MASTÈRE INTELLIGENCE ARTIFICIELLE ET SCIENCES DE DONNÉES (IASD)

Systems paradigms and algorithms for Big Data

Réalisé par:

Amenallah Salem,
Mohamed Amine Gaidi
Aziz ben ammar

Professeur:
Mr Dario Colazzo

Université Paris-Dauphine | Tunis

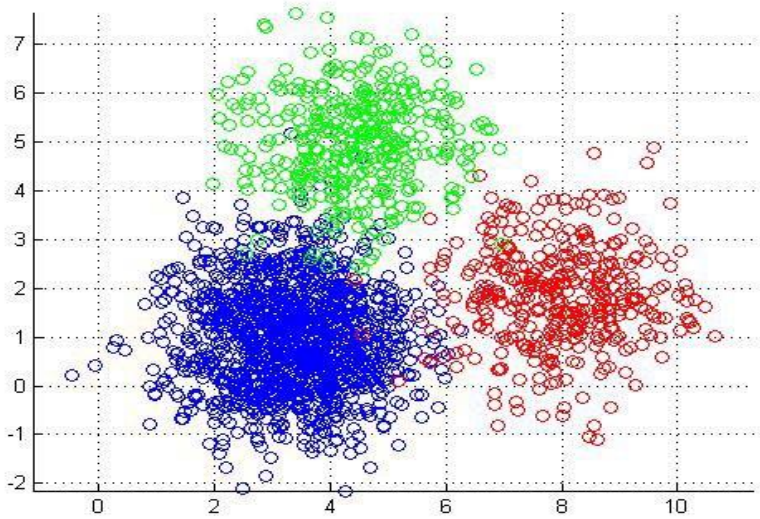


Table des matières

1 Introduction

Le problème du partitionnement d'un ensemble de données (data clustering) non labelisé en groupes (ou clusters) apparait dans une grande variété d'applications (voir le lien suivant pour quelques applications des algorithmes de clustering (voir par exemple https://datafloq.com/read/7-innovative-uses-of-clustering-algorithms/6224).

L'un des algorithmes de clustering les plus connus et les plus utilisés est l'algorithme k-means [1]. La popularité de k-means est due principalement à sa simplicité (le seul paramètre qui doit être choisi est (k= le nombre des clusters souhaité), et aussi sa vitesse.

Dans ce contexte plusieurs efforts et articles scientifiques ont été réalisés pour améliorer la qualité des résultats produit par k-means. Une telle tentative est l'algorithme k-means++, qui utilise la même méthode itérative mais une initialisation différente [2]. Cet algorithme a tendance à produire de meilleurs clusters dans la pratique, bien qu'il s'exécute parfois plus lentement en raison de l'étape d'initialisation supplémentaire. Compte tenu de l'omniprésence du clustering k-means et de ses variantes, il est naturel de se demander comment cet algorithme pourrait être adapté à un environnement distribué.

Dans ce projet, notre but est d'optimiser de l'algorithme **kmeans-dario-x.py** en termes de temps d'exécution. De plus, nous effectuant quelques changements sur l'algorithme pour implémenter kmeans++. Enfin, nous décrivons un autre algorithme k-means++ MapReduce, qui peut également être efficace et terme de vitesse et simplicité sur un ensembles de données très grande tout en implémentant ces algorithmes distribués dans Spark.

2 Premier pas avec kmeans-dario-x.py & iris.data.txt

La première étape consiste à télécharger la Dataset Iris.txt
wget https ://www.dropbox.com/s/9kits2euwawcsj0/iris.data.txt
et la charger sur notre répertoire HDFS et vérifier qu'elle est bien enregistrer
hdfs dfs -put iris.data.txt /user/user81.
hdfs dfs -ls /user/user81.

Etapes de l'algorithme kmeans-dario-x.py
L'algorithme se décompose en quatre étapes principales : Après choisir k=3 points comme centres de départ
==========
k-means
==========
(i) On chosit un vecteur
(ii) On trouve tous les points les plus proches de chaque centre (on affecte chaque point à son centroïde le plus proche) i.e on minimise J par rapport à z c'est-à-dire
$z_i^k = 1 si ||x_i - \mu_k||^2 = \min_s ||x_i - \mu_s||^2$
(iii) Trouvez le nouveau centre de chaque cluster (on recalcule les coordonnées des centroides de chaque cluster (en calculant la moyenne des coordonnées des points qui lui sont rattachés). i.e on minimise J par rapport à
$\mu_k = \frac{\sum_i s_i^k x_i}{\sum_i z_i^k}$
(iv) on refait l'étapes (ii) et (iii) en calculant les coordonnées des centroïdes de chaque cluster (en calculant la moyenne des coordonnées des points qui lui sont rattachés), jusqu'à ce que la distance totale entre les points d'une itération et le suivant soit inférieure à la distance de convergence spécifiée (i.e l'algorithme converge vers un optimum local).

Le temp d'exécution du code principale est a peut prés égale à 18 minutes

```
(PythonRDD[162] at RDD at PythonRDD.scala:48, 0.06576926814733124, 4)
Temps d execution de l'algorithme : 1030.6789103  secondes ---
user81@vmhadoopmaster:~$
```

3 Etapes d'optimisation du code

Notre code source : kmeans-dario-xOptimizez.py

L'idée de l'optimisation est la suivante. Pour le calcul des nouveaux centroïdes a chaque fois le code de départ a fait appel à trois phases de **shuffles** : dans ce qui suit on va voir ces phases et comment remplacer les lignes qui induisent des shuffles par d'autres qui sont sans shuffle

3.1 Optimisation n°1 :

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. i.e. toutes les données seront regroupées sur un seul nœud pour effectuer le groupe by Key. Par contre, reduceByKey() combine les données sur chaque nœud du cluster avant d'être envoyer sur le réseau pour faire le Reduce. Ainsi une seule sortie pour une clé donnée sur chaque partition sera envoyée sur le réseau.

On remplace la ligne suivante :

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

Par

```
dist_list = dist.reduceByKey(lambda acc, w :acc+v)
```

3.2 Optimisation n°2 :

Le code original contient deux méthodes reduceByKey() et un join(). Cette phase de l'optimisation consiste, on a reformuler ces ligne de tel sorte on aura un seul reduceByKey(). On remplace le bout de code suivant :

```
75
76  clusters = assignment.map(lambda x: (x[1][0][0], x[1][1][:-1]))
77  # (2, [5.1, 3.5, 1.4, 0.2])
78
79  count = clusters.map(lambda x: (x[0],1)).reduceByKey(lambda x,y: x+y)
80  somme = clusters.reduceByKey(sumList)
81  centroidesCluster = somme.join(count).map(lambda x : (x[0],moyenneList(x[1][0],x[1][1])))
82
83  ############################
84  # Is the clustering over ? #
85  ############################
```

Par le code ci-dessous :

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

somme = clusters.reduceByKey(sumList)
centroidesCluster = somme.map(lambda x : (x[0],moyenneList(x[1][:-1],x[1][len(x[1])-1])))
```

3.3 Optimisation n°3 :

La fonction qui calcule le produit cartésien : cartesien() vas surement doubler le nombre de partitions sur les données. Cette augmentation du nombre de partitions provoque plus de complexité et ainsi augmente le temps d'exécution.

on utilise la fonction coalesce(). Si on passe en paramètre limite 20 par exemple à coalesce(), la fonction réduit le nombre de partition s'il dépasse la limite. D'autre part, si le nombre est inférieur à la limite donné, le nombre de partition reste tell qu'il est.
Donc on remplace la ligne suivante

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

Par

```
joined = data.cartesian(centroides).coalesce(20)
# ((0, [5.1, 3.5, 1.4, 0.2, 'Iris-setosa']), (0, [4.4, 3.0, 1.3, 0.2]))
```

Remarque 1
On peut aussi avant le calcul du produit cartésien et pour accélérer le calcule on enlève la colonne des classes des feurres. Par cette méthode en enlève une colonne de la boucle. Puis on l'a récupéré après. Ainsi cette fonction s'applique seulement uniquement les centroïdes des clusters.

3.4 Optimisation n°4 :

On ajoute la ligne suivante pour appliquer la méthode persist() au centroides ce qui vas réduire l'échange de données sur le réseau du cluster. On a utilisé la fonction persisit() après chaque calcul des centroïdes.

```
96          else:
97              centroides = centroidesCluster
98              centroides.persist()
99              prev_assignment = min_dist
100             number_of_steps += 1
```

Le temps d'exécution est

```
(PythonRDD[67] at RDD at PythonRDD.scala:48, 0.11381371209237857, 1)
Temps d execution de l'algorithme : 7.71601200104 secondes ---
user81@vmhadoopmaster:~$
```

4 K-Means++

Code source : kmeansPlusPlus.py

Un inconvénient de l'algorithme K-means est qu'il est responsable de l'initialisation des centroïdes. Donc ces derniers, sont initialisés pour être un centre du cluster. Ils peuvent être par hasard des ponts du même cluster par exemple

[(0, [5.5, 2.3, 4.0, 1.3, 'Iris- setosa']),
(1, [6.1, 3.0, 4.6, 1.4, 'Iris- setosa']),
(2, [5.9, 3.0, 5.1, 1.8, 'Iris- setosa'])]

Ce qui vas entraîner plus de temps pour corriger cette mauvaise initialisation.
Pour surmonter cet inconvénient, nous utilisons K-means ++. Cet algorithme améliorer la qualité du clustering.
A part l'initialisation, le reste de l'algorithme est le même que l'algorithme K-means standard.

==========\\
k-means++ \\
==========

(i) Choisissez la première moyenne μ au hasard dans l'ensemble Xet ajoutez-la à l'ensemble M= {μ1, …, μk}.
(ii) Pour chaque point x dans X, calculez la distance au carré D (x) entre x et la moyenne la plus proche en M.
(iii) Choisissez la prochaine moyenne au hasard dans l'ensemble X, où la probabilité qu'un point x dans X soit choisi est proportionnelle à D (x), et ajoutez à M.
(iv) Répétez les étapes (ii) et (iii) k – 1 fois pour produire k moyennes initiales.
(v) Appliquer l'algorithme k-means standard fournis dans la section 1, initialisé avec ces moyens

Remarque 2 :
En suivant la procédure ci-dessus pour l'initialisation, nous évitant le choix centroïdes qui sont proches les uns des autres. Donc le choix des centroïdes initiaux avec l'algorithme K means ++ est plus pertinent que le choix de l'algorithme standard

Donc on ajoute la fonction initCentroidesKpp pour l'initialisation des centroïdes

```
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
```

Après initialisation des centroïdes, on appelle un Broadcast() pour partager les centroïdes dans chaque machine. Tout comme l'étape d'initialisation.

4.1 Optimisation n°5 :

En met en cache les RDD

```
lines = sc.textFile("hdfs:/user/user81/iris.data.txt")
lines.cache
```

* On ne sera pas besoin calculer le résultat à chaque itération donc une accélération du temps d'exécution

Conclusion :
Le temps d'exécution est une seconde et quelques millisecondes

```
Temps d execution de l'algorithme : 1.71601200104 secondes ---
user81@vmhadoopmaster:~$
```

5 K-Means++ avec MapReduce

Code : (premier essai) mapperkm.py & reducerkm.py

Nous considérons maintenant comment reformuler ces derniers algorithmes pour résoudre le problème K-Means++ afin qu'ils puissent être appliqués dans un cadre distribué. Plus précisément, nous formulerons une version distribuée de cet algorithme à l'aide du paradigme MapReduce. Nous pensons que cet algorithme peut avoir un grand potentiel et une grande efficacité sur des datasses à grande échelle.

Rappelons que chaque itération dans l'initialisation de k-means++ a deux phases,

- la première calcule la distance au carré D (x) entre chaque point x et la moyenne la plus proche de x,
- la seconde échantillonne un membre de X avec probabilité proportionnelle à D (x). Ces deux phases correspondent aux phases Map et Reduce de notre algorithme MapReduce pour k-means++.

Notre approche

La phase Map opère sur chaque point x du datset. Pour un x donné, nous calculons la distance au carré entre x et chaque moyenne en M et trouver la distance au carré minimale D (x). Nous émettons ensuite une seule valeur (x, D(x)), sans clé. Notre fonction est donc comme suit

```
kmeansppMap(x):
    emit (x, min_{μ∈M} ||x − μ||_2^2)
```

La phase Reduce regroupe toutes les couples emit de la phase Map, car ces émissions n'ont pas de clé. Nous réduisons le premier élément de deux paires de valeurs, en choisissant l'un de ces éléments avec une probabilité proportionnelle au second élément dans chaque paire, et réduire le deuxième élément des paires par sommation. Notre fonction est donc

```
kmeansppReduce([(x, p), (y, q)]):
    avec une probabilité p = (p + q):
        return (x, p + q)
    else:
        return (y, p + q)
```

Le MapReduce caractérisé par ces deux fonctions produit une seule valeur de la forme (x, 1) où x est un membre de l'ensemble X, et la probabilité qu'un élément particulier x dans X soit renvoyé est proportionnelle à la distance D (x) entre x et la moyenne la plus proche de M. Cette valeur x est ensuite ajoutée à M comme moyenne initiale suivante. Comme précédemment, il faut diffuser le nouvel ensemble de moyens M à l'ensemble du cluster entre chaque itération. Nous pouvons écrire l'algorithme entier (désormais appelé k-means++) sous la forme suivante:

==========
k-means++ MapReduce
==========
(i) Initialisez M, comme précédemment avec µ1 est choisi uniformément au hasard parmi X.
(ii) Appliques MapReduce kmeansppMap et kmeansppReduce à X.
(iii) Ajoutez le point x résultant à M.
(iv) Broadcast le nouvel ensemble M sur chaque machine du cluster.
(v) Répétez les étapes (ii) à (iv) au total k-1 fois pour produire k moyennes initiales.
(vi)Appliquez l'algorithme MapReduce k-means standard.

Le temps d'exécution est dans ce cas :

```
(PythonRDD[67] at RDD at PythonRDD.scala:48, 0.11381371209237857, 1)
Temps d execution de l'algorithme : 1.04700300104 secondes ---
user81@vmhadoopmaster:~$
```

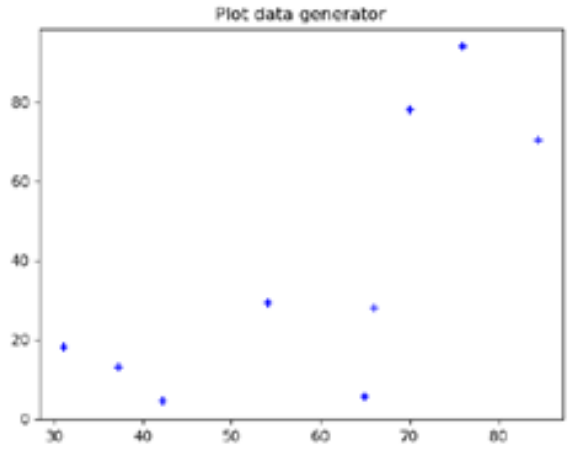
6 Teste de performance sur une grande Dataset.

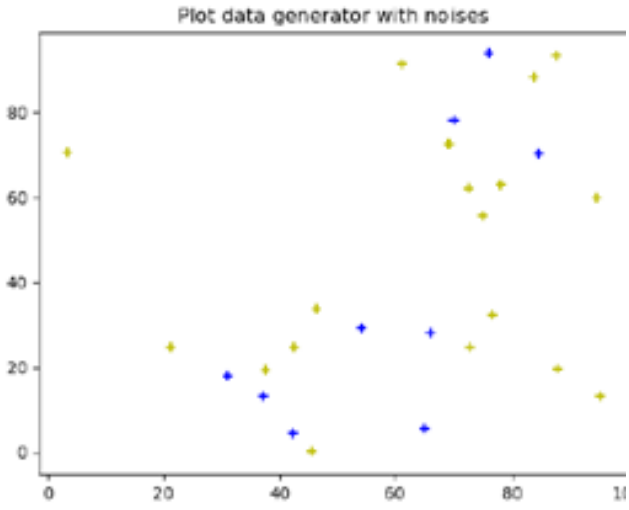
Générateur de données

Afin d'évaluer nos algorithmes KMeans et kmeans++ implémenté, un générateur de données est mis en place. On a mis en place 2 fichiers de python : generator.py et generator-noise.py.
Le fichier generator.py permet de générer des points et de les sauvegarder dans
Un fichier. Le deuxième fichier, generator-bruit.py, permet de générer les points ainsi que
Des points bruits qui sont générés aléatoirement.
Afin de sauvegarder les données dans un fichier externe,
*saveAsTextFile() : qui est une méthode déjà présente qui sauvegarde un rdd

D'une part :
En exécutant, le generator.py avec la commande suivante :
$ spark-submit générator. py out 9 3 2 10
Les points sont générés, et en affichant les résultats dans le graphique on obtient :

D'autre part
En executant, le generator-bruit.py avec la commande suivante :
$ spark-submit générator_noise.py out 9 3 2 10
Le nombre de bruits est le double du nombre des points à générer. Les points sont générés, et en affichant les résultats dans le graphique on obtient :





6.1 Tableau de performance

Algorithme	Temp d'exécution Iris.txt	Temp d'exécution grandData.txt
kmeans-dario-x	1030,678 sec	1817.16 sec
kmeans-dario-xOptimizez	7,410 sec	20.891 sec
kmeansPlusPlus	1.6760 sec	3.87 sec
kmeansMapReduce	1.04700 sec	2.001 sec

6.2 Interprétation & conclusion

Pour ce projet, nous avons examiné le code fournis kmeans-dario-x.py. On a vu comment cet algorithme de clustering pourraient être implémenté et optimisé. Tout d'abord, nous avons modifié cet algorithme pour produire une version plus rapide et optimisée 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.

On a vu aussi que la sélection initiale des centroïdes a un impact significatif sur le temps d'exécution des k-means, pour cela on a donner et analyser une version du k-means qui est k-means ++ pour accélérer l'algorithme. Aussi on a vu que cette dernière assertion n'est vrai que pour les petits dataset avec un petit nombre de clusters, car l'initialisation de K-means ++ prend O (n k) pour s'exécuter. C'est assez rapide pour les petits k et les grands n, mais si on choisit k trop grand, cela prendra un certain temps.

Enfin, on a vu comment on peut construire une extension très intéressante de cet algorithme dans un environnement distribué à l'aide du paradigme de MapReduce.

Après avoir utilisé toutes les optimisations mentionnées ci-dessus, nous avons réussi à réduire le temps de l'algorithme K-means de 1030.76 secondes jusqu'a 1.04700 secondes.

7 Références

[1] M. Lichman. UCI Machine Learning Repository. University of California, Irvine, School of Information and Computer Sciences, http://archive.ics.uci.edu/ml, 2013 .
[2] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of care-ful seeding. In Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms, pages 1027{1035. Society for Industrial and Applied Mathematics, 2007.
[3] https://github.com/ArsMing276/Kmeans_Implementation_with_Mapreduce