Dauphine | PSL
UNIVERSITÉ PARIS

Optimisation de l'algorithme de K-means en Spark

Olivier RANDAVEL
Adèle THORNER
Valentin VU VAN

28 mars 2020

Table des matières

1 Introduction **2**

2 Analyse de la solution existante **3**
2.1 L'algorithme K-means . . . 3
2.2 Description de la solution existante . . . 3

3 Exploration d'améliorations **6**
3.1 Optimisations liées au code . . . 6
3.1.1 Join des trois centroïdes par point . . . 6
3.1.2 Retirer le calcul de la racine dans la distance . . . 7
3.1.3 Éviter le groupByKey pour calculer le minimum . . . 7
3.2 Optimisation de l'algorithme des k-means . . . 8
3.2.1 Initialisation des centroïdes et k-means++ . . . 8
3.2.2 Utilisation de la distance de Manhattan . . . 9
3.2.3 Exploration aléatoire des features . . . 10
3.2.4 Condition d'arrêt . . . 11
3.2.5 Création d'une métrique pour évaluer la précision du résultat . . . 12

4 Passage à l'échelle et analyse des résultats **13**
4.1 Jeu de données - Classer des voitures par type . . . 13
4.2 Protocole d'expérimentation . . . 13
4.3 Résultats . . . 14

5 Implémentation avec des dataframes **14**
5.1 Fonction principale . . . 15
5.2 df_get_nb_switch() . . . 16
5.3 Résultats . . . 16

6 Conclusion **17**

7 Annexes **18**
7.1 RDD . . . 18
7.1.1 Fonctions . . . 18
7.1.2 Algorithme initial . . . 20
7.1.3 Algorithme optimisé . . . 22
7.1.4 Main . . . 24
7.2 Dataframe . . . 26
7.2.1 Fonctions . . . 26
7.2.2 Algorithme . . . 28
7.2.3 Main . . . 29
7.3 Jeu de test pour les dataframes . . . 30
7.4 Tableau des résultats . . . 31

1 Introduction

Les algorithmes de machine learning nous proposent des solutions d'apprentissage efficaces. En utilisant un grand volume de données, les algorithmes deviennent d'autant plus performants. Cependant, si les résultats théoriques s'améliorent avec la quantité de données, la rapidité de calcul se voit ralentie. En effet, on ne peut plus imaginer réaliser nos calculs de la même manière, les temps d'exécution seraient trop longs. C'est pour cela que l'on fait appel aux technologies de Big Data et du calcul distribué. Hadoop nous propose des solutions pour mettre en place des solutions stockées et exécutées de manière distribuée, partagée sur plusieurs machines. Spark est un framework permettant de créer et gérer des applications distribuées inspiré par l'algorithme MapReduce d'Hadoop. Ce framework est plus performant que les outils proposés par Hadoop mais nécessite d'être connecté à un système de gestion de fichiers distribués (celui d'Hadoop par exemple).

L'objectif de ce projet est de réaliser une implémentation efficace de l'algorithme de clustering des k-means, en utilisant la technologie Spark. Un code de base nous a été fourni et il nous faut l'analyser afin de proposer une solution améliorée qui met en avant les avantages du calcul distribué. Afin de mener à bien ce projet, nous avons analysé ensemble le code fourni, ligne par ligne et mis en oeuvre plusieurs optimisations, tout en gardant un historique de nos modifications, pour comparer l'évolution de nos fonctionnalités.

Au cours de ce rapport, nous analyserons dans un premier temps la solution existante. Ensuite, nous présenterons l'intégralité de nos optimisations en distinguant les améliorations du code et les optimisations de l'algorithme des k-means. Ensuite, nous décrirons l'ensemble des résultats que nous avons obtenu, en termes de temps d'exécution, avec le jeu de données fourni et un jeu plus volumineux. Depuis les résultats précédents, nous présenterons le passage des Resilient Distributed Datasets aux Dataframes et les résultats de performance obtenus.

2 Analyse de la solution existante

2.1 L'algorithme K-means

L'algorithme des K-moyennes (ou K-means en anglais) est un algorithme d'apprentissage non supervisé qui regroupe n observations x_i en K clusters. L'algorithme consiste à minimiser la distance de chaque point au centre (ou centroïde) de son cluster, de manière itérative. La fonction à minimiser est donc :

$$J(\mu, z) = \sum_{i=1}^{n} \sum_{j=1}^{K} z_j^i ||x_i - \mu_j||^2$$

avec μ_j le centre du cluster j, et z^i le vecteur d'affectation de l'observation x_i : $z_j^i = 1$ si l'observation x_i appartient au cluster j, 0 sinon.

L'algorithme se déroule de la manière suivante. A l'initialisation, on choisit K centroïdes de manière aléatoire parmi les n observations. A chaque itération, on calcule la distance de chaque point aux K centroïdes et on affecte chacun de ces points au cluster dont il est le plus proche. On met ensuite à jour les centroïdes en calculant les moyennes de chaque cluster. L'algorithme s'arrête lorsque les clusters n'ont pas changé d'une itération à une autre.

2.2 Description de la solution existante

L'algorithme proposé initialement est défini par la fonction *simpleKmeans*. Cette fonction prend en entrée la donnée préalablement indexée et le nombre de labels. Cette solution a pour but de séparer la donnée en un nombre de groupes défini par le paramètre d'entrée *nb_clusters*.

L'algorithme s'exécute ainsi :

Dans un premier temps, les centroïdes sont définis de façon aléatoire. Les centroïdes sont sélectionnés parmi les points donnés et sont tous différents. Enfin, ils sont numérotés pour pouvoir associer chaque observation à un centroïde. Nous obtenons donc un RDD constitué d'un id et des coordonnées du centroïde.

```
1      def simpleKmeans(data, nb_clusters):
2          centroides =  sc.parallelize(data.takeSample('withoutReplacment',nb_clusters))\
3                          .zipWithIndex()\
4                          .map(lambda x: (x[1],x[0][1][:-1]))
5          # (0, [4.4, 3.0, 1.3, 0.2])
```

Une fois les centroïdes initialisés, la boucle principale de l'algorithme commence. Voici les différentes étapes. Tout d'abord, la donnée est associée à chaque centroïde à l'aide d'un produit cartésien. La donnée est donc répétée autant de fois que le nombre de centroïdes. Ensuite, un calcul de distance est effectué

dans le but de fournir la distance entre chaque donnée et un centroïde. Puis, une opération *groupBy* permet de relier l'id d'une donnée à la distance la séparant des trois centroïdes. Enfin, il ne reste plus qu'à choisir la distance minimale et à assigner cette distance à la donnée.

```
    def simpleKmeans(data, nb_clusters):
        while not clusteringDone:
            joined = data.cartesian(centroides)
            # [((0, [5.1, 3.5, 1.4, 0.2, 'Iris-setosa']), (0, [6.0, 3.0, 4.8, 1.8])),
            #((1, [4.9, 3.0, 1.4, 0.2, 'Iris-setosa']), (0, [6.0, 3.0, 4.8, 1.8])), ...]

            dist = joined.map(lambda x: (x[0][0],(x[1][0],
            computeDistance(x[0][1][:-1], x[1][1]))))
            # (0, (0, 0.866025403784438))

            dist_list = dist.groupByKey().mapValues(list)
             # (0, [(0, 0.866025403784438), (1, 3.7), (2, 0.5385164807134504)])

             min_dist = dist_list.mapValues(closestCluster)
             # (0, (2, 0.5385164807134504))

            assignment = min_dist.join(data)
            # (0, ((2, 0.5385164807134504), [5.1, 3.5, 1.4, 0.2, 'Iris-setosa']))
```

Le code au-dessus utilise deux fonctions. *computeDistance* permet de calculer la distance euclidienne entre deux coordonnées. Dans notre cas, cette fonction est utilisée pour calculer la distance entre les coordonnées d'un point donné et un centroïde.

$$\sqrt{\sum_{i,j=1}^{n} (x_i - y_j)^2}$$

La seconde fonction est *closestcluster*. Elle prend en entrée l'id de la donnée en question et les distances calculées aux trois centroïdes. Cette fonction retourne la distance minimale entre le point et les trois centroïdes.

```
    def computeDistance(x,y):
        return sqrt(sum([(a - b)**2 for a,b in zip(x,y)]))

    def closestCluster(dist_list):
        cluster = dist_list[0][0]
        min_dist = dist_list[0][1]
        for elem in dist_list:
             if elem[1] < min_dist:
                cluster = elem[0]
                min_dist = elem[1]
        return (cluster,min_dist)
```

Cette première étape terminée, vient alors le calcul des nouveaux centroïdes. Dans un premier temps, les coordonnées de chaque point sont rattachées à un numéro de centroïde. Puis, un comptage est effectué dans le but de comparer la population liée à chaque centroïde. Enfin, les nouveaux centroïdes sont calculés en prenant la moyenne de chaque variable de toutes les données liées au centroïde en question.

```
1       clusters = assignment.map(lambda x: (x[1][0][0], x[1][1][:-1]))
2       # (2, [5.1, 3.5, 1.4, 0.2])
3
4       count = clusters.map(lambda x: (x[0],1)).reduceByKey(lambda x,y: x+y)
5       # count = (1 , 100),(2 , 200), (3, 50)   # (numero de cluster, nb d'element)
6
7       somme = clusters.reduceByKey(sumList)
8       # (1,2000), (2, 6000) , (3, 1500)     # (numero de cluster, somme des elements )
9
10      centroidesCluster = somme.join(count).map(lambda x : (x[0],moyenneList(x[1][0],x
        [1][1])))
11      # (0, 4.1, 2.5, 2.4, 0.8])   # (numero de cluster, average)
```

On retrouve deux fonctions utilisées dans le code ci-dessus. La première est *sumList*. Elle permet de sommer deux à deux les éléments de deux listes.

$$x = [x_1, x_2]\ ,\ \ y = [y_1, y_2]$$
$$z = [x_1 + y_1, x_2 + y_2]$$

La seconde fonction *moyenneList* retourne la moyenne des éléments d'une liste.

```
1   def sumList(x,y):
2     return [x[i]+y[i] for i in range(len(x))]
3
4   def moyenneList(x,n):
5     return [x[i]/n for i in range(len(x))]
```

La dernière étape permet d'évaluer le changement d'état et vérifie si l'algorithme doit s'arrêter. A partir de la deuxième boucle, il est possible de comparer les répartitions précédentes et actuelles. Si on constate un nombre de changements nul ou que le nombre de boucles est de 100, alors l'algorithme s'arrête et renvoie l'erreur de classification ainsi que la donnée classée. Dans le cas contraire, l'algorithme effectue une nouvelle boucle en se basant sur les nouveaux centroïdes. L'erreur ainsi calculée est retournée par l'algorithme. La formule de l'erreur suit cette formulation : prendre la racine carrée de la somme des distances entre les centroïdes et les points, divisée par le nombre de points.

$$erreur = \sqrt{\frac{\sum_i^n (dist_i)}{n}}$$

```
    if number_of_steps > 0:
        switch = prev_assignment.join(min_dist)\
                                .filter(lambda x: x[1][0][0] != x[1][1][0])\
                                .count()
      else:
        switch = 150
      if switch == 0 or number_of_steps == 100:
        clusteringDone = True
        error = sqrt(min_dist.map(lambda x: x[1][1])
            .reduce(lambda x,y: x + y))/nb_elem.value
      else:
        centroides = centroidesCluster
        prev_assignment = min_dist
```

3 Exploration d'améliorations

En explorant différentes pistes d'amélioration de la solution proposée, nous avons mis en place un certain nombre d'optimisations, de deux types différents. Les premières améliorations sont liées à de l'optimisation du code, notamment afin d'optimiser les requêtes Spark. Les secondes améliorations mettent en avant l'optimisation de l'algorithme des K-moyennes pour obtenir de meilleurs temps d'exécution. Nous avons réalisé l'ensemble de notre implémentation sur la plateforme Databricks, afin d'utiliser des clusters distants.

3.1 Optimisations liées au code

3.1.1 Join des trois centroïdes par point

Solution existante L'algorithme des k-means nécessite le calcul, à chaque itération, des distances de chaque point aux différents centroïdes. Dans la solution proposée, on crée un RDD contenant un point et un cluster à l'aide d'un produit cartésien. De cette manière, le RDD obtenu est redondant dans le sens où chaque point est présent sur 3 lignes.

```
joined = data.cartesian(centroides)
```

Notre amélioration Nous avons donc choisi de regrouper ces trois lignes en une seule afin de proposer un RDD contenant un point et une liste des trois centroïdes. Le RDD *joined* devient alors :

```
joined = join_centroide_data(data,3)
```

avec

```
def join_centroide_data(data_indexed, nb_clusters):
    centroides = centroides_input(data_indexed)
    centroides = centroides.collect()
    joined = data_indexed.map(lambda x : (x,centroides))
  return joined
```

3.1.2 Retirer le calcul de la racine dans la distance

Nous avons fait le choix de nous séparer du calcul de la racine carrée de la formule la distance euclidienne, car notre utilisation de la distance est simplement comparative. La fonction racine étant croissante, si $\sqrt{a} > \sqrt{b}$, alors $a > b$. Cela permet d'effectuer un calcul de moins lors du calcul de chaque distance, qui est une opération répétée de nombreuses fois lors du déroulé de l'algorithme. Nous avons ainsi réécrit la fonction de distance de la manière suivante :

```
def computeDistance_improve(x,y):
  return sum([(a - b)**2 for a,b in zip(x,y)])
```

3.1.3 Éviter le groupByKey pour calculer le minimum

Solution existante A chaque itération de l'algorithme, il est nécessaire de calculer la distance de chaque point aux centroïdes et de comparer ces trois distances afin de choisir la distance minimale. La solution proposée était la suivante :

```
joined = data.cartesian(centroides)
dist = joined.map(lambda x: (x[0][0],(x[1][0], computeDistance(x[0][1][:-1], x[1][1]))))
dist_list = dist.groupByKey().mapValues(list)
min_dist = dist_list.mapValues(closestCluster)
assignment = min_dist.join(data)
```

Le code ci-dessus crée d'abord un RDD avec un point et un cluster comme nous l'avons vu précédemment. Ensuite, pour chaque couple de points il calcule la distance à l'aide de la fonction *compute_distance* et réalise un groupByKey pour associer à chaque point les trois distances aux centroïdes. Enfin, on associe à chaque point le centroïde le plus proche et on effectue un join avec les données de départ pour récupérer les coordonnées du point.

Notre amélioration L'idée de notre amélioration était de se débarrasser du groupByKey que l'on sait si coûteux. Nous utilisons l'amélioration vue dans la section 3.1.1 qui consistait à regrouper les trois centroïdes pour un point. Ainsi, pour calculer les distances, on parcourt une fois le RDD et pour chaque ligne on effectue 3 calculs de distance. Nous voyons une vraie optimisation au moment du calcul du minimum. En effet, au lieu de réaliser un groupByKey, il nous suffit de comparer les trois distances associées au point

et de choisir la plus petite. Notre fonction *min_* ci-dessous nous permet de combiner les calculs des distances et leur comparaison en une fonction. Ainsi, on limite les temps de lecture du RDD, il nous suffit d'accéder une fois à l'élément pour obtenir son cluster.

```
joined = join_centroide_data(data,3)
min_dist = min_(joined)
assignment = min_dist.join(data)
```

avec

```
def min_(joined) :
  dist_list = joined.map(lambda x : (x[0][0],
    ((x[1][0][0], computeDistance_improve(x[0][1][:-1], x[1][0][1])),
    (x[1][1][0], computeDistance_improve(x[0][1][:-1], x[1][1][1])),
    (x[1][2][0], computeDistance_improve(x[0][1][:-1], x[1][2][1]))
  )))
  min_dist = dist_list.mapValues(closestCluster)
  return min_dist
```

3.2 Optimisation de l'algorithme des k-means

3.2.1 Initialisation des centroïdes et k-means++

L'idée de ne pas choisir aléatoirement les centroïdes initiaux nous est venue après avoir fait tourner la version "basique" du code. En effet, après différents essais, nous avons remarqué que les temps d'exécution pouvaient s'avérer beaucoup plus longs selon les choix des premiers centroïdes, et parfois mener à des erreurs de type "Executor heartbeat timed out after 146532 ms". L'une des explications à cette erreur est le choix initial des centroïdes, qui sont choisis aléatoirement (cf partie 2.2) et il arrive donc que ceux-ci soient trop proches, entraînant un grand nombre d'itérations supplémentaires. Nous avons alors pensé à une autre méthode pour éviter ce genre de "départs ratés", dont l'idée est très simple : prendre le point A le plus grand possible, le point B le plus éloigné de ce point A et enfin le point C étant au milieu de A et B. Cette méthode d'initialisation existe en fait dans un algorithme nommé k-means ++, qui est, comme son nom l'indique, une optimisation de k-means.

Remarque : la méthode, telle que nous l'avons implémentée, n'est applicable que dans notre cas car le nombre de clusters à former est inférieur ou égal à 3. Pour ce faire, nous avons défini les fonctions suivantes :

Création des centroïdes

```
def centroides_input(data) :
  centroide1 = data.map(lambda x : (0, (sum(x[1][:-1]),x[1][:-1])))
    .map(lambda x : (x[0],x[1][1])).max(lambda x: x[1][0])
  y = centroide1[1]
  centroide2 = data.map(lambda x : (1,(computeDistance_improve(y,x[1][:-1]),x[1][:-1] )))
    .map(lambda x : (x[0],x[1][1])).min(lambda x: x[1])
```

```
    y2 = centroide2[1]
    point = [x + y for x, y in zip(y, y2)]
    point = [x / 2 for x in point]
    centroide = sc.parallelize([centroide1,centroide2,(2,point)])
    return centroide
```

On commence par sommer la valeur de chaque feature pour chaque point de notre jeu de données : on obtient une valeur (la somme de chacun des 4 features pour chaque point). On en extrait alors le maximum, que l'on stocke dans la variable *y*.
Ensuite, grâce à la fonction $computeDistance\_improve$ présentée plus haut, on identifie *y2*, le point le plus éloigné du point *y*. Pour finir, on calcule le point situé au milieu de *y* et *y2*. Ces trois points seront les centroïdes initiaux de notre algorithme.

Remarque : le troisième centroïde n'est pas forcément un point appartenant à notre jeu de données. Ce n'est pas un problème car au fur et à mesure des itérations, le centroïde de chaque cluster évolue et n'est généralement plus équivalent à un point du jeu de données.

Limites : Si le jeu de données n'est pas propice à une bonne initialisation des centroïdes malgré cette méthode, le fait de relancer l'algorithme ne pourra pas permettre d'avoir un meilleur résultat, car les clusters initiaux seront toujours les mêmes. Pour cela, nous avons inséré dans notre fonction principale un paramètre de valeur $True$ ou $False$, qui permet d'activer ou non cette initialisation truquée.

3.2.2 Utilisation de la distance de Manhattan

Nous avons vu dans la partie précédente que le calcul de la distance euclidienne pouvait être optimisé en supprimant le passage à la racine, ce dernier n'étant pas nécessaire pour obtenir la distance minimale. Il existe un grand nombre de distances, plus ou moins complexes à implémenter. Nous cherchons à améliorer le temps d'exécution de l'algorithme. Nous avons choisi de tester une distance différente, pour observer son impact sur la vitesse de convergence de l'algorithme.

La distance de Manhattan

$$D(x,y) = \sum_{i=0}^{n} |x_i - y_i|$$

Pour pouvoir utiliser cette distance, en la combinant avec les différentes améliorations (centroïdes regroupés, random features, etc), nous avons implémenté les fonctions suivantes, qui sont utilisées en substitutions des fonctions de départ qui utilisent la distance euclidienne.

```
# if the centroides are not joined in 1 row
def computeDistance_manhattan(x,y):
  return sum([(a - b) for a,b in zip(x,y)])

# use manhattan distance with random features and joined centroides
def compute_distance_manhattan_features(x,y,features):
  sum = 0
  for f in features:
    sum += (x[f]-y[f])
```

```
    return sum
#use manhattan distance with all features and joined centroides
def compute_distance_manhattan(x,y):
    return compute_distance_manhattan(x,y,[0,1,2,3])

# get min of manhattan distances with all features
def min_manhattan(joined) :
    # [(0, ((0, 0.30000000000000016), (1, 5.31224999411737), (2, 3.057776970284131)) ) )
    dist_list = joined.map(lambda x : (x[0][0],
        ((x[1][0][0], compute_distance_manhattan(x[0][1][:-1], x[1][0][1])),
        (x[1][1][0], compute_distance_manhattan(x[0][1][:-1], x[1][1][1])),
        (x[1][2][0], compute_distance_manhattan(x[0][1][:-1], x[1][2][1]))
    )))
    min_dist = dist_list.mapValues(closestCluster)
    return min_dist

# get min of manhattan distances with random features
def min_manhattan_selectedfeatures(joined,features) :
    dist_list = joined.map(lambda x : (x[0][0],
        ((x[1][0][0],compute_distance_manhattan_features(x[0][1][:-1], x[1][0][1],features)),
        (x[1][1][0],compute_distance_manhattan_features(x[0][1][:-1], x[1][1][1],features)),
        (x[1][2][0], compute_distance_manhattan_features(x[0][1][:-1], x[1][2][1],features))
    )))
    min_dist = dist_list.mapValues(closestCluster)
    return min_dist
```

3.2.3 Exploration aléatoire des features

A chaque itération, l'algorithme évalue la distance de chaque point aux différents centroïdes en calculant la distance euclidienne. Cette dernière dépend de l'ensemble des features des observations. Nous avons voulu mettre en place une solution permettant de réduire le temps d'exécution en évaluant seulement une distance partielle à chaque itération.

L'idée était la suivante. Pendant les premières itérations de l'algorithme, choisir aléatoirement 2 features parmi les 4 et calculer les distances de l'itération par rapport à ces deux caractéristiques uniquement. Ainsi, on effectue deux fois moins de calculs que si l'on prenait en compte toutes les itérations.

De cette idée ont découlé plusieurs problématiques. Tout d'abord, nous nous sommes rendu compte que l'algorithme ne convergerait pas toujours au sens des K-moyennes (aucun changement dans les affectations au cluster) en regardant seulement un échantillon des features. Il fallait donc déterminer une solution pour calculer la distance "complète", c'est-à-dire la distance qui prend en compte les 4 features, à la fin de l'algorithme. Nous avons souhaité prendre en paramètre un seuil, nommé dans le code ci-dessous "epsilon_change_nbfeatures", qui détermine lorsque l'on doit changer le nombre de features à prendre en compte (2, puis 3, puis 4). Ce seuil permet d'évaluer lorsque l'algorithme est en train de stagner. Nous avons établi que l'algorithme stagnait lorsque le nombre d'observations qui changeaient de cluster étaient en-dessous de ce seuil à deux itérations de suite. En effet, on considère alors que nous n'avons plus assez d'informations (de features) pour améliorer l'affectation.

Une autre problématique qui nous est venue à l'esprit était que l'algorithme risquait de partir dans la mauvaise direction en ne prenant en compte que certaines features. Pour remédier à cela, nous avons ajouté une option qui permet d'effectuer une itération de "contrôle" régulièrement. On peut alors préciser un entier qui détermine l'intervalle entre deux "contrôles". On appelle une itération de "contrôle" une itération qui prend en compte l'ensemble des features pour le calcul de la distance.

```
    # start by checking if we need to change the number of features
    if(number_of_steps > 0) and switch < epsilon_change_nbfeatures and switch_prev <
    epsilon_change_nbfeatures and nb_features_selected < 4:
            nb_features_selected += 1
    random.seed(number_of_steps)
    features = random.sample([0,1,2,3], k=nb_features_selected) #choose randomly the
    features to observe

    min_dist = min_selectedfeatures(joined,features) # specify the features selected
```

$features$ va ainsi prendre un nombre de valeurs aléatoires entre 0 et 3 selon la valeur de k. Il est important de changer les features sur lesquels sont faits les calculs à chaque itération pour ne pas partir dans une mauvaise direction. k commence avec une valeur de 2 et augmente lorsque l'algorithme commence à stagner. Cette méthode par paliers permet de converger plus vite vers une solution acceptable, et finit par effectuer ses calculs sur tous les features (lorsque $k = 4$) pour s'assurer de finir sur un résultat correct, c'est-à-dire arriver à convergence au sens de k-means.

Utilisation de random.seed() Afin de pouvoir comparer de manière pertinente les résultats obtenus sur plusieurs exécutions de l'algorithme, nous utilisons la fonction *random.seed()* qui retournera le même nombre si on l'appelle avec le même entier. Ici on choisit en paramètre le numéro de l'itération, les features choisis différeront d'une itération à une autre mais d'un run à un autre, on choisira les mêmes features à la même itération.

```
def min_selectedfeatures(joined,features) :
  dist_list = joined.map(lambda x : (x[0][0],
    ((x[1][0][0], compute_distance_selectedfeatures_nosqrrt(x[0][1][:-1], x[1][0][1],
    features)),
    (x[1][1][0], compute_distance_selectedfeatures_nosqrrt(x[0][1][:-1], x[1][1][1],
    features)),
    (x[1][2][0], compute_distance_selectedfeatures_nosqrrt(x[0][1][:-1], x[1][2][1],
    features))
  )))
  min_dist = dist_list.mapValues(closestCluster)
  return min_dist
```

3.2.4 Condition d'arrêt

La condition d'arrêt initialement implémentée se base sur deux aspects. Le premier concerne une limite sur le nombre de fois où les centroïdes sont calculés, cette limite est fixée à 100. La seconde permet

de stopper l'algorithme si aucune donnée n'a été labelisée différemment (l'état présent est identique au précédent).
Dans notre cas pour chaque itération, comme présenté précédemment, nous n'utilisons pas toujours toutes les caractéristiques pour constituer nos groupes. Donc, une nouvelle condition s'ajoute. Nous vérifions que toutes les variables explicatives disponibles ont été utilisées si aucune observation n'a changé d'affectation. Ces points sont présentés dans le code ci-dessous.

```
    if (switch == 0 and nb_features_selected == 4) or number_of_steps > 50:
          clusteringDone = True
          error_improve = erreur_improve(assignment)
          error = sqrt(min_dist.map(lambda x: x[1][1]).reduce(lambda x,y: x + y))/nb_elem
    .value
```

3.2.5 Création d'une métrique pour évaluer la précision du résultat

L'algorithme du k-means permet de retourner les données labelisées. Chaque donnée est alors associée à un chiffre entre 0 et le nombre de labels distincts. Nous avons identifié deux solutions pour évaluer la précision de l'algorithme. Le but consiste à rapprocher un chiffre d'un label.

La première solution se déroule ainsi. Pour chaque label, nous calculons la moyenne des caractéristiques, puis nous évaluons la distance euclidienne entre ces moyennes et chaque centroïde résultant de l'algorithme des k-means. Enfin, le label est associé au centroïde le plus proche. Cette première approche s'est avérée peu concluante. En effet, deux labels pouvaient être rapprochés d'un même centroïde. De plus, le choix de la moyenne comme moyen de comparaison est discutable, la médiane ou encore un calcul de quartile auraient aussi pu être envisagés.

Au vu de ces problèmes, une deuxième méthode a été implémentée. Celle-ci consiste à retourner l'erreur de classification optimale. Dans le cadre d'un jeu de données labelisées avec trois labels distincts, il suffit de faire 3! (3 factoriel) opérations (6 opérations) et de prendre la valeur pour laquelle le moins d'erreurs ont été effectuées. Cette méthode est présentée ci-dessous.

```
    def erreur_improve(assigment):
      list1 = [1,1,2,2,3,3]
      list2 = [2,3,3,1,2,1]
      list3 = [3,2,1,3,1,2]
      label = assigment.map(lambda x: x[1][1][4]).distinct().collect()
      percentage_error_list = []
      for a,b,c in zip(list1,list2,list3) :
        combinaison = [(label[0],a),(label[1],b),(label[2],c)]
        pourcentage_error = 0
        combianaison_assigment = assigment.map(lambda x: (x[1][1][4], x[1][0][0] )).
    collect()
        for i in range(len(combianaison_assigment)) :
            if combianaison_assigment[i] not in (combinaison) :
              pourcentage_error += 1
        pourcentage_error /=  len(combianaison_assigment)
```

```
15            percentage_error_list.append(pourcentage_error)
16        return min_list(percentage_error_list)
17
18        def min_list(array) :
19          min = array[0]
20          for i in range(1,len(array)) :
21            if min > array[i] :
22              min = array[i]
23        return min
```

4 Passage à l'échelle et analyse des résultats

Nous avons souhaité implémenter les différentes optimisations décrites précédemment et comparer les résultats obtenus avec les résultats donnés par la solution initiale. Pour ce faire, nous avons créé une fonction globale custom_kmeans qui prend en paramètres des booléens permettant de préciser les différentes optimisations à mettre en oeuvre. Cette fonction retourne les résultats de l'algorithme, c'est-à-dire les différentes erreurs calculées, le nombre d'itération effectuées et les centres des clusters.

A chaque appel du *main*, on précise combien de fois on veut exécuter les algorithmes, et on indique si on veut faire tourner l'algorithme initial, l'algorithme amélioré ou les deux. Le main retourne le temps d'exécution moyen de chacun de ces algorithmes ainsi que chacun de ces temps d'exécution. Cette fonction globale nous a permis de garder une trace des différentes optimisations et éventuellement de combiner des optimisations compatibles. Nous avons choisi de comparer les résultats en termes de temps d'exécution.

4.1 Jeu de données - Classer des voitures par type

Afin d'obtenir des résultats significatifs, nous avons choisi d'effectuer nos expériences sur un jeu de données plus conséquent que le jeu iris. Étant donné que l'algorithme initial et, par conséquent, l'algorithme amélioré, étaient tous deux adaptés à un problème à 4 features et 3 clusters, nous avons cherché un jeu de données du même format.

Nous avons trouvé un jeu de données de plus de 6000 lignes portant sur la classification de voitures en trois gammes (inférieure, luxe et supérieure), selon 4 caractéristiques (puissance maximale, consommation, émission de CO2 et poids du véhicule à vide). Ces données proviennent du site gouvernemental https ://www.data.gouv.fr/fr/datasets/emissions-de-co2-et-de-polluants-des-vehicules-commercialises-en-france/ .

4.2 Protocole d'expérimentation

Nous avons implémenté le protocole suivant pour obtenir des résultats comparables de nos différents algorithmes.

Définition des différentes améliorations Nous avons commencé par définir une liste de fonctions que l'on souhaitait comparer, c'est-à-dire une liste de combinaisons d'améliorations. Nous avons réalisé un tableau permettant de mettre en évidence les valeurs à donner aux paramètres de la fonction custom_kmeans, pour chaque expérience. Ce tableau est donné en annexe.

Pour que nos résultats soient significatifs, nous avons choisi de lancer chaque expérience 5 fois de suite, sur les datasets iris et voitures_types. Un tableau regroupe l'ensemble de ces résultats, pour chaque expérience et chaque dataset. Ce tableau est également ajouté en annexe de ce rapport.

4.3 Résultats

L'exécution de nos différentes expériences nous a permis de mettre en valeur plusieurs résultats.

Suppression du groupBy En comparant les temps d'exécution moyens, nous avons pu constater qu'une de nos fonctionnalités améliorait considérablement la performance de l'algorithme. Il s'agit de notre option *join_centroids_in_rdd* qui nous permet de représenter les trois centres pour une observation plutôt que d'exploser ces informations en trois lignes. Comme nous l'avons précisé dans la section 3.1.1, cette fonctionnalité nous permet d'éviter de réaliser une opération groupBy, que l'on sait très coûteuse. Les résultats nous prouvent cette affirmation. En effet, on constate que lorsque cette option n'est pas activée, les algorithmes n'ont pas réussi à converger. On obtient, sans cette amélioration, une erreur d'exécution due à un temps trop long.

Choix des centroïdes initiaux Cette deuxième fonctionnalité permet d'améliorer les performances du modèle, lorsqu'elle est combinée à la précédente. En effet, le fait de choisir les premiers centroïdes biaise le modèle et permet de converger plus rapidement.

Random features Contrairement à ce que l'on pensait lors de l'implémentation de cette fonctionnalité, *random_features* n'améliore pas la performance de l'algorithme. Au contraire, les temps d'exécution moyens observés sont même un peu ralentis. Ceci est dû au fait que la sélection d'un sous ensemble des features entraîne une convergence en plus d'itérations. De plus, cette fonctionnalité ne paraît pas très pertinente pour un jeu de données à seulement 4 features. La différence de temps de calcul de la distance avec 2 ou 4 features est négligeable. En revanche, il serait intéressant de tester cette option sur des jeux de données comportant un grand nombre de features.

Distances Les résultats nous montrent que l'utilisation d'une distance plutôt qu'une autre ne changeait pas les temps d'exécution.

5 Implémentation avec des dataframes

Jusqu'ici, nous avons uniquement manipulé nos données sous la forme de Resilient Distributed Datasets (RDD). Dans cette partie, nous présenterons notre migration vers l'utilisation de dataframes. Une dataframe est une structure de données qui s'apparente à un tableau. Cette structure de données a été pensée sur Spark pour pouvoir être manipulée de manière distribuée.

Comme décrit dans la partie 3 de ce rapport, nous avons mis en oeuvre un grand nombre d'optimisations du code initialement fourni. L'analyse des performances de ces différentes optimisations nous a permis de mettre en avant la meilleure combinaison de fonctionnalités, en termes de performance. Les différentes optimisations liées à la gestion propre des RDD ne peut se retranscrire avec des dataframes. En revanche, les modifications de l'algorithme en lui même peuvent être maintenues lors du passage en dataframes. Nous avons choisi de ne mettre en oeuvre que la meilleure combinaison d'optimisations liées à l'algorithme des k-means, lors du passage aux dataframes.

5.1 Fonction principale

```
    def df_kmeans(data,nb_steps_max):
      clusteringDone = False
      number_of_steps = 0
      current_error = float("inf")
      nb_elem = sc.broadcast(data.count())
      switch = -1
      # compute initial centroids
      centroide1,centroide2,centroide3 = centroide_input_df(data)
      data = data.drop("_c0").drop("_c1").drop("_c2").drop("_c3").drop('total')
      while(not clusteringDone and number_of_steps < nb_steps_max):
        min_dist_df = df_compute_min_distances(data,centroide1,centroide2,centroide3)
        #index, coordonnees, dist1, dist2, dist3, cluster
        new_centroids_df = df_update_centroids(min_dist_df) # cluster, avg_coords
        centroide1,centroide2,centroide3 = df_get_new_centroids(new_centroids_df)
        if(number_of_steps > 0):
          switch = df_get_nb_switch(min_dist_df_prev,min_dist_df)
        min_dist_df_prev = min_dist_df
        number_of_steps += 1

        if(switch == 0):
          clusteringDone = True
          labels = data.select("label").distinct().collect()
          error = df_compute_error(min_dist_df, labels)
        if number_of_steps %10 ==0 :
          print("number_of_steps :", number_of_steps)
      assignment = min_dist_df.drop("dist1").drop("dist2").drop("dist3")
      centroides = [centroide1,centroide2,centroide3]
      return (assignment, error, number_of_steps, centroides)
```

Le développement de cette fonction est évidemment fortement similaire à celui réalisé pour l'algorithme présenté dans la partie 3. Nous avons simplement pu profiter de certaines spécificités des dataframes. Il est par exemple inutile de copier la totalité des coordonées de chaque cluster pour chaque élément de la base de données (ce qui rajouterai en effet 12 colonnes, 4 colonnes pour chaque cluster). Ces informations sont stockées dans les variables $centroidei$.
Concernant la valeur de $switch$, elle est ici obtenue en effectuant une jointure entre les clusters de l'étape précédente et de l'étape suivante. On ne garde que les éléments ayant subi un changement dans le dataframe et on compte ce nombre d'éléments.

5.2 df_get_nb_switch()

```
def df_get_nb_switch(df_prev,df_suiv):
  join = df_prev.withColumnRenamed("cluster","cluster_prev")
    .join(df_suiv,"index")
    .filter(F.col("cluster_prev")!=F.col("cluster"))
  return join.count()
```

5.3 Résultats

Après avoir lancé la fonction principale exécutant le code avec les dataframes, les résultats obtenus ne se sont pas avérés satisfaisants. Nous obtenons une convergence systématique de l'algorithme mais les temps obtenus sont supérieurs à ceux obtenus avec l'utilisation des RDD. Deux explications logiques nous viennent en tête :

1. La présence d'un join pour le caclul du switch, qui est en effet lourde en terme de temps de calcul ;
2. Les centroïdes stockés dans des variables, nous ne savons pas comment le système gère cela de manière distribuée.

6 Conclusion

De nombreuses solutions efficaces existent déjà en Python pour l'algorithme des k-means. Malheureusement, lorsque notre jeu de données est trop conséquent, ces approches "classiques" ne sont plus adaptées. Il est alors nécessaire de faire appel au calcul distribué, et c'est dans ce cadre que se situe ce projet. Ces méthodes de calcul distribué impliquent une réflexion en amont plus poussée. Si certaines solutions nous paraissent plus intuitives, elles ne seront pas nécessairement performantes. Cette notion est parfaitement illustrée par le code fourni pour commencer le projet : logique mais peu efficace. Ce rapport résume les différentes idées que nous avons eues et implémentées pour répondre à ce problème.

Les résultats obtenus nous permettent de mettre en avant deux optimisations du code. La première consiste à éviter l'utilisation d'un groupBy. La deuxième permet de choisir les centroïdes initiaux pour amener plus rapidement à convergence (cette approche est inspirée de l'algorithme K-means++). De manière à mieux observer nos améliorations, il serait intéressant d'adapter puis d'appliquer notre code à des jeux de données comportant plus de features, et répartis dans plus de clusters. En particulier, notre exploration aléatoire des features (cf 3.2.3) apporterait probablement une meilleure performance en grande dimension.

7 Annexes

Le code peut être trouvé au lien suivant :

https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/3631309044632400/4360712146874161/8024126452304013/latest.html

7.1 RDD

7.1.1 Fonctions

```
from pyspark import SparkContext, SparkConf
from math import sqrt
import pyspark.sql.functions as f
import time
from time import gmtime, strftime
from datetime import datetime
import pandas as pd
import random

def seconds_between(d1, d2):
    d1 = datetime.strptime(d1, "%H:%M:%S")
    d2 = datetime.strptime(d2, "%H:%M:%S")
    return abs((d2 - d1).seconds)

def centroides_input(data) :
  centroide1 = data.map(lambda x : (0, (sum(x[1][:-1]),x[1][:-1]))).map(lambda x : (x[0],
    x[1][1])).max(lambda x: x[1][0])
  y = centroide1[1]
  centroide2 = data.map(lambda x : (1,(computeDistance_improve(y,x[1][:-1]),x[1][:-1] )))
    .map(lambda x : (x[0],x[1][1])).min(lambda x: x[1])
  y2 = centroide2[1]
  point = [x + y for x, y in zip(y, y2)]
  point = [x / 2 for x in point]
  centroide = sc.parallelize([centroide1,centroide2,(2,point)])
  return centroide

def join_centroide_data(data_indexed, nb_clusters):
  centroides = centroides_input(data_indexed)
  centroides = centroides.collect()
  joined = data_indexed.map(lambda x : (x,centroides))
  return joined


def computeDistance_improve(x,y):
  return sum([(a - b)**2 for a,b in zip(x,y)])

def min_(joined) :
  dist_list = joined.map(lambda x : (x[0][0],
    ((x[1][0][0], computeDistance_improve(x[0][1][:-1], x[1][0][1])),
    (x[1][1][0], computeDistance_improve(x[0][1][:-1], x[1][1][1])),
    (x[1][2][0], computeDistance_improve(x[0][1][:-1], x[1][2][1]))
  )))
```

```
    min_dist = dist_list.mapValues(closestCluster)
    return min_dist

  def min_list(array) :
    min = array[0]
    for i in range(1,len(array)) :
      if min > array[i] :
        min = array[i]
    return min

  def graphe_display(nb_runs,custom_algo_times) :
    d = {'times': pd.Series(range(1,nb_runs+1)), 'time': custom_algo_times}
    df = pd.DataFrame(data=d)
    display(df)

  def erreur_improve(assigment):
    list1 = [1,1,2,2,3,3]
    list2 = [2,3,3,1,2,1]
    list3 = [3,2,1,3,1,2]
    print("assigment :", assigment)
    label = assigment.map(lambda x: x[1][1][4]).distinct().collect()
    print("label :", label)
    percentage_error_list = []
    for a,b,c in zip(list1,list2,list3) :
      combinaison = [(label[0],a),(label[1],b),(label[2],c)]
      pourcentage_error = 0
      combianaison_assigment = assigment.map(lambda x: (x[1][1][4], x[1][0][0] )).collect()
      print("combianaison_assigment : ",combianaison_assigment)
      for i in range(len(combianaison_assigment)) :
          if combianaison_assigment[i] not in (combinaison) :
            pourcentage_error += 1
      pourcentage_error /=  len(combianaison_assigment)
      percentage_error_list.append(pourcentage_error)
    return min_list(percentage_error_list)

  def compute_distance_selectedfeatures_nosqrrt(x,y,features):
    sum = 0
    for f in features:
      sum += (x[f]-y[f])**2
    return sum

  def min_selectedfeatures(joined,features) :
    dist_list = joined.map(lambda x : (x[0][0],
      ((x[1][0][0], compute_distance_selectedfeatures_nosqrrt(x[0][1][:-1], x[1][0][1],
      features)),
      (x[1][1][0], compute_distance_selectedfeatures_nosqrrt(x[0][1][:-1], x[1][1][1],
      features)),
      (x[1][2][0], compute_distance_selectedfeatures_nosqrrt(x[0][1][:-1], x[1][2][1],
      features))
    )))
    min_dist = dist_list.mapValues(closestCluster)
    return min_dist

  # if the centroides are not joined in 1 row
  def computeDistance_manhattan(x,y):
    return sum([(a - b) for a,b in zip(x,y)])

  # use manhattan distance with random features and joined centroides
```

```
96  def compute_distance_manhattan_features(x,y,features):
97    sum = 0
98    for f in features:
99      sum += abs(x[f]-y[f])
100   return sum
101
102 #use manhattan distance with all features and joined centroides
103 def compute_distance_manhattan(x,y):
104   return compute_distance_manhattan(x,y,[0,1,2,3])
105
106 # get min of manhattan distances with all features
107 def min_manhattan(joined) :
108   dist_list = joined.map(lambda x : (x[0][0],
109     ((x[1][0][0], compute_distance_manhattan(x[0][1][:-1], x[1][0][1])),
110     (x[1][1][0], compute_distance_manhattan(x[0][1][:-1], x[1][1][1])),
111     (x[1][2][0], compute_distance_manhattan(x[0][1][:-1], x[1][2][1]))
112   )))
113   min_dist = dist_list.mapValues(closestCluster)
114   return min_dist
115
116 # get min of manhattan distances with random features
117 def min_manhattan_selectedfeatures(joined,features) :
118   dist_list = joined.map(lambda x : (x[0][0],
119     ((x[1][0][0], compute_distance_manhattan_features(x[0][1][:-1], x[1][0][1], features)
      ),
120     (x[1][1][0], compute_distance_manhattan_features(x[0][1][:-1], x[1][1][1], features))
      ,
121     (x[1][2][0], compute_distance_manhattan_features(x[0][1][:-1], x[1][2][1], features))
122   )))
123   min_dist = dist_list.mapValues(closestCluster)
124   return min_dist
125
126 def computeDistance(x,y):
127   return sqrt(sum([(a - b)**2 for a,b in zip(x,y)]))
128
129 def closestCluster(dist_list):
130   cluster = dist_list[0][0]
131   min_dist = dist_list[0][1]
132   for elem in dist_list:
133       if elem[1] < min_dist:
134           cluster = elem[0]
135           min_dist = elem[1]
136   return (cluster,min_dist)
137
138 def sumList(x,y):
139   return [x[i]+y[i] for i in range(len(x))]
140
141 def moyenneList(x,n):
142   return [x[i]/n for i in range(len(x))]
```

7.1.2 Algorithme initial

```
1  def simpleKmeans(data, nb_clusters):
```

```
    clusteringDone = False
    number_of_steps = 0
    current_error = float("inf")
    nb_elem = sc.broadcast(data.count())

    ############################
    # Select initial centroides #
    ############################

    centroides = sc.parallelize(data.takeSample('withoutReplacment',nb_clusters))\
              .zipWithIndex()\
              .map(lambda x: (x[1],x[0][1][:-1]))
    print("initial_centroide", centroides.collect())
    # In the same manner, zipWithIndex gives an id to each cluster

    while not clusteringDone:
      joined = data.cartesian(centroides)

      # We compute the distance between the points and each cluster
      dist = joined.map(lambda x: (x[0][0],(x[1][0], computeDistance(x[0][1][:-1], x[1][1])
      )))

      dist_list = dist.groupByKey().mapValues(list)

      # We keep only the closest cluster to each point.
      min_dist = dist_list.mapValues(closestCluster)

      # assignment will be our return value : It contains the datapoint,
      # the id of the closest cluster and the distance of the point to the centroid
      assignment = min_dist.join(data)

      ############################################
      # Compute the new centroid of each cluster #
      ############################################

      clusters = assignment.map(lambda x: (x[1][0][0], x[1][1][:-1]))

      count = clusters.map(lambda x: (x[0],1)).reduceByKey(lambda x,y: x+y)
      somme = clusters.reduceByKey(sumList)
      centroidesCluster = somme.join(count).map(lambda x : (x[0],moyenneList(x[1][0],x
      [1][1])))
      ############################
      # Is the clustering over ? #
      ############################

      # Let's see how many points have switched clusters.
      if number_of_steps > 0:
          switch = prev_assignment.join(min_dist)\
                                  .filter(lambda x: x[1][0][0] != x[1][1][0])\
                                  .count()
      else:
          switch = 150
      if switch == 0 or number_of_steps == 100:
          clusteringDone = True
          error_improve = erreur_improve(assignment) #adding to compare
          error = sqrt(min_dist.map(lambda x: x[1][1]).reduce(lambda x,y: x + y))/nb_elem.
      value
      else:
```

```
57            centroides = centroidesCluster
58            prev_assignment = min_dist
59            if number_of_steps % 10 == 0 :
60              print("number_of_steps :", number_of_steps)
61            number_of_steps += 1
62
63      return (assignment, error, number_of_steps, centroides, error_improve)
```

7.1.3 Algorithme optimisé

```
1   def custom_kmeans(data, nb_clusters=3, join_centroids_inrdd=False,
        choose_initial_centroid=False, random_features=False, epsilon_change_nbfeatures=0,
        with_regular_check=False, interval_regular_check=10,distance="eucledian"):
2     """
3     This function executes the kmeans algorithm with combinations of the different
        implemented improvements.
4
5     Parameters :
6       * data
7       * nb_clusters
8       * join_centroids_inrdd (bool) : if set to True, at each iteration, one rdd line
        contains a point and the 3 centroids (and no longer 3 lines for 3 centroids)
9       * choose_initial_centroid (bool) : if set to True, (initialize according to kmeans+
        algo) the initial centroids are chosen with our improvement (1 : max, 2 : furthest
        from 1, 3 : middle between 1 & 2)
10      * random_features (bool) : if set to True, the algorithm will choose a subset of the
        features to compute the distance (2 to start, then 3, then 4)
11      * epsilon_change_nbfeatures (int) : if random_features is set to True, this parameter
         tells when to increase the number of features to select
12      * with_regular_check (bool) : if set to True and with random_features, performs a
        step with all features regularly
13      * interval_regular_check (int) : sets the number of iterations between two full
        feature checks
14      * distance (string) : the name of the distance to use (supported yet are "eucledian"
        and "manhattan")
15
16    Returns :
17      (assignment, error, number_of_steps)
18
19    """
20    clusteringDone = False
21    number_of_steps = 0
22    current_error = float("inf")
23    nb_elem = sc.broadcast(data.count())
24    nb_features_selected = 2
25
26    ############################
27    # Select initial centroides #
28    ############################
29    if choose_initial_centroid == True :
30      centroides = centroides_input(data)
31      print("initial_centroide", centroides.collect())
32    else :
```

```
33      centroides = sc.parallelize(data.takeSample('withoutReplacment',nb_clusters))\
34              .zipWithIndex()\
35              .map(lambda x: (x[1],x[0][1][:-1]))
36
37    while not clusteringDone:
38
39        ############################
40        # Assign points to clusters #
41        ############################
42        if join_centroids_inrdd == True :
43          joined = join_centroide_data(data,3)
44          if random_features == True :
45            features = []
46            if with_regular_check == True and (number_of_steps%interval_regular_check == 0)
      :
47              features = [0,1,2,3]
48            else:
49              # start by checking if we need to change the number of features
50              if(number_of_steps > 0) and switch < epsilon_change_nbfeatures and
      switch_prev < epsilon_change_nbfeatures and nb_features_selected < 4:
51                nb_features_selected += 1
52              random.seed(number_of_steps)
53              features = random.sample([0,1,2,3], k=nb_features_selected) #choose randomly
      the features to observe
54
55
56            if(distance == "manhattan"):
57              min_dist = min_manhattan_selectedfeatures(joined,features)
58            else :
59              min_dist = min_selectedfeatures(joined,features) # specify the features
      selected
60
61          else:
62            if(distance == "manhattan"):
63              min_dist = min_manhattan(joined)
64            else :
65              min_dist = min_(joined)
66
67          assignment = min_dist.join(data)
68
69        else :
70          joined = data.cartesian(centroides)
71
72          # We compute the distance between the points and each cluster
73          if(distance == "manhattan"):
74            dist = joined.map(lambda x: (x[0][0],(x[1][0], computeDistance_manhattan(x
      [0][1][:-1], x[1][1]))))
75          else:
76            dist = joined.map(lambda x: (x[0][0],(x[1][0], computeDistance(x[0][1][:-1], x
      [1][1]))))
77
78          dist_list = dist.groupByKey().mapValues(list)
79
80          # We keep only the closest cluster to each point.
81          min_dist = dist_list.mapValues(closestCluster)
82
83          # assignment will be our return value : It contains the datapoint,
84          # the id of the closest cluster and the distance of the point to the centroid
```

```
          assignment = min_dist.join(data)

        ##############################################
        # Compute the new centroid of each cluster #
        ##############################################

        clusters = assignment.map(lambda x: (x[1][0][0], x[1][1][:-1]))

        count = clusters.map(lambda x: (x[0],1)).reduceByKey(lambda x,y: x+y)
        #count = (1 , 100),(2 , 200), (3, 50)
        #       (numero de cluster, nb d'element)
        somme = clusters.reduceByKey(sumList)
        # somme = (1,sum of rows du clust 1), (2, 6000) , (3, 1500)
        # .    (numero de cluster, sommedes elements )
        centroidesCluster = somme.join(count).map(lambda x : (x[0],moyenneList(x[1][0],x
    [1][1])))
        # (numero de cluster, average)
        ############################
        # Is the clustering over ? #
        ############################

        # Let's see how many points have switched clusters.
        if number_of_steps > 0:
            switch_prev = switch
            switch = prev_assignment.join(min_dist)\
                                    .filter(lambda x: x[1][0][0] != x[1][1][0])\
                                    .count()

        else:
            switch = 150
            switch_prev = switch
        if (switch == 0 and nb_features_selected == 4) or number_of_steps > 3:
            clusteringDone = True
            error_improve = erreur_improve(assignment)
            error = sqrt(min_dist.map(lambda x: x[1][1]).reduce(lambda x,y: x + y))/nb_elem
    .value
        else:
          if nb_features_selected != 4 and switch == 0:
              nb_features_selected += 1
          centroides = centroidesCluster
          prev_assignment = min_dist
          if number_of_steps % 10 == 0 :
            print("number_of_steps :", number_of_steps)
          number_of_steps += 1

    return (assignment, error, number_of_steps, centroides, error_improve)
```

7.1.4 Main

```
def main_function(path='/FileStore/tables/iris_data-04fff.txt',nb_runs=3, teacher_runs =
    True, student_runs=True):
  """
  Global function that includes all the improvements.
```

```
4
5     Parameters :
6       path (string) : path to the data file
7       nb_runs (int) : number of runs to compute the mean for the time spent
8     Returns :
9       avg_custom (float) : the average of execution time for the custom algo
10      avg_basic (float) : the average of execution time for the original algo
11    """
12
13    # include parameters in case we use different data : nb of clusters ?, nb of features ?
14
15    startTimeQuery = time.clock()
16    lines = sc.textFile(path)
17    data = lines.map(lambda x: x.split(','))\
18            .map(lambda x: [float(i) for i in x[:4]]+[x[4]])\
19            .zipWithIndex()\
20            .map(lambda x: (x[1],x[0]))
21    custom_algo_times = []
22    basic_algo_times = []
23    if student_runs ==  True :
24      for i in range(nb_runs) :
25        print("")
26        start =strftime("%H:%M:%S", gmtime())
27        clustering = custom_kmeans(data, nb_clusters=3, join_centroids_inrdd=True,
        choose_initial_centroid=True, random_features=True, epsilon_change_nbfeatures=5,
        with_regular_check=True, interval_regular_check=10,distance="manhattan") # specify
        here the improvements to do in the parameters
28        print("CUSTOM :", "pourcentage_miss_labelised :" ,clustering[4], "error :",
        clustering[1], " clustering_nbr_loop : ", clustering[2], " clusters : ", clustering
        [3].collect())
29        custom_algo_times.append(seconds_between(start, strftime("%H:%M:%S", gmtime())))
30
31    if teacher_runs ==  True :
32      for i in range(int(nb_runs/10)) :
33        #compare to the teacher's function
34        start =strftime("%H:%M:%S", gmtime())
35        clustering = simpleKmeans(data, 3)
36        print("")
37        print("TEACHER :", "pourcentage_miss_labelised :" ,clustering[4], "error :",
        clustering[1], " clustering_nbr_loop : ", clustering[2]," clusters : ", clustering
        [3].collect())
38        basic_algo_times.append(seconds_between(start, strftime("%H:%M:%S", gmtime())))
39
40    avg_custom = sum(custom_algo_times)/nb_runs
41    avg_basic = 0
42    print("Nb of runs = ",nb_runs)
43    if(teacher_runs ==  True):
44      avg_basic = sum(basic_algo_times)/int(nb_runs/10)
45      print("Average of execution time for the original algo : ",avg_basic)
46      print("Execution times for the initial algo : ",basic_algo_times)
47    print("Average of execution time for the custom algo :",avg_custom)
48    print("Execution times for the custom algo : ",custom_algo_times)
49    return avg_custom,avg_basic,custom_algo_times,basic_algo_times
50
51  #-----------------------------------
52  #            MAIN
53  #-----------------------------------
54
```

```
#IRIS DATASET STUDENT ALGO

if __name__ == "__main__":
  path = '/FileStore/tables/iris_data-04fff.txt'
  nb_runs = 5
  data = sc.textFile(path).map(lambda x: x.split(',')).map(lambda x: [float(i) for i in x[:4]]+[x[4]])
  avg_custom,avg_basic,custom_algo_times,basic_algo_times = main_function(path=path, nb_runs=nb_runs, student_runs = True, teacher_runs = False)

#-----------------------------------
#            MAIN - VOITURES
#-----------------------------------

#VOITURES DATASET STUDENT ALGO

if __name__ == "__main__":
  path = '/FileStore/tables/voiture_type.txt'
  nb_runs = 5
  data = sc.textFile(path).map(lambda x: x.split(',')).map(lambda x: [float(i) for i in x[:4]]+[x[4]])
  avg_custom,avg_basic,custom_algo_times,basic_algo_times = main_function(path=path, nb_runs=nb_runs, student_runs = True, teacher_runs = False)
```

7.2 Dataframe

7.2.1 Fonctions

```
from pyspark.sql.functions import udf
from pyspark.sql import SQLContext
from pyspark.sql.functions import row_number, monotonically_increasing_id
from pyspark.sql import Window
import pyspark.sql.functions as F
import numpy as np

def centroide_input_df(df):
  df = df.withColumn('total', sum(df[col] for col in ["_c0","_c1", "_c2", "_c3"]))
  row1 = df.agg({"total": "max"}).collect()[0]
  max_tot = row1["max(total)"]
  max_point = df.where(F.col('total') == max_tot).select('_c0','_c1','_c2','_c3')
  list_max_point_coordonees = list(max_point.collect()[0][0:4])
  row2 = df.agg({"total": "min"}).collect()[0]
  min_tot = row2["min(total)"]
  min_point = df.where(F.col('total') == min_tot).select('_c0','_c1','_c2','_c3')
  list_min_point_coordonees = list(min_point.collect()[0][0:4])
  list_moy_point_coordonees = [(x+y)/2 for x,y in zip(list_min_point_coordonees, list_max_point_coordonees)]
  return list_min_point_coordonees,list_moy_point_coordonees,list_max_point_coordonees


"""
Création de la première dataframe
```

```
"""

def get_dataframe(path):
  sqlContext = SQLContext(sc)
  df = sqlContext.read.format('com.databricks.spark.csv').options(header='false',
    inferschema='true')\
      .load(path).withColumnRenamed('_c4','label')

  df = df.withColumn("index", row_number().over(Window.orderBy(
    monotonically_increasing_id()))-1)
  df = df.withColumn("coordonnees",F.array(F.col("_c0"),F.col("_c1"),F.col("_c2"),F.col("
    _c3")))
  return df

@udf("float")
def eucledian_distance(x,y):
  d = 0
  for i in range(len(x)):
    d += (x[i] - y[i])**2
  return d

@udf("integer")
def min_index(d1,d2,d3):
  if(d1 < d2 and d1 < d3):
    return 1
  if(d2 < d1 and d2 < d3):
    return 2
  return 3

def df_compute_min_distances(df,c1,c2,c3): # df contains id of the observation + features
     of the observation + features of each centroid as a list
  df = df.withColumn("c1",F.array([F.lit(i) for i in c1]))\
      .withColumn("c2",F.array([F.lit(i) for i in c2]))\
      .withColumn("c3",F.array([F.lit(i) for i in c3]))
  distances = df.withColumn("dist1",eucledian_distance(F.col("coordonnees"),F.col("c1")))
    \
                .withColumn("dist2",eucledian_distance(F.col("coordonnees"),F.col("c2")))
    \
                .withColumn("dist3",eucledian_distance(F.col("coordonnees"),F.col("c3")))
    \
                .drop("c1").drop("c2").drop("c3")
  min_df = distances.withColumn("cluster",min_index(F.col("dist1"),F.col("dist2"),F.col("
    dist3")))
  return min_df

@udf("array<float>")
def average_list(list_of_list):
  center = [0,0,0,0]
  nb = len(list_of_list)
  for i in range(nb):
    for j in range(0,4):
      center[j] += list_of_list[i][j]/nb
  return center

def df_update_centroids(df):
  df_gp = df.groupBy("cluster").agg(F.collect_list(F.col("coordonnees")).alias("
    list_of_coords"))
  return df_gp.withColumn("avg_coords",average_list(F.col("list_of_coords"))).drop("
```

```
      list_of_coords")

def df_get_new_centroids(df):
  array = []
  for i in range(1,4):
    array.append(df.filter(F.col("cluster")==i).collect()[0]["avg_coords"])
  return array[0],array[1],array[2]

def df_get_nb_switch(df_prev,df_suiv):
  join = df_prev.withColumnRenamed("cluster","cluster_prev").join(df_suiv,"index").filter
    (F.col("cluster_prev")!=F.col("cluster"))
  return join.count()

def df_compute_error(df, labels):
  list1 = [1,1,2,2,3,3]
  list2 = [2,3,3,1,2,1]
  list3 = [3,2,1,3,1,2]
  percentage_error_list = []
  for a,b,c in zip(list1,list2,list3) :
    combinaison = [(labels[0][0],a),(labels[1][0],b),(labels[2][0],c)]
    pourcentage_error = 0
    combianaison_assigment = df.select("label","cluster").collect()
    for i in combianaison_assigment :
      i = (i[0], i[1])
      if i not in (combinaison) :
        pourcentage_error += 1
    pourcentage_error /=  len(combianaison_assigment)
    percentage_error_list.append(pourcentage_error)
  return min_list(percentage_error_list)
```

7.2.2 Algorithme

```
def df_kmeans(data,nb_steps_max):
  clusteringDone = False
  number_of_steps = 0
  current_error = float("inf")
  nb_elem = sc.broadcast(data.count())
  switch = -1
  centroide1,centroide2,centroide3 = centroide_input_df(data)
  data = data.drop("_c0").drop("_c1").drop("_c2").drop("_c3").drop('total')
  while(not clusteringDone and number_of_steps < nb_steps_max):
    min_dist_df = df_compute_min_distances(data,centroide1,centroide2,centroide3)
    new_centroids_df = df_update_centroids(min_dist_df)
    centroide1,centroide2,centroide3 = df_get_new_centroids(new_centroids_df)
    if(number_of_steps > 0):
      switch = df_get_nb_switch(min_dist_df_prev,min_dist_df)
    min_dist_df_prev = min_dist_df
    number_of_steps += 1

    if(switch == 0):
      clusteringDone = True
      labels = data.select("label").distinct().collect()
      error = df_compute_error(min_dist_df, labels)
```

```
      if number_of_steps %10 ==0 :
        print("number_of_steps :", number_of_steps)
    assignment = min_dist_df.drop("dist1").drop("dist2").drop("dist3")
    centroides = [centroide1,centroide2,centroide3]
    return (assignment, error, number_of_steps, centroides)
```

7.2.3 Main

```
def df_main(path='/FileStore/tables/iris_data-04fff.txt',nb_runs=3):
  """
  Global function that includes all the improvements.

  Parameters :
    path (string) : path to the data file
    nb_runs (int) : number of runs to compute the mean for the time spent
  Returns :
    avg_df (float) : the average of execution time for the kmeans dataframe algo
  """

  startTimeQuery = time.clock()
  path = '/FileStore/tables/voiture_type.txt'

  data = get_dataframe(path)
  df_algo_times = []
  for i in range(nb_runs) :
    print("")
    start =strftime("%H:%M:%S", gmtime())

    clustering = df_kmeans(data,150)
    print("DATAFRAME :", "error :", clustering[1], " clustering_nbr_loop : ",
    clustering[2], " clusters : ", clustering[3])
    df_algo_times.append(seconds_between(start, strftime("%H:%M:%S", gmtime())))

  avg_df = sum(df_algo_times)/nb_runs
  print("Nb of runs = ",nb_runs)

  print("Average of execution time for the dataframe algo :",avg_df)
  print("Execution times for the dataframe algo : ",df_algo_times)
  return avg_df,df_algo_times

#-----------------------------------
#            MAIN
#-----------------------------------

#IRIS DATASET STUDENT ALGO

if __name__ == "__main__":
  path = '/FileStore/tables/iris_data-04fff.txt'
  nb_runs = 5
  df_main(path,nb_runs)

#-----------------------------------
#            MAIN - VOITURES
```

```
#-------------------------------------

#VOITURES DATASET STUDENT ALGO

if __name__ == "__main__":
  path = '/FileStore/tables/voiture_type.txt'
  nb_runs = 5
  df_main(path,nb_runs)
```

7.3 Jeu de test pour les dataframes

```
clust1 = [0,0,0,0]
clust2 = [1,1,1,1]
clust3 = [2,2,2,2]

from pyspark.sql.types import *

cSchema = StructType([StructField("index", IntegerType())\
                      ,StructField("x1",FloatType())\
                      ,StructField("x2",FloatType())\
                      ,StructField("x3",FloatType())\
                      ,StructField("x4",FloatType())])

test_list = [[1,0.0,0.0,0.0,0.0],[2,0.0,0.0,0.0,0.0],[3,1.0,1.0,1.0,1.0],
    [4,2.0,2.0,2.0,2.0],[5,1.2,1.2,1.2,1.2]]

df = spark.createDataFrame(test_list,schema=cSchema)
df = df.select(F.col("index"),F.array(F.col("x1"),F.col("x2"),F.col("x3"),F.col("x4")).
    alias("coordonnees"))

df.show()
dis = df_compute_min_distances(df,clust1,clust2,clust3)
dis.show()
gp_df = df_update_centroids(dis)
gp_df.show()
print("NEW CENTROIDS = ",df_get_new_centroids(gp_df))
print("NB switch = ",df_get_nb_switch(dis,dis))
```

7.4 Tableau des résultats

Expérimentation			Conditions									
Algorithme initial	Numéro	Requête	NB CLUSTERS	join_centroids_in_rdd	choose_initial_centroid	random_features	epsilon_change_nb_features	with_regular_check	interval_regular_check	distance
	0	simpleKmeans(data, nb_clusters)	3	x	x	x	x	x	x	x
Algorithme personnalisé	1	custom_kmeans(data,nb_clusters=3,join_centroids_in_rdd=False,choose_initial_centroids=False,random_features=False,epsilon_change_nb_features=5,with_regular_check=False,interval_regular_check=10,distance=eucledian)	3	False	True	False	5	False	10	eucledian
	2	custom_kmeans(data,nb_clusters=3,join_centroids_in_rdd=True,choose_initial_centroids=False,random_features=False,epsilon_change_nb_features=5,with_regular_check=False,interval_regular_check=10,distance=eucledian)		True						
	3	custom_kmeans(data,nb_clusters=3,join_centroids_in_rdd=False,choose_initial_centroids=True,random_features=False,epsilon_change_nb_features=5,with_regular_check=False,interval_regular_check=10,distance=eucledian)		False	True					
	4	custom_kmeans(data,nb_clusters=3,join_centroids_in_rdd=True,choose_initial_centroids=True,random_features=False,epsilon_change_nb_features=5,with_regular_check=False,interval_regular_check=10,distance=eucledian)								
	5	custom_kmeans(data,nb_clusters=3,join_centroids_in_rdd=True,choose_initial_centroids=False,random_features=True,epsilon_change_nb_features=5,with_regular_check=False,interval_regular_check=10,distance=eucledian)	3	True	False	False	5	True	10	eucledian
	6	custom_kmeans(data,nb_clusters=3,join_centroids_in_rdd=True,choose_initial_centroids=False,random_features=True,epsilon_change_nb_features=5,with_regular_check=True,interval_regular_check=10,distance=eucledian)								
	7	custom_kmeans(data,nb_clusters=3,join_centroids_in_rdd=True,choose_initial_centroids=True,random_features=True,epsilon_change_nb_features=5,with_regular_check=False,interval_regular_check=10,distance=eucledian)			True	True		False		
	8	custom_kmeans(data,nb_clusters=3,join_centroids_in_rdd=True,choose_initial_centroids=True,random_features=True,epsilon_change_nb_features=5,with_regular_check=True,interval_regular_check=10,distance=eucledian)						True		manhattan
	9	custom_kmeans(data,nb_clusters=3,join_centroids_in_rdd=True,choose_initial_centroids=True,random_features=True,epsilon_change_nb_features=5,with_regular_check=True,interval_regular_check=10,distance=manhattan)								

FIGURE 1: Description des expériences

		Temps moyen sur 5 essais						Temps moyen sur 5 essais					
Numéro	**Format**	**1**	**2**	**3**	**4**	**5**	**Moyenne**	**1**	**2**	**3**	**4**	**5**	**Moyenne**
0	RDD	Iris						Vehicule_types					
1	RDD	Executor heartbeat timed out after 136447 ms					400	x	x	x	x	x	Aucun résultat après 2h ..
2	RDD	5	6	6	6	5	5,6	9	10	10	9	9	9,4
3	RDD	170	172	170	170	171	170,6	x	x	x	x	x	Aucun résultat après 2h ..
4	RDD	5	4	5	4	5	4,6	6	5	5	5	5	5,2
5	RDD	23	22	25	27	22	23,8	21	22	25	21	23	22,4
6	RDD	27	27	28	27	27	27,2	34	34	33	35	37	34,6
7	RDD	18	18	18	17	18	17,8	20	20	20	19	20	19,8
8	RDD	26	25	26	25	25	25,4	33	35	33	33	34	33,6
9	RDD	29	30	30	31	30	30	33	33	34	33	35	33,6
	Dafaframe							81	85	87			84,33333333

FIGURE 2: Résultats obtenus en secondes