Université Paris Dauphine

Data Analytics

Implementation de KMeans avec Spark

Etudiants
Elie Abi Hanna Daher
Bilal El Chami
Badr Erraji

Professeur
M. Benjamin Negrevergne

29 avril 2018

Table des matières

La méthode K-means est une des méthodes de clustering les plus utilisée lors de l'implémentation d'algorithmes cherchant à regrouper un ensemble de données disparates. Cette méthode repose principalement sur le *unsupervised learning*. L'objectif étant alors de grouper ces dernières en implémentant l'algorithme KMean avec Spark (python) et en évaluant la performance de notre implémentation basée sur des données que nous générons.

1 Travail effectué

1.1 Implementation

1.1.1 Kmeans

Toutes les fonctions demandées étaient implémentées en respectant le format donné. Ce format nous a énormément aider pour comprendre et développer l'algorithme en suivant le paradigme MapReduce en Spark.
Les méthodes de l'algorithme sont les suivantes :

- *customSplit(...)*
- *loadData(...)*
- *initCentroids(...)*
- *assignToCluster(...)*
- *calculateDistance(...)*
- *minDist(...)*
- *computeCentroids(...)*
- *reCalculating(...)*
- *hasConverged(...)*
- *computeIntraClusterDistance(...)*

Les méthodes *customSplit* et *loadData* préparent les données passées par l'utilisateur et les formatent en affectant un identifiant pour chaque enregistrement.

initCentroid est la fonction critique de l'algorithme qui sert à initialiser les centroïdes. Dans un premier temps, nous avons choisis ces centres aléatoirement parmi les données existantes en utilisant la fonction *takeSample(...)* de la classe RDD. Nous avons implémenté aussi l'algorithme K-means++ qui propose une procédure d'initialisation plus pratique (voire la partie suivante).

Ensuite, la méthode *assignToCluster()* prends les données et calcule la distance relative de chaque centroïdes. Puis elle choisit le centroïde ayant la distance minimale, en utilisant les méthodes *calculateDistance()* et *minDist()*, et l'affecte comme cluster de l'enregistrement. Le format utilisé nous a permis de profiter de l'utilisation de la méthode groupByKey de la classe RDD ce qui améliore les performances en passant à l'échelle.

Dans la fonction *computeCentroids()*, nous avons regroupé les résultats d'affectation des clusters, en utilisant autant que possible les fonctions de la classe

RDD (*join()* et *groupByKey()*). Après avoir grouper les données par cluster, nous avons recalculer les nouvelles valeurs des centroïdes à l'aide de la méthode *reCalculating()*.

La méthode *computeIntraClusterDistance()* permet de calculer la distance intra-cluster. Cette méthode calcule la somme des moyennes des distances des points aux clusters dont qu'il appartient.

Finalement, le programme recalcule à chaque fois les nouvelles valeurs des centroïdes et vérifie si l'algorithme a convergé. La condition de convergence est renvoyé par la méthode *hasConverged()*.

Observation importante
Une amélioration considérable des performances a été remarqué lors de la modification de l'instruction suivante qui casse le lineage du RDD centroids et crée un nouveau RDD pour l'itération suivante : *centroids = sc.parallelize(newCentroids.collect())*. Alors que l'initialisation de centroïdes pouvait se faire de manière aléatoire, ce qui restait fonctionnel.

1.1.2 Kmeans++

L'initialisation de centroïdes pouvait se faire de manière aléatoire, ce qui restait fonctionnel.

En effet, le fait de générer des centroïdes de manière aléatoire pouvait parfois mener à des situations bloquantes tel que le choix de deux centroids avec exactement les même coordonnées. Par ailleur, le clustering peut donner des résultats différents selon les centroids initiaux choisis qui peuvent parfois être très loin de l'optimum rallongeant les calculs donc avant convergence.

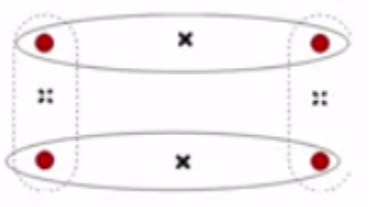


FIGURE 1 – Initialize centroids

Comme nous pouvons le voir sur l'exemple ci-dessus, le clustering peut se faire dans ce cas verticalement ou horizontalement selon points initiaux.

Nous avons décidé de suivre une autre technique certes quelque peu plus complexe mais qui réduisait le nombre d'itérations considérablement. Nous nous sommes basés sur l'algorithme K-means++ pour implémenter la méthode init centroïdes :

Kmeans ++ a été proposé en 2007 par David Arthur et Sergei Vassilvitskii. Le premier centre du cluster est choisi de manière aléatoire uniforme parmi les points de données. Ensuite, chaque centre de cluster suivant est choisi parmi les

points de données restants avec une probabilité proportionnelle à sa distance au carré du centre de cluster existant le plus proche du point.
Malheureusement, nous n'avons pas pu implémenter ce dernier en MapReduce à cause de quelques limitations de calcul qui utilisent la librairie numpy. Donc une diminution de performance peut affecter la partie d'initialisation des centroïdes.

1.1.3 Generateur

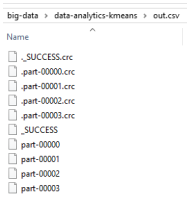
Afin d'évaluer l'algorithme KMeans implémenté, un générateur de données est mis en place.

On a mis en place 2 fichiers de python : *generator.py* et *generator-noise.py*
Le fichier *generator.py* permet de générer des points et de les sauvegarder dans un fichier.
Le deuxième fichier, *generator-noise.py*, permet de générer les points ainsi que des points bruits qui sont générés aléatoirement.

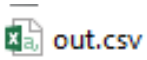
Afin de sauvegarder les données dans un fichier externe, on a implémenté 2 façons différentes :

- *saveAsTextFile()* : qui est une méthode déjà présente qui sauvegarde un rdd
- *write-into-csv()* qui est une méthode que nous avons créé qui permet d'écrire directement dans un fichier les valeurs du RDD

La méthode que nous avons créé *write-into-csv*, permet de générer un seul fichier mais n'est pas efficiente et ne profite pas des caractéristiques du RDD. C'est pour cela que la méthode saveAsTextFile est meilleur en terme d'efficacité. Cette dernière crée plusieurs fichiers.



(a) Méthode *saveAsTextFile*



(b) Méthode *write-into-csv*

FIGURE 2 – Fichier output de chaque méthode du générateur de données

En exécutant, le *generator.py* avec la commande suivantes :

```
$ spark-submit generator.py out 9 3 2 10
```

Les points sont générés, et en affichant les résultats dans le graphique on obtient :

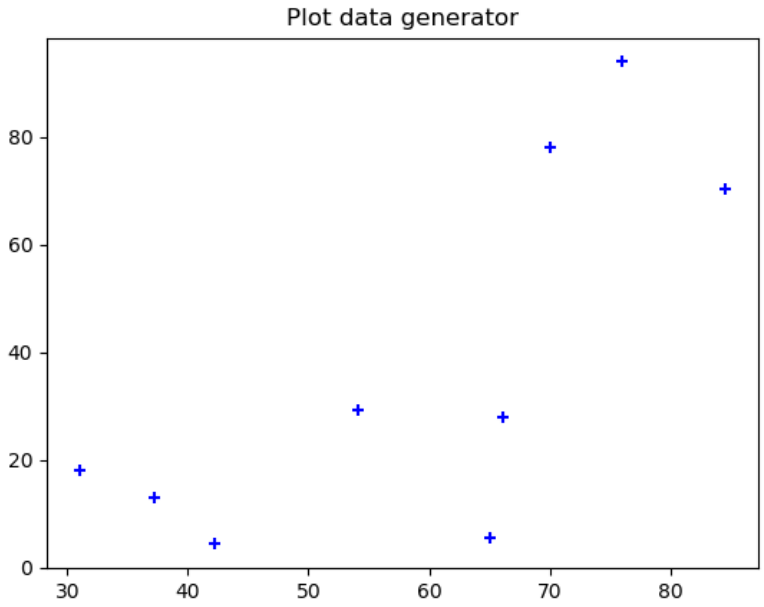


FIGURE 3 – Generateur des points

En executant, le *generator-noise.py* avec la commande suivante :

```
$ spark-submit generator_noise.py out 9 3 2 10
```

Le nombre de bruits est le double du nombre des points à générer. Les points sont générés, et en affichant les résultats dans le graphique on obtient :

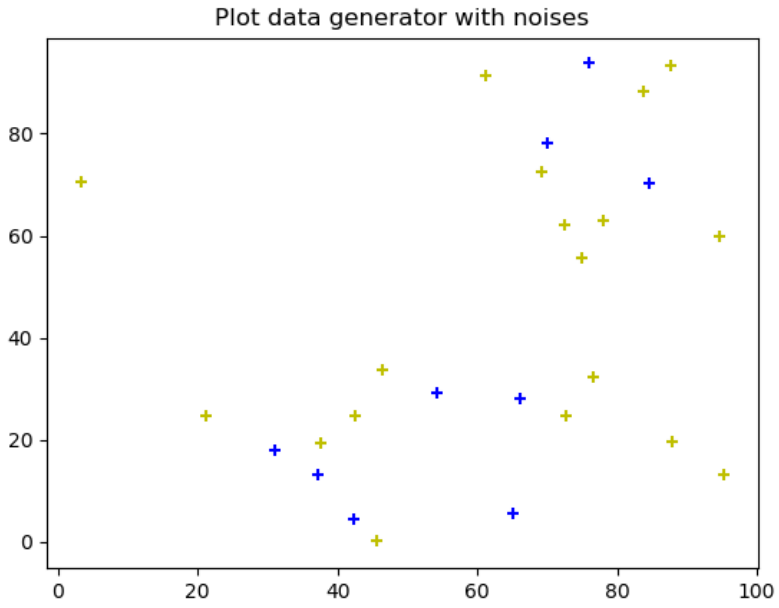


FIGURE 4 – Générateur des points avec des bruits

1.1.4 Plot

Afin de générer les graphiques, nous avons utilisé la librairie *matplotlib.pyplot*. Pour les fichiers KmeansPlusPlus, Kmeans nous avons créé 2 fichiers pour chacun, 1 sans plot et 1 avec plot.

Pour chaque itération, une image est sauvegardée dans le répertoire afin d'analyser l'évolution du résultat.

Les graphes générés sont de 2 dimensions, ce qui indique que si les points ont plus de 2 coordonnées ca ne seras pas très bien représenté graphiquement. Afin de voire des exemples d'exécution, veuillez voir la section Simulation.

1.2 Simulation

En exécutant, le *kmeans-plot.py* avec la commande suivantes :

```
$ spark-submit kmeans-plot.py data/iris-small.dat 3 10
```

Le graphe suivant représente les résultats de la première itération et en affichant les 2 premiers coordonnées : sepal width et sepal height

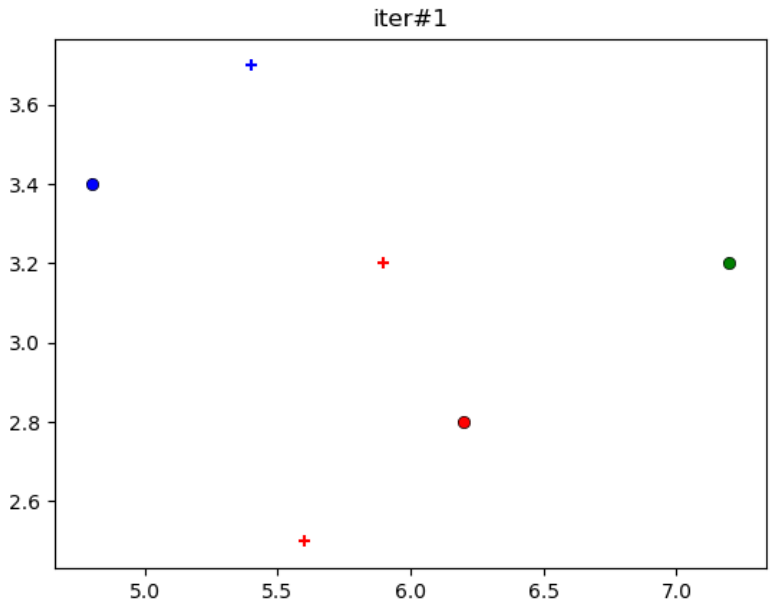


FIGURE 5 – Iteration 1

Pour trouver le resultat final on a eu besoin de 2 iterations avec une distance finale de 0.9711 d'ou le resultat final est le suivant :

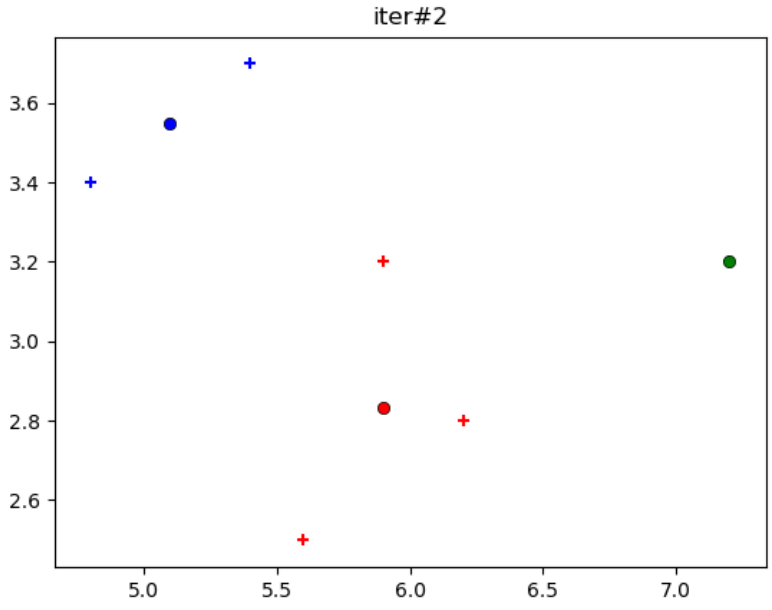


FIGURE 6 – Iteration finale

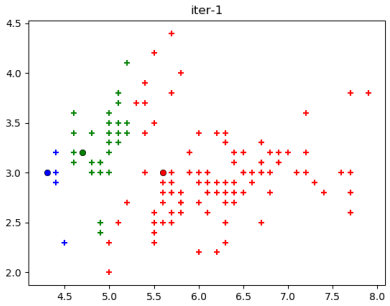
Et sur la console on obtient le résultat suivant :

```
c:\workspaces\big-data\data-analytics-kmeans>spark-submit kmeans.py data/iris_small.dat 3 10
18/04/29 17:40:30 WARN NativeCodeLoader: Unable to load native-hadoop library for your platfor
asses where applicable
iter #1: 1.3175589008869322
iter #2: 1.3795241880176898
Elapsed time: 0:00:24.838000
Number of iterations: 2
Final distance: 1.3795241880176898
```

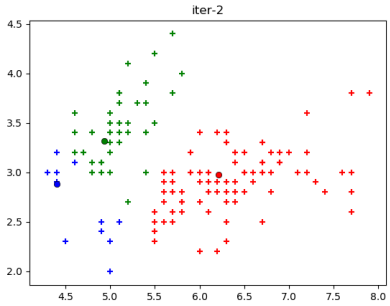
FIGURE 7 – Console

1.2.1 Resultat plot - Sepal

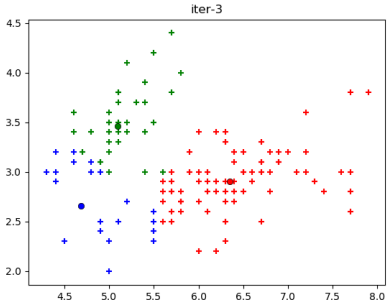
Ci-dessus vous pouvez voire la progression et les résultats de chaque itération sur l'execution de l'algorithm KMeans sur les données iris clustering avec seulement les 2 coordonnées de sepal : sepal width en cm et sepal height en



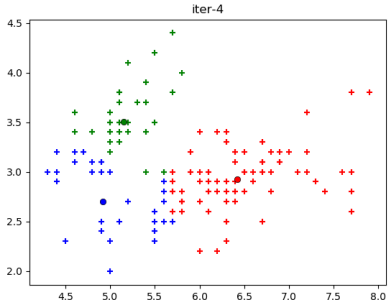
(a) Iteration 1



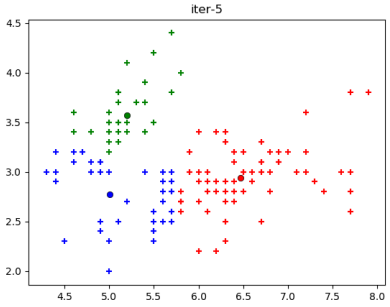
(b) Iteration 2



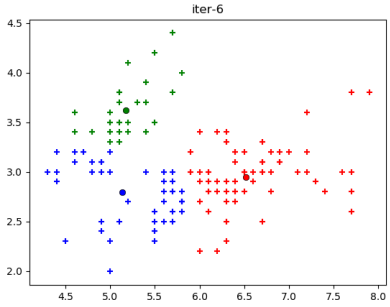
(a) Iteration 3



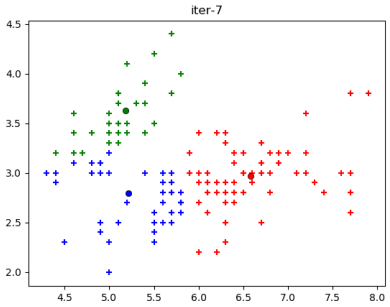
(b) Iteration 4



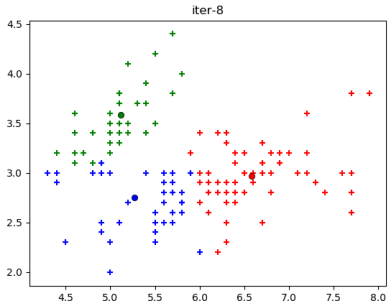
(a) Iteration 5



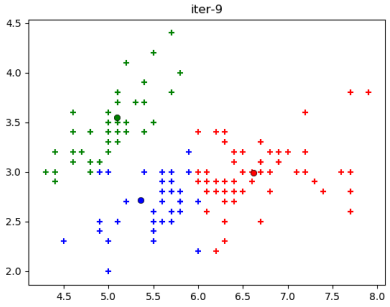
(b) Iteration 6



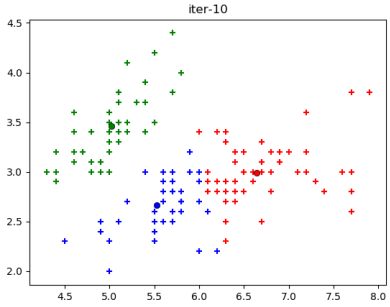
(a) Iteration 7



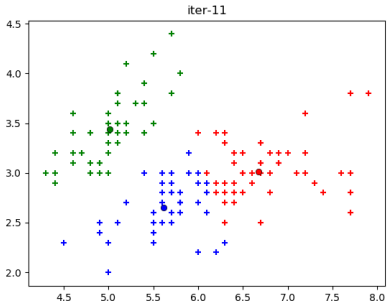
(b) Iteration 8



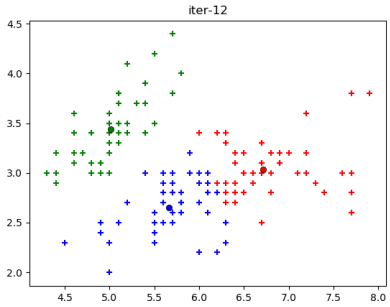
(a) Iteration 9



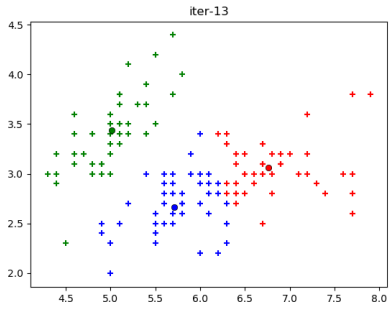
(b) Iteration 10



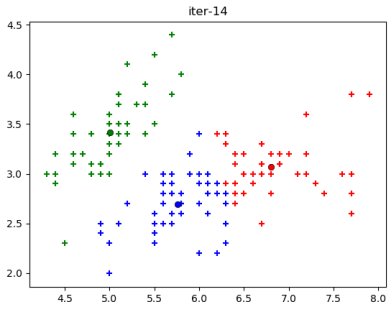
(a) Iteration 11



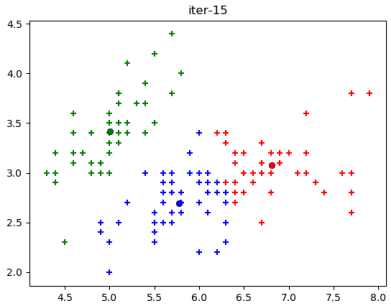
(b) Iteration 12



(a) Iteration 13



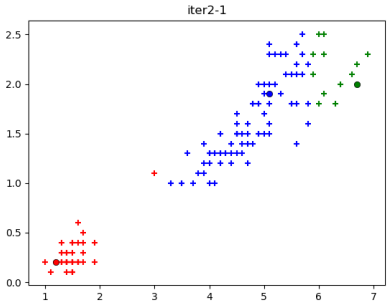
(b) Iteration 14



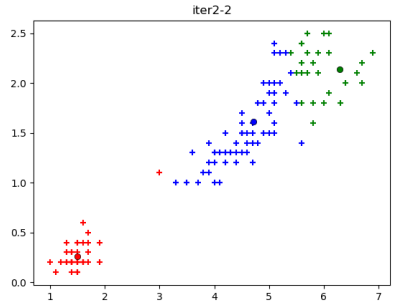
(a) Iteration 15

1.2.2 Resultat plot - Petal

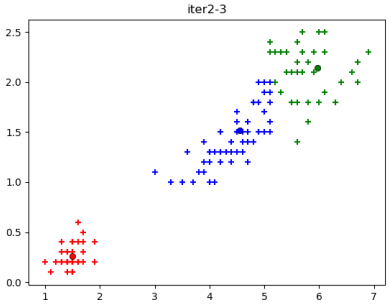
Ci-dessus vous pouvez voire la progression et les résultats de chaque itération sur l'execution de l'algorithm KMeans sur les données iris clustering avec seulement les 2 coordonnées de sepal : petal width en cm et petal height en



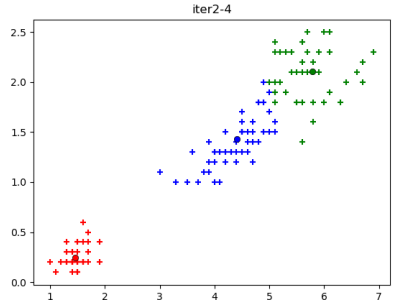
(a) Iteration 1



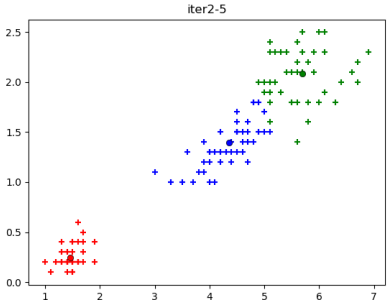
(b) Iteration 2



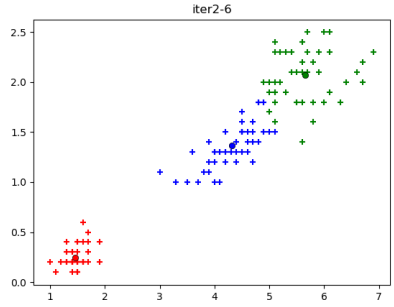
(a) Iteration 3



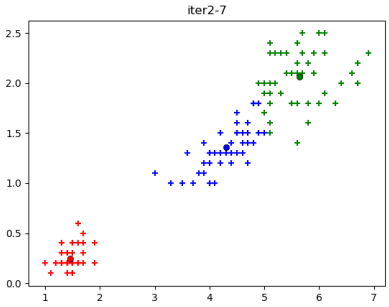
(b) Iteration 4



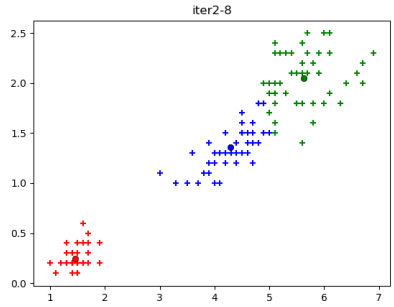
(a) Iteration 5



(b) Iteration 6



(a) Iteration 7



(b) Iteration 8

2 Analyse

2.1 Distance intra-cluster

En exécutant le programme, nous avons bien remarqué que la distance intra-cluster diminue progressivement jusqu'à atteindre un minimum durant la derniere iteration. On aperçoit parfois une légère augmentation suivi d'une diminution notable.

2.2 Convergence

L'algorithme s'arrête dès que les centroïdes gardent leurs valeurs durant deux itérations consécutives.

K-means avec une initialisation aléatoire des centres converge en 7 iterations, tandis que k-means++ réussi à converger en 6 iterations.

2.3 Tableau de performance

TABLE 1 – 100 executions

Algorithme	Temps d'exécution	Iterations	Distance finale
KMeans - implémenté	10.25 s	6.75	2.41
KMeans ++- implémenté	9.5 s	6.35	2.41
KMeans	10.25	6.75	2.41

3 Application

Références

[1] T. Cazenave. *Monte-Carlo Kakuro*

[2] Wikipedia page on *Kakuro*. [Online].
Available : `https://en.wikipedia.org/wiki/Kakuro`

[3] Wikipedia page on *CSP* . [Online].
Available : `https://en.wikipedia.org/wiki/Constraint_satisfaction_problem`

[4] Wikipedia page on *Backtracking algorithm*. [Online].
Available : `https://en.wikipedia.org/wiki/Look-ahead_(backtracking)`