LAB1. K-MEANS PARALLELIZATION in R and PYTHON

Memory

Luis Angel Rodriguez Gracia, Sara Dovalo del Río and Alejandra Estrada Sanz

15/03/2022

1. Serial version

1.1. k-means algorithm

El algoritmo k-means es un método de agrupamiento cuyo objetivo es dividir un conjunto de datos en un determinado número de grupos, donde cada observación del conjunto de datos pertenece al grupo cuyo valor medio es más cercano a dicha observación.

Para implementar este algoritmo hemos creado una función llamada "custom_kmeans" la cuál depende de tres parámetros, el dataset a estudiar, "X", el número de grupos o clusters en lo que se quiere dividir dicho dataset, "k", y la semilla que vamos a utilizar, "seed_value".

Lo primero que hacemos en nuestra función es determinar el número de filas y columnas que tiene nuestro dataset, "n" y "p", y a continuación creamos la matriz "assig_cluster" de dimensiones similares a nuestro dataset pero con una columna más en la cual especificaremos a que cluster pertenece cada uno de los puntos de nuestro dataset. El siguiente paso es crear un escalar lógico, "centroids_not_equal", al que designaremos por defecto el valor TRUE. Luego hemos creado el escalar numérico, "ite", con el objetivo de ir midiendo las interacciones que necesita llevar a cabo nuestra función. Seguidamente establecemos la semillas que vamos a utilizar, "seed_value", y creamos el vector "centroids_index" eligiendo aleatoriamente "k" números de entre las "n" observaciones de nuestro dataset. Una vez escogidos aleatoriamente los centroides recogemos en la matriz "centroids", de dimensiones "k" y "p", todas las componentes de dichos centroides.

El siguiente paso es crear un bucle "while" que mientras "centroids_not_equal" sea TRUE seguirá ejecutandose indefinidamente y dentro de este bucle creamos la matriz "distance_cluster" de dimensiones "n" y "k".

A continuación creamos otro bucle, dentro del primero, que recorre los valores de "k", dentro del cual rellenamos la matriz "distance_cluster" con la distancia en módulo de cada punto a cada uno de los centroides que hemos elegido antes aleatoriamente.

Cerramos el segundo bucle, creamos el vector "cluster" de longitud "n" y abrimos un tercer bucle dentro del bucle "while", el cual recorre los valores de "n". Dentro de este tercer bucle rellenamos el vector "cluster" con el número de cluster al cuál pertenece cada punto, es decir, el cluster al cual la distancia es menor. Para implementar este paso en Python hemos necesitado de una única linea de código, mientras que en R han sido necesarias algunas más y hacer uso de la función "if". Una vez rellenado "cluster" cerramos el tercer bucle, integramos estos datos en la matriz "assig_cluster" añadiendo a nuestro dataset "X" una columna más con los valores del vector "cluster", y creamos la matriz "new_centroids" de iguales dimesiones a la matriz "centroids".

Seguidamente creamos un cuarto bucle, también dentro del primero, que recorre de nuevo los valores de "k", mediante el cual rellenamos la matriz "new_centroids" con las coordenadas de los nuevos centroides, es decir, los centroides de los clusters que acabamos de crear. Dichas coordenadas las calculamos haciendo la media de cada variable en cada uno de los clusters asignados en el paso anterior. En este caso implementar el calculo de las coordenadas de los nuevos centroides en R requiere de un única línea de codigo, mientras que esta vez es en Python donde hemos necesitado de pasos intermedios para implementar este código.

Cerramos este cuarto bucle y utilizamos la función "if" para cambiar el valor de "centroids_not_equal" a FALSE si el vector "centroids" y el vector "new_centroids" son iguales y si esta condicion no ocurrieran con el comando "else" sobreescribimos el vector "centroids" con los valores del vector "new_centroids". Con estas cuatro líneas de código lo que conseguimos es comparar los centroides iniciales con los que hemos empezado el proceso con los que hemos creados tras asignar los clusters a los puntos, la idea es cada vez que ejecutamos este proceso los clusters sean cada vez más óptimos y más diferenciados entre si por lo que los centroides iniciales y finales irán cambiando, y el bucle se seguirá ejecutando hasta que demos con los clusters óptimos y los centroides de los que hemos partido sean los mismo que calculamos a traves de las medias de los cluster, ya que la disposición de los clusters, al haber alcanzado la forma óptima, permanece invariable. En este momento el vector lógico "centroids_not_equal" tomará el valor FALSE, el bucle inicial "while" dejará de ejecutarse y habremos obtenido nuestros clusters óptimos.

Antes de cerrar el bucle "while" hemos ido sumando 1 al escalar "ite" para poder ir midiendo en cuantas iteracciones se lleva a cabo el proceso. Por último, fuera del bucle pedimos a la función que nos devulva la matriz "assig_cluster", es decir, nuestro dataset orginal pero con una columna más donde se especifíca el cluster final al que pertenece cada punto.

1.2. Elbow graph

The elbow graph is a method used to determine the optimal number of clusters for a set of data. The resultant plot represents the sum of squared distances between points belonging to the same cluster over the total number of clusters.

Generally speaking, the graph has a steep slope at the beginning, due to the difference of distances between having one and two clusters is significant, and little by little the slope is getting less steep. Therefore, the larger the number of clusters the more softened the slope (smaller sum of squared errors to each centroid). We are looking for the elbow point, this means that we are searching for the point where there is a significant change in the slope. The associated total number of clusters of this point is the optimal number of groups for the given dataset.

We have implemented the `elbow_graph` function which helps us to create the elbow graph. This function depends on three parameters:

- The dataset we want to analyze, identified as `X`.
- The total number of cluster subject to study, identified as `total_k`.
- The seed value in order to reproduce the executions, identified as `seed_value`.

The first thing to do is to get the number of rows (n) and the number of columns (p). Later on, we initialize the vector `sum_sq_dist_total` resultant with the max number of clusters obtaining thanks to the parameter `total_k`. In this variable we will store for each number of clusters the total sum of squared distances in each centroid to the belonging points.

The next step is about executing the method k-means, which we have already implemented, for each number of clusters with the given data and the given seed value. The execution of this method produces a list which contains for each number of cluster the resultant matrix with the observations associated to each cluster.

In R we have used the method `lapply` to execute this method as many times as the value of the parameter `total_k` is defined, meanwhile, in Python this functions is not available and we need to defined an explicit for-loop which iterates `total_k` times.

The final step is to calculate the sum of squared distances for each possible number of groups using the list of matrices obtained in the previous step (this will be the result of executing the method `elbow_graph`). Therefore, we have defined a for-loops, iterating for each possible number of clusters, and one nested loop which iterates over each cluster inside. In this last loop we calculate each centroid, applying the average function per columns, and their distances to each point belonging to this group. The resultant is the sum of squared distances for the k-th number of groups.

Once the previous step is completed, we will have a list contained the sum of squared distances for each possible group. For instance, if we pick the element of the position 2 in the resultant list we will have the

sum of squared distance associated to the result of executing the k-means with k equal to 2.

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

At this point, we have already implemented the serial versions of the `custom_kmeans` and the `elbow_graph` functions, then we proceed to process our dataset using them. To do this we have to make beforehand some tweaks in our dataset:

- Scaling the data, then all the variables do have the same weight in out dataset. There won't be some variables with more "power" than others when the distance is measured.
- Dropping the categorical variables from the analysis since the algorithm is not working properly when categorical variables are presented. It is not possible to measure the distance between categories.

After making these changes, we have proceeded to apply the `custom_kmeans` function to our dataset for a value of `k` equal to 2, since, as we will see later, when we represent the elbow graph, 2 is the optimal number of clusters for our dataset.

1.4. Measure time

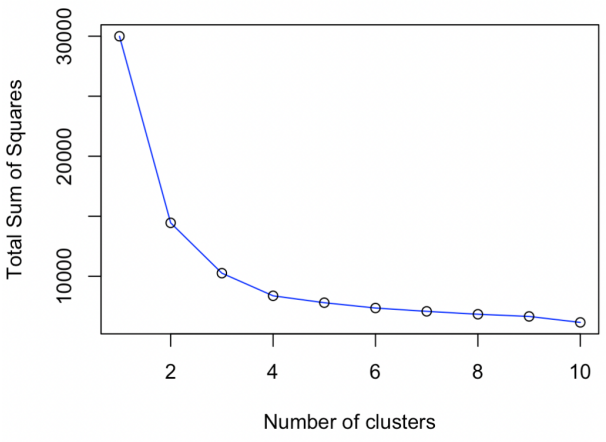
Regarding the measurement of time, our results for a dataset of 500,000 rows have been shown that:

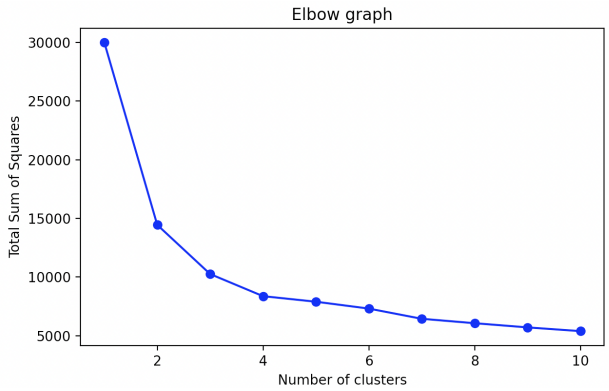
- For the `custom_kmeans` function:
  - In R
    - Call the function once took 5.407493 seconds
    - Call the function ten times took 213.03666 seconds
  - In Python:
    - Call the function once took 8.744184732437134 seconds
    - Call the function ten times took 345.5383791923523 seconds
- For the `elbow_graph` function:
  - In R
    - Call the function elbow graph took 187.9113 seconds
  - In Python
    - Call the function once took 346.7162780761719 seconds

These results make sense since we have assessed the time consumption when we call 10 times the `custom-kmeans` and the result is slightly smaller than the time used to execute the method `elbow_graph` which calls the function also k times.

1.5. Plot the results of the elbow graph

In the following images we can see the elbow graph that is produced after applying the function `elbow_graph` to our dataset, in both R and Python:

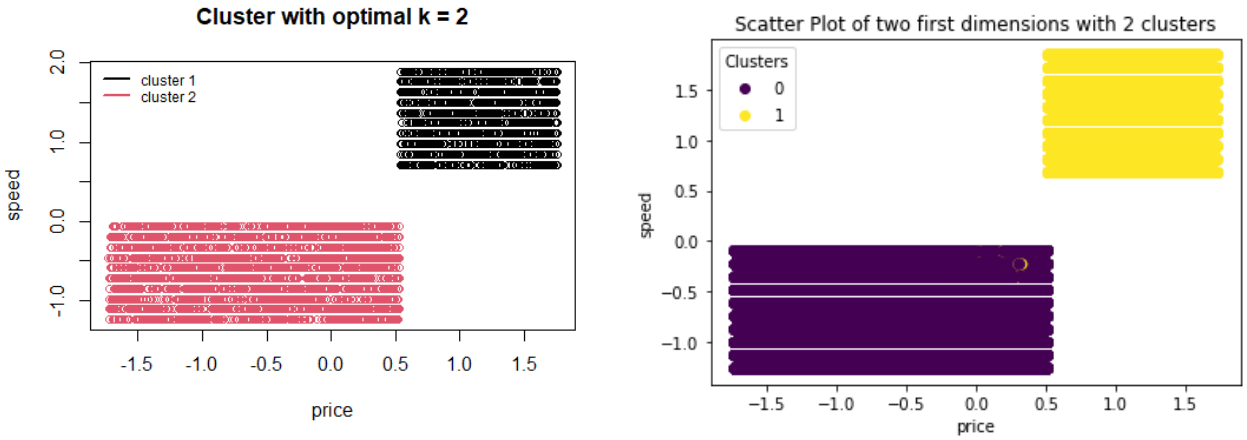




In the graphs above it can be seen that the elbow point, that is the point where the graph presents an "elbow" due to a significant change in slope, is associated with k = 2, so the optimal number of clusters for our dataset will be 2, as we mentioned before.

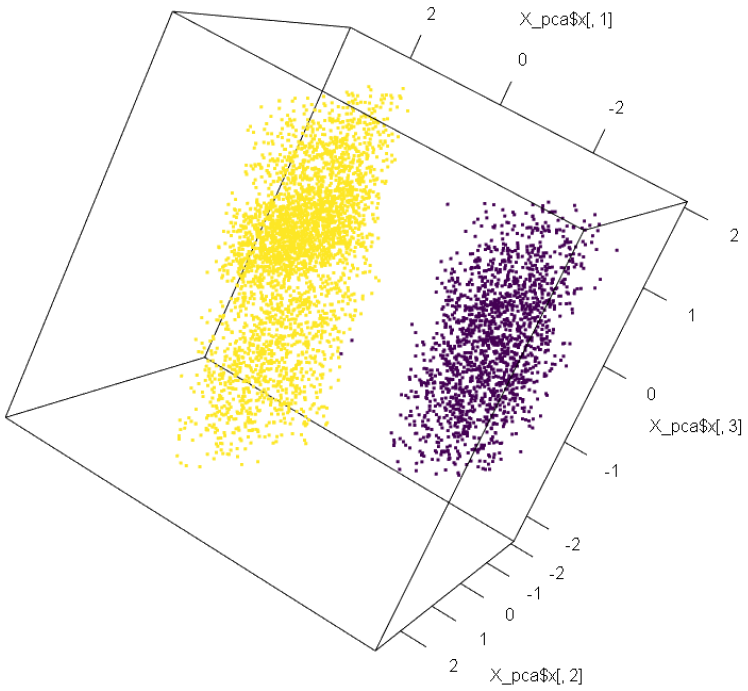
1.6. Plot the first 2 dimensions of the clusters

If we represent the first two dimensions of our dataset, "price" and "speed", for the two clusters created previously with the "custom_kmeans" function, we obtain the following graphs, in both R and Python:



Despite the fact that two dimensions are too few to determine the clusters of a dataset of six variables, we can observe that when we graph these two variables together, two differentiated groups of data are formed that correspond practically 100% with the clusters that we have created, which gives us an idea that our clusters are probably an optimal solution for our dataset.

As we have said, two dimensions are not enough to appreciate the true nature of the clusters, so in the following image we have decided to plot the first three PCA of our dataset for our two clusters, in order to observe them better:



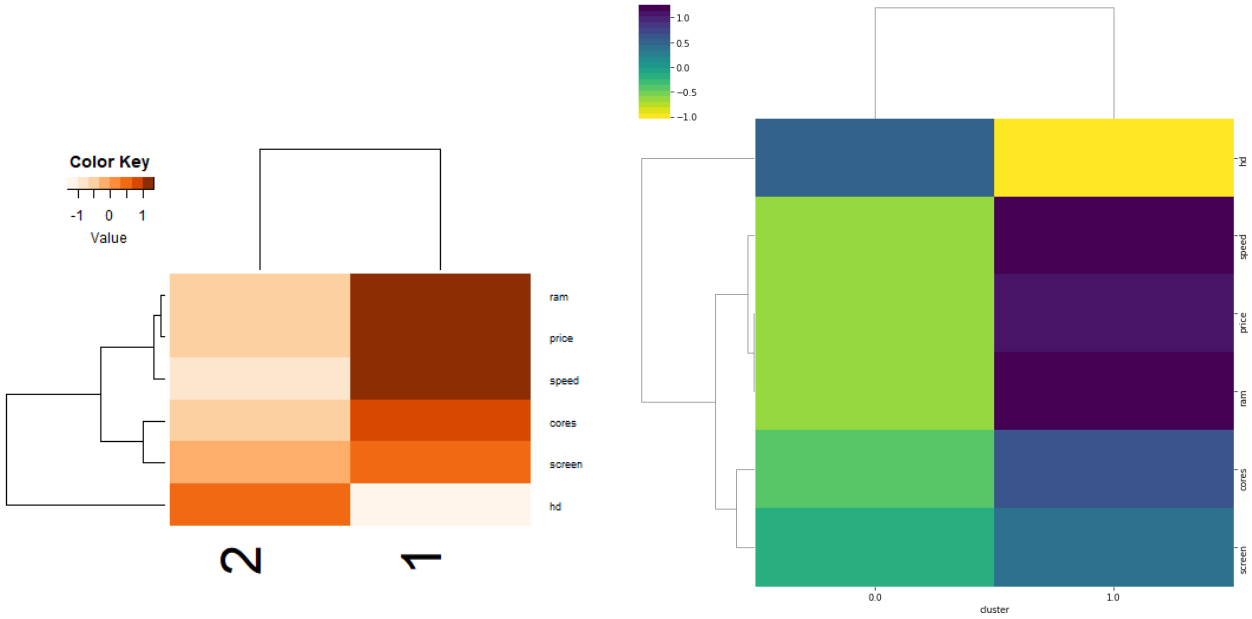
In this graph we can observe two different groups of data that again correspond almost 100% with our clusters, which again leads us to the conclusion that we have created optimal clusters already suitable for our dataset.

1.7. Find the cluster with the highest average price and print it

After calculating the average price of both clusters, we have obtained that the cluster with the highest average price is cluster 1, as we had already been able to observe when we plotted the variables "price" and "speed" for our two clusters.

1.8. Print a heat map using the values of the clusters centroids

Next we have created a heat map using the values of the clusters centroids, in both R and Python:



In this graphs we can observe the dependency of belonging to each cluster depending on each of the variables of the dataset. Cluster 1 is characterized as having the highest average for "ram", "price", "speed", "cores" and "screen", while cluster 2 has the highest average for "hd".

2. Parallel implementation, multiprocessing

Our function in this case using multiprocessing will be called **"custom_kmeans_mp"** which, as we said before, will be in charge of dividing our dataset into a certain number of groups, where each observation of the dataset belongs to the group whose average value is closest to that observation.

The first thing we will do is to detect the number of logical cores in the machine using detectCores(). We have selected half of the cores since this is usually what works best to keep the operating system and the rest of the tasks of our machine running normally. To this quantity we will assign the name "num_cores".

Next, we will assign a new variable "par_cluster" to create the cluster, where we specify the number of threads that we are going to use by means of the function makeCluster().

In what follows, we use the "foreach" library, which, as we said before, allows us to process loops in parallel.

That is, we will go from calculating the distance in modulus from each point to each of the centroids (previously chosen randomly) using a loop to perform it in parallel using the foreach() function and we will choose the minimum distance, now using parApply() (function are similar to those known from the base of R that takes care of all the heavy work in the parelization) instead of performing a loop going through each of the calculated distances.

We then calculate the cluster number to which each point belongs, i.e. the cluster to which the distance is smaller and store it in the variable "cluster" using the max.col() function. As in the serial case, we add a column to our dataset (variable "assig_cluster") where each point is assigned the cluster for which its distance to the respective centroid is the minimum.

Another modification would be to calculate the new centroids instead of using an iterative process, also using the foreach() function.

Finally we will use the autostopCluster() function which must be executed to close all R environments created in the threads (if this is not done, our subsequent procedures may present problems).

On the other hand to perform the elbow graph using multiprocessing we build the function **"elbow_graph_mp"** and which differs from the previous function performed serially in the following

aspects:

- As in the case of the previous function, we will first detect the number of logical cores in the machine using detectCores(). We have selected half of the cores and to this amount we will assign the name "num_cores".
- Then we apply to our dataset the function "custom_kmeans" for each of the values of "total_k" (column corresponding to the cluster assigned to each point). To apply this function in parallel we will use the function "parLapply" in R, while in Python. . . .
- The iterative process performed to obtain "sum_sq_dist_total" (squared distances between each point and the centroid of the cluster associated to that point for each of the cluster values we want to study) in our serial version will be replaced by the foreach() function which will allow us to determine the optimal number of clusters for our dataset by command "%dopar%" which performs the same steps as the "elbow_graph" function but in parallel.
- Finally we will use the autostopCluster() function which must be executed to close all the R environments created in the threads.

2.1. Write a parallel version of you program using multiprocessing

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

2.3. Plot the first 2 dimensions of the clusters

2.4. Find the cluster with the highest average price and print it

2.5. Print a heat map using the values of the clusters centroids

3. Parallel implementation, threading

3.1. Write a parallel version of you program using threads

In this section we have implemented two different approaches depending on the programming language we are working with.

First we are going to describe the case when the programming language is R, it doesn't allow to use a proper thread, then what we are going to use are clusters of type *fork*. This type of parallelization copies the entire current version of R and moves it to a new core. They are usually faster than sockets, the approach described in the multiprocessing point, nevertheless they are only available in POSIX systems (MacOS, Linux and so on). The way we have implemented them is very similar that the way we have implemented the multiprocessing, although in this case we have to explicitly indicate that we want forking clusters.

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

3.3. Plot the first 2 dimensions of the clusters

3.4. Find the cluster with the highest average price and print it

3.5. Print a heat map using the values of the clusters centroids