uc3m | Universidad Carlos III de Madrid

Master in Big Data Analytics
Technological Fundamentals in the Big Data World
Parallelizing K-means in Python and R

Didier Dirks - 100443386
Jordan Wicht - 100437856
Jose Antonio Jijon Vorbeck- 100438045

1. Introduction

For this project, we create a program which clusters the computers.csv based on their characteristics such as speed, pricing, ram, etc. To do this, first we find the optimal number of clusters to create based on the inertia of each cluster (sum of squares of each point compared to its cluster center), then we plot a graph based on speed and pricing of the cluster centroids. The end result is a heatmap which displays all of the characteristics of each cluster, based on its cluster centroids. We also record the processing time in two parts: first, the amount of time to identify the optimal number of clusters; second, the amount of time to cluster the data and display the heatmap. This process is executed in both Python and R programming languages, each with a serial version and two parallelization versions, threads and multiprocessing for python, and sockets or foreach for R . The results and description of each program are separated below by programming language and serial and parallelization method.

Improvements in CPU clock rates have tapered off over time. As such, nowadays it is more efficient to utilize parallelization to reduce processing time. In theory, running programs in parallel should yield quicker processing times as compared to their serial counterparts because similar tasks are run concurrently, instead of one after another. The improved efficiency of a parallelized program is the ratio between the parallelized processing time and the processing time of the original program. In this project, we run a parallelized version of our program using threads and multiprocessing in Python. Threading runs similar basic components (or 'threads') within a process using shared resources (e.g., memory, executable code). Multiprocessing runs independent parallel subprocesses (no more than the number of processors in the computer) in separate memory locations. One key difference between threading and multiprocessing is that while the subprocesses in multiprocessing are independent, threads are subsets of a single process (and thus dependent on one another).

For our R programs, we parallelize the serial program using forking, sockets, and foreach. Forking creates duplicate programs that are copies of the serial program, after variables and objects are defined. A major drawback of this approach is that forking does not work with a Windows operating system. For sockets, variables and objects are not shared between each thread, so a continuous communication is needed between the threads and the serial program. This results in a slower parallelization method, but it is amenable to any operating system. As for the foreach process, all the variables and functions are available at all cores by default, so the system has access to them while it is running in parallel.

With massive amounts of data that have no explicitly defined groupings, k-means can be a useful tool to cluster the data using the data itself. Data is grouped, or "clustered" based on a common point (the centroid) that each cluster is closest to.

To decide which is the optimal number of clusters (groups) that the data should be separated into, we must plot an elbow graph with the distance of each point to the cluster centroid (inertia or sum of squares) in the y-axis and the number of clusters in the x-axis. The elbow graph tells us the optimal number of clusters based on where the 'elbow' appears; the number of clusters corresponds to the number of points right before they level off on the y-axis.

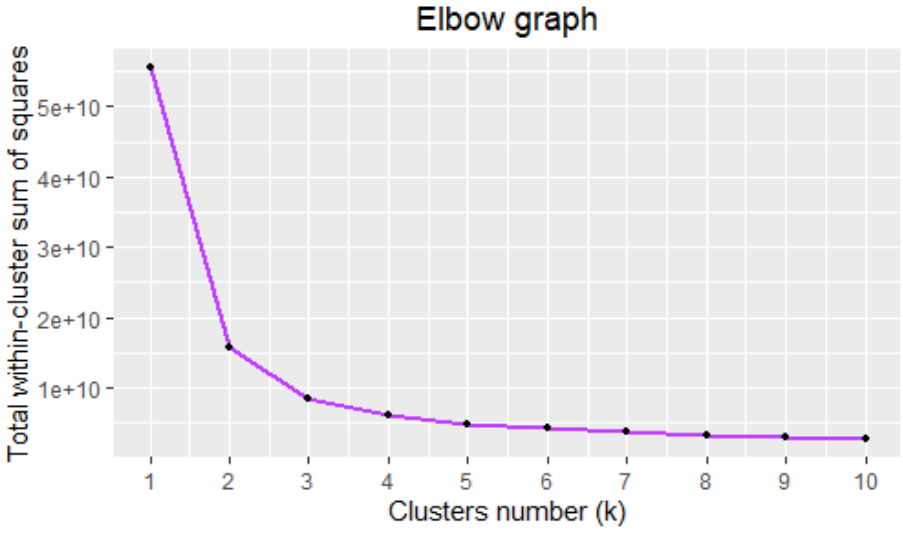
Since this method can be very time consuming and involves several stages of iterations for each clustering process, we could expect our results to show more efficient processing time for the parallelized version of our serial program. Below, we run our two serial programs in R and Python, and then compare the processing times with their parallel counterparts.

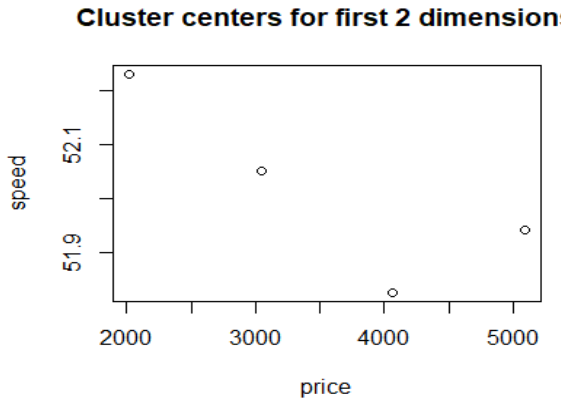
2. Program in R

A short description of the program will be given under the serial part, and then the parallelized versions will be only a presentation of the timing results, since the core and functionalities of the script are fundamentally the same.

Serial Program

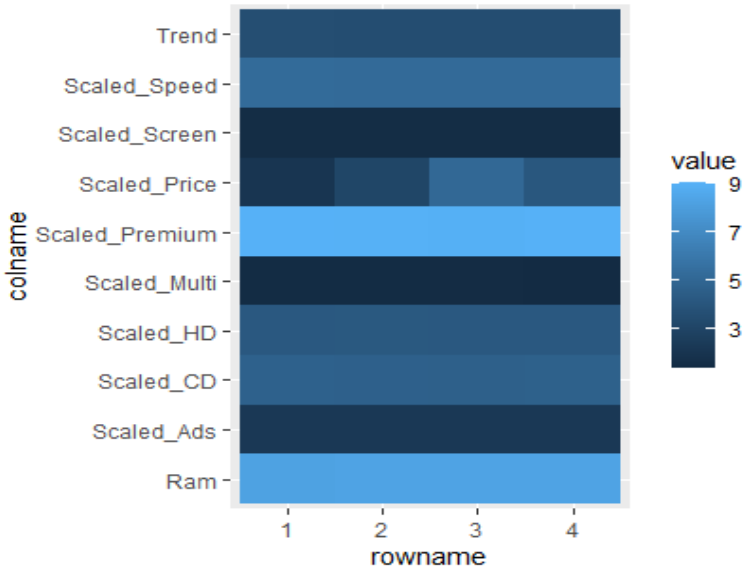
On our serial program in R, we first transform the variables CD, MULTI, and PREMIUM, so that 'yes' = 1 and 'no' =0 . We then run models with 'map_dbl' with various centroids "k" (1-10) and plot these as an elbow graph:





We use this graph to make a judgment on the number of clusters to create. In this example, we will create 4 clusters. We then plot the centroids of each cluster using variables PRICE and SPEED. We then print the cluster with the highest average price, which in this example is 5093.53.

We then scale the variables PRICE, SCREEN, and ADS to standardize the values. This is done so the heatmap is more readable when comparing the variables from each cluster. The heatmap is presented below with the scaled values, which are only used for visualization purposes:



We measure the processing time by subtracting the system time at the beginning of the program from another system time at the end of the program. In this example our time was 2.53 seconds. Here is a print of the processing time:

```
> print(paste("Cluster with highest price average is:", max(Custer_avgs[,2])))
[1] "Cluster with highest price average is: 5093.37077472181"
>
> print('Total time is:')
[1] "Total time is:"
> endtime - starttime
Time difference of 2.53035 secs
```

Parallelization using Sockets

This program is similar to the serial one, expect here we use sockets to reduce our processing time. After loading the data frame and transforming the binary variables, we set the environment for parallelization by counting the cores and creating the corresponding number of clusters.

The elbow graph, scatterplot, and heatmap are identical to the serial program. So here, we will only display the printouts, with includes the processing time:

```
> print(paste("Cluster with highest price average is:", max(Custer_avgs[,2])))
[1] "Cluster with highest price average is: 5093.37077472181"
>
> print("Total time to perform process is:")
[1] "Total time to perform process is:"
> endTime - startTime
Time difference of 3.445656 secs
```

Parallelization using Forking

This program is similar to the serial one, expect here we use forking to reduce our processing time.

The elbow graph, scatterplot, and heatmap are identical to the serial program. So here, we will only display the printouts, with includes the processing time:

```
> print(paste("Cluster with highest price average is:", max(Custer_avgs[,2])))
[1] "Cluster with highest price average is: 5093.5328152893"
>
> print("Total time to perform process is:")
[1] "Total time to perform process is:"
> endTime - startTime
Time difference of 1.975993 secs
>
```

Parallelization using ForEach

This program is similar to the serial one, except here we use foreach to reduce our processing time. The elbow graph, scatterplot, and heatmap are identical to the serial program. So here, we will only display the printouts, with includes the processing time:

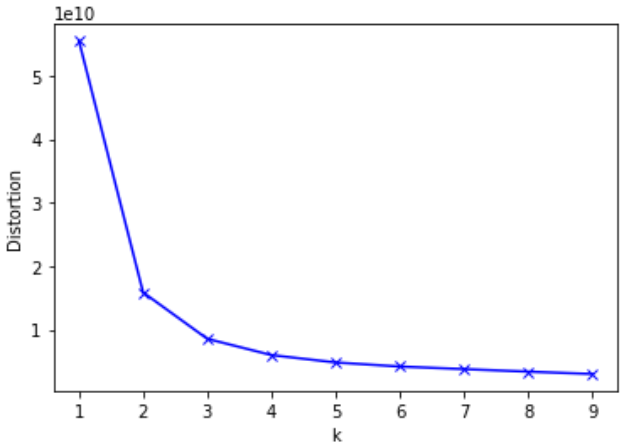
```
> print(paste("Cluster with highest price average is:", max(Custer_avgs[,2])))
[1] "Cluster with highest price average is: 5093.37077472181"
>
> print("Total time to perform process is:")
[1] "Total time to perform process is:"
> endTime - startTime
Time difference of 2.577235 secs
```

3. Program in Python

Similar to the R program, a small description will be given in the serial program part, while in the parallelized versions we will limit to a short presentation of the timing results, since presenting the graphs and results again would be a redundancy, since all programs perform exactly the same.

Serial Program

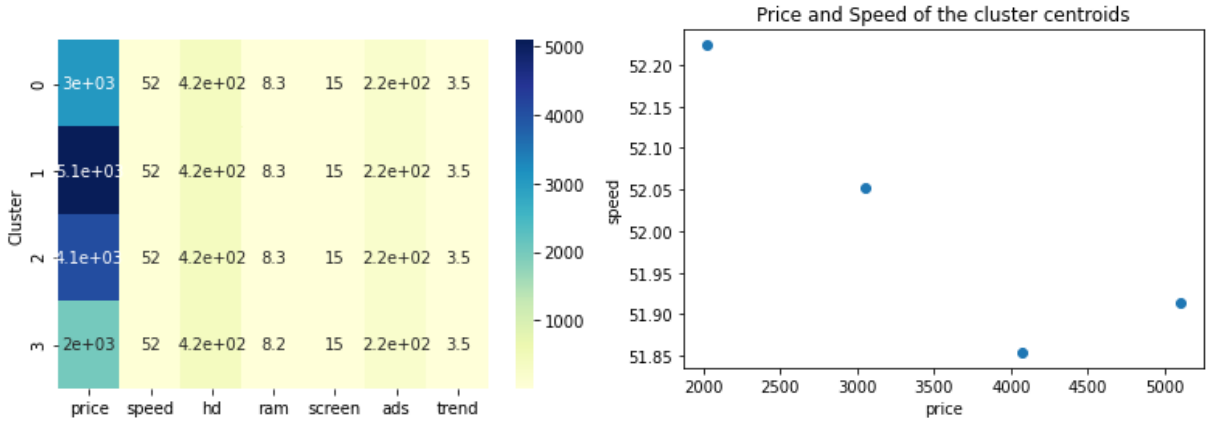
For our serial program, we first define three functions. The first creates our elbow graph to display the optimal number of clusters to form, as well as the time taken (using the *timeit* module) to process the graph. The second function produces a scatter plot of the centroids with the variable 'speed' on the y-axis, and the variable 'price' on the x-axis .This tells us how far apart the means of each of these variables for each cluster is. Finally the third function displays a heatmap of each variable separated by cluster.
When executing the program, the user is given the elbow graph:



This graph is then followed by the processing time of creating this elbow graph, which is 3.518 seconds, and a prompt in the console asking for the number of clusters:

```
didierdirks@MacBook-Pro-van-Didier lab02 % python3 kmeans_serial.py
Time up to here is: 3.5180583300243597
How many clusters?:
```

Based on the elbow graph, for this example we will choose 4 clusters. Once we enter this, we are given the scatterplot with the four centroids as well as the heatmap displaying the values of each variable for each cluster showed in the next page.



And finally a message telling us the cluster with the highest average price and the processing time from the time we entered the number of clusters to the printing of the heatmap:

```
How many clusters?: 4
Cluster with highest Price avg has price = 5096.23
Time of second part: 0.7199203759955708
```

The timing of the program is split up into two parts. One part for creating the elbow graph until the user is prompted to enter a number of clusters, and the second part to perform the rest of the operations. This is done so because it would not be fair to keep the timer running while the user is prompted to enter an input. The total duration of the program would be both parts added together. In this case that would be 3.518 + 0.7199, which results in a total of 4.2379 seconds.

Parallelization using Threading

The goal of this application was identical to the previous one, except that in this case threading is used. This program creates different threads that each simultaneously create clusters for the k-means algorithm. The process of simultaneously creating clusters is meant to speed up the execution time. We have concluded that the optimum number of threads is 10, and as a standard always 10 threads are created. This means that each thread is targeted at about 900,000/10 rows, meaning 90,000 rows per thread. Users do, however, have the flexibility to specify another amount of threads to be created if necessary. After the threads are joined together the occurrences are concatenated.

After plotting the clusters with the elbow graph, once again we chose 4 clusters as the optimal number. The elbow graph, centroid scatterplot, and heatmap are identical to the serial program, so they will be omitted here. Below are the processing times and a print of the centroid with the highest average price:

```
didierdirks@MacBook-Pro-van-Didier lab02 % python3 kmeans_parallel_threads.py
Time up to here is: 2.9655646479805
How many clusters?: 4
Cluster with highest Price avg has price = 5094.13
Time of second part: 0.4531261119991541
```

The total duration of processing time for this program is 3.4187 seconds.

Parallelization using Multiprocessing

Again, this program has the same goal as the previous ones, but here we are using multiprocessing. For the multiprocessing version the file is divided into the same number of chunks as the CPUs of the computer that it is running on. This means that if the computer has 4 CPUs, the file will be divided into 4 different chunks. This way all of the computational power of the CPUs of the computer is optimally used and will hopefully speed up execution time.

We created two programs using multiprocessing. The first uses multiprocessing throughout the entire process, saving the clustering information for every cluster number, while the second does not save the clustering information, only the inertia values. This enables the second program to run a bit faster in the first part, but then in the second pard it has to recalculate the k-means algorithm, while the first version stores the results and then only picks the clustering method with the selected number of clusters.

This is the time for the first method, that does not store the values:

```
didierdirks@MacBook-Pro-van-Didier lab02 % python3 kmeans_parallel_multiproc.py
Time up to here is: 2.1553672879817896
How many clusters?: 4
Cluster with highest Price avg has price = 5095.85
Time of second part: 0.5498109220061451
```

This method however, stores the values in the first part, so we see it takes longer in the first part but is faster in the second part of the program.

```
didierdirks@MacBook-Pro-van-Didier lab02 % python3 kmeans_parallel_multiproc2.py
Time up to here is: 2.3938517200003844
How many clusters?: 4
Cluster with highest Price avg has price = 5095.1
Time of second part: 0.48753889501676895
```

The total duration of processing time for the first program is 2.7052, the processing time for the second program is 2.8815 seconds.

4. Conclusion

We took three versions of a similar program--both in R and Python-- designed to find the optimal number of clusters using the k-means algorithm. The processing times for each program to execute are recorded for comparison. We see that for some cases the parallelization method is not optimal and takes actually more time than for the serial versions. This is the case of sockets and foreach in R, while all the parallel versions in python are faster than the serial part.

In our Python example, the serial program in total ran in about 4.24 seconds, the parallelized version using threads ran in 3.4186 seconds, and the parallelized version using multiprocessing ran in 2.7052 seconds.

In our example using R, the serial program took about 2.53 seconds. For the parallel versions, our processing time took about 3.445 seconds using sockets, and our processing time took about 1.975993 seconds using forking. For the foreach case, it took 3.567 seconds to complete.

Processing times in R		Processing times in Python	
Process Type	Improved Efficiency	Process Type	Improved Efficiency
Serial	1	Serial	1
Parallel with Sockets	2.530 /3.445 = 0.734	Parallel with Threads	4.2379 / 3.4187 = 1.24
Parallel with Forking	2.53/1.9759 = 1.28	Parallel with MultiP version 1	4.2379 / 2.7052 = 1.5666
Parallel with For-each	2.53 /3.567 = 0.709	Parallel with MultiP version 2	4.2379 / 2.8815 = 1.4707

As for the clustering method used in this laboratory, we have to mention that it might not be the optimal one, since we have not scaled the data before using the k-means algorithms, and therefore we can see that the variable price is the dominant one, and all the other variables are more or less dependent on the clustering of the variable price. But we have decided not to change much the programs, since the objective of this laboratory was not to optimize and get the best possible clustering method, but the objective was to compare the parallel versions of the system against the sequential one.