Publié par Jonathan Raffre et Aurore De Amaral

BigData Data data engineering Data Science scala tools

Il y a 2 ans -Temps de lecture 19 minutes

Exploration de données en Scala

Aujourd'hui, c'est la première journée où Henri va faire de la data. Henri est développeur back Scala à la base, et aimerait contribuer à la partie data de son projet.

Or, la plupart des outils sont en Python. On peut certes trouver rapidement des *How-To* sur Internet traitant de la data en Python, mais Henri, lui, aime bien le Scala. Peut-il rapidement explorer les données métier avec son langage favori ?

Ce matin, en prenant son café, Nina, ingénieur de métier, dans un contexte data, aimerait rendre le partage de code plus efficace entre les data-scientists et elle. Or, Nina utilise Spark en Scala et ses collègues R. Cela semble plus compliqué de communiquer et transformer les règles métiers... Elle aimerait donner des outils avec le langage qu'elle utilise, pour faciliter la coopération.

Il était une fois la data

Il faut savoir que ces exemples sont fictifs, mais notre idée générale est de se demander s'il est possible de faire rapidement de l'exploration de données. L'exploration de données est une méthode préliminaire à tout projet data science. Il est effectué par un professionnel de la data, afin de vérifier l'état des données, valider des suppositions ou inférer des corrélations. Ainsi, donner une visualisation rapide au métier. Le but ici est de tester s'il est possible de le faire avec du Scala, sans avoir à réfléchir à la phase de compilation ou à comment lire du CSV tout en transformant ces données en tableaux ou matrices.

L'exploration de données se fait en plusieurs étapes et se charge de :

- Charger/Transformer les données.
- Tenter d'extraire ou créer des *features* (caractéristiques de la donnée).

- Visualiser ces *features*.

Nous allons passer en revue toutes ces demandes data avec des outils utilisables en Scala, à travers deux exemples :

- Le premier, sur des données en table, sera traité dans cet article ;
- Le second, sur des données continues (du texte plus précisément), sera abordé dans un prochain article.

Les deux articles présentent plusieurs exemples de code, qui nous l'espérons, couvriront vos besoins.

Premier exemple

Airbnb a fourni à la communauté il y a quelques mois des données sur l'enregistrement et la réservation, permettant de tester des méthodes afin de prédire quelle sera la prochaine destination de l'utilisateur.

Nous allons explorer ces données (que vous pouvez retrouver sur Kaggle), en particulier une table appelée user qui contient des informations sur les utilisateurs inscrits d'Airbnb ; et ce, grâce au langage Scala, tout simplement.

Mais avant cela, voyons voir quels sont les outils qu'Henri pourrait utiliser (aujourd'hui, c'est lui qui s'y colle).

Avec quels outils ?

Henri a entendu plusieurs fois des noms de bibliothèques comme *NumPy*, *Pandas*, *Scikit Learn,* sans trop savoir ce que chacune d'entre elle fournit. Essayons donc de définir ce que ces outils fournissent dans le monde Python, et comment y répondre de manière équivalente en Scala, lequel possède déjà quelques *frameworks* et bibliothèques autour de la data, contribués par la communauté.

- *NumPy* est une bibliothèque de gestion de données matricielles (organisées de manière tabulaire), laquelle implémente des fonctionnalités principalement mathématiques de transformation de ces données (filtre passe-bas, approximations de données d'une série) ou de calcul de fonctions caractéristiques de ces données (moyenne, médiane). *Breeze*, en Scala, cherche à fournir une équivalence fonctionnelle à celui-ci, il est donc possible de le remplacer aisément (équivalences *NumPy* vs *Breeze*).
- *Pandas* repose sur NumPy et fournit principalement une abstraction de la donnée nommée le *DataFrame*, laquelle pouvant travailler sur des données autres que des nombres. Il permet ainsi la sélection de données dans un tableau par critère, quelques opérations ensemblistes (jointures,

fusion de tables, etc.) et fournit des capacités de visualisation. Son équivalent Scala est *Apache Spark* (qui couvre plus de fonctionnalités que celles de Pandas).
- *Scikit Learn* fournit un ensemble d'outils facilitant l'extraction de *features*, lesquelles sont aussi utilisables dans un contexte de *Machine Learning*. Il fonctionne donc en complément d'outils comme *NumPy* et peut être remplacé dans une certaine mesure par *MLlib*, un sous-ensemble de fonctionnalités d'Apache Spark, lequel peut être complété par *Featran* pour assister dans l'extraction de *features* particulières.

Il existe, en Scala, d'autres bibliothèques qui ont été contribuées par Spotify comme *SCIO*, un *framework* abstrayant les APIs d'*Apache Beam* et *Google Cloud Dataflow* et visant à fournir une API similaire à *Apache Spark*.

Dans la suite, nous avons fait le choix d'utiliser Spark car elle couvre l'ensemble du spectre qui nous intéresse dans un premier temps : l'exploration des données. Ce choix est aussi en cohérence avec la tendance actuelle d'utiliser Spark en production pour déployer de nouvelles fonctionnalités reposant sur des données.

Analyse de la donnée

Spark permet de charger des données sous plusieurs formats. Ici, on s'intéresse au CSV, qui représente une donnée tabulaire.

```
import org.apache.spark.sql.SparkSession
val sc: SparkContext
val spark = SparkSession.builder
  .master("local")
  .appName("data-exploration")
  .getOrCreate()

val users = spark.read.format("csv")
  .option("header", "true")
  .option("inferSchema", "true")
  .load("train_users_2.csv")
```

Dans ce premier exemple de code, on charge un fichier CSV qui contient en première ligne l'en-tête (header) avec le nom des colonnes et on demande à Spark d'inférer les types de données.

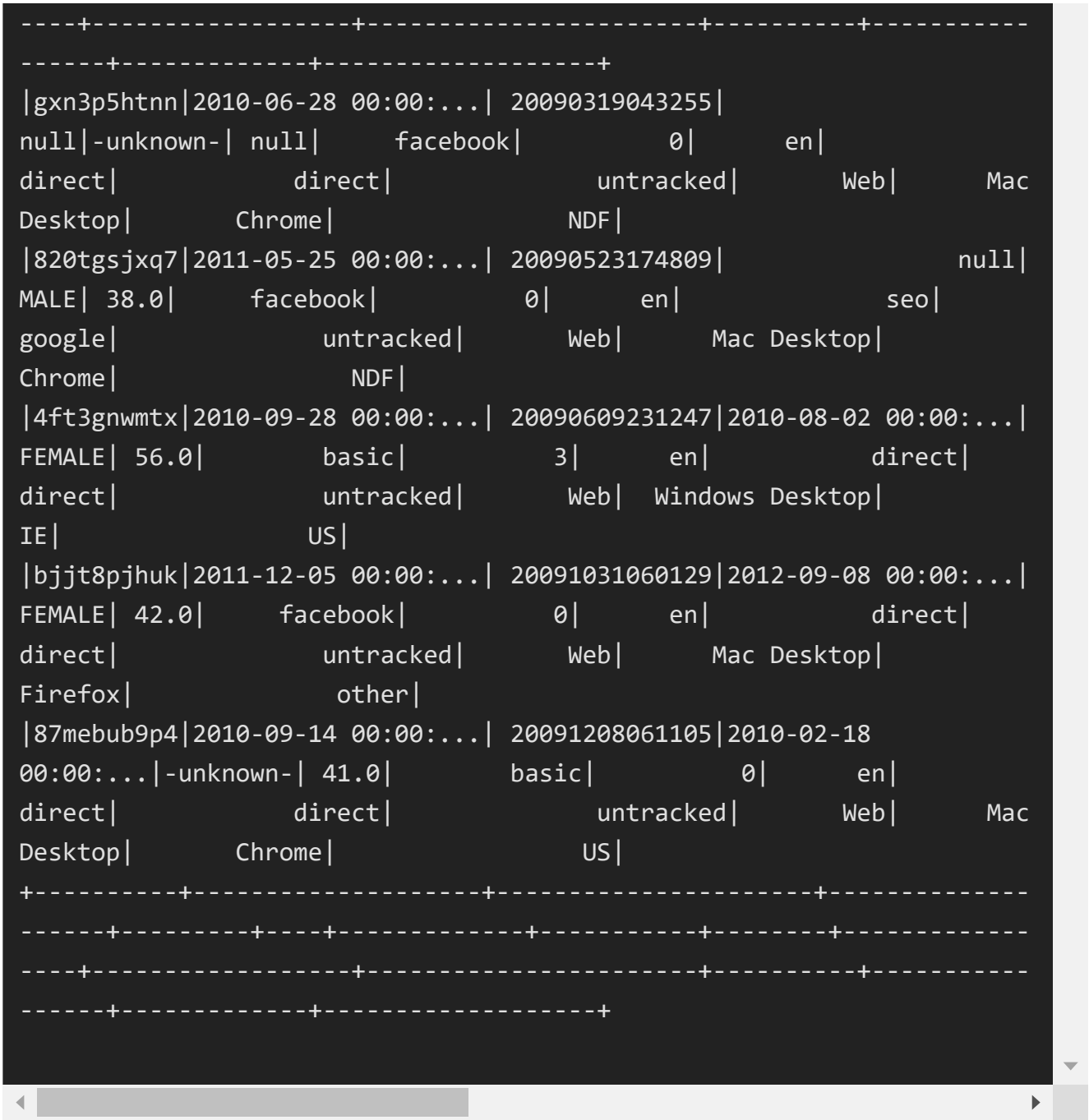
Pour vérifier l'inférence de types, on peut afficher le schéma du DataFrame en sortie. Le DataFrame est une forme structurée dans Spark, que l'on va toujours utiliser.

À noter, après Spark 2.0, on utilise Dataset en Scala et SparkSession comme l'exemple ci-dessus, les versions précédentes étant assez différentes. Plus d'information sur la documentation officielle.

```
users.printSchema

root
  |-- id: string (nullable = true)
  |-- date_account_created: timestamp (nullable = true)
  |-- timestamp_first_active: long (nullable = true)
  |-- date_first_booking: timestamp (nullable = true)
  |-- gender: string (nullable = true)
  |-- age: double (nullable = true)
  |-- signup_method: string (nullable = true)
  |-- signup_flow: integer (nullable = true)
  |-- language: string (nullable = true)
  |-- affiliate_channel: string (nullable = true)
  |-- affiliate_provider: string (nullable = true)
  |-- first_affiliate_tracked: string (nullable = true)
  |-- signup_app: string (nullable = true)
  |-- first_device_type: string (nullable = true)
  |-- first_browser: string (nullable = true)
  |-- country_destination: string (nullable = true)
```

Affichons maintenant les premières lignes.

```
users.show(5)
+----------+--------------------+----------------------+-------------
------+---------+----+-------------+-----------+--------+-------------
----+------------------+----------------------+----------+-----------
------+-------------+-------------------+
| id|date_account_created|timestamp_first_active|  date_first_booking|
gender|
age|signup_method|signup_flow|language|affiliate_channel|affiliate_provi

+----------+--------------------+----------------------+-------------
------+---------+----+-------------+-----------+--------+-------------
```



```
----+------------------+----------------------+----------+-----------
------+-------------+------------------+
|gxn3p5htnn|2010-06-28 00:00:...| 20090319043255|
null|-unknown-| null|     facebook|          0|      en|
direct|            direct|              untracked|       Web|      Mac
Desktop|       Chrome|                NDF|
|820tgsjxq7|2011-05-25 00:00:...| 20090523174809|               null|
MALE| 38.0|     facebook|          0|      en|              seo|
google|              untracked|       Web|      Mac Desktop|
Chrome|                NDF|
|4ft3gnwmtx|2010-09-28 00:00:...| 20090609231247|2010-08-02 00:00:...|
FEMALE| 56.0|        basic|          3|      en|           direct|
direct|              untracked|       Web|  Windows Desktop|
IE|                US|
|bjjt8pjhuk|2011-12-05 00:00:...| 20091031060129|2012-09-08 00:00:...|
FEMALE| 42.0|     facebook|          0|      en|           direct|
direct|              untracked|       Web|      Mac Desktop|
Firefox|              other|
|87mebub9p4|2010-09-14 00:00:...| 20091208061105|2010-02-18
00:00:...|-unknown-| 41.0|        basic|          0|      en|
direct|            direct|              untracked|       Web|      Mac
Desktop|       Chrome|                US|
+----------+--------------------+----------------------+-------------
------+---------+----+-------------+-----------+--------+-------------
----+------------------+----------------------+----------+-----------
------+-------------+------------------+
```

Henri voit des dates qui ont été présentées sous cette forme « 2010-06-28 00:00:00.0 » pour deux colonnes : date_first_booking et date_account_created.

Et une autre colonne suivant un autre format en nombre celui-ci : timestamp_first_active.

Pour plus de lisibilité, nous aimerions formater ces trois colonnes sous la forme « annee-mois-jour ». Les deux premières ont été correctement reconnues par Spark comme étant des dates, mais pas la dernière.

Utilisons une UDF (User Defined Function) qui va être lue par l'API pour adapter cette dernière colonne.

```
import java.time.format.DateTimeFormatter

val to_date = udf((date: Long, numberFormat: String, destFormat:
String) => {
  Option(date).map(number => {
  val date = number.toString
  val formatFrom = DateTimeFormatter.ofPattern(numberFormat)
  val formatTo = DateTimeFormatter.ofPattern(destFormat)
  formatTo.format(formatFrom.parse(date))
 })
})
```

Une UDF est une fonction appliquée sur une colonne d'un DataFrame qui renvoie une colonne avec la transformation appliquée. La fonction d'ordre supérieure ci-dessus va prendre en entrée une date sous forme de Long, un format d'entrée et un format de sortie, ce qui permettra de la réutiliser dans d'autres domaines.

Pour les autres, nous utiliserons une fonction standard de Spark.
Sélectionner les colonnes qui nous intéressent permet de voir si le résultat correspond. Nous allons utiliser select et filter peu après lors du nettoyage de données.

```
val destFormat = "yyyy-MM-dd"
val convertedDates =
users.select(date_format(col("date_first_booking"), destFormat),

to_date(col("timestamp_first_active"), lit("yyyyMMddHHmmss"),
lit(destFormat)),

date_format(col("date_account_created"), destFormat))

+------------------+---------------------+--------------------+
|date_first_booking|timestamp_first_active|date_account_created|
+------------------+---------------------+--------------------+
|              null|           2009-03-19|          2010-06-28|
```

```
|              null|            2009-05-23|          2011-05-25|
|        2010-08-02|            2009-06-09|          2010-09-28|
|        2012-09-08|            2009-10-31|          2011-12-05|
|        2010-02-18|            2009-12-08|          2010-09-14|
+------------------+----------------------+--------------------+
```

Une des tâches de l'exploration de données est de constater des valeurs nulles, manquantes ou aberrantes. Pour les valeurs nulles et manquantes, on peut filtrer sur des colonnes et voir si certaines lignes en possèdent.

La commande filter de Spark prend des conditions sur des colonnes. On peut lui passer des requêtes en SQL, avec AND et OR par exemple.

Pour les valeurs aberrantes, nous pouvons afficher toutes les valeurs distinctes d'une colonne, ou encore faire un describe du DataFrame, équivalent un peu modeste du pandas.describe.

Voyons voir les dates, les âges, les genres et les langues.

```
// Pour récupérer les valeurs aberrantes et nulles
users.filter("age is NULL or date_first_booking is NULL or
date_account_created is NULL or timestamp_first_active is NULL or
language is NULL ")

// Pour sélectionner des valeurs distinctes de colonnes
users.select("language").distinct
users.select("gender").distinct

// Pour afficher des métriques
users.describe()

+-------+----------+----------------------+---------+-----------------
-+-------------+------------------+--------+-----------------+--------
----------+----------------------+----------+-----------------+------
-------+-------------------+
|summary|        id|timestamp_first_active|   gender|
age|signup_method|
signup_flow|language|affiliate_channel|affiliate_provider|first_affiliat
```

```
+-------+----------+----------------------+---------+-----------------
-+-------------+-----------------+--------+----------------+--------
----------+----------------------+----------+----------------+------
-------+------------------+
| count |    213451|               213451|   213451|
125461|       213451|           213451|  213451|           213451|
213451|                207386|    213451|           213451|
213451|             213451|
| mean  |      null|  2.013085041736729...|     null|
49.66833517985669|         null|3.2673868944160485|    null|
null|             null|                  null|      null|
null|         null|               null|
| stddev|      null|   9.2537170465515 46E9|
null|155.66661183021515|         null|  7.63770686943509|    null|
null|             null|                  null|      null|
null|         null|               null|
|    min|00023iyk9l|        20090319043255|-unknown-|
1.0|        basic|                 0|      ca|             api|
baidu|               linked|   Android|    Android Phone|    -
unknown-|                AU|
|    max|zzzlylp57e|        20140630235824|    OTHER|
2014.0|       google|               25|      zh|             seo|
yandex|             untracked|       iOS|           iPhone|
wOSBrowser|             other|
+-------+----------+----------------------+---------+-----------------
-+-------------+-----------------+--------+----------------+--------
----------+----------------------+----------+----------------+------
-------+------------------+;
```

Nettoyage de la donnée

Avec ces trois méthodes, nous avons déjà beaucoup d'informations et de données particulières :

- Des personnes dont le genre n'est pas connu et noté -unknown-
- Des âges bien en-dessous de la politique d'inscription d'Airbnb, ou bien trop grandes pour être possibles...
- Des âges ou des dates nulles.

On va devoir remplacer ces données et les filtrer pour avoir un DataFrame sur lequel il sera possible de faire de l'exploration et de la sélection de feature. Henri est motivé !

Il va placer un âge minimal et maximal (18-100), supprimer les utilisateurs dont le genre est -unknown-, et enfin supprimer les lignes qui n'ont pas de country_destination défini.

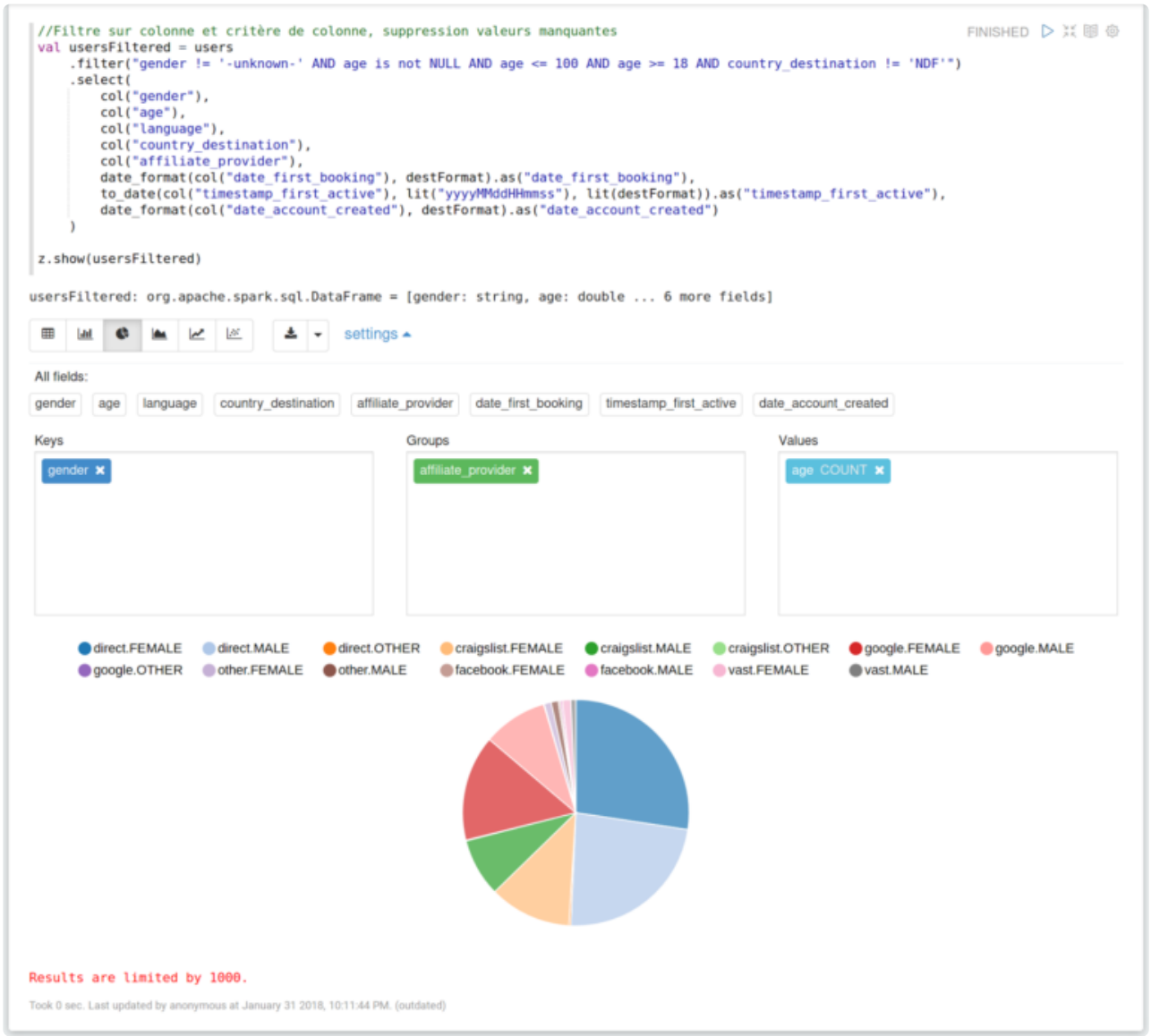
```
val usersFiltered = users
  .filter("gender != '-unknown-' AND age is not NULL AND age <= 100
AND age >= 18 AND country_destination != 'NDF'")
  .select(
      col("gender"),
      col("age"),
      col("language"),
      col("country_destination"),
      col("signup_method"),
      col("affiliate_provider"),
      date_format(col("date_first_booking"),
destFormat).as("date_first_booking"),
      to_date(col("timestamp_first_active"), lit("yyyyMMddHHmmss"),
lit(destFormat)).as("timestamp_first_active"),
      date_format(col("date_account_created"),
destFormat).as("date_account_created")
 )

-------------------+------------+------------------+--------------
---+------------------+----------------------+--------------------+
|gender|
age|language|signup_method|country_destination|affiliate_provider|date_f

+------+----+--------+------------+------------------+-------------
----+------------------+----------------------+--------------------+
|FEMALE|56.0|      en|       basic|                US|
direct|       2010-08-02|           2009-06-09|         2010-09-28|
|FEMALE|42.0|      en|    facebook|             other|
direct|       2012-09-08|           2009-10-31|         2011-12-05|
|FEMALE|46.0|      en|       basic|                US|
craigslist|       2010-01-05|           2010-01-02|          2010-
01-02|
|FEMALE|47.0|      en|       basic|                US|
direct|       2010-01-13|           2010-01-03|         2010-01-03|
|FEMALE|50.0|      en|       basic|                US|
craigslist|       2010-07-29|           2010-01-04|          2010-
01-04|
+------+----+--------+------------+------------------+-------------
----+------------------+----------------------+--------------------+
```

Visualisation de la donnée

Le résultat est sous la forme d'un tableau, mais l'idée serait d'avoir une version plus « *user friendly* » pour le présenter à ses collègues.

Pour cela, le moyen le plus convivial serait d'utiliser un notebook, c'est à dire, une interface graphique permettant de facilement lancer ses codes Scala et d'obtenir des histogrammes associés.

Henri voit souvent ses collègues data-scientist présenter leurs résultats avec Jupyter, un système de *notebook* permettant cela à l'aide de la librairie de graphing *MatPlotLib*, mais son interopérabilité avec Scala est moins avancée que celle avec Python. Son choix se porte donc sur Zeppelin, créé par la fondation Apache.



On peut aussi faire des visualisations textuelles, certes moins tape-à-l'oeil mais qui ont le mérite d'être utilisables sur un terminal.

Voici un exemple de tri sur colonne par ordre alphabétique.

```
users.sort("signup_method").select("age")
users.sort(col("gender").desc).select("age")
```

Extraction de features

Maintenant qu'Henri a un tableau sur lequel il peut travailler pour sélectionner des features et en créer, nous allons en voir quelques unes ! Et même en visualiser certaines avec des histogrammes.

Henri aimerait travailler sur les âges, et voudrait définir des classes d'âges. C'est-à-dire des intervalles d'âges, de 0 à 10 ans, 10 à 20 ans, etc.

```
val range10 = udf((age: Float) =&gt; { (age / 10).toInt * 10 })

val usersFilteredWithRange = usersFiltered.withColumn("ageRange",
range10(col("age")))

+------+----+--------+-------------+-------------------+-------------
----+------------------+----------------------+--------------------+--
------+
|gender|
age|language|signup_method|country_destination|affiliate_provider|date_f

+------+----+--------+-------------+-------------------+-------------
----+------------------+----------------------+--------------------+--
------+
|FEMALE|56.0|      en|        basic|                 US|
direct|        2010-08-02|            2009-06-09|          2010-09-28|
50|
|FEMALE|42.0|      en|     facebook|              other|
direct|        2012-09-08|            2009-10-31|          2011-12-05|
40|
|FEMALE|46.0|      en|        basic|                 US|
craigslist|        2010-01-05|            2010-01-02|          2010-
01-02|      40|
```

```
|FEMALE|47.0|      en|        basic|                 US|
direct|        2010-01-13|            2010-01-03|          2010-01-03|
40|
|FEMALE|50.0|      en|        basic|                 US|
craigslist|        2010-07-29|            2010-01-04|          2010-
01-04|      50|
+------+----+--------+-------------+-------------------+-------------
----+------------------+----------------------+--------------------+--
------+
```

Ne pas oublier en Scala que les val sont des variables immutables par défaut. Il faut redéfinir une variable si l'on veut rajouter une colonne.

Nous venons ici de faire du *feature engineering* en créant cette nouvelle colonne. Nous partirons de ces intervalles pour la suite.

Avec ce tableau, il nous manque encore quelques informations. Nous aimerions savoir :

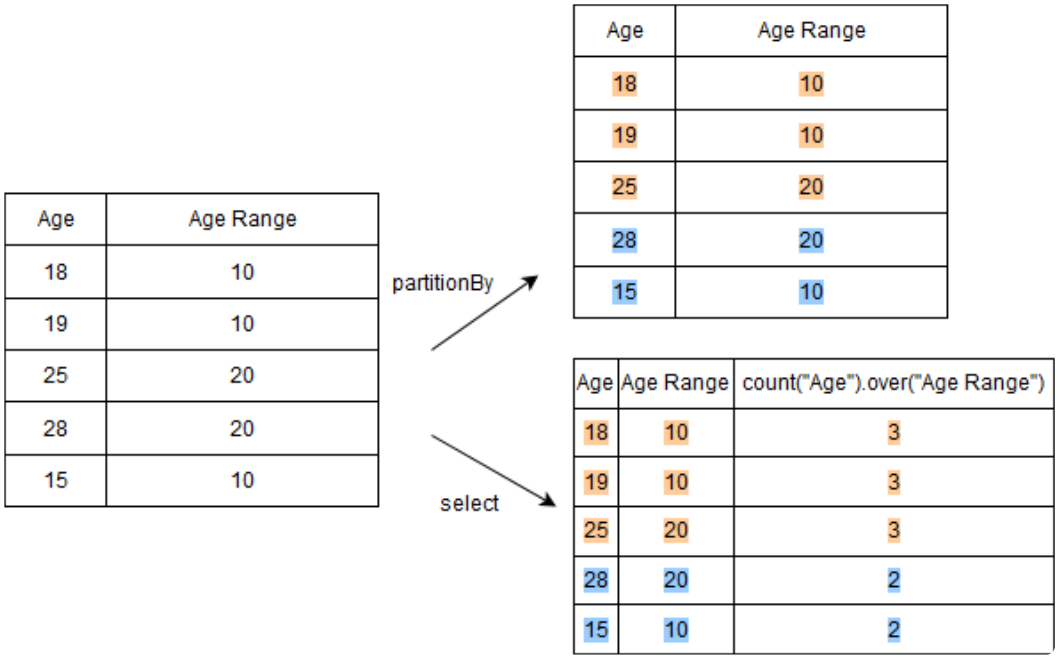
- Le nombre de personnes qui utilisent Airbnb par classe d'âge, ainsi que la moyenne et le mode pour chaque classe d'âge.
- Le nombre de personnes qui utilisent Airbnb par pays de destination.
- Le nombre de personnes par genre regroupées par période de première réservation sur le site.
- Enfin, essayer de trouver une corrélation entre la langue de la personne et le fournisseur d'authentification (dénommé par la colonne *provider)* utilisé lors de sa première connexion.

Répartition par classes d'âge

La répartition par classe d'âge permet de faire du *feature engineering* sur des données qui peuvent être regroupées selon certains critères.

Pour chaque classe correspondant à un intervalle d'âge, nous voulons connaître l'âge maximal, l'âge moyen et le mode de l'âge (l'âge ayant le nombre d'occurences le plus élevé).

Pour partitionner un DataFrame, nous pouvons utiliser la fonction Window et les *Window functions*. La fonction partitionBy permet de définir un critère selon lequel nous allons découper la donnée en groupes, auquel peut s'ajouter une définition de fenêtre glissante (les 2 lignes précédentes et suivantes, par exemple).



Une fois cette fonction définie, on l'utilise dans nos select pour appliquer des fonctions sur ces groupes de données, par exemple l'âge maximum dans ce groupe.

Nous utilisons la méthode distinct pour éviter les duplications de lignes qui ont le même âge et le même intervalle d'âge.

Nous pouvons quand même avoir des lignes différentes correspondant aux count (un pour chaque âge de 18 à 100). Cependant, nous ne voulons qu'une seule ligne par intervalle, qui correspond au nombre maximal de personnes dans un certain intervalle.

Pour cela, nous utiliserons filter associé à une colonne qui utilise Window mais qui ne sera pas affichée à la fin (la colonne CountMax).

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

val ageW = Window.partitionBy("age")
 val ageRangeW = Window.partitionBy("ageRange")

val usersCheckPoint = usersWithRange
 .select(
 count("age").over(ageW).as("count"),
 round(mean("age").over(ageRangeW)).as("mean"),
 col("age"),
 col("ageRange")
 )
 .distinct
 .withColumn("CountMax", max("count").over(ageRangeW))
```

```
// On montre l'intervalle 10-20 car c'est un exemple significatif
 // C'est la seconde ligne qui sera filtrée
 usersCheckPoint.filter("ageRange == 10").show

 +-----+----+----+--------+--------+
 |count| mean| age|ageRange|CountMax|
 +-----+----+----+--------+--------+
 |  172| 19.0|18.0|      10|     296|
 |  296| 19.0|19.0|      10|     296|
 +-----+----+----+--------+--------+

usersCheckPoint.orderBy("ageRange")
   .filter(
     col("CountMax") === col("count")
   )
   .select(
      col("ageRange"),
      col("count"),
      col("age").as("mode"),
      col("mean")
 )

 +--------+-----+-----+-----+
 |ageRange|count| mode| mean|
 +--------+-----+-----+-----+
 |      10|  296| 19.0| 19.0|
 |      20| 2890| 29.0| 26.0|
 |      30| 3094| 30.0| 34.0|
 |      40| 1228| 40.0| 44.0|
 |      50|  549| 51.0| 54.0|
 |      60|  301| 60.0| 64.0|
 |      70|   98| 70.0| 73.0|
 |      80|   15| 80.0| 84.0|
 |      90|   14| 95.0| 94.0|
 |     100|    8|100.0|100.0|
 +--------+-----+-----+-----+
```

Distribution par pays

Pour une information en particulier, nous utilisons groupBy, sur une ou plusieurs colonnes, ici sur les pays de destination. Le fonctionnement de groupBy est le même que partitionBy, à savoir regrouper les données par valeur dans les colonnes passées en argument, mais chaque groupe de lignes doit être ensuite résumé en une seule ligne par le biais de la fonction agg. Le résultat ne contient ainsi que les données décrites dans l'agrégation.

Age	Country
18	France
19	Allemagne
25	France
28	Italie
15	France

groupBy + agg →

Country	Count("Age")
France	3
Allemagne	1
Italie	1

Pour chaque pays, nous allons compter le nombre d'utilisateurs (count et max sont les méthodes les plus utilisées à l'intérieur d'une agrégation, une liste exhaustive est disponible dans la documentation).

```
usersFilteredWithRange
 .groupBy("country_destination")
 .agg(count("age").as("Count"))
```

Transposition du flux de période de réservation pour les genres

La transposition permet d'avoir une vision en flux de notre donnée. Le but est de transformer nos lignes en colonnes, tout en les fusionnant.

Dans notre exemple, nous voudrions voir, par période, le nombre de personnes, et leur genre en particulier, qui ont pris leur première réservation. Depuis Spark 1.6, la fonction pivot appliquée à un groupBy permet de réaliser cette transformation.

```
val groupByGenderStereotype = usersWithRange
 .withColumn("first_booking_period",
date_format(col("date_first_booking"), "yyyy-MM"))
 .groupBy("first_booking_period", "gender")
 .agg(count(lit(1)).as("Count"))

groupByGenderStereotype.show(10)

+-------------------+------+-----+
```

```
|first_booking_period|gender|Count|
+--------------------+------+-----+
|             2010-05|  MALE|   21|
|             2015-05|  MALE|   65|
|             2013-02|  MALE|  521|
|             2011-03|FEMALE|   62|
|             2013-05|FEMALE|  954|
|             2014-11|  MALE|  184|
|             2010-02| OTHER|    1|
|             2011-01|FEMALE|   36|
|             2015-04|FEMALE|  103|
|             2015-03| OTHER|    2|
+--------------------+------+-----+

groupByGenderStereotype
 .groupBy("first_booking_period")
 .pivot("gender", Seq("MALE", "FEMALE", "OTHER"))
 .agg(sum("count")).show(10)

+--------------------+----+------+-----+
|first_booking_period|MALE|FEMALE|OTHER|
+--------------------+----+------+-----+
|             2013-05| 855|   954|    5|
|             2013-09| 831|   971|    4|
|             2010-08|  33|    39| null|
|             2013-12| 833|   742|    2|
|             2010-11|  31|    35| null|
|             2013-06| 838|  1025|    3|
|             2010-02|   3|    15|    1|
|             2010-04|   8|    26| null|
|             2011-05|  56|    73| null|
|             2015-05|  65|    84|    1|
+--------------------+----+------+-----+
```

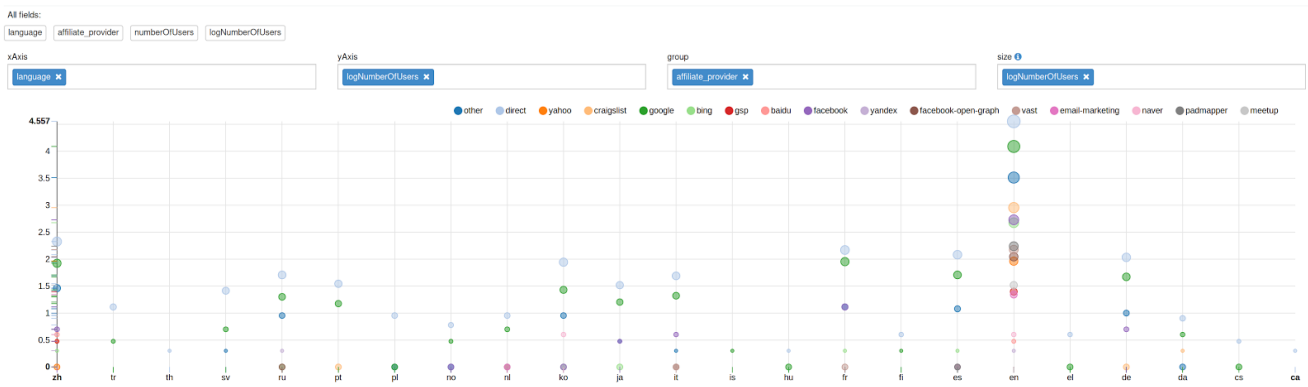
Corrélation langue et provider

Maintenant que nous savons comment faire des répartitions et distributions, nous allons essayer de faire une corrélation entre des *features* pour voir si elles sont liées par un schéma. Nous analysons les données, et essayons de trouver des corrélations entre elles.

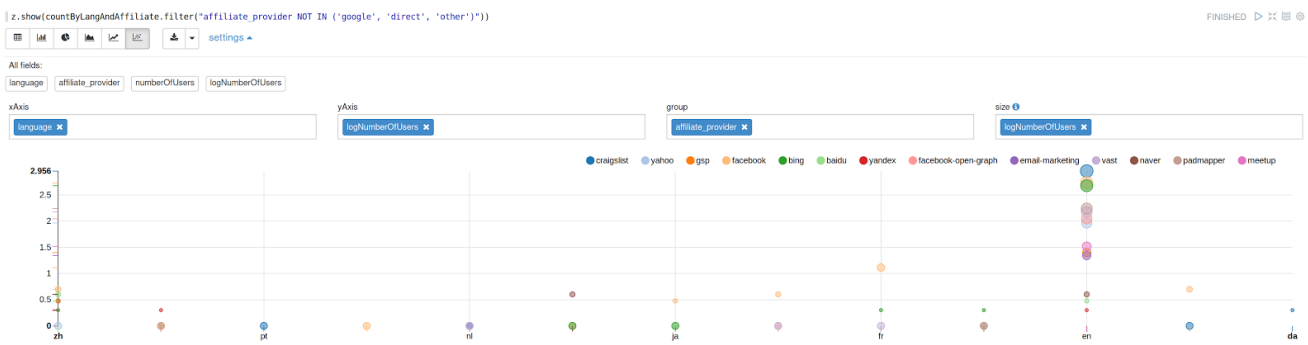
Henri se dit, peut-être à tort, qui sait, qu'il y a une corrélation entre la langue d'un utilisateur et son moyen d'identification. C'est-à-dire le *provider*. Par exemple, il se dit qu'une personne parlant anglais va plus se connecter via Google ou Facebook, et qu'une personne parlant chinois utilisera Baidu. Créons le DataFrame que nous allons ensuite analyser graphiquement.

```
usersFilteredWithRange
 .groupBy("language", "affiliate_provider")
 .agg(count("language").as("numberOfUsers"))
 .orderBy(col("language").desc)
 .withColumn("logNumberOfUsers", log10(col("numberOfUsers")))
```

Et le graphique correspondant :



Bah, Henri s'est trompé... finalement les utilisateurs semblent préférer se connecter directement à Airbnb sans intermédiaire, toutes langues confondues. Nous notons tout de même trois principaux biais de connexion : directement, Google, et d'autres médias non catégorisés. Tentons d'ignorer ceux-ci pour voir quelle tendance se dégage...



Nous pouvons en déduire de nouvelles informations : la plupart semble utiliser Facebook après Google, sauf dans quelques pays comme la Russie (Yandex), la Chine (Baidu/Facebook) ou la Corée du Sud (Naver).

Henri n'a pas pu tirer de corrélations entre le *provider* et la langue de l'utilisateur, mais il a réussi à explorer les données et tenter d'en tirer des caractéristiques sans encombre en Scala !

Maintenant, il s'agit de faire la même chose avec des données discontinues, comme du texte. C'est ce que nous verrons dans un prochain article avec Nina !