**Machine Learning – HW 6**

Atieh Kashani

In this assignment an unsupervised learning method (k-means) has been implemented to group the given dataset that is the 9 different indicators of quality of life in 329 U.S. Cities into K clusters such that cities in the same clusters have similar developing indicators.

The data set file contains 9 different indicators and the number of cities is 329.

The file “main\_clustering.py” contains the provided code and the codes that I have implemented to complete this script for clustering.

The following steps have been taken in this assignment:

**Step-1:** load the dataset and set environment variables including the number of clusters, and the maximal iterations

**Step-2:** Initialize centroids that can be generated randomly or randomly select K samples

**Step 3:** Assign every sample to its closed cluster using Euclidean distance

**Step 4:** Re-calculate the centroid of every cluster

**Step 5:** Check stop conditions by specifying the condition using one of the following observations:

* Number of samples that switch their memberships since last iteration
* The distance of the two centroid vectors of the sample cluster over two consecutive iterations
* Average within-cluster distance

**Step 6:** Calculate the sum of squared errors (SSE) as a quantitative metric for each K value

The scripts contain the following functions:

* run\_kmeans\_clustering(): main function for K-mean algorithm.
* initialize\_dict(): create a data structure for storing K clusters.
* assign\_points\_to\_groups(): assign data points to groups.
* calc\_euclidean\_dist\_vector(): calculate the Euclidean distance between two vectors.
* check\_minimum\_changes\_met(): check to see if points within groups changed or not.
* get\_centroids(): calculate centroid for each group.
* get\_sum\_of\_squares(): calculate the sum of square errors for the current clustering result.

**The placeholders in the following functions have been completed**:

* calc\_euclidean\_dist\_vector() : I have used “np.linalg.norm(x – y)” that calculates the Euclidean distance between the vectors x and y.

**def** calc\_euclidean\_dist\_vector(vector1, vector2):  
 *##################placeholder # start ####################* result = np.linalg.norm(vector1-vector2)  
 *##################placeholder # end ####################* **return** result

* get\_ centroids(): Calculate the centroid for each group

**def** get\_centroids(dimensions, xy\_groups):  
 *""" Takes tuple of coordinates  
 returns:  
 2,2 array of centroid points  
 """* **def** get\_point\_mean(array\_values):  
 *""" take care of issue of empty list  
 """* **if** len(array\_values) == 0:  
 **return** 0  
 **return** int(array\_values.mean())  
  
  
 length = len(xy\_groups)  
 centroid\_points = np.zeros((length, dimensions))  
 *# for each group, get the average point  
 ##################placeholder # start ####################* **for** i **in** range(length):  
 centroid\_points[i, :] = get\_point\_mean(xy\_groups[i])  
 *##################placeholder # end ####################* **return** centroid\_points

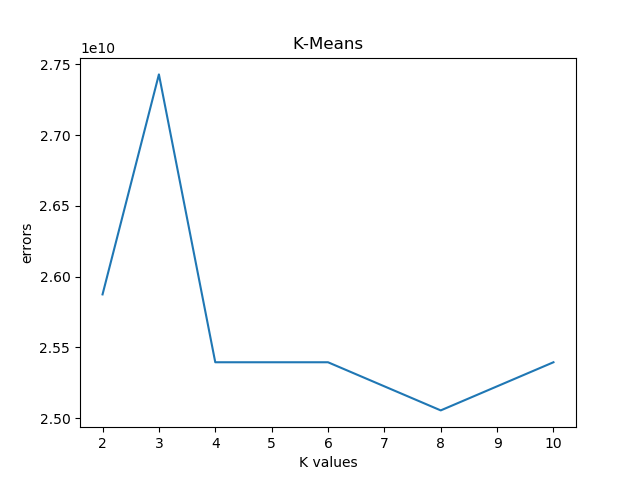
* get\_sum\_of\_squares(): Calculate the sum of squares error
* **def** get\_sum\_of\_squares(dimensions, center, samples):

*""" Get the sum of squared error, given centroid and all of its grouped points """  
 ##################placeholder # start ####################* **if** (len(samples) == 0):  
 samples = np.zeros((1, dimensions))  
 sse = np.sum(np.square(samples-center))  
 *##################placeholder # end ####################* **return** sse

**Results:**

The following two outputs are the graphs and the results of the Sum of Square Errors and number of iterations for different k- values:

**1)**



* K = 2:

SSE = 25874719561.0

12 iterations

* K = 3

SSE = 27429828940.0

2 iterations

* K = 4

SSE = 25395072831.0

17 iterations

* K = 6

SSE = 25395072831.0

22 iterations

* K = 8

SSE = 25054769132.0

15 iterations

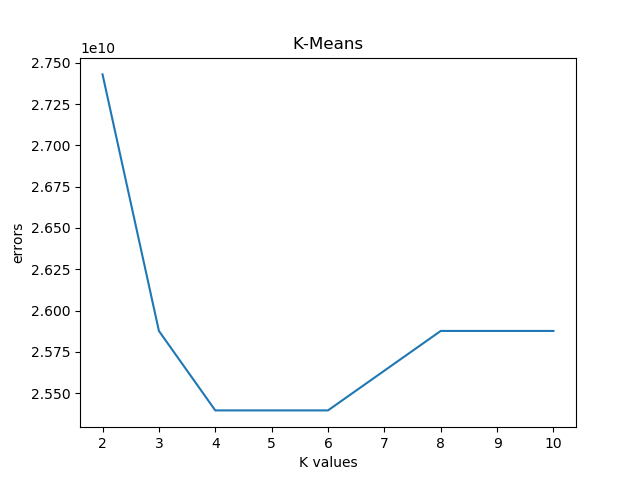
* K = 10

SSE = 25395072831.0

7 iterations

The best k-value is 8 with the Sum of Square Error of 2.50\*1e10.

**2)**



* K = 2

SSE = 27429828940.0

2 iterations

* K = 3

SSE = 25876286977.0

11 iterations

* K = 4

SSE = 25395072831.0

23 iterations

* K = 6

SSE = 25395072831.0

14 iterations

* K = 8

SSE = 25876286977.0

10 iterations

* K = 10

SSE = 25876286977.0

6 iterations

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

* For both k-value = 4 and k-value = 6 the SSE is minimum, and it is 2.53\*1e10. For k-value = 4 the number of iterations is 23 so it takes more time to converge. For K-value =6 the number of iterations is less (14 iterations) so the best k-value is 6.