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 "computers5k.csv" is about a list of computers and it contains some information about them: "id","speed","hd","ram","screen","cd","multi","premium","ads","trend".

Implementation in R: Part one - Serial version

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.

2.- Implement the k-means algorithm.

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 and 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){
  return(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){
  kmeans_data=as.matrix(scale(data_wo_factors))
  colX=ncol(kmeans_data)
  rowX=k
  X=matrix(ncol = colX,nrow = rowX)
  for(i in 1:rowX){
    X[i,]=apply(X=kmeans_data,MARGIN = 2,generate_random)
  }
  X=cbind(X,1:2)
  centroids_equal=FALSE
  x=matrix(ncol=k,nrow=nrow(kmeans_data))
  ncolX=ncol(X)
  nrowkmeans=nrow(kmeans_data)
```

```
  count=0
  err=0
  while(centroids_equal==FALSE){
    count=count+1
    x=matrix(ncol=k,nrow=nrow(kmeans_data))
    for(i in seq_len(k)){
      x[,i]=apply(X=kmeans_data,MARGIN = 1,FUN = euclidian,b=X[i,-ncolX])
    }
    cluster=c()
    error=c()
    for(i in 1:nrowkmeans){
      error[i]<-min(x[i,])
      cluster[i]<-which(x[i,]==min(x[i,]))
    }

    kmeans_data=cbind(kmeans_data,error,cluster)

    X_new = kmeans_data %>% as.data.frame() %>% 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.matrix()


    if(round(sum(error),0)==round(err,0)){
      centroids_equal=TRUE
    }else{
      X=X_new
      kmeans_data=kmeans_data[,-(7:8)]
      err=sum(error)
      X_new=NULL
      x=NULL
    }
    print(count)
  }
  return(as.data.frame(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 and then add another row representing the cluster assigned to each centroid. After that, we select the centroids as we previously explained and create this new column.

Now, it is time to do clustering. First we need 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.

After that, the cluster centroids are reassigned towards the mean of each variable and we compute again the distances to this new centroids.

This process is repeated until the error between the previous interaction and the actual one is not significant.

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.

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)
stop=Sys.time()
```

```
print(stop-start)
```

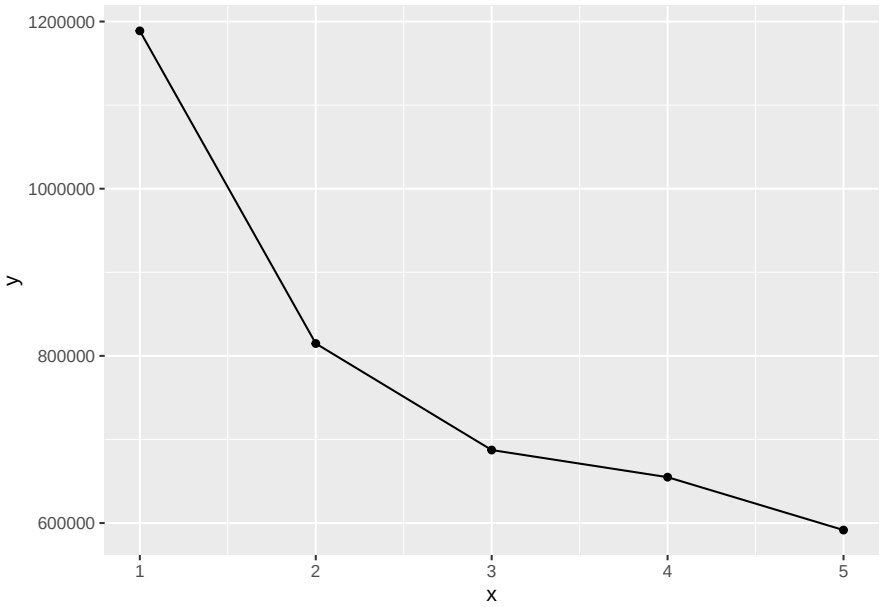
```
## Time difference of 10.29063 mins
```

Thanks to this count, we can know the time it takes for the main function to evaluate the cases k=1,2,3,4,5. 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:

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

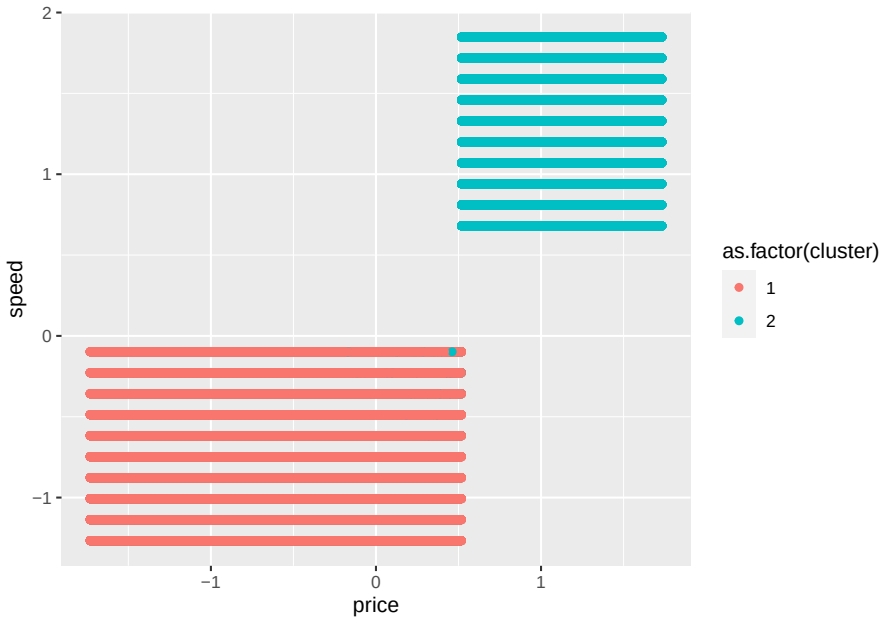


This elbow graph suggests that the optimal number of clusters is k=2, since from k=2 the angle of the graph tends to change rather slowly, while from k=1 to k=2 that change in the slope of the curve is quite big, forming an "elbow" that allows us to confirm that the optimal number of clusters is 2.

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.

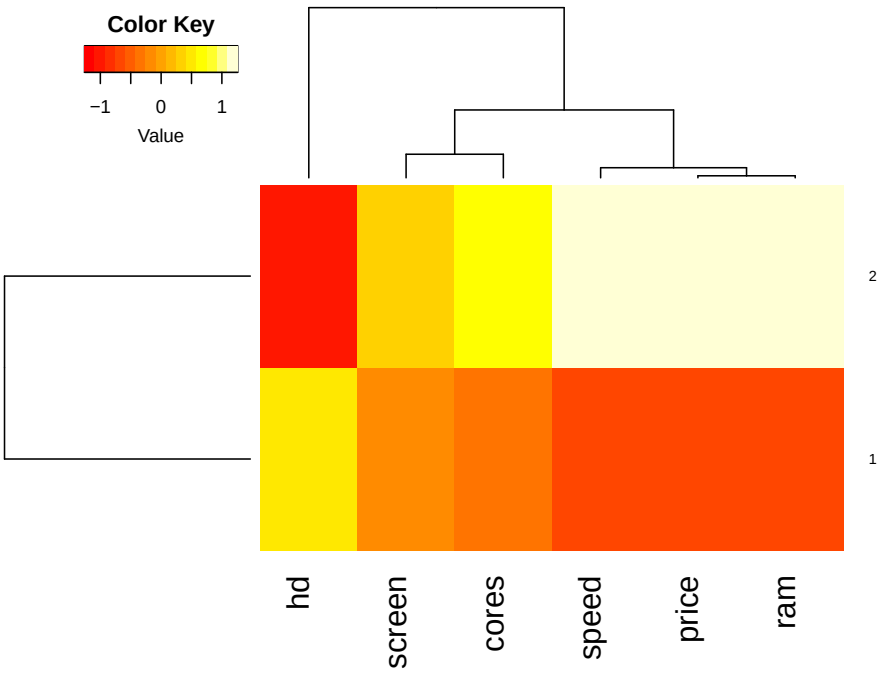
```
hpricefun <- function(datos){
  x = list()
  n = ncol(datos)
  datos[,n] %<>% as.factor()
  k = length(levels(datos[,n]))
  for(i in 1:k){
    ind1 <- which(datos$cluster==i)
    price1 <- datos$price[ind1]
    x[i]=mean(price1)
  }
  return(x)
}
hpricefun(k_means[[2]])
```

```
## [[1]]
## [1] -0.6082116
##
## [[2]]
## [1] 1.123766
```

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.

For this part of the project, the function that we will use to compute kmeans is the same as the one seen before. Only few changes were made on the centroids recalculation, because we were having trouble using the pipe notation "%>%". What is different is the call to the main function, made to compute the elbow graph and to choose the best k possible. For this point we had to implement the "ClusterExport" function to export all the variables and functions that could be used by the different processes.

That being said, here is how to implement it:

```
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_mp,data=data_wo_factors)
end <- Sys.time()


stopCluster(clust)
```

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

Again, we chose to compute 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, which is made to allow parallelism in R.

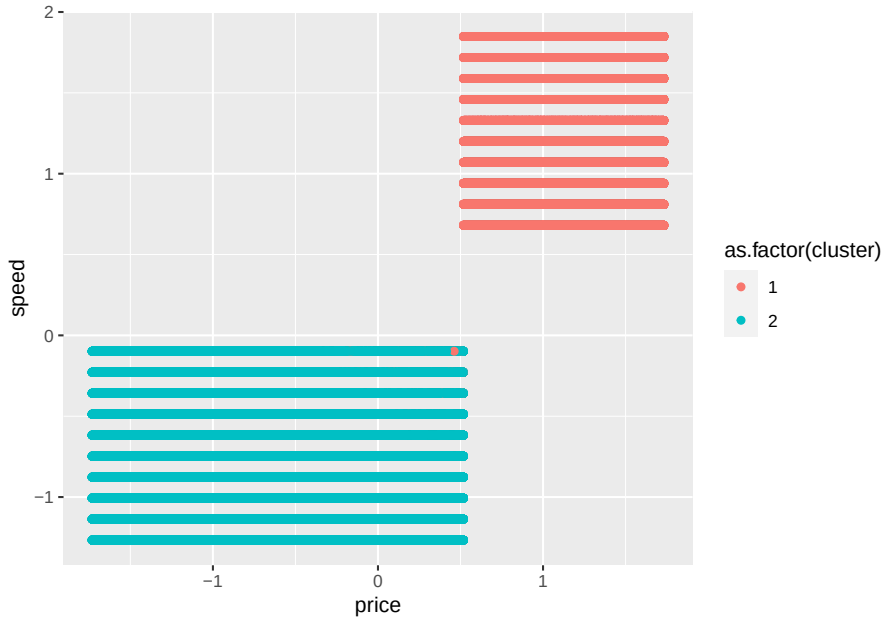
```
end-Start
```

```
## Time difference of 6.115963 mins
```

The measured time is now is about two times faster 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.

The function used is exactly the same as in the serial version, but this time applied to the dataset generated with the multiprocess k-means function.

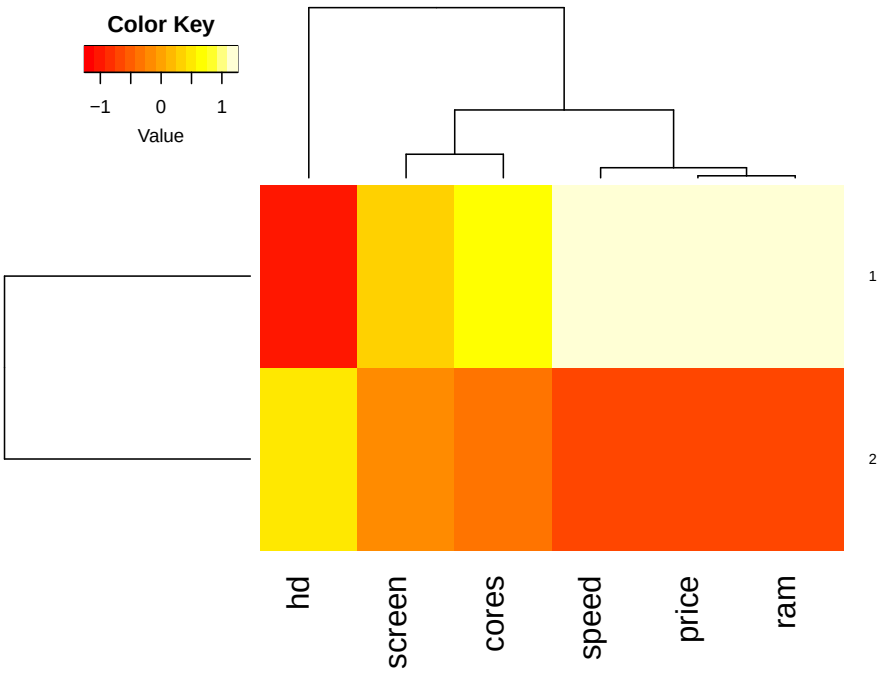
```
hpricefun(k_means_mp[[2]])
```

```
## [[1]]
## [1] 1.123766
##
## [[2]]
## [1] -0.6082116
```

Here, the highest average price corresponds to the case k=1, i.e., with 1 cluster.

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

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



Implementation in R: Part three – Parallel implementation, threading

1.- Write a parallel version of you program using Threads.

For the parallel version using threads we tried two different approaches. First we changed the main function to compute knn means and add parallelism for calculating euclidean distances. We the tested this function for different values of k (as we did on the serial version) and the time of execution was about 7.8 minutes.

```
no_cores=detectCores()
kmeans_diy_th=function(data,k){
 kmeans_data=as.matrix(scale(data_wo_factors))
  colX=ncol(kmeans_data)
  rowX=k
  X=matrix(ncol = colX,nrow = rowX)
  for(i in 1:rowX){
    X[i,]=apply(X=kmeans_data,MARGIN = 2,generate_random)
  }
  X=cbind(X,1:2)
  centroids_equal=FALSE
  count=0
  err=0
  nrowX=nrow(X)
  ncolX=ncol(X)

  while(centroids_equal==FALSE){
    count=count+1
    #Threads
    clusts=makeCluster(no_cores,type = "FORK")
    registerDoParallel(clusts)
    x=foreach(i=1:nrowX,.combine = cbind) %dopar% apply(X =kmeans_data,MARGIN = 1,FUN = euclidian,b=X[i
    stopCluster(clusts)
    #
    cluster=c()
```

```
    error=c()
    for(i in 1:nrowkmeans){
      error[i]<-min(x[i,])
      cluster[i]<-which(x[i,]==min(x[i,]))
    }
    kmeans_data=cbind(kmeans_data,error,cluster)

    X_new = kmeans_data %>% as.data.frame() %>% 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.matrix()

    if(round(sum(error),0)==round(err,0)){
      centroids_equal=TRUE
    }else{
      X=X_new
      kmeans_data=kmeans_data[,-(7:8)]
      err=sum(error)
      X_new=NULL
      x=NULL
    }
    print(count)
    }
  return(as.data.frame(kmeans_data))
}
```

Then, as we wanted to compare the speed of processing with the rest of cases we also kept the function "kmeans_diy()" that we created initially and compute the algorithm to obtain the elbow graph in parallel using threads.

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

```
no_cores=detectCores()
clust=makeCluster(no_cores,type = "FORK")
registerDoParallel(clust)


#Do function to obtain elbow graph in parallel
obtain_k_optimal_th=function(k,data){
  k_means_th=foreach(i=1:k) %dopar% kmeans_diy(data,i)
}

#MEASURE TIME
start=Sys.time()
k_means_th=obtain_k_optimal_th(5,data_wo_factors)
stop=Sys.time()
```

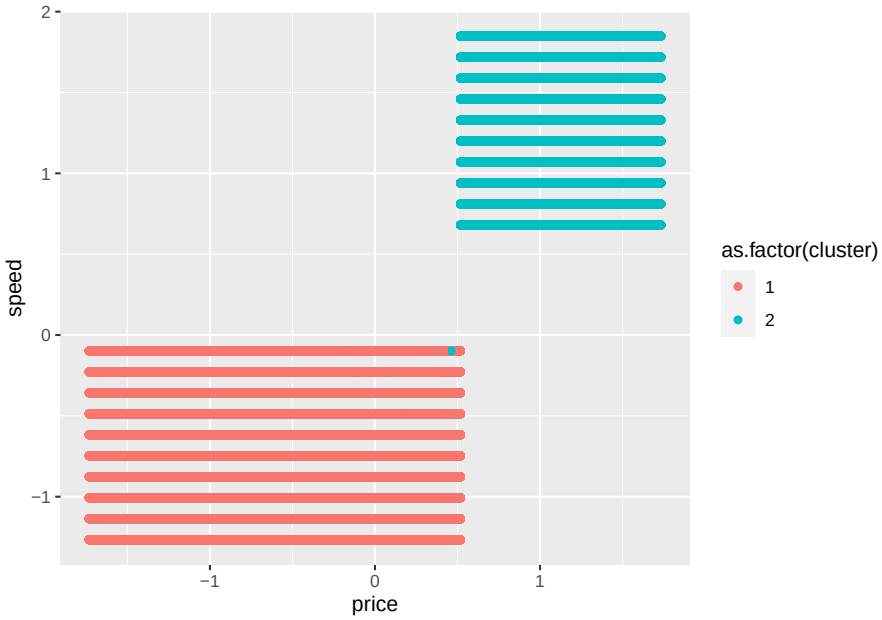
```
stopCluster(clust)

print(stop-start)
```

```
## Time difference of 4.074853 mins
```

In this approach we got that the time of execution was a little bit faster than the multiprocess version, and much faster than the serial one, as expected.

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

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



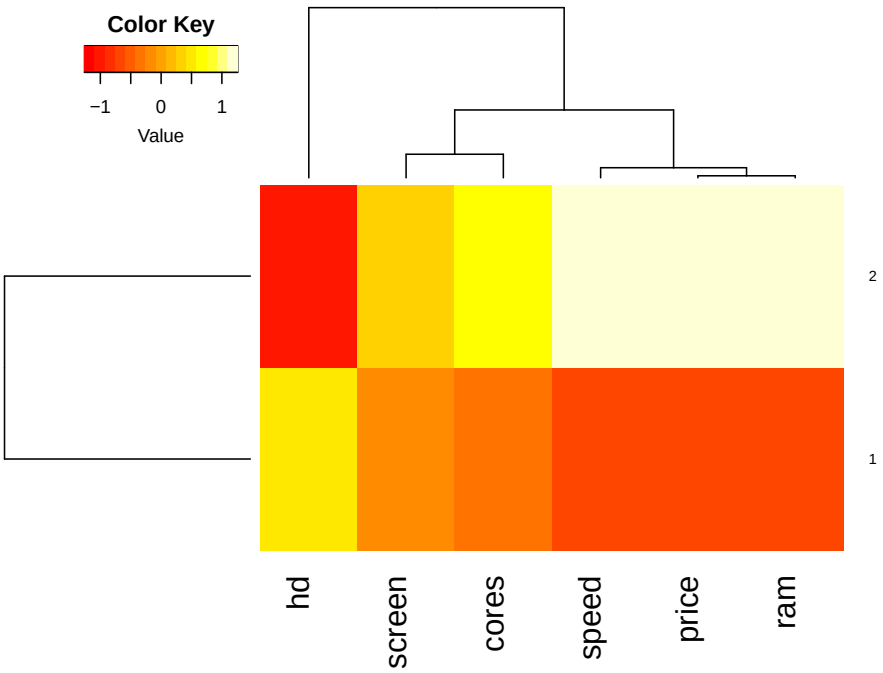
4- Find the cluster with the highest average price and print it.

```
hpricefun(k_means_th[[2]])
```

```
## [[1]]
## [1] -0.6082116
##
## [[2]]
## [1] 1.123766
```

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

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


Color Key
−1
0
1
Value
2
1
hd
screen
cores
speed
price
ram