Universit Paris Dauphine PSL
Mastere Intelligence Artificielle et Sciences de Données (IASD)
Systems paradigms and algorithms for Big Data

Optimisation de l'algorithme K-Means

en utlisant les API de Apache Spark

Réalisé par:

Riadh Balti & Wael Chabir

Professeur:
Dario Colazzo

Tunis, Janvier 2020

Table des matières

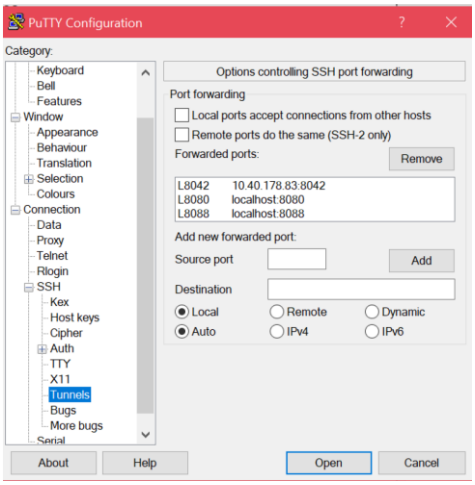
Introduction

K-Means aussi connu sous le nom K-moyennes est une méthode de partitionnement de données selon les similitudes entre les attributs des données. Cette méthode fait parti des méthodes de Machine Learning non supervisé. L'objectif de ce projet est d'optimiser l'implémentation de l'algorithme K-Means avec Spark en langage Python. La métrique d'évaluation est le temps nécessaire pour terminer la génération des clusters.

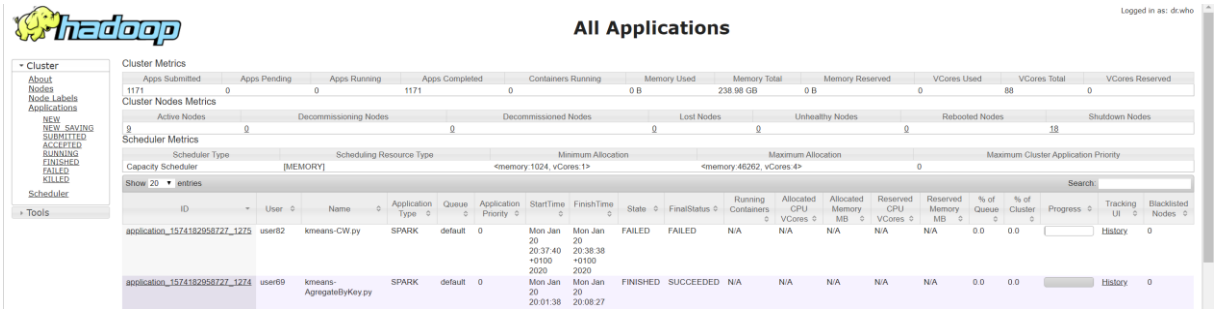
1. Mise en place de l'environnement de travail

Afin de travailler sur le cluster est d'avoir un accès à l'interface web Hadoop, on doit faire quelques configurations. Pour faire ce, on doit créer un tunnel SSH.

En utilisant PuTTY, on crée le mapping du port du port 8088 de la machine distante (cluster) vers le port 8088 de la machine locale. Ci-dessous une capture écran de la configuration faite sur PuTTY :



Une fois toute la configuration est en place, on lance la connexion SSH et on peut ainsi accéder à l'interface de Hadoop sur le lien suivant : http://localhost:8088/cluster



2. Travail effectué

2.1. Evaluation de l'algorithme initial

On est donné une implémentation de K-Means ainsi qu'un dataset qui comporte 150 enregistrements. L'implémentation initiale comporte des appels aux API Spark. Dans tout le reste du travail on va évaluer notre optimisation face à l'implémentation initiale.

2.2. Optimisation K-Means

2.2.1. Optimisation du code

La première étape de l'optimisation consiste à réduire le nombre de Shuffle. On doit donc remplacer au maximum les instructions qui induisent des Shuffles par d'autres qui sont avec moins de Shuffles ou bien sans Shuffles.

On sait déjà que les instructions Map n'induisent plus de shuffles. Donc on va plus changer ces instructions. Etant donné que tout le logique K-Means se trouve dans la fonction simpleKmeans(…), on va donc focaliser notre optimisation sur cette fonction.

Optimisation 1 :

On remplace la ligne suivante :

```
dist_list = dist.groupByKey().mapValues(list)
# (0, [(0, 0.866025403784438), (1, 3.7), (2, 0.5385164807134504)])
```

Par la suivante :

```
dist_list = dist.reduceByKey(lambda accum, v: accum + v)
# [(0, (2, 0.8062257748298554, 1, 5.047771785649585, 0, 0.14142135623730995)), ...]
```

On change grouByKey() par la fonction reduceByKey() pour la raison suivante : groupByKey peut provoquer des problèmes lorsque les données sont envoyées sur le réseau et collectées sur les serveurs de Reduce. C'est-à-dire toutes les données doivent être regrouper sur un seul nœud pour faire le groupement par Key. Dans l'autre côté, reduceByKey() les données sont combiné sur chaque nœud avant d'etre envoyer sur le réseau pour faire le Reduce. Une seule sortie pour une clé donné sur chaque partition est envoyée sur le réseau.

Optimisation 2 :

On remplace le bout de code suivant :

```
clusters = assignment.map(lambda x: (x[1][0][0], x[1][1][:-1]))
# (2, [5.1, 3.5, 1.4, 0.2])

count = clusters.map(lambda x: (x[0],1)).reduceByKey(lambda x,y: x+y)
somme = clusters.reduceByKey(sumList)
centroidesCluster = somme.join(count).map(lambda x : (x[0],moyenneList(x[1][0],x[1][1])))
```

Par le code ci-dessous :

```
clusters = assignment.map(lambda x: (x[1][0][0], (x[1][1][:-1] + [1])))
# (2, ([5.1, 3.5, 1.4, 0.2, 1]))
sommes = clusters.reduceByKey(sumList)
centroidesCluster = sommes.map(lambda x: (x[0],moyenneList(x[1][:-1],x[1][len(x[1])-1]) ))
```

Le code original contient deux appels reduceByKey() et un appel join(). Pour faire l'optimisation, on a remplacé par un nouveau code contenant un seul reduceByKey(). Ce changement réduit considérablement le temps d'exécution de K-Means.

Optimisation 3 :

La fonction cartesien() double le nombre de partitions alloués pour les données de l'ordre 2*n. La croissance du nombre de partitions provoque plus de complexité et ainsi augmente le temps

d'exécution. Pour remédier ce probleme et réduire le temps d'exécution, on utilise la fonction coalesce(). Si on passe en paramètre MAX_PARTITIONS_COUNT à coalesce(), la fonction réduit le nombre de partition s'il dépasse la limite. Dans l'autre côté, si le nombre est inférieur à la limite donné, le nombre de partition reste inchangé. On voit ainsi le code original avant le changement :

```
joined = data.cartesian(centroides)
# ((0, [5.1, 3.5, 1.4, 0.2, 'Iris-setosa']), (0, [4.4, 3.0, 1.3, 0.2]))
```

Après le changement :

```
joined = data.cartesian(centroides).coalesce(MAX_PARTITIONS_COUNT)  #No
# ((0, [5.1, 3.5, 1.4, 0.2, 'Iris-setosa']), (0, [4.4, 3.0, 1.3, 0.2]))
```

Optimisation 4 :

On a éliminer la fonction cartesien() et on l'a remplacer par un autre bout de code qui ne fait plus de Shuffle sur les données. Dans la nouvelle implémentation, on a utilisé la fonction collect() qui collecte uniquement les centroïdes des clusters. Ci-dessous on donne la nouvelle implémentation de collect() :

```
L = [i for i in range(nb_clusters)]
dist = data.flatMap(lambda x :((i, x) for i in L ))\
    .map(lambda x : (x[1][0] , (x[0], computeDistance(x[1][1][:-1] , centroid_dict[x[0]] ))))
# (0, (0, 0.866025403784438))
```

Optimisation 5 :

La fonction persist() permet de stocker les résultats qui peuvent être utilisé dans le futur calcul dans la mémoire des workers. L'utilisation de cette fonction réduit considérablement l'échange de données sur le réseau. On a utilisé la fonction persisit() après chaque calcul des centroïdes.

```
    centroides = centroidesCluster
    centroides.persist()
```

2.2.2. Optimisation des Hyperparamètres

i) Variation des k

On varie le paramètre K (nombre de clusters) tout en fixant les autres paramètres tels que :

- Num-executors
- Executor-cores
- Executor-memory
- Driver-memory
- Driver-cores

❖ Architecture du cluster :

spark-submit --master yarn --deploy-mode cluster

--executor-cores 4

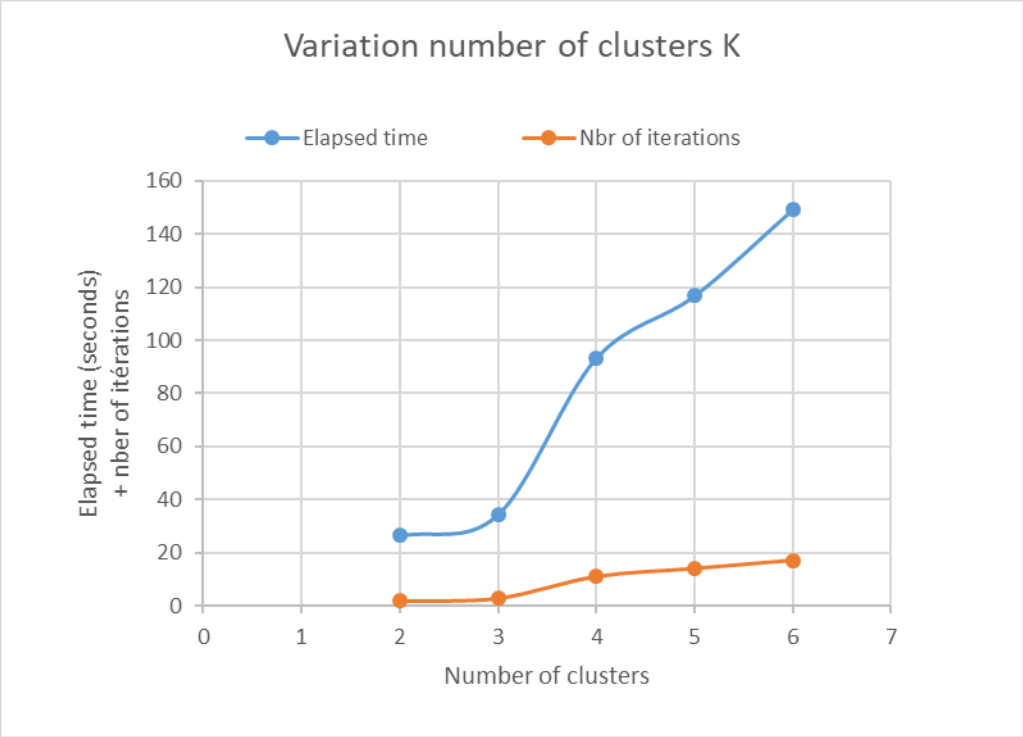
--num-executors 10

--executor-memory 8g

--conf spark.yarn.executor.memoryOverhead=4g

--conf spark.driver.memory=10g

K	num_partitions (initial)	Data_count (nbre of lines)	Elapsed time (seconds)	Number of iterations	error
2	2	150	26.4	2	0.076
3	2	150	34.3	3	0.066
4	2	150	93.1	11	0.061
5	2	150	116.7	14	0.063
6	2	150	149.1	17	0.063



Interprétation :

Il est bien clair qu'en augmentant le nombre de clusters K à prédire, l'algorithme ou plus précisément la fonction responsable « **simpleKmeans »** demande beaucoup plus de temps pour converger, ceci est dû principalement à l'augmentation du nombre d'itérations nécessaires pour arriver à cette convergence. D'autre part, on remarque que l'erreur à son tour diminue légèrement.

ii) Variation architecture cluster

Variation du nombre d'exécuteurs : Afin d'identifier la performance du cluster, nous avons voulu changer à chaque fois le nombre d'exécuteurs en utilisant cette architecture :

Architecture du cluster :

spark-submit --master yarn --deploy-mode cluster

--executor-cores 4

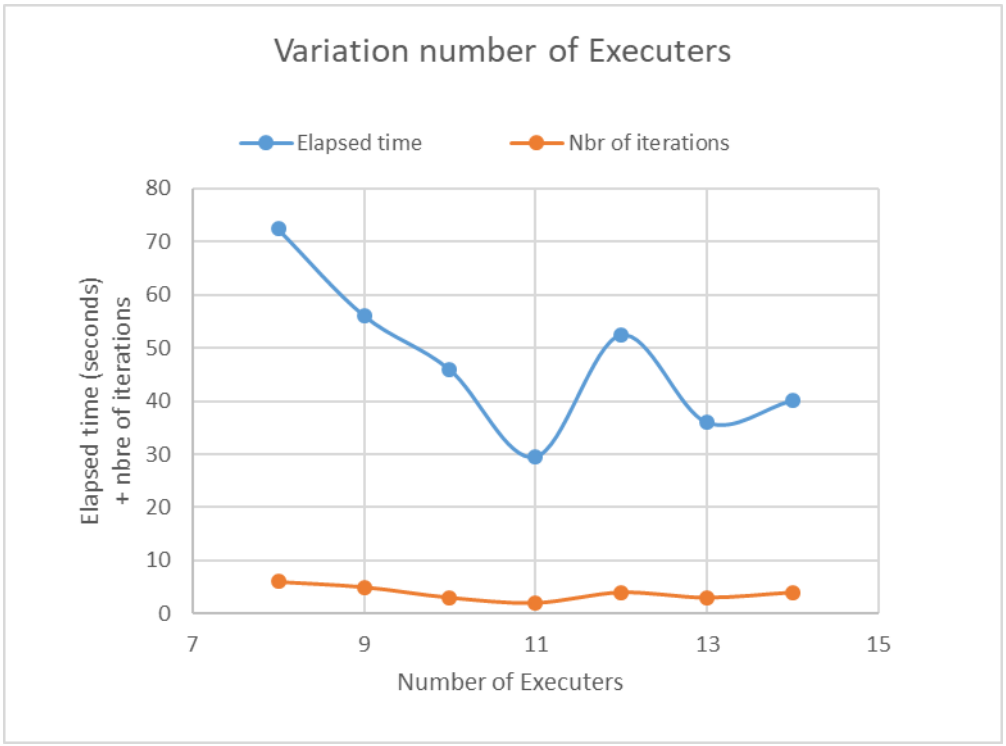
--num-executors 8**9**10**11**12*13**14

--executor-memory 8g

--conf spark.yarn.executor.memoryOverhead=4g

--conf spark.driver.memory=10g

Num_Executers	num_partitions	Data_count (nbre of lines)	Elapsed time (seconds)	Number of iterations	error
8	2	150	72.4	6	0.074
9	2	150	56.1	5	0.074
10	2	150	46.0	3	0.066
11	2	150	29.6	2	0.074
12	2	150	52.5	4	0.066
13	2	150	36.1	3	0.066
14	2	150	40.2	4	0.074



Interprétation :

D'après la figure ci-dessus, on remarque que le temps d'exécution diminue proportionnellement au nombre alloué d'exécuteurs. A un moment donné lorsqu'on continue l'augmentation de nombre d'exécuteurs, le temps reprends son augmentation. Ce phénomène nous indique qu'il existe un nombre n optimal d'exécuteurs. Le temps d'exécution augmente lorsqu'on utilise un nombre de exécuteurs supérieur à ce nombre optimal, cette observation s'avère bien clair en adoptant 11 exécuteurs.

ii) Variation de nombre initial des partitions

Dans cette partie on voudrait bien débuter avec un nombre différent de partitions initial, il s'agit de de changer le nombre de partitions de notre dataset iris en utilisant la commande suivante :

sc.textFile("hdfs:../ "iris.data.txt ",x)

x : nombre initial de partitions

❖ Architecture du cluster :

spark-submit --master yarn --deploy-mode cluster

--executor-cores 4

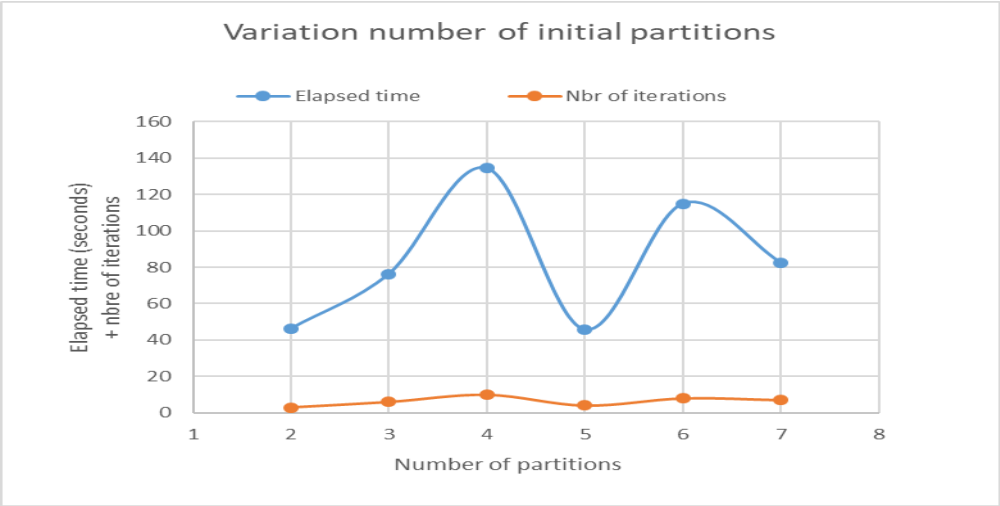
--num-executors 10

--executor-memory 8g

--conf spark.yarn.executor.memoryOverhead=4g

--conf spark.driver.memory=10g

Num_partitions	Data_count (nbre of lines)	Elapsed time (seconds)	Number of iterations	error
2	150	46.0	3	0.066
3	150	76.1	6	0.074
4	150	134.4	10	0.066
5	150	45.6	4	0.066
6	150	114.9	8	0.066
7	150	82.2	7	0.066



Interprétation :

D'après la courbe ci-dessus, on remarque que le choix du nombre de partitions initial fait varier le temps d'exécution. On remarque que pour le choix de 5 partitions fait chuter le temps d'exécution. On peut tirer de cette figure que le choix du nombre de partitions doit être adéquat à la taille de données.

De plus, le dépassement du nombre de partitions idéal induit la création de partitions vides. Ce qui résulte à avoir plus de traitement inutile ce qui augmente le temps d'exécution.

2.3. Optimisation K-Means++

Au cours de notre travail, on a implémenté l'algorithme K-Means++. Le K-Means++ consiste à choisir des centroides initiaux qui sont parmi les points du dataset et qui sont les plus écarté possible pour réduir au maximum le tax d'erreur. Ci-dessous la méthode qui rend les centroides de K-Means++ :

```
def initCentroidsKpp(data, numClusters):
    centroids = data.takeSample(False, 1)
    dataArray = data.collect()
    while len(centroids) < numClusters:
        D2 = distanceFromCentroids(dataArray, centroids)
        centroids.append(chooseNextCentroid(D2, dataArray))
    centroids = sc.parallelize(centroids)
    centroids = centroids.map(lambda (index, data): data[:-1])
    centroids = centroids.zipWithIndex()
    centroids = centroids.map(lambda (data, index): (index, data))
    return centroids
```

2.4. Générateur de données

2.4.1. Description de l'outil

Afin d'évaluer la performance des algorithmes, on doit voir la capacité de chaque implémentation à manipuler de grande taille de données. Pour ce faire, on a développé un script qui augmente le nombre

d'enregistrements du dataset. A un dataset donné qui contient par exemple 150 enregistrements, le script génère une nouvelle copie contenante $(\mathrm{n} * 150)$ enregistrements ; $n \geq 1$.

La génération de données se fait de la façon suivante : on récupère chaque point du dataset, et on ajoute du bruit aux composantes du point initial sans changer de classe.

On fait l'hypothèse que le fait d'ajouter un bruit minimal aux composantes d'un point donnée ne change pas la classe du point.

Exemple :

$$point = (x_1, x_2, x_3, x_4, CLASSE)$$

$$point_{nouveau} = (x_1 + bruit, x_2 + bruit, x_3 + bruit, x_4 + bruit, CLASSE)$$

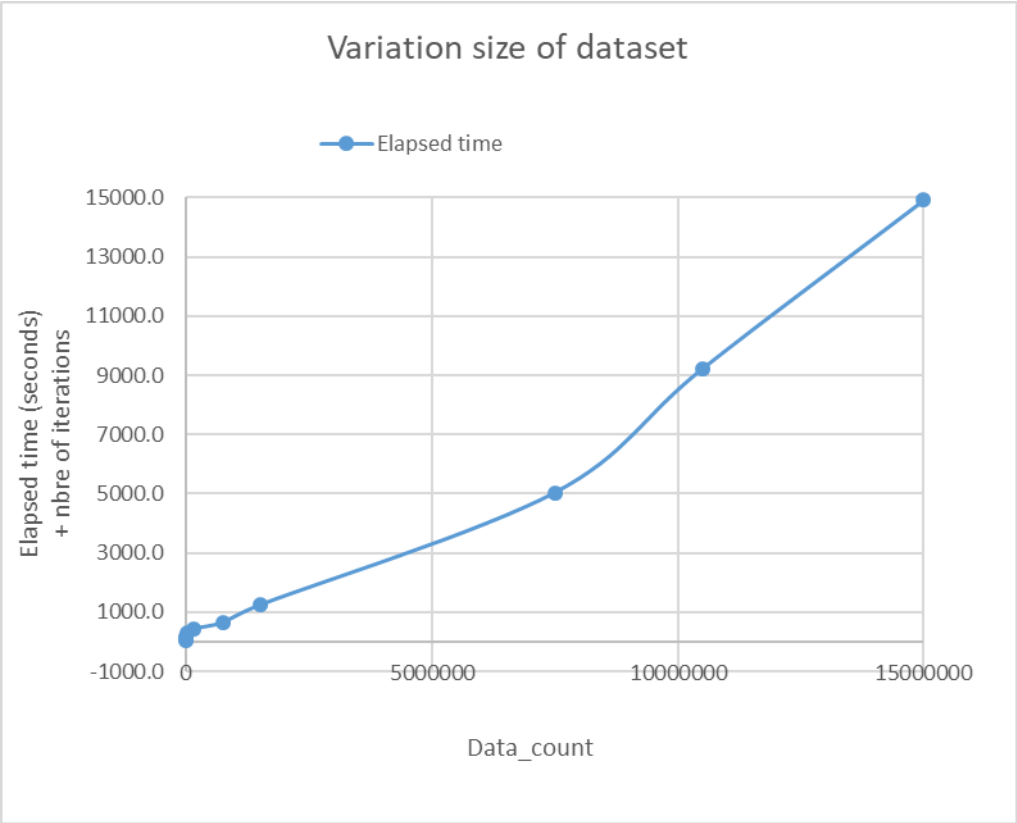
2.4.2. Exploitation de l'outil

Pour exploiter l'outil, on s'est basé sur notre dataset inital **"iris.data.txt "** qui contient 150 lignes, on a multiplié cette taille par un une liste contenant à chaque fois un nombre **n** différent pour générer des nouveaux datasets tels que leurs tailles égales à **150*n**:

List=[2, 10 ,50, 10 ,1000, 5000, 10 000, 50 000, 70 000, 100 000]

Le tableau ci-dessous illustre la variation de la taille des datasets (Data_count (number of lines))

Num_partitions	Data_count (nbre of lines)	Elapsed time (seconds)	Number of iterations	error
2	150	46.02	3	0.066
2	150*2=300	83.26	4	0.052
2	150*10=1500	125.93	16	0.022
2	150*50=7500	159.50	14	0.009
2	150*100=15000	320.36	20	0.007
2	150*1000=150000	430.23	22	0.002
2	150*5000=750000	639.62	30	0.001
2	150*10000=1500000	1230.25	43	0.001
2	750*10000=7500000	5035.2	53	0.001
2	105*10000=10500000	9222.4	62	0.001
2	150*100000=15000000	14922.22	70	0.001



Interprétation :

D'après la figure ci-dessus, on peut dire qu'il est évident que plus la taille du dataset augment plus le temps d'exécution de notre algorithme est grand, même chose pour le nombre d'itérations.

3. Conclusion

Dans l'objectif d'optimiser l'algorithme, on a travaillé sur deux volets :

- Optimisation de code
- Variation des hyperparamètres

L'optimisation de code porte sur l'élimination des instructions qui provoquent des Shuffles. Ainsi que la réduction de l'échange des données sur le réseau en persistant quelques résultats de calcul dans la mémoire des workers.

En deuxième temps, lorsqu'on a varié les hyperparamètres on a remarqué qu'un bon choix influe le temps d'exécution de l'algorithme.

En conclusion, on a réussi à optimiser notre algorithme en termes de réduction de temps d'exécution et d'exploitation efficace de la capacité du cluster qui nous favorise une très bonne scalabilité de calcul parallèle.

Code source

https://github.com/waelchabir/kmeans_optimisation/tree/master/code