TP1 :BIG DATA

Réalisé par : Mehdi KARECH

Partie I : RDD sur databricks

1)2)3) création du vecteur aléatoire le charger dans un rdd et afficher le nombre de partition

```
import numpy as np
data = np.random.rand(20000)
rdd = sc.parallelize(data)
print(rdd.getNumPartitions())

```

```
8
```

Le nombre de partition égale 8 Cela veut dire qu'on peut lancer 8 taches sur 8 noeud en parallèles (une tache (task) par partition)

4)5) Chargement du fichier “text_gradient.txt” et suppréssion des steps words de ce dernier

```
stop_words = ['the', 'a', 'of']
rdd2 = sc.textFile('FileStore/tables/text_gradient.txt')
rdd_filtred= rdd2.map(lambda x: x.replace(' the ', ' ')).map(lambda x: x.replace(' a ', ' ')).map(lambda x: x.replace(' of ', ' '))
rdd_filtred.take(5)

```

```
▸ (1) Spark Jobs
Out[4]: ['Gradient descent is first-order iterative optimization algorithm for ',
 'finding local minimum differentiable function.',
 'The idea is to take repeated steps in opposite direction of',
 'the gradient (or approximate gradient) function at current point, ',
 'because this is direction steepest descent. ']
```

Partie III : manipuler un dataset volumineux sur databricks

2) chargement dans un dataframe spark

```
sparkDf=spark.read.load('houses.csv')
sparkDf.printSchema()
```

```
▸ sparkDf: pyspark.sql.dataframe.DataFrame = [size: double, nb_rooms: long ... 3 more fields]
root
 |-- size: double (nullable = true)
 |-- nb_rooms: long (nullable = true)
 |-- garden: long (nullable = true)
 |-- orientation: string (nullable = true)
 |-- price: double (nullable = true)
```

Mémoire consomée

```python
from pyspark.serializers import PickleSerializer, AutoBatchedSerializer
def _to_java_object_rdd(rdd):

    rdd = rdd._reserialize(AutoBatchedSerializer(PickleSerializer()))
    return rdd.ctx._jvm.org.apache.spark.mllib.api.python.SerDe.pythonToJava(rdd._jrdd, True)

JavaObj = _to_java_object_rdd(sparkDf.rdd)

nbytes = sc._jvm.org.apache.spark.util.SizeEstimator.estimate(JavaObj)
print(nbytes)
```

```
4224424344
```

D'après le résultats la mémoire consommée est de 4224424344 bytes ce qui est équivalent à 4.2244 giga

3) Sélection des maisons qui ont plus de 2 chambres dans un nouveau dataframe

```python
df_filtred = sparkDf.filter(sparkDf.nb_rooms > 2)
df_filtred.show(10)
```

▸ (1) Spark Jobs

▸ df_filtred: pyspark.sql.dataframe.DataFrame = [size: double, nb_rooms: long ... 3 more fields]

```
+------------------+--------+------+-----------+------------------+
|              size|nb_rooms|garden|orientation|             price|
+------------------+--------+------+-----------+------------------+
|137.30027526131454|       3|     1|      Ouest|  312824.225304512|
|148.59258955996802|       3|     1|        Est|312356.19881789363|
|192.67056190501847|       3|     1|        Est|323977.13614897546|
|118.48163264872223|       3|     0|       Nord|235242.31960432642|
|180.16142351158868|       3|     1|        Est| 320972.4814387185|
|167.61000386988792|       3|     1|        Sud|  340741.477043372|
| 155.4553500321651|       3|     1|       Nord| 297374.6799684082|
|131.83054115223706|       3|     1|        Est|307008.33851207944|
|200.06973888120532|       3|     1|        Sud| 348664.3313370723|
| 174.8451659379434|       3|     1|        Sud|342632.17409832845|
+------------------+--------+------+-----------+------------------+
only showing top 10 rows
```

4) Calcul de la surface moyenne et du prix moyen

Cmd 9

```python
from pyspark.sql import functions as F
prix_et_surface_moyenne = sparkDf.agg(F.avg('price'),F.avg('size'))
print("le prix moyen du dataset original : ",prix_et_surface_moyenne.show())

```

▸ (2) Spark Jobs

▸ prix_et_surface_moyenne: pyspark.sql.dataframe.DataFrame = [avg(price): double, avg(size): double]

```
+------------------+------------------+
|        avg(price)|         avg(size)|
+------------------+------------------+
|267153.29091201985|150.04028343747882|
+------------------+------------------+
```

5) méthodes envisagées pour réduire la taille du dataframe en mémoire

On peut appliquer une transformation sur la colonne orientation pou réduire la taille des entrées et ainsi réduire la taille en mémoire (on peut par exemple remplacer Ouest par O **)**

Partie IV : Machine learning avec pyspark

2) Normalisation de la colonne size

```
from pyspark.ml.feature import StringIndexer, OneHotEncoder
from pyspark.sql.functions import stddev, mean, col
df_stats = sparkDf.select(
    mean(col('size')).alias('mean'),
    stddev(col('size')).alias('std')
).collect()

mean = df_stats[0]['mean']
std = df_stats[0]['std']
sparkDf_normalized = sparkDf.withColumn('size', (F.col('size') - mean)/std)
sparkDf_normalized.show(5)
sparkDf.show(5)

```

▸ (4) Spark Jobs

▸ sparkDf_normalized: pyspark.sql.dataframe.DataFrame = [size: double, nb_rooms: long ... 3 more fields]

```
+--------------------+--------+------+-----------+------------------+
|                size|nb_rooms|garden|orientation|             price|
+--------------------+--------+------+-----------+------------------+
|   1.355121757813147|       2|     1|        Est|309433.17891415354|
|-0.25540820679048665|       3|     1|      Ouest|  312824.225304512|
| -1.2914434589113652|       1|     0|      Ouest| 201769.8367561039|
|-0.02902297173783479|       3|     1|        Est|312356.19881789363|
|  0.8546401876526548|       3|     1|        Est|323977.13614897546|
+--------------------+--------+------+-----------+------------------+
```

pour normaliser la colonne ils suffit de faire (valeur - moyenne de la colonne) /écart type de la colonn**e**

3) le one hot encoding de la colonne orientation

```
df_train, df_test = sparkDf.randomSplit([0.8, 0.2], seed=91)

string_indexer = StringIndexer(inputCol='orientation', outputCol='orientation'+'Index')
one_hot_encoder = OneHotEncoder(inputCol=string_indexer.getOutputCol(),
                                outputCol='orientation'+'OHE')
```

df_train: pyspark.sql.dataframe.DataFrame = [size: double, nb_rooms: long ... 3 more fields]
df_test: pyspark.sql.dataframe.DataFrame = [size: double, nb_rooms: long ... 3 more fields]

Command took 0.47 seconds -- by em_karech@esi.dz at 6/1/2021, 10:51:47 AM on TP

Cmd 13

```
label_to_index = StringIndexer(inputCol='price', outputCol='label')
```

Command took 0.05 seconds -- by em_karech@esi.dz at 6/1/2021, 10:51:51 AM on TP

Cmd 14

```
string_model_indexer= string_indexer.fit(df_train)
display(string_model_indexer.transform(df_train))
```

(3) Spark Jobs

	size	nb_rooms	garden	orientation	price	orientationIndex
1	0.00008298303231413229	1	1	Nord	30003.734081523573	1
2	0.00019212839873716805	3	1	Ouest	91008.64494749981	2
3	0.0012582259508064908	2	0	Est	17056.584577154976	0
4	0.0025701801760646963	2	1	Ouest	71115.50973101736	2
5	0.004157608722209716	1	0	Nord	-19813.295461324036	1
6	0.004203202703592979	2	0	Nord	188.7477264845802	1
7	0.004958708220783592	2	0	Est	17222.590444377183	0

Showing the first 1000 rows.

Command took 4.02 minutes -- by em_karech@esi.dz at 6/1/2021, 10:52:01 AM on TP

4) Groupage des colonnes dans un seul vecteur

```
from pyspark.ml.feature import VectorAssembler
numeric_cols = ["orientationOHE","size","nb_rooms","garden"]
assembler_inputs = numeric_cols
vec_assembler = VectorAssembler(inputCols=assembler_inputs, outputCol='features')
```

Command took 0.08 seconds -- by em_karech@esi.dz at 6/1/2021, 10:56:14 AM on TP

Cmd 16

```
from pyspark.ml.regression import LinearRegression
lr = LinearRegression(featuresCol = 'features', labelCol='label',regParam=0.3)
```

5) Entraînement du modèle

```
from pyspark.ml import Pipeline

pipeline = Pipeline(stages=[string_indexer, one_hot_encoder, label_to_index, vec_assembler, lr])
pipeline_model = pipeline.fit(df_train)
```

On ne peux pas entrainer ce modèle avec la configuration actuel de databricks on obtient une erreur de type Java heap space **ce qui veut dire qu'on a pas assez de mémoire pour entraîner notre modèle**

```
▸ (3) Spark Jobs

⊞org.apache.spark.SparkException: Job aborted due to stage failure: Task 3 in stage 29.0 failed 1 times, most recent failure: Lost task 3.0 in stag
  e 29.0 (TID 95) (ip-10-172-186-248.us-west-2.compute.internal executor driver): java.lang.OutOfMemoryError: Java heap space

Command took 8.16 minutes -- by em_karech@esi.dz at 6/7/2021, 5:24:12 AM on TP
```

On peut essayer avec un fichier d'une plus petite taille ou augmenter nos ressources

6) Prédiction des valeurs

```
1  df_pred = pipeline_model.transform(df_test)
2  display(df_pred)
```

7) Sauvegarder le résultat dans fichier

df_pred.write.csv(predcitions.csv)