LAB1. K-MEANS PARALLELIZATION in R and PYTHON

Alberto José Díaz Cuenca, Ignacio Almodóvar Cárdenas and Javier Muñoz Flores

12/3/2022

Creation of the dataset.

To begin with the project, we had to generate a random dataset in Python thanks to the file named "computers-generator.py". For the development of the project we generated a dataset of 5000 observations. This dataset, saved in the file "computers.csv" is about a list of computers and it contains some information about them: "id","speed","hd","ram","screen","cd","multi","premium","ads","trend".

Laboratory Description: You are asked to extract useful information the computer data set implementing a program using the k-means algorithm in Python and in R.

Implementation in R: Part one - Serial version

1.- Construct the elbow graph and find the optimal clusters number (k).

OPTION A

2.- Implement the k-means algorithm

3.- Cluster the data using the optimum value using k-means.

OPTION B

3.- Cluster the data using the optimum value using k-means with an existing function.

We have chosen `Option A`, where we had to implement the k-means algorithm from zero instead of using the k-means existing function from the R libraries.

The first thing that we had to do was to read the dataset generated. After removing the variable "id" and transforming the categorical variables "cd" and "laptop" into binary variables, we began creating the function for the k-means algorithm.

```
library(plyr)
library(dplyr)
library(factoextra)
library(magrittr)
library(ggplot2)
library(microbenchmark)

set.seed(13)
```

```
data5k=read.csv(file = "computers5k.csv",header = T)
data5k$id = NULL
data5k$cd %<>% mapvalues(from = c("yes","no"), to = c("1","0"))  %>% as.factor()
data5k$laptop %<>% mapvalues(from = c("yes","no"), to = c("1","0")) %>% as.factor()
data5k$trend %<>% as.factor()

data_wo_factors = data5k %>% dplyr::select(c(-cd,-laptop,-trend))
```

First of all, we created a function called "generate_random", which selects a random value from the elements of a given vector. Inside the k-means algorithm, this function will be used for every column of the dataset in order to generate the random centroids.

```
generate_random=function(vector){
  return(runif(1,min(vector),max(vector)))
}
```

Right after that ant before the creation of the k-means function, we implemented another small function called "euclidean", this time to compute the euclidean distance between two points in space.

```
euclidian=function(a,b){
  sqrt(sum((a-b)^2))
}
```

Now, we proceed with the implementation of the k-means algorithm. Let's show the function created and explain every step taken afterwards.

```
kmeans_diy=function(data,k){

  #Scale data
  kmeans_data=as.data.frame(scale(data))
  n=ncol(kmeans_data)
  #Generate random centroids
  X=matrix(nrow=k,ncol=(n+1))
  #clusters=letters[1:k]
  for (i in 1:nrow(X)) {
    for(j in 1:n){
      X[i,j]=generate_random(kmeans_data[,j])
    }
  }
  X[,n+1]=as.factor(letters[1:k])


  #Compute Distances
  n=ncol(kmeans_data)
  m=nrow(X)
  nX=ncol(X)
  x=matrix(nrow = nrow(kmeans_data),ncol = m)
  for(i in 1:m){
    x[,i]=apply(X = kmeans_data,MARGIN = 1,FUN = euclidian,b=X[i,-nX])
  }
  for(i in 1:nrow(kmeans_data)){
    kmeans_data$error[i]<-min(x[i,])
```

```
    kmeans_data$cluster[i]<-which(x[i,]==min(x[i,]))
  }
  x=NULL

  #Check errors
  error=c(0,sum(kmeans_data$error))
  e=2

  while(round(error[e],0)!= round(error[e-1],0)){
    #Recode Clusters
    #kmeans_data$cluster %<>% as.factor()
    X = kmeans_data %>% group_by(cluster) %>%
      dplyr::summarize(price=mean(price),
                speed=mean(speed),
                hd=mean(hd),
                ram=mean(ram),
                screen=mean(screen),
                cores=mean(cores)) %>%
      mutate(n_centroide=cluster) %>%
      select(-cluster) %>%
      ungroup() %>% as.data.frame(.)

    #Compute distances
    n=ncol(kmeans_data)-2
    m=nrow(X)
    nX=ncol(X)
    x=matrix(nrow = nrow(kmeans_data),ncol = m)
    for(i in 1:m){
      x[,i]=apply(X = kmeans_data[,-c(7,8)],MARGIN = 1,FUN = euclidian,b=X[i,-nX])
    }
    for(i in 1:nrow(kmeans_data)){
      kmeans_data$error[i]<-min(x[i,])
      kmeans_data$cluster[i]<-which(x[i,]==min(x[i,]))
    }
    x=NULL

    #Write error
    error=c(error,sum(kmeans_data$error))

    #Next iteration
    e=e+1
    #
    print(error)
    #
  }

  return(kmeans_data)
}
```

First, as it is stated, we scaled the data and set the number of columns of this scaled data. Secondly, thanks to the "generate_random" function explained before, we generated the random centroids. In order to do so, we created an empty matrix of "k" (parameter that must be introduced in the function) rows and the number of columns of the dataset plus one, since we want to generate a new column corresponding to the

centroids. After that, we select the centroids as we previously explained and create this new column.

Now, it is time to compute the euclidean distances between the observations and the centroids. This algorithm works as follows: first, it computes the distances between each observation and the first centroid. After that, the distances between every observation and the second centroid, and so on. Once every distance is computed, it selects the minimum values of the distances towards each centroid and they are stacked in a column vector.

Finally, the function returns a new dataset (the original is not altered at all) with the new columns corresponding to the errors (the distances between the observation and the centroids) and the centroids that every observation are assigned to.

Once the main function is created, we are asked to answer a few exercises.

2.- Implement the k-means algorithm.

3.- Cluster the data using the optimum value using k-means.

In order to answer these two sections, we created a function called "obtain_k_optimal_serial":

```
obtain_k_optimal_serial=function(data,k){
  k_means=NULL
  for (i in 1:k) {
    k_means[i]=list(kmeans_diy(data,i))
  }
  return(k_means)
}
```

This function computes the main function created for the k-means algorithm for every number of centroids that we introduce. If we introduce k=5, it will evaluate the function for k=1,2,...,5. Once these operations are done, we can plot the corresponding elbow graph to find out which of the k values is the optimal (section number 5).

4.- Measure time.

In order to avoid having to compute the main function several times and then doing it again in order to measure the time, we are going to start a counting system right before calling the function, then proceed to call the function and save the results in the variable "k_means" and then stop the count once the function has finished. Right after that, we operate the end time minus the start time and get the time it takes for the function to work. Thus, we can measure the time and implement the function at the same time. Moreover, having the results stored in a variable allows us to answer another sections without having to call the function again (for instance, section number 6.)

```
start = Sys.time()
k_means = obtain_k_optimal_serial(data_wo_factors,5)
```

```
## [1]     0.00 15899.05 11883.44
## [1]     0.00 15899.05 11883.44 11883.44
## [1]     0.00 14268.40 10636.96
## [1]     0.000 14268.396 10636.961  9428.028
## [1]     0.000 14268.396 10636.961  9428.028  8205.692
## [1]     0.000 14268.396 10636.961  9428.028  8205.692  8195.561
## [1]     0.000 14268.396 10636.961  9428.028  8205.692  8195.561  8196.131
## [1]     0.000 12474.648  7969.034
```

```
## [1]     0.000 12474.648  7969.034  7352.875
## [1]     0.000 12474.648  7969.034  7352.875  6955.447
## [1]     0.000 12474.648  7969.034  7352.875  6955.447  6909.239
## [1]     0.000 12474.648  7969.034  7352.875  6955.447  6909.239  6906.912
## [1]     0.000 12474.648  7969.034  7352.875  6955.447  6909.239  6906.912
## [8]  6906.169
## [1]     0.000 12474.648  7969.034  7352.875  6955.447  6909.239  6906.912
## [8]  6906.169  6905.888
## [1]     0.000 10676.636  6862.758
## [1]     0.000 10676.636  6862.758  6613.513
## [1]     0.000 10676.636  6862.758  6613.513  6577.614
## [1]     0.000 10676.636  6862.758  6613.513  6577.614  6574.553
## [1]     0.000 10676.636  6862.758  6613.513  6577.614  6574.553  6574.512
## [1]     0.000 12028.333  7574.526
## [1]     0.000 12028.333  7574.526  6651.955
## [1]     0.000 12028.333  7574.526  6651.955  6304.857
## [1]     0.000 12028.333  7574.526  6651.955  6304.857  6085.344
## [1]     0.000 12028.333  7574.526  6651.955  6304.857  6085.344  5994.344
## [1]     0.000 12028.333  7574.526  6651.955  6304.857  6085.344  5994.344
## [8]  5964.471
## [1]     0.000 12028.333  7574.526  6651.955  6304.857  6085.344  5994.344
## [8]  5964.471  5946.713
##  [1]     0.000 12028.333  7574.526  6651.955  6304.857  6085.344  5994.344
##  [8]  5964.471  5946.713  5941.494
##  [1]     0.000 12028.333  7574.526  6651.955  6304.857  6085.344  5994.344
##  [8]  5964.471  5946.713  5941.494  5940.973
```

```
stop=Sys.time()

stop-start
```

```
## Time difference of 10.17197 mins
```

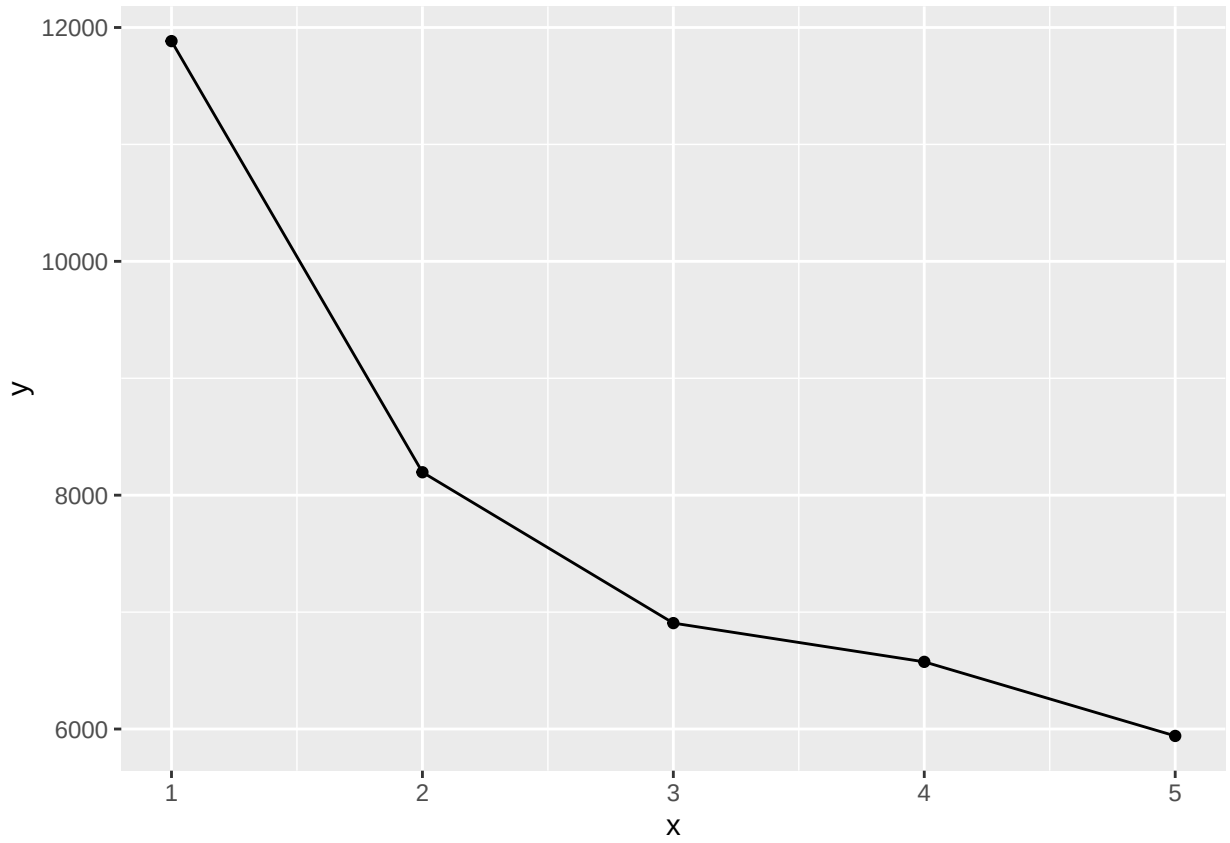
Thanks to this count, we can confirm that the time it takes for the main function to evaluate the cases k=1,2,3,4,5, is 10.17197 minutes. Of course, when we implement this same function with parallelism and threads, a lower time will be expected.

5.- Plot the results of the elbow graph.

For the elbow graph, we created a data frame (named "df") with two vectors: the vector x, corresponding to the X axis, which contains indexes from 1 to the number of centroids; and vector y, corresponding to the Y axis, containing the sum of the errors from the data returned by the main function. Once that was done, we implemented the following ggplot:

```
x=NULL
y=NULL
for (i in 1:length(k_means)) {
  y[i]=sum(k_means[[i]]$error)
  x[i]=i
}

df=data.frame(x,y)
```

```
ggplot(data = df, aes(x=x,y=y)) + geom_point() + geom_line()
```

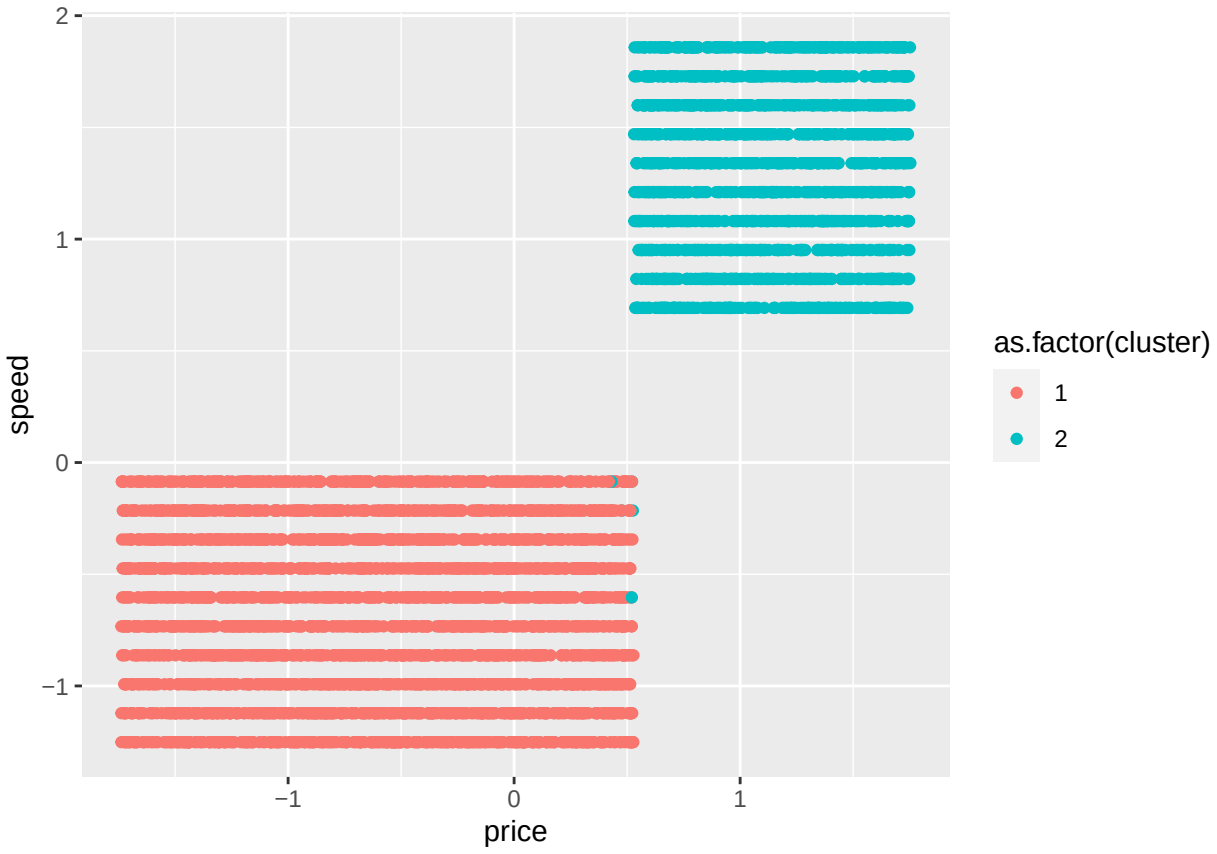


This elbow graph suggests that the optimal number of clusters is k=2, since from k=2 the angle of the graph tends to change rather slowly, while from k=1 to k=2 that change in the slope of the curve is quite big, forming an "elbow" that allows us to confirm that the optimal number of clusters is 2.

6.- Plot the first 2 dimensions of the clusters.

As we stated before, having the results stored in a variable can be quite useful and this is one of those cases. In order to plot the first 2 dimensions of clusters, we just had to plot the following:

```
ggplot(k_means[[2]],aes(x=price,y=speed,color=as.factor(cluster))) + geom_point()
```



7.- Find the cluster with the highest average price and print it.

Here, we created a function to find the cluter with the highest average price. In this function you need to enter the dataset and, after that, it makes a number of computations.

First of all, it creates an empty list and establishes the elements of the last column of the dataset (the column corresponding to the clusters) as factors.

Next, it calculates the number of clusters present in the given dataset. Right after knowing this number, it finds all the observations belonging to each cluster (it is a loop from 1 to k, once for each cluster) and finds the mean of the variable price (the average price) for each of these groups. Then, it returns the list "x" with the average price of each group.

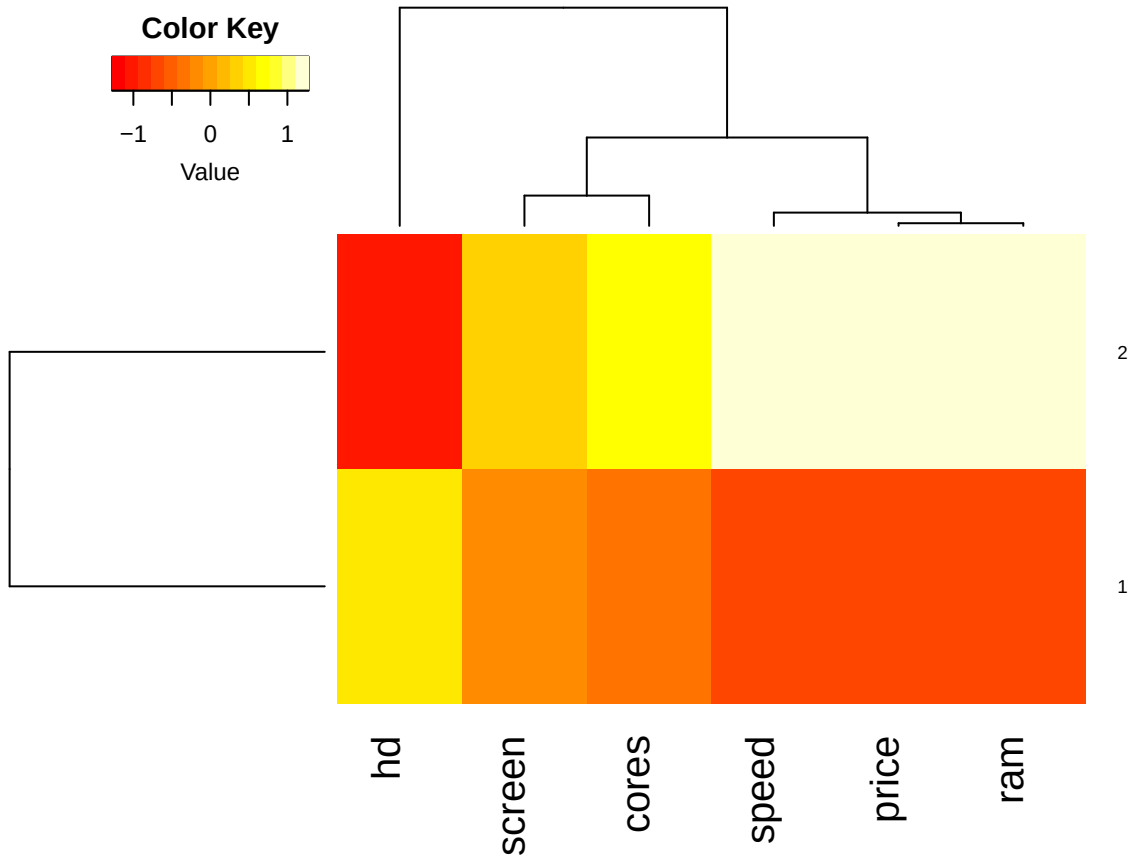
```
hpricefun <- function(datos){
  x = list()
  n = ncol(datos)
  datos[,n] %<>% as.factor()
  k = length(levels(datos[,n]))
  for(i in 1:k){
    ind1 <- which(datos$cluster==i)
    price1 <- datos$price[ind1]
    x[i]=mean(price1)
  }
  return(x)
}
hpricefun(k_means[[2]])
```

```
## [[1]]
## [1] -0.5990875
##
## [[2]]
## [1] 1.134383
```

It is clear that the highest average price corresponds to the second case, i.e., k=2.

8.- Print a heat map using the values of the clusters centroids.

```
clustersum=k_means[[2]] %>% group_by(cluster) %>%  dplyr::summarize(price=mean(price),
                                                                    speed=mean(speed),
                                                                    hd=mean(hd),
                                                                    ram=mean(ram),
                                                                    screen=mean(screen),
                                                                    cores=mean(cores)) %>%
  dplyr::select(-1) %>% as.matrix()

gplots::heatmap.2(x=clustersum,scale = "none",cexRow = 0.7,trace="none",density.info = "none")
```



Implementation in R: Part two - Parallel implementation, multiprocessing

1.- Write a parallel version of your program using multiprocessing.

2.- Measure the time and optimize the program to get the fastest version you can.

For this part of the project, all the first steps and identical. What is different is the call to the main function, where we had to implement the "ClusterExport" function.

That being said, here is how to implement it:

```
library(plyr)
library(dplyr)
library(factoextra)
library(magrittr)
library(ggplot2)
library(microbenchmark)
library(parallel)
library(doParallel)
library(foreach)

set.seed(15)

data5k=read.csv(file = "computers5k.csv",header = T)
data5k$id = NULL
data5k$cd %<>% mapvalues(from = c("yes","no"), to = c("1","0"))  %>% as.factor()
data5k$laptop %<>% mapvalues(from = c("yes","no"), to = c("1","0")) %>% as.factor()
data5k$trend %<>% as.factor()

summary(data5k)
```

```
##      price          speed             hd             ram          screen
##  Min.   : 500   Min.   :15.00   Min.   : 1.00   Min.   : 2.0   Min.   :11.00
##  1st Qu.:1479   1st Qu.:18.00   1st Qu.: 6.50   1st Qu.: 6.0   1st Qu.:15.00
##  Median :2398   Median :22.00   Median :12.00   Median :14.0   Median :17.00
##  Mean   :2416   Mean   :24.66   Mean   :13.19   Mean   :18.2   Mean   :17.69
##  3rd Qu.:3354   3rd Qu.:32.00   3rd Qu.:20.00   3rd Qu.:28.0   3rd Qu.:21.00
##  Max.   :4349   Max.   :39.00   Max.   :28.00   Max.   :60.0   Max.   :23.00
##      cores       cd       laptop   trend
##  Min.   : 4.00   0:3787   0:4313   1:1457
##  1st Qu.: 8.00   1:1213   1: 687   2:1436
##  Median :14.00                     4:1072
##  Mean   :15.52                     5:1035
##  3rd Qu.:22.00
##  Max.   :30.00
```

```
#kmeans only work with numeric vectors
data_wo_factors = data5k %>% dplyr::select(c(-cd,-laptop,-trend))

# 1.- Write a parallel version of you program using multiprocessing

############## K means DIY Process ####################

#Used to generate random numbers
```

```
generate_random=function(vector){
  return(runif(1,min(vector),max(vector)))
}

euclidian=function(a,b){
  sqrt(sum((a-b)^2))
}

kmeans_diy=function(data,k){

  #Scale data
  kmeans_data=as.data.frame(scale(data))

  #Generate random centroids
  X=matrix(nrow=k,ncol=ncol(kmeans_data)+1)
  clusters=letters[1:k]
  for (i in 1:nrow(X)) {
    for(j in 1:ncol(kmeans_data)){
      X[i,j]=generate_random(kmeans_data[,j])
    }
  }
  X[,ncol(kmeans_data)+1]=as.factor(letters[1:k])


  #Compute Distances
  n=ncol(kmeans_data)
  m=nrow(X)
  nX=ncol(X)
  x=matrix(nrow = nrow(kmeans_data),ncol = m)
  for(i in 1:m){
    x[,i]=apply(X =kmeans_data,MARGIN = 1,FUN = euclidian,b=X[i,-nX])
  }
  for(i in 1:nrow(kmeans_data)){
    kmeans_data$error[i]<-min(x[i,])
    kmeans_data$cluster[i]<-which(x[i,]==min(x[i,]))
  }
  x=NULL

  #Check errors
  error=c(0,sum(kmeans_data$error))
  e=2

  while(round(error[e],0)!= round(error[e-1],0)){


    X=as.data.frame(dplyr::ungroup(dplyr::select(plyr::mutate(.data = dplyr::summarize(.data=dplyr::grou
                                                                                       price=mean(price)
                                                                                       speed=mean(speed)
                                                                                       hd=mean(hd),
                                                                                       ram=mean(ram),
                                                                                       screen=mean(scree
                                                                                       cores=mean(cores)
                                                                  n_centroide=cluster),-cluster)))
```

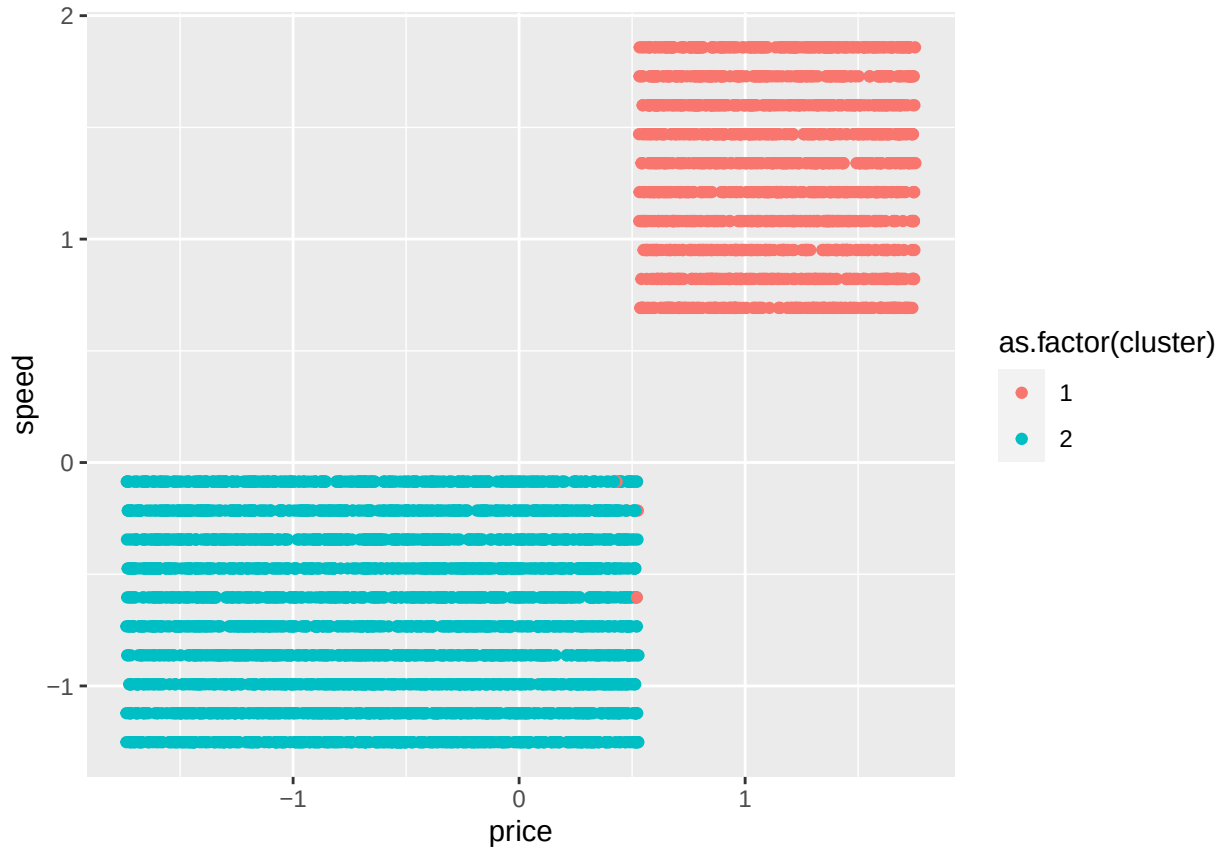
```
    #Compute distances
    n=ncol(kmeans_data)-2
    m=nrow(X)
    nX=ncol(X)
    x=matrix(nrow = nrow(kmeans_data),ncol = m)
    for(i in 1:m){
      x[,i]=apply(X = kmeans_data[,-c(7,8)],MARGIN = 1,FUN = euclidian,b=X[i,-nX])
    }
    for(i in 1:nrow(kmeans_data)){
      kmeans_data$error[i]<-min(x[i,])
      kmeans_data$cluster[i]<-which(x[i,]==min(x[i,]))
    }
    x=NULL

    #Write error
    error=c(error,sum(kmeans_data$error))

    #Next iteration
    e=e+1
    #
    print(e)
    #
  }

  return(kmeans_data)
}



no_cores=detectCores()
clust=makeCluster(no_cores)
clusterExport(clust,"data_wo_factors",envir = environment())
clusterExport(clust,"generate_random",envir = environment())
clusterExport(clust,"euclidian",envir = environment())

Start <- Sys.time()
k_means_mp=parLapply(cl = clust,X = 1:5,fun = kmeans_diy,data=data_wo_factors)
end <- Sys.time()


stopCluster(clust)
```

Again, we opted for computing the measuring of the time and the main function at the same time. The main function of the k-means algorithm is now called through a parLapply function.

The measured time is now of 8.187165 minutes, which is smaller than the time with the serial version.

3.- Plot the first 2 dimensions of the clusters.

```
ggplot(k_means_mp[[2]],aes(x=price,y=speed,color=as.factor(cluster))) + geom_point()
```



4- Find the cluster with the highest average price and print it.

```
hpricefun <- function(datos){
  x = list()
  n = ncol(datos)
  datos[,n] %<>% as.factor()
  k = length(levels(datos[,n]))
  for(i in 1:k){
    ind1 <- which(datos$cluster==i)
    price1 <- datos$price[ind1]
    x[i]=mean(price1)
  }
  return(x)
}

hpricefun(k_means_mp[[2]])
```

```
## [[1]]
## [1] 1.134383
##
## [[2]]
## [1] -0.5990875
```

5.- Print a heat map using the values of the clusters centroids.

```
clustersum=k_means_mp[[2]] %>% group_by(cluster) %>%  dplyr::summarize(price=mean(price),
                                                                       speed=mean(speed),
                                                                       hd=mean(hd),
                                                                       ram=mean(ram),
                                                                       screen=mean(screen),
                                                                       cores=mean(cores)) %>%
  dplyr::select(-1) %>% as.matrix()

gplots::heatmap.2(x=clustersum,scale = "none",cexRow = 0.7,trace="none",density.info = "none")
```