Prise en main de Spark

Introduction

Internet, l'apparition de la 3G, 4G et actuellement de la 5G, les grandes installations scientifiques génèrent une immensité de données traitables. En 2020 une analyse a révélé que 40 zéctaoctets sont générés par an, 204 millions d'e-mails sont envoyés chaque minute. L'ensemble de ces données ainsi que les enjeux qui y sont associés représente la notion de Big Data.

Concevoir une infrastructure Big data aujourd'hui consiste à distribuer de manière intelligente et efficace le stockage et les traitements de ces données. Il s'agit principalement de repenser le stockage comme le fait d'utiliser systèmes de fichiers tels que HDFS de Hadoop et les algorithmes de traitement des données comme l'utilisation de MapReduce ou de Spark.

Dans cette partie, nous allons utiliser Spark en Python via l'API PySpark.

PySpark est une interface pour Apache Spark en Python. Elle nous permet non seulement d'écrire des applications Spark à l'aide d'API Python, mais fournit également le shell PySpark pour analyser interactivement les données dans un environnement distribué.

PySpark supporte la plupart des fonctionnalités de Spark telles que Spark SQL, DataFrame, Streaming, MLlib (Machine Learning) et Spark Core.

PySpark SQL

Spark introduit un module de programmation pour le traitement des données structurées appelé Spark SQL. Il fournit une abstraction de programmation appelée DataFrame, très similaire au RDD à la différence qu'il permet de stocker de manière distribuée les données structurées.

Les caractéristiques de Spark SQL

Les caractéristiques de Spark SQL sont les suivantes :

- **Intégré :** Mélanger de manière transparente des requêtes SQL avec des programmes Spark. Spark SQL nous permet d'interroger des données structurées sous forme d'ensemble de données distribuées (RDD) dans Spark, avec des API intégrées en Python, Scala et Java. Cette intégration étroite permet d'exécuter facilement des requêtes SQL aux côtés d'algorithmes analytiques complexes.

- **Accès unifié aux données :** Charger et interroger des données provenant de diverses sources. Les Schema RDD fournissent une interface unique pour travailler efficacement avec des données structurées, notamment des tableaux Apache Hive, des fichiers parquet et des fichiers JSON.

- **Compatibilité Hive :** Exécuter des requêtes Hive non modifiées sur des entrepôts existants. Spark SQL réutilise le front-end et le MetaStore de Hive, offrant une compatibilité totale avec les données, les requêtes et les UDF Hive existants. Il suffit de l'installer à côté de Hive.

- **Connectivité standard :** Connecter via JDBC ou ODBC. Spark SQL comprend un mode serveur avec une connectivité JDBC et ODBC standard.

- **Évolutivité :** Utiliser le même moteur pour les requêtes interactives et longues. Spark SQL tire parti du modèle RDD pour prendre en charge la tolérance aux pannes en milieu de requête, ce qui lui permet d'évoluer vers des travaux de grande envergure.

Les classes du module PySpark SQL

Voici les classes que nous pouvons voir dans le module PySpark SQL :

- **pyspark.sql.SparkSession :** Elle représente le point d'entrée principal pour les fonctionnalités DataFrame et SQL.

- **pyspark.sql.DataFrame :** Elle représente une collection distribuée de données regroupées dans des colonnes nommées.

- **pyspark.sql.Column :** Elle représente une expression de colonne dans un DataFrame.

- **pyspark.sql.Row :** Elle représente une ligne de données dans un DataFrame.
- **pyspark.sql.GroupedData :** Elle représente des méthodes d'agrégation, retournées par DataFrame.groupBy().
- **pyspark.sql.DataFrameNaFunctions :** Il s'agit de méthodes de traitement des données manquantes (valeurs nulles).
- **pyspark.sql.DataFrameStatFunctions :** Elle représente les méthodes pour la fonctionnalité de statistiques.
- **pyspark.sql.functions :** Elle représente une liste de fonctions intégrées disponibles pour DataFrame.
- **pyspark.sql.types :** Elle représente une liste de types de données disponibles.

Les opérations possibles sur un Dataframe PySpark

Les opérations de sélections :

Ce sont les différentes méthodes qui nous permettent d'accéder aux données enregistrées dans un DataFrame PySpark.

- **show() :**

.show() est utilisée pour afficher le contenu du DataFrame dans un format de tableau de lignes et de colonnes. Par défaut, il n'affiche que 20 lignes, et les valeurs des colonnes sont tronquées à 20 caractères.

- **select()**

Nous pouvons sélectionner une ou plusieurs colonnes du DataFrame en passant les noms des colonnes que nous souhaitons sélectionner à la fonction **select().** Comme le DataFrame est immuable, cela crée un nouveau DataFrame avec les colonnes sélectionnées.

- **collect()**

.collect() est une opération d'action qui est utilisée pour récupérer tous les éléments de l'ensemble de données (de tous les nœuds) vers le nœud pilote. Nous devrions utiliser **collect()** sur les petits ensembles de données, généralement après **filter(), group(),** etc. La récupération de plus grands ensembles de données entraîne une erreur **OutOfMemory.**

- **filter() ou .where()**

La méthode PySpark **filter()** est utilisée pour filtrer les lignes d'un RDD/DataFrame sur la base d'une condition donnée ou d'une expression SQL, nous pouvons également utiliser la clause **where()** au lieu de la fonction **filter()** si nous sommes un habitué du SQL, ces deux fonctions fonctionnent exactement de la même manière.

- **.orderBy() ou .sort()**

La méthode **.sort()** ou **orderBy()** de PySpark DataFrame trie un DataFrame par ordre croissant ou décroissant sur la base d'une ou plusieurs colonnes, nous pouvons également utiliser les fonctions de tri SQL de PySpark.

- **.groupBy()**

Semblable à SQL "GROUP BY", la fonction Spark groupBy() est utilisée pour collecter les données identiques en groupes sur DataFrame/Dataset et exécuter des fonctions d'agrégation sur les données regroupées.

Les opérations d'aggrégations

PySpark fournit des fonctions d'agrégation standard intégrées, définies dans l'API de DataFrame, qui s'avèrent pratiques lorsque nous devons effectuer des opérations d'agrégation sur les colonnes de DataFrame. Les fonctions d'agrégation opèrent sur un groupe de lignes et calculent une seule valeur de retour pour chaque groupe.

- **.avg()**

La fonction avg() renvoie la moyenne des valeurs de la colonne d'entrée.

- **.count()**

La fonction count() renvoie le nombre d'éléments dans une colonne.

- **.max()**

La fonction max() renvoie la valeur maximale d'une colonne.

PySpark vs Pandas

Pandas et Spark DataFrame sont conçus pour le traitement des données structurelles et semi-structurelles. Les deux partagent certaines propriétés similaires (structure, fonctions). Les quelques différences entre Pandas et un PySpark DataFrame sont les suivantes :

- Les opérations sur un DataFrame Pyspark s'exécutent en parallèle sur différents nœuds du cluster mais, dans le cas de Pandas, ce n'est pas possible.

- Les opérations dans un DataFrame PySpark sont paresseuses par nature (lazy evaluation) mais, dans le cas de pandas, nous obtenons le résultat dès que nous appliquons une opération.

- Dans un DataFrame PySpark, nous ne pouvons pas modifier le DataFrame en raison de sa propriété immuable, nous devons le transformer. Mais dans Pandas, ce n'est pas le cas.

- Les opérations complexes sont plus faciles à réaliser dans Pandas que dans un DataFrame PySpark.

Requetes SparkSQL

Importer les packages

Utiliser package findSpark pour rendre un contexte Spark disponible dans le code.

Pour installer findspark :

```
pip install findspark
```

```
Requirement already satisfied: findspark in c:\users\hp\anaconda3\lib\site-packages (1.4.2)
Note: you may need to restart the kernel to use updated packages.
```

```
import findspark
findspark.init()
findspark.find()
import pyspark
findspark.find()
```

```
'C:\\spark'
```

```
from pyspark import SparkContext
```

```
from pyspark.sql import SQLContext
```

SQLContext

Spark SQL fournit SQLContext afin d'encapsuler les fonctions du monde relationnel dans Spark.

In [3]:
```
from pyspark import SparkContext
```

Spark Session

A été créée afin d'unifier l'accès aux RDD, Dataframes et Datasets.
Le point d'entrée dans toutes les fonctionnalités de Spark est la SparkSession classe. Pour créer un basique SparkSession, il suffit d'utiliser SparkSession.builder :

```
from pyspark.sql import SparkSession

sc = SparkSession \
    .builder \
    .appName("Python Spark SQL ") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

In [6]:
```
sc
```

Out[6]: **SparkSession - in-memory**

SparkContext

Spark UI

Version	v3.1.2
Master	local[*]
AppName	Python Spark SQL

On voit bien la création de la SparkSession.

Importer les données

```
from pyspark.sql import SparkSession
scSpark = SparkSession \
    .builder \
    .appName("Python Spark SQL: Reading CSV file ") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

- **.show()**

Est utilisée pour afficher le contenu du DataFrame dans un format de tableau de lignes et de colonnes. Par défaut, il n'affiche que 20 lignes, et les valeurs des colonnes sont tronquées à 20 caractères.

```
sdfData = scSpark.read.csv('bank.csv', header=True, sep=",")
# Affiche le contenu du dataframe
sdfData.show()
```

```
+---+-----------+--------+---------+-------+-------+-------+----+-------+---+-----+--------+--------+-----+--------+--------+-------+
|age|        job| marital|education|default|balance|housing|loan|contact|day|month|duration|campaign|pdays|previous|poutcome|deposit|
+---+-----------+--------+---------+-------+-------+-------+----+-------+---+-----+--------+--------+-----+--------+--------+-------+
| 59|     admin.| married|secondary|     no|   2343|    yes|  no|unknown|  5|  may|    1042|       1|   -1|       0| unknown|    yes|
| 56|     admin.| married|secondary|     no|     45|     no|  no|unknown|  5|  may|    1467|       1|   -1|       0| unknown|    yes|
| 41| technician| married|secondary|     no|   1270|    yes|  no|unknown|  5|  may|    1389|       1|   -1|       0| unknown|    yes|
| 55|   services| married|secondary|     no|   2476|    yes|  no|unknown|  5|  may|     579|       1|   -1|       0| unknown|    yes|
| 54|     admin.| married| tertiary|     no|    184|     no|  no|unknown|  5|  may|     673|       2|   -1|       0| unknown|    yes|
| 42| management|  single| tertiary|     no|      0|    yes| yes|unknown|  5|  may|     562|       2|   -1|       0| unknown|    yes|
| 56| management| married| tertiary|     no|    830|    yes| yes|unknown|  6|  may|    1201|       1|   -1|       0| unknown|    yes|
| 60|    retired|divorced|secondary|     no|    545|    yes|  no|unknown|  6|  may|    1030|       1|   -1|       0| unknown|    yes|
| 37| technician| married|secondary|     no|      1|    yes|  no|unknown|  6|  may|     608|       1|   -1|       0| unknown|    yes|
| 28|   services|  single|secondary|     no|   5090|    yes|  no|unknown|  6|  may|    1297|       3|   -1|       0| unknown|    yes|
| 38|     admin.|  single|secondary|     no|    100|    yes|  no|unknown|  7|  may|     786|       1|   -1|       0| unknown|    yes|
| 30|blue-collar| married|secondary|     no|    309|    yes|  no|unknown|  7|  may|    1574|       2|   -1|       0| unknown|    yes|
| 29| management| married| tertiary|     no|    199|    yes| yes|unknown|  7|  may|    1689|       4|   -1|       0| unknown|    yes|
| 46|blue-collar|  single| tertiary|     no|    460|    yes|  no|unknown|  7|  may|    1102|       2|   -1|       0| unknown|    yes|
| 31| technician|  single| tertiary|     no|    703|    yes|  no|unknown|  8|  may|     943|       2|   -1|       0| unknown|    yes|
| 35| management|divorced| tertiary|     no|   3837|    yes|  no|unknown|  8|  may|    1084|       1|   -1|       0| unknown|    yes|
| 32|blue-collar|  single|  primary|     no|    611|    yes|  no|unknown|  8|  may|     541|       3|   -1|       0| unknown|    yes|
| 49|   services| married|secondary|     no|     -8|    yes|  no|unknown|  8|  may|    1119|       1|   -1|       0| unknown|    yes|
| 41|     admin.| married|secondary|     no|     55|    yes|  no|unknown|  8|  may|    1120|       2|   -1|       0| unknown|    yes|
| 49|     admin.|divorced|secondary|     no|    168|    yes| yes|unknown|  8|  may|     513|       1|   -1|       0| unknown|    yes|
+---+-----------+--------+---------+-------+-------+-------+----+-------+---+-----+--------+--------+-----+--------+--------+-------+
only showing top 20 rows
```

Afficher les colonnes de notre data

In [10]:
```
# show the dataframe columns
sdfData.columns
```

Out[10]:
```
['age',
 'job',
 'marital',
 'education',
 'default',
 'balance',
 'housing',
 'loan',
 'contact',
 'day',
 'month',
 'duration',
 'campaign',
 'pdays',
 'previous',
 'poutcome',
 'deposit']
```

printSchema()

Spark Schéma définit la structure du DataFrame que vous pouvez obtenir en appelant la méthode printSchema() sur l'objet DataFrame.

In [11]:
```
# show the schema of the dataframe
sdfData.printSchema()
```

```
root
 |-- age: string (nullable = true)
 |-- job: string (nullable = true)
 |-- marital: string (nullable = true)
 |-- education: string (nullable = true)
 |-- default: string (nullable = true)
 |-- balance: string (nullable = true)
 |-- housing: string (nullable = true)
 |-- loan: string (nullable = true)
 |-- contact: string (nullable = true)
 |-- day: string (nullable = true)
 |-- month: string (nullable = true)
 |-- duration: string (nullable = true)
 |-- campaign: string (nullable = true)
 |-- pdays: string (nullable = true)
 |-- previous: string (nullable = true)
 |-- poutcome: string (nullable = true)
 |-- deposit: string (nullable = true)
```

Exploration des données

- **select()**

Pour sélectionner une ou plusieurs colonnes du DataFrame en passant les noms des colonnes que vous souhaitez sélectionner à la fonction select().

In [12]:

```
#By using col() function
from pyspark.sql.functions import col
sdfData.select(col("age"),col("job")).show()
```

```
+---+-----------+
|age|        job|
+---+-----------+
| 59|     admin.|
| 56|     admin.|
| 41| technician|
| 55|   services|
| 54|     admin.|
| 42| management|
| 56| management|
| 60|    retired|
| 37| technician|
| 28|   services|
| 38|     admin.|
| 30|blue-collar|
| 29| management|
| 46|blue-collar|
| 31| technician|
| 35| management|
| 32|blue-collar|
| 49|   services|
| 41|     admin.|
| 49|     admin.|
+---+-----------+
only showing top 20 rows
```

Donc on a sélectionné et afficher les deux colonnes âge et job.

- **.max()**

La fonction max() renvoie la valeur maximale d'une colonne.

In [13]:

```
from pyspark.sql.functions import  max
sdfData.select(max("balance")).show(truncate=False)
```

```
+------------+
|max(balance)|
+------------+
|9994        |
+------------+
```

On a affiché le max de la variable 'balance' qui est égale à : 9994.

- **groupBy**

Est utilisée en SQL pour grouper plusieurs résultats et utiliser une fonction de totaux sur un groupe de résultat.

In [14]:
```
# Count by age
sdfData.groupBy("age").count().show()
```

```
+---+-----+
|age|count|
+---+-----+
| 51|  194|
| 54|  187|
| 29|  328|
| 69|   20|
| 42|  295|
| 73|   29|
| 87|    4|
| 64|   37|
| 30|  456|
| 34|  466|
| 59|  184|
| 28|  292|
| 22|   48|
| 85|    5|
| 35|  461|
| 52|  205|
| 71|   27|
| 47|  255|
| 43|  249|
| 31|  496|
+---+-----+
only showing top 20 rows
```

In [15]:
```
# Regrouper par deposit
sdfData.groupBy("deposit").count().show(10)
```

```
+-------+-----+
|deposit|count|
+-------+-----+
|     no| 5873|
|    yes| 5289|
+-------+-----+
```

Statistiques récapitulatives

In [16]:
```
# Calcule les statistiques récapitulatives
sdfData.describe().show()
```

```
+-------+-----------------+-------+--------+---------+-------+-----------------+-------+-----+--------+-----------------+-----+-----------------+
|summary|              age|    job| marital|education|default|          balance|housing| loan| contact|              day|month|         duration|
campaign|            pdays|         previous|poutcome|deposit|
+-------+-----------------+-------+--------+---------+-------+-----------------+-------+-----+--------+-----------------+-----+-----------------+
| count|            11162|  11162|   11162|    11162|  11162|            11162|  11162|11162|   11162|            11162|11162|            11162|
11162|           11162|            11162|   11162|  11162|
|   mean|41.231947679627304|   null|    null|     null|   null|1528.5385235620856|   null| null|    null|15.658036194230425| null|371.99381831213043|
2.508421429851281| 51.33040673714388|0.8325568894463358|    null|   null|
| stddev|11.913369192215518|   null|    null|     null|   null| 3225.413325946149|   null| null|    null| 8.420739541006462| null|347.12838571630687|
2.7220771816614824|108.75828197197717| 2.292007218670508|    null|   null|
|    min|               18| admin.|divorced|  primary|     no|               -1|     no|   no|cellular|                1|  apr|               10|
1|               -1|                0| failure|     no|
|    max|               95|unknown|  single|  unknown|    yes|             9994|    yes|  yes| unknown|                9|  sep|              999|
9|               99|                9| unknown|    yes|
+-------+-----------------+-------+--------+---------+-------+-----------------+-------+-----+--------+-----------------+-----+-----------------+
-----------------+-----------------+-----------------+--------+-------+
```

In [18]:
```
sdfData.describe().dtypes
```

Out[18]:
```
[('summary', 'string'),
 ('age', 'string'),
 ('job', 'string'),
 ('marital', 'string'),
 ('education', 'string'),
 ('default', 'string'),
 ('balance', 'string'),
 ('housing', 'string'),
 ('loan', 'string'),
 ('contact', 'string'),
 ('day', 'string'),
 ('month', 'string'),
 ('duration', 'string'),
 ('campaign', 'string'),
 ('pdays', 'string'),
 ('previous', 'string'),
 ('poutcome', 'string'),
 ('deposit', 'string')]
```

Supprimer quelques colonnes

In [19]:
```
dataframe_remove = sdfData.drop("day", "month","marital").show(8)
```

```
+---+----------+---------+-------+-------+-------+----+-------+--------+--------+-----+--------+--------+-------+
|age|       job|education|default|balance|housing|loan|contact|duration|campaign|pdays|previous|poutcome|deposit|
+---+----------+---------+-------+-------+-------+----+-------+--------+--------+-----+--------+--------+-------+
| 59|    admin.|secondary|     no|   2343|    yes|  no|unknown|    1042|       1|   -1|       0| unknown|    yes|
| 56|    admin.|secondary|     no|     45|     no|  no|unknown|    1467|       1|   -1|       0| unknown|    yes|
| 41|technician|secondary|     no|   1270|    yes|  no|unknown|    1389|       1|   -1|       0| unknown|    yes|
| 55|  services|secondary|     no|   2476|    yes|  no|unknown|     579|       1|   -1|       0| unknown|    yes|
| 54|    admin.| tertiary|     no|    184|     no|  no|unknown|     673|       2|   -1|       0| unknown|    yes|
| 42|management| tertiary|     no|      0|    yes| yes|unknown|     562|       2|   -1|       0| unknown|    yes|
| 56|management| tertiary|     no|    830|    yes| yes|unknown|    1201|       1|   -1|       0| unknown|    yes|
| 60|   retired|secondary|     no|    545|    yes|  no|unknown|    1030|       1|   -1|       0| unknown|    yes|
+---+----------+---------+-------+-------+-------+----+-------+--------+--------+-----+--------+--------+-------+
only showing top 8 rows
```

On a supprimé les 3 colonnes day, month et marital.

Filtre

In [20]:
```
sdfData.filter(sdfData.duration < 500).count()
```

Out[20]:
```
8389
```

In [21]:
```
sdfData.filter(sdfData.duration > 500).count()
```

Out[21]:
```
2766
```

In [22]:
```
sdfData.filter(sdfData.balance == 0.0).count()
```

Out[22]:
```
774
```

Utilisation d’une requête SQL dans un programme

Le SparkSession fournit une méthode appelée sql qui peut être utilisée pour exécuter une requête SQL. La méthode sql prend une instruction SQL comme argument et renvoie un DataFrame représentant le résultat de la requête donnée.

Malheureusement, les requêtes SQL ne peuvent pas être exécutées directement sur un DataFrame.

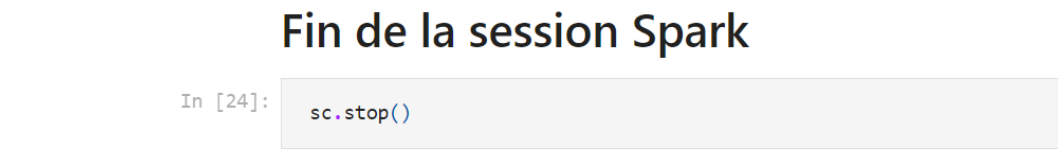
Pour émettre des requêtes SQL sur un DataFrame existant, nous pouvons tirer parti de la fonction createOrReplaceTempView pour créer une table temporaire.

In [23]:

```
# Enregistrement du DataFrame comme une vue temporaire
sdfData.createOrReplaceTempView("bank")

# Utilisez la commande suivante pour sélectionner tous les enregistrements de la table bank .
# Ici, nous utilisons la variable job pour capturer toutes les données des enregistrements.

sqlDF = sc.sql("SELECT * FROM bank WHERE job = 'admin.'")

# Pour afficher ces enregistrements, appelez la méthode show () .
sqlDF.show()
```

```
+---+------+--------+---------+-------+-------+-------+----+-------+---+-----+--------+--------+-----+--------+--------+-------+
|age|   job| marital|education|default|balance|housing|loan|contact|day|month|duration|campaign|pdays|previous|poutcome|deposit|
+---+------+--------+---------+-------+-------+-------+----+-------+---+-----+--------+--------+-----+--------+--------+-------+
| 59|admin.| married|secondary|     no|   2343|    yes|  no|unknown|  5|  may|    1042|       1|   -1|       0| unknown|    yes|
| 56|admin.| married|secondary|     no|     45|     no|  no|unknown|  5|  may|    1467|       1|   -1|       0| unknown|    yes|
| 54|admin.| married| tertiary|     no|    184|     no|  no|unknown|  5|  may|     673|       2|   -1|       0| unknown|    yes|
| 38|admin.|  single|secondary|     no|    100|    yes|  no|unknown|  7|  may|     786|       1|   -1|       0| unknown|    yes|
| 41|admin.| married|secondary|     no|     55|    yes|  no|unknown|  8|  may|    1120|       2|   -1|       0| unknown|    yes|
| 49|admin.|divorced|secondary|     no|    168|    yes| yes|unknown|  8|  may|     513|       1|   -1|       0| unknown|    yes|
| 28|admin.|divorced|secondary|     no|    785|    yes|  no|unknown|  8|  may|     442|       2|   -1|       0| unknown|    yes|
| 31|admin.| married|secondary|     no|    307|    yes|  no|unknown|  9|  may|     538|       1|   -1|       0| unknown|    yes|
| 40|admin.| married|secondary|     no|   1535|    yes|  no|unknown| 14|  may|     704|       1|   -1|       0| unknown|    yes|
| 41|admin.| married|  primary|     no|   -306|    yes|  no|unknown| 15|  may|     500|       1|   -1|       0| unknown|    yes|
| 29|admin.|  single|secondary|     no|    163|    yes| yes|unknown| 16|  may|     803|       1|   -1|       0| unknown|    yes|
| 50|admin.|divorced|secondary|     no|    363|    yes|  no|unknown| 16|  may|    1340|       1|   -1|       0| unknown|    yes|
| 41|admin.|  single|secondary|     no|   1020|    yes|  no|unknown| 19|  may|     882|       1|   -1|       0| unknown|    yes|
| 31|admin.| married|secondary|     no|     97|    yes|  no|unknown| 19|  may|     607|       1|   -1|       0| unknown|    yes|
| 31|admin.|  single|secondary|     no|    431|    yes| yes|unknown| 20|  may|    1045|       1|   -1|       0| unknown|    yes|
| 45|admin.| married|secondary|     no|    149|    yes|  no|unknown| 23|  may|     893|       3|   -1|       0| unknown|    yes|
| 32|admin.|  single|secondary|     no|    620|    yes|  no|unknown| 26|  may|    1234|       3|   -1|       0| unknown|    yes|
| 38|admin.|divorced|secondary|     no|    198|    yes|  no|unknown| 26|  may|     431|       2|   -1|       0| unknown|    yes|
| 28|admin.|  single|secondary|     no|    460|    yes|  no|unknown| 26|  may|     741|       6|   -1|       0| unknown|    yes|
| 37|admin.| married|secondary|     no|    274|    yes|  no|unknown| 27|  may|     731|       3|   -1|       0| unknown|    yes|
```

Fin de la session Spark



Spark MLLib

Spark MLLib est la librairie d'apprentissage automatique (Machine Learning) de Spark. Son but est de rendre son utilisation facile et scalable. À un haut niveau d'abstraction, elle fournit des outils tel que:

- Des algorithmes de machine learning classiques tel que la classification, régression, clustering et filtrage collaboratif.
- L'extraction de caractéristiques (Features), transformation, réduction de dimensions et sélection
- Les pipelines pour construire, évaluer et régler les pipelines ML.
- La persistance, pour sauvegarder et charger des algorithmes, modèles et pipelines.
- Des utilitaires tel que l'algèbre linéaire, statistiques, manipulation des données, etc.

A partir de la version 2.0 de Spark, la structure principale utilisée pour l'API MLlib est DataFrame, en opposition aux RDD, car son utilisation est plus intuitive et uniforme, et qu'elle facilite les transformations.

Chargement des données

L'ensemble de données provient de l'**UCI Machine Learning repository**, il est lié aux campagnes de marketing direct (appels téléphoniques) d'une institution bancaire portugaise. L'objectif de la classification est de prédire si le client souscrira (Oui/Non) à un dépôt à terme. L'ensemble de données peut être téléchargé à partir de Kaggle: https://www.kaggle.com/rouseguy/bankbalanced/data

Input variables:

age, job, marital, education, default, balance, housing, loan, contact, day, month, duration, campaign, pdays, previous, poutcome.

Output variable:

deposit

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('ml-bank').getOrCreate()
df = spark.read.csv('bank.csv', header = True, inferSchema = True)
df.printSchema()
```

```
root
 |-- age: integer (nullable = true)
 |-- job: string (nullable = true)
 |-- marital: string (nullable = true)
 |-- education: string (nullable = true)
 |-- default: string (nullable = true)
 |-- balance: integer (nullable = true)
 |-- housing: string (nullable = true)
 |-- loan: string (nullable = true)
 |-- contact: string (nullable = true)
 |-- day: integer (nullable = true)
 |-- month: string (nullable = true)
 |-- duration: integer (nullable = true)
 |-- campaign: integer (nullable = true)
 |-- pdays: integer (nullable = true)
 |-- previous: integer (nullable = true)
 |-- poutcome: string (nullable = true)
 |-- deposit: string (nullable = true)
```

L'ensemble de données fournit les informations des clients de la banque. Il comprend **11 162** enregistrements et **17** champs.

- Pour afficher la 1ere ligne du DataFrame:

```
df.head()
```

```
Row(age=59, job='admin.', marital='married', education='secondary', default='no', balance=2343, housing='yes', loan='no', contact='unknown', day=5, mont
h='may', duration=1042, campaign=1, pdays=-1, previous=0, poutcome='unknown', deposit='yes')
```

- Pour afficher la liste des différentes colonnes du DataFrame:

```
print(list(df.columns))
```

```
['age', 'job', 'marital', 'education', 'default', 'balance', 'housing', 'loan', 'contact', 'day', 'month', 'duration', 'campaign', 'pdays', 'previous',
'poutcome', 'deposit']
```

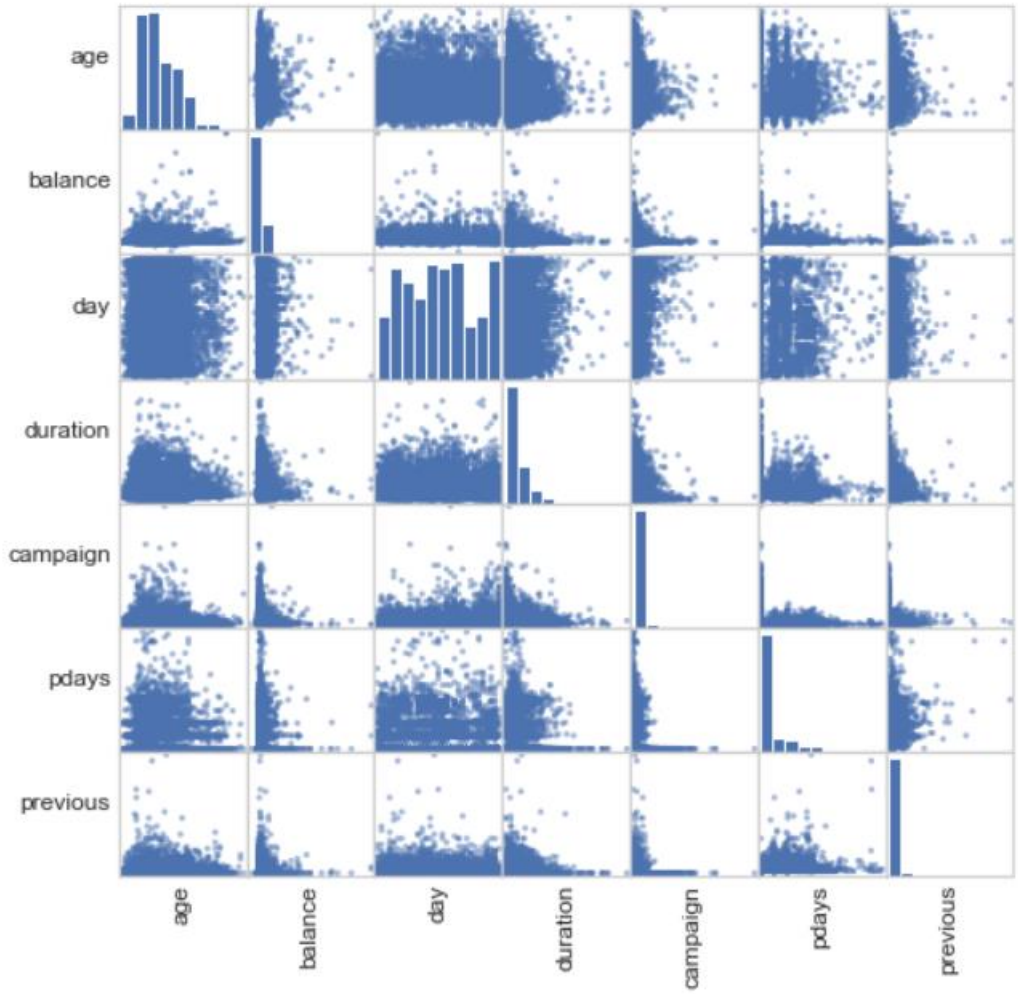
".show()" est utilisée pour afficher le contenu du DataFrame dans un format de tableau de lignes et de colonnes. Par défaut, il n'affiche que 20 lignes.

```
df.show()
```

```
+---+-----------+--------+---------+-------+-------+-------+----+-------+---+-----+--------+--------+-----+--------+--------+-------+
|age|        job| marital|education|default|balance|housing|loan|contact|day|month|duration|campaign|pdays|previous|poutcome|deposit|
+---+-----------+--------+---------+-------+-------+-------+----+-------+---+-----+--------+--------+-----+--------+--------+-------+
| 59|     admin.| married|secondary|     no|   2343|    yes|  no|unknown|  5|  may|    1042|       1|   -1|       0| unknown|    yes|
| 56|     admin.| married|secondary|     no|     45|     no|  no|unknown|  5|  may|    1467|       1|   -1|       0| unknown|    yes|
| 41| technician| married|secondary|     no|   1270|    yes|  no|unknown|  5|  may|    1389|       1|   -1|       0| unknown|    yes|
| 55|   services| married|secondary|     no|   2476|    yes|  no|unknown|  5|  may|     579|       1|   -1|       0| unknown|    yes|
| 54|     admin.| married| tertiary|     no|    184|     no|  no|unknown|  5|  may|     673|       2|   -1|       0| unknown|    yes|
| 42| management|  single| tertiary|     no|      0|    yes| yes|unknown|  5|  may|     562|       2|   -1|       0| unknown|    yes|
| 56| management| married| tertiary|     no|    830|    yes| yes|unknown|  6|  may|    1201|       1|   -1|       0| unknown|    yes|
| 60|    retired|divorced|secondary|     no|    545|    yes|  no|unknown|  6|  may|    1030|       1|   -1|       0| unknown|    yes|
| 37| technician| married|secondary|     no|      1|    yes|  no|unknown|  6|  may|     608|       1|   -1|       0| unknown|    yes|
| 28|   services|  single|secondary|     no|   5090|    yes|  no|unknown|  6|  may|    1297|       3|   -1|       0| unknown|    yes|
| 38|     admin.|  single|secondary|     no|    100|    yes|  no|unknown|  7|  may|     786|       1|   -1|       0| unknown|    yes|
| 30|blue-collar| married|secondary|     no|    309|    yes|  no|unknown|  7|  may|    1574|       2|   -1|       0| unknown|    yes|
| 29| management| married| tertiary|     no|    199|    yes| yes|unknown|  7|  may|    1689|       4|   -1|       0| unknown|    yes|
| 46|blue-collar|  single| tertiary|     no|    460|    yes|  no|unknown|  7|  may|    1102|       2|   -1|       0| unknown|    yes|
| 31| technician|  single| tertiary|     no|    703|    yes|  no|unknown|  8|  may|     943|       2|   -1|       0| unknown|    yes|
| 35| management|divorced| tertiary|     no|   3837|    yes|  no|unknown|  8|  may|    1084|       1|   -1|       0| unknown|    yes|
| 32|blue-collar|  single|  primary|     no|    611|    yes|  no|unknown|  8|  may|     541|       3|   -1|       0| unknown|    yes|
| 49|   services| married|secondary|     no|     -8|    yes|  no|unknown|  8|  may|    1119|       1|   -1|       0| unknown|    yes|
| 41|     admin.| married|secondary|     no|     55|    yes|  no|unknown|  8|  may|    1120|       2|   -1|       0| unknown|    yes|
| 49|     admin.|divorced|secondary|     no|    168|    yes| yes|unknown|  8|  may|     513|       1|   -1|       0| unknown|    yes|
+---+-----------+--------+---------+-------+-------+-------+----+-------+---+-----+--------+--------+-----+--------+--------+-------+
only showing top 20 rows
```

Corrélations entre les variables indépendantes

```
numeric_data = df.select(numeric_features).toPandas()

axs = pd.plotting.scatter_matrix(numeric_data, figsize=(8, 8));

# Rotate axis labels and remove axis ticks
n = len(numeric_data.columns)
for i in range(n):
    v = axs[i, 0]
    v.yaxis.label.set_rotation(0)
    v.yaxis.label.set_ha('right')
    v.set_yticks(())
    h = axs[n-1, i]
    h.xaxis.label.set_rotation(90)
    h.set_xticks(())
```



Il est évident qu'il n'y a pas de variables numériques fortement corrélées. Par conséquent, nous les conserverons tous pour le modèle. Cependant, les colonnes day et month ne sont pas vraiment utiles, nous allons supprimer ces deux colonnes.

```
df = df.select('age', 'job', 'marital', 'education', 'default', 'balance', 'housing', 'loan', 'contact', 'duration', 'campaign', 'pdays', 'previous', '
cols = df.columns
df.printSchema()
```

```
root
 |-- age: integer (nullable = true)
 |-- job: string (nullable = true)
 |-- marital: string (nullable = true)
 |-- education: string (nullable = true)
 |-- default: string (nullable = true)
 |-- balance: integer (nullable = true)
 |-- housing: string (nullable = true)
 |-- loan: string (nullable = true)
 |-- contact: string (nullable = true)
 |-- duration: integer (nullable = true)
 |-- campaign: integer (nullable = true)
 |-- pdays: integer (nullable = true)
 |-- previous: integer (nullable = true)
 |-- poutcome: string (nullable = true)
 |-- deposit: string (nullable = true)
```

Préparation du jeu de données pour le Machine Learning

Le processus comprend le **Category Indexing**, **OneHot Encoding** et **VectorAssembler**, un transformateur de fonctionnalités qui fusionne plusieurs colonnes en une colonne vectorielle.

```
from pyspark.ml.feature import OneHotEncoder, StringIndexer, VectorAssembler

categoricalColumns = ['job', 'marital', 'education', 'default', 'housing', 'loan', 'contact', 'poutcome']
stages = []

for categoricalCol in categoricalColumns:
    stringIndexer = StringIndexer(inputCol = categoricalCol, outputCol = categoricalCol + 'Index')
    encoder = OneHotEncoder(inputCols=[stringIndexer.getOutputCol()], outputCols=[categoricalCol + "classVec"])
    stages += [stringIndexer, encoder]

label_stringIdx = StringIndexer(inputCol = 'deposit', outputCol = 'label')
stages += [label_stringIdx]

numericCols = ['age', 'balance', 'duration', 'campaign', 'pdays', 'previous']
assemblerInputs = [c + "classVec" for c in categoricalColumns] + numericCols
assembler = VectorAssembler(inputCols=assemblerInputs, outputCol="features")
stages += [assembler]
```

Le code ci-dessus est extrait du site officiel de **Databricks :** [2]

Pipeline

Nous utilisons Pipeline pour chaîner plusieurs transformateurs et estimateurs afin de spécifier notre flux de travail d'apprentissage automatique. Les étapes d'un Pipeline sont spécifiées sous la forme d'un tableau ordonné.

```
from pyspark.ml import Pipeline
pipeline = Pipeline(stages = stages)
pipelineModel = pipeline.fit(df)
df = pipelineModel.transform(df)
selectedCols = ['label', 'features'] + cols
df = df.select(selectedCols)
df.printSchema()
```

```
root
 |-- label: double (nullable = false)
 |-- features: vector (nullable = true)
 |-- age: integer (nullable = true)
 |-- job: string (nullable = true)
 |-- marital: string (nullable = true)
 |-- education: string (nullable = true)
 |-- default: string (nullable = true)
 |-- balance: integer (nullable = true)
 |-- housing: string (nullable = true)
 |-- loan: string (nullable = true)
 |-- contact: string (nullable = true)
 |-- duration: integer (nullable = true)
 |-- campaign: integer (nullable = true)
 |-- pdays: integer (nullable = true)
 |-- previous: integer (nullable = true)
 |-- poutcome: string (nullable = true)
 |-- deposit: string (nullable = true)
```

```
pd.DataFrame(df.take(5), columns=df.columns).transpose()
```

	0	1	2	3	4
label	1.0	1.0	1.0	1.0	1.0
features	(0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, ...	(0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, ...	(0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...	(0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, ...	(0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, ...
age	59	56	41	55	54
job	admin.	admin.	technician	services	admin.
marital	married	married	married	married	married
education	secondary	secondary	secondary	secondary	tertiary
default	no	no	no	no	no
balance	2343	45	1270	2476	184
housing	yes	no	yes	yes	no
loan	no	no	no	no	no
contact	unknown	unknown	unknown	unknown	unknown
duration	1042	1467	1389	579	673
campaign	1	1	1	1	2
pdays	-1	-1	-1	-1	-1
previous	0	0	0	0	0
poutcome	unknown	unknown	unknown	unknown	unknown
deposit	yes	yes	yes	yes	yes

Nous avons maintenant une colonne de caractéristiques (**features**) et une colonne d'étiquette (**Label**).

Split – Training - Test

Nous avons divisé le jeu de données en 2 parties: Une partie pour **l'entrainement** (70%) et une partie pour **le test** (30%)

```
train, test = df.randomSplit([0.7, 0.3], seed = 2018)
print("Training Dataset Count: " + str(train.count()))
print("Test Dataset Count: " + str(test.count()))
```

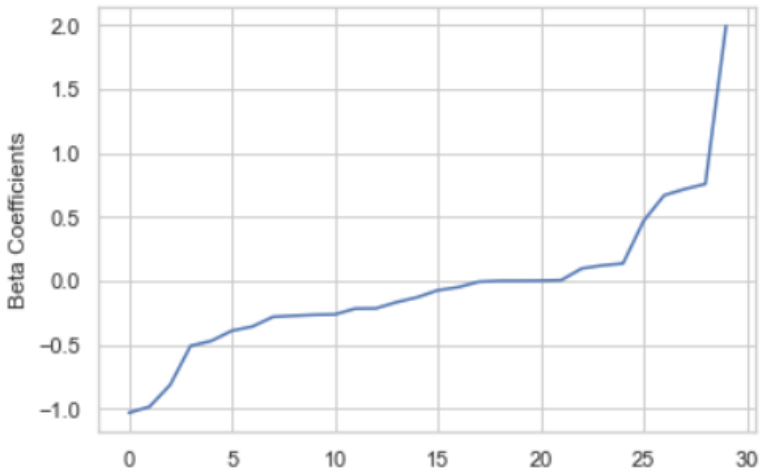
```
Training Dataset Count: 7855
Test Dataset Count: 3307
```

Modèle de régression logistique

```
from pyspark.ml.classification import LogisticRegression

lr = LogisticRegression(featuresCol = 'features', labelCol = 'label', maxIter=10)
lrModel = lr.fit(train)
```

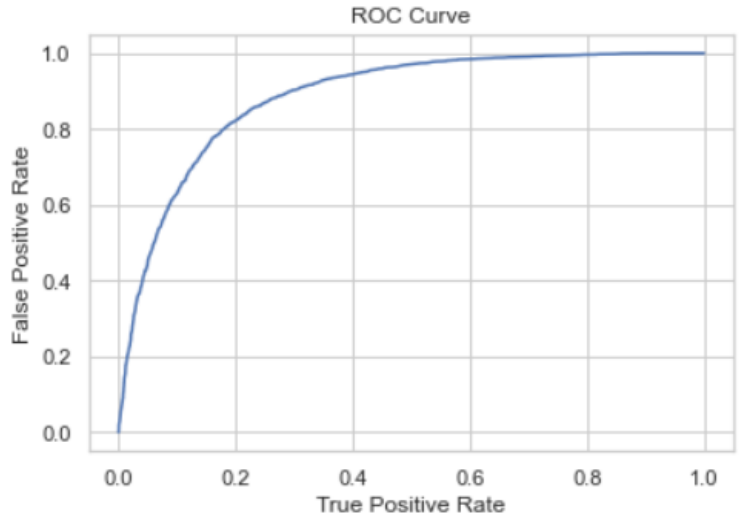
Nous pouvons obtenir les coefficients en utilisant les attributs de LogisticRegressionModel.

```
import matplotlib.pyplot as plt
import numpy as np

beta = np.sort(lrModel.coefficients)

plt.plot(beta)
plt.ylabel('Beta Coefficients')
plt.show()
```



Nous pouvons également obtenir le **receiver-operating characteristic** et **areaUnderROC**.

```
trainingSummary = lrModel.summary

roc = trainingSummary.roc.toPandas()
plt.plot(roc['FPR'],roc['TPR'])
plt.ylabel('False Positive Rate')
plt.xlabel('True Positive Rate')
plt.title('ROC Curve')
plt.show()

print('Training set areaUnderROC: ' + str(trainingSummary.areaUnderROC))
```




```
Training set areaUnderROC: 0.8865551189326405
```

Faire des prédictions sur l'ensemble de test

```
predictions = lrModel.transform(test)
predictions.select('age', 'job', 'label', 'rawPrediction', 'prediction', 'probability').show(10)
```

```
+---+----------+-----+--------------------+----------+--------------------+
|age|       job|label|       rawPrediction|prediction|         probability|
+---+----------+-----+--------------------+----------+--------------------+
| 33|management|  0.0|[1.93500470498079...|       0.0|[0.87380233100701...|
| 49|management|  0.0|[2.04809361205496...|       0.0|[0.88575484704982...|
| 52|management|  0.0|[-0.7002122133357...|       1.0|[0.33176517908283...|
| 53|management|  0.0|[1.03431610227357...|       0.0|[0.73775180622517...|
| 58|management|  0.0|[2.74273671692568...|       0.0|[0.93950183380104...|
| 32|management|  0.0|[1.25033004378244...|       0.0|[0.77735698810451...|
| 57|management|  0.0|[1.19540519867323...|       0.0|[0.76770638557283...|
| 52|management|  0.0|[3.26456982709641...|       0.0|[0.96319314409492...|
| 46|management|  0.0|[2.15036209979036...|       0.0|[0.89570260888977...|
| 31|management|  0.0|[1.47751802200862...|       0.0|[0.81419739986183...|
+---+----------+-----+--------------------+----------+--------------------+
only showing top 10 rows
```

Évaluez notre modèle de régression logistique

```
from pyspark.ml.evaluation import BinaryClassificationEvaluator

evaluator = BinaryClassificationEvaluator()
print('Test Area Under ROC', evaluator.evaluate(predictions))
```

```
Test Area Under ROC 0.8837112925002687
```

Decision Tree Classifier

Les arbres de décision sont largement utilisés car ils sont faciles à interpréter, gèrent les caractéristiques catégorielles, s'étendent à la classification multiclasse, ne nécessitent pas de mise à l'échelle des caractéristiques et sont capables de capturer les non-linéarités et les interactions de caractéristiques.

```
from pyspark.ml.classification import DecisionTreeClassifier
dt = DecisionTreeClassifier(featuresCol = 'features', labelCol = 'label', maxDepth = 3)
dtModel = dt.fit(train)
predictions = dtModel.transform(test)
predictions.select('age', 'job', 'label', 'rawPrediction', 'prediction', 'probability').show(10)
```

```
+---+----------+-----+--------------+----------+--------------------+
|age|       job|label| rawPrediction|prediction|         probability|
+---+----------+-----+--------------+----------+--------------------+
| 33|management|  0.0|[2498.0,481.0]|       0.0|[0.83853642161799...|
| 49|management|  0.0|[2498.0,481.0]|       0.0|[0.83853642161799...|
| 52|management|  0.0|[520.0,1931.0]|       1.0|[0.21215830273357...|
| 53|management|  0.0|[2498.0,481.0]|       0.0|[0.83853642161799...|
| 58|management|  0.0|[2498.0,481.0]|       0.0|[0.83853642161799...|
| 32|management|  0.0|[2498.0,481.0]|       0.0|[0.83853642161799...|
| 57|management|  0.0|[2498.0,481.0]|       0.0|[0.83853642161799...|
| 52|management|  0.0|[2498.0,481.0]|       0.0|[0.83853642161799...|
| 46|management|  0.0|[2498.0,481.0]|       0.0|[0.83853642161799...|
| 31|management|  0.0|[2498.0,481.0]|       0.0|[0.83853642161799...|
+---+----------+-----+--------------+----------+--------------------+
only showing top 10 rows
```

Évaluez notre modèle d'arbre de décision

```
evaluator = BinaryClassificationEvaluator()
print("Test Area Under ROC: " + str(evaluator.evaluate(predictions, {evaluator.metricName: "areaUnderROC"})))
```

```
Test Area Under ROC: 0.7808118726917547
```

Un arbre de décision simple a mal fonctionné car il est trop faible. La précision de prédiction des arbres de décision peut être améliorée par differents méthodes telles que **Random Forest** et **Gradient-Boosted Tree**.

Random Forest Classifier

```
from pyspark.ml.classification import RandomForestClassifier
rf = RandomForestClassifier(featuresCol = 'features', labelCol = 'label')
rfModel = rf.fit(train)
predictions = rfModel.transform(test)
predictions.select('age', 'job', 'label', 'rawPrediction', 'prediction', 'probability').show(10)
```

```
+---+----------+-----+--------------------+----------+--------------------+
|age|       job|label|       rawPrediction|prediction|         probability|
+---+----------+-----+--------------------+----------+--------------------+
| 33|management|  0.0|[13.3319994510978...|       0.0|[0.66659997255489...|
| 49|management|  0.0|[12.8274882120090...|       0.0|[0.64137441060045...|
| 52|management|  0.0|[7.51139490276589...|       1.0|[0.37556974513829...|
| 53|management|  0.0|[11.7913876191042...|       0.0|[0.58956938095521...|
| 58|management|  0.0|[14.2737109939265...|       0.0|[0.71368554969632...|
| 32|management|  0.0|[13.8512305758578...|       0.0|[0.69256152879289...|
| 57|management|  0.0|[13.7311667026663...|       0.0|[0.68655833513331...|
| 52|management|  0.0|[16.5516422079630...|       0.0|[0.82758211039815...|
| 46|management|  0.0|[16.6564667517940...|       0.0|[0.83282333758970...|
| 31|management|  0.0|[13.2196082553671...|       0.0|[0.66098041276835...|
+---+----------+-----+--------------------+----------+--------------------+
only showing top 10 rows
```

Évaluez le classificateur du Random Forest

```
evaluator = BinaryClassificationEvaluator()
print("Test Area Under ROC: " + str(evaluator.evaluate(predictions, {evaluator.metricName: "areaUnderROC"})))
```

```
Test Area Under ROC: 0.8770900512243783
```

Gradient-Boosted Tree Classifier

```
from pyspark.ml.classification import GBTClassifier
gbt = GBTClassifier(maxIter=10)
gbtModel = gbt.fit(train)
predictions = gbtModel.transform(test)
predictions.select('age', 'job', 'label', 'rawPrediction', 'prediction', 'probability').show(10)
```

```
+---+----------+-----+--------------------+----------+--------------------+
|age|       job|label|       rawPrediction|prediction|         probability|
+---+----------+-----+--------------------+----------+--------------------+
| 33|management|  0.0|[0.94992370947246...|       0.0|[0.86987425549896...|
| 49|management|  0.0|[1.28281748604659...|       0.0|[0.92861688965145...|
| 52|management|  0.0|[-0.5021429054421...|       1.0|[0.26809961471379...|
| 53|management|  0.0|[0.09646513340517...|       0.0|[0.54808351175538...|
| 58|management|  0.0|[1.05831377292163...|       0.0|[0.89250881497055...|
| 32|management|  0.0|[0.57268875896254...|       0.0|[0.75866558918304...|
| 57|management|  0.0|[0.34932946844840...|       0.0|[0.66789037418062...|
| 52|management|  0.0|[1.33264456943866...|       0.0|[0.93494709840561...|
| 46|management|  0.0|[1.28262818068783...|       0.0|[0.92859178839697...|
| 31|management|  0.0|[1.16094954405484...|       0.0|[0.91067454521219...|
+---+----------+-----+--------------------+----------+--------------------+
only showing top 10 rows
```

Évaluez le classificateur du Gradient-Boosted Tree

```
evaluator = BinaryClassificationEvaluator()
print("Test Area Under ROC: " + str(evaluator.evaluate(predictions, {evaluator.metricName: "areaUnderROC"})))
```

```
Test Area Under ROC: 0.8935091626908479
```

Gradient-Boosted Tree a obtenu les meilleurs résultats, nous allons essayer de régler ce modèle avec **ParamGridBuilder** et **CrossValidator**.

ParamGridBuilder et CrossValidator

```
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator
paramGrid = (ParamGridBuilder()
             .addGrid(gbt.maxDepth, [2, 4, 6])
             .addGrid(gbt.maxBins, [20, 60])
             .addGrid(gbt.maxIter, [10, 20])
             .build())
cv = CrossValidator(estimator=gbt, estimatorParamMaps=paramGrid, evaluator=evaluator, numFolds=5)
# Run cross validations.
cvModel = cv.fit(train)
predictions = cvModel.transform(test)
evaluator.evaluate(predictions)
```

```
0.8954659543871663
```

Pour résumer, nous avons créer une application de classification binaire à l'aide de **PySpark** et de l'**API Pipelines MLlib**. Nous avons essayé **quatre algorithmes** de **Machine Learning** et l'amplification du gradient a donné les meilleurs résultats sur notre jeu de données.