Dauphine | PSL
UNIVERSITÉ PARIS

Rapport de projet

L'algorithme kmeans

Auteurs :
Shahram MAHDAVI
Charly SIMONIAN
Yusuf YILDIZ

Encadrant :
Dario COLAZZO

10 Janvier 2020

Table des matières

Kmeans avec MLlib..... 2
Kmeans ++ 4
Optimisation du code..... 5
Application sur un jeux de données plus grande : 7
Exécutions 8
Graphiques 8

Kmeans avec MLlib

Nous avons tout d'abord opté pour l'implémentation de Kmeans avec une librairie de spark qui se nomme MLlib(https://spark.apache.org/docs/latest/ml-clustering.html). Cette bibliothèque nous à permis d'avoir une référence sur le temp d'exécution de nos différents algorithmes sur les data set que nous utiliserons. En étudiant son fonctionnement nous avons pu trouver deux axes d'analyses supplémentaires.

Le premier se pose sur la condition d'arrêt de notre entrainement, en effet lorsque on entraine le model nous pouvons définir un paramètre qui se nomme « *epsilon* ». Ce paramètre nous sert à stopper l'algorithme avant d'atteindre le nombre maximum d'itération. Pour choisir quand l'algorithme va devoir stopper ses itérations nous allons calculer à chaque itération l'erreur quadratique (*i.e. la distance moyenne de chaque point à son centroïde*) grâce à la variable présente dans le code fourni au début et qui se nomme « *err* ».
A chaque itération nous calculons la différence entre l'erreur quadratique actuelle et la précédente, si la différence obtenue est inférieure à notre epsilon fixé alors le programme va s'arrêté.

Par la suite nous avons décidé de l'implémenter dans tous nos algorithmes que nous allons développer. L'idée derrière cela est de dire que quand le déplacement des centroïdes n'engendre plus d'amélioration significative du clustering ou qu'ils ne se déplacent plus alors il ne sert à rien de continuer les itérations. Il est cependant important de noter que tout comme pour le nombre d'itération c'est un paramètre que l'on doit définir au lancement donc sa valeur optimale dépend totalement du data set ainsi que du nombre de cluster que l'on souhaite créer.

Le deuxième est la découverte d'une autre méthode pour initialiser les centroïdes avec une version améliorée de kmens qui se nomme kmeans++. Tout comme pour la valeur d'epsilon c'est un paramètre que l'on doit définir dans cette librairie et qui se nomme « *initializationMode* ». Le premier mode possible et par défault est « *random* » qui choisit les centroïdes au hasard et le deuxième est « *k-means||* » qui implémente kmeans++.

```python

from numpy import array
from math import sqrt
from pyspark import SparkContext
from pyspark.mllib.clustering import KMeans, KMeansModel

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

if __name__ == "__main__":

    sc = SparkContext(appName="exercice")  # SparkContext

    # Load and parse the data
    data = sc.textFile("hdfs:/user/user15/iris.data.txt")

    data_km = data.map(lambda x: x.split(','))

    data_km = data_km.map(lambda x: [float(i) for i in x[:4]])

    parsedData = data_km.map(lambda x: array(x))

    # Build the model (cluster the data)
    clusters = KMeans.train(parsedData, 3, maxIterations=100, initializationMode="k-means||")

    # la fonction qui calcule la somme des distances entre chaque point et son centroide associes
    def error(point):
        center = clusters.centers[clusters.predict(point)]
        return sqrt(sum([x**2 for x in (point - center)]))

    WSSSE = parsedData.map(lambda point: error(point)).reduce(lambda x, y: x + y)
    print("la somme des distances = " + str(WSSSE))

    # Save and load model
    data2 = data.map(lambda x: x.split(','))\
            .map(lambda x: [float(i) for i in x[:4]]+[x[4]])

    data3=data2.zipWithIndex().map(lambda x: (x[1],x[0]))

    data6=data3.map(lambda x : (x[0],((clusters.predict(array(x[1][:-1]))\
           ,computeDistance(x[1][:-1],clusters.centers[clusters.predict(array(x[1][:-1]))]\
           .tolist())),x[1])))

    data6.saveAsTextFile("hdfs:/user/user15/output_final5")

    sc.stop()

```

Kmeans ++

Nous avons vu que dans l'algorithme Kmeans classique l'initialisation des centroïdes se fait de manière aléatoire et cela est un des inconvénients de cet algorithme. Ainsi, si un centroïde est initialisé pour être un point « lointain », il pourrait simplement se retrouver sans aucun point qui lui est associé et en même temps, plusieurs clusters pourraient se retrouver liés à un seul centroïde.
En effet, il est important de bien initialiser les centroïdes avec des coordonnées différentes car si l'initialisation se fait avec des points ayant des cordonnées similaires, il est possible de choisir 2 centroïdes dans le même cluster, ce qui entraîne un mauvais clustering. Par ailleurs, un mauvaise clustering peut donner des résultats aberrants.
Voici un exemple de mauvaise clustering dans l'image ci-dessous.

Pour surmonter l'inconvénient mentionné ci-dessus, nous avons décidé de suivre une autre technique un peu plus complexe qui réduit le nombre d'itérations considérablement. Nous avons choisi l'algorithme Kmeans ++. Cet algorithme assure une initialisation plus intelligente des centroïdes et améliore la qualité du clustering. Hormis l'initialisation, le reste de l'algorithme est le même que l'algorithme Kmeans standard. C'est-à-dire que Kmeans ++ est l'algorithme standard de Kmeans mais avec une initialisation plus intelligente des centroïdes.

L'initialisation de kmeans++ s'effectue de la manière suivante :

On choisit le premier centre de cluster de manière aléatoire parmi les points de données, Ensuite On choisit le centre de cluster suivant parmi les points de données restants avec une probabilité

proportionnelle à sa distance au carré du centre de cluster existant le plus proche de point. On continue cette démarche jusqu'à ce que tous les centroïdes aient été sélectionnées.

En suivant cette procédure d'initialisation, nous prenons des centroïdes qui sont loin les uns des autres et Cela augmente les chances de ramasser initialement les centroïdes qui se trouvent dans différents clusters. De plus, comme les centroïdes sont récupérés à partir des points de données, chaque centroïde a des groupes de points de données associés à la fin.

Optimisation du code

Défauts du programme initial

Le programme K-Means de base bien que fonctionnel, a un temps d'exécution peu optimal. Son temps d'exécution sur un jeu de données de 150 lignes (iris) dépasse les 10 minutes et ne serait pas viable pour des datasets plus conséquents.
Il nous a donc fallu cibler les défauts du code et les corriger pour améliorer sa complexité.

Le problème majeur correspond au nombre de **Shuffle&Sort** effectués par le programme. En effet les tasks tels que : *join*, *groupByKey*, *reduceByKey* ou encore *cartesian* demandent beaucoup d'efforts à Spark en raison des croisements entre RDD suivis des tris, nécessaires à leurs exécutions.

Une fois le problème ciblé, il faut le corriger. Nous avons procédé à la suppression de tous les Shuffle&Sort non obligatoirement nécessaires au programme en modifiant les RDD de la sorte :

- Ajout des coordonnées de tous les centroides sur chaque lignes du RDD sans jointure cartesienne.
    - L'opération *cartesian* était trop gourmande et pouvait être facilement remplacée.
      Par ailleurs ça nous évite de faire un *groupByKey*.
- Excès d'informations dans les RDD pour ne pas avoir à faire des *join* plus tard.
    - Exemple : on garde les données de chaque point dans chaque RDD pour ne pas avoir à faire un *join* avec le RDD *data* par la suite.
- Plusieurs opérations en une action.
    - Exemple : pour le calcul des nouveaux centroides, au lieu de calculer la somme des points puis leur nombre, on fait cela en une seule opération et on stocke ces 2 chiffres dans la même ligne de RDD.
- Simplifier la condition d'arrêt du programme lors de convergence.
    - Le programme initial pour vérifier la convergence, faisait la comparaison entre le RDD de l'itération actuelle et celui de la précédente itération. Au lieu de ça nous comparant juste les erreurs de chaque itération. Si la différence des 2 erreurs est inférieure à un epsilon donné, on considère qu'il y a convergence.

En plus de ces opérations, nous avons partitionné le RDD de départ ce qui est non négligeable pour les très gros datasets lors de l'exécution du programme sur cluster.
Par ailleurs 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.

```
while not clusteringDone:

    joined = data.map(lambda x:(x,[c.value[i] for i in range(nb_clusters)]))

    min_dist = joined.map(lambda x: closestClusterBis(((x[0][0],x[0][1]),[(x[1][i][0], computeDistance(x[0][1][:-1], x[1][i][1])) for i

    min_distBis = min_dist.map(lambda x:(x[0][0],x[1]))
    min_distBis.persist()

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

    count=clusters.reduceByKey(lambda x,y: (sl(x[0],y[0]), x[1]+y[1]))
    centroidesCluster=count.map(lambda x: (x[0],moyenneList(x[1][0],x[1][1])))

    if number_of_steps == 0:
        switch = 150
        prev_assignment = 0
    else:
        switch = sqrt(min_distBis.map(lambda x: x[1][1]).reduce(lambda x,y: x + y))/nb_elem.value

    if abs(switch-prev_assignment)>epsilon and number_of_steps != 50:
        c = sc.broadcast([list(row) for row in centroidesCluster.collect()])
        prev_assignment = sqrt(min_distBis.map(lambda x: x[1][1]).reduce(lambda x,y: x + y))/nb_elem.value
        print("******************************NofSteps:"+str(number_of_steps))
        print("switch"+str(switch))
        number_of_steps += 1

    else:
        clusteringDone = True
        error = sqrt(min_distBis.map(lambda x: x[1][1]).reduce(lambda x,y: x + y))/nb_elem.value
```

Application sur un jeux de données plus grande :

Pour tester notre algorithme sur un jeux de données plus grande, nous avons décidé de récupérer le dataset de **Minute weather** qui contient les données météorologiques brutes capturées à des intervalles d'une minute. Ces données proviennent d'une station météorologique située à San Diego, en Californie.

La station météorologique est équipée de capteurs qui capturent les mesures météorologiques telles que la température de l'air, la pression de l'air et l'humidité relative. Les données ont été collectées pour une période de trois ans, de septembre 2011 à septembre 2014, afin de garantir la collecte de données suffisantes pour différentes saisons et conditions météorologiques.

Avant de faire le clustering sur ce jeu de données, nous avons effectué un nettoyage sur ces données, par exemple, nous avons supprimé les valeurs NaN et nous avons supprimé également les variables qui ne sont pas nécessaire pour le clustering. A la fin de cette étape, nous avons un jeu de donnée qui contient 158680 points qu'on souhaite classer dans le 12 cluster.

Chaque ligne ou chaque échantillon de notre dataset se compose des variables suivantes :

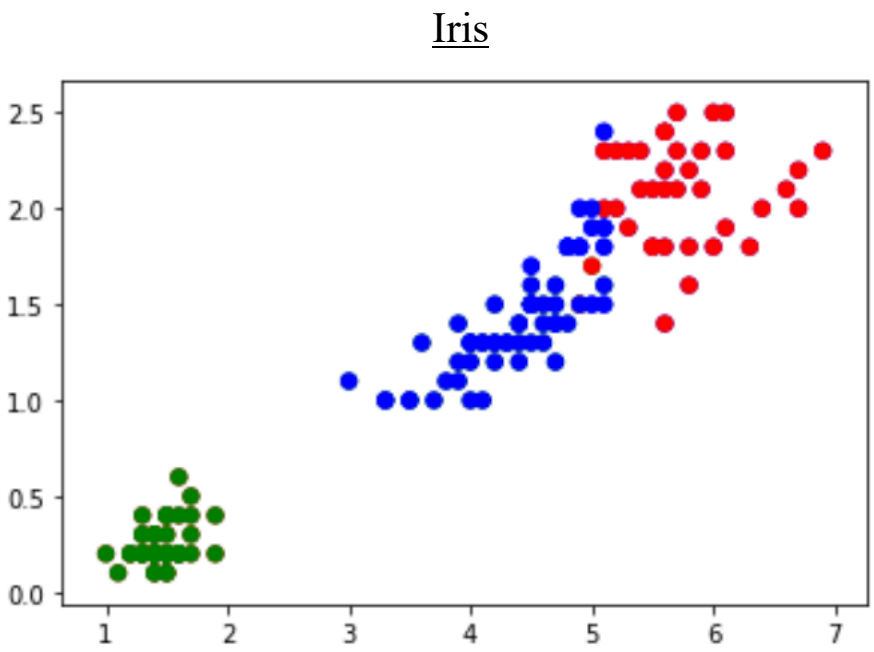
- **air_pressure:** air pressure measured at the timestamp (*Unit: hectopascals*)
- **air_temp:** air temperature measure at the timestamp (*Unit: degrees Fahrenheit*)
- **avg_wind_direction:** wind direction averaged over the minute before the timestamp (*Unit: degrees, with 0 means coming from the North, and increasing clockwise*)
- **avg_wind_speed:** wind speed averaged over the minute before the timestamp (*Unit: meters per second*)
- **max_wind_direction:** highest wind direction in the minute before the timestamp (*Unit: degrees, with 0 being North and increasing clockwise*)
- **max_wind_speed:** highest wind speed in the minute before the timestamp (*Unit: meters per second*)
- **relative humidity**: measured at the timestamp (*Unit: percent*)

Exécutions

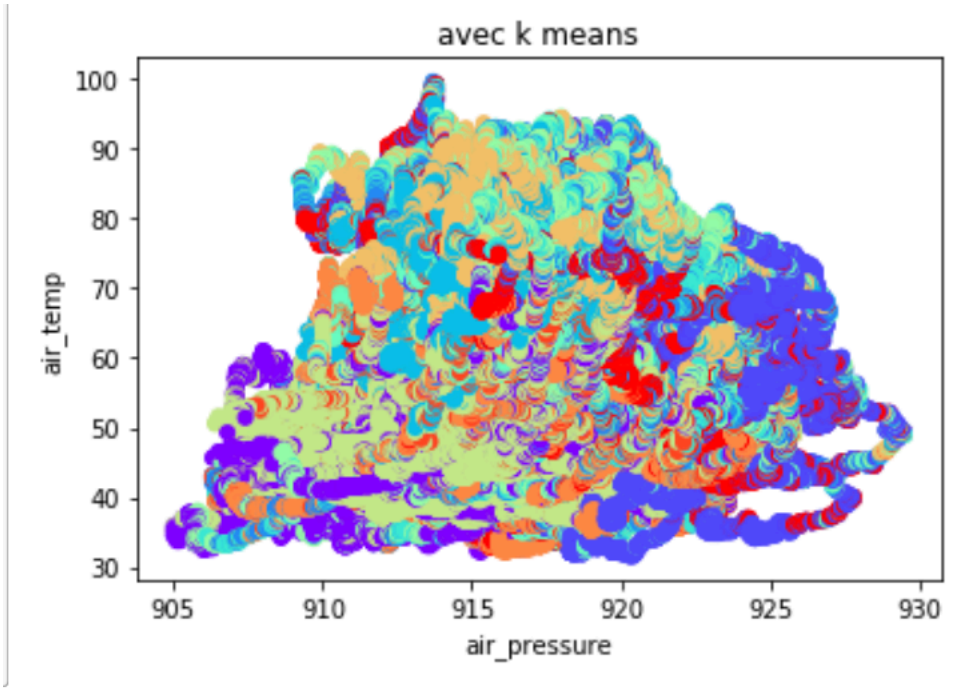
Voici les chiffres des différentes exécutions avec leurs paramètres, en incluant les optimisations décrites plus haut :

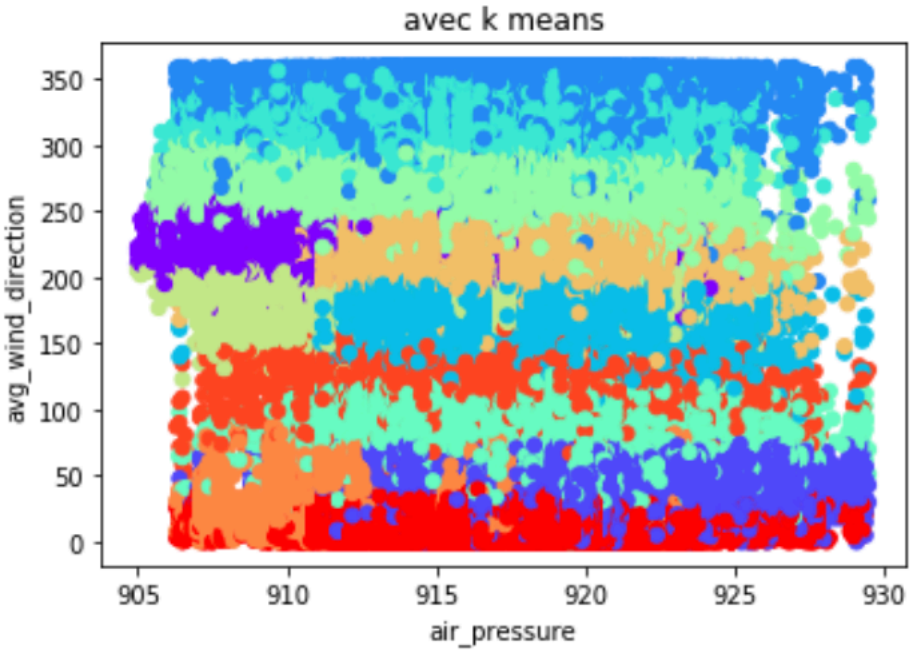
Algorithme (fichier)	Temps d'exécution	Itérations	Clusters
Kmeans (iris.data.txt)	10.8s	12	3
Kmeans&Km++ (iris.data.txt)	5.8s	5	3
Kmeans MLlib (iris.data.txt)	6.6s	Inconnu	3
Kmeans (Minute_Weather.txt)	282.3s	37	12
Kmeans&Km++ (Minute_Weather.txt)	130.7s	15	12
Kmeans MLlib (Minute_Weather.txt)	17.3s	Inconnu	12

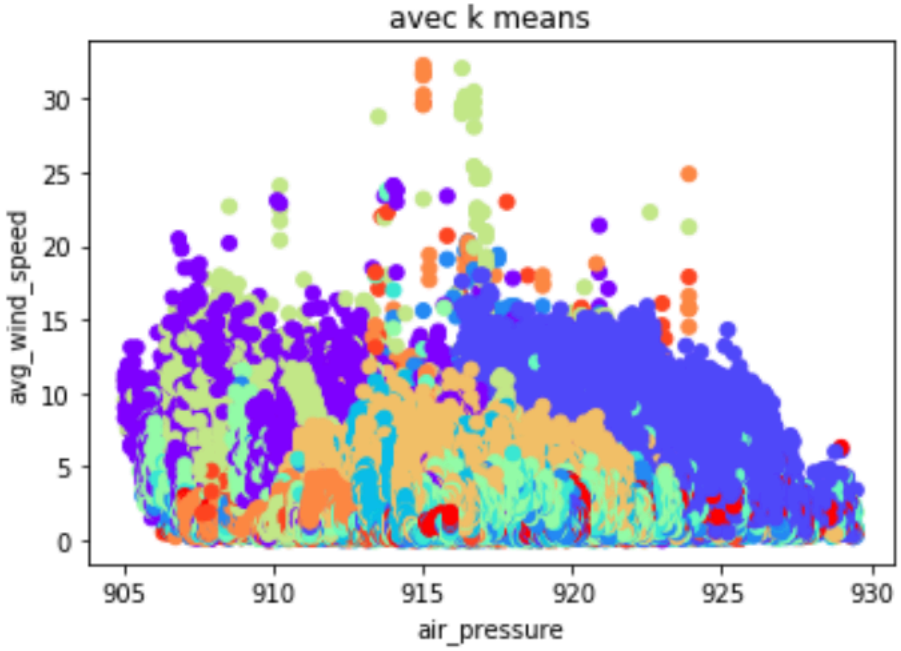
Graphiques

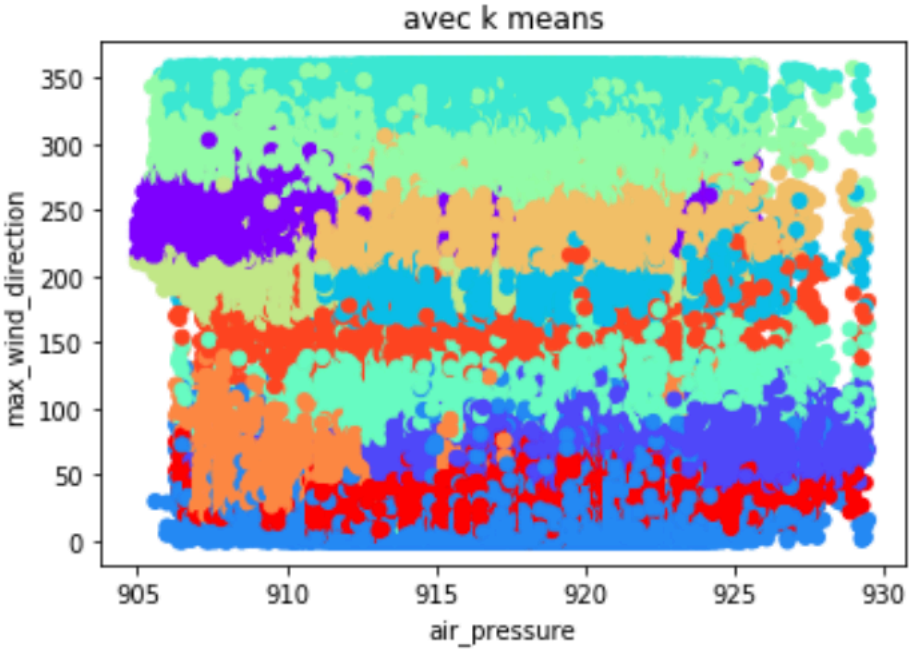


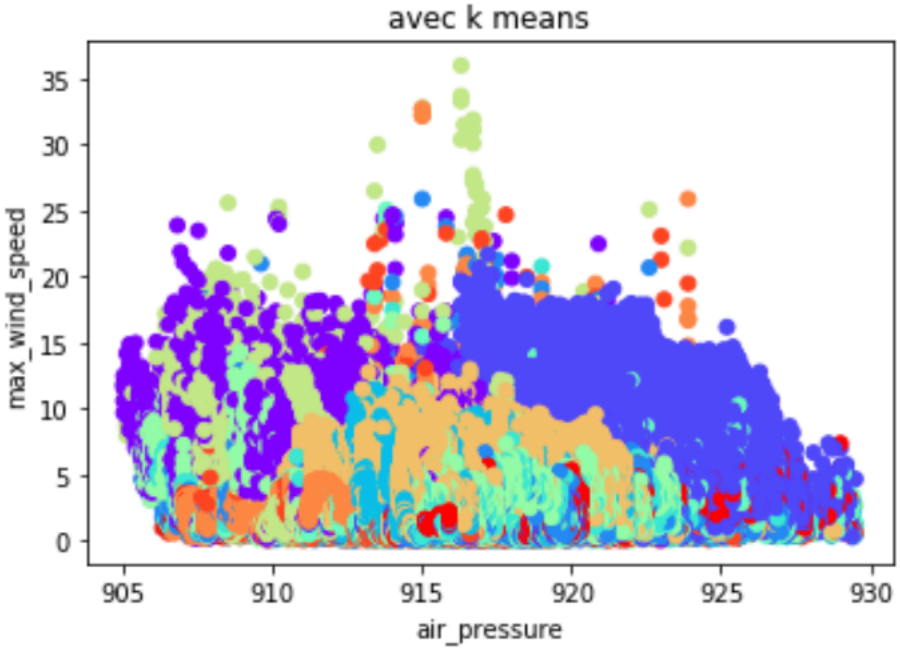
Minute Weather

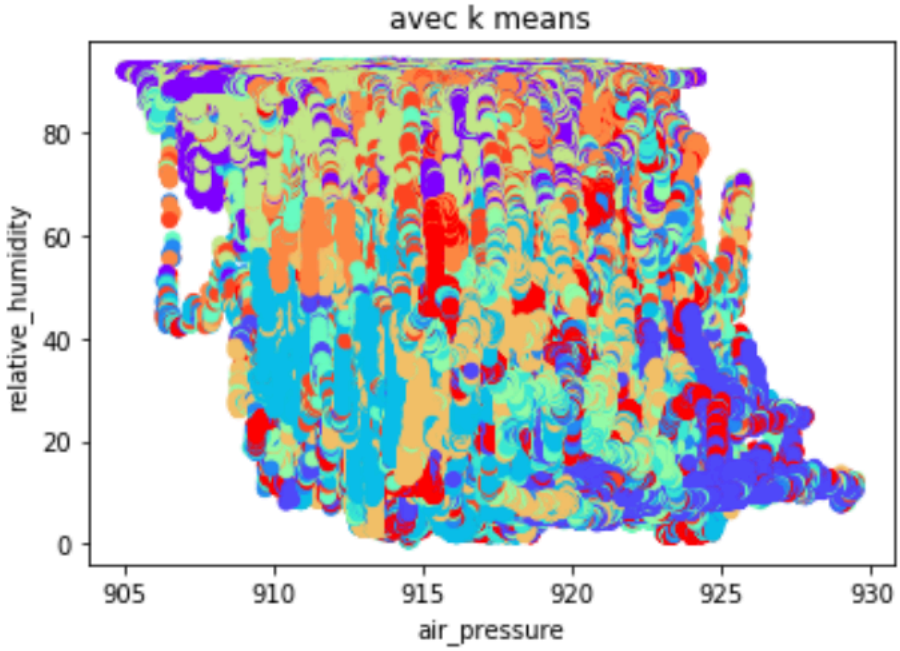





avec k means
avg_wind_speed
air_pressure
0
5
10
15
20
25
30
905
910
915
920
925
930


avec k means
max_wind_direction
air_pressure
0
50
100
150
200
250
300
350
905
910
915
920
925
930


avec k means
max_wind_speed
air_pressure
0
5
10
15
20
25
30
35
905
910
915
920
925
930


avec k means
relative_humidity
air_pressure
0
20
40
60
80
905
910
915
920
925
930

Bibliographie

(1) https://spark.apache.org/docs/latest/ml-clustering.html

(2) https://www.geeksforgeeks.org/ml-k-means-algorithm/

(3) https://github.com/st186/Weather-Data-Clustering-Using-k-means

(4) https://www.math.univ-toulouse.fr/~besse/Wikistat/pdf/st-tutor4-pyspark-mllib.pdf