

TP6 Dataframes

Description des bases de données annuelles des accidents corporels de la circulation routière

Intro

Pour les exercices suivant on va utiliser une base de donnees publique du gouvernement qui contient des donnees sur les accidents de la route (<https://www.data.gouv.fr/fr/datasets/base-de-donnees-accidents-corporels-de-la-circulation/> (<https://www.data.gouv.fr/fr/datasets/base-de-donnees-accidents-corporels-de-la-circulation/>))

Nous allons utiliser des dataframes et datasets pour manipuler ces donnees.

Environement

Vous pouvez faire ces excercices dans le spark shell.

Description des donnes

Pour chaque accident corporel (soit un accident survenu sur une voie ouverte à la circulation publique, impliquant au moins un véhicule et ayant fait au moins une victime ayant nécessité des soins), des saisies d'information décrivant l'accident sont effectuées par l'unité des forces de l'ordre (police, gendarmerie, etc.) qui est intervenue sur le lieu de l'accident.

Ces saisies sont rassemblées dans une fiche intitulée bulletin d'analyse des accidents corporels (BAAC). Cela comprend des informations de localisation de l'accident, telles que renseignées ainsi que des informations concernant les caractéristiques de l'accident et son lieu, les véhicules impliqués et leurs victimes.

Les bases de données de 2005 à 2015 sont désormais annuelles et composées de 4 fichiers (Caractéristiques – Lieux – Véhicules – Usagers) au format csv.

La description des differents fichiers se trouve ici: https://www.data.gouv.fr/s/resources/base-de-donnees-accidents-corporels-de-la-circulation/20160926-173908/Description_des_bases_de_donnees_ONISR_-_Annees_2005_a_2015.pdf (https://www.data.gouv.fr/s/resources/base-de-donnees-accidents-corporels-de-la-circulation/20160926-173908/Description_des_bases_de_donnees_ONISR_-_Annees_2005_a_2015.pdf).

Documentation

Documentation dataframes: <http://spark.apache.org/docs/latest/sql-programming-guide.html#creating-dataframes> (<http://spark.apache.org/docs/latest/sql-programming-guide.html#creating-dataframes>)

Documentation API scala:

<http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset> (<http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset>).

Dans les exercices suivants remplacez les *TODO* par le code nécessaire

Lecture et exploration des donnees CSV

```
import org.apache.spark.sql.SparkSession

def TODO = ??? // ceci est juste un marquer d'une valeur que vous devez remplacer de

// on cree une session Spark avec un nom particulier pour la retrouver plus facilement
val mySession = SparkSession
    .builder()
    .config(new SparkConf()
        .setAppName("Accidents")) //rajouter comme parametre le nom de votre appl.
    .getOrCreate()
```

Took: 3 seconds 695 milliseconds, at 2019-12-5 10:46

```
// TODO: verifier/modifier le chemin vers les fichiers de donnees:
val path="notebooks/telecom2016/data/"
```

Took: 3 seconds 493 milliseconds, at 2019-12-5 10:49

```
// Lecture des donnees dans des DataFrames en utilisant la premier ligne du fichier
val lieux /*:DataFrame*/ = mySession.read.option("header", "true").format("com.databricks.spark.csv")
    .load(path + "/lieux_2015.csv")
    .cache
val caracteristiques = mySession.read.option("header", "true").format("com.databricks.spark.csv")
    .load(path + "/caracteristiques_2015.csv")
    .cache
val usagers = mySession.read.option("header", "true").format("com.databricks.spark.csv")
    .load(path + "/usagers_2015.csv")
    .cache
val vehicules = mySession.read.option("header", "true").format("com.databricks.spark.csv")
    .load(path + "/vehicules_2015.csv")
    .cache
```

Took: 7 seconds 545 milliseconds, at 2019-12-5 10:54

```
// Un DataFrame est un DataSet[Row] ou par default chaque ligne est une "String"
lieux.printSchema
```

Took: 5 seconds 177 milliseconds, at 2019-12-5 10:54

```
// afficher le schema des 3 autres Dataframes
caracteristiques.printSchema
usagers.printSchema
vehicules.printSchema
```

Took: 5 seconds 258 milliseconds, at 2019-12-5 10:56

```
// on peut utiliser la methode show pour afficher le contenu d'un DataFrame
usagers.show(4) // affiche 4 lignes du dataframe usagers
```

Took: 13 seconds 598 milliseconds, at 2019-12-5 10:58

On peut suivre dans le SparkUI le plan d'execution de l'application de la methode show : <http://localhost:4040> (<http://localhost:4040>) ainsi que la mise en cache des dataframes: <http://localhost:4040/storage/> (<http://localhost:4040/storage/>). Combien de lignes on a dans notre jeu de donnees ?

```
print {s"Le jeu de donnees contiens ${lieux.count} accidents qui ont implique ${veh:
print {s"Le nombre des caracteristiques des accidents: ${caracteristiques.count}"}
```

Took: 5 seconds 117 milliseconds, at 2019-12-5 11:21

```
//Afficher le nombre des victimes groupees par la gravite de la blessure. Utiliser
val victimes = usagers.groupBy("grav").count
victimes.show // pour afficher le contenu
```

Took: 28 seconds 989 milliseconds, at 2019-12-5 11:11

```
// Hmm, les valeurs numeriques ne sont pas trop comprehensible, nous pouvons les de
victimes.map{
  row => {
    val mapping = Map(
      "1" -> "Indemne",
      "2" -> "Tué",
      "3" -> "Blessé hospitalisé",
      "4" -> "Blessé léger"
    )
    mapping(row.getAs[String]("grav")) + " : " + row.getAs[String]("count")
  }
}.show(10, false)
```

Took: 46 seconds 691 milliseconds, at 2019-12-5 11:23

1. Quel est le nombre des blesses de la route en 2015?

```
usagers.filter($"grav" === 3).count // filtrer sur la gravite === 3
```

27717

Took: 11 seconds 599 milliseconds, at 2019-12-5 11:30

2. Quelle est la repartition des victimes par age ?

Quel est le type de la colonne "an_nais" du dataframe "_usagers" ? Comment peut-on calculer la date de naissance?

```
import org.apache.spark.sql.types._

// d'abord on doit creer une nouvelle colonne an_nais_int = pour convertir la colonne an_nais en int
val usagers1 = usagers.withColumn("an_nais_int", col("an_nais").cast(IntegerType))
```

Took: 6 seconds 113 milliseconds, at 2019-12-5 11:40

```
// puis on defini une User Defined Function qui calcule l'age a partir de la date de naissance
val computeAge = udf {(an_nais: Int) => 2016 - an_nais}

//et on rajoute une nouvelle colonne qui sera peulee avec le resultat de l'application de la fonction
val usagersAge = usagers1.withColumn("age", computeAge($"an_nais_int"))

usagersAge.show()
```

Took: 9 seconds 714 milliseconds, at 2019-12-5 11:39

```
// Afficher la repartition des victimes par age, trie par l'age des victimes
usagersAge.groupBy("age").count().sort(desc("count")).show(10) //affichez uniquement les 10 premiers
```

Took: 35 seconds 330 milliseconds, at 2019-12-5 11:41

Utiliser le SparkUI pour afficher le plan d'exécution de la requête précédente <http://localhost:4040> (<http://localhost:4040>).

4. Quelles sont les caractéristiques de 3 accidents le plus meurtriers de 2015 (date, nb véhicules, conditions météo)

```
val accidents = usagers
  .filter($"grav" === 2) // on filtre les victimes
  .groupBy("Num_Acc") // on groupe sur l'id de l'accident
  .count() // on compte le nombre des victimes
  .sort(desc("count")) // on trie par le nombre des victimes
  .limit(3) // on garde uniquement les 3 accidents les plus meurtriers
  .withColumnRenamed("count", "nb_victimes")
  .join(caracteristiques, usagers("Num_acc") === caracteristiques("Num_Acc")) // on joint les caractéristiques
  .join(lieux, usagers("Num_acc") === lieux("Num_Acc")) // on fait la jointure avec les lieux
  .join(vehicules, usagers("Num_Acc") === vehicules("Num_Acc")) // on fait la jointure avec les véhicules
  .groupBy(caracteristiques("an"), caracteristiques("mois"), caracteristiques("jour"))
  .count().withColumnRenamed("count", "nb_vehicules") // on compte le nombre des véhicules par jour/mois/an
accidents.show()
```

Took: 1 minute 13 seconds 646 milliseconds, at 2019-12-5 11:44

Datasets (typer les DataFrames)

Dans les exercices précédents on a manipulé des DataFrames = ensembles de lignes de chaînes de caractères. Les datasets nous permettent de rajouter de types aux colonnes pour les manipuler plus facilement.

```
// On commence par definir une case class qui va definir le types des lignes de not.
case class Caracteristiques(NumAcc: String, latitude: Double, longitude:Double,adr:
```

Took: 6 seconds 706 milliseconds, at 2019-12-5 11:45

```
//On doit transformer chaque ligne (Row) dans un objet Caracteristiques
// on defini une fonction de mapping qui prends un objet de type Row et retourne un
def mapRow(r:Row): Caracteristiques = {
  Caracteristiques(
    r.getAs[String]("Num_Acc"),
    r.getAs[String]("lat").toDouble/100000,
    r.getAs[String]("long").toDouble/100000,
    r.getAs[String]("adr"),
    r.getAs[String]("col").toInt)
}
```

Took: 8 seconds 83 milliseconds, at 2019-12-5 11:47

```
// On transforme le dataframe caracteristique (Dataset[Row]) dans un DataSet caracti
val caractDS/*:Dataset[Caracteristiques]*/ = caracteristiques.filter(
  x => {
    try{
      mapRow(x)
      true
    }catch {
      case e: Exception => false
    }
  }).map{ mapRow }
```

Took: 11 seconds 260 milliseconds, at 2019-12-5 11:48

```
// On definit une methode qui pour une Caracteristique retourne true si la Caracteri
def nearPlace(c:Caracteristiques): Boolean = {
  ( Math.abs(c.latitude - 48.71)< 0.06 && Math.abs(c.longitude - 2.24) < 0.06) //r
}
nearPlace(Caracteristiques("test",48.71,2.24," my home", 0))// on test la methode en
```

true

Took: 10 seconds 604 milliseconds, at 2019-12-5 11:50

```
// Trouver un accident proche
val points = caractDS.filter(nearPlace _).take(1)
```

Took: 15 seconds 264 milliseconds, at 2019-12-5 11:52

5. [Optional] Exploration des donnees avec spark-notebook

```
// Trouver la liste des accidents proche de ma maison
```

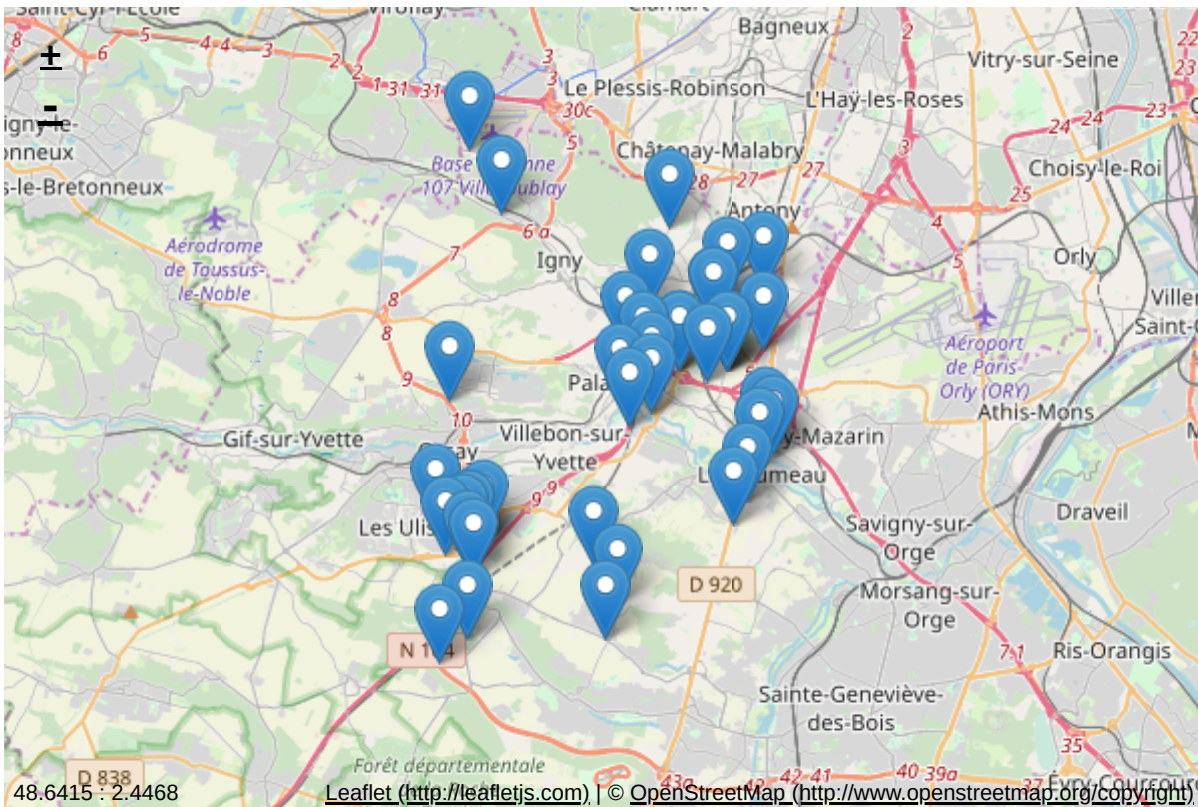
```
val points = caractDS.filter(nearPlace(_)).map(c=> (c.latitude,c.longitude,c.colType
```

Took: 19 seconds 903 milliseconds, at 2019-12-5 11:53

```
// Afficher les accidents proche sur une carte via un widget du notebook
```

```
val w = GeoPointsChart(points, maxPoints=500)
```

```
w
```



Took: 12 seconds 259 milliseconds, at 2019-12-5 11:53

```
// Via le widget PivotChart on peut faire des analyses comme dans un tableur
// TODO: faites des aggregations/heatmaps/charts en utilisant le widget PivotChart
PivotChart(caracteristiques,maxPoints=300)
```

300 or more entries total (Warning: showing only first 300 rows)
show/hide options

Heatmap ▼

Count ▼


mois ▼

agg ▼ lum ▼ hrnm ▼ gps ▼ jour ▼ long ▼ col ▼ int ▼ an

dep ▼

	dep	590	620	Totals
mois				
1		18	8	26
2		7	6	13
3		14	5	19
4		17	8	25
5		19	10	29
6		12	15	27
7		22	10	32
8		13	8	21
9		10	10	20
10		18	12	30
11		27	7	34
12		15	9	24
Totals		192	108	300

Took: 668 milliseconds, at 2016-12-12 20:49



Build: | **buildTime**-*Mon Nov 21 09:45:10 UTC 2016* | **formattedShaVersion**-0.7.0-
c955e71d0204599035f603109527e679aa3bd570 | **sbtVersion**-0.13.8 | **scalaVersion**-2.10.6 | **sparkNotebookVersion**-
0.7.0 | **hadoopVersion**-2.7.2 | **jets3tVersion**-0.7.1 | **jlineDef**-(*org.scala-lang,2.10.6*) | **sparkVersion**-2.0.2 | **withHive**-true |.