Projet Big Data

M2 MLDS/AMSD - 2023-2024

Consignes générales

Le projet est à faire par groupes de 3 personnes. La date limite de rendu est à définir ultérieurement. Un mail est à envoyer à stanislas.morbieu@gmail.com avec pour objet « M2-MLDS/AMSD projet big data ». Le corps du mail contiendra les noms et prénoms de chacun des membres du groupe.

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.

D. Implémentation de k-means séquentiel distribuée (Apache Beam)

Implémenter avec Apache Beam l'algorithme décrit en partie A.

On pourra s'inspirer du code suivant qui permet de garder un état (voir explication après le code).

```
1  import apache_beam as beam
2  from apache_beam.transforms.userstate import BagStateSpec
3
4
5  class MyClass(beam.DoFn):
6     state = BagStateSpec(name='numbers',
coder=beam.coders.PickleCoder())
7
8     def process(self, element, numbers=beam.DoFn.StateParam(state)):
9       key, value = element
10      numbers.add(value)
11      yield key, set(numbers.read())
12
13
14 with beam.Pipeline() as pipeline:
15  data = (
16      pipeline
17      | "Create" >> beam.Create(
18          # un état est créé par clé (première valeur du tuple)
19          # la clé vaut 0 ou 1 (reste de la division entière de k par
2)
20          [(k % 2, k) for k in range(10)]
21      )
22  )
23
24  (data
25   | "Update state" >> beam.ParDo(MyClass())
26   | "Print" >> beam.Map(print)
27  )
```

Le code précédent produit :

```
(0, {0})
(1, {1})
(0, {0, 2})
(1, {1, 3})
(0, {0, 2, 4})
(1, {1, 3, 5})
(0, {0, 2, 4, 6})
(1, {1, 3, 5, 7})
(0, {0, 2, 4, 6, 8})
(1, {1, 3, 5, 7, 9})
```

Explication :
Les lignes 17 à 21 créent le jeu de données suivant :

```
[(0, 0),
 (1, 1),
 (0, 2),
 (1, 3),
 (0, 4),
 (1, 5),
 (0, 6),
 (1, 7),
 (0, 8),
 (1, 9)]
```

La fonction définie ligne 8 est appliquée à chaque élément de la `PCollection` (ligne 26). Celle-ci crée un état pour chaque valeur de clé différente (ici deux état puisque la clé est soit 0 soit 1). L'état est stocké dans la variable `numbers`. On peut ajouter un élément dans l'état avec la méthode `add` (ligne 10). On peut lire la valeur de la variable d'état avec la méthode `read` (ligne 11). Dans l'exemple donné, on retient toutes les valeurs dans l'état. La nouvelle `PCollection` produite est le résultat de la fonction `process` appliquée à chaque élément.
Trois méthodes peuvent s'appliquer à la variable d'état :

- `numbers.add(value)` ajoute `value` dans la variable d'état (un élément en plus de ce qu'il y avait déjà) ;
- `numbers.clear()` vide la variable d'état ;
- `numbers.read()` permet de récupérer ce qui est dans la variable d'état.

Pour implémenter l'algorithme k-means séquentiel, on peut conserver dans l'état les centres et les effectifs des classes. On pourra dans un premier temps utiliser une clé unique pour n'avoir qu'un seul état partagé.

Quel est l'implication d'avoir qu'une seule clé unique (et donc un seul état) en termes de parallélisation ? Peut-on optimiser ça ?

E. Implémentation d'une version *streaming* et distribuée de k-means (Apache Beam)

Implémenter avec Apache Beam l'algorithme décrit en partie B.

On peut utiliser le même principe que pour la partie D et conserver les *batches* dans la variable d'état.