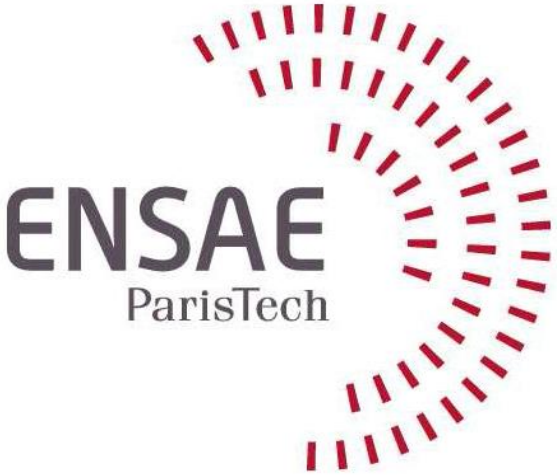
ENSAE ParisTech

Éléments logiciels pour le traitement des données massives

MapReduce Affinity Propagation Clustering Algorithm



Maxence BROCHARD, Alexis ROSUEL, Thomas SELECK

05/02/2017

Introduction

Au cours de ce projet, nous allons proposer une implémentation de l'algorithme « MapReduce Affinity Propagation Clustering Algorithm » proposé par Wei-Chih Hung, Chun-Yen Chu, et Yi-Leh Wu.

Explication de l'algorithme Affinity Propagation Clustering Algorithm

Avant de s'intéresser à l'algorithme type « MapReduce », commençons par comprendre le fonctionnement de « Affinity Propagation Clustering ». Cet algorithme fait partie de la famille des algorithmes de clustering, c'est-à-dire qu'à partir d'une liste d'individus, il doit les répartir en différentes catégories. A la différence de l'algorithme k-means qui prend en entrée aussi le nombre de clusters que l'on souhaite créer, Affinity Propagation détermine tout seul ce paramètre.

Son principe est le suivant :

- On a à disposition une liste de n individus définis par p variables
- On calcule pour chaque couple d'individus leur « similarité » (plus elle est élevée, plus les points sont « proches », et inversement). Lorsque les variables sont toutes continues, on utilise la norme euclidienne.
- On calcule récursivement deux matrices (« Availability » et « Responsibility », respectivement A et R dans la suite) représentants pour chaque couple de points (i, j) comparativement à tous les autres points avec quelle qualité i peut être représenté par j, et avec quelle qualité j pourrait être choisi en tant que représentant de i.

Lorsque la convergence est atteinte (i.e. les matrices A et R ont atteintes un point d'équilibre), on peut extraire directement les points qui sont choisis comme étant des représentants de tous les autres, c'est-à-dire les centroïdes.

Le fait que cet algorithme soit capable de décider seul du nombre de centres à utiliser en fait un excellent candidat pour en écrire une forme distribuée. En effet, lors de l'étape du mapper, les données sont séparées en potentiellement de nombreux sous-jeux de données, et il est possible que certains clusters présents dans le jeu en entier ne soit plus présent dans un des sous-jeux. Il ne faudrait donc pas forcer l'algorithme à absolument ajuster k centres dans ce cas (figure 1).

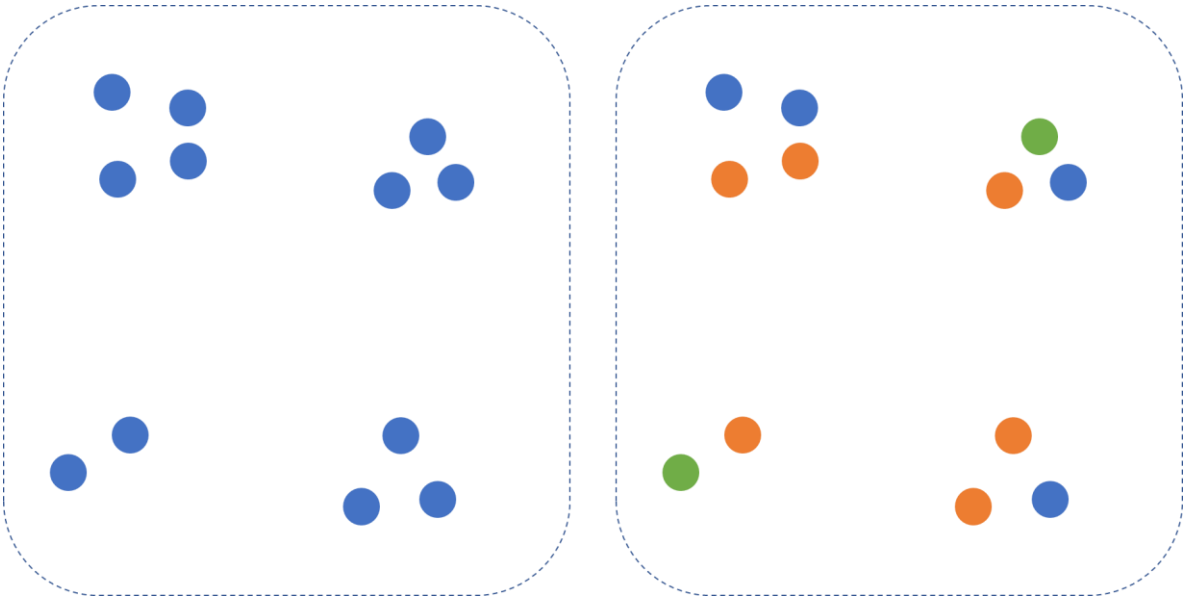


Figure 1 - 4 clusters initialement, mais seulement 3 avec le sous-jeu bleu, et 2 avec le sous-jeu vert

Explication de la version MapReduce de l'algorithme

L'idée des auteurs de l'article est d'appliquer l'algorithme précédent à différents sous-jeux de données pour plusieurs raisons :

- Tirer profit de la parallélisation des nœuds de calcul
- Utiliser les capacités de stockage distribuées pour appliquer l'algorithme à des volumes de données qui ne tiennent pas sur une seule machine.
- Le coût de l'algorithme étant au moins quadratique (il faut maintenir trois matrices de taille $n \times n$), il est très intéressant de séparer les données en plusieurs parties afin de réduire la complexité algorithmique.

Le mapper est donc l'implémentation exacte de l'algorithme original. Il reste à écrire les reducers afin de consolider les résultats de chacun des mappers. En effet, plusieurs représentants appartenant à un unique « vrai cluster » peuvent être calculés par les différents mapper, il faut donc écrire une méthode qui définit comment choisir le représentant final (figure 2).

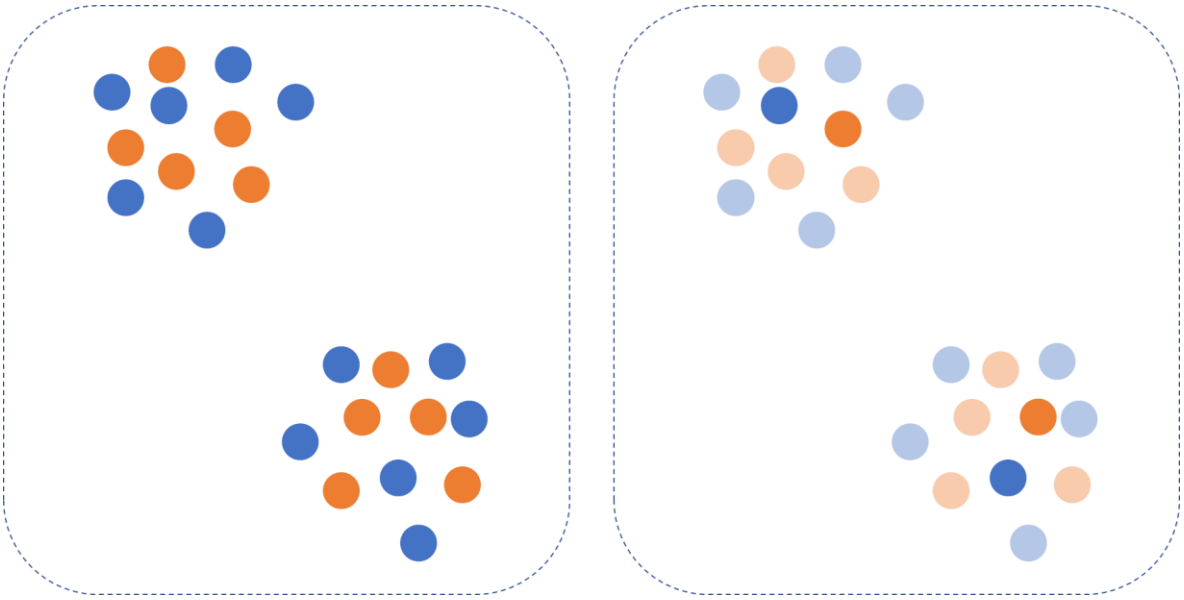


Figure 2 - Représentants choisis pour chacun des sous jeux de données

L'idée générale est donc de :

- Identifier les représentants issus d'un même cluster
- Les agréger en prenant leur moyenne

Une fois que cette étape est réalisée, on associe à chaque individu le centroïde dont la similarité est la plus grande. L'étape de clustering est donc terminée.

Résultats

Nous avons implémenté le mapper de deux manières différentes. Au début, nous étions partis sur une implémentation en utilisant uniquement les RDD Spark, mais on s'est rendu compte que les temps de calculs étaient beaucoup trop important. Nous avons alors suivi les indications de Monsieur Xavier Dupré, et proposé une implémentation utilisant la structure de dataframe Spark (et qui repose sur des requêtes SQL), mais le temps d'exécution est devenu bien plus important. A noter que les temps d'exécution divergent selon les machines utilisées.

Nombre de points / itérations	RDD	Dataframe
12 points / 10 itérations	93 secondes	3 heures

Tableau 1 - Temps d'exécution des deux méthodes

Nous n'avons pas trouvé d'explication pour ce problème. En effet, nous nous attendions à un gain de temps étant donné que les données et les requêtes SQL doivent être optimisées par rapport à la structure de base (RDDs).

Concernant les résultats, ils sont sensiblement les mêmes entre les deux méthodes. En effet, le calcul de la médiane avec les RDD est exact, alors qu'il est estimé via un autre algorithme dans le cas des dataframes. Ainsi, de légères différences dans les calculs des matrices A et R apparaissent et s'amplifient légèrement au fur et à mesure des itérations. Les résultats finaux (ie. la liste des individus choisis comme représentants) n'est cependant pas modifiée.

Aussi, voici ci-dessous la liste des fichiers que nous vous livrons :

- *Classic Affinity Propagation.ipynb* : implémentation Python non distribuée de l'algorithme
- *Map Reduce Affinity Propagation algorithm - current version.ipynb* : implémentation du Mapper en utilisant les RDDs Spark
- *Map Reduce Affinity Propagation algorithm - SQL DataFrame Version.ipynb* : implémentation du Mapper en utilisant les dataframes Spark

Voici le résultat de l'algorithme sur un jeu de donnée que nous avons créé (mixture de trois lois normales bidimensionnelles).

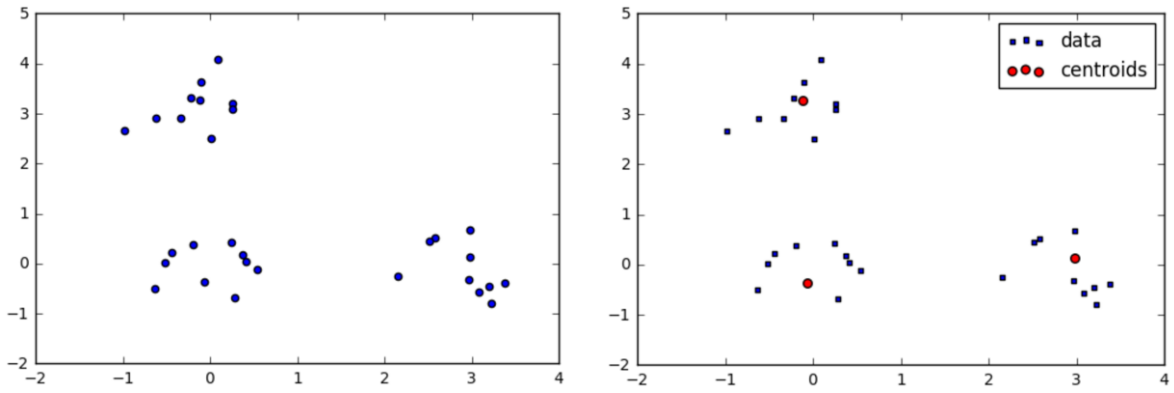


Figure 3 - Résultat de l'algorithme

Conclusion

Ce projet nous a permis de passer beaucoup de temps à comprendre la manière dont fonctionne Spark, à la fois en utilisant les RDD mais également à l'aide de DataFrames Spark. Malheureusement, nous n'avons pas pu aller jusqu'au bout de l'implémentation proposée par l'article.

Aussi, et de manière générale, nous confirmons que les performances de Spark en local sont médiocres comparées à une utilisation « normale » de Python sur une machine. Spark prend tout son intérêt lorsque nous avons à disposition de nombreux cœurs de calculs et de stockage. On notera également que Spark repose avant tout sur une implémentation Java qui nécessite une machine virtuelle – la JVM – pour pouvoir fonctionner. Quant à PySpark, il exécute du code Python sur du Java, ce qui crée de nombreuses conversions de données et de nombreux transferts entre l'interpréteur Python et la JVM. Sans compter qu'étant donné que ces deux langages ne sont pas compilés, il sont donc naturellement plus lents que du C, par exemple.