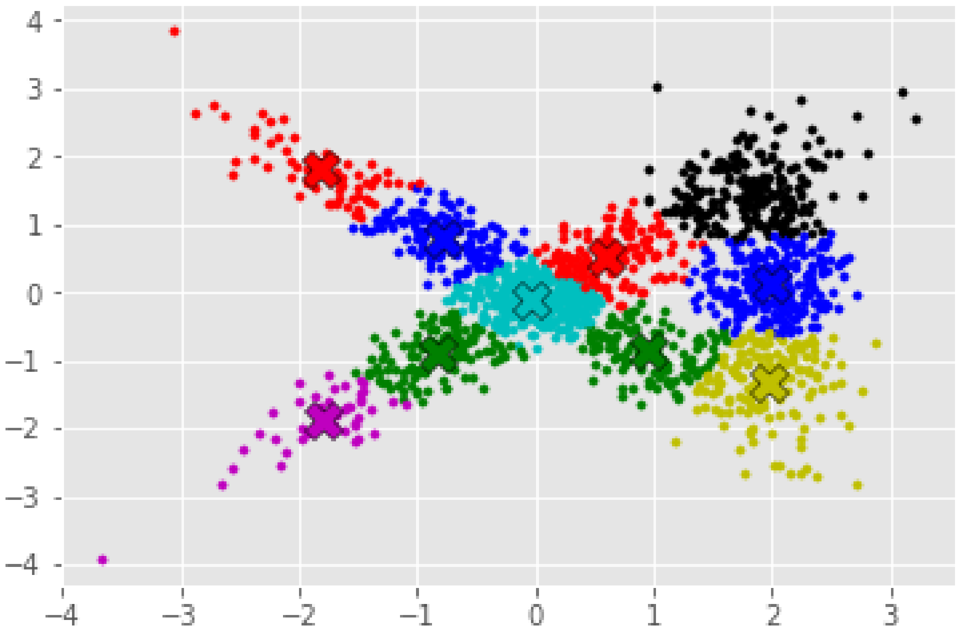


KMeans_graph4_step2
SquareError: 660.7516104525172

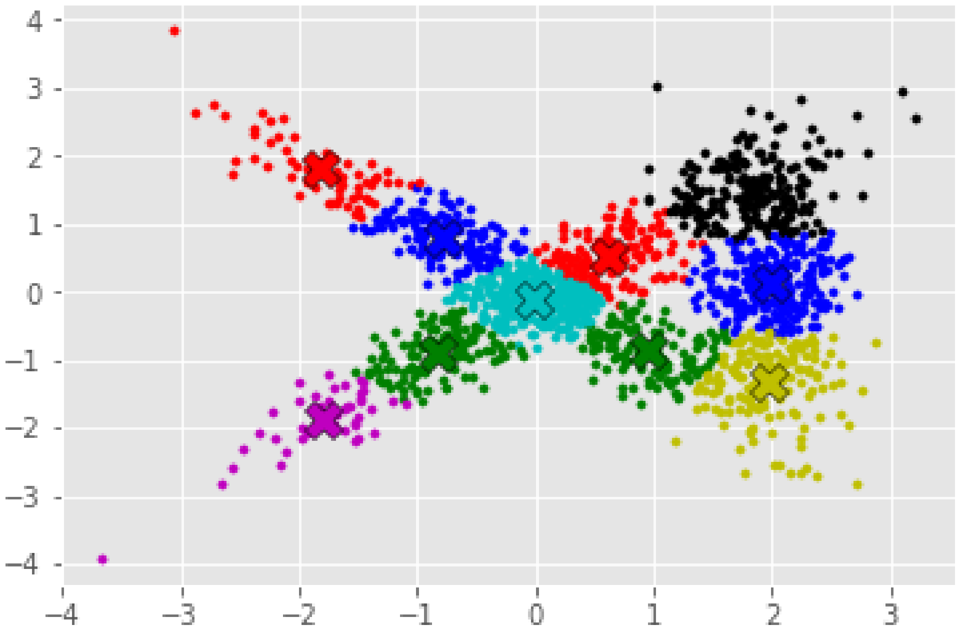

KMeans_graph5_step2
SquareError: 659.9283176937054
4
3
2
1
0
−1
−2
−3
−4
−4
−3
−2
−1
0
1
2
3

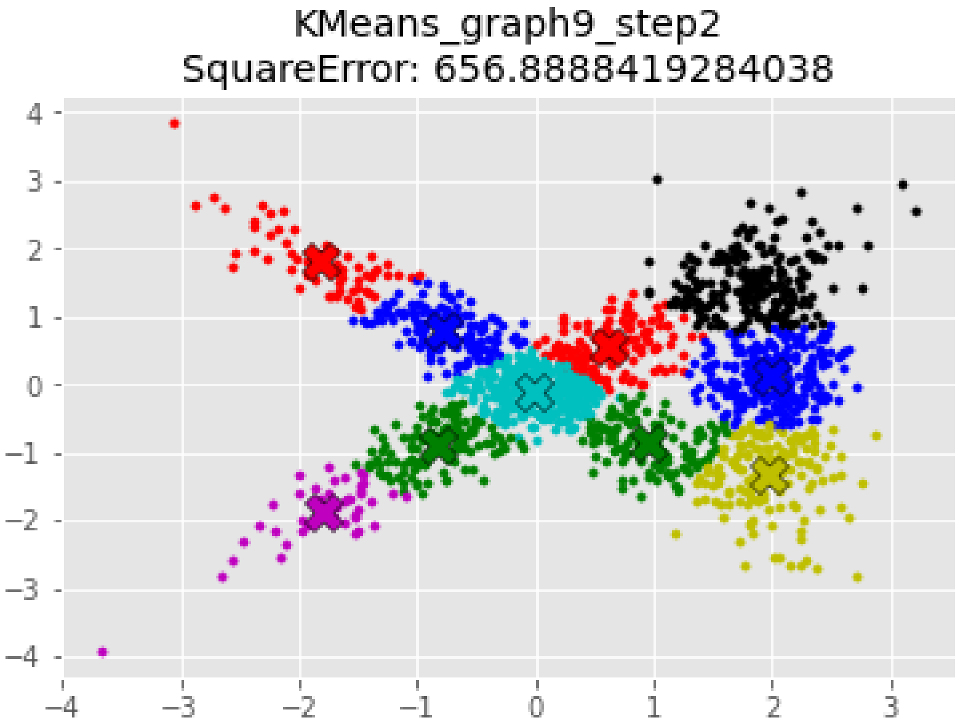

KMeans_graph6_step2
SquareError: 658.4663216674323
4
3
2
1
0
−1
−2
−3
−4
−4
−3
−2
−1
0
1
2
3

KMeans_graph7_step2
SquareError: 657.4033977570986

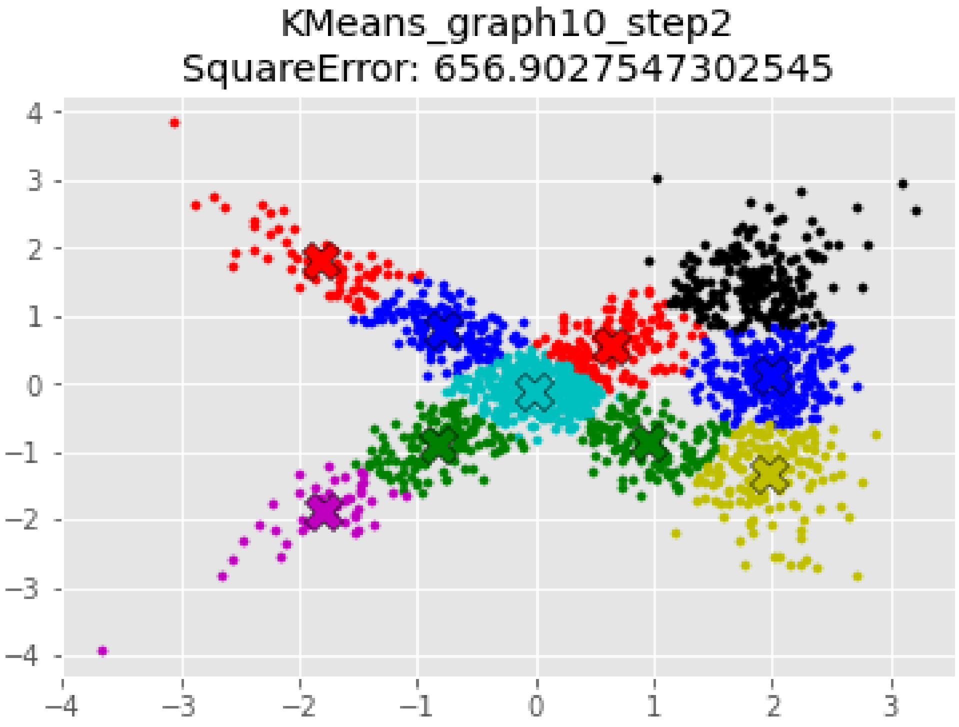


KMeans_graph8_step2
SquareError: 657.0339299401193

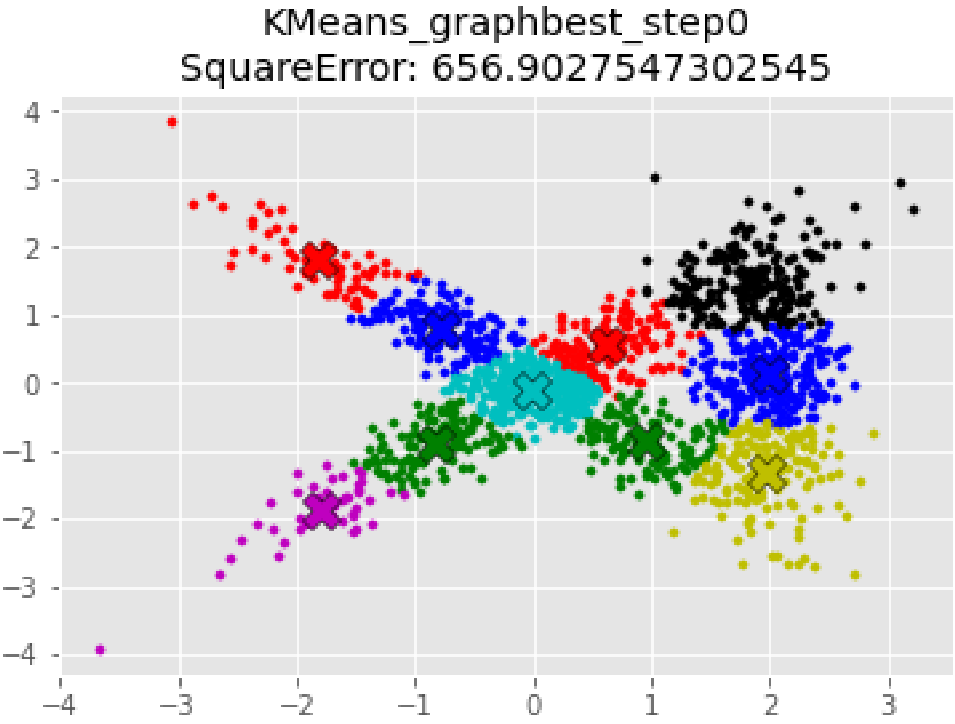




```
Epsilon reached. Early halting
```



```
All squareErrors sums:
[987.0437204371035, 1072.163038292803, 707.5834889380822, 671.9901768410426,
664.8112702114249, 661.9837724757779, 660.7516104525172, 659.9283176937054,
658.4663216674323, 657.4033977570986, 657.0339299401193, 656.8888419284038,
656.9027547302545]
Best squareError sum:
656.8888419284038
The best graph is . . .
```


KMeans_graphbest_step0
SquareError: 656.9027547302545