Projet Big Data : k-means Clustering

20 janvier 2020

Salem Amenallah, [1]

Université Paris-Dauphine | Tunis

Table des matières

1. amenallah.salem@dauphine.tn / id cluster de Dauphine : **user81**

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 (à voir le lien suivant pour quelques applications des algorithmes de clustering 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 de *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 (plusieurs extensions de ce algorithmes ont été le but des travaux de recherche).

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, nous montrons comment implémenter k-means et k-means++ en utilisant le framework MapReduce en utilisant Hadoop & Spark. De plus, nous décrivons un autre algorithme MapReduction, k-means++, qui peuvent également être efficaces et terme de vitesse et simplicité sur un ensembles de données (Dataset) trés grande tout en implémentant ces algorithmes distribués dans Spark, les testons sur la dataset `Iris` et un autre grands ensembles de données du monde réel en rapportons et discutant les résultats.

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

La première étape consiste à téléchargez la Dataset Iris
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 5000-8.txt /user/user81.
hdfs dfs -ls /user/user81.
Ensuite on télecharge notre fichier Python à optimiser
wget https ://www.dropbox.com/s/tm9lk6vffci5yzr/kmeans-dario-x.py.
Finalement on lancer notre programme Python (kmeans-dario-x.py) à partir du master node du cluster. en utilisan spark-submit kmeans-dario-x.py On ajoute :
import time
startTime = time.time()
puis dans la main function on ajoute
print("Le temp d'exécution est %s seconds —" % (time.time() - startTime)) pour la vusualisation du temp d'éxécution du code.

2.1 Notations :

Dans ce qui suit, on utilisera les notations suivantes :

1. Soit les $x_i \in \mathbb{R}^d$, $i \in \{1, \ldots, n\}$ les observations que nous voulons partitionner.

2. $\mu_k \in \mathbb{R}^d, k \in \{1, \ldots, K\}$ sont des moyennes où μ_k est le centre du cluster k. Nous allons noté μ la matrice associée.
3. z_i^k sont des variables indicatrices associées à x_i telles que $z_i^k =$ si x_i appartien au cluster k, et $z_i =$ sinon
4. z est la matrice don't les composantes sont égaux à z_i^k

Finalement en définie la déformation

$$J(\mu, z) = \sum_{i=1}^{n} \sum_{k=1}^{K} z_i^k ||x_i - \mu_k||^2$$

2.2 Etapes de l'algorithme

L'algorithme se décompose en quatres étapes principales : Aprés choisir K points géolocalisés aléatoires comme centres de départ

==========
k-means
==========

(i) On chosit un vecteur μ

(ii) On trouve tous les points les plus proches de chaque centre (on affecte chaque point à son centroide le plus proche) i.e on minimuse J par rapport à z : $z_i^k = 1 si ||x_i - \mu_k||^2 = \min_s ||x_i - \mu_s||^2$

(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'etapes (ii) et (iii) en calculant les coordonnées des centroides 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).

2.3 Analyse & remarques

On commence cette partie par une remarque trés importante qu'il Il y a un nombre fini de partitions possibles à k clusters (classes ici dans le cas de la dataset Iris). De plus, chaque étape de l'algorithme fait strictement diminuer la fonction de coût, positive, et fait découvrir une meilleure partition. Cela permet d'affirmer que l'algorithme converge toujours en temps fini.

Le critère de convergence pour l'étape (iv) est généralement lorsque l'erreur totale cesse de changer entre les étapes, dans ce cas un minimum local de la fonction du cout est atteint. À k fixé, la complexité lisse est polynomiale pour certaines configurations, dont des points dans un espace euclidien. Si k fait partie de l'entrée, la complexité lisse est encore polynomiale pour le cas euclidien. Ces résultats expliquent en partie l'efficacité de l'algorithme en pratique.

Remark 1. *Dans d'autres cas, certaines implémentations mettent fin à la recherche lorsque le changement d'erreur entre les itérations tombe en dessous d'un certain seuil. C'est pour cela on trouve dans des livres et des réferences que le partitionnement final n'est pas toujours optimal.*

et le temps de calcul peut-être exponentiel en fonction du nombre de vecteurs à classer, même dans le plan. Généralement (Dans la pratique) et aussi dans notre cas, on a imposer une limite sur le nombre d'itérations ou un critère sur l'amélioration entre itérations.

Remark 2. *En principe, le nombre d'itérations nécessaires pour que l'algorithme converge complètement peut être très grand, mais sur des ensembles de données réels, l'algorithme converge généralement en au plus quelques dizaines d'itérations.*

```
(PythonRDD[162] at RDD at PythonRDD.scala:48, 0.065769268147331 24, 4)
Temps d execution de l'algorithme : 196.366789103 secondes ---
user81@vmhadoopmaster:~$
```

FIGURE 1 – Le temps d'éxécution de kmeans-dario-x.py

3 Notre approche pour l'Optimisation : k-means ++

Code : kmeansProject.py
L'algorithme est le suivant :

3.1 Etapes de l'algorithme k-means ++

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

(*i*) Choisissez la première moyenne μ_1 **au hasard** dans l'ensemble $X = \{x_1, \ldots, x_k\}$ et ajoutez-la à l'ensemble $M = \{\mu_1, \ldots, \mu_k\}$.
(*ii*) Pour chaque point $x \in X$, 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 $x \in X$ 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

3.2 Points faible de l'algorithme k-means-dario-x.py

• Tout d'abord aprés le calcule du produit cartésien et pour accélérer le calcule on enlève la colone des classes des fleures par exemple 'Iris-setosa'. Par cette méthode en enlève une colonne de la boucle. Puis on la récupère aprés `assignment = min_dist.join(data)`
• Dans l'étape (ii) de k-means-dario-x.py au lieu d'affecte chaque point à son centroide le plus proche(opération faite pour chaque point) on calcule juste la distance $D(x)$ entre x et la moyenne la plus proche dans M et choisir la prochaine moyenne au (hasard), qui plus rapide en tèrme de complexité

Analyse & remarques

Cet algorithme est conçu pour choisir un ensemble de moyens initiaux bien séparés les uns des autres. Dans l'article [2] où ils décrivent d'abord l'algorithme k- means ++, les auteurs Arthur et Vassilvitskii le testent sur quelques ensembles de données du monde réel et démontrent qu'il conduit à des améliorations de l'erreur finale sur l'algorithme k-means standard. Étant donné que chaque itération de cette initialisation prend du temps $O(|M|nd)$ et que la taille de M augmente de 1 à chaque itération jusqu'à ce qu'elle atteigne k, la complexité totale de k-means ++ est $O(k^2nd)$, plus $O(nkd)$ par itération une fois la méthode k-means commence.

4 Temps d'exécution & output

```
(PythonRDD[64] at RDD at PythonRDD.scala:48, 0.11381371209237857, 1)
Temps d execution de l'algorithme : 14.529309988 secondes ---
user81@vmhadoopmaster:~$
```

FIGURE 2 – Le temps d'éxécution

Et on a le meme output que kmeans-dario-x.py

```
user81@vmhadoopmaster: ~
-rw-r--r--   3 user81 cluster      10841 2020-01-21 02:09 /user/user81/output
rt-00000
user81@vmhadoopmaster:~$ hdfs dfs -ls /user/user81/output/part-00000
-rw-r--r--   3 user81 cluster      10841 2020-01-21 02:09 /user/user81/output
rt-00000
user81@vmhadoopmaster:~$ hdfs dfs -ls /user/user81/output/part-00000/*
ls: `/user/user81/output/part-00000/*': No such file or directory
user81@vmhadoopmaster:~$ hdfs dfs -cat /user/user81/output/part-00000/*
cat: `/user/user81/output/part-00000/*': No such file or directory
user81@vmhadoopmaster:~$ hdfs dfs -cat /user/user81/output/part-00000/
(0, ((0, 2.7040902351807707), [5.1, 3.5, 1.4, 0.2, u'Iris-setosa']))
(130, ((0, 2.9088549866456628), [7.4, 2.8, 6.1, 1.9, u'Iris-virginica']))
(5, ((0, 2.4058755855890244), [5.4, 3.9, 1.7, 0.4, u'Iris-setosa']))
(135, ((0, 3.18510659162295 66), [7.7, 3.0, 6.1, 2.3, u'Iris-virginica']))
(10, ((0, 2.590901516203706), [5.4, 3.7, 1.5, 0.2, u'Iris-setosa']))
(140, ((0, 2.360022033795448), [6.7, 3.1, 5.6, 2.4, u'Iris-virginica']))
(15, ((0, 2.751672945682317), [5.7, 4.4, 1.5, 0.4, u'Iris-setosa']))
(145, ((0, 2.0067811705979977), [6.7, 3.0, 5.2, 2.3, u'Iris-virginica']))
(20, ((0, 2.3562054239815335), [5.4, 3.4, 1.7, 0.2, u'Iris-setosa']))
(25, ((0, 2.5241442114110666), [5.0, 3.0, 1.6, 0.2, u'Iris-setosa']))
(30, ((0, 2.597659972616379), [4.8, 3.1, 1.6, 0.2, u'Iris-setosa']))
(35, ((0, 2.8769145509266223), [5.0, 3.2, 1.2, 0.2, u'Iris-setosa']))
(40, ((0, 2.786174916739195), [5.0, 3.5, 1.3, 0.3, u'Iris-setosa']))
(45, ((0, 2.7317340036443265), [4.8, 3.0, 1.4, 0.3, u'Iris-setosa']))
(50, ((0, 1.5118985856641767), [7.0, 3.2, 4.7, 1.4, u'Iris-versicolor']))
(55, ((0, 0.8030591509969863), [5.7, 2.8, 4.5, 1.3, u'Iris-versicolor']))
(60, ((0, 1.3887058723862291), [5.0, 2.0, 3.5, 1.0, u'Iris-versicolor']))
(65, ((0, 1.0898795040431475), [6.7, 3.1, 4.4, 1.4, u'Iris-versicolor']))
(70, ((0, 1.2126433935827965), [5.9, 3.2, 4.8, 1.8, u'Iris-versicolor']))
(75, ((0, 1.0135600623544727), [6.6, 3.0, 4.4, 1.4, u'Iris-versicolor']))
(80, ((0, 0.7463493373302699), [5.5, 2.4, 3.8, 1.1, u'Iris-versicolor']))
(85, ((0, 0.9246101881333566), [6.0, 3.4, 4.5, 1.6, u'Iris-versicolor']))
(90, ((0, 0.8574986880456438), [5.5, 2.6, 4.4, 1.2, u'Iris-versicolor']))
(95, ((0, 0.4671587881366822), [5.7, 3.0, 4.2, 1.2, u'Iris-versicolor']))
(100, ((0, 2.6431239093163987), [6.3, 3.3, 6.0, 2.5, u'Iris-virginica']))
(105, ((0, 3.460400362193177), [7.6, 3.0, 6.6, 2.1, u'Iris-virginica']))
(110, ((0, 1.7011282530524663), [6.5, 3.2, 5.1, 2.0, u'Iris-virginica']))
(115, ((0, 1.9798579073593479), [6.4, 3.2, 5.3, 2.3, u'Iris-virginica']))
(120, ((0, 2.473776599991736), [6.9, 3.2, 5.7, 2.3, u'Iris-virginica']))
(125, ((0, 2.692032193963018), [7.2, 3.2, 6.0, 1.8, u'Iris-virginica']))
(1, ((0, 2.730098410924655), [4.9, 3.0, 1.4, 0.2, u'Iris-setosa']))
(131, ((0, 3.52209748114198), [7.9, 3.8, 6.4, 2.0, u'Iris-virginica']))
(6, ((0, 2.8348728366542293), [4.6, 3.4, 1.4, 0.3, u'Iris-setosa']))
(136, ((0, 2.271997065725805), [6.3, 3.4, 5.6, 2.4, u'Iris-virginica']))
(11, ((0, 2.6201979568981666), [4.8, 3.4, 1.6, 0.2, u'Iris-setosa']))
(141, ((0, 2.032429744583234), [6.9, 3.1, 5.1, 2.3, u'Iris-virginica']))
```

FIGURE 3 – output Part 1

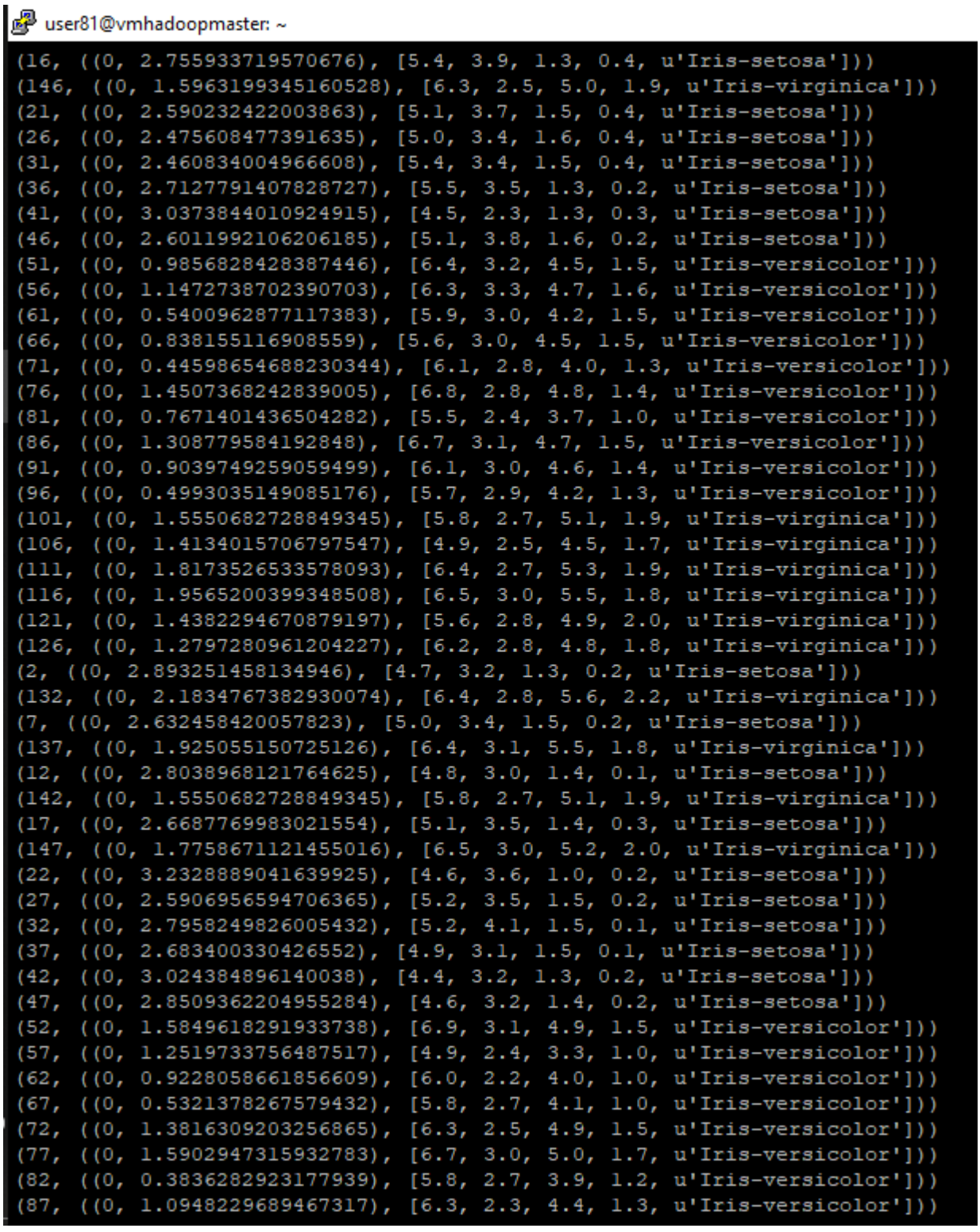

FIGURE 4 – output Part 2

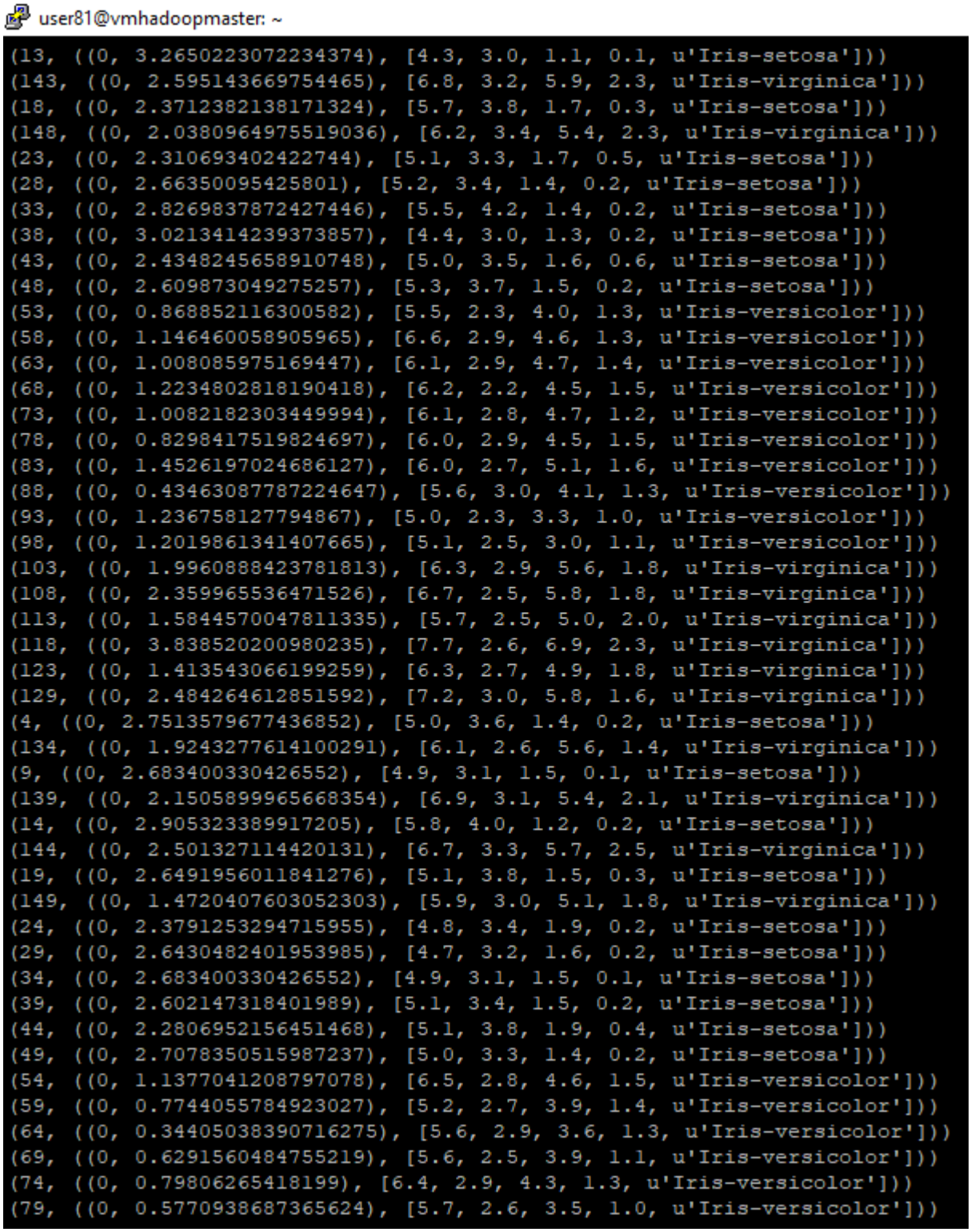

FIGURE 5 – output Part 3

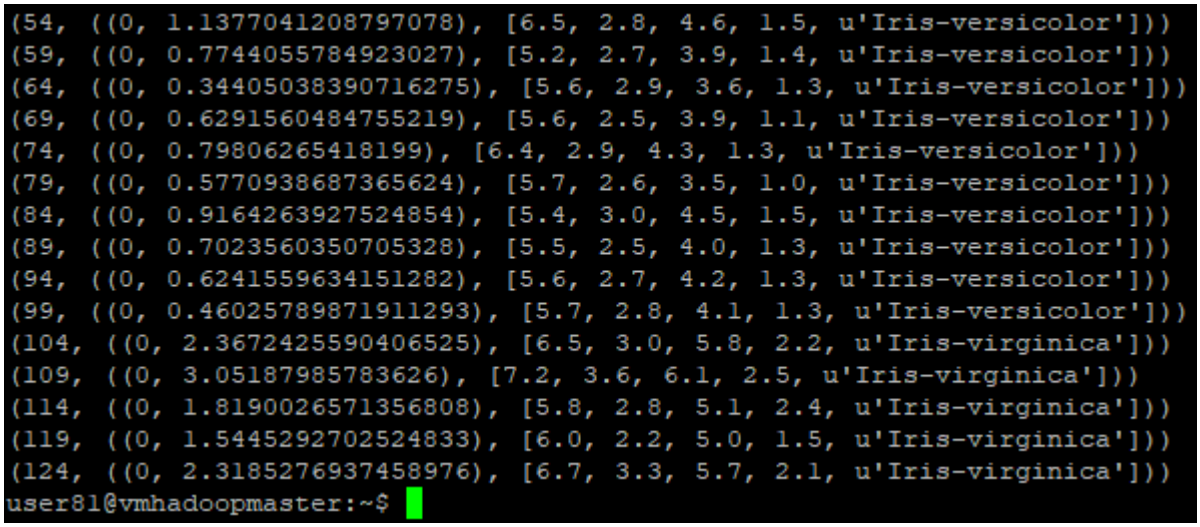

FIGURE 6 – output Part 4

5 Travaux futurs et perspectives : k-means ++ avec MapReduce

Nous consideons maintenant comment reformuler ce dernier algorithmes pour résoudre le problème des k-means++ afin qu'ils puissent être appliqués dans un cadre distribué. Plus précisément, nous formulerons une versions distribuées 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 datasets 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
textttMap et `Reduce` de notre algorithme MapReduce pour k-means++.

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` $(x, \min_{\mu \in M} ||x - \mu||_2^2)$

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 $x \in X$ 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 suit : $M = \{\mu_1\}$, où μ_1 est choisi uniformément au hasard parmi X.
(*ii*) Appliquez 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, initialisé avec ces

6 Conclusion

Pour ce projet, nous avons examiné le code fournis kmeans-dario-c.py. On a vu comment cet algorithmes de clustering pourraient être implémenté et optimisé.Tout d'abord, nous avons décrit les formulations et analysé leur complexité et leurs coûts. Ensuite, nous avons modifié cet algorithme pour produire une version plus rapide et optimisé. Aussi, on a vu comment on peut construire une extension trés interessante de cet algorithme dans un environnement informatique distribué à l'aide du paradigme de MapReduce.

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 10271035. Society for Industrial and Applied Mathematics, 2007.
3. https ://github.com/ArsMing276/Kmeans_Implementation_with_Mapreduce