Travaux Dirigés n°3

Data Analytics
—M2 IF – Apprentissage—

Implémentation de KMeans avec Spark

— 1 séance + travail personnel —

— KMeans
— Spark, RDDs

Dans ce TD/Projet, vous allez implémenter l'algorithme KMeans grâce à Spark et évaluer les performances de votre implémentation en utilisant un générateur de données synthétiques que vous devez également implémenter. Contrairement aux TDs précédents, nous vous encourageons à utiliser les `RDDs` (1) qui sont efficaces et plus adaptés que les DataFrame pour l'implémentation d'algorithmes.

Ce TD est à finir chez vous **en binome** (2), il doit être accompagné d'un rapport d'environ 5 pages ou plus. Le rapport et le code seront relevés et notés. Voici quelques conseils pour le rendu.

— Respectez les consignes avant tout, votre projet sera pénalisé si votre code ne fonctionne pas comme prévu.
— Une partie de la note sera attribuée pour la qualité du code. Faites un effort de clarté (utilisez des fonctions, nommez vos variables, etc.)
— Le rapport doit décrire vos choix d'implémentation et rapporter les résultats des expériences menées au cours de cet exercice. Vous êtes relativement libre de son contenu, à vous de faire un rapport intéressant et de mettre en avant les résultats non triviaux. Vous êtes fortement encouragés à tracer des courbes et à les inclure dans votre rapport lorsque cela est nécessaire.

Attention :
— Vous devez travailler en binome, toute collaboration inter-binome est strictement interdite et sera fortement pénalisés.
— vous devez **systématiquement** faire référence à tout code dont vous vous inspirez. Vous n'avez pas le droit de vous inspirer du code de vos camarades à l'exception de celui de votre binome. Un détecteur de plagia sera systématiquement exécuté sur l'ensemble des projets ainsi que sur différents projets trouvés sur internet. En cas de fraude avérée, votre projet sera très fortement pénalisé.

(1). Et les méthodes qui leur sont associées : `map()`, `reduce()`, `reduceByKey()`...
(2). Pas d'exception.

▸ **Exercice 1. Implémentation de KMeans**

L'algorithme KMeans, est un algorithme bien connu qui permet de découvrir efficacement des clustering de bonne qualité, mais pas forcément le clustering optimal. Il est décrit par Françis Back dans le document suivant : `https://www.di.ens.fr/~fbach/courses/fall2010/cours3.pdf`.

L'algorithme se décompose en deux étapes principales : une première étape dans laquelle on affecte chaque point à son centroïde le plus proche, et une deuxième étape dans laquelle on recalcule les coordonnées des centroïdes de chaque cluster en calculant la moyenne des coordonnées des points qui lui sont rattachés. Puis, ces deux étapes sont répétées jusqu'à ce que l'algorithme converge vers un optimum local.

1. Écrire une fonction `loadData(...)` qui permet charger les données dans un RDD. Aidez vous de la méthode `sparkContext.textFile()` pour y parvenir. Utilisez également split et filter, pour supprimer les entrées invalides. Ajoutez ensuite un identifiant à chaque point avec la méthode `zipWithIndex()`. L'exécution de la méthode `loadData()` doit retourner un RDD au format suivant (le format est donné à titre indicatif, d'autres méthodes sont possibles) :
   ```
   (0, [5.1, 3.5, 1.4, 0.2, 'Iris-setosa'])
   ...
   ```
2. Créez une fonction `initCentroids(...)`, qui retourne un nouveau RDD contenant les K centroïdes auquels vous aurez attribué des coordonnées tirées aléatoirement (aidez vous du module `mllib.random` pour la génération de nombre aléatoires). `InitCentroids()` doit retourner un nouveau RDD au format suivant :
   ```
   (0, [4.4, 3.0, 1.3, 0.2])
   ...
   ```
   Où `[4.4, 3.0, 1.3, 0.2]` sont les coordonnées du centroïde du premier cluster (cluster `0`).
3. Écrivez une nouvelle fonction `assignToCluster(...)` pour affecter chaque point des données à un cluster. Une méthode simple pour y parvenir est d'effectuer le produit cartésien (avec la méthode `RDD.cartesian()`) entre le RDD contenant les données, et le RDD contenant les centroïdes. Calculez ensuite la distance entre chaque point et chaque centroïde grâce à un appel à `RDD.map()`, et construisez un nouveau RDD associant à chaque point, le centroïde qui lui est le plus proche. La fonction doit retourner un RDD ayant la forme suivante :
   ```
   (123, (0, 2.3))
   (125, (1, 1.72))
   (127, (1, 0.3))
   ```
   Dans la première ligne, 123 est le numéro de l'instance, 0 est le numéro du centroïde le plus proche, et 2.3 est la distance entre le point et le centroïde le plus proche. Ceci constitue la première étape de KMeans.
4. La deuxième étape consiste à calculer les nouveaux centroïdes en calculant la moyenne des coordonnées des points qui lui sont rattachés. Écrivez une nouvelle fonction `computeCentroids(...)` qui retourne un RDD contenant les nouveaux

centroïdes. Le RDD doit avoir le même format que celui retourné par la fonction `initCentroids()`.

5. Écrivez maintenant une méthode `computeIntraClusterDistance(...)` qui calcule la distance intra cluster d'une affectation (c'est à dire, au RDD retourné par votre fonction `assignToCluster()`). Utilisez les fonctions que vous avez implémenté pour calculer la distance intra cluster sur le jeu de données Iris avant et après le premier appel à `computeCentroids()`. Comment doit évoluer cette distance ? Vérifiez que c'est bien le cas.
6. Placez vos appels de fonctions dans une boucle pour que KMeans répète les deux étapes jusqu'à convergence. Comment détecte-t-on la convergence ? En combien d'étapes votre algorithme converge-t-il sur Iris ?
7. Modifiez votre algorithme pour qu'il fonctionne avec un nombre quelconque de clusters (déterminé par l'utilisateur) et un nombre quelconque de dimensions (déterminé par les données). Rajoutez également une borne supérieur pour fixer le nombre maximal d'itérations. Votre algorithme doit pouvoir s'exécuter avec la commande suivante (Impératif !)

   `spark-submit kmeans.py data.csv k m` où

   — `data.csv` est le nom de fichier contenant les points à partitionner
   — `k` est le nombre de clusters
   — `m` est le nombre de maximum d'itérations.

   L'algorithme doit terminer en affichant (au moins) les lignes suivantes :

   ```
   Nombre iteration: X
   Distance finale: Y
   ```

   (Soyez rigoureux, ces deux lignes seront utilisées pour tester et noter votre projet)
8. Reproduisez l'expérience 100 fois et donnez la distance intra cluster moyenne obtenue, et son écart type. Comparez votre algorithme avec le KMeans disponible dans la librairie MLlib.
9. Instrumentez votre code pour afficher la distance intra cluster aux différentes itérations.
10. Quel problème peut-on rencontrer si la position intiale des centroïdes n'est pas favorable ? Illustrer avec un dessin, et proposer une ou plusieurs solutions pour limiter ou éviter ce problème (nombreuses solutions possibles). Tentez d'améliorer votre algorithme et valider expérimentalement le gain de performance en reproduisant les expériences précédentes.

▸ Exercice 2. Évaluation de KMeans

Dans cet exercice, vous allez utiliser Spark pour générer de grands volume de données synthétiques et évaluer les performances de votre implémentation de KMeans.

Les coordonnées des points générés pour le cluster k seront tirées aléatoirement en suivant une loi normale de dimension p, paramétrée par une moyenne μ_k choisie

aléatoirement entre 0 et 100 et différente pour chaque cluster, et par un écart type σ fixé par l'utilisateur et identique pour tous les clusters.

Le générateur devra être un programme indépendant, pouvant s'exécuter de la manière suivante : `spark-submit generator.py out.csv n k p s` où

- `out.csv` est le nom de fichier à générer
- `n` est le nombre total de points à générer
- `k` est le nombre de clusters
- `d` est la dimensionalité des données, donc `d` = 4, signifie que vous devez générer un vecteur de dimension 4 pour chaque point.
- `s` est l'écart type σ.

L'exécution du programme devra produire un fichier csv nommé `out.csv` contenant `n` lignes au format suivant : `coord-1, coord-2, ..., coord-p, cluster-id`.

Par exemple, la commande `spark-submit generator.py out.csv 9 3 2 10` devra générer un fichier `out.csv` contenant 9 points à 2 coordonnées répartis dans 3 clusters, semblable au fichier suivant :

```
30.0, 110.0, 0
35.8, 104.0, 0
7.7, 70.1, 0
-5.4, 9.9, 1
30.6, 37.2, 1
6.9, 45.5, 1
10.2, 34.2, 2
0.5, 37.1, 2
34.0, 46.2, 2
```

Pour information, les lois de probabilités utilisées pour la génération des trois clusters de ce jeu de données ont les moyennes suivantes : $\mu_0 = (23.3, 90.2), \mu_1 = (15.2, 67.8)$ et $\mu_2 = (12.8, 31.0)$. L'écart type σ est fixé à 10 pour tous les clusters et toutes les dimensions, comme indiqué dans la ligne de commande.

1. Implémentez le générateur de données synthétiques en vous aidant de la classe `mllib.random.RandomRDDs` et de la méthode statique `random.RandomRDDs.normalVectorRDD`
2. Utilisez votre générateur de données pour vérifier que vôtre algorithme permet de retrouver, les cluster dans la plupart des cas.
3. Utilisez votre générateur de données pour mesurer les temps d'exécution en fonction du nombre de différents paramètres, et comparer les différentes approches que vous avez implémenté.