

My color codes:

```
find . -name '*solution*' : shell instruction
:quit : spark-shell instruction.
variable : file name or variable
```

I exhort you to peruse the [doc de spark](#). Nonetheless, to save time, I also provide links toward the book co-authored by spark's designer: Spark - The Definitive Guide: Big Data Processing.

In this lab, we shall work on a dataset retrieved from [Chicago city's data portal](#). The dataset records informations about food establishment inspections carried by the city's public health services. In particular, for each inspection one records the result of that inspection as well as a list of violations. In this lab, you must write a [Logistic regression](#) program in Spark, that will predict inspection results from the list of violations. *Warning: this lab was adapted from a Microsoft Azure tutorial. Several other solutions may also be available online. I strongly recommend that you do not check any online solution before you complete the lab.*

For our first steps on spark, we shall first write a traditional wordcount application (for which you should try to discover Scala) on a tiny dataset: a medieval poem from the 12<sup>th</sup> century.

## Environment

On the machines at PUIO, Java 11 is installed by default (`printenv | grep -E 'JAVA'`), but Spark needs Java 8. So you have to use the proper JAVA version through

```
JAVA_HOME=/opt/jdk1.8.0_221
```

Alternatively, you may launch a jupyter pyspark notebook in docker through:

```
docker run -it -p 8888:8888 jupyter/pyspark-notebook
```

but I do not advise it for now.

## 1 First steps with the REPL, Spark queries on RDDs

1. Retrieve the dataset: `wget https://www.lri.fr/~groz/documents/lai-eliduc.txt`
2. Launch the spark interpreter: (`spark-shell` for scala, `pyspark` for python).

To obtain syntax highlighting in the Scala shell:

```
--conf spark.driver.extraJavaOptions="-Dscala.color"
```

For python, one had better switch the python version used by pyspark for `ipython`. This would be achieved through `export PYSPARK_PYTHON=ipython` but this raises errors that I have not been able to solve when using UDF, so for the second part of the lab (food inspection) we will have to rely on the standard shell. Untested: another solution may be to try `PYSPARK_DRIVER_PYTHON=ipython ./bin/pyspark` or `PYSPARK_DRIVER_PYTHON=jupyter ./bin/pyspark`.

Shell is a bit verbose. To eliminate warnings, you may replace the default settings by executing:

```
sc.setLogLevel("ERROR")
```



3. Load a Dataframe `df1` from file `food.csv`. Then, display the Dataframe schema using `.show()` and `.printSchema()`
4. Create a dataframe `inspections` by projecting on the columns that we shall use: 'Inspection ID', 'DBA Name', 'results' then removing from `df1` all lines containing a "NULL". Make sure you drop the nulls after projecting, otherwise you will end up with an empty dataframe. `.dropna()`  
Would you save or waste time if you loaded the `inspections` dataframe from file `food.csv` instead of `df1`? Explain. p110
5. Compute from `inspections` the list `nbre` of possible inspection results, and the number of inspections yielding such results, ordered by increasing number of inspections. Then:
  - Display the first 4 lines of the result: what difference between `.take(4)` and `.show(4)`?
  - Display the execution plan `nbre` with `explain(true)`.
  - check in the web interface the DAG of tasks.
  - Display stats about `nbre` using `.describe()`
  - Visualize the number of inspections for each possible result in a pie chart using matplotlib.

For the list of available methods in Dataframes: [the doc](#).

`.sort()...`  
p36

The web interface web is available on port 4040, so use the URL <http://localhost:4040>.

The function `toPandas()` is not available on the relatively outdated version of spark on our machines. To extract the labels and values for matplotlib, you may therefore: return the df as a python collection – it will be a collection of "Row" objects. Then access the "result" field of each row and convert it to a string.`plt.axis('equal')`

```
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
labels = ...
sizes = ...
plt.pie(sizes, labels=labels, autopct='%1.1f%%')
plt.axis('equal')
plt.savefig('pie.pdf')
```

## 3 Predictions with spark.

We next turn on to the MLlib library of Spark in order to perform predictions from our data.

### 3.1 Labelling data: defining categories

1. Observe a few lines of the "results" and "violations" columns.


Logistic regression is a binary “classification” method, so we will group results into 2 categories using the following function:

```
def labelForResults(s):
    if s == 'Fail':
        return 0.0
    elif s == 'Pass w/ Conditions' or s == 'Pass':
        return 1.0
    else:
        return -1.0
```

2. Register the above function as a udf, and use it to transform the data: we want to obtain a 2-column DataFrame `labeledData` : the columns are `label` and `violations`, where category `label` takes values 0.0 or 1.0.

`udf()` p121

⚠ `label` must be of numeric data type, so we may use  
`monudf = udf(mafonction, DoubleType())`

3. We naturally distinguish two phases; a training phase and a validation phase. To keep things simple, we split our data in 2: a `training` Dataframe containing 25% of `labeledData` records will be used to train the model, whereas Dataframe `validationDf` containing the remaining records will be used for validation. Partition `labeledData` records through random sampling . Use the Spark function `randomSplit` , taking 105 as a ( seed) to initialize the random generator.

`.randomSplit()`  
p409

## 3.2 Defining the model: specifying predictive variables and tuning parameters

1. We now have data labeled with their category. We still have to extract from the text field `violations` the variables from which our model will build predictions. For this, we convert that field to a vector of numbers. The logistic regression will then be applied to those vectors.

We define a 3-steps pipeline:

- the first step splits `violations` into a sequence of words (Tokenizer)
- the second step converts the sequences to a frequency vector (each word gets assigned an index, then we map each list to its corresponding frequency vector simply by counting the occurrences of each word)
- the last step applies the linear regression (use 10 iterations, with regularisation parameter equal 0.01)

The book is not the best source of information on those aspects, better check [the spark doc](#), which uses logistic regression as an example for ML pipelines.

2. Train your pipeline on the training data, then validate the pipeline on test data.

## 4 Executing a spark application.

1. Instead interacting with spark through the REPL, we next write an independent application. Start with a simple application: just adapt SimpleApp from [spark doc](#), or reuse your wordcount code from above to write an application that takes as parameters the name of input and output files:

If you want to use scala, check the [spark doc](#) for indications about how to compile the jar with dependencies.

```
/spark/bin/spark-submit \  
--class "SimpleApp" \  
--master local[1] \  
target/scala-2.11/simple-project_2.11-1.0.jar
```

We will not bother to remove warnings as long as there are no error messages.

2. Implement the above logistic regression in an independent application that writes the result into files. Execute locally on 1 processor, then on several. Compare running times

You may use UNIX instructions `time` and `lscpu`.

`time` displays running time (real and cumulative): `time /spark/bin/spark-submit...`

`lscpu` displays the number of available processors on your machine. *Do not use all processors, for instance with 8 processors, limit Spark to 6 processors.*

:

```
for i in {1..10}; do cat fichier.csv >> fichier10.csv; done
```