Assignment-3

1 CS171 - Winter 2022

1.0.1 Instructor: Vagelis Papalexakis

In this assignment we will implement the K-means clustering algorithm. We are going to use the same dataset as in the previous two assignments (Note: make sure you copy the dataset from Assignment 1 to the folder of this assignment!).

[1]:
```
import numpy as np
import math
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sb
import random as rand
from sklearn.model_selection import train_test_split


data_names = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width',␣
 ↪'label']
data = pd.read_csv('iris.data',
                   names = data_names)
```

1.1 Question 1: Implementing and testing K-means clustering [100%]

1.1.1 Question 1a: Implementing K-Means clustering [50%]

In this question you should implement a function that performs k-means clustering, using the Euclidean distance (you may use Numpy libraries for the distance computation). For calculation of the centroid you should use the 'mean' function.

For uniformity, you should implement a function with the following specifications:

```
def kmeans_clustering(all_vals,K,max_iter = 100, tol = pow(10,-3) ):
```

where 1) 'all_vals' is the $N \times M$ matrix that contains all data points (N is the number of data points and M is the number of features, each row of the matrix is a data point), 2) 'K' is the number of clusters, 3) 'max_iter' is the maxium number of iterations, and 4) 'tol' is the tolerance for the change of the sum of squares of errors that determines convergence.

Your function should return the following variables: 1) 'assignments': this is a $N \times 1$ vector (where

N is the number of data points) where the i-th position of that vector contains the cluster number that the i-th data point is assigned to, 2) 'centroids': this is a $K \times M$ matrix, each row of which contains the centroid for every cluster, 3) 'all_sse': this is a vector that contains all the sum of squares of errors per iteration of the algorithm, and 4) 'iters': this is the number of iterations that the algorithm ran.

Here we are going to implement the simplest version of K-means, where the initial centroids are chosen entirely at random among all the data points.

As we saw in class, the K-means algorithm iterates over the following steps: - Given a set of centroids, assign all data points to the cluster represented by its nearest centroid (according to Euclidean distance) - Given a set of assignments of points to clusters, compute the new centroids for every cluster, by taking the mean of all the points assigned to each cluster.

Your algorithm should converge if 1) the maximum number of iterations is reached, or 2) if the SSE between two consecutive iterations does not change a lot (as in the gradient descent for linear regression we saw in Assignment 2). In order to check for the latter condition, you may use the following piece of code:

```
if np.absolute(all_sse[it] - all_sse[it-1])/all_sse[it-1] <= tol
```

In order to calculate the SSE (sum of squares of error) first you need to define what an 'error' is. In k-means, error per data point refers to the Euclidean distance of that particular point from its assigned centroid. SSE sums up all those squared Euclidean distances for all data points and comes up with a number that reflects the total error of approximating every data points by its assigned centroid.

[2]:
```
def remove_labels(d):

    ##removing the last column which contains the labels
    new_d = np.array(d)
    new_d = new_d[:,:-1]

    return new_d


def kmeans_clustering(all_vals,K,max_iter, tol):

    #initialize data
    centroids = []
    ##assignments = []
    all_sse=[]
    iters=0
    error,ssqe= 0,0

    #initializng the centroids for each cluster, K x M (3 x 4) matrix with␣
 ↪random numbers from datapoints

    rand_centroid = all_vals[np.random.randint(0, len(all_vals)-1, size=K)]
```

```
    centroids = rand_centroid
#     print("random centroids for first iteration: ", centroids,"\n")
#     print("all val", all_vals)

    for iters in range(max_iter):

        assignments = []

        for i in range(len(all_vals)):
            #datapoint is closer to the first clusterpoint
            if ((np.linalg.norm(centroids[0][0:4] - all_vals[i][0:4]) <
                np.linalg.norm(centroids[1][0:4] - all_vals[i][0:4])) and
               (np.linalg.norm(centroids[0][0:4] - all_vals[i][0:4]) <
                np.linalg.norm(centroids[2][0:4] - all_vals[i][0:4]))):

                error=np.linalg.norm(centroids[0][0:4] - all_vals[i][0:4])
                assignments.append(0)


            #datapoint is closer to the second clusterpoint
            elif((np.linalg.norm(centroids[1][0:4] - all_vals[i][0:4]) <
                np.linalg.norm(centroids[0][0:4] - all_vals[i][0:4])) and
               (np.linalg.norm(centroids[1][0:4] - all_vals[i][0:4]) <
                np.linalg.norm(centroids[2][0:4] - all_vals[i][0:4]))):

                error=np.linalg.norm(centroids[1][0:4] - all_vals[i][0:4])
                assignments.append(1)

            #datapoint is closer to the third clusterpoint
            else:
                error=np.linalg.norm(centroids[2][0:4] - all_vals[i][0:4])
                assignments.append(2)

            ##computing sum of sse
            error=pow(error,2)
            ssqe+=error

        all_sse.append(ssqe)

        ##computing new centroid
        f1,f2,f3,f4=0,0,0,0
        f1_arr_k1,f2_arr_k1,f3_arr_k1,f4_arr_k1=[],[],[],[]
        f1_arr_k2,f2_arr_k2,f3_arr_k2,f4_arr_k2=[],[],[],[]
        f1_arr_k3,f2_arr_k3,f3_arr_k3,f4_arr_k3=[],[],[],[]

        c_0_0,c_0_1,c_0_2,c_0_3=0,0,0,0
        c_1_0,c_1_1,c_1_2,c_1_3=0,0,0,0
```

```
    c_2_0,c_2_1,c_2_2,c_2_3=0,0,0,0

    for j in range(len(assignments)):
        if assignments[j]==0:
            f1=all_vals[j][0]
            f1_arr_k1.append(f1)
            f2=all_vals[j][1]
            f2_arr_k1.append(f2)
            f3=all_vals[j][2]
            f3_arr_k1.append(f3)
            f4=all_vals[j][3]
            f4_arr_k1.append(f4)
        elif assignments[j]==1:
            f1=all_vals[j][0]
            f1_arr_k2.append(f1)
            f2=all_vals[j][1]
            f2_arr_k2.append(f2)
            f3=all_vals[j][2]
            f3_arr_k2.append(f3)
            f4=all_vals[j][3]
            f4_arr_k2.append(f4)
        else:
            f1=all_vals[j][0]
            f1_arr_k3.append(f1)
            f2=all_vals[j][1]
            f2_arr_k3.append(f2)
            f3=all_vals[j][2]
            f3_arr_k3.append(f3)
            f4=all_vals[j][3]
            f4_arr_k3.append(f4)

    ##taking the mean of all the points assigned to each cluster

    c_0_0=np.mean(f1_arr_k1)
    c_0_1=np.mean(f2_arr_k1)
    c_0_2=np.mean(f3_arr_k1)
    c_0_3=np.mean(f4_arr_k1)

    c_1_0=np.mean(f1_arr_k2)
    c_1_1=np.mean(f2_arr_k2)
    c_1_2=np.mean(f3_arr_k2)
    c_1_3=np.mean(f4_arr_k2)

    c_2_0=np.mean(f1_arr_k3)
    c_2_1=np.mean(f2_arr_k3)
    c_2_2=np.mean(f3_arr_k3)
    c_2_3=np.mean(f4_arr_k3)
```

```
        new_c=[[c_0_0,c_0_1,c_0_2,c_0_3],
               [c_1_0,c_1_1,c_1_2,c_1_3],
               [c_2_0,c_2_1,c_2_2,c_2_3]]

        ##print("new c: ", new_c)
        centroids=new_c

        ##converge if this condition is true
        if(iters>=1):
            if(np.absolute(all_sse[iters] - all_sse[iters-1])/all_sse[iters-1]␣
↪<= tol):
                break

    return assignments,iters,centroids,all_sse




new_data=remove_labels(data)
assignment,iters,centroids,all_sse=kmeans_clustering(new_data,3,100, pow(10,-3))
print("assignment array for last iteration: ", assignment,"\n")
print("centroids for last iteration: ", centroids,"\n")
print("number of iterations for convergence: ", iters+1,"\n")
print("all sse array: ", all_sse)
```

```
assignment array for last iteration:  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 2, 0, 2, 2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 0, 0, 2, 2, 2, 2, 0, 2, 0,
2, 0, 2, 2, 0, 0, 2, 2, 2, 2, 2, 0, 2, 2, 2, 2, 0, 2, 2, 2, 0, 2, 2, 2, 0, 2, 2,
0]

centroids for last iteration:  [[5.88360655737705, 2.7409836065573767,
4.388524590163935, 1.4344262295081966], [5.006, 3.418, 1.464, 0.244],
[6.853846153846154, 3.076923076923077, 5.715384615384615, 2.0538461538461537]]

number of iterations for convergence:  100

all sse array:  [119.37000000000006, 201.70137226813605, 282.8717064090595,
362.83468624367254, 442.2684503889993, 521.2791601112215, 600.224225937199,
679.1692917631765, 758.114357589154, 837.0594234151315, 916.004489241109,
994.9495550670865, 1073.8946208930633, 1152.8396867190406, 1231.784752545018,
1310.7298183709952, 1389.6748841969725, 1468.6199500229498, 1547.565015848927,
1626.5100816749043, 1705.4551475008816, 1784.4002133268589, 1863.3452791528362,
1942.2903449788134, 2021.2354108047907, 2100.180476630766, 2179.1255424567407,
```

```
2258.0706082827155, 2337.0156741086903, 2415.960739934665, 2494.90580576064,
2573.8508715866146, 2652.7959374125894, 2731.741003238564, 2810.686069064539,
2889.6311348905137, 2968.5762007164885, 3047.5212665424633, 3126.466332368438,
3205.411398194413, 3284.3564640203876, 3363.3015298463624, 3442.246595672337,
3521.191661498312, 3600.1367273242868, 3679.0817931502615, 3758.0268589762363,
3836.971924802211, 3915.916990628186, 3994.8620564541607, 4073.8071222801354,
4152.752188106115, 4231.697253932094, 4310.642319758073, 4389.587385584053,
4468.532451410032, 4547.477517236011, 4626.422583061991, 4705.36764888797,
4784.312714713949, 4863.257780539929, 4942.202846365908, 5021.147912191887,
5100.092978017867, 5179.038043843846, 5257.983109669825, 5336.928175495805,
5415.873241321784, 5494.818307147763, 5573.763372973743, 5652.708438799722,
5731.653504625701, 5810.598570451681, 5889.54363627766, 5968.488702103639,
6047.433767929619, 6126.378833755598, 6205.323899581577, 6284.268965407557,
6363.214031233536, 6442.159097059515, 6521.104162885495, 6600.049228711474,
6678.994294537453, 6757.939360363433, 6836.884426189412, 6915.829492015391,
6994.774557841371, 7073.71962366735, 7152.664689493329, 7231.6097553193085,
7310.554821145288, 7389.499886971267, 7468.4449527972465, 7547.390018623226,
7626.335084449205, 7705.2801502751845, 7784.225216101164, 7863.170281927143,
7942.1153477531225]
```

1.1.2 Question 1b: Visualizing K-means [10%]

In this question we wll visualize the result of the K-means algorithm. For ease of visualization, we will focus on a scatterplot of two of the four features of the Iris dataset. In particular: run your K-means code with K=3 and default values for the rest of the inputs. Subsequently, make a single scatterplot that contains all data points of the dataset for features 'sepal_length' and 'petal_length' and color every data point according to its cluster assignment.

[3]:
```
#your code here: changed code to only compute for 3 clusters and 2 feautres␣
↪'sepal_length' and 'petal_length'
def remove_labels(d):

    ##removing the last column which contains the labels
    new_d = np.array(d)
    new_d = new_d[:,:-1]

    return new_d


def kmeans_clusteringv3(all_vals,K,max_iter, tol):

    #initialize data
    centroids = []
    ##assignments = []
    all_sse=[]
    iters=0
    error,ssqe= 0,0
```

```
    #initializng the centroids for each cluster, K x M (3 x 2) matrix with␣
 ↪random numbers from datapoints

    rand_centroid = all_vals[np.random.randint(0, len(all_vals)-1, size=K)]

    centroids = rand_centroid
#     print("random centroids for first iteration: ", centroids,"\n")

    ##print("all_val",all_vals)
    for iters in range(max_iter):

        assignments = []

        for i in range(len(all_vals)):
            #datapoint is closer to the first clusterpoint
            if ((np.linalg.norm(centroids[0][0:2] - all_vals[i][0:2]) <
                np.linalg.norm(centroids[1][0:2] - all_vals[i][0:2])) and
               (np.linalg.norm(centroids[0][0:2] - all_vals[i][0:2]) <
                np.linalg.norm(centroids[2][0:2] - all_vals[i][0:2]))):

                error=np.linalg.norm(centroids[0][0:2] - all_vals[i][0:2])
                assignments.append(0)


            #datapoint is closer to the second clusterpoint
            elif((np.linalg.norm(centroids[1][0:2] - all_vals[i][0:2]) <
                np.linalg.norm(centroids[0][0:2] - all_vals[i][0:2])) and
               (np.linalg.norm(centroids[1][0:2] - all_vals[i][0:2]) <
                np.linalg.norm(centroids[2][0:2] - all_vals[i][0:2]))):

                error=np.linalg.norm(centroids[1][0:2] - all_vals[i][0:2])
                assignments.append(1)

            #datapoint is closer to the third clusterpoint
            else:
                error=np.linalg.norm(centroids[2][0:2] - all_vals[i][0:2])
                assignments.append(2)

            ##computing sum of sse
            error=pow(error,2)
            ssqe+=error

        all_sse.append(ssqe)

        ##computing new centroid
        f1,f2,f3,f4=0,0,0,0
        f1_arr_k1,f2_arr_k1=[],[]
```

```
        f1_arr_k2,f2_arr_k2=[],[]
        f1_arr_k3,f2_arr_k3=[],[]

        c_0_0,c_0_1=0,0
        c_1_0,c_1_1,=0,0
        c_2_0,c_2_1=0,0

        for j in range(len(assignments)):
            if assignments[j]==0:
                f1=all_vals[j][0]
                f1_arr_k1.append(f1)
                f2=all_vals[j][1]
                f2_arr_k1.append(f2)
            elif assignments[j]==1:
                f1=all_vals[j][0]
                f1_arr_k2.append(f1)
                f2=all_vals[j][1]
                f2_arr_k2.append(f2)
            else:
                f1=all_vals[j][0]
                f1_arr_k3.append(f1)
                f2=all_vals[j][1]
                f2_arr_k3.append(f2)

        ##taking the mean of all the points assigned to each cluster

        c_0_0=np.mean(f1_arr_k1)
        c_0_1=np.mean(f2_arr_k1)


        c_1_0=np.mean(f1_arr_k2)
        c_1_1=np.mean(f2_arr_k2)


        c_2_0=np.mean(f1_arr_k3)
        c_2_1=np.mean(f2_arr_k3)


        new_c=[[c_0_0,c_0_1],
               [c_1_0,c_1_1],
               [c_2_0,c_2_1]]

        ##print("new c: ", new_c)
        centroids=new_c

        ##conberge if this condition is true
        if(iters>=1):
```

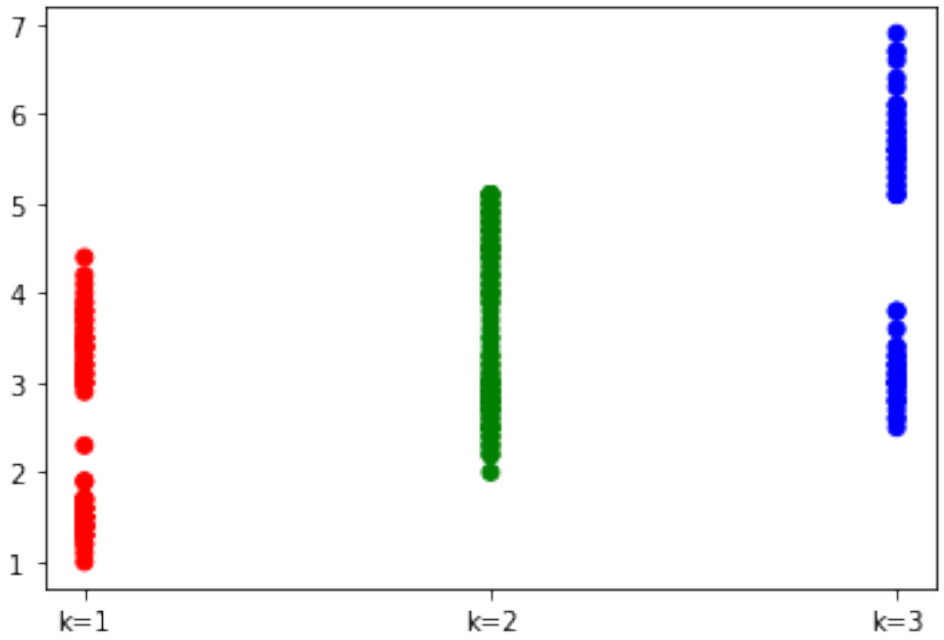
```
            if(np.absolute(all_sse[iters] - all_sse[iters-1])/all_sse[iters-1]␣
 ↪<= tol):
                break

    return assignments,iters,centroids,all_sse




petal_vs_sepal = data[['sepal_width', 'petal_length']].values
# print(petal_vs_sepal)
##new_data=remove_labels(petal_vs_sepal)
assignment1b,iters,centroids,all_sse=kmeans_clusteringv3(petal_vs_sepal,3,100,␣
 ↪pow(10,-3))
print("assignment array for last iteration: ", assignment1b,"\n")
# print("centroids for last iteration: ", centroids,"\n")
# print("number of iterations for convergence: ", iters+1,"\n")
# print("all sse array: ", all_sse)
```

```
assignment array for last iteration:  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 2, 1, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1, 1, 2, 2, 2, 2, 1, 2, 1,
2, 1, 2, 2, 1, 1, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 1, 2, 2, 2, 1, 2, 2, 2, 1, 2, 2,
2]
```

[11]:
```
fig1,kplot1 = plt.subplots()

##_k = [x for x in range(1,4)]

categories=assignment1b
colormap=np.array(['r', 'g', 'b'])

d1_cluster_list,d1_list,d2_cluster_list,d2_list=[],[],[],[]

for c in range(len(assignment1b)):
    if assignment1b[c]==0:
        cluster='k=1'
        d1=petal_vs_sepal[c][0]
        d2=petal_vs_sepal[c][1]
    elif assignment1b[c]==1:
        cluster='k=2'
        d1=petal_vs_sepal[c][0]
        d2=petal_vs_sepal[c][1]
    else:
```

```
        cluster='k=3'
        d1=petal_vs_sepal[c][0]
        d2=petal_vs_sepal[c][1]

    d1_cluster_list.append(cluster)
    d1_list.append(d1)
    d2_cluster_list.append(cluster)
    d2_list.append(d2)

kplot1.scatter(d1_cluster_list, d1_list, c=colormap[categories])
kplot1.scatter(d2_cluster_list, d2_list, c=colormap[categories])
print("first cluster k=1: red, second cluster k=2: green , third cluster k=3:␣
 ↪blue")
```

```
first cluster k=1: red, second cluster k=2: green , third cluster k=3: blue
```



1.1.3 Question 1c: Testing K-means [40%]

Selecting the right number of clusters K is a very challenging problem, especially when we don't have some side-information or domain expertise that can help us narrow down a few reasonable values for that parameter.

In the absence of any other information, a very useful exercise is to create the plot of SSE (sum of squares of errors) as a function of K. Ideally, for a very small K, the error will be high (since we are trying to approximate a whole lot of points with a very small number of centroids) and as K increases, the error decreases. However, after a certain value (or a couple of values) for K, we will notice diminishing returns, i.e., the error will be decreasing, but not to a great degree. Typically,

the value(s) for K where this behavior is observed (the threshold point after which we observe diminishing returns) is usually a good guess for the number of clusters.

In this question, we will have to create the SSE vs. K plot for $K = 1 \cdots 10$. Furthermore, because K-means uses randomized initialization, we need to do a number of iterations per value of K in order to get a good estimate of the actual SSE (which may not be caused by randomness in the initialization). For this question, you will have to run the entire K-means algorithm to completion, and repeat it 50 different times per K, and collect all SSEs. In the figure, you should report the mean SSE per K, surrounded by error-bars which will encode the standard deviation.

[5]:
```
#your code here: changed code so number of K could be changed
def remove_labels(d):

    ##removing the last column which contains the labels
    new_d = np.array(d)
    new_d = new_d[:,:-1]
    return new_d


def kmeans_clusteringv2(all_vals,K,max_iter, tol):

    #initialize data
    centroids = []
    all_sse=[]
    iters=0
    error,ssqe= 0,0

    #initializng the centroids for each cluster, K x M (3 x 4) matrix with␣
 ↪random numbers from datapoints
    rand_centroid = all_vals[np.random.randint(0, len(all_vals)-1, size=K)]
    centroids = rand_centroid
#     print("centroids: \n", centroids)


    for iters in range(max_iter):

        assignments = []

        for i in range(len(all_vals)):
            _min = math.inf
            c=-1
            for k in range(0,K):
                if (_min > np.linalg.norm(centroids[k][0:4] - all_vals[i][0:4])):
                    _min=np.linalg.norm(centroids[k][0:4] - all_vals[i][0:4])
                    c=k
                    error=_min

            error=pow(error,2)
```

```
            ssqe+=error
            assignments.append(c)

        all_sse.append(ssqe)


        ##computing new centroid
        f1,f2,f3,f4=0,0,0,0
        list_k=[k for k in range(K)]

        ##declaring dynamic variable names since K changes everytime the␣
↪function is called
        for g in range(K):
            globals()[f"f1_arr_k{g}"] = []
            globals()[f"f2_arr_k{g}"] = []
            globals()[f"f3_arr_k{g}"] = []
            globals()[f"f4_arr_k{g}"] = []
            globals()[f"c_k{g}_0"] = 0
            globals()[f"c_k{g}_1"] = 0
            globals()[f"c_k{g}_2"] = 0
            globals()[f"c_k{g}_3"] = 0

        for j in range(len(assignments)):

            if assignments[j] in list_k:
                f1=all_vals[j][0]
                f2=all_vals[j][1]
                f3=all_vals[j][2]
                f4=all_vals[j][3]
                globals()[f"f1_arr_k{assignments[j]}"].append(f1)
                globals()[f"f2_arr_k{assignments[j]}"].append(f2)
                globals()[f"f3_arr_k{assignments[j]}"].append(f3)
                globals()[f"f4_arr_k{assignments[j]}"].append(f4)

        new_c=[]

        ##computing the mean of all points assigned to each cluster
        for k in range(K):

#             print("#", k)

            globals()[f"c_k{k}_0"]=np.mean(globals()[f"f1_arr_k{k}"])
            globals()[f"c_k{k}_1"]=np.mean(globals()[f"f2_arr_k{k}"])
            globals()[f"c_k{k}_2"]=np.mean(globals()[f"f3_arr_k{k}"])
            globals()[f"c_k{k}_3"]=np.mean(globals()[f"f4_arr_k{k}"])

#             print("c", float(globals()[f"c_k{k}_0"]), "type")
```

```
#             print("c", float(globals()[f"c_k{k}_1"]), "type")
#             print("c", float(globals()[f"c_k{k}_2"]), "type")
#             print("c", float(globals()[f"c_k{k}_3"]), "type")

            ##computing each row of the new centroid/each cluster
            c_row=[]
            c_row.append(float(globals()[f"c_k{k}_0"]))
            c_row.append(float(globals()[f"c_k{k}_1"]))
            c_row.append(float(globals()[f"c_k{k}_2"]))
            c_row.append(float(globals()[f"c_k{k}_3"]))

            ##print("c row", c_row)

            ##adding each row to the new centroid matrix
            new_c.append(c_row)


        ##print("new c: ", new_c)
        ##updating the new centroid
        centroids=new_c

        if(iters>=1):
            if(np.absolute(all_sse[iters] - all_sse[iters-1])/all_sse[iters-1]␣
 ↪<= tol):
                break


    return assignments,centroids,all_sse
```

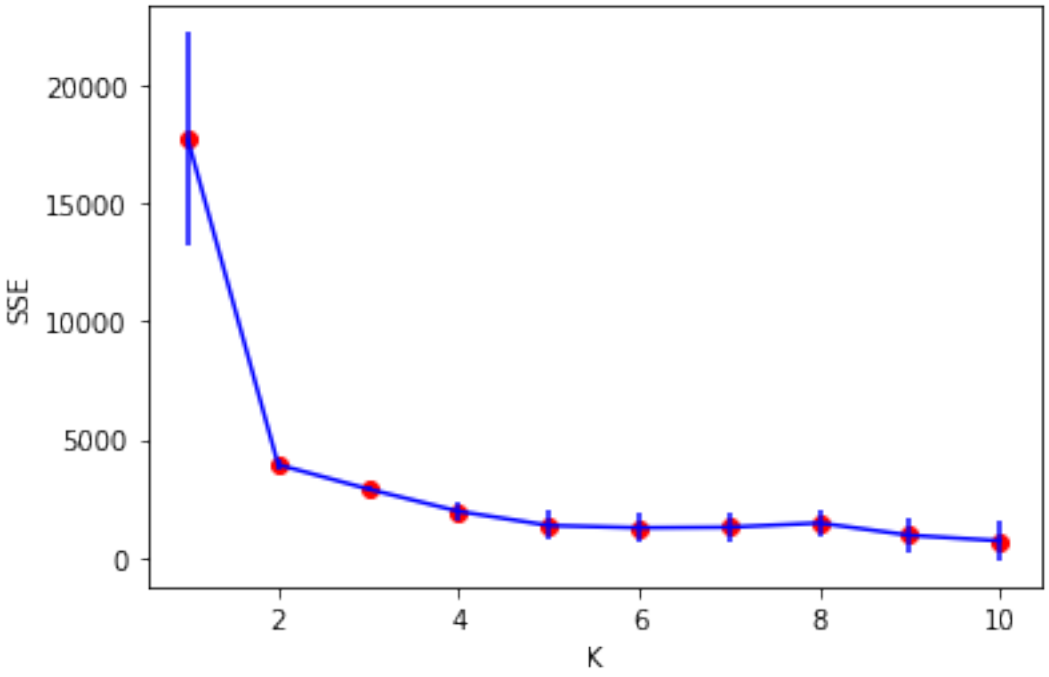
[7]:
```
all_msse_arr=[]
new_data=remove_labels(data)
##print(new_data)
assignment_k1,centroids_k1,all_sse_k1=kmeans_clusteringv2(new_data,1,50,␣
 ↪pow(10,-3))
# print("all sse array: ", all_sse_k1)
mean_sse_k1=np.mean(all_sse_k1)
all_msse_arr.append(mean_sse_k1)

assignment_k2,centroids_k2,all_sse_k2=kmeans_clusteringv2(new_data,2,50,␣
 ↪pow(10,-3))
# print("all sse array: ", all_sse_k2)
mean_sse_k2=np.mean(all_sse_k2)
all_msse_arr.append(mean_sse_k2)

assignment_k3,centroids_k3,all_sse_k3=kmeans_clusteringv2(new_data,3,50,␣
 ↪pow(10,-3))
# print("all sse array: ", all_sse_k3)
```

```
mean_sse_k3=np.mean(all_sse_k3)
all_msse_arr.append(mean_sse_k3)

assignment_k4,centroids_k4,all_sse_k4=kmeans_clusteringv2(new_data,4,50,␣
↪pow(10,-3))
# print("all sse array: ", all_sse_k4)
mean_sse_k4=np.mean(all_sse_k4)
all_msse_arr.append(mean_sse_k4)

assignment_k5,centroids_k5,all_sse_k5=kmeans_clusteringv2(new_data,5,50,␣
↪pow(10,-3))
# print("all sse array: ", all_sse_k5)
mean_sse_k5=np.mean(all_sse_k5)
all_msse_arr.append(mean_sse_k5)

assignment_k6,centroids_k6,all_sse_k6=kmeans_clusteringv2(new_data,6,50,␣
↪pow(10,-3))
# print("all sse array: ", all_sse_k6)
mean_sse_k6=np.mean(all_sse_k6)
all_msse_arr.append(mean_sse_k6)

assignment_k7,centroids_k7,all_sse_k7=kmeans_clusteringv2(new_data,7,50,␣
↪pow(10,-3))
# print("all sse array: ", all_sse_k5)
mean_sse_k7=np.mean(all_sse_k7)
all_msse_arr.append(mean_sse_k7)

assignment_k8,centroids_k8,all_sse_k8=kmeans_clusteringv2(new_data,8,50,␣
↪pow(10,-3))
# print("all sse array: ", all_sse_k8)
mean_sse_k8=np.mean(all_sse_k8)
all_msse_arr.append(mean_sse_k8)

assignment_k9,centroids_k9,all_sse_k9=kmeans_clusteringv2(new_data,9,50,␣
↪pow(10,-3))
# print("all sse array: ", all_sse_k9)
mean_sse_k9=np.mean(all_sse_k9)
all_msse_arr.append(mean_sse_k9)

assignment_k10,centroids_k10,all_sse_k10=kmeans_clusteringv2(new_data,10,50,␣
↪pow(10,-3))
# print("all sse array: ", all_sse_k10)
mean_sse_k10=np.mean(all_sse_k10)
all_msse_arr.append(mean_sse_k10)

print(all_msse_arr)
```

```
[17685.47780000008, 3926.6652479654267, 2895.228347418268, 1951.4806189142712,
1354.325878364995, 1257.1546578796958, 1280.0577206377914, 1456.9465562836517,
938.6994771468194, 704.6408013506516]
```

[8]:
```
avg_=0
avg_ = np.mean(all_msse_arr)
#deviation_total = 0
#computing standard deviation for each k 1,10
stdev = [0 for x in range(10)]
for i in range(10):
    deviation_total = 0
    deviation_total += (avg_ - all_msse_arr[i])**2
    deviation = deviation_total/len(stdev)
    stdev[i] = (deviation)**(1/2)
##print("devvv", stdev)
```

[9]:
```
fig,kplot = plt.subplots()

all_k = [x for x in range(1,11)]
##print(all_k)

#plot of k vs avg accuracy
kplot.scatter(all_k , all_msse_arr,color='r')

#using plt.errorbar() method to show error-bar around the points that encode the␣
 ↪standard deviation
kplot.errorbar(all_k , all_msse_arr, yerr=stdev, color='b')
#plot labels
plt.ylabel("SSE")
plt.xlabel("K")
plt.show()
```


20000
15000
10000
5000
0
SSE
2
4
6
8
10
K