Module 7 Big data

7.2 Introduction to *Spark* using Python

By Francisco Javier García Castellano

Associate Professor at the Department of Computer Science and Artificial Intelligence (DECSAI), University of Granada.

Reminder: Introduction to this NoteBook.

In this NoteBook, you will be guided step-by-step,through loading a dataset to the descriptive analysis of its contents.

The *Jupyter NoteBook* (Python) is an approach that combines text blocks (like this one) and code blocks or cells. The great advantage of this system is its interactivity because cells can be executed to directly check the results they contain. *Very important*: the order of the instructions is fundamental and so each cell of this NoteBook must be executed sequentially. In any are omitted, the program may throw an error and so if there is any doubt, you will have to start from the beginning again.

First, it is very important to select "*Open in draft mode*" (draft mode) at the top left at the beginning. Otherwise, for security reasons, it will not be allowed to execute any code blocks. When the first of the blocks is executed, the following message will appear: "*Warning: This NoteBook was not created by Google*". Don't worry, you can trust the contents of the *NoteBook* and click on "Run anyway".

Let's start!

Click on the "play" button on the left side of each code cell. Remember that lines beginning with a hashtag (#) are comments and do not affect the execution of the script.

You can also click on each cell and press "*Ctrl+enter*" (*Cmd+enter* on Mac).

Each time you execute a block, you will see the output just below it. The information is usually always the last statement, along with any `print()` commands present in the code.

INDEX

In this *NoteBook*:

1. We learn how to work with Apache *Spark* using Python.
2. We see the options available to us for workingg with data in *Spark*.
3. We learn how to load data from a file and work with big data.

Contents:

1. AN INTRODUCTION TO SPARK USING PYTHON: PYSPARK

Apache *Spark* supports different programming languages, including Python. The standard way to work with Apache *Spark* using Python is through *PySpark*. We can use *PySpark* to run our Python programs using *Spark* or inside a (*NoteBook*) like this one.

There are several ways to run *Spark*:

- Computer cluster: You can run *Spark* on a computer cluster if you have access to one or the economics means to set one up. This option is not recommended for beginners but a lab in this need.
- Cloud servers: Most cloud computing service providers offer the possibility of using Spark. The best-known examples are the cloud services offered iby the Google Cloud Service, Microsoft Azure, Amazon Elastic MapReduce (EMR), or DataBricks (which allows you to set up a very basic free account). This is a good option that does not require a large investment and can be adjusted to changing research needs.

We can also use *Spark* on our personal computer, but this will only be useful for learning or program development and debugging. We will not be able to use our personal computer for real big data problems, and if we can, it means that the problem probably does not really involve big data.

The examples that we will see in this course are run in Google Colaboratory which does not have high computational capabilities but is sufficiente for the examples and exercises that we will see. You can check these capabilities with the following code snippets.

In [1]:
```
!cat /proc/cpuinfo | grep model\ name #This code shows the number of processors.
!cat /proc/meminfo | grep MemTotal    #This code shows the total amount of RAM memory available.
```

```
model name      : Intel(R) Xeon(R) CPU @ 2.20GHz
model name      : Intel(R) Xeon(R) CPU @ 2.20GHz
MemTotal:       13297200 kB
```

1.1. Install Java, Apache *Spark*, *PySpark*, and *Findspark*

The Python programs we are going to see will require you to install Java and Apache *Spark*. Java is required to run *Spark* and if we want to work on our personal computers, we need to install both Java and *Spark*. However, to avoid having to install anything on our computer, we will use Google Colaboratory. Thus, with the following code, we will install Apache *Spark* inside Google Colaboratory (Java is already installed).

In [2]:
```
#Download Spark with Hadoop.
!wget -q --show-progress http://www-eu.apache.org/dist/spark/spark-3.2.3/spark-3.2.3-bin-hadoop2.7.tgz
#Install Spark.
!tar xf spark-3.2.3-bin-hadoop2.7.tgz
```

```
spark-3.2.3-bin-had 100%[===================>] 260.23M  28.0MB/s    in 10s
```

Now we install `PySpark` which allows us to use *Spark* from Python. We must also install `Findspark`, a Python library that helps us to use *Spark* in a NoteBook.

In [3]:
```
!pip install pyspark   #Install PySpark.
!pip install findspark #Install FindSpark.
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting pyspark
  Downloading pyspark-3.3.1.tar.gz (281.4 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 281.4/281.4 MB 4.5 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Collecting py4j==0.10.9.5
  Downloading py4j-0.10.9.5-py2.py3-none-any.whl (199 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 199.7/199.7 KB 15.3 MB/s eta 0:00:00
Building wheels for collected packages: pyspark
  Building wheel for pyspark (setup.py) ... done
  Created wheel for pyspark: filename=pyspark-3.3.1-py2.py3-none-any.whl size=281845512 sha256=38aba41424849c80ad
0f57dcc79c117ff82757633e52c84a6ce2c67f7773e15e
  Stored in directory: /root/.cache/pip/wheels/43/dc/11/ec201cd671da62fa9c5cc77078235e40722170ceba231d7598
Successfully built pyspark
Installing collected packages: py4j, pyspark
Successfully installed py4j-0.10.9.5 pyspark-3.3.1
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting findspark
  Downloading findspark-2.0.1-py2.py3-none-any.whl (4.4 kB)
Installing collected packages: findspark
Successfully installed findspark-2.0.1
```

1.2. First *Spark* Script

Once all the programs and libraries are installed, we must then give values to the environment variables to indicate where Java and *Spark* are installed. We can modify these environment variables when we install *Spark* and Java. The method for giving them values depends on our operating system (Windows, Mac, Linux). However, in these examples, we will do it from Python.

Also, note that we must specify the complete path where both programs are installed.

In [4]:
```
#Set the environment variables.
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/default-java"
os.environ["SPARK_HOME"] = "/content/spark-3.2.3-bin-hadoop2.7"
```

All that remains is to tell `findspark` which folder *Spark* is in so that it can link with Python.

In [5]:
```
#Indicate where Spark is.
import findspark
findspark.init("spark-3.2.3-bin-hadoop2.7")#SPARK_HOME
```

Once we have everything installed and configured in our work environment, we can begin to work with *Spark* from Python. We will have to create a session when at the start of any program in PySpark.

In [6]:
```
#Create a Spark session to start working.
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local[2]").getOrCreate()
```

Now everything is ready for us to start using *Spark*. We will create a small data set (in this case, a set of nucleotides) with a *Pandas* DataFrame. We will convert it to a *Spark* DataFrame to check that everything works. Next, we will show the Dataframe contents.

In [7]:
```
import random
import pandas as pd

#Create a random string of nucleotides.
nucleotideList=[{"ADN": random.choice("ATCG")} for x in range(1000)]

#Convert the nucleotide string to a Pandas DataFrame.
df = pd.DataFrame(nucleotideList)

#Convert the Pandas DataFrame to Spark DataFrame and display it.
ddf = spark.createDataFrame(df)
ddf.show(10)
```

```
+---+
|ADN|
+---+
|  G|
|  T|
|  G|
|  C|
|  A|
|  G|
|  A|
|  A|
|  G|
|  T|
+---+
only showing top 10 rows
```

2. *SPARK* DATA TYPES: RDD, DATASET, AND DATAFRAME

Three data types can be used in Spark with big data: RDD, Dataset, and DataFrame. Thus, in principle, there are three different ways to work in *Spark*, which can be a bit confusing. Let's now take a look at their differences.

2.1. Resilient distributed dataset (RDD)

As we have already seen, resilient distributed datasets (RDD) -a set of objects spread across the various nodes in a cluster- are the basic means of data abstraction in *Spark*. RDDs were used from Spark version 1.0, and regardless of the type of data we use (DataFrame or Dataset), internally, *Spark* always uses RDDs. They are a simple and flexible data type, which allows us to work with unstructured data (i.e., data with no defined structure).

2.2. DataFrame

DataFrame appeared in *Spark* version 1.3 and is a modification of RDD that allows us to define a schema on the data. This means that we can see the data in a table format and thereby making it easier to work with. Another important feature is that it is structured data, and so the algorithms that work with it are more optimized and can run faster than RDD.

2.3. Dataset

Dataset is a modification of DataFrame that appeared in Spark version 1.6 which was designed to detect programming errors earlier. However, the way of working with Dataset is almost identical to the methods used to work with DataFrame.

2.4. Which one should we use?

In this course, we will work with data in a table form (structured data), which is the most common data format in the field of machine learning. Moreover, Dataset does not exist in Python due to the characteristics of this programming language. Therefore, it is best to use DataFrame because is faster than RDD.

The DataFrames in *Spark* are very similar to Pandas DataFrames presented in previous modules, except for one notable difference. *Spark* DataFrames can be spread across multiple computers while Pandas DataFrames can only be present on a single computer. Nonetheless, we can easily and efficiently change one data type to the other data type by employing Apache Arrow, as long as the data fits in the memory of only one computer. Let's see how:

In [8]:
```
# As before, we store the nucleotide string in a Pandas DataFrame.
pdf = pd.DataFrame(nucleotideList)

# Convert the Pandas DataFrame to Spark.
sdf = spark.createDataFrame(pdf)

# Convert the Spark DataFrame to a Pandas DataFrame and show it.
psdf = sdf.select("*").toPandas()
psdf.head()
```

Out[8]:

	ADN
0	G
1	T
2	G
3	C
4	A

3. A *PYSPARK* SESSION.

The gateway to all *Spark* functionality is the `SparkSession` class. To create a session in *Spark*, we use `SparkSession.builder`. In computer science, a session is a semi-permanent interactive information exchange between two or more entities, in our case between *Spark* and Python. The most common attributes used with `SparkSession.builder` are:

- `appName(name)` : Sets a name to the script.
- `config(key, value)` : Sets a value to a configuration property. All the configuration options can be found in the Spark configuration documentation.
- `master(value)` : Indicates the internet address (URL) of the master node of the computer cluster. It can be executed locally with a single execution thread on a personal computer with `local`. With `local[n]` it is executed locally with `n` threads. Finally, if we use `local[*]`, it runs *Spark* on a personal computer with as many threads as the machine will allow.
- `getOrCreate()` : This method gets an existing *Spark* session or, if there are none, it creates one.

SparkSession has been the Python gateway to *Spark* since Apache *Spark* version 2.0. Previously, `SparkContext` or `SQLContext` were used to access the different functionalities.

In [9]:
```
#Create a Spark session to start working.
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .master("local[*]") \
    .appName("PySpark basic example") \
```

```
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

4. READING DATA FROM A FILE

Here we will learn how to load a file in a *Spark* DataFrame. For this purpose, we will work with the problem seen in Module 5, Capsule 2 (*Standard classification methods*)- a supervised learning problem for skin melanoma prediction.

First, let's load the data into Google Colaboratory.

In [10]:
```
#Download the data files in Google Colaboratory.
!wget -nv -c --no-check-certificate 'https://docs.google.com/uc?export=download&id=1J3jo-YRKGX1lBjA0-B8U1ZJibHYEA
!wget -nv -c --no-check-certificate 'https://docs.google.com/uc?export=download&id=1hHQfcvrFa5Jds-9tW_X4sHjKpYKdi

#Show the header of the files and two rows, in order to check that
#they have been downloaded correctly.
!head -3 inmune_X.csv
!head -3 inmune_Y.csv
```

```
2023-01-25 11:58:46 URL:https://doc-0g-90-docs.googleusercontent.com/docs/securesc/ha0ro937gcuc7l7deffksulhg5h7mb
p1/m8r26hollgc491j2jnv13nho1b584p3b/1674647925000/05131985187150583765/*/1J3jo-YRKGX1lBjA0-B8U1ZJibHYEAbmL?e=down
load&uuid=445648f9-ead0-465f-80c1-e18a03725bff [9152821/9152821] -> "inmune_X.csv" [1]
2023-01-25 11:58:47 URL:https://doc-0c-90-docs.googleusercontent.com/docs/securesc/ha0ro937gcuc7l7deffksulhg5h7mb
p1/i4ntdk7hdb62egoq4dfu9jplt10j443j/1674647925000/05131985187150583765/*/1hHQfcvrFa5Jds-9tW_X4sHjKpYKdii9s?e=down
load&uuid=86b935d7-da43-45d8-9c07-54afa66eebb7 [3949/3949] -> "inmune_Y.csv" [1]
,TYRP1,RPS4Y1,KRT6A,XIST,KRT14,KRT16,KRT5,XAGE1B,KRT6B,PAEP,KRT6C,MAGE...
0,2.7504627735116602,-9.25208847558951,0.42007312693424703,8.320289342...
1,-5.566030015996501,-8.78199418294173,0.0260954433791682,6.9285253178...
,RNASEQ-CLUSTER_CONSENHIER
0,immune
1,MITF-low
```

Once the data is loaded onto a disk, we will use a DataFrameReader to read both files.

To read the first file, methods with different options are used. For example, with `load()` we indicate the name of the file, `format()` indicates that it is a CSV file, `option()` is used to get the schema of the data (`inferSchema`) and confirm that the file has a first header row with the name of the columns (`header`). In addition, many more options can be specified, such as encoding, separators, comments, etc.

Furthermore, we can also open different file type formats (`JSON, parquet, orc, libsvm, CSV, text`) locally, or from a database (using `JDBC`) or using HDFS.

In [11]:
```
#Read the input variables.
dfX = spark.read \
    .format("csv") \
    .option("inferSchema",True) \
    .option("header", True) \
    .load("inmune_X.csv")
dfX.show(3)

#Read the output variable.
dfY = spark.read.csv('inmune_Y.csv',
                     inferSchema=True,
                     header=True)
dfY.show(3)
```

```
+---+------------------+-----------------+------------------+-------...
|_c0|             TYRP1|           RPS4Y1|             KRT6A|       ...
+---+------------------+-----------------+------------------+-------...
|  0| 2.7504627735116602|-9.25208847558951|0.42007312693424703|  8.320...
|  1| -5.566030015996501|-8.78199418294173| 0.0260954433791682|  6.928...
|  2|-4.9072628904726505|-1.86658582313499| 0.8504174516070371|-3.0423...
+---+------------------+-----------------+------------------+-------...
only showing top 3 rows

+---+-------------------------+
|_c0|RNASEQ-CLUSTER_CONSENHIER|
+---+-------------------------+
|  0|                   immune|
|  1|                MITF-low|
|  2|                MITF-low|
+---+-------------------------+
only showing top 3 rows
```

5. WORKING WITH DATAFRAMES

Now that we have our data in *Spark*, let's learn how to work with it.

5.1. Exploring DataFrames

To see how many rows we have, we can use the `.count()` method. To see the number of columns (variables) we have the `.columns` property to see their size. To display the name and the data type of each column (that is, the schema), we can use the `.printSchema()` method.

Finally, to show the DataFrame we can use the `.show()` method, as before. We can also use it in combination with the `.describe()` method that calculates a statistical summary of the DataFrame.

In [12]:
```
#Display the number of input variables and the number of instances.
print ("We have", len(dfX.columns)-1, "input variables and" ,
       dfX.count(), "instances.")

#Show the schema and a statistical summary of the output variable.
print ("\nSchema of dfY:")
dfY.printSchema()
print ("\nStatistical summary of dfY:")
dfY.describe().show()
```

```
We have 1491 input variables and 336 instances.

Schema of dfY:
root
 |-- _c0: integer (nullable = true)
 |-- RNASEQ-CLUSTER_CONSENHIER: string (nullable = true)


Statistical summary of dfY:
+-------+-----------------+-------------------------+
|summary|              _c0|RNASEQ-CLUSTER_CONSENHIER|
+-------+-----------------+-------------------------+
|  count|              336|                      336|
|   mean|            167.5|                     null|
| stddev|97.13907555664713|                     null|
|    min|                0|                 MITF-low|
|    max|              335|                   immune|
+-------+-----------------+-------------------------+
```

To work with DataFrames we can select one or several columns using the `select()` method. To identify the columns, we can use a string with the column name; the column name as a property of the DataFrame;the DataFrame as a dictionary, indicating the column name as the key; or the DataFrame indicating the column number and considering that we start counting from zero.

In [13]:
```
#There are several ways to get second, fourth and sixth input variables.
dfX.select('TYRP1','KRT6A','KRT14').show(5) #column name string.
dfX.select(dfX.TYRP1,dfX.KRT6A,dfX.KRT14).show(5) #column name as a property.
dfX.select(dfX['TYRP1'],dfX['KRT6A'],dfX['KRT14']).show(5) # column name as dictionary key.
dfX.select(dfX[1],dfX[3],dfX[5]).show(5) #using the column number.
```

```
+------------------+------------------+------------------+
|             TYRP1|             KRT6A|             KRT14|
+------------------+------------------+------------------+
| 2.7504627735116602|0.42007312693424703|-2.8987916629252592|
| -5.566030015996501| 0.0260954433791682| -0.531967458719549|
|-4.9072628904726505| 0.8504174516070371|  0.375261221152995|
|   5.54681507588922|  -2.37481458991564|-0.7988656239782429|
| 7.0367948190278495| 3.0978539408002996| 1.4826106729146697|
+------------------+------------------+------------------+
only showing top 5 rows

+------------------+------------------+------------------+
|             TYRP1|             KRT6A|             KRT14|
+------------------+------------------+------------------+
| 2.7504627735116602|0.42007312693424703|-2.8987916629252592|
| -5.566030015996501| 0.0260954433791682| -0.531967458719549|
|-4.9072628904726505| 0.8504174516070371|  0.375261221152995|
|   5.54681507588922|  -2.37481458991564|-0.7988656239782429|
| 7.0367948190278495| 3.0978539408002996| 1.4826106729146697|
+------------------+------------------+------------------+
```

```
only showing top 5 rows

+------------------+------------------+------------------+
|             TYRP1|              KRT6A|             KRT14|
+------------------+------------------+------------------+
| 2.7504627735116602|0.42007312693424703|-2.8987916629252592|
| -5.566030015996501| 0.0260954433791682| -0.531967458719549|
|-4.9072628904726505| 0.8504174516070371|  0.375261221152995|
|   5.54681507588922|  -2.37481458991564|-0.7988656239782429|
| 7.0367948190278495| 3.0978539408002996| 1.4826106729146697|
+------------------+------------------+------------------+
only showing top 5 rows

+------------------+------------------+------------------+
|             TYRP1|              KRT6A|             KRT14|
+------------------+------------------+------------------+
| 2.7504627735116602|0.42007312693424703|-2.8987916629252592|
| -5.566030015996501| 0.0260954433791682| -0.531967458719549|
|-4.9072628904726505| 0.8504174516070371|  0.375261221152995|
|   5.54681507588922|  -2.37481458991564|-0.7988656239782429|
| 7.0367948190278495| 3.0978539408002996| 1.4826106729146697|
+------------------+------------------+------------------+
only showing top 5 rows
```

We can also filter the data we want to display with the `filter()` method, group common values with the `groupby()` method, or sort the data with the `sort()` method.

In [14]:
```
#Obtain the rows where the value of TYRP1 is positive.
dfX.filter(dfX.TYRP1>0).show(5)

#Group the different values of the output variable.
dfY.groupby(dfY[1]).count().show(5)

#Sort the values of the TYRP1 variable.
dfX.select(dfX[0], 'TYRP1').sort('TYRP1').show(5)
```

```
+---+------------------+------------------+------------------+-------...
|_c0|             TYRP1|            RPS4Y1|              KRT6A|       ...
+---+------------------+------------------+------------------+-------...
|  0|2.7504627735116602| -9.25208847558951|0.42007312693424703|8.32028...
|  3|  5.54681507588922|-7.544166181117211|  -2.37481458991564| 9.4757...
|  4|7.0367948190278495| 0.983460294336822| 3.0978539408002996| -3.487...
|  6|  5.34314482098534|  3.06441128263618|-0.0722837209380733| -3.487...
| 11|  6.94320218627926| 0.504180947256865|   5.12460456547909| 8.4293...
+---+------------------+------------------+------------------+-------...
only showing top 5 rows

+-------------------------+-----+
|RNASEQ-CLUSTER_CONSENHIER|count|
+-------------------------+-----+
|                MITF-low|  168|
|                  immune|  168|
+-------------------------+-----+

+---+----------------+
|_c0|           TYRP1|
+---+----------------+
| 41|-9.3022776907826|
|110|-9.3022776907826|
| 14|-9.3022776907826|
|210|-9.3022776907826|
| 73|-9.3022776907826|
+---+----------------+
only showing top 5 rows
```

5.2 Modifying DataFrames

With the `withColumn()` method, we can change the column type, change its values, and can also be used to create a new column. To rename a column, we will use the `withColumnRenamed()` method.

To delete one or more columns the `drop()` method canbe used, and with `dropDuplicates()` we can delete duplicate rows, leaving only one example of each.

With the `join()` method we can join tables.

In [15]:

```
#Change the first column of dfY from Integer to Double.
dfY.withColumn("_c0",dfY._c0.cast("Double")).show(5)

#Multiply the values of the first column of dfY by two.
dfY.withColumn("_c0",dfY._c0*2).show(5)

#Add a column with the values of the first column of dfY multiplied by 3.
dfY.withColumn("three",dfY._c0*3).show(5)

#Change the name of the output variable.
dfY=dfY.withColumnRenamed("RNASEQ-CLUSTER_CONSENHIER","class")
dfY.show(5)

#Remove the first column of dfY.
dfY.drop('_c0').show(5)

#Delete the first column of dfY and then remove repeated rows.
dfY.drop('_c0').dropDuplicates().show(5)

#Add the output varible to the input variable table.
dfX.join(dfY, dfX._c0 == dfY._c0).show(5)
```

```
+---+------------------------+
|_c0|RNASEQ-CLUSTER_CONSENHIER|
+---+------------------------+
|0.0|                  immune|
|1.0|                MITF-low|
|2.0|                MITF-low|
|3.0|                  immune|
|4.0|                  immune|
+---+------------------------+
only showing top 5 rows

+---+------------------------+
|_c0|RNASEQ-CLUSTER_CONSENHIER|
+---+------------------------+
|  0|                  immune|
|  2|                MITF-low|
|  4|                MITF-low|
|  6|                  immune|
|  8|                  immune|
+---+------------------------+
only showing top 5 rows

+---+------------------------+-----+
|_c0|RNASEQ-CLUSTER_CONSENHIER|three|
+---+------------------------+-----+
|  0|                  immune|    0|
|  1|                MITF-low|    3|
|  2|                MITF-low|    6|
|  3|                  immune|    9|
|  4|                  immune|   12|
+---+------------------------+-----+
only showing top 5 rows

+---+--------+
|_c0|   class|
+---+--------+
|  0|  immune|
|  1|MITF-low|
|  2|MITF-low|
|  3|  immune|
|  4|  immune|
+---+--------+
only showing top 5 rows

+--------+
|   class|
+--------+
|  immune|
|MITF-low|
|MITF-low|
|  immune|
|  immune|
+--------+
only showing top 5 rows

+--------+
|   class|
+--------+
|MITF-low|
|  immune|
+--------+
```

```
+---+------------------+-----------------+------------------+------...
|_c0|             TYRP1|            RPS4Y1|             KRT6A|      ...
+---+------------------+-----------------+------------------+------...
|  0| 2.7504627735116602| -9.25208847558951|0.42007312693424703|  8.32...
|  1| -5.566030015996501| -8.78199418294173| 0.0260954433791682|  6.92...
|  2|-4.9072628904726505| -1.86658582313499| 0.8504174516070371|-3.042...
|  3|   5.54681507588922|-7.544166181117211|  -2.37481458991564|   9.4...
|  4| 7.0367948190278495| 0.983460294336822| 3.0978539408002996|   -3....
+---+------------------+-----------------+------------------+------...
only showing top 5 rows
```

6. APPLYING SQL QUERIES TO DATAFRAMES.

We can work with DataFrames using Structured Query Language (SQL) queries in the same way we previously worked with DataFrames in *Spark*. Since this is an introductory course, we do not assume any SQL knowledge here, but if you have knowledge of this language, using it may be the most convenient option. To work with DataFrames as tables, we will have to define the views with the `createOrReplaceTempView()` method. Once we have defined a view, we can use the `sql()` method to work comfortably using SQL.

In [16]:

```
#Register the DataFrame as a SQL temporary view so that we can query it using SQL.
dfX.createOrReplaceTempView("inputVarsTable")
dfY.createOrReplaceTempView("outputVarTable")

#Show the tables.
spark.sql("SHOW TABLES").show()

#Equivalent to dfX.select('TYRP1','KRT6A','KRT14').show(5)
sqlDF = spark.sql("SELECT TYRP1,KRT6A,KRT14 FROM inputVarsTable")
sqlDF.show(5)

#Equivalent to dfY.groupby(dfY[1]).count().show(5)
spark.sql("SELECT class, count(class) FROM outputVarTable GROUP BY class").show(5)

#Equivalent to dfX.select(dfX[0], 'TYRP1').sort('TYRP1').show(5)
spark.sql("SELECT _c0,TYRP1 FROM inputVarsTable ORDER BY TYRP1").show(5)

#Equivalent to  dfX.join(dfY, dfX._c0 == dfY._c0).show(5)
spark.sql("SELECT * FROM inputVarsTable,outputVarTable WHERE inputVarsTable._c0 == outputVarTable._c0").show(5)
```

```
+---------+-------------+-----------+
|namespace|    tableName|isTemporary|
+---------+-------------+-----------+
|         |inputvarstable|       true|
|         |outputvartable|       true|
+---------+-------------+-----------+

+------------------+------------------+------------------+
|             TYRP1|             KRT6A|             KRT14|
+------------------+------------------+------------------+
| 2.7504627735116602|0.42007312693424703|-2.8987916629252592|
| -5.566030015996501| 0.0260954433791682| -0.531967458719549|
|-4.9072628904726505| 0.8504174516070371|  0.375261221152995|
|   5.54681507588922|  -2.37481458991564|-0.7988656239782429|
| 7.0367948190278495| 3.0978539408002996| 1.4826106729146697|
+------------------+------------------+------------------+
only showing top 5 rows

+--------+------------+
|   class|count(class)|
+--------+------------+
|MITF-low|         168|
|  immune|         168|
+--------+------------+

+---+---------------+
|_c0|           TYRP1|
+---+---------------+
| 41|-9.3022776907826|
|110|-9.3022776907826|
| 14|-9.3022776907826|
|210|-9.3022776907826|
| 73|-9.3022776907826|
+---+---------------+
only showing top 5 rows

+---+------------------+-----------------+------------------+------...
|_c0|             TYRP1|            RPS4Y1|             KRT6A|      ...
```

```
+---+------------------+------------------+------------------+------...
|  0| 2.7504627735116602| -9.25208847558951|0.42007312693424703|  8.32...
|  1| -5.566030015996501| -8.78199418294173| 0.0260954433791682|  6.92...
|  2|-4.9072628904726505| -1.86658582313499| 0.8504174516070371|-3.042...
|  3|   5.54681507588922|-7.544166181117211|  -2.37481458991564|   9.4...
|  4| 7.0367948190278495| 0.983460294336822| 3.0978539408002996|   -3....
+---+------------------+------------------+------------------+------...
only showing top 5 rows
```

BIBLIOGRAPHICAL REFERENCES

- Karlijn Willems. "Apache Spark in Python: Beginner's Guide". (2017). [Accessed 7 June 2020]. Available from: https://www.datacamp.com/community/tutorials/apache-spark-python
- Achilleus. "A tale of Spark Session and Spark Context". (2019). [Accessed 7 June 2020]. Available from: https://medium.com/@achilleus/spark-session-10d0d66d1d24
- Chandan Prakash. "Apache Spark : RDD vs DataFrame vs Dataset". (2016). [Accessed 7 June 2020]. Available from: https://www.linkedin.com/pulse/apache-spark-rdd-vs-dataframe-dataset-chandan-prakash
- Databricks. "Introduction to DataFrames - Python". (2020). [Accessed 7 June 2020]. Available from: https://docs.databricks.com/spark/latest/dataframes-datasets/introduction-to-dataframes-python.html#
- The Apache Software Foundation. "Spark SQL, DataFrames and Datasets Guide". (2020). [Accessed 7 June 2020]. Available from: https://spark.apache.org/docs/latest/sql-programming-guide.html#spark-sql-guide

Additional References

- The Apache Software Foundation." Welcome to Spark Python API Docs!". (2020). [Accessed 7 June 2020]. Available from: https://spark.apache.org/docs/latest/api/python/index.html
- Evan Heitman. "A Neanderthal's Guide to Apache Spark in Python". (2019). [Accessed 7 June 2020]. Available from: https://towardsdatascience.com/a-neanderthals-guide-to-apache-spark-in-python-9ef1f156d427
- Akbani, R., Akdemir, K. C., Aksoy, B. A., Albert, M., Ally, A., Amin, S. B., et. al. (2015). "Genomic classification of cutaneous melanoma". Cell, 161(7), 1681-1696.
- Jules Damji. "A Tale of Three Apache Spark APIs: RDDs vs DataFrames and Datasets". (2016). [Accessed 8 June 2020]. Available from: https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html
- Geeky Theory. "Apache Spark: qué es y cómo funciona". (2015). [Accessed 8 June 2020]. Available from: https://geekytheory.com/apache-spark-que-es-y-como-funciona