Jose Maria Martínez Marín - 100443343

Francesco Mora – 100439601

Ignacio Soria Ramírez - 100443660

TECHNOLOGICAL FUNDAMENTALS IN THE BIG DATA WORLD

ASSIGNMENT LAB 2: K-MEANS CLUSTERING

1. R programming

1.1. Introduction

The given dataset is a table organized with these columns:

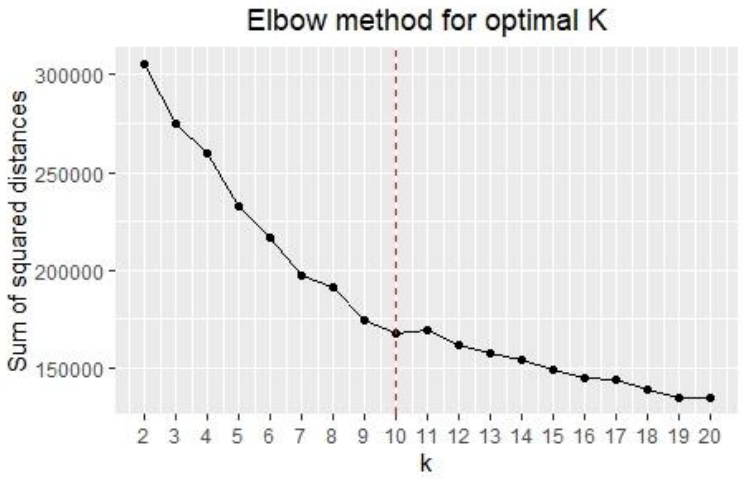
- ID: it's an integer number from 1 to 37554
- Price: it's an integer number from 1500 to 5600
- Speed: it's an integer number from 25 to 100
- Hd: it's an integer from 80 to 2100
- Ram: it's an integer number from 2 to 32
- Screen: it's an integer number from 14 to 17
- Cd: it contains Boolean values
- Multi: it contains Boolean values
- Premium: it contains Boolean values
- Ads: it's an integer number from 39 to 339
- Trend: it's an integer number from 1 to 6

The total number of rows is 37554.
The aim of the table is to organize computers depending on their main characteristics.
The first column label contains strange characters (ÈÀid), thus it's necessary to change it so that the program can run without giving errors.
For the purpose of this work, as this column doesn't provide any added value, it is decided to delete it.
The three Boolean values are transformed than into dummy variables (zeros and ones) so that they can be treated as numbers, as all the rest of the variables. The if-else loop is used to perform this transformation.
Finally, as the all the variables are of incomparable units (€, Gb, inches, …), it is necessary to normalize the data before applying the k-means for clustering. Thus, starting from the original dataset, a new one has been built scaling each variable into a new one with the mean equal to zero, and the standard deviation equal to 1.
After such operations, the dataset is clean enough to run the program in the R environment.
The dataset that is used from now on is called CPscaled.

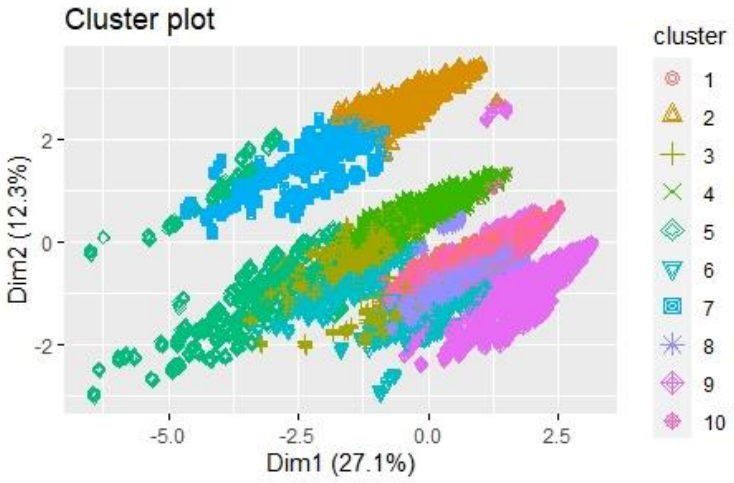
1.2. R Serial program

The first part of the program consists of the finding of the optimal k for the clustering.
A function called kmeansfunc is defined to find the k optimal number of clusters. To end the loop it is decided to limit the possible number of k up to 20.
The function is then ran with the sapply method, and the execution time is measured. The output is an array of numbers, one for each k up to the selected maximum k.
Every number represents the sum of squared distances, which describes the dispersions of the points from the considered centroids.

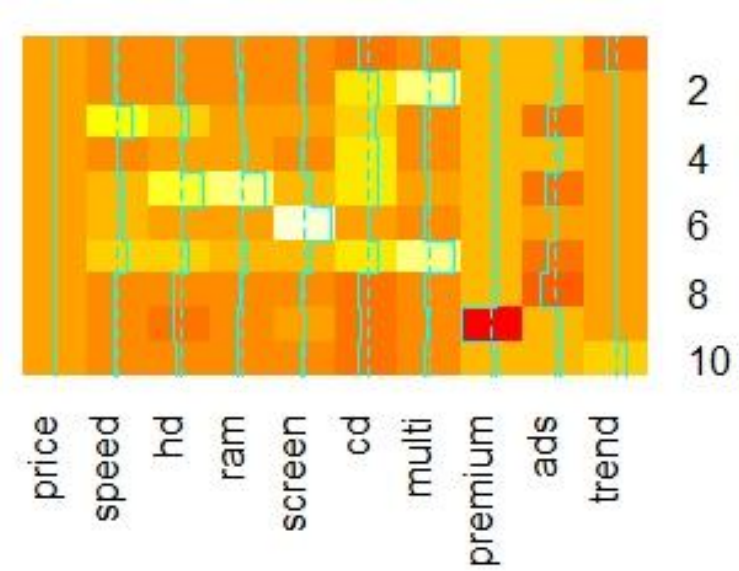
The function elbowPoint is then invoked to determine the optimum number of k clusters according to the elbow method.
Using the knee of the curve is a common method to choose the optimum point, because adding more clusters is no longer worth the additional cost.
The result of the elbow method is then plotted using the ggplot2 library:



The vertical red line is indicating the best number of clusters according to the elbow method, which in this case is 10.
Once the number k is set, the kmeans function is called to build the k clusters. The outputs of this functions are many, for instance the coordinates of the centroids, or the cluster which every point belongs to.
The points are then plotted into the first 2 dimension to get a x-y graphic. A color is associated to each point, representing the cluster of belonging.



The cluster with the highest average price (first column) is then found checking the centroids matrix, and the result can be verified as well checking the plot.
Finally, a heatmap of the clustering is performed:



This plot gives a visual idea of where every cluster is, considering all of the variables taken into account in this problem.

1.3. R Parallel program

A parallel program is implemented to speed up the performance, taking advantage of the available cores.
The first part of the program is the same as for the serial version of the program, and the dataset is cleaned to make the clustering possible.
After defining the kmeansfunc, the socket approach is followed to run the program in parallel.
With the makeCluster function (belonging to the parallel library) a cluster is created and the number of cores can be chosen depending on the kind of application.
In this case, it was decided to work in over-provisioning (the number of tasks is bigger than the number of cores) because the execution time was the minimum one.
The parLapply function is invoked to operate the desired function in parallel, and after the computing the cluster is closed.
The execution time is then measured.
The rest of the program can not be parallelized, thus the results are the same as in the serial version.

1.4. R Parallel program with foreach

The foreach library is a direct translation of OpenMP, and it allows to process loops in parallel.

Both foreach and doParallel libraries have to be uploaded.
The registerDoParallel function is used to open the cluster with the desired number of cores.
The output is then collected by invoking a foreach loop that runs the kmeansfunc in parallel.
As the previous examples, the execution time is collected. The rest of the program is serial, thus no difference are found.

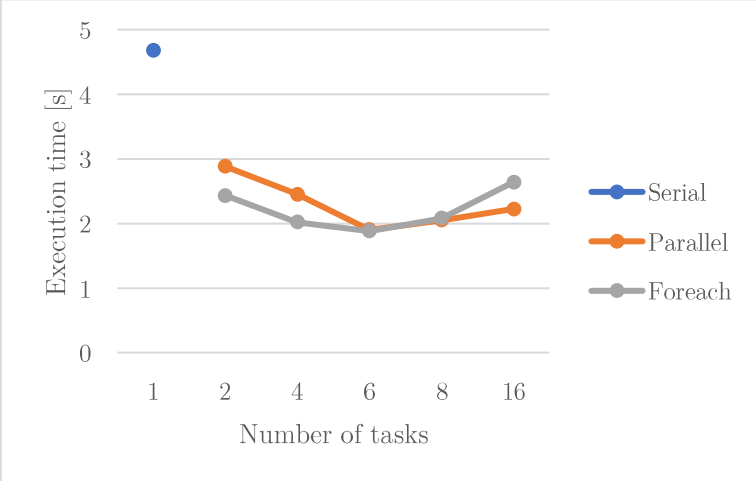
1.5. Conclusions for the R part

Here below the achieved results in terms of execution time are shown.
The conditions were maintained the same between each measurement. The machine on which the trials were performed has a 4 cores processor and Windows OS.

Execution time [s] with the three different programs

	Number of tasks					
	1	**2**	**4**	**6**	**8**	**16**
Serial	4.68					
Parallel		2.88	2.45	1.90	2.05	2.22
Foreach		2.43	2.02	1.88	2.08	2.64

These results are plotted in a graph:



The following conclusions are stated:

- The parallel programs performance draws a curve depending on the number of tasks per cluster. In both cases, the minimum of the curve is reached with 6 tasks, thus over-provisioning.
- Both parallel programs are much faster than the serial one. Considering clusters with 6 cores, the speed-up can be calculated as:

$$S_{parallel} = \frac{t_{ser}}{t_{par}} = 2.46$$

$$S_{foreach} = \frac{t_{ser}}{t_{par}} = 2.48$$

- The programs could be speeded up some more, by writing the k-means function from zero. This way, another parallelization could be possible and the execution could decrease more.

2. Python programming

2.1. Introduction

The goal of the program is to cluster the data using the kmeans algorithm with different clusters, from 1 to 20 and to return

- An elbow graph of the score plotted against the number of clusters together with the ideal number of clusters
- An implementation of kmeans with the ideal number of clusters
    - The centroid with the highest average price
    - A heatmap of the first two dimensions

Different approaches will be tried in the multiprocessing programming:

- Pool.starmap() function
- Threads

With each of them, the execution time will be calculated slicing the program and the data in different chunks, from 2 to 16.

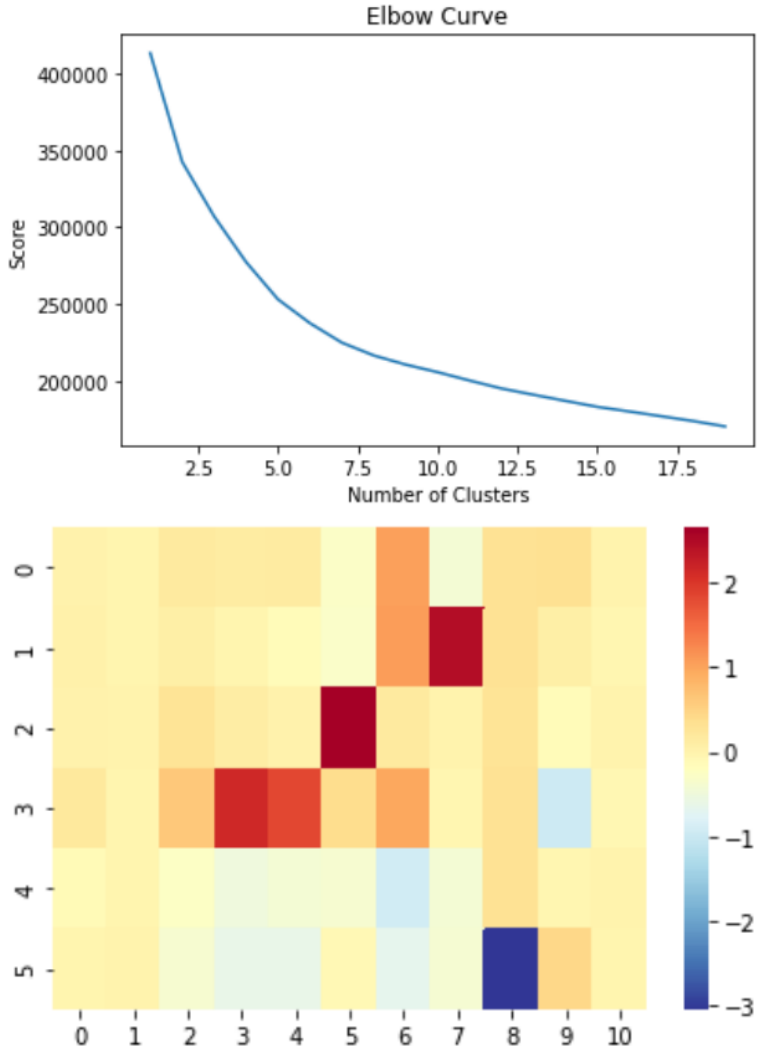
Finally, the best solution for the machine which this assignment was made with will be chosen.

2.2. Python serial program

The schema of this code is as follows:

- reads the data file and transform 3 variables from strings into 0 or 1
- Data is standardized with StandardScaler()
- A function is defined to read a list of integers and to calculate the kmeans for a number of clusters equal to each item on the list. The output is a list of the squared distances appended.
- The function is called with a list from 1 to 20
- The score is plotted against the number of clusters
- With the function KneeLocator.knee, the ideal number of clusters is identified
- Ideal number of clusters is printed together with the squared distance
- The data is then clustered with the ideal number of clusters found
- The centroid with the highest average price is found and printed
- A heatmap of the centroids of the first two dimensions is plotted

The results of the serial program are shown below:



2.3. Python multiprocessing program

The idea with the multiprocessing approach is to divide the function and the number of clusters to check into several independent chunks. In this way, it is possible to run a big query in parallel, taking advantage of the available cores in the CPU, and thus speeding up the program. It is thought that it is not possible to split the dataset and then merge the results as the clustering must be done on the whole dataset, therefore the iterations are performed on groups of numbers of clusters to check from 1 to 19.

The library in charge of the multiprocessing activity is called *multiprocessing*.

Starmap was used as the method of implementing the function in parallel.

The schema of this code is equivalent to the serial version except for:

a. The arguments of the function are stored in a list of arrays, each array will contain the scaled data and a set of integers. If the data is divided in 2 for example, we would have two lists in the following shape

$$arguments = [(CPscaled, range(1,10)), (CPscaled, range(10,20))]$$

b. Starmap is called using this list and the function itself as arguments which returns the output as a list of lists with the results. In order to append the results together the following statement is used:

$$join\ output = [item\ for\ sublist\ in\ output\ for\ item\ in\ sublist]$$

The results achieved are the same as in the serial version as expected

2.4. Python threads program

In the implementation with threads, the number of clusters to check is divided into the same number as the number of threads created in order for each thread to have a block of clusters to check.

The schema of this code is equivalent to the serial version except for:

A. The output results are a new argument of the function and the function now appends the results into a list named output

B. The variable Nc which contains numbers from 1 to 19 is split into lists equal to the number of clusters in order to parallelize when calling the threads

$$N_c = range(1,20)$$
$$threads = 19$$
$$splitdataNc = np.array\_split(Nc, threads)$$

The results achieved are the same as in the serial version as expected.

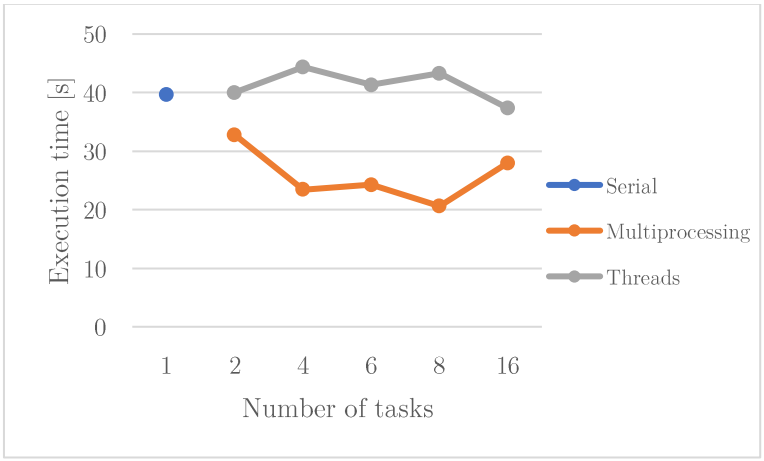
2.5. Conclusion for the Python part

Here below the achieved results in terms of execution time are shown.
The conditions were maintained the same between each measurement. The machine on which the trials were performed has an 8-core processor and Windows OS.

Execution time [s] with the three different programs

	Number of tasks					
	1	**2**	**4**	**6**	**8**	**16**
Serial	39.6	-	-	-	-	-
Pool.starmap	-	32.80	23.41	24.25	20.60	27.96
Threads	-	40.02	44.37	41.32	43.27	37.34

These results are plotted in a graph:



The following conclusions are stated:

- A very interesting relationship can be observed in the graph of performance against partitions. A mirror relationship can be observed, where the optimum number of partitions for starmap lies in the middle and that of threads in the upper and lower edges.
- Multiprocessing achieved a reduction in time of 19 seconds which is equivalent to a speed-up of 1.92:

$$S = \frac{t_{ser}}{t_{par}} = 1.92$$

- Pool.starmap() was found to be the multiprocessing method achieving the fastest time.
- 8 partitions was found to be the fastest approach, possibly as the computer used has 8 cores and this maximized the efficiency of the resources.
- The program with the threads is not much faster than the serial one, with a speed improvement of around 2 seconds.
- 19 threads was found to achieve the best results, probably as what we are dividing is the list of number of clusters to be checked and not the data itself. In the case of 19 threads, each thread would check only 1 cluster number, but for the whole dataset.

- The complication inherent to this problem is that it is not possible to split the dataset as each cluster number must be checked on all the data. This is likely what is limiting performance.
- It was found that there is no difference in performance between using dataframes and lists as may be expected, possibly due to pandas functions acting on whole columns being optimized.