Dr. Heni Bouhamed
Data Scientist, Maître Assistant à la Fseg-Université de Sfax
Formation Data Science
Module 7 : Apache Spark

TP2 Apache Spark : mllib et RDDs

--K-means--

1) Installer numpy si vous utiliser ubuntu (sur windows, numpy est déjà présent puisqu'on a installé anaconda) avec pip install numpy.
2) Importer la fonction KMeans de mllib avec : from pyspark.mllib.clustering import KMeans.
3) Importer la fonction array avec : from numpy import array.
4) Importer la fonction sqrt avec : from math import sqrt.
5) Placer le fichier test.txt sur D: pour windows, sur le home pour ubuntu.
6) Charger le fichier test.txt avec data = sc.textFile("test.txt") (data = sc.textFile("D:\\test.txt") pour windows).
7) #Etape est optionnelle# Distribuer les données en RDD (selon les lignes de données, c'est une action*) avec : list=data.collect().
8) Une autre façon d'afficher les données distribuées avec :
   for l in list:
     print(l)
9) Préparer la transformation des données mais en séparant les éléments de chaque ligne en les transformant en float avec : parsedData = data.map(lambda line:array([float(x) for x in line.split(' ')])).
   Rq : lambda représente dans python une fonction sans avoir un nom.
10) #Etape optionnelle# Exécuter en affichant les données transformées avec : print(parsedData.collect()).
11) Distribuer les données en RDD avec : parsedData.collect().
12) Lancer l'algorithme kmeans avec : clusters = KMeans.train(parsedData, 2, maxIterations=10, runs=10, initializationMode="random").
13) Exécuter (avec affichage) le résultat avec : clusters.predict(parsedData).collect().

--wordcount--

14) Placer le fichier wordcount sur D: pour windows, sur le home pour ubuntu.
15) Charger le fichier wordcount avec : text_file = sc.textFile("wordcount.txt") (text_file = sc.textFile("D:\\wordcount.txt") pour windows).
16) Préparer la transformation des lignes en séparant, d'abord, les mots, en transformant, ensuite, chaque mot en couple (mot,1) et en comptant, enfin, le nombre de chaque mot-clé avec :
    counts = text_file.flatMap(lambda line: line.split(" ")).map(lambda word: (word,1)).reduceByKey(lambda a, b: a + b)
17) Exécuter l'action avec : counts.collect().
18) Enregistrer le résultat avec : counts.saveAsTextFile("D:\wordcountresults.txt").

--Arbre de décision--

19) Placer le fichier iris num.csv sur D: pour windows, sur le home pour ubuntu.
20) Charger le fichier iris num.csv avec : data = sc.textFile("iris num.csv").
21) Importer array de numpy avec : from numpy import array.
22) Préparer la transformation des lignes en séparant les valeurs et en les convertissant en float avec : pdata = data.map(lambda line:array([float(x) for x in line.split(',')])).
23) Exécuter l'action avec : pdata.collect().

24) Importer la fonction LabeledPoint avec : from pyspark.mllib.regression import LabeledPoint.
25) Créer une fonction appelée parse permettant de labellisé une la valeur classe d'une ligne de données reçu en entrée avec :
    def parse(l):
        return LabeledPoint(l[4],l[0:4])
26) Passer les lignes une par une afin de labelliser toute les données avec : fdata=pdata.map(lambda l:parse(l)).
27) Diviser aléatoirement les données afin d'avoir une base d'apprentissage et une autre de test avec : (trainingData, testData) = fdata.randomSplit([0.8, 0.2],0.0).
28) Importer la fonction des arbres de décision avec : from pyspark.mllib.tree import DecisionTree.
29) Préparer le modèle avec : model = DecisionTree.trainClassifier(trainingData, numClasses=3, categoricalFeaturesInfo={}).
30) Effectuer la prédiction pour la base test avec : predictions = model.predict(testData.map(lambda r: r.features)).
31) Construire un tableau de deux colonnes opposant les prédictions aux valeurs réelles avec : predictionAndLabels = testData.map(lambda lp: lp.label).zip(predictions).
32) Importer la fonction MulticlassMetrics pour l'évaluation du modèle avec : from pyspark.mllib.evaluation import MulticlassMetrics.
33) Lancer la fonction d'évaluation avec : metrics = MulticlassMetrics(predictionAndLabels).
34) Calculé la précision du modèle avec : precision = metrics.precision().
35) Calculé le recall du modèle avec recall = metrics.recall().
36) Calculé le f1Score avec : f1Score = metrics.fMeasure().
37) Afficher un titre aux résultats avec : print("Summary Stats").
38) Afficher la précision du modèle avec : print("Precision = %s" % precision).
39) Afficher le recall du modèle avec : print("Recall = %s" % recall).
40) Afficher le f1score du modèle avec : print("F1 Score = %s" % f1Score).

--Régression logistique--

41) Charger le fichier iris num.csv avec : data = sc.textFile("iris num.csv").
42) Importer array de numpy avec : from numpy import array.
43) Préparer la transformation des lignes en séparant les valeurs et en les convertissant en float avec : pdata = data.map(lambda line:array([float(x) for x in line.split(',')])).
44) Exécuter l'action avec : pdata.collect().
45) Importer la fonction LabeledPoint avec : from pyspark.mllib.regression import LabeledPoint.
46) Créer une fonction appelée parse permettant de labellisé une la valeur classe d'une ligne de données reçu en entrée avec :
    def parse(l):
        return LabeledPoint(l[4],l[0:4])
47) Passer les lignes une par une afin de labelliser toute les données avec : fdata=pdata.map(lambda l:parse(l)).
48) Diviser aléatoirement les données afin d'avoir une base d'apprentissage et une autre de test avec : (trainingData, testData) = fdata.randomSplit([0.8, 0.2]).
49) Importer la fonction des arbres de décision avec : from pyspark.mllib.classification import LogisticRegressionWithLBFGS.
50) Préparer le modèle avec : model = LogisticRegressionWithLBFGS.train(trainingData, numClasses=3).

51) Effectuer la prédiction pour la base test avec : predictions = model.predict(testData.map(lambda r: r.features)).
52) Convertir predictions à float (par défaut le modèle de régression logistique, qu'on a utilisé, retourne des entiers) avec : predictions=predictions.map(lambda x:float(x)).
53) Construire un tableau de deux colonnes opposant les prédictions aux valeurs réelles avec : predictionAndLabels = testData.map(lambda lp: lp.label).zip(predictions).
54) Importer la fonction MulticlassMetrics pour l'évaluation du modèle avec : from pyspark.mllib.evaluation import MulticlassMetrics.
55) Lancer la fonction d'évaluation avec : metrics = MulticlassMetrics(predictionAndLabels).
56) Calculé la précision du modèle avec : precision = metrics.precision().
57) Calculé le recall du modèle avec recall = metrics.recall().
58) Calculé le f1Score avec : f1Score = metrics.fMeasure().
59) Afficher un titre aux résultats avec : print("Summary Stats").
60) Afficher la précision du modèle avec : print("Precision = %s" % precision).
61) Afficher le recall du modèle avec : print("Recall = %s" % recall).
62) Afficher le f1score du modèle avec : print("F1 Score = %s" % f1Score).

--Quelques transformations avec des exemples--

Transformation	Description
`map(func)`	return a new distributed dataset formed by passing each element of the source through a function *func*
`filter(func)`	return a new dataset formed by selecting those elements of the source on which *func* returns true
`distinct([numTasks]))`	return a new dataset that contains the distinct elements of the source dataset
`flatMap(func)`	similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item)

```
>>> rdd = sc.parallelize([1, 2, 3, 4])
>>> rdd.map(lambda x: x * 2)
RDD: [1, 2, 3, 4] → [2, 4, 6, 8]

>>> rdd.filter(lambda x: x % 2 == 0)
RDD: [1, 2, 3, 4] → [2, 4]

>>> rdd2 = sc.parallelize([1, 4, 2, 2, 3])
>>> rdd2.distinct()
RDD: [1, 4, 2, 2, 3] → [1, 4, 2, 3]
>>> rdd = sc.parallelize([1, 2, 3])
>>> rdd.Map(lambda x: [x, x+5])
RDD: [1, 2, 3] → [[1, 6], [2, 7], [3, 8]]

>>> rdd.flatMap(lambda x: [x, x+5])
RDD: [1, 2, 3] → [1, 6, 2, 7, 3, 8]
```

--Quelques actions avec exemples--

Action	Description
`reduce(func)`	aggregate dataset's elements using function *func*. *func* takes two arguments and returns one, and is commutative and associative so that it can be computed correctly in parallel
`take(n)`	return an array with the first *n* elements
`collect()`	return all the elements as an array WARNING: make sure will fit in driver program
`takeOrdered(n, key=func)`	return n elements ordered in ascending order or as specified by the optional key function

```
>>> rdd = sc.parallelize([1, 2, 3])
>>> rdd.reduce(lambda a, b: a * b)
Value: 6

>>> rdd.take(2)
Value: [1,2] # as list

>>> rdd.collect()
Value: [1,2,3] # as list
>>> rdd = sc.parallelize([5,3,1,2])
>>> rdd.takeOrdered(3, lambda s: -1 * s)
Value: [5,3,2] # as list
```