|  |
| --- |
| 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](http://www.ijeee.net/uploadfile/2014/0807/20140807114023665.pdf).

# 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 ». 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 (« Availibity » et « Responsibility », respectivement et dans la suite) représentants pour chaque couple de point comparativement à tous les autres points avec quelle qualité peut être représenté par , et avec quelle qualité pourrait être choisi en tant que représentant de .

Lorsque la convergence est atteinte (ie. les matrices et ), on peut extraire directement les points qui sont choisis comme étant des représentants de tous les autres, c’est donc les centroïdes.

Le fait que cet algorithme soit capable de décider seul du nombre de centre 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 centres dans ce cas (figure 1).

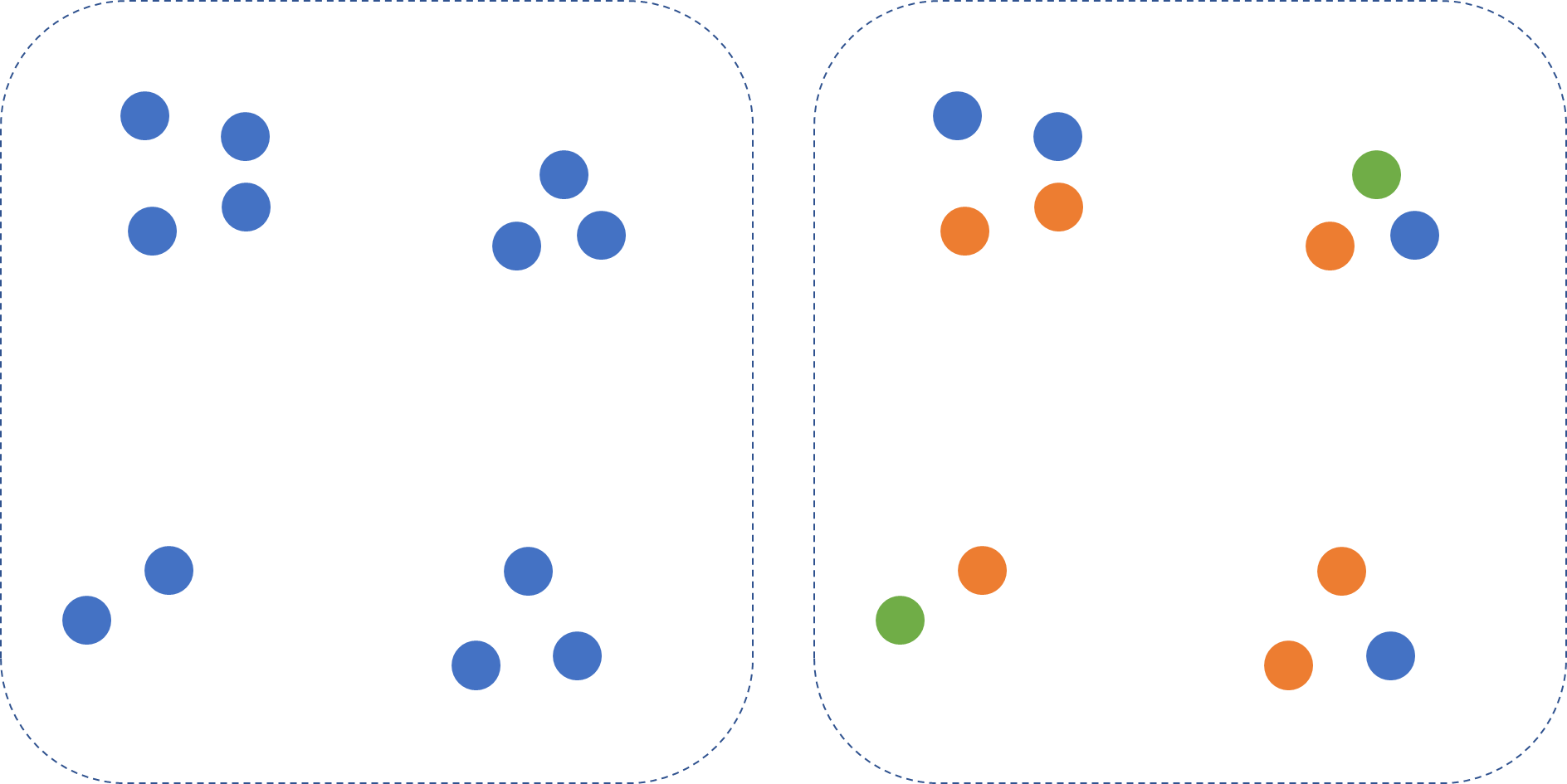


Figure - 4 clusters initialement, mais seulement 3 avec le sous-jeu bleu, et 2 avec le sous-jeu vert

## 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 ne tenant localement
* Le coût de l’algorithme étant au moins quadratique (il faut maintenir trois matrices de taille ), il est très intéressant de séparer les données en plusieurs parties.

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).

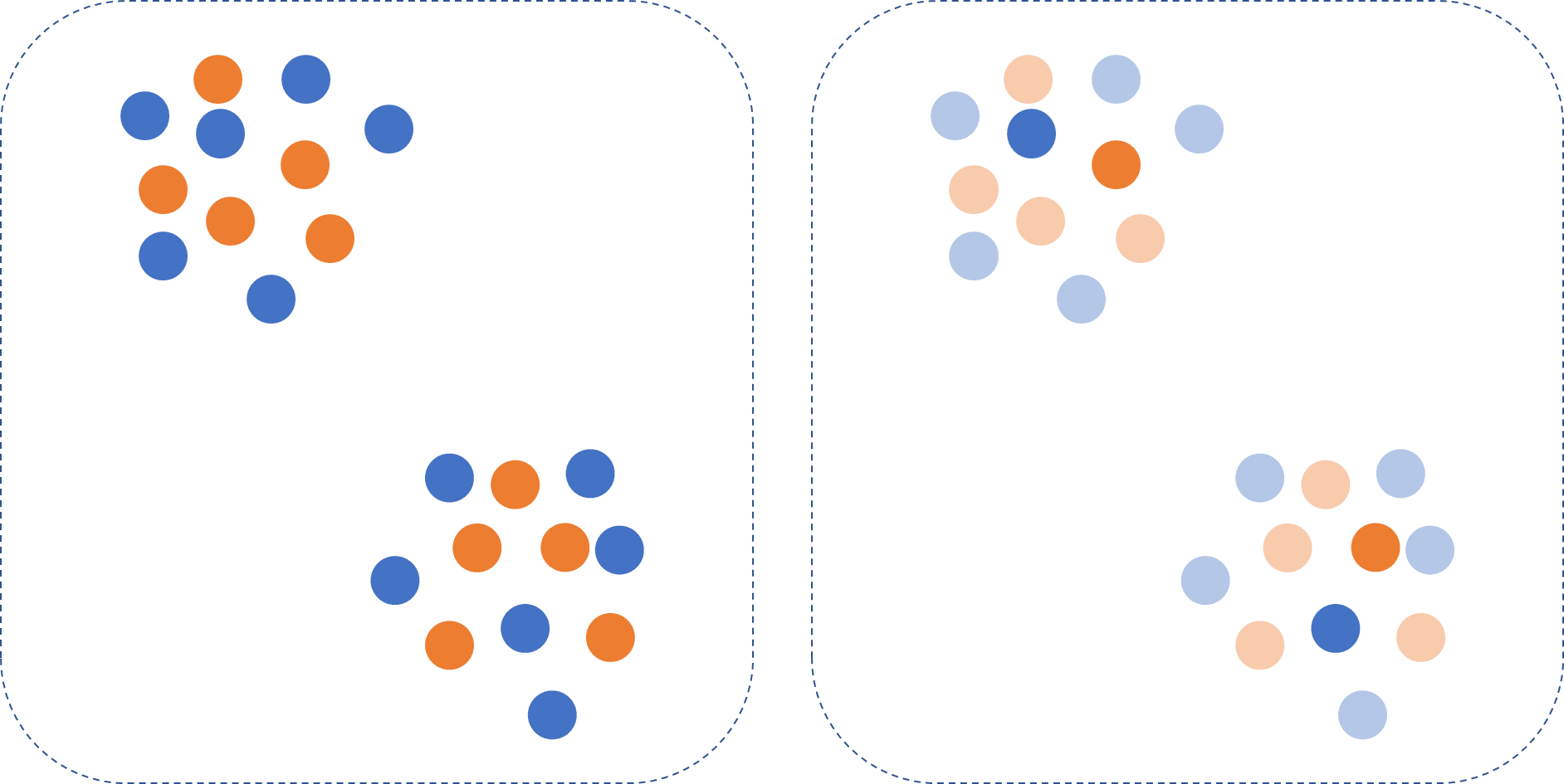


Figure - Représentants choisis pour chacun des sous jeux de données

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 aucun gain de temps n’a été observé sur les données que nous avons utilisées.

Voici ci-dessous les temps d’exécution (tableau 1).

|  |  |  |
| --- | --- | --- |
| Nombre de points / itérations | RDD | Dataframe |
| 12 points / 10 itérations | 93 secondes | 95 secondes |

Tableau - Temps d'exécution des deux méthodes

Les résultats 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 et 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 (tableau 2).

|  |  |
| --- | --- |
| 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 |
| Mapper.py | ?? |

Tableau - Liste des notebooks lvrés

Conclusion

Ce projet nous a permis de passer beaucoup de temps à comprendre la manière dont fonctionne Spark, à la fois les RDD mais aussi sa partie SQL. Malheureusement, nous n’avons pas pu aller jusqu’au bout de l’implémentation proposé par l’article.

Aussi, et de manière générale, on confirme que les performances de Spark en local sont médiocres comparées à une utilisation « normale » de Python sur une machine. Spark prend bien tout son intérêt lorsqu’on a à disposition de nombreux cœurs de calculs et de stockage.