UNIVERSITE PARIS-DAUPHINE TUNIS

MASTER INTELLIGENCE ARTIFICIELLE SCIENCE DES DONNEES

Optimisation de l'algorithme K-means en Pyspark

Professeur :

- Mr Dario Colazzo

Réalisé par :

- Ahmed Khalil Boulahia

Contents

1. Introduction : 3
2. Exploration du code de départ : 4
3. Optimisations : 5
- **Optimisation 1 : Agrégation des nouveaux centroïdes :** 5
- **Optimisation 2 : K-Means ++ :** 6
- **Optimisation 3 : Régler le niveau de parallélisme :** 8
- **Optimisation 4 : Broadcast() :** 8
- **Optimisation5 : Cache() :** 9

4. K-means en grand dimension : 10
- **Implémentation non optimisée :** 10
- **Implémentation optimisée :** 11

5. Conclusion : 11
6. Reference : 12

1. Introduction :

Dans le monde d'aujourd'hui, une quantité croissante de données est générée et collectée. Chaque algorithme traditionnel qui existe pour gérer une quantité importante de données se révèle inefficace lorsque les ensembles de données deviennent plus grands et que les itérations pour fixer certains paramètres initiaux augmentent également.

Dans le clustering K-Means, le nombre d'itérations dépend des centroïdes initiaux. La précision des centroïdes initiaux pris à un impact majeur sur le temps nécessaire pour terminer le clustering.

Les k centroïdes initiaux pris, pour lesquels certaines pratiques sont suivies, telles que le facteur de prise en compte de l'ordre, où le point de données arrive en premier k tuples est pris, et lorsque l'ordre change, les points de données pris initialement changent également, apportant ainsi une différence dans le temps d'exécution en augmentant le nombre d'itérations. Malheureusement, lorsque les points de données initiaux ont tendance à être aberrants, cela entraînerait des itérations inutiles pour localiser les points centroïdes réels en premier lieu.

Ce projet est une tentative pour atténuer ce problème en modifiant l'algorithme K-Means proposé afin d’atteindre un temps d’exécution acceptable.

2. Exploration du code de départ :

Les données d'entrée sont stockées sous la forme d'un fichier dans Hadoop Distributed File System (HDFS). Donc, nous devons d'abord charger ces données d'entrée dans des RDD, où les données sont divisées horizontalement et réparties sur plusieurs machines.

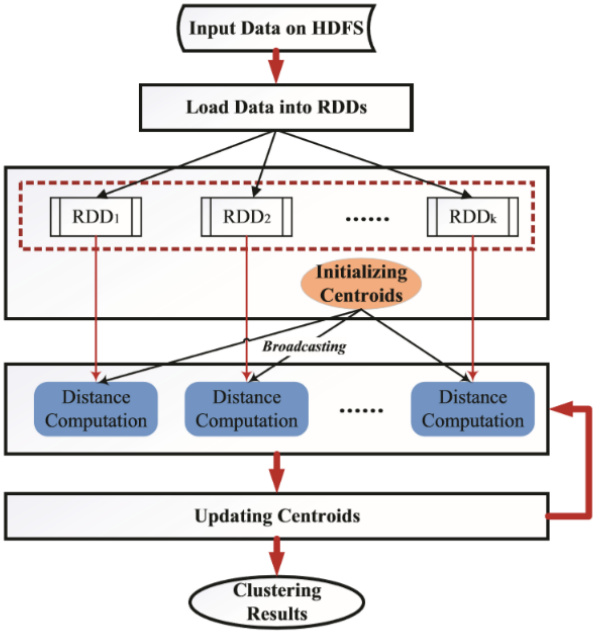


Figure [1]

L'initialisation des centroïdes est souvent exécutée en mode nœud avec un traitement centralisé. La manière la plus simple est de sélectionner au hasard K instances, mais on va procéder à une optimisation K-Means ++ .

L'exécution du code principale a donné le résultat suivant : 17min 32s

```
step 0 --> Num points have switched clusters:  150
step 1 --> Num points have switched clusters:  23
step 2 --> Num points have switched clusters:  8
step 3 --> Num points have switched clusters:  5
step 4 --> Num points have switched clusters:  3
step 5 --> Num points have switched clusters:  3
step 6 --> Num points have switched clusters:  2
step 7 --> Num points have switched clusters:  0
number of steps finale:  7
CPU times: user 2 s, sys: 517 ms, total: 2.52 s
Wall time: 17min 32s
```

3. Optimisations :

Le code initial met beaucoup de temps pour afficher un résultat d'où la nécessité d'introduire quelques optimisations.

- **Optimisation 1 : Agrégation des nouveaux centroïdes :**

Pour calculer les nouveaux centroïdes le code de départ a fait appel à trois **shuffles** :

- Un premier shuffle au niveau de **sum** : **Reduce Bykey(**)
- Le deuxième au niveau de **count** : **ReduceBykey ()**
- Le dernier shuffle fait appel à **join()** pour calculer la moyenne .

Pour réduire le nombre de shuffle on peut introduire un **CombineBykey** .
En effet, le CombineBykey prend trois arguments :

1-Create Combiner : qui sert à créer une nouvelle clé qu'il vient de voir pour la première fois dans une partition et à lui associé 1 .
2-Merge Value : qui sert à sommer deux éléments et implémenter le count

3-Merge combiners : qui combine des éléments de différentes partitions.

```
step 0 --> Num points have switched clusters:  150
step 1 --> Num points have switched clusters:  39
step 2 --> Num points have switched clusters:  13
step 3 --> Num points have switched clusters:  6
step 4 --> Num points have switched clusters:  3
step 5 --> Num points have switched clusters:  0
number of steps finale:  5
CPU times: user 738 ms, sys: 149 ms, total: 887 ms
Wall time: 2min 41s
```

- **Optimisation 2 : K-Means ++ :**

Pourquoi utiliser l'algorithme K-means plus plus ?

Un inconvénient de l'algorithme K-means est qu'il est sensible à l'initialisation des centroïdes ou des points moyens. Ainsi, si un centroïde est initialisé pour être un point «lointain», il pourrait simplement se retrouver sans aucun point associé et en même temps, plusieurs clusters pourraient se retrouver liés à un seul centroïde. De même, plus d'un centroïde peut être initialisé dans le même cluster, ce qui entraîne un mauvais clustering.

Comment surmonter cet inconvénient ?

Pour surmonter cet inconvénient, nous utilisons K-means ++. Cet algorithme assure une initialisation plus intelligente des centroïdes et améliore la qualité du clustering.

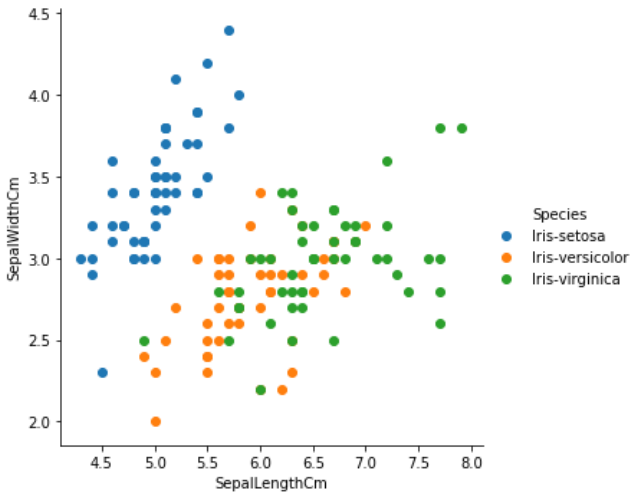
Outre l'initialisation, le reste de l'algorithme est le même que l'algorithme K-means standard. C'est-à-dire que K-means ++ est l'algorithme standard de K-means couplé à une initialisation plus intelligente des centroïdes.

Les étapes impliquées dans l'algorithme K-means plus plus

1. Sélectionnez au hasard le premier centroïde à partir des points de données.
2. Pour chaque point de données, calculez sa distance par rapport au centre de gravité précédemment choisi.
3. Sélectionnez le centroïde suivant parmi les points de données de sorte que la probabilité de choisir un point comme centroïde soit directement proportionnelle à sa distance par rapport au centroïde précédemment choisi. (C'est-à-dire que le point ayant la distance maximale du centroïde le plus proche est le plus susceptible d'être sélectionné ensuite comme centroïde)
4. Répétez les étapes 2 et 3 jusqu'à ce que k centroïdes aient été échantillonnés

L'application de K-means++ sur l'ensemble de données iris :

En suivant la procédure ci-dessus pour l'initialisation, nous prenons des centroïdes qui sont loin les uns des autres. Cela augmente les chances de ramasser initialement les centroïdes qui se trouvent dans différents groupes. On remarque que le Dataset Iris a la distribution suivante :



Voici les centres fixés avec l'algorithme k means :

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

Voici les centres fixés avec l'algorithme k means plus plus :

[(0, [6.6, 3.0, 4.4, 1.4, 'Iris-virginica']),
(13, [4.3, 3.0, 1.1, 0.1, 'Iris-setosa']),
(94, [5.6, 2.7, 4.2, 1.3, 'Iris-versicolor'])]

Puisque nous avons 3 espèces de Iris dans notre dataset, on remarque que le choix des centroïdes initiaux avec l'algorithme K means plus plus est plus pertinent que le choix de l'algorithme standard, on a 3 centroïdes chacun initialisé dans un cluster diffèrent, ce qui entraîne un bon clustering.

- **Optimisation 3 : Régler le niveau de parallélisme :**

L'ensemble de données dans Spark DataFrames et RDD est séparé en ensembles de données plus petits appelés partitions. Par défaut, Spark utilise la configuration du cluster (nombre de nœuds, mémoire, etc.) pour décider du nombre de partitions. Les partitions sont réparties entre les nœuds du cluster avec redondance. Les partitions aident au calcul distribué et au traitement parallèle.

Lorsque on effectue des agrégations ou des regroupements, nous pouvons demander à Spark d'utiliser un nombre spécifique de partitions.

Comme la répartition est une opération coûteuse : en effet Repartition() effectue un shuffle complet , nous pouvons utiliser coalesce () mais avant d'appeler coalesce (), nous devons appeler getNumPartitions ().

Étant donné que le calcul des distances dans Kmeans prend le plus de temps on a donc intérêt à diminuer le nombre de partitions afin de minimiser le traffic entre les nœuds du cluster.

- **Optimisation 4 : Broadcast() :**

Après initialisation des centroïdes, on appelle un Broadcast() pour partager les centroïdes dans chaque machine. Tout comme l'étape d'initialisation, les nouveaux centroïdes sont stockés en tant que variable de diffusion sur Spark.

```
step 0 --> Num points have switched clusters:  150
step 1 --> Num points have switched clusters:  31
step 2 --> Num points have switched clusters:  4
step 3 --> Num points have switched clusters:  3
step 4 --> Num points have switched clusters:  1
step 5 --> Num points have switched clusters:  0
number of steps finale:  5
CPU times: user 640 ms, sys: 162 ms, total: 803 ms
Wall time: 2min 31s
```

Une nouvelle itération est alors lancée jusqu'à ce que certaines conditions d'arrêt soient remplies.

- **Optimisation5 : Cache() :**

Il existe plusieurs avantages de la mise en cache RDD et du mécanisme de persistance dans Spark.

-Rentable : Pas besoin de calculer le résultat à chaque étape.

-Gain de temps : le calcul sera plus rapide

Cela permet de réduire le temps d'exécution d'un algorithme itératif sur de très grands ensembles de données.

```
step 0 --> Num points have switched clusters:  150
step 1 --> Num points have switched clusters:  36
step 2 --> Num points have switched clusters:  8
step 3 --> Num points have switched clusters:  0
number of steps finale:  3
CPU times: user 353 ms, sys: 50.2 ms, total: 403 ms
Wall time: 20.8 s
```

4. K-means en grand dimension :

Comme spark est un cadre applicatif de traitements big data, on va tester les performances des optimisations précédentes en grand dimension.

Pour cela, on va utiliser une dataset disponible sur kaggle. (https://www.kaggle.com/arjunbhasin2013/ccdata/download/wk0ZZ1jQdTPN7nbaKNph%2Fversions%2FoNKg9zDAcb3OXz1HBX7c%2Ffiles%2FCC%20GENERAL.csv?datasetVersionNumber=1)

Cette dataset contient 8950 instances de détails de carte de crédit client au format csv.

On a normalisé les données et les conservé au forma .txt

- **Implémentation non optimisée :**

On a d'abord expérimenté les données avec l'implémentation non optimisé de k-means.

L'exécution du code a donné le résultat suivant :

```
step 0 --> Num points have switched clusters:  8950
step 1 --> Num points have switched clusters:  2098
step 2 --> Num points have switched clusters:  1017
step 3 --> Num points have switched clusters:  392
step 4 --> Num points have switched clusters:  218
step 5 --> Num points have switched clusters:  102
step 6 --> Num points have switched clusters:  51
step 7 --> Num points have switched clusters:  38
step 8 --> Num points have switched clusters:  16
step 9 --> Num points have switched clusters:  12
step 10 --> Num points have switched clusters:  9
step 11 --> Num points have switched clusters:  0
number of steps finale:  11
CPU times: user 22.8 s, sys: 7.6 s, total: 30.4 s
Wall time: 2h 11s
```

- **Implémentation optimisée :**

Une amélioration remarquable est constatée en expérimentant les données sur l'implémentation optimisée de k-means.

En effet, l'exécution du code a donné le résultat suivant :

```
step 0 --> Num points have switched clusters:  8950
step 1 --> Num points have switched clusters:  402
step 2 --> Num points have switched clusters:  99
step 3 --> Num points have switched clusters:  178
step 4 --> Num points have switched clusters:  37
step 5 --> Num points have switched clusters:  3
step 6 --> Num points have switched clusters:  0
number of steps finale:  6
CPU times: user 1.18 s, sys: 272 ms, total: 1.46 s
Wall time: 4min 2s
```

5. Conclusion :

Après avoir utilisé toutes les optimisations mentionnées ci-dessus, nous avons réussi à réduire le temps de l'algorithme K-means de 50 fois de 17 minutes à 20 secondes.

Les sélections initiales des centoides ont un impact significatif sur le temps d'exécution des k-means, cela 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.

6. Reference :

- [1] : figure numero 1:
  Bowen Wang, Jun Yin, Qi Hua , Zhiang Wu, and Jie
  Cao, ‘ParallelizingK-means-basedClusteringonSpark ‘,pp 32.