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

El gráfico elbow es un método utilizado para determinar el número de clusters de un conjunto de datos. En el gráfico se representa el número de clusters frente a la suma de la distancia al cuadrado entre cada punto del dataset y el centroide del cluster al que se ha asignado dicho punto. En general el gráfico tiene inicialmente una pendiente muy pronunciada, ya que la diferencia de distancias entre tener un cluster o dos es notoría, y poco a poco se irá suavizando dicha pendiente a medida que el número de clusters sea más alto y la diferencia entre trabajar con un cluster más o menos no sea tan significativa en las distancias. El punto que buscamos haciendo este tipo de analísis se denomina elbow point, es decir, el punto donde el gráfico presenta un "codo" debido a un cambio significativo en la pendiente, dicho punto estará asociado al número de clusters optimos para nuestro dataset.

Hemos denominado "elbow_graph" a la función que hemos construido para crear nuestro gráfico elbow. Dicha función depende de tres parámetros, el dataset que se quiere analizar, "X", el número máximo de clusters a estudiar, "total_k", y la semilla que vamos a utilizar, "seed_value". Lo primero que hacemos en nuestro código es determinar el número de filas y columnas que tiene el dataset, "n" y "p", y crear un vector, "sum_sq_dist_total", de logitud "total_k", donde posteriormente guardaremos la suma de las distancias al cuadrado una vez calculada en función de "total_k". A continuación aplicamos a nuestro dataset la función "custom_kmeans", cuya contrucción explicaremos en el próximo apartado, para cada uno de los valores de "total_k", lo cual nos devuelve el propio dataset con una columna más correspodiente al cluster que se le ha asignado a cada punto. Para aplicar a esta función cada valor de "total_k" utilizamos la función "lapply" en R, mientras que en Python creamos un bucle que recorre todos los valores de "total_k". Una vez tenemos ya los datos asignados a clusters dentro del bucle que recorre los valores de "total_k" creamos un escalar denominado "sum_sq_distance" donde recogeremos la suma de las distancias al cuadrado para cada valor de "total_k". Seguidamente creamos otro bucle dentro del primero que recorra los valores del anterior, es decir, recorremos cada uno de los cluster para cada número total de clusters. Dentro de este bucle guardamos los elementos que pertenecen a un mismo cluster en la matriz "elements_cluster" y dentro del vector "centroidekth" calculamos las coordenadas del centroide asociado dicho cluster haciendo la media de las columnas de la matriz "elements_cluster". Una vez ya tenemos los puntos y el centroide de un mismo cluster recogemos en el vector "distance_centroid" la distancia al cuadrado en módulo de cada punto del cluster al centroide asociado, y guardamos en el escalar "sum_sq_distance" la suma total de cada una de las distancias. Ya para finalizar cerramos el segundo bucle y dentro del primero asociamos el escalar "sum_sq_distance" para cada valor de "total_k" a un elemento del vector "sum_sq_dist_total" creado anteriormente. Finalmente cerramos también el primer bucle y pedimos a nuestra función que como resultado final nos devuelva el vector "sum_sq_dist_total" donde veremos reflejada la suma de la distancia al cuadrado entre cada punto y el centroide del cluster asociado a dicho punto para cada valor de "total_k", es decir, para cada uno de los valores de clusters que queremos estudiar, lo cuál nos permitirá determinar el número de clusters óptimo para nuestro dataset.

1.2. Implement the 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 depepende 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 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 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, tambien dentro del primero, que recorre de nuevo los valores de "k"...

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

Once the "custom_kmeans" function has been implemented, we proceed to apply it to our dataset. To do this we must make some previous modifications in our dataset, such as scaling the data so that some variables do not have more weight than others when measuring the distance and eliminate the categorical variables from the analysis since 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 have been the following for the "custom_kmeans" function:

- Call the function k-means once for 500,000 rows in dataset in R: 5.407493 seconds
- Call the function k-means ten times for 500,000 rows in dataset in R: 213.03666 seconds
- Call the function k-means once for 500,000 rows in dataset in Python: 8.744184732437134 seconds
- Call the function k-means ten times for 500,000 rows in dataset in Python: 345.5383791923523 seconds

While for the "elbow_graph" function we have obtained the following times:

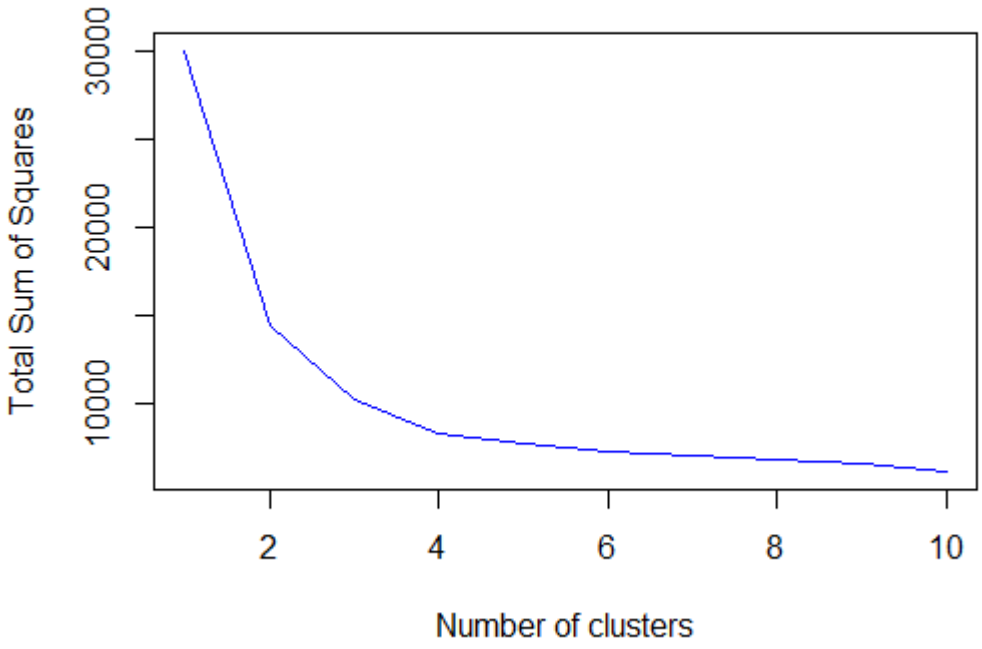
- Call the function elbow graph for 500,000 rows in dataset in R: 187.9113 seconds

- Call the function elbow graph for 500,000 rows in dataset in Python: 346.7162780761719 seconds

These results make sense since, given that for the "elbow_graph" function we have determined that the maximum number of clusters to study is 10, each time we execute "elbow_graph" with "total_k" = 10 it is being executed 10 times the "custom_kmeans" function, so the execution time of the "elbow_graph" function with "total_k" = 10 and the execution time of 10 times the "custom_kmeans" function are similar.

1.5. Plot the results of the elbow graph

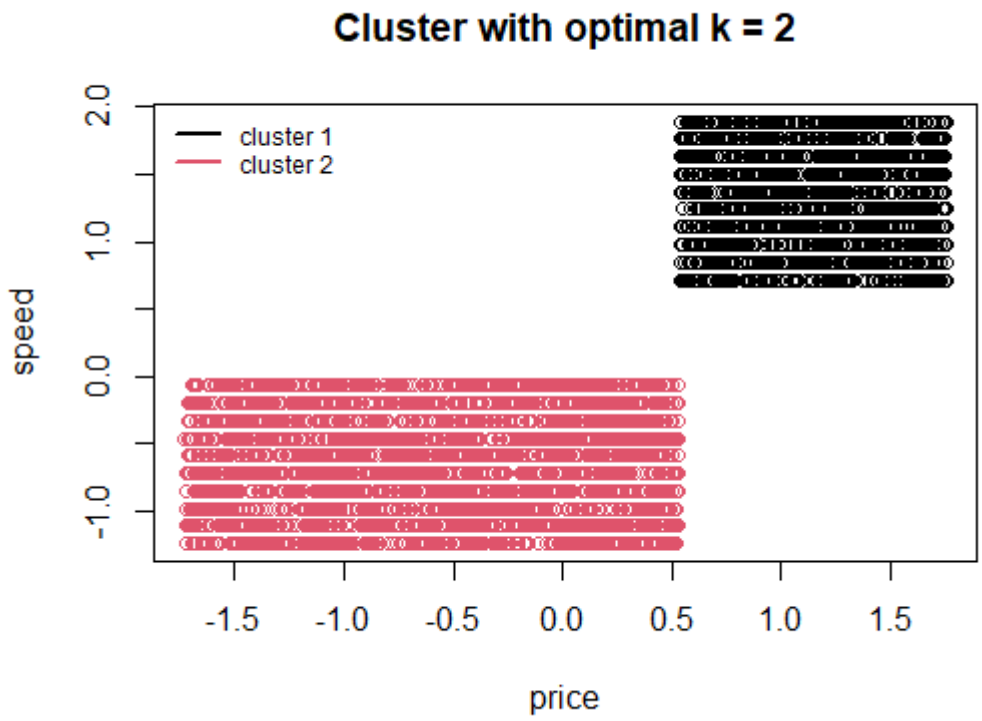
In the following image we can see the elbow graph that is produced after applying the "elbow_graph" function to our dataset:



In the graph 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.

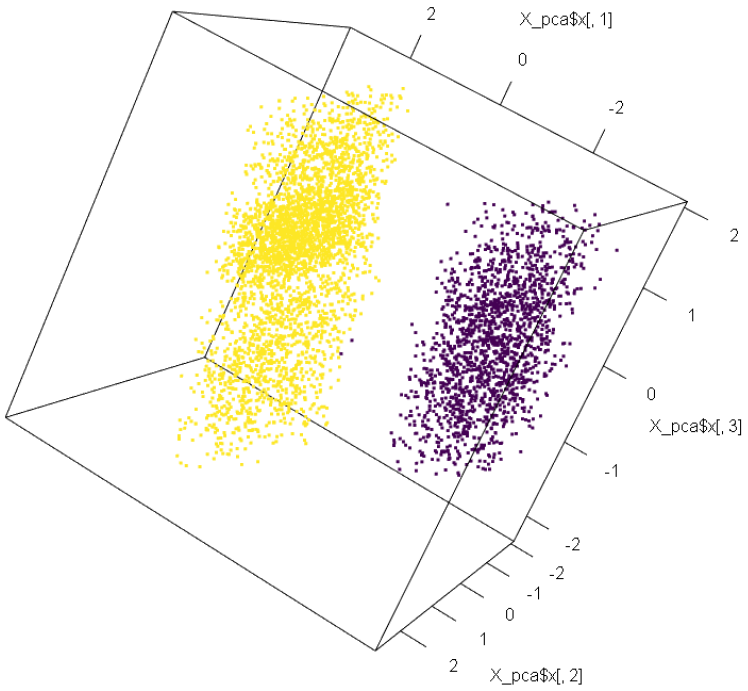
1.6. Plot the first 2 dimensions of the clusters

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 graph:



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.

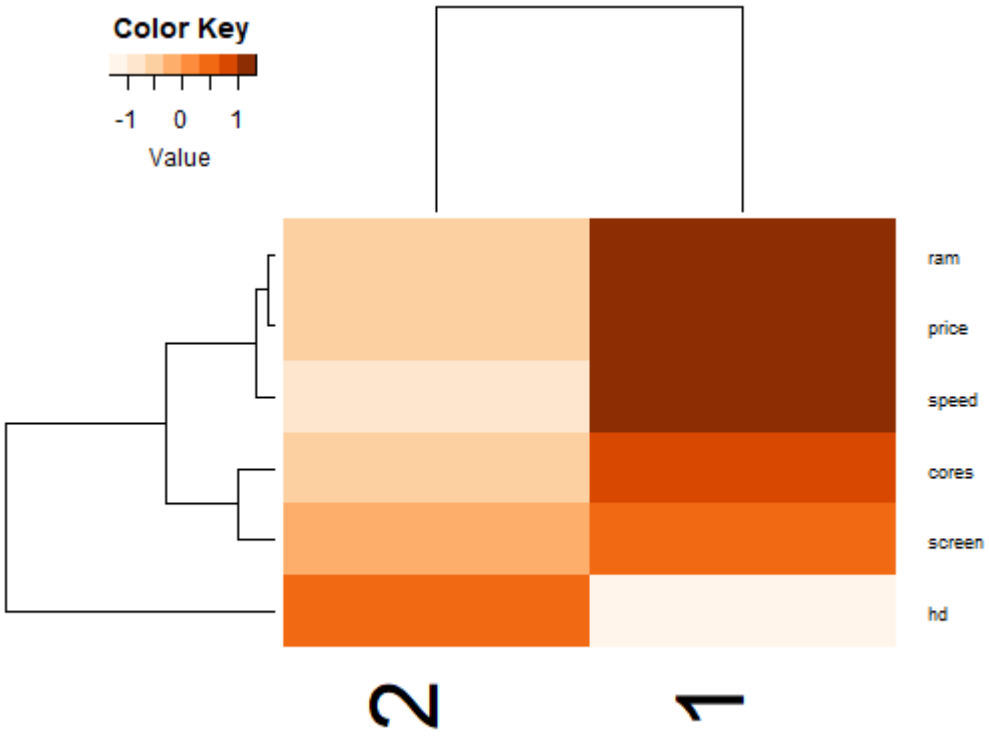


In this graph we can observe two different groups of data that again correspond almost 100% with our clusters, which again leads us to the conclusion that we have created optimal clusters already suitable for our dataset.

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.

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



2. Parallel implementation, multiprocessing

To do this in R we will use mainly two libraries: "doParallel" and "foreach".

The first one is a library that is oriented to the parallelization of problems where each thread performs its work independently, that is to say that they do not need to communicate in any way. The basic computational model that we should follow is:

1. Start M processes (threads).
2. Send the data required for each task to the threads.
3. Establish the task to be parallelized.
4. Wait for all threads to finish and get the results.
5. Close the processes (threads).

The second library, "foreach" allows to process loops in parallel and also needs the "doParallel" library to make the process in parallel using the registerDoParallel() function. One of the benefits of this library is that you can execute commands using the foreach() function and do them in parallel using the %dopar% command. In addition, this foreach() function has a ".combine" argument that is used to specify the type of output argument needed (c: vector, rbind: matrix, list:list, data.frame: output of type data.frame...).

As we know there are two ways to perform parallelism in R:

- via sockets: it creates a copy of the current R environment, in each core and performs the indicated operation, this has no major implementation difficulty but is not available for Windows.

- via forking: is more general and requires a little more work to implement, here a cluster is created on the same machine (it can also be used to create a cluster with more machines) and a new version of R is launched to each core, since it is a new environment, it must send to the threads each object it needs to perform its task, including the invocation of libraries, if necessary. This approach is available on any operating system (including Windows).

We will see that if a script takes a few seconds, parallelization is probably not worth it as we will see in our case.

2.1. Write a parallel version of you program using 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.

2.2. 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

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

4. Conclusions

- La función kmeans serial tiene buena performance. Usar paralelismo aquí parece que no mejora nada.
- Añadir procesamiento mutiproceso/threading parece que empeora las cosas. Esto puede deberse al tiempo de acceso a memoria ya que los sistemas multiprocesadores se comparte la memoria RAM entre hilos. Por ello, es altamente probable que no exista beneficio al realizar procesamiento en paralelo.
- En cambio, llamar al método elbow si mejora para cada iteracion k=i. Como es normal, el enfoque forking tiene ligeramente mejor rendimiento que vía sockets.
- Pudimos observar que el paralelismo funciona mejor con menos núcleos que el total. Obtenemos un tiempo óptimo con la mitad de núcleos.

Notes Python

Se utiliza el numPy