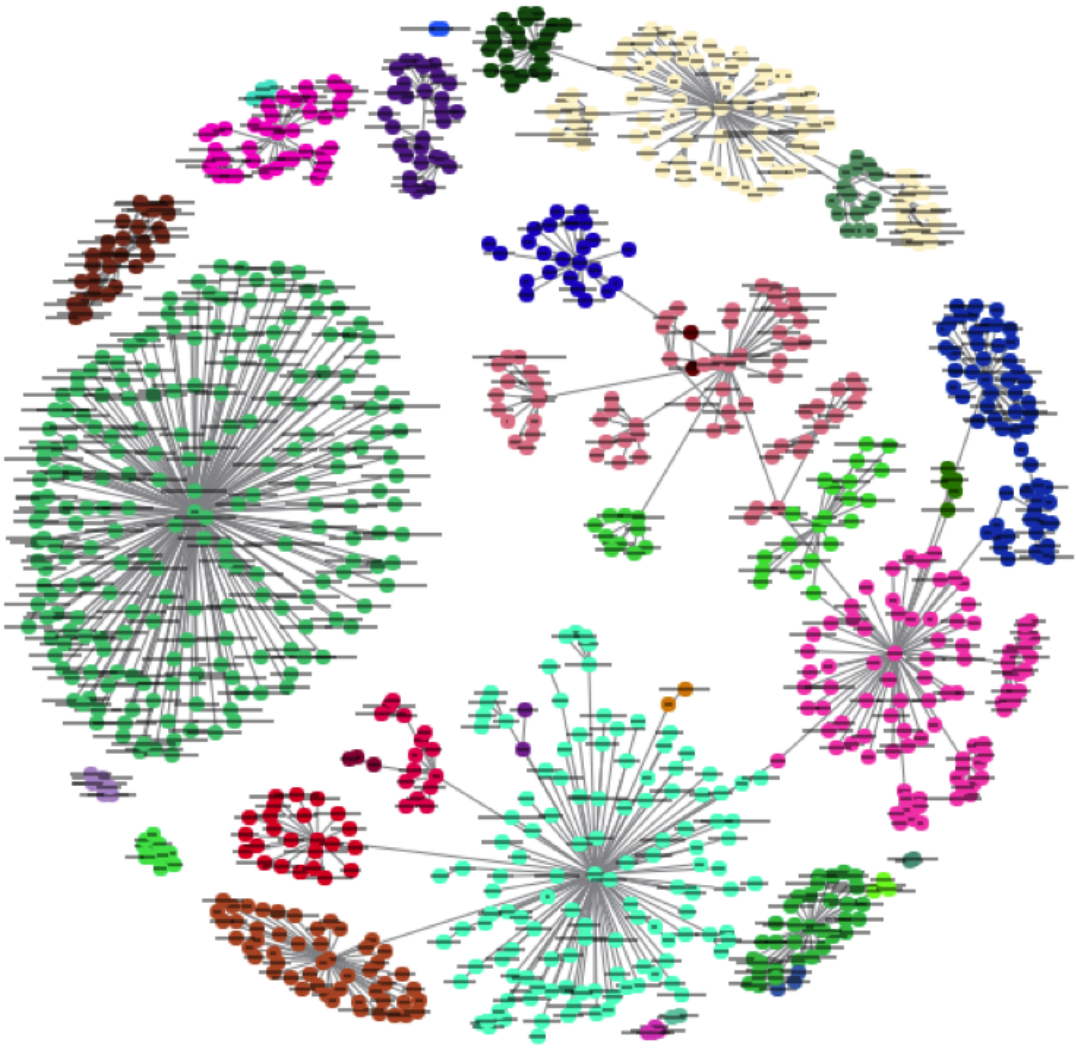
RAPPORT PROJET BIG DATA

K-Means in Spark



AZOULAY Mikael, DANET ROBIN, Komi Bi-Ayéfo ATSOU

Janvier 2020
M2 MIAGE Informatique décisionnelle, Formation classique
Université Paris Dauphine - PSL

Sommaire

Introduction

Ce projet consiste à implémenter l'algorithme K-Means de clustering dans un contexte big data avec Spark. Il s'agissait de faire des expérimentations afin d'identifier et implémenter des possibilités d'optimisation sur un code de base fourni par l'enseignant. Le code fourni est en Python et il fallait faire les améliorations en Python puis en Scala afin de faire des comparaisons et analyses de performance.

I- Optimisation de K-Means

Nous avons décidé d'ajouter 2 optimisations à l'algorithme K-means de base. La première au niveau de l'initialisation, la seconde au niveau de la parallélisation.

1- Optimisation #1 : une meilleure initialisation, K-Means++

Une amélioration importante de l'algorithme des K-Means a été proposée dans un article paru en 2006 par David Arthur et Sergei Vassilvitskii. Dans cet article ils introduisent l'algorithme K-Means++ qui optimise l'initialisation de l'algorithme. Dans ce cas on cherche à initialiser le K-Means avec des centroïdes distants les uns des autres. Ainsi on est moins susceptible de tomber dans un minimum local et on converge plus rapidement. Pour se faire, à la place du choix traditionnel où les centroïdes sont choisis de façon totalement aléatoire, ici des calculs supplémentaires sont à effectuer. Seulement ces calculs valent le coup puisqu'ils réduisent drastiquement le nombre de fois que l'on doit lancer le simple K-Means pour atteindre une solution optimale.
Voici le « K-Means++ initialization algorithm » énoncé Par David Arthur et Sergei Vassilvitskii :

> We propose a specific way of choosing centers for the `k-means` algorithm. In particular, let $D(x)$ denote the shortest distance from a data point to the closest center we have already chosen. Then, we define the following algorithm, which we call `k-means++`.
>
> 1a. Take one center c_1, chosen uniformly at random from $\mathcal{X}$.
> 1b. Take a new center c_i, choosing $x \in \mathcal{X}$ with probability $\frac{D(x)^2}{\sum_{x \in \mathcal{X}} D(x)^2}$.
> 1c. Repeat Step 1b. until we have taken k centers altogether.
> 2-4. Proceed as with the standard `k-means` algorithm.
>
> We call the weighting used in Step 1b simply "D^2 weighting".

Source : http://ilpubs.stanford.edu:8090/778/1/2006-13.pdf

En bref le principe est le suivant : choisir un premier centroïde de manière aléatoire uniforme parmi les data points, ensuite chaque centroïde supplémentaire est choisi parmi les points de données restants avec une probabilité en proportion de sa distance au carré du centre de cluster existant le plus proche du point.

2- Optimisation #1 : implémentation

Le défi pour nous est désormais d'implémenter cette optimisation tout en respectant les contraintes imposées par PySpark. Pour se faire nous implémentons la méthode suivante :

KmeansOpti_init(rdd, K, RUNS)

Cette méthode prend en entrée un RDD *"rdd"*, un nombre de clusters *"K"* ainsi qu'un nombre d'exécutions *"RUNS"* (nous y reviendrons plus bas).
En retour la méthode renvoie une matrice *"centroides"* de dimensions RUNS x K x A où A est le nombre d'attributs de notre dataset (dans le cas d'Iris, A=4). Cette matrice contient les K points nécessaires pour initialiser le K-Means.

Détail de son fonctionnement :

a. Initialisation

La variable 'centroides' est ce que nous voulons retourner. La valeur 'dist' dans la méthode update_dist() est la distance la plus proche de chaque data point aux points sélectionnés dans l'ensemble initial, où **dist [i]** est la distance la plus proche des points dans le i-ème ensemble initial. On collecte au préalable les vecteurs de caractéristiques de tous les points de données, cela nous sera utile dans la boucle. On sélectionne au hasard le premier point pour chaque exécution de K-Means ++, c'est-à-dire qu'on sélectionne au hasard RUNS points que l'on ajoute à la variable 'centroides'.

b. La boucle For

A chaque itération, on sélectionne un point pour chaque ensemble de points initiaux (donc RUNS points au total) de la façon suivante.
Pour chaque data point x, soit D_i (x) la distance entre x et le centre le plus proche qui a déjà été ajouté au i-ème ensemble. On choisit le nouveau data point de données à ajouter au i-ème ensemble en utilisant les probabilités énoncées dans K-Means++ à l'aide de la méthode choice(). Les probabilités sont proportionnelles à la distance des points à ceux déjà sélectionnés, autrement dit on choisit des points espacés les uns des autres.

Si on résume, à chacune de nos K itérations où K= nombre de clusters :

Pour chacune de nos RUNS exécutions voulues :

- Mettre à jour les distances
- Calculer la somme de D_i (x) ^ 2
- Normaliser chaque distance pour obtenir les probabilités
- Sélectionner un nouveau point selon ces probabilités
- Ajouter ce point à nos centroïdes

3- Optimisation #2 : parallélisation des exécutions

Une seconde façon d'optimiser les résultats de K-Means est de le lancer plusieurs fois (avec des initialisation différentes) et de retenir le cas où il a convergé vers la meilleure solution et idéalement la solution optimale du problème.
Dans le simple K-Means si l'on veut le lancer plusieurs fois les exécutions se font les unes à la suite des autres, ce qui surtout dans un contexte Big Data ne tire pas partie de la parallélisation des tâches qu'offre Spark. L'enjeu de cette seconde optimisation est donc de pouvoir lancer plusieurs exécutions de K-Means en parallèle et de retenir la plus optimale.

4- Optimisation #2 : implémentation

On introduit donc la variable entière RUNS qui détermine le nombre de K-Means que l'on souhaite exécuter en parallèle et la méthode suivante qui permet ces exécutions :

KmeansOpti(data, nb_clusters, max_steps, RUNS, min_switch=0, init="kmeans++")

Cette méthode prend en entrée nos données *data*, le nombre de clusters voulus *nb_clusters*, le nombre maximum d'itérations autorisées *max_steps* ainsi que le nombre d'exécutions parallèles souhaitées *RUNS*. La méthode retourne alors le résultat de l'exécution qui présente l'erreur la plus faible sous la forme d'un assignement points->cluster comme dans simple K-Means.

Détail de son fonctionnement :

a. Initialisation

Il faut initialiser en prenant K (= nb_clusters) points pour chacune des exécutions soit en tout (RUNS x K) points. Pour cela on utilise l'optimisation 1 qui nous fournit ces points intelligemment initialisés en appelant la fonction KmeansOpti_init() avec RUNS comme paramètre. On obtient ainsi le RDD centroides composé de RUNS x K centers. On génère une clé unique pour chacun de ces centers.

b. Convergence

Tant que max_steps n'est pas atteint (ou une fois que la moyenne des switch des RUNS exécutions <= min_switch) :

- On multiplie le volume des données manipulées par le nombre d'exécutions RUNS en générant une clé double unique.
- Ensuite on procède comme dans le simple K-Means mais avec RUNS x N éléments où N est le nombre de données de notre dataset (dans le cas d'Iris N = 150).

c. Fin

Une fois une des deux conditions d'arrêts remplies, on calcule l'erreur de nos RUNS assignments et on garde celui qui dispose de la plus faible erreur entre eux. On retourne enfin best_assignment, best_error, number_of_steps.

5. Autres possibilités d'optimisation

D'autres optimisations sont également possibles en termes de nombre de join (Shuffle and Sort) réalisés et aussi en termes de traitements faisant appels à des transferts de données importants d'une machine à l'autre. Ces optimisations sont possibles mais demandent d'avoir de façon plus ou moins permanente une architecture appropriée à disposition afin de faire les tests. Pour l'ensemble de nos tests nous avons utilisé le cluster de Dauphine, nos machines en local ou avec des environnements virtuels pour certains tests et Google Cloud Platform.

De plus dans notre cas, nous n'avions pas beaucoup de variables de broadcast.

II- Comparatifs sur les données Iris

Dans cette partie nous allons comparer les performances de notre K-Means optimisé face au K-Means simple. Nous les comparerons sur le dataset Iris et sur les aspects suivants : rapidité de convergence, qualité des résultats, vitesse d'exécutions.

1- Iris Dataset : présentation

Le dataset contient 3 classes de chacune 50 instances où chaque classe réfère à un type de plante Iris différent. Chaque plante est décrite selon quatre attributs.

Fichier : iris.dataset.txt

Poids : 4 698 octets

Instances : 150

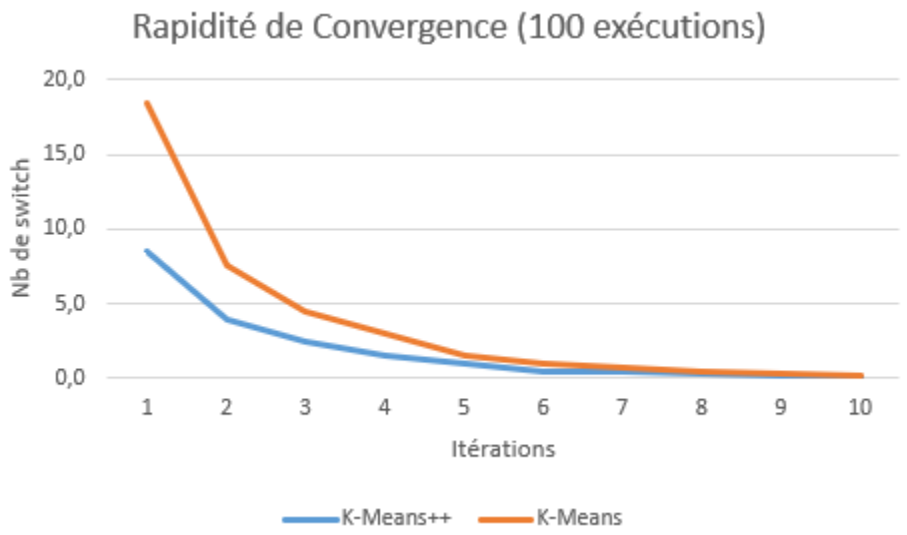
Attributs :
- sepal length in cm
- sepal width in cm
- petal length in cm
- petal width in cm

Classes :
- Iris-Setosa
- Iris-Versicolour
- Iris-Virginica

Nous sommes dans un problème de classification où l'on doit associer chaque plante à sa bonne classe. Nous allons utiliser K-Means pour séparer notre dataset en trois parties et observer les résultats (K-Means sera dans ce cas utilisé pour une classification supervisée même si habituellement on s'en sert de façon non-supervisé).

2- Rapidité de convergence

Comme nous l'avons vu plus haut, théoriquement l'initialisation K-Means++ est censée nous garantir une convergence plus rapide et plus sûre (moins de chance de tomber dans un minimum local), voyons ce qu'il en est.
Le premier aspect à comparer est la rapidité de convergence. On confronte donc l'évolution du nombre de switch moyen au fil des itérations pour le K-Means et le K-Means++ (rappel : le nombre de switch est le nombre d'éléments ayant changé de centroïdes entre deux itérations).



Résultats : L'implémentation confirme la théorie, K-Means++ converge en moyenne à 4.5 itérations contre 7.4 pour le simple K-Means.

3- Qualité des résultats

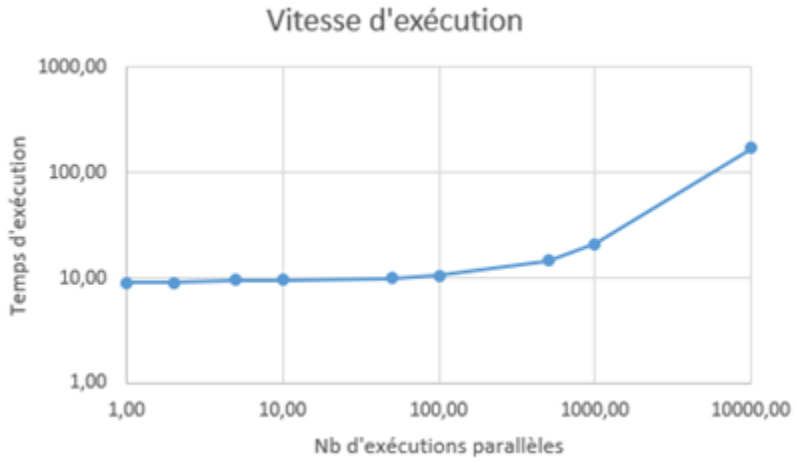
Une fois encore on compare l'exécution de 100 simple K-Means à celle de 100 K-Means++. Cette fois si on se penche sur leur score. On retrouve les résultats dans le tableau suivant :

		K-Means	K-Means++
Itération			
	Moyenne	7,4	4,5
	Min	2	1
	Max	16	13
Erreur			
	Moyenne	0,06762	0,06668
	Min	0,06570	0,06570
	Max	0,07573	0,07419

Résultats : Notre implémentation confirme encore la théorie, notre K-Means++ a une erreur moyenne plus faible ce qui est du au faite qu'il atteint plus souvent la solution optimale. La solution optimale en jaune est atteinte par les deux algorithmes au moins une fois lors des 100 exécutions.

4- Vitesse d'exécution

La deuxième optimisation nous permet de lancer plusieurs exécutions de K-Means simultanément en parallèle.



Résultat : Que l'on lance 1 ou 100 exécutions simultanément la vitesse est quasiment inchangée, la parallélisation est réussie.

5- Clustering non-supervisé : résultats

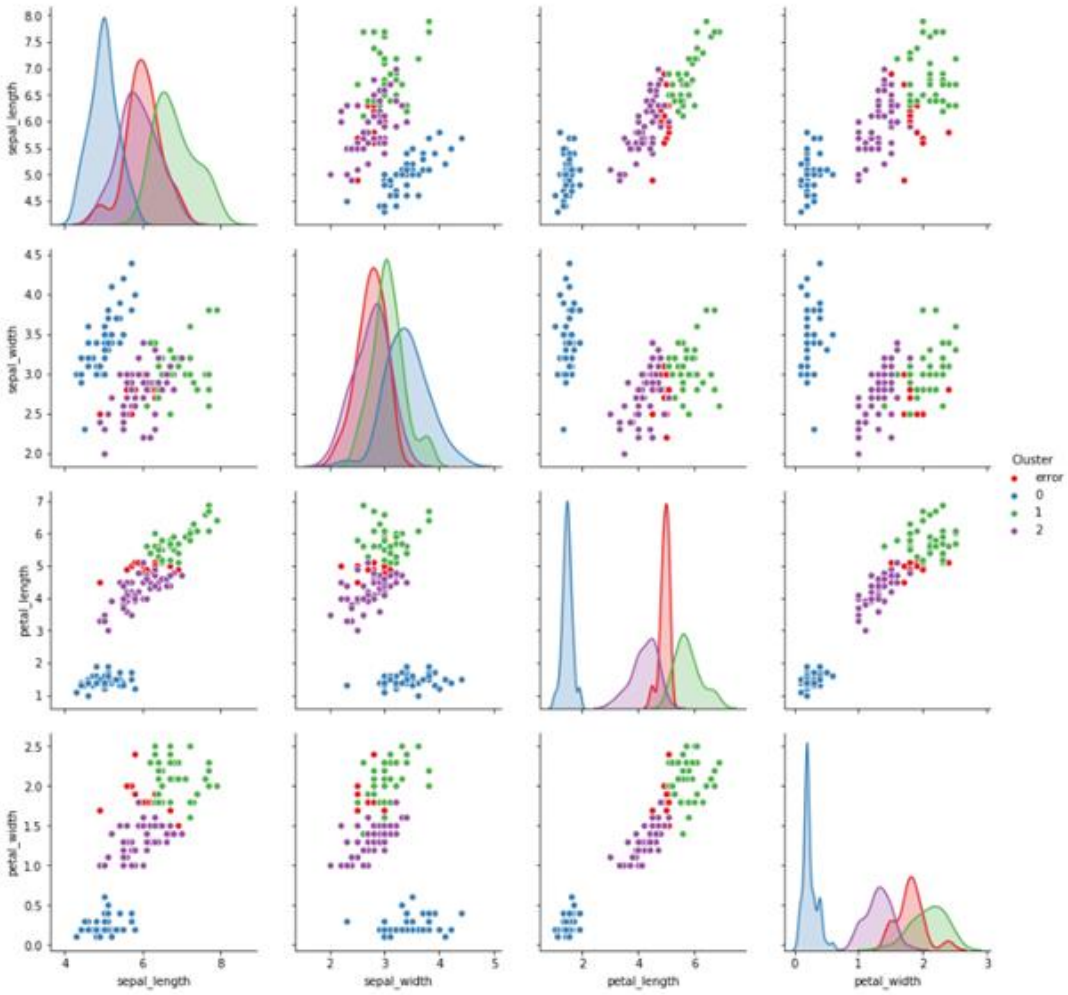
On choisit d'observer les résultats du meilleur K-Means optimisé à l'issue de 100 exécutions. Son erreur vaut 0.06570…

Idéalement notre clustering aurait séparé notre dataset en 3 parties de chacune 50 instances de même classe. Observons ce qu'il en est :

Cluster	f5_Iris-setosa	f5_Iris-versicolor	f5_Iris-virginica
0	50	0	0
1	0	2	36
2	0	48	14

On remarque que notre premier cluster est parfait il contient toute la classe Iris-Setosa ni plus ni moins. En revanche pour nos 2 autres clusters il y'a certaines instances qui n'ont pu être séparées correctement.

Tentons de comprendre pourquoi. On va visualiser les individus de nos clusters selon chacun des attributs :



Résultat : On remarque que certains points des clusters 1 et 2 ne sont pas séparables avec un clustering classique et que le cluster 0 (en bleu) contenant l'ensemble des 50 Iris-setosa est linéairement séparable des deux autres pour chaque attribut.

III- Segmentation client à l'aide de K-Means

1- Credit-Card dataset : présentation

Dans ce cas nous cherchons à établir une segmentation client qui aidera à établir des stratégies marketing. Le dataset résume le comportement d'utilisation d'environ 9000 titulaires de carte de crédit actifs au cours des 6 derniers mois. Le fichier est au niveau client avec 18 variables comportementales. Nous sommes dans un cas non-supervisé où nous cherchons à établir des profils clients. Le dataset a été trouvé sur Kaggle.

Fichier : credit.card.txt
Poids : 1,4 Mo
Instances : 8950
Attributs :

1. CUST_ID: Identification du titulaire de la carte de crédit
2. BALANCE: Montant du solde restant dans son compte pour effectuer des achats.
3. BALANCE_FREQUENCY: À quelle fréquence le solde est mis à jour, note entre 0 et 1 (1 = fréquemment mis à jour, 0 = pas fréquemment mis à jour)
4. PURCHASES: Montants des achats effectués à partir du compte
5. ONEOFF_PURCHASES: Montant maximum d'un achat effectué en une seule fois
6. INSTALLMENTS_PURCHASES: Montant des achats effectués en plusieurs fois
7. CASH_ADVANCE: Paiement anticipé donné par l'utilisateur
8. PURCHASES_FREQUENCY: Fréquence des achats effectués, score entre 0 et 1 (1 = fréquemment acheté, 0 = pas fréquemment acheté)
9. ONEOFFPURCHASESFREQUENCY: fréquence à laquelle les achats en une seule fois ont lieu(1 = fréquemment acheté, 0 = pas fréquemment acheté)
10. ACHATSINSTALLMENTSFREQUENCY: fréquence des achats en plusieurs fois (1 = fréquemment effectué, 0 = pas souvent fait)
11. CASHADVANCEFREQUENCY: À quelle fréquence l'argent comptant est-il payé d'avance
12. CASHADVANCETRX: Nombre de Transactions effectuées avec "Cash in Advanced"
13. PURCHASES_TRX: Nombre de transactions d'achat effectuées
14. CREDIT_LIMIT: Limite de carte de crédit pour l'utilisateur
15. PAYMENTS: Nombre de paiements effectués par l'utilisateur
16. MINIMUM_PAYMENTS: Montant minimum des paiements effectués par l'utilisateur
17. PRCFULLPAYMENT: Pourcentage du paiement intégral payé par l'utilisateur
18. TENURE : Durée du service de carte de crédit pour l'utilisateur

2- Data préparation

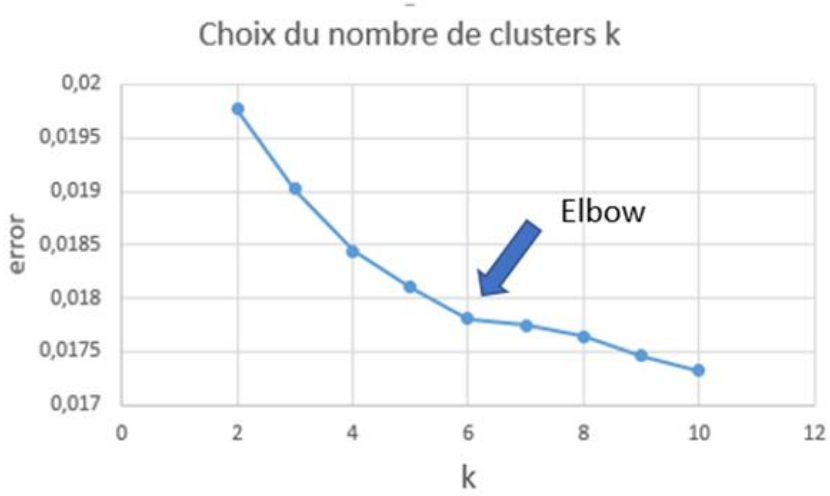
Par soucis de performances nous avons en amont effectué les traitements suivants sur le dataset en utilisant la librairie python, Pandas :

- Remplacement des valeurs manquantes par la moyenne
- Elimination des valeurs aberrantes (outliers)
- Normalisation des valeurs

Le dataset credit_card.csv originale devient credit_card.data.txt à sa sortie du script python credit_card_data_preprocess.

3- Choix du nombre de clusters k

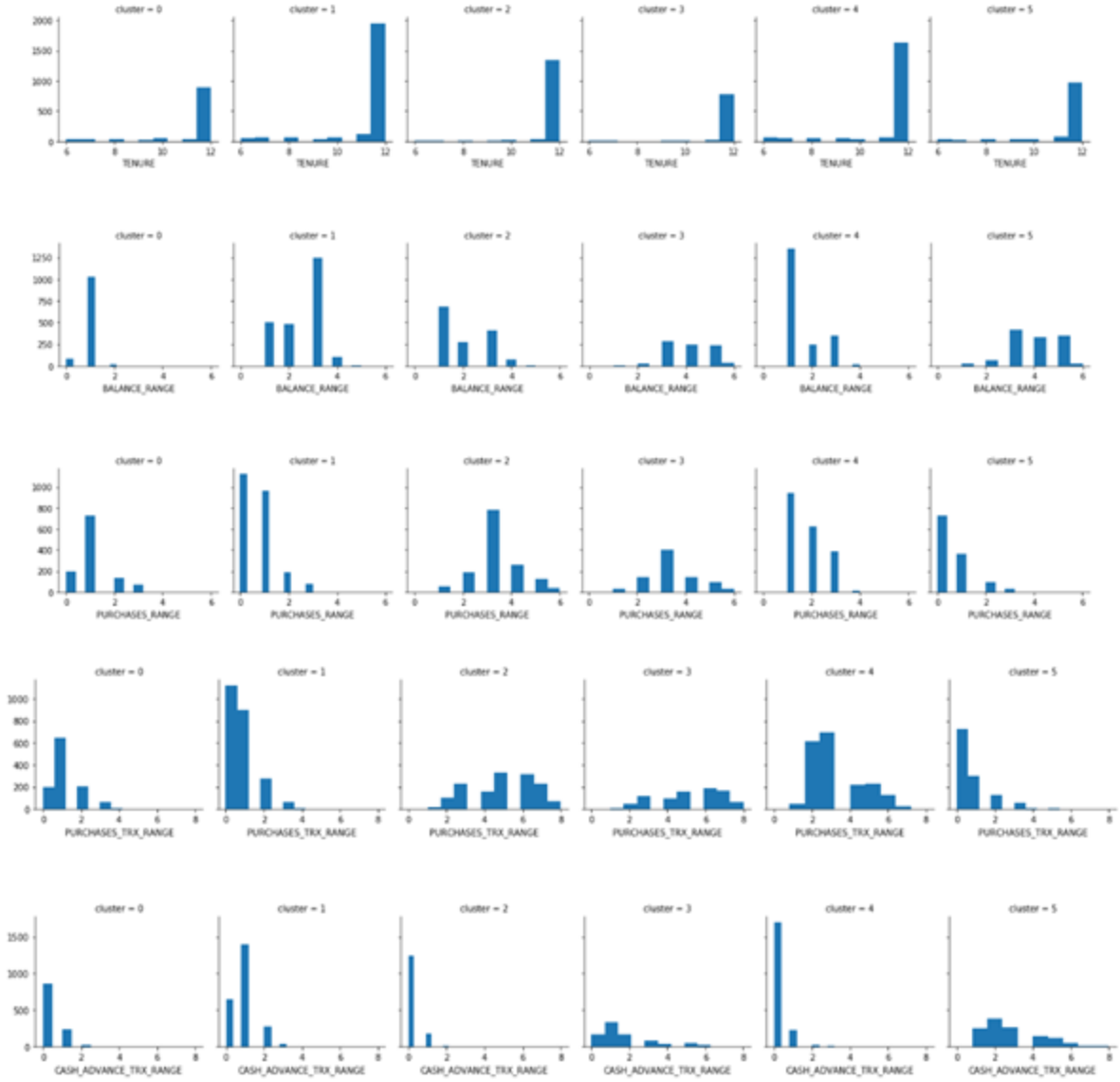
K-Means laisse à l'utilisateur le choix du nombre k de clusters. Il n'existe pas de procédé automatique pour trouver la meilleure valeur de k à utiliser, cependant une technique couramment utilisée consiste à calculer la moyenne des distances intra-cluster pour différentes valeurs de k comme on peut le voir ci-dessous. On utilisera alors la valeur de k à partir de laquelle l'ajout d'un cluster ne diminue plus beaucoup la moyenne des distances intra-cluster : on choisira donc la valeur k pour laquelle il y'a une inflexion de la courbe. Ce point s'appelle « Elbow ». Voir figure ci-dessous :

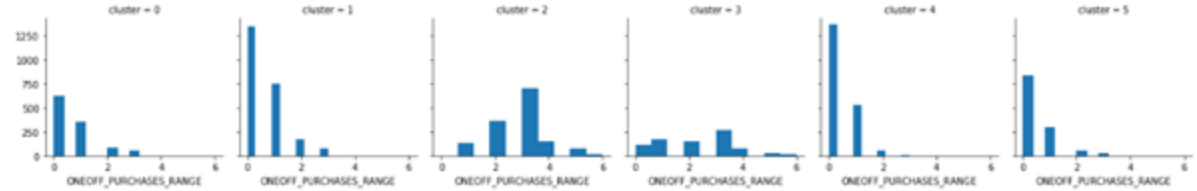


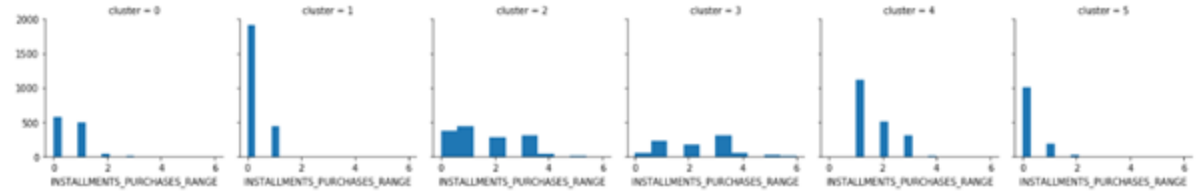
Résultat : Dans notre cas nous choisirons k = 6. Nous tenterons donc d'identifier 6 types de clients différents.

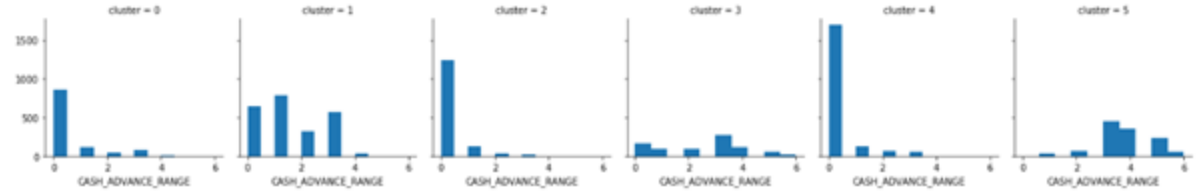
4- Visualisation des résultats

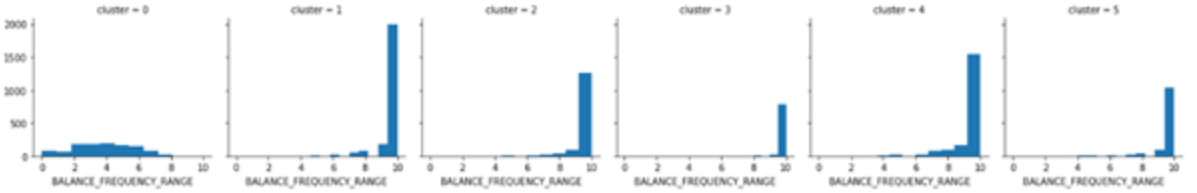
Ici on visualise la répartition des individus selon chaque attribut au sein de nos 6 clusters :

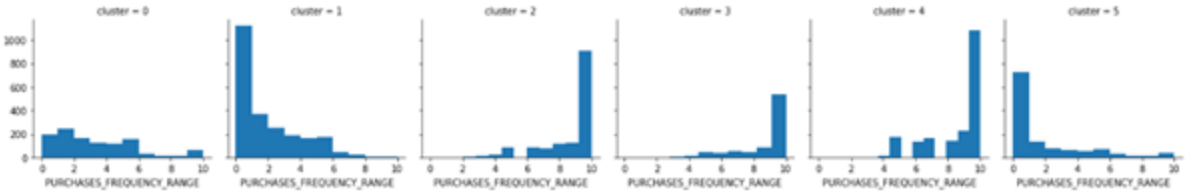

cluster = 0
cluster = 1
cluster = 2
cluster = 3
cluster = 4
cluster = 5
TENURE
BALANCE_RANGE
PURCHASES_RANGE
PURCHASES_TRX_RANGE
CASH_ADVANCE_TRX_RANGE

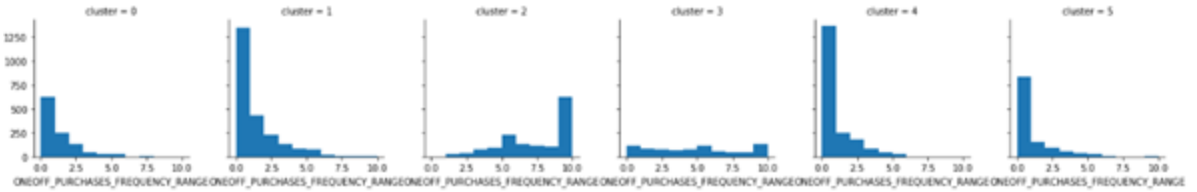

cluster = 0
cluster = 1
cluster = 2
cluster = 3
cluster = 4
cluster = 5
ONEOFF_PURCHASES_RANGE


cluster = 0
cluster = 1
cluster = 2
cluster = 3
cluster = 4
cluster = 5
INSTALLMENTS_PURCHASES_RANGE


cluster = 0
cluster = 1
cluster = 2
cluster = 3
cluster = 4
cluster = 5
CASH_ADVANCE_RANGE


cluster = 0
cluster = 1
cluster = 2
cluster = 3
cluster = 4
cluster = 5
BALANCE_FREQUENCY_RANGE


cluster = 0
cluster = 1
cluster = 2
cluster = 3
cluster = 4
cluster = 5
PURCHASES_FREQUENCY_RANGE


cluster = 0
cluster = 1
cluster = 2
cluster = 3
cluster = 4
cluster = 5
ONEOFF_PURCHASES_FREQUENCY_RANGE

5- Analyse des résultats

Cluster 0 : Personnes ayant une limite de crédit moyenne à élevée et effectuant tous types d'achats.

Cluster 1 : Personnes avec des paiements dus qui paient d'avance en espèces.

Cluster 2 : Personnes qui dépensent le moins d'argent, qui ont des limites de crédit moyennes à élevées et qui achètent principalement en plusieurs versements

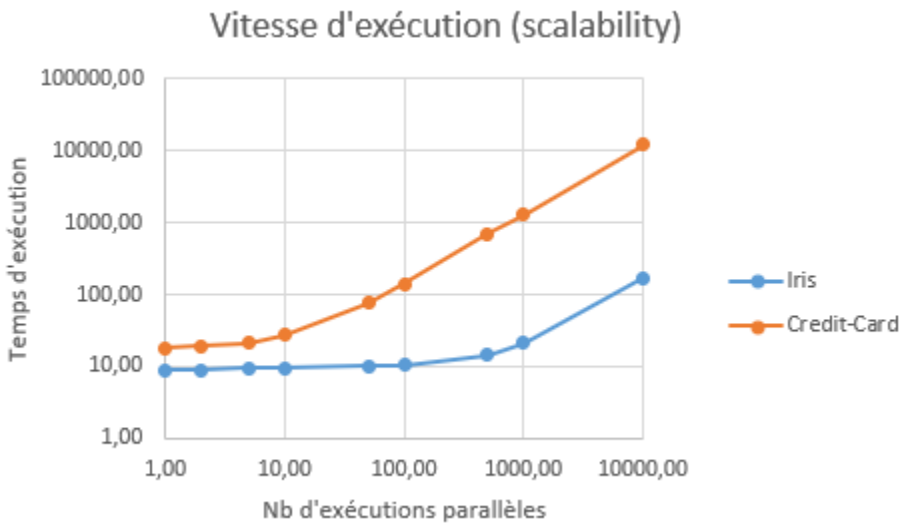
Cluster 3 : Personnes avec une limite de crédit élevée qui paient à l'avance

Cluster 4 : Personnes avec des dépenses élevées avec une limite de crédit élevée et qui font des achats coûteux

Cluster 5 : Personnes qui ne dépensent pas beaucoup d'argent et qui ont une limite de crédit moyenne à élevée

6- Scalabilité

Reprenons le tableau des vitesses d'exécution pour le Iris dataset et comparons les à celles obtenues lors d'exécutions sur le credit-card dataset.
On rappelle que le data des cartes de crédit est 280 fois plus volumineux que celui des Iris.



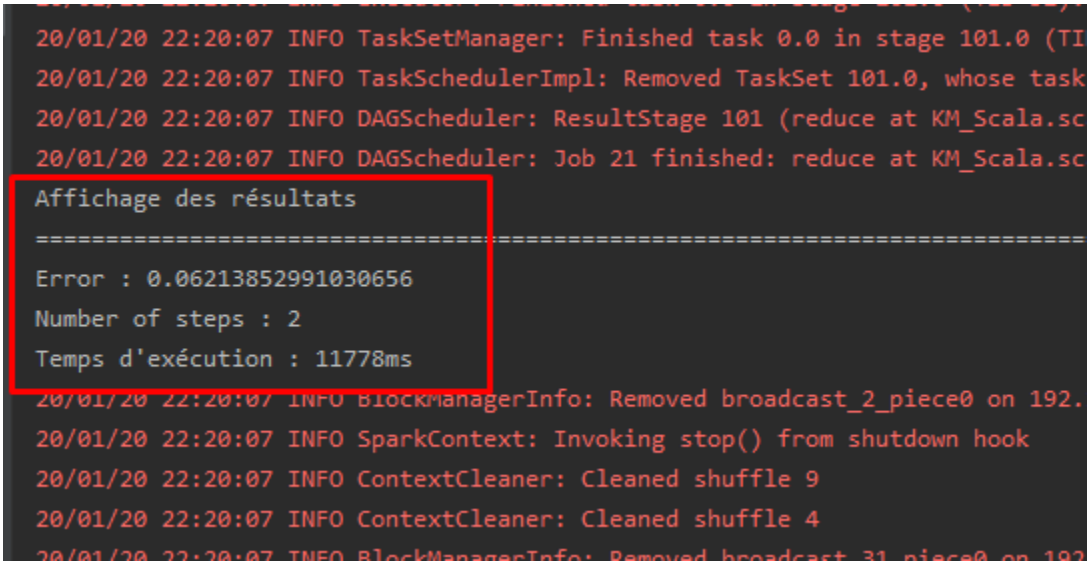
IV- Implémentation en Scala et comparaisons

1. Implémentation en Scala

Afin de comparer nos temps de traitement ainsi que la scalabilité de notre code, nous avons fait une implémentation en Scala comme recommandé par l'enseignant. La partie qui a été implémentée concerne le code de base mais les gains en termes de performance et de temps d'exécution sont considérables. Nous analysons dans cette partie certains constats que nous avons pu faire en ce sens.

2. Analyses et comparaisons

Le constat le plus important que nous avons noté concerne la rapidité d'exécution. Le code Python de base a été réimplémenté en Scala sans les optimisations mais nous remarquons déjà des gains importants en termes de temps d'exécution. Nous avons fait les tests simplement sur une machine Windows 10 avec PySpark et Scala 2.12 et Spark 2.4.1 et nous avons remarqué que le code Python qui a l'air de tourner sans s'arrêter, s'exécute sur Scala en moins de 12s, avec une erreur raisonnable, et converge après 2 itérations comme le montre la capture ci-dessous :



Ce petit test ne suffit pas à comparer les performances de Python et Scala mais il constitue un bon indicateur permettant d'approfondir d'avantage les expérimentations.

Sur Internet certains ont tenté de comparer les 2 langages en ce qui concerne l'utilisation avec Spark. Comme le mentionne un article[1] que nous avons lus, plusieurs critères permettent de faire une comparaison entre les deux langages, mais en ce qui concerne les performances, Scala s'avère plus performant surtout sur des architectures disposant de plusieurs cœurs. Par contre pour des architectures hautement parallèles avec beaucoup de machines les performances de Scala se réduisent pour se rapprocher de ceux de Python.

[1] https://www.dezyre.com/article/scala-vs-python-for-apache-spark/213