Projet Big Data

Début du sujet

M2 MLDS/AMSD - 2023-2024

Consignes générales

Conditions de rendu à définir ultérieurement.

Le livrable consistera en un **rapport concis** ainsi que l'**implémentation** (code). Il pourra se faire dans un même document sous forme de notebook.

Les questions sont données pour guider l'implémentation, une attention particulière sera portée à l'**explication** des choix d'implémentation, à l'analyse des forces et faiblesses en termes de **parallélisation** et de **gestion de la mémoire**. L'exécution du code sur des données illustrera le rapport permettant de montrer que les résultats des implémentations sont cohérents.

Contexte et objectifs

Lorsque les données sont très volumineuses, il n'est plus possible d'appliquer des méthodes qui supposent de les avoir toutes en mémoire en même temps sur une seule machine.
Le but de ce projet est de voir plusieures méthodes qui visent le même objectif : classifier des points de manière non supervisée. K-means est choisi comme objet d'étude. Chaque partie correspond à une manière de l'implémenter, aucune n'est universellement meilleure que les autres puisque chacune fait des choix différents pour répondre à certaines contraintes.

A. Implémentation de k-means séquentiel (Python)

1. Générer un jeu de données consistant en des points répartis en deux classes. On pourra par exemple générer une première classe de 100 points répartis à proximité d'un centroïde de coordonnées (5, 5) et une autre classe où 100 points sont autour des coordonnées (10, 10). Enregistrer ces données dans un fichier (par exemple au format CSV).
2. Lire les données en consommant peu de mémoire (l'exécution doit pouvoir être possible si le nombre de points augmente drastiquement).
3. Implémenter l'algorithme k-means séquentiel en Python :
    a. Choix aléatoire (ou simplement les premiers points) de k centres μ_1, ..., μ_k

b. Pour chaque nouveau point x_i :
    i. Calculer le centre μ_j le plus proche de x_i. On note n_j l'effectif de la classe associée.
    ii. Mettre à jour le centre μ_j et l'effectif n_j :
        - $\mu_j \leftarrow \mu_j + \frac{1}{n_j+1}(x_i - \mu_j)$
        - $n_j \leftarrow n_j + 1$
4. Enregistrer les résultats de l'algorithme dans un fichier de manière à consommer peu de mémoire.
5. Valider la cohérence des résultats. On pourra pour cela les visualiser et/ou utiliser des mesures d'évaluation adaptées.

B. Implémentation d'une version *streaming* de k-means (Python)

L'implémentation séquentielle de k-means souffre d'un inconvénient : si la distribution des données change au cours du temps (*concept drift*), les centres se déplacent et l'affectation aux clusters n'est alors plus toujours cohérente. Reboucler sur les données permet d'atténuer cet effet. Dans cette partie, une autre solution est proposée. Elle permet de s'affranchir de reboucler sur toutes les données en conservant en mémoire un sous-ensemble des données : les dernières arrivées.

Implémenter l'algorithme suivant :

Hyper-paramètres : Soit T le nombre maximum de *batches* à garder en mémoire, et un paramètre r qui contrôle le poids à accorder à l'historique (plus il est grand, moins les anciens *batches* pèseront dans la contribution aux centres).

Entrées : X l'ensemble des *batches* précédents partitionnés par P (aléatoire si il n'y a pas de *batch* précédent), et B^0 le nouveau *batch*.

Algorithme :
- Si la taille de X est T : enlever le plus vieux *batch* de X
- Ajouter B^0 à X
- Initialiser les centroïdes C avec la partition P
- Obtenir les centroïdes C et la partition P avec l'algorithme k-means pondéré :
    - les points des *batches* sont pondérés par r^t où t est le numéro du *batch* ordonné par ordre décroissant : 0 est le *batch* le plus récent, 1 est le *batch* précédent, etc.

- utiliser l'argument `sample_weight` de la méthode `fit` de l'implémentation k-means de scikit-learn : https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html#sklearn.cluster.KMeans.fit

Sortie : l'ensemble des centroïdes C et la partition associée P.

C. Implémentation de k-means distribué (Apache Beam)

Pour cette partie, on pourra dans un premier temps se placer en dimension 1 (une seule coordonnée par point).

1. Créer une `PCollection` qui contient l'ensemble des points.
2. Pour l'initialisation, transformer la `PCollection` précédente pour que chaque élément de la nouvelle `PCollection` soit un tuple où :
    a. le premier élément est le numéro de cluster choisi aléatoirement ;
    b. le second élément du tuple soit les coordonnées du point.
3. Implémenter l'étape de calcul des centres : Une `PCollection` nommée `centroids` sera créée pour ça. Chaque élément sera un tuple (numéro de cluster, coordonnées du centroïde).
4. Implémenter l'étape de partitionnement :
    a. Créer une fonction `assign_cluster` qui prend deux entrées :
        i. un point (ses coordonnées)
        ii. un dictionnaire de centroïdes où la clé correspond au numéro de cluster et la valeur correspond aux coordonnées du centroïde.

        La fonction retourne un tuple avec :
        - le numéro du cluster le plus proche du point ;
        - les coordonnées du point.
    b. Assigner à chaque point son numéro de cluster :
        i. Utiliser la méthode Map avec les centroïdes comme entrée complémentaire sous forme de dictionnaire : https://beam.apache.org/documentation/transforms/python/elementwise/map/#example-8-map-with-side-inputs-as-dictionaries
        ii. Quelle supposition fait-on pour passer les centroïdes sous cette forme ? Peut-on optimiser cette étape ? Si oui, modifier l'implémentation de l'étape de partitionnement pour pallier ça.