Assignment 2

K-Means Algorithm

ALEXANDER CROLL

Data Mining, IT721A

Data Science, University of Skövde

Alan Said, Niclas Ståhl

September 20, 2017

Table of Contents

Table of Contents ..... **II**

1. Abstract **1**

2. Description of Solution **2**

2.1. K-means 2

2.2. Step 1 - Label Assignment 3

2.3. Step 2 – Calculating Centers 3

2.4. Additional Functions 4

3. Results **5**

3.1. Finding the best value for k 5

1. Abstract

The purpose of this document is to provide documentation for the second programming assignment in Data Mining (IT721A) course.

This paper accompanies the Python code within the .zip file *DM2017_b16alecr_assignment2_code.zip*. There, code for an implementation of k-means clustering algorithm was written. In this document, the solution will be briefly described and the obtained results will be presented.

‘

2. Description of Solution

All relevant code to k-means clustering is contained in the file *K-Means.py*. There are no additional files, e.g. for data loading.

The libraries *matplotlib* and *numpy* are imported for plotting and handling data in Python, but no other machine learning libraries were used for this implementation.

The library *sklearn* (scikitlearn) is imported to make use of one of the datasets which are included in this library. Here, the well-known “Iris” dataset is used for clustering and loaded by using the function *load_data()*.

The last import is the library *sys* which is used to provide a large number (i.e. the maximum integer number) for a calculation.

The assignment is divided into several steps/functions which will be described in the subsequent paragraphs.

2.1. K-means

The *k_means()* function is the main function of the k-means clustering algorithm. It takes two arguments:

- X, which is the data (Iris dataset)
- K, which is the number of clusters to cluster the data in

As a first step, the function selects randomly K amount of data points that will represent the original centers of the clusters.

The second part consists of

- *assign_labels()*, a function that will go through every data point and assign a cluster label to each point based on the lowest distance to any of the centers.
- *calculate_centers()*, a function that will take all data points from each cluster and calculate a more accurate center for each cluster.
- *test_convergence()*, a function that will check whether or not the clusters have converged, meaning that the improvements of additional iteration will be below 0.001 in distance.

The second part will continue iterating, i.e. updating label assignments and center calculations, until the function *test_convergence()* returns *True*. In this case, the mean squared error is calculated by calling the function *evaluate_performance()*.

2.2. Step 1 - Label Assignment

The function *assign_labels()* takes two parameters:

- X, which is the data (Iris dataset)
- Centers, which is an array of k amount of cluster centers, depending on the number of clusters required

The function will take each data point from the dataset and will then check the data point against each center iteratively. The center which is closest to the data point indicated the cluster that the data point should be assigned to. Once the best fit is determined, the cluster number (i.e. an index, range(0, K)) is written into an array which contains all of the labels for all data points. Indexing helps matching up label indices with data points.

2.3. Step 2 – Calculating Centers

The function *calculate_centers()* take three parameters:

- X, which is the data (Iris dataset)
- Labels, which are the label assignments for all data points (from previous function)
- K, which is the number of clusters

The function will try to add up all data points from each respective cluster and then calculate an average of each cluster, resulting in a point that is the best possible center at this time.

To do so, the function will iterate through each data point and find the assigned label for each point. By knowing the label index, the function knows the corresponding cluster. It will then add the point to the corresponding cluster and will eventually come up with a sum of all points from each cluster. A counter keeps track of the number if points contained in each cluster.

In a last step, the function divides the sum of all points in each cluster by the respective number of items in the cluster, resulting in an average point, representing the new center.

2.4. Additional Functions

- *distance()*, which takes two points as input parameters in order to calculate the Euclidian distance between these two points.
- *test_convergence()*, which takes two points as input parameters (i.e. centers) to determine the distance between these two. If the distance between both points is lower than 0.001, the function will return True, indicating that the algorithm has stabilized (converged).
- *evaluate_performance()*, which takes three input paramters:
    - X, which is the data (Iris dataset)
    - Labels, which are the label assignments for all data points
    - Centers, which are the centers for each cluster

  The function will loop through all data points and for each point it will find the corresponding label and therefore also the matching center point. With this information the squared error between both points is calculated and saved in an additional array.

  Once the iteration is done, the function will return the mean squared error, defined by

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(X_i - X)^2$$

  which gives an indication of how well each cluster is defined.

3. Results

In this chapter, the clustering results from using different values for k will be briefly discussed.

3.1. Finding the best value for k

Depending on the dataset, the best value for k will change. Therefore, the k-means algorithm should be run with different values for the same dataset in order to find the optimal value.

The below for-loop helps running the k-means algorithm for different values for k, with the highest value being k=20 (i.e. 20 clusters).

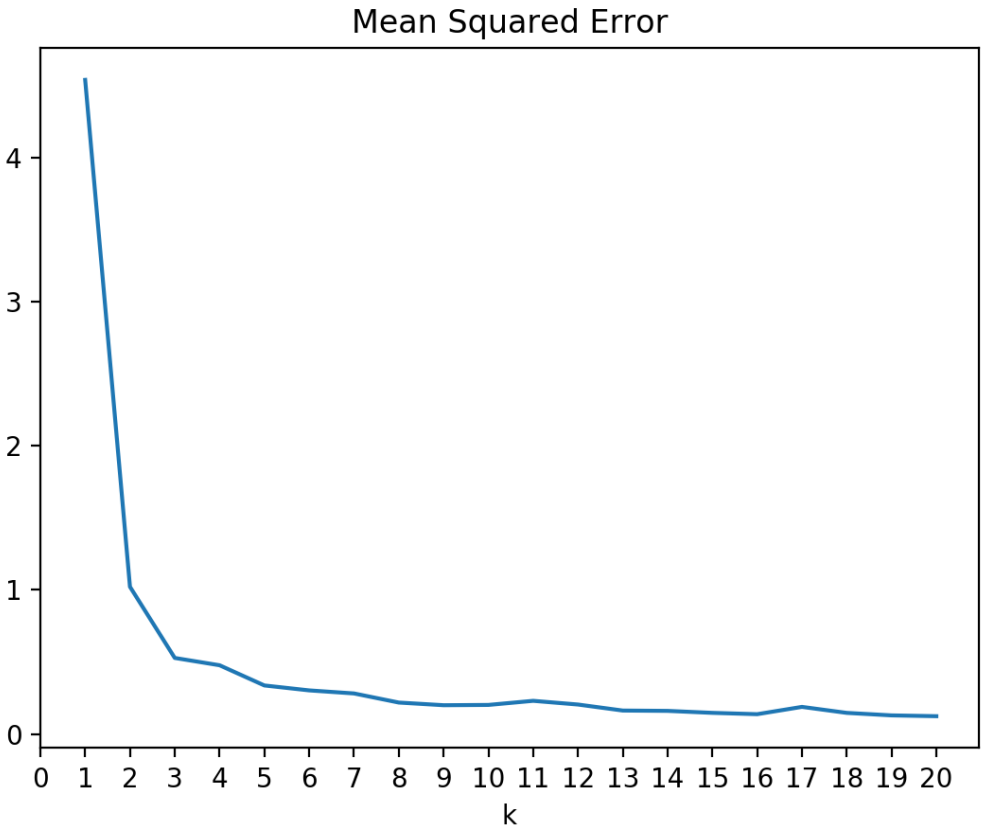
```
X = load_data()
k = 20
mse = []
for i in range(1, k+1):
    mse.append(0)
    labels, mean, centers = k_means(X, i)
    mse[i - 1] = mean
print(mse)
mse = [None] + mse
```

First, data is loaded into X and the value for k is set to 20 (meaning that the algorithm will be run for k=1, k=2, …, k=20). The for-loop then calls the k-means algorithm 20 times. At each iteration, the return value of the mean squared error is saved into an array. By looking at the printed values of the mean squared errors, one can already get a good idea of which values are good and less optimal for k.

A better method for identifying a good value for k is the *Elbow method*, which can be used by plotting the values in a graph. The below code does the trick:

```
plt.plot(mse)
plt.xlabel('k')
plt.title('Mean Squared Error')
plt.xticks(range(0, k+1, 1))
plt.show()
```

The result is the below graph.



Here, it can be seen that for k=5 the algorithm already provides good results which do not improve significantly by increasing the number of clusters. If one wanted to be a bit more accurate by ~ 0.1 in MSE, k=8 would also provide a valid fit. Anything beyond this number of clusters does not promise any improvements and will only use computing power.