iris_playground_p3

January 4, 2021

1 Iris Playground - Part 2

1.0.1 Instructors: Vagelis Papalexakis, Yorgos Tsitsikas

1.0.2 Code and Responses: Amirsadra Mohseni

1.0.3 University of California, Riverside

In this assignment we will implement the K-means clustering algorithm. We are going to use the same dataset as in the previous two parts

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
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)
data.head()
```

[1]:

	sepal_length	sepal_width	petal_length	petal_width	label
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

1.1 1. Implementing and testing K-means clustering

1.1.1 Section 1a: Implementing K-Means clustering

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

We 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
4. 'tol' is the tolerance for the change of the sum of squares of errors that determines convergence.

Our 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
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.

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.

Our algorithm should converge if

1. the maximum number of iterations is reached
2. if the SSE between two consecutive iterations does not change a lot (as in the gradient descent for linear regression we saw in part 2). In order to check for the latter condition, we will 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 we 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]:
```
#k-means clustering
def kmeans_clustering(all_vals, K, max_iter=100, tol=pow(10,-3)):
    # store only numeric values
    all_vals = all_vals.select_dtypes(include=np.number)
    N = all_vals.shape[0]

    # K x M matrix, each row of which contains the centroid for every cluster
    centroids = all_vals.sample(n=K)

    # contains the cluster number
    # that the i-th data point is assigned to
```

```
    assignments = np.zeros(N)

    # Initial centroids (centroid 0...k) belong to the 0...K clusters
    for k in range(0, K):
        index = centroids.iloc[k].name
        assignments[index] = k

    centroids = centroids.reset_index(drop=True)

    # contains all the sum of squares of errors per iteration of the algorithm
    all_sse = []
    iters = 0

    for it in range(0, max_iter):
        iter_sse = []
        for data_index, data_row in all_vals.iterrows():
            # distances will hold at most K elements at any time
            distance_to_clusters = []

            # Find the distances between this data point and each centroid
            for _, cent_row in centroids.iterrows():
                distance_to_clusters.append(np.linalg.norm(data_row - cent_row,␣
↪axis=0))

            # Find nearest centroid.
            # This ranges from 0 to K clusters
            min_dist = min(distance_to_clusters)
            iter_sse.append(min_dist)
            nearest_cluster = distance_to_clusters.index(min_dist)

            # Assign this data point to the nearest cluster
            assignments[data_index] = nearest_cluster

        # calculate error for this iteration
        all_sse.append(np.mean(iter_sse))

        if (len(all_sse) > 1) and (np.absolute(all_sse[it] - all_sse[it - 1]) /␣
↪all_sse[it - 1] <= tol):
            iters = it
            break # we are done

        for cluster_index,_ in centroids.iterrows():
            points_in_cluster_indices = np.where(assignments == cluster_index)
            data_points_in_cluster = all_vals.iloc[points_in_cluster_indices]

            # calculate new centroid with mean
            cluster_mean = np.mean(data_points_in_cluster)
```

```
            centroids.iloc[cluster_index] = cluster_mean

    return assignments, centroids, all_sse, iters
```

[3]:
```
kmeans_clustering(data, 3)
```

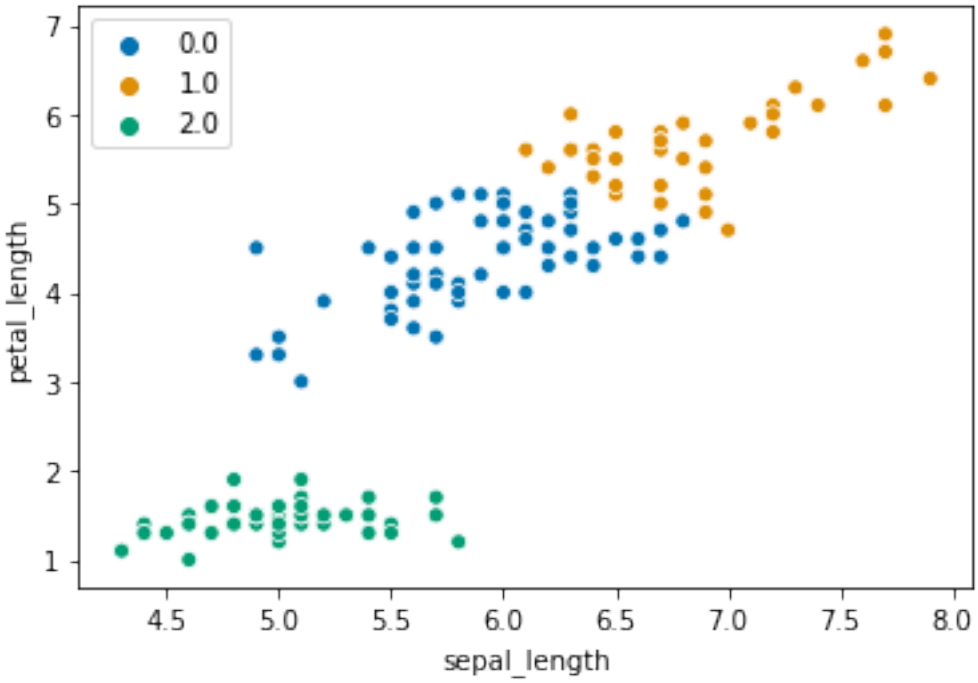
[3]:
```
(array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 2.,
        1., 2., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1., 1., 1., 1., 1., 1., 1., 2., 2., 1., 1., 1., 1., 1., 1., 1.,
        1., 2., 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., 2., 2., 2., 2., 2.,
        1., 2., 1., 2., 2., 2., 2., 1., 2., 2., 2., 2., 2., 2., 2., 2., 2.,
        2., 2., 1., 2., 2., 2., 1., 2., 2., 2., 2., 2., 2., 2.]),
   sepal_length  sepal_width  petal_length  petal_width
0      5.006000     3.418000      1.464000     0.244000
1      5.804167     2.714583      4.235417     1.345833
2      6.684615     3.017308      5.525000     1.980769,
[1.0476394812688379,
 0.6570736498623181,
 0.6528260503594228,
 0.6508006851682077,
 0.6502322252594368],
4)
```

1.1.2 Section 1b: Visualizing K-means

In this section we will 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 assignments.

[4]:
```
assignments = kmeans_clustering(data, 3, max_iter=100, tol=pow(10,-3))[0]
```

[5]:
```
ax = sns.scatterplot(x="sepal_length", y="petal_length", hue=assignments,␣
↪palette='colorblind', data=data)
```



1.1.3 Section 1c: Testing K-means

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, we 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, we report the mean SSE per K, surrounded by error-bars which will encode the standard deviation.

[6]:
```
def iterate_kmeans_clustering(all_vals, K_min, K_max, repeat=50, max_iter=100,␣
↪tol=pow(10,-3)):
    k_num = K_max - K_min + 1
    sse = np.zeros((k_num, repeat))
```

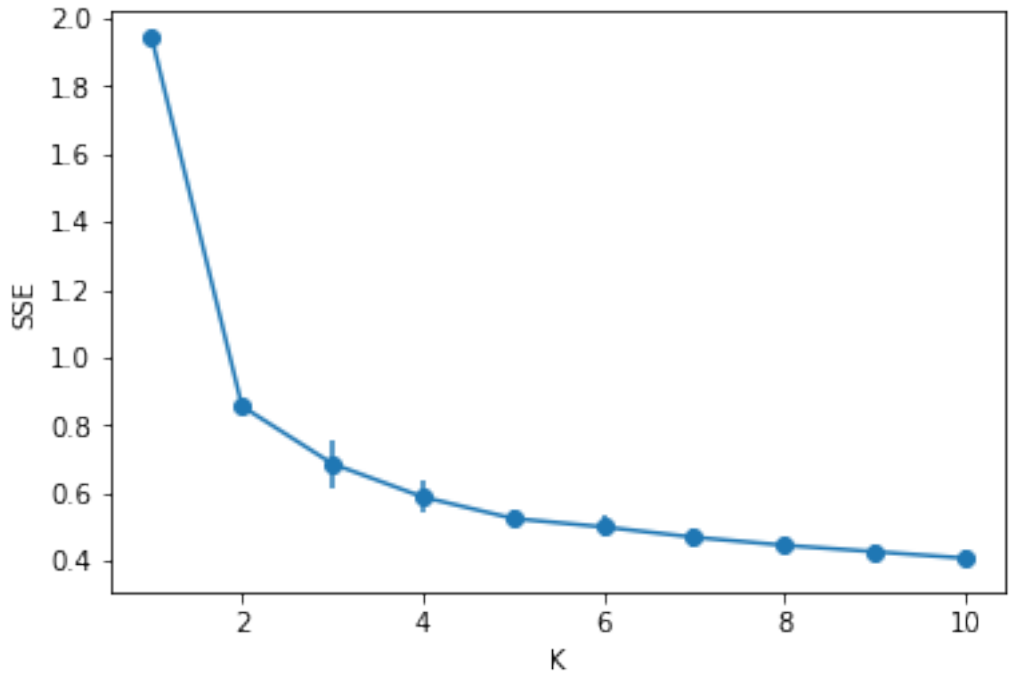
```
    i = 0
    for k in range(K_min, K_max + 1):
        for r in range(0, repeat):
            # kmeans_clustering returns all_sse as its third return value,␣
↪hence [2]
            # we only need the last reported sse, hence the [-1]
            sse[i][r] = kmeans_clustering(all_vals, k, max_iter, tol)[2][-1]
        i += 1

    sse_mean = np.mean(sse, axis=1)
    sse_std = np.std(sse, axis=1)

    K_array = list(range(K_min, K_max + 1))

    plt.xlabel('K')
    plt.ylabel('SSE')
    plt.errorbar(K_array, sse_mean, sse_std, marker='o')
```

[7]:
```
iterate_kmeans_clustering(data, 1, 10, repeat=50, max_iter=100, tol=pow(10,-3))
```



Observe that the rate of improvement significantly plateaus after $K = 4$