# Algoritmos para Big Data
# Lab class handout #1

**Data processing with Apache Spark** **2024/2025**

## Goals

This lab class aims to introduce basic PySpark operations to process data that was stored in csv or json files.

Note: It is acceptable that the work to be undertaken may require some time beyond the class. If that is the case, work should continue outside of class. And if needed, assistance will be provided during the office hours established or to be agreed with the lecturer.

## Supporting information

- Course slides
- PySpark Overview
- Plotly Open Source Graphing Library for Python

## Introduction

Built upon the previous introductory lab class – where the main goal was to show what can be achieved with Apache Spark in a programming environment set by VS Code and Jupyter notebooks – this class is focus on the sequencing of basic data preprocessing operations, in particular from cleaning to data aggregation.

Alongside this handout, that describes some tasks to accomplish, a notebook is also provided, that it is expected to be completed with the implementation of the tasks.

As for the datasets needed, the two files can be downloaded from, respectively [1]:

`https://bigdata.iscte-iul.eu/datasets/iot_devices.json`
`https://bigdata.iscte-iul.eu/datasets/fire-calls.csv`

## Tasks to accomplish

### A. Data ingestion

In this initial task, and after setting up a *SparkSession*, data will be read and stored in DataFrames. Then it follows an initial checking of the data. Hence, the operations to perform are the following:

1. Download the datafiles mentioned above and store them in a proper directory, within the working directory of your Pyspark instalation.

[1] To download a file in the VS Code Terminal, one can use the command `wget` *weblink*

2. After setting up the basic Spark imports that are required, create a *SparkSession* to work with and then read both datasets. The data must be stored in two DataFrames, respectively. For the sake of explanation, let us call them `df_iot` and `df_calls`.

3. Check the contents of both `df_iot` and `df_calls`, namely regarding the schema, the number of rows, as well as showing some rows of data. Hence, you are able to present the basic structures and data types of the data (columns and rows).

4. Just to grab internal information of a DataFrame, store the first row of `df_iot` in a variable and validate the kind of object that was obtained.

5. Repeat the previous point but dealing with the first five rows at once instead of the first one.

## B. Data cleaning

In this task, the focus is on guaranteeing some data quality via cleaning operations. For example, filling in missing values; removing outliers and correcting erroneous, irrelevant, or duplicate parts of the data, etc. Execute the following instructions:

1. For both `df_iot` and `df_calls`, check if there are duplicated rows or not. If there are, then remove the duplicates (only one will stay in the DataFrame). Hence, each data point will be unique.

2. Turning now the attention to missing values, for each column in `df_calls` figure out the number of values that are nulls. Likewise for `df_iot`.

3. Get rid of those columns with so many nulls that we can assume they are worthless for further data analysis. These columns are `AvailableDtTm`, `OriginalPriority` and `CallTypeGroup`.

4. Repeat now the point above in respect to checking the nulls in the columns.

5. Finally, drop any row containing at least one null value. Overall, how many rows have been left?

6. Leaving aside a more detailed analysis of the values in the DataFrames, therefore, no need to identify or fix major mistakes in data, let us focus on the formats of data. In particular, columns that were supposed to be of type *timestamp*.

   Verify that in `df_calls` the columns `CallDate` and `WatchDate` are not properly set. Hence, create two new corresponding columns but properly set (Hint: use the function `to_timestamp` with format `MM/dd/yyyy`)

## C. Data transformation

Since the data is fairly cleaned, and with proper datatypes it seems, let us make some improvements like, for example, converting data types into a more suitable format for analysis, or adding derived columns.

Due to teaching purposes, the focus now is on `df_calls` so we leave further work on `df_iot` as an additional exercise. Hence, we ask you to execute the following operations:

1. Add two new columns to `df_calls`, with information respectively about the month of the call and the day of the week of the call (Hint: use the functions `month` and `dayofweek`).

2. Change the name of the column `NumAlarms` for a better name, like `NumberAlarms`.

3. Make sure that the data is sorted by the column `CallNumber` but the highest number is at the top, that is, in the first row.

### D. Data aggregation

In this task, we will carry out simple basic aggregation, solely with the purpose of summarizing data via simple statistical or mathematical operations. Then, we will look at enhanced grouped aggregation, for which the drawing of graphs will help to understand the results. All data aggregation is to be applied solely to `df_calls`.

1. For the column `Delay`, show independently the minimum, maximum, average, sum, as well as the existing number of rows.
2. Provide a summary of major statistical measures – count, mean, stddev, min, max – in respect to all the columns (Hint: use the function `describe`).
3. Create a grouped aggregation over the column `CallType` that can show the following columns, besides `CallType`: `avg(CallDateMonth)`, `avg(Delay)`, and `avg(CallDateWeekDay)`.
4. Repeat the previous point but the output being ordered by `avg(Delay)` in descending order.
5. Create a grouped aggregation over two columns, first `CallType` and then `Neighborhood`, that can show the following columns, besides `CallType` and `Neighborhood`:

   `count(CallNumber)` with alias `CountCalls`, `avg(Delay)` with alias `AvgDelay`,

   `min(Delay)` with alias `MinDelay`, and `max(Delay)` with alias `MaxDelay`.

   The output should be ordered by the two columns upon which the aggregation was applied to, in descending order: first `CallType` and then `Neighborhood`.
6. Finally, create a bar plot using *plotly.express* [2], that shows the counting of each occurrence in the column `CallType`. The bars should be sorted according to number of occurrences, in descending order. (Hint: first, the PySpark aggregation is created; then, using the function `toPandas`, a corresponding Pandas dataframe is created, so it can be used by the function `bar` from *plotly.express*).

## Additional exercise

Given the final `df_calls` from task D, provide answers to the following questions:

1. How many distinct types of calls were made?
2. What are the distinct types of calls that were made?
3. Find out the calls which delay is bigger that 3.0?
4. What are the most common call types, listed in descending order by counting?
5. What week of the year in the most recent year has more calls?

[2] See the link provided in the initial section of supporting information.