Name: Chenyang Yu SUID: 862052273

Question 1&2. Implement Lloyd's algorithm to cluster data points from Iris data set.

First step is read data from iris.data and randomly choose k of them as initial centroids. In order to represent data more conveniently, I store the label as 0/1/2, represented Iris-setosa/ Iris-versicolor/ Iris-virginica respectively. The sample is

```
context = data[0:4]
label = data[4][:len(data[4])-1]
if label == 'Iris-setosa':
    context.append(0)
elif label == 'Iris-versicolor':
    context.append(1)
elif label == 'Iris-virginica':
    context.append(2)
else:
    print("in reading file, unexpected label")
attr_target_list.append(context)
```

The sample is done by python random library, I sample k number from 150 (the size of data point size) and store the data point according to index. Notice that there is possibility that we choose the same data point (with same attributes). So in check_init() we examine if there is any duplicate point. Re-initialize if there is.

```
init_cent_index = random.sample(range(len(attr_target_list)), k)
init_centroids = []
for i in range(k):
    init_centroids.append(attr_target_list[init_cent_index[i]])
# print("init_centroids:",init_centroids)
while check_init(init_centroids, k):
    init_cent_index = random.sample(range(len(attr_target_list)), k)
    init_centroids = []
    for i in range(k):
        init_centroids.append(attr_target_list[init_cent_index[i]])
```

Next part is the main program of k-means. Starting from k initial centroids, each centroids represents one cluster. We assign each data point into different cluster depending on their distance: For a given data point, calculating the distance between thie point and all k centroids. Choose the minimum distance and change the label of this given data point. As following

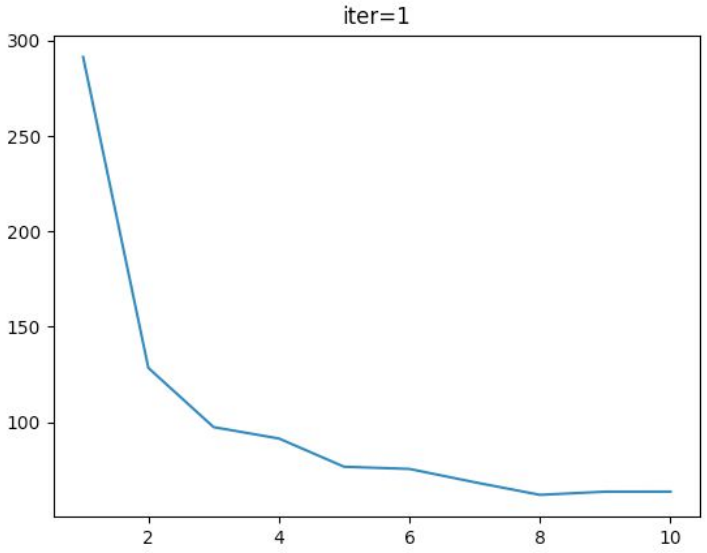
```
def assign(x_input, centroids):
    k = len(centroids)
    # initial each label as -1
    label = [-1]*len(x_input)
    for i in range(len(x_input)):
        dist = [-1]*k
        # calculate the distance to all centroids
        for j in range(k):
            dist[j] = distance(x_input[i], centroids[j], 2)
        # choose the minimum one as the label
        label[i] = dist.index(min(dist))
    # return list of label
    return label
```

The new centroids will be re-calculated by average all data points belongs to the same cluster. Use the new centroids to partition all data point until the label remains the same after we partition them. **label[]** is used for the updated cluster record and **cluster_assignments[]** is used for the previous cluster record.

```
    # first time cluster assign
    label = assign(x_input, centroids)
    while cluster_assignments != label:
        iter +=1
        cluster_assignments = label
        # clean centroids
        # centroids[] to accumulate the attribute of same cluster,
        centroids=[ [0] * len(x_input[0]) for i in range(k) ]
        # count[] to accumulate of different cluster, initialize as [0,0,0,0,0,...]
        count = [0]*k
        # update centroids[] and count[]
        for i in range(len(label)):
            count[label[i]] += 1
            for s in range(len(centroids[0])):
                centroids[label[i]][s] += float(x_input[i][s])
        # calculate the new centroids
        for i in range(len(centroids)):
            for s in range(len(centroids[i])):
                centroids[i][s] = centroids[i][s]/count[i]
        # update cluster_assignments
        label = assign(x_input, centroids)
    print("iterator:",iter)
    return cluster_assignments,centroids
```

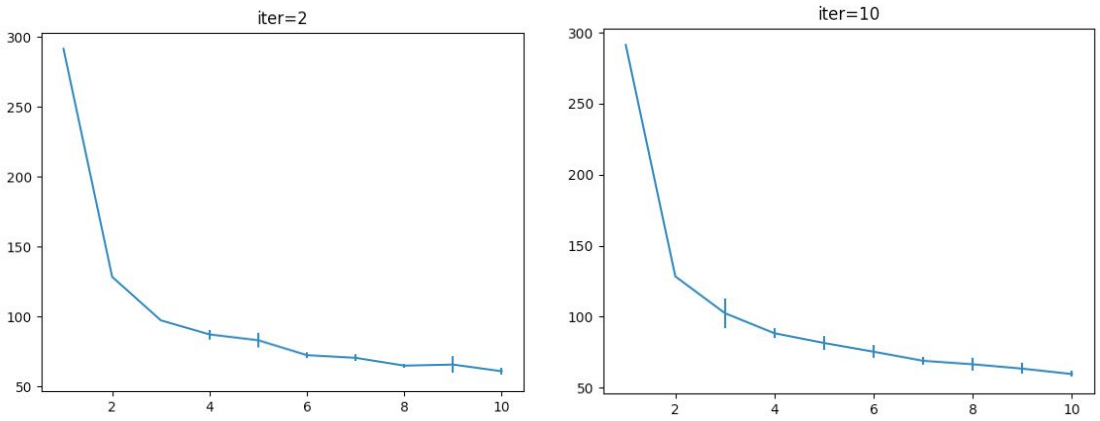
To see the how does the number of initial centroids affect our result, we change the initial centroids k form 1 to 10 and calculate the sum of square error. The bigger this value, the more diverse for each cluster. The result is represented as knee plot. Although there will be some differences between different trials (since the initial centroids is chosen by random, it is possible we choose a set of initial centroids that ends up in a centroid more close to egd), we still can observe the knee is at 2 and 3. It fits the original iris data set, which contains 3 classes.

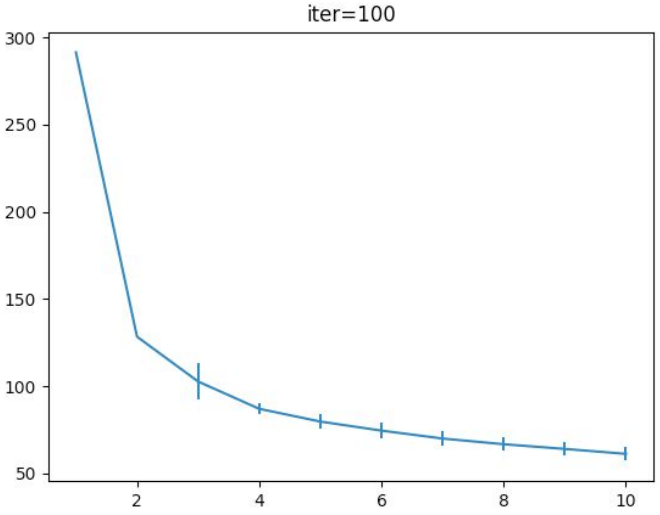
Result:



In order the learn how the randomness affects our sum of square error result, we apply sensitivity analysis here. By repeat the k-means algorithm multiple times and record them, we can use mean and standard deviation to draw an error plot. It tells the dispersion of results between different trail. As we can see, the knee is still obvious and the standard deviation is rather smaller, so the result is stable.

Result:



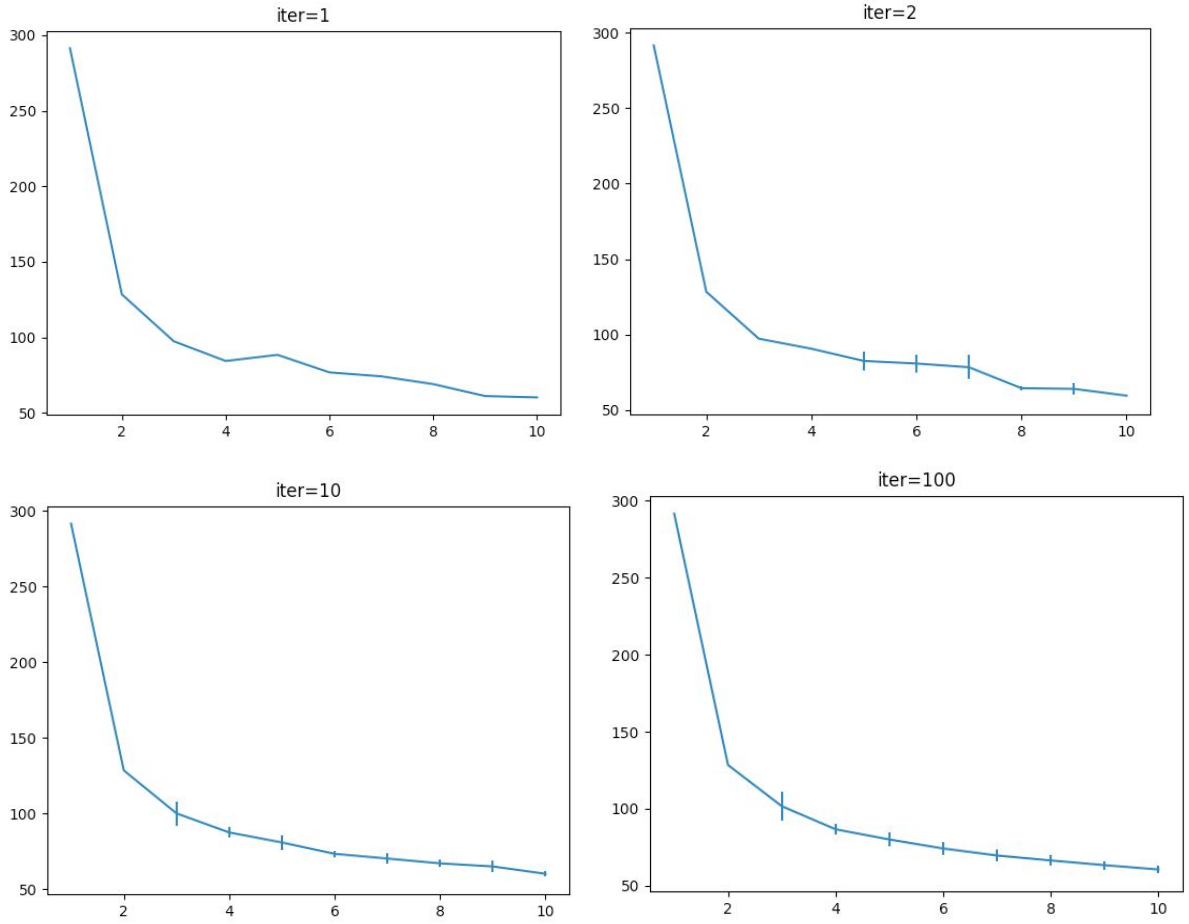


Question 3. K-means ++

We use k-means ++ algorithm to replace the random initialization. The first centroid is still randomly pick (so if k =1, k-means ++ is the same as random initial). For the rest of centroids, we choose from the data point based on the distance between data point and the chosen centroids. We can achieve this by **random.choice(data_point, weight)**. For each round, we assign a distance for each point and use it as weight. The more distance leads to more weight, which increase the possibility to be chosen.

```
dist = [0]*len(x_input)
# calculate distance for each data point (skip the point already chosen)
for j in range(len(x_input)):
    if j not in init_cent_index:
        dist_tmp = 0
        # consider all chosen centroids and pick the largest one
        for s in range(len(init_cent_index)):
            dist_tmp = max( dist_tmp, distance(x_input[j], x_input[init_cent_index[s]], 2) )
        dist[j] = dist_tmp
# choose one sample depending on the weight (proportional to distance)
init_cent_index.extend(random.choices(range(len(x_input)),dist))
```

The same, we use knee-plot with different repeat time (max_iter = 2,10,100).
Result:

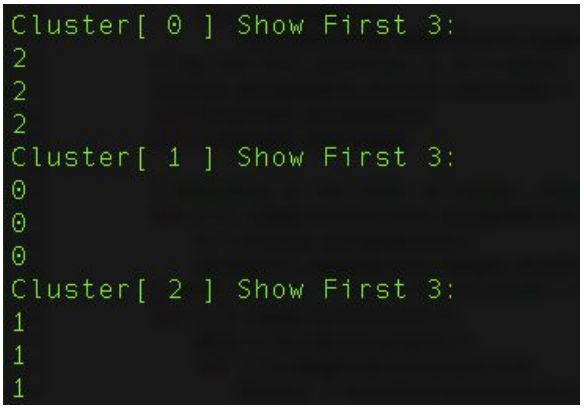


Question 4. Top data points

Since we actually know what the data points are exactly. When we get the final cluster, we can check the label and see if we partition two different data point into the same cluster. Here we use k=3 and output the three data point which is most closest to the final controids.

```
# depending on the label we assign, insert them into cluster[0]/ cluster[1]/cluster[2]
for i in range(len(cluster_assignments)):
    c = cluster_assignments[i]
    cluster[c].append(attr_target_list[i])
# check the top 3 point in cluster[0]/ cluster[1]/cluster[2]
for i in range(len(cluster)):
    dist = [0]*len(cluster[i])
    for j in range(len(cluster[i])):
        dist[j] = distance(cluster[i][j], cluster_centroids[i], 2)
    # print("dist ",i," = ",dist)
    index = sorted(range(len(dist)), key=lambda _k: dist[_k])
    print("Show First 3:")
    for m in range(3):
            print(cluster[i][index[m]][4])
```

Result:


Cluster[0] Show First 3:
2
2
2
Cluster[1] Show First 3:
0
0
0
Cluster[2] Show First 3:
1
1
1