Mining Big Data Assignment 3

`Exercise 2 PageRank`

- What is Google PageRank algorithm?

Michigan alum Larry Page and his fellow Stanford PhD student Sergey Brin introduced PageRank in 1998. It works by counting the number and quality of links to a page to determine how important the website is.

Implementation:

Code File:

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/496edd2b-0f9e-4249-95da-51863f8ea169/pagerank.py

- *General idea of the Algorithm:*

The algorithm uses a hash map data structure to store all the nodes and their associative edges. In every iteration, the PageRank of each node gets updated by following the algorithm:

$$PR(x) = \frac{(1-\beta)}{N} + \sum_{y \to x} \frac{PR(y)}{Out(y)}$$

In the above formula, Beta stands for damping factor, PR(y) is the PageRank value for node y, and Out(y) is the number of out links for node y.

- *Details about the PageRank Algorithm:*

The first and also the most important step in the PageRank algorithm is to store each node and its edges in an appropriate way so that we can not only access this information easily but also very time and memory efficiently.

step 1: create a dictionary called edges, the key is the name of one node, and the value is all its children.

```
edge_list = []
for edge in edge_list:
    from_, to_ = edge.split(',')
    from_, to_ = int(from_), int(to_)
    if to_ not in edges[from_]:
        edges[from_].append(to_)
```

```
BD>
BD> & C:/Users/huang/AppData/Local/Programs/Python/Python310/python.exe "c:/Users/huang/Desktop/PageRan
ify_version_pageRank.py"
{2076: [4785, 5793, 6338, 9484, 2564, 6395, 9994, 5016], 5793: [9484], 2564: [5793, 6395, 9994]})
BD>
```

The figure shown above demonstrated the results from edges. Take first node 2076 as an example, the node 2076 has 8 out links, they're 4785, 5793, 6338, 9484, 2544, 6395, 99944, and 5016.

step 2: In order to update the PageRank value for each node, we have to know the parents of the node, as well as the children number of each parent. Therefore, in this step, a new dictionary has been created to store the information needed.

```
for parent in edges:
    for child in edges[parent]:
        nodes[child].append([parent, len(edges[parent])])
```

```
k MBD/modify_version_pageRank.py/modify_version_pageRank.py"
{2076: [[]], 2564: [[2076, 8]], 4785: [[2076, 8]], 5016: [[2076, 8]], 5793: [[2076, 8], [2564, 3]], 6338: [[2076, 8]], 6395: [[2076, 8], [2
564, 3]], 9484: [[2076, 8], [5793, 1]], 9994: [[2076, 8], [2564, 3]]}
```

The figures shown above demonstrate the relationships of each node. Take node 2564 as an example, node 2564 has one parent node which is node 2076, and the children number of 2076 is 8. In this way, all the information is kept nice and very memory and time efficient.

step 3: In this step, we iteratively update each node's PageRank value. The iteration will stop in two cases: 1. when the loop has reached the end, 2. when the difference is smaller than a threshold.

```
def update_pagerank(node, parents, d):
        pagerank_sum = 0
        for parent in parents:
            if len(parent) == 0:
                pagerank_sum += 0
            else:
                pagerank_value = pr[parent[0]]
                outlinks = parent[1]
                pagerank_sum +=(pagerank_value / outlinks)

        pr[node] = (1-d)/(n) + d*pagerank_sum
```

The figure above has demonstrated how the PageRank value got updated.

- *The evaluation of the algorithm in terms of time and memory efficiency:*

Traditional method:

For the traditional way of calculating PageRank, we're using a matrix to represent graphs and using a vector to represent PageRank values for each node. This may raise two issues that may lead the algorithm to fail. The first issue is: that given the size of the Web, there may have millions of nodes. If we still use the matrix to represent this graph, the main memory of our computer cannot fit. The space complexity is O(n^2) where n is the number of nodes in the graph. The second issue is: that since each node's PageRank value gets updated through the multiplication of matrix and vector, the time complexity would be extremely high. The time complexity would essentially be O(kv^2) where K is the number of iterations and v is the number of nodes in the graph.

Proposed method:

In order to reduce high time complexity, we take advantages of quick retrieval nature of dictionary data structure. By observing the traditional method for updating PageRank values, matrix-vector multiplication is very time wasteful. By using dictionary, we can directly access the out links and in links of the current node with O(1) time complexity, and then we can just using the algorithm:

In order to reduce the high space complexity mentioned above, a dictionary of this data structure is used to represent a graph instead of a matrix, to be more specific, `defaultdict` is used. The `defaultdict` is a subdivision of the `dict` in python, it allows each new key to be given a default value based on the type of dictionary being created. In our algorithm, we define the value of each key to be a list, and in the list, we store the parent node and the number of children for each parent node. For example: {1: [2,3], 2: [1,4], 3: [1,4], 4: [1, 2]}. In this example. node 1 has one parent which is node 2 and the children number(out links) of node 2 is 3, meaning there are 3 websites that website 2 is pointing to. In this way, all the nodes and their associative edges have been stored, the space complexity has been reduced to O(V+E). By comparing the traditional method in which a lot of space is wasted due to 0s present in the matrix, this method is more space-efficient.
In order to reduce high time complexity, we take advantage of the quick retrieval nature of dictionary data structure. By observing the traditional method for updating PageRank values, matrix-vector multiplication is very time wasteful. By using the dictionary, we can directly access the out links and in links of the current node with O(1) time complexity, and then we can just use the algorithm:

$$PR(x) = \frac{(1-\beta)}{N} + \sum_{y \to x} \frac{PR(y)}{Out(y)}$$

to update each node's PageRank value.

In this way, the time complexity can be reduced to O(V+E) since we only need to go through each node and their edges once. This time complexity is way smaller than O(V^2) in sparse graph.

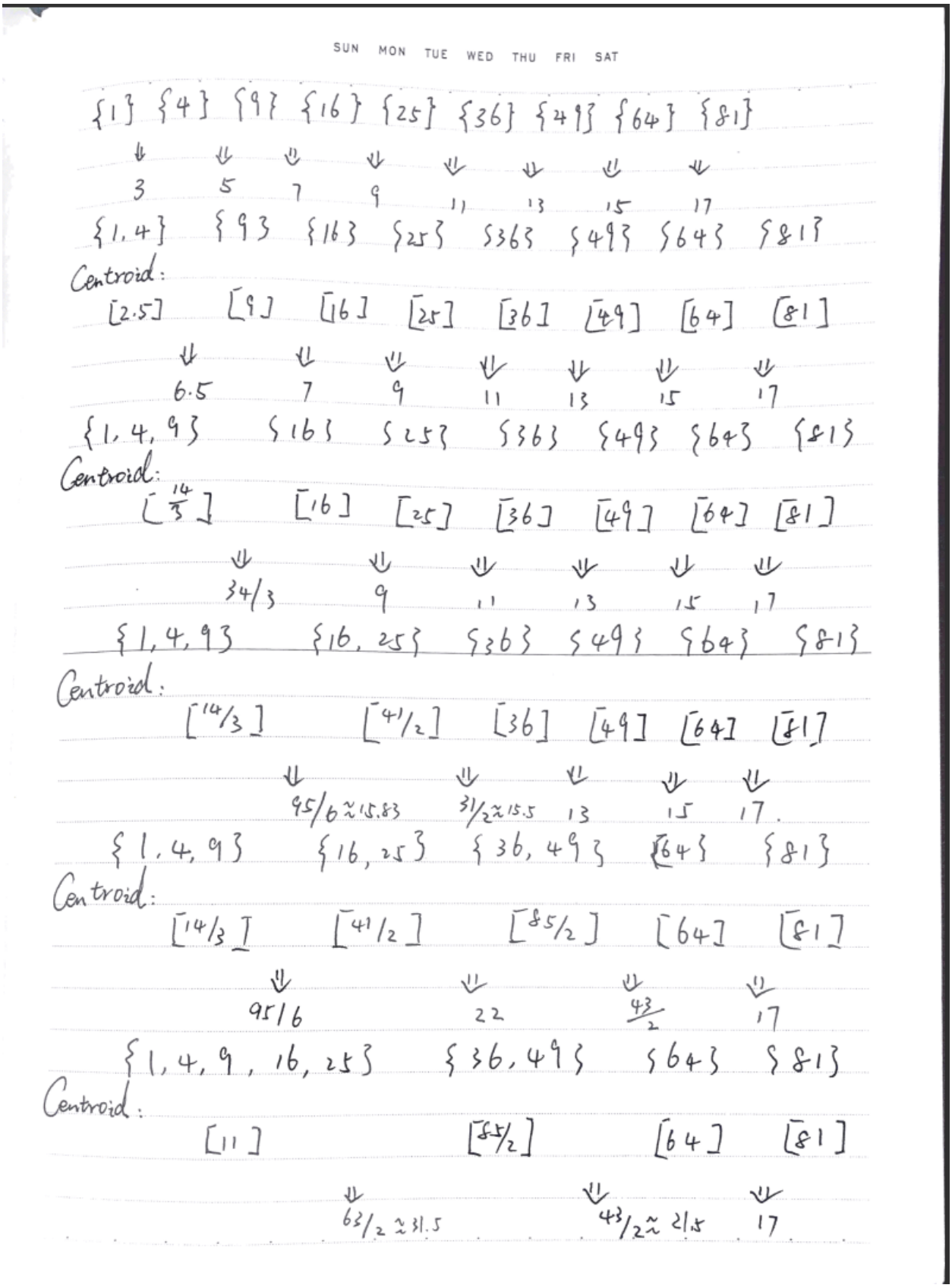
- Results of ten nodes having the largest PageRank：

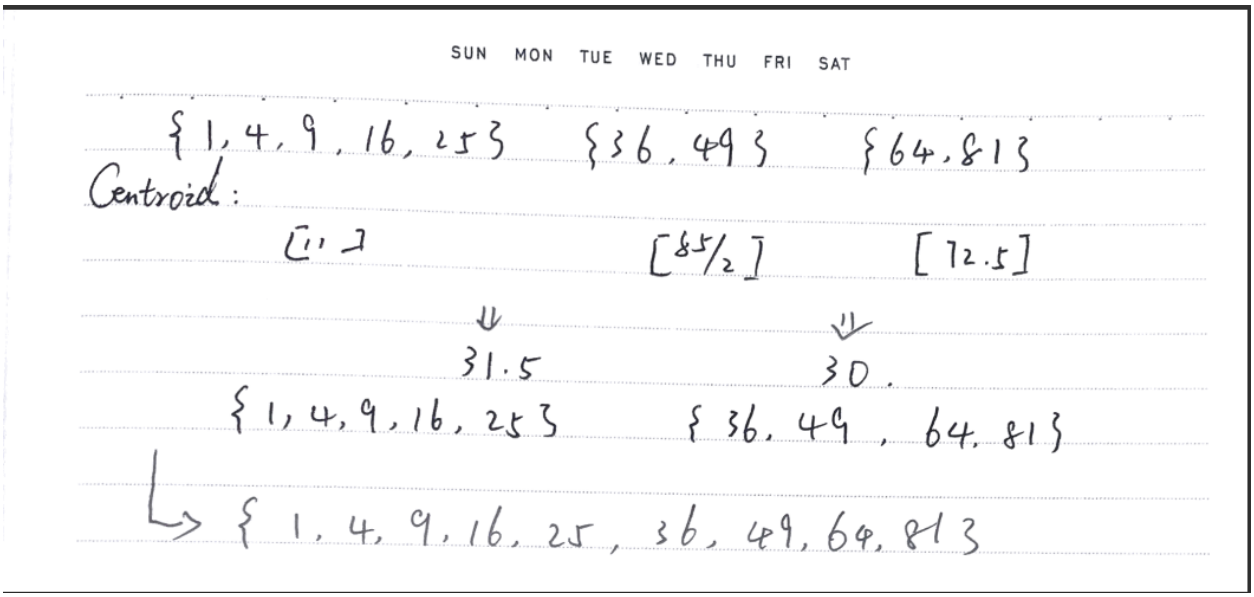
PageRank_Result.txt

1	163075	0.0009521123337668
2	597621	0.0009013686628239
3	537039	0.0008953815726589
4	837478	0.0008761661604313
5	885605	0.0008216087428295
6	551829	0.0007901082073484
7	41909	0.0007794946931031
8	605856	0.0007791356753662
9	504140	0.0007457503352830
10	819223	0.0007101828701990

Exercise 3 Clustering

Question 1:


SUN MON TUE WED THU FRI SAT
{1} {4} {9} {16} {25} {36} {49} {64} {81}
3 5 7 9 11 13 15 17
{1,4} {9} {16} {25} {36} {49} {64} {81}
Centroid:
[2.5] [9] [16] [25] [36] [49] [64] [81]
6.5 7 9 11 13 15 17
{1,4,9} {16} {25} {36} {49} {64} {81}
Centroid:
[14/3] [16] [25] [36] [49] [64] [81]
34/3 9 11 13 15 17
{1,4,9} {16,25} {36} {49} {64} {81}
Centroid:
[14/3] [41/2] [36] [49] [64] [81]
95/6 ≈ 15.83 31/2 ≈ 15.5 13 15 17
{1,4,9} {16,25} {36,49} {64} {81}
Centroid:
[14/3] [41/2] [85/2] [64] [81]
95/6 22 43/2 17
{1,4,9,16,25} {36,49} {64} {81}
Centroid:
[11] [85/2] [64] [81]
63/2 ≈ 31.5 43/2 ≈ 21.5 17



- What is Hierarchical or agglomerative algorithms:

Hierarchical or agglomerative algorithms start with each point in its own cluster. In Exercise 3 question 1 example, each integer forms a cluster. For instance: {1}, {4}, {9} …. Clusters are combined based on their closeness, in our case, the definition of “close” is each cluster’s centroid, and the clusters with the closest centroids are merged.

- The steps to perform hierarchical clustering:

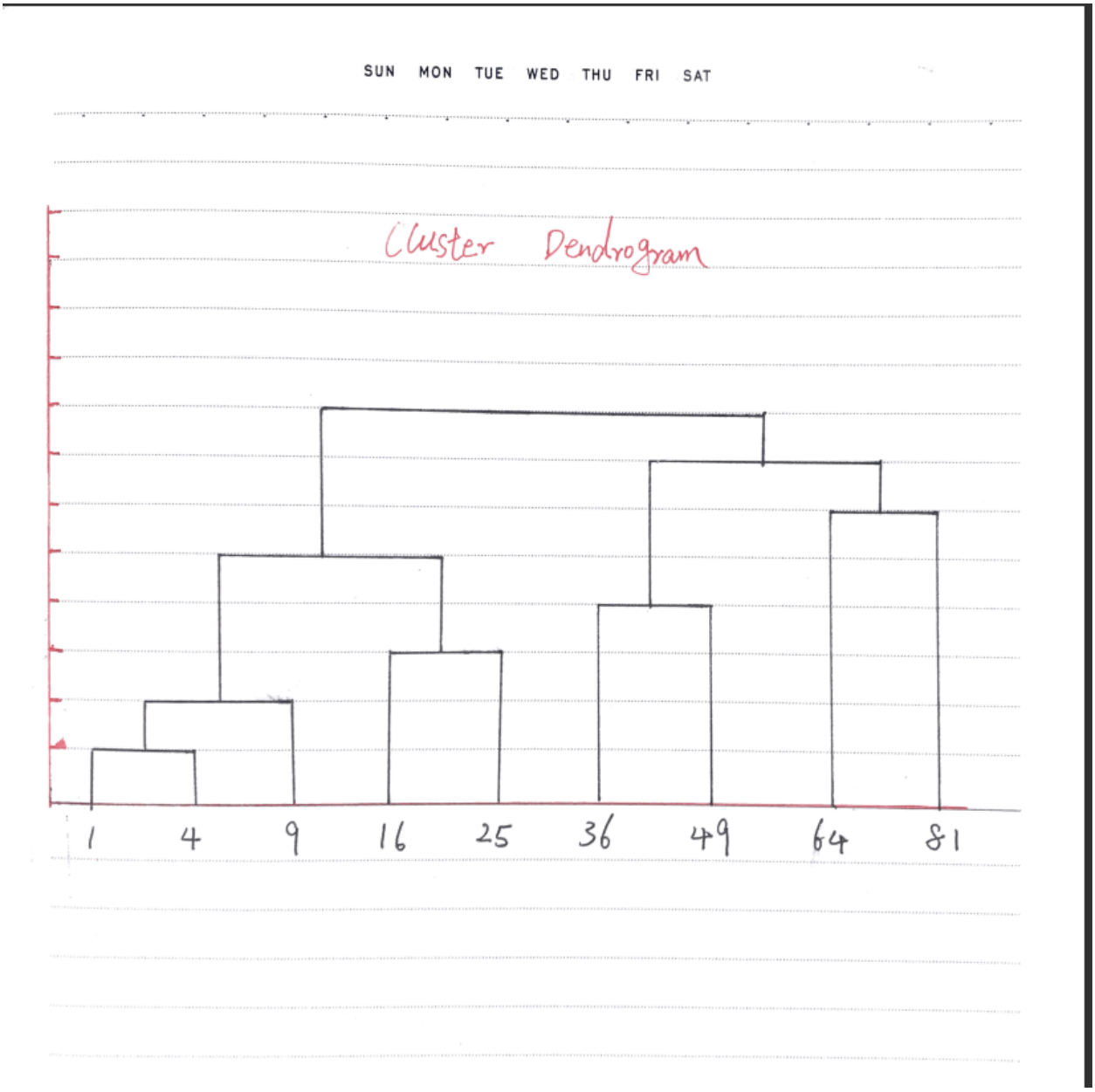
Before we begin, a few things need to be decided in advance:

1. The clusters are represented as “{}”.
2. The clusters with the closest centroids are merged.
3. The process will stop when no more clusters need to be merged.

To begin, since we’re dealing with one-dimensional points, each point can form a cluster and that point is the centroid. As the figure shows above, the clusters are initialised as {1} {4} {9} {16} {25} {36} {49} {64] {81]. Then we calculate the distance between each cluster by subtracting every two clusters’ centroids. As figure show above, the distance between cluster {1] and cluster {4} is 3, cluster {4} and cluster{9} is 5.... Then we merge two clusters with the smallest distance, in our case, cluster {1} and cluster {3] are

merged to be cluster {1, 3}. The process goes on until no more clusters need to be merged, which means only one cluster remains. As figure shown above, the last cluster is {1,4,9,16,25,36,49,64,81}.

- The results shown in dendrograms:



`Question 2:`

- General idea of the K-means Algorithm:

K-means algorithm is one type of point-assignment algorithm. The algorithm assumes a Euclidean space, and it also assumes the number of clusters, k, is known in advance. The algorithm will first choose k points that are likely to be indifferent clusters and make these points the centroids of their clusters. As long as the new clusters are not different from precious clusters, the algorithm will iteratively find the centroid to which point is closest, then add p to the cluster of that centroid, and finally, adjust the centroid.

- Code implementation of K-means Algorithm:

Code file:

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/edc2f75d-e5b2-42c6-8299-e94f4b613763/K-means_Algorithm.py

- step 1: initially choose k points that are likely to be in different clusters

```
p1, p2, p3 = random.sample(range(0, len(data)), 3)
centroids.append(data[p1])
centroids.append(data[p2])
centroids.append(data[p3])
```

The above code demonstrated the process of randomly choosing k points (in our case K = 3) that are likely to be in different clusters and make these points the centroids of their clusters.

- step 2: for each points in the data set, calculate Euclidean distance between them to the centroids in each clusters.

```
while True:
    for p in data:
        curr = []
        for c in centroids:
            Euclidean_dis = Euclidean__distance(p,c)
            #print(Euclidean_dis)
            curr.append(Euclidean_dis)
        distance_p_to_c.append(curr)
```

```
def Euclidean__distance(p, c):
    return sqrt(sum((e1-e2)**2 for e1, e2 in zip(p,c)))
```

- step 3: find the centroid to which p is closest. If the new clusters is same as the precious clusters, then we can end the process, otherwise, make the new clusters the current clusters.

```
for p in distance_p_to_c:
    #print(distance_p_to_c[p])
    new_result_list.append(np.argmin(p))

if  new_result_list == result_list:
    return result_list
else:
    result_list = new_result_list
    distance_p_to_c = []
```

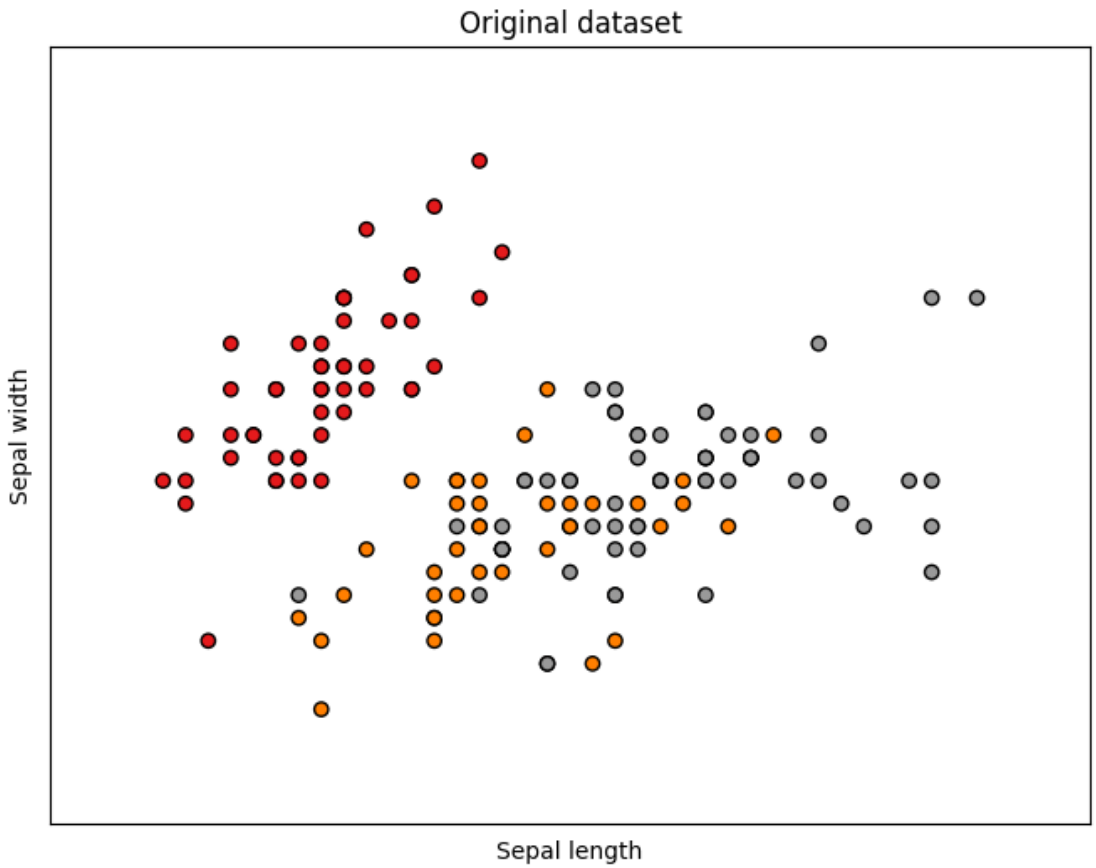
- step 4: adjust the centroids by calculating the mean value in each clusters.

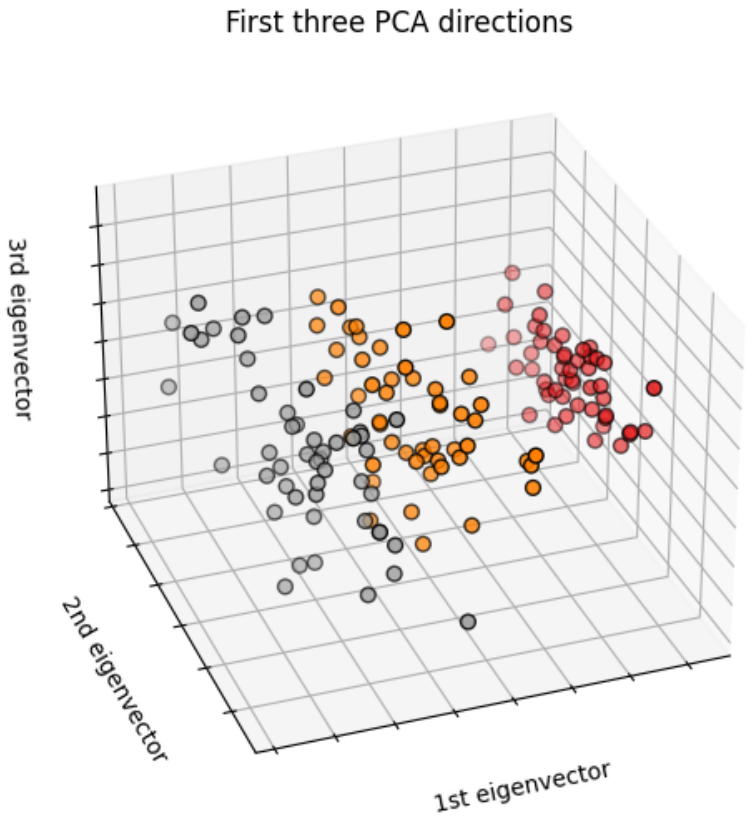
```
for i in range(len(centroids)):
    point_set = []
    for index, p in enumerate(result_list):
        if p == i:
            point_set.append(data[index])

    centroids[i] = np.mean(point_set,axis = 0)
    #print("centroids[i]: ", centroids[i])
```

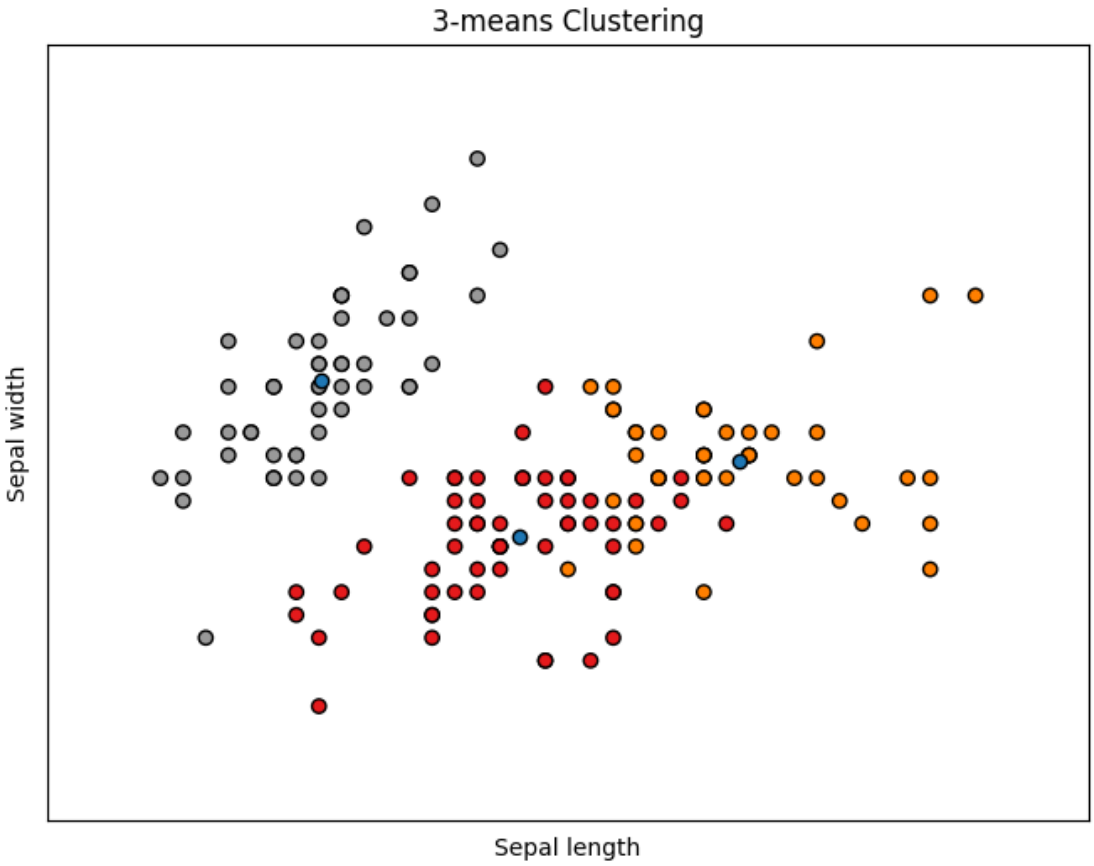
- **K-means clustering results:**

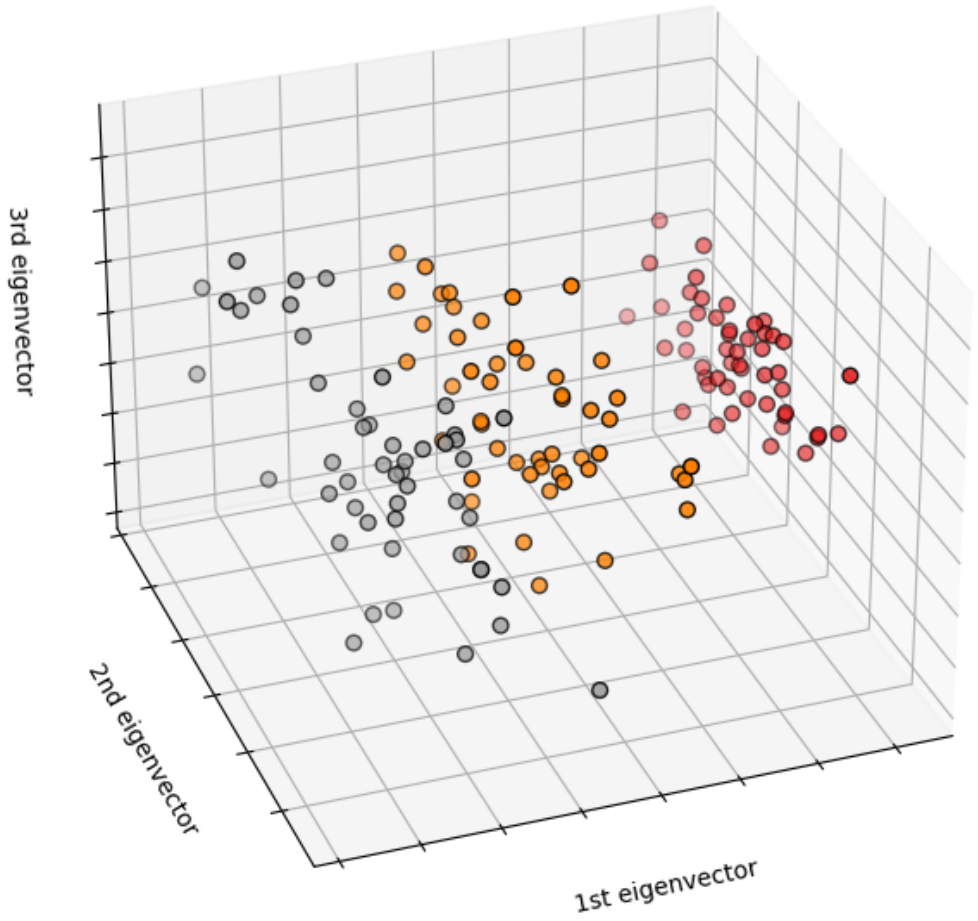
The original data set:



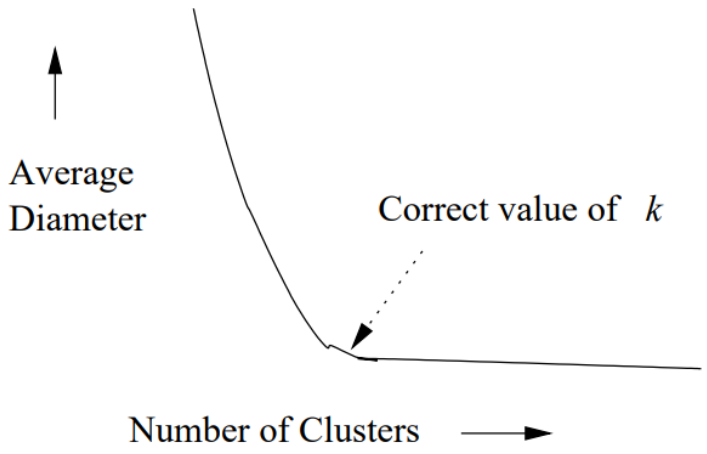


3-means clustering:


3-means Clustering
Sepal width
Sepal length



- **Discussion on how to pick the right value of K:**



As the above figure shows, if we take a measure of correctness for clusters, such as the y-axis value shown in the above figure, average diameter or radius, that value will grow slowly as long as the number of clusters we measure remains at or above the true number of clusters, otherwise, if the number of clusters we measure is less than there are, the measure will rise drastically.

One way to determine the value of K is to repeatedly run the clustering operation by using different K values. For example: set the K value to be 1, 2, 4, 8, 16... Eventually, there will be two numbers v and 2v between which you will find there is very little decrease in the average diameter or radius. Then we can conclude that the k value is between v and 2v. After that, you can use binary search to recursively narrow the range of K until you find the best value of K. The binary search works as follow: If we somehow know how to calculate the change of average diameter by some formula, and we know that there is too much change between v/2 and v, or else we would not have gone on to run a clustering for 2v clusters. Suppose we have narrowed the range of K to between a and b. Let c = (a+b)/2. We can then know the true value of k must lie between c and an if there is not too much change between c and b by running a clustering with c as the target number of clusters. In this way, we can recursively narrow the range of K down in log2v clustering operations.