

(7082CEM)

Coursework

Demonstration of a Big Data Program

MODULE LEADER: Dr. Marwan Fuad

Student Name: Georgios Chantziplakis

SID: 10454439

“Data Exploratory for countries based on their Status and forecast model and
visualization for Life Expectancy”

I can confirm that all work submitted is my own: Yes

Introduction

This coursework is part of the module Big Data Management & Data Visualization (7082 CEM) and its main goal is to apply Machine Learning techniques to get information from a Dataset and use Visualization tools to present some findings.

Big Data analysis has become one of the most developed fields nowadays and that is because in modern societies the collected data that we get from electronic devices and the Internet is supernumerary and massive in size. The high volume of the interconnected machines on the web leads to the storage of millions of data every day that come in different structures and from different sources. As a result, the ability to handle them and gain important information that can be used in real-life problems is getting too difficult and complicated.

However, new methods and modern software have been developed in order to ease the complexity of Big Data Analysis and benefit from the data. Apache Hadoop and Apache Spark, which is also the main focus of the module, are some of this software that has been advanced to handle Big datasets. The name Big Data can easily be correlated with size, but the truth is that there is no clear definition of Big Data. Instead, Big Data analysis depends on the computational resources that are available for the analysis.

Apache Hadoop software allows for the distributed process of big datasets. In that way, we can deal with large information since it is equal distributed in our system making the process much easier to handle in different nodes. The Hadoop Ecosystem consists of different projects, each one of which is ideal for a task regarding data exploratory and data analysis.

(Apache Software Foundation, 2020)

Apache Spark is a distributed analytics engine build for big datasets providing speed and flexibility. The main reason we use Spark instead of Hadoop is because of its in-memory computation use and the ability to use different API libraries for ease of use. ***(Apache Spark, 2020)***

In this coursework, Pyspark is being used which is a Python API for Spark. For data exploration, data cleaning, and Machine Learning techniques that are being applied, Jupyter Notebook is employed because is a friendly web environment that allows us to analyze our data with ease. The dataset is a CSV file so for that reason SparkSQL is being utilized which is ideal for managing structured data.

For visualization, Tableau 2020.3 is being used which is a powerful tool that allows its users to explore data and have a finer picture of them so as to explain better any findings. Some of the exploratory analysis in this coursework has been done with the aim of Tableau because is an easy way to identify clusters and missing values easily.

Parts of the coursework marked with () indicate that they will be discussed in the end*

Installation of PySpark

- The installation has been done on a machine with OS Ubuntu 20.04 LTS which has an i7 processor and 12GB of RAM.

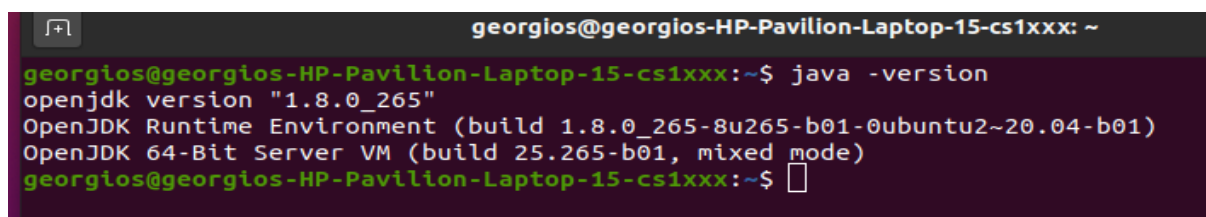
1. Installation of necessary software in Linux.

After Ubuntu was installed, the following commands were run:

'sudo apt-get update'

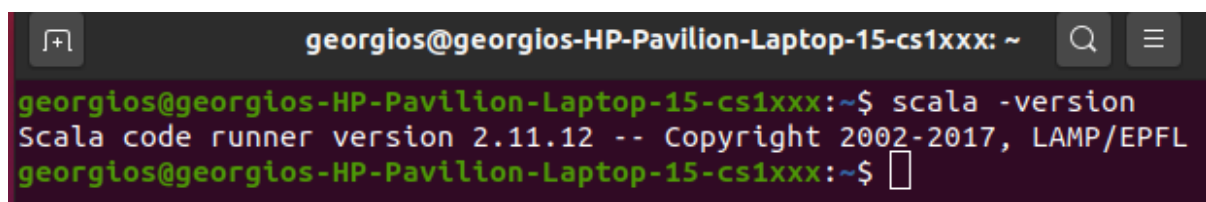
'sudo apt-get upgrade'

So, these are the versions of the software that is being used for the coursework.

A terminal window with a dark purple background. The title bar reads 'georgios@georgios-HP-Pavilion-Laptop-15-cs1xxx: ~'. The prompt is 'georgios@georgios-HP-Pavilion-Laptop-15-cs1xxx:~\$'. The command 'java -version' has been entered, and the output is: 'openjdk version "1.8.0_265"', 'OpenJDK Runtime Environment (build 1.8.0_265-8u265-b01-0ubuntu2~20.04-b01)', and 'OpenJDK 64-Bit Server VM (build 25.265-b01, mixed mode)'. The cursor is on the line following the output.

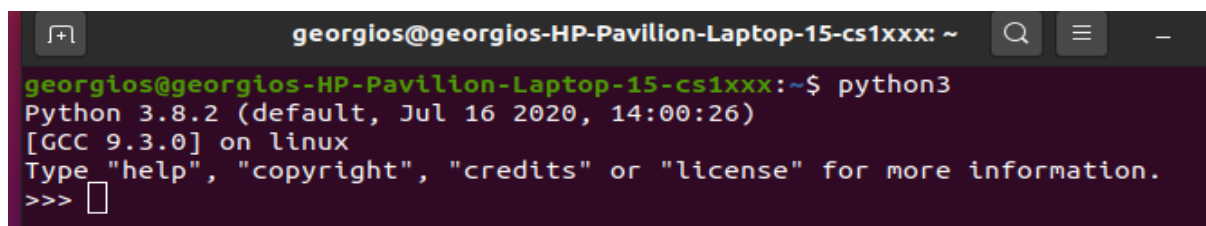
```
georgios@georgios-HP-Pavilion-Laptop-15-cs1xxx:~$ java -version
openjdk version "1.8.0_265"
OpenJDK Runtime Environment (build 1.8.0_265-8u265-b01-0ubuntu2~20.04-b01)
OpenJDK 64-Bit Server VM (build 25.265-b01, mixed mode)
georgios@georgios-HP-Pavilion-Laptop-15-cs1xxx:~$
```

Figure 1. Java Version

A terminal window with a dark purple background. The title bar reads 'georgios@georgios-HP-Pavilion-Laptop-15-cs1xxx: ~'. The prompt is 'georgios@georgios-HP-Pavilion-Laptop-15-cs1xxx:~\$'. The command 'scala -version' has been entered, and the output is: 'Scala code runner version 2.11.12 -- Copyright 2002-2017, LAMP/EPFL'. The cursor is on the line following the output.

```
georgios@georgios-HP-Pavilion-Laptop-15-cs1xxx:~$ scala -version
Scala code runner version 2.11.12 -- Copyright 2002-2017, LAMP/EPFL
georgios@georgios-HP-Pavilion-Laptop-15-cs1xxx:~$
```

Figure 2. Scala Version

A terminal window with a dark purple background. The title bar reads 'georgios@georgios-HP-Pavilion-Laptop-15-cs1xxx: ~'. The prompt is 'georgios@georgios-HP-Pavilion-Laptop-15-cs1xxx:~\$'. The command 'python3' has been entered, and the output is: 'Python 3.8.2 (default, Jul 16 2020, 14:00:26)', '[GCC 9.3.0] on linux', and 'Type "help", "copyright", "credits" or "license" for more information.'. The cursor is on the line following the output.

```
georgios@georgios-HP-Pavilion-Laptop-15-cs1xxx:~$ python3
Python 3.8.2 (default, Jul 16 2020, 14:00:26)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Figure 3. Python Version

2. Installation of Spark (Karthikeyan Mohanra, n.d.)

The steps followed for installing Spark are explained below.

i.

First, **Apache Spark version 3.0.1 pre-built for Apache Hadoop 2.7** was downloaded to the **'home'** directory from the official webpage.

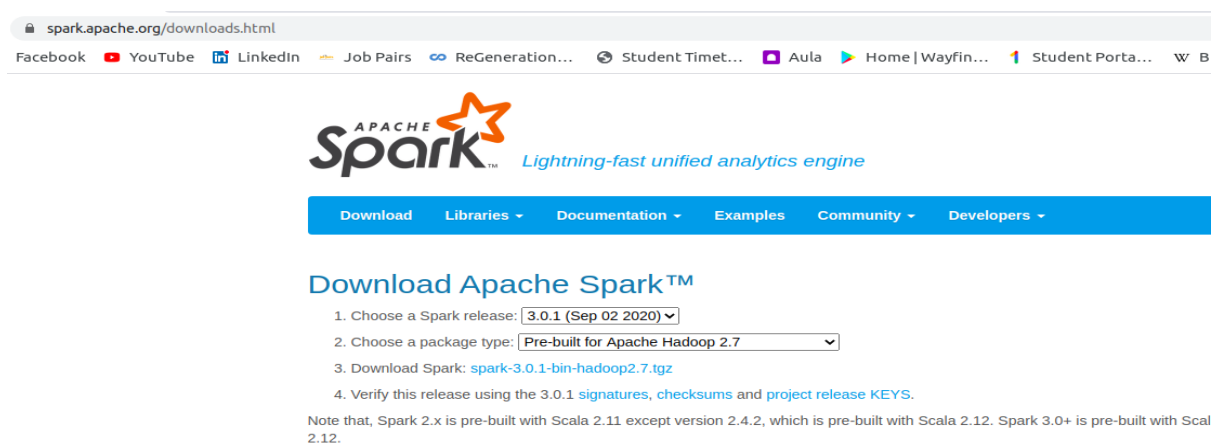


Figure 4. Spark Download

ii. Next, the file was extracted by using the command:

'sudo tar -zxvf spark3.0.1-bin-hadoop2.7.tgz'

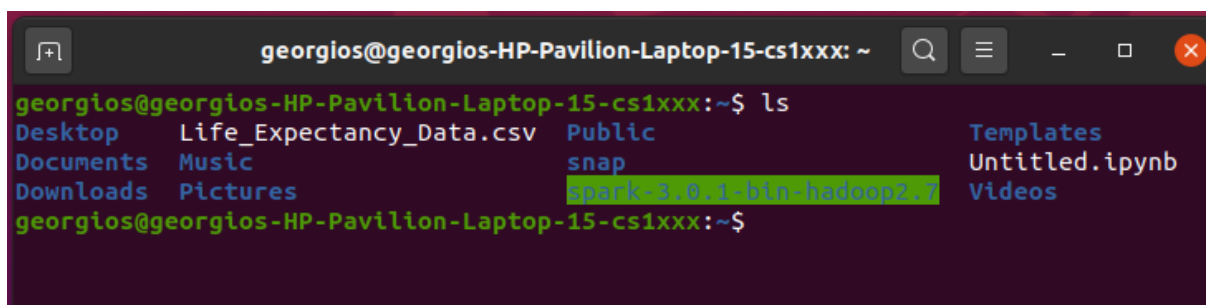


Figure 5. Spark extracted in the 'home' directory

- ```
GNU nano 4.8 /home/georgios/.bashrc Modified

#####
Path for PySpark
#####

export SPARK_HOME="/home/georgios/spark-3.0.1-bin-hadoop2.7"
export PATH=$SPARK_HOME:$PATH
export PYTHONPATH=$SPARK_HOME/python:$PYTHONPATH

#####
Jupyter Notebook
#####

export PYSARK_DRIVER_PYTHON="jupyter"
export PYSARK_DRIVER_PYTHON_OPTS="notebook"
export PYSARK_PYTHON=python3
```

```
georgios@georgios-HP-Pavilion-Laptop-15-cs1xxx:~$ ls
Desktop Music snap Untitled1.ipynb Videos
Documents Pictures spark-3.0.1-bin-hadoop2 Untitled2.ipynb
Downloads Public Templates Untitled.ipynb
georgios@georgios-HP-Pavilion-Laptop-15-cs1xxx:~$ cd spark-3.0.1-bin-hadoop2.7/
georgios@georgios-HP-Pavilion-Laptop-15-cs1xxx:~/spark-3.0.1-bin-hadoop2.7$ cd bin/
georgios@georgios-HP-Pavilion-Laptop-15-cs1xxx:~/spark-3.0.1-bin-hadoop2.7/bin$./spark-shell --version
20/09/30 13:03:34 WARN Utils: Your hostname, georgios-HP-Pavilion-Laptop-15-cs1xxx resolves to a loopback ad
dress: 127.0.0.1; using 100.76.123.135 instead (on interface wlo1)
20/09/30 13:03:34 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.apache.spark.unsafe.Platform (file:/home/georgios/spark-3.0.1-bin-
hadoop2.7/jars/spark-unsafe-2.12-3.0.1.jar) to constructor java.nio.DirectByteBuffer(long,int)
WARNING: Please consider reporting this to the maintainers of org.apache.spark.unsafe.Platform
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
Welcome to

 //_/_/_/
 /_/_/_/_/_/ version 3.0.1
 /_/_/_/_/_/

Using Scala version 2.12.10, OpenJDK 64-Bit Server VM, 11.0.8
Branch HEAD
Compiled by user ubuntu on 2020-08-28T07:36:48Z
Revision 2b147c4cd50da32fe2b4167f978142102a0510d
Url https://gitbox.apache.org/repos/asf/spark.git
georgios@georgios-HP-Pavilion-Laptop-15-cs1xxx:~/spark-3.0.1-bin-hadoop2.7/bin$
```

7082CEM CW 2020 - 2021

## Dataset

The dataset that will be explored in the coursework has been chosen from Kaggle, an online community of Data Scientists that allows users to find and publish data sets, explore and build models and enter competitions to solve data science challenges. The CSV file consists of 22 columns and the dataset is a collection from WHO and United Nations website that shows different factors that influence Life Expectancy. The data collected is from 193 different countries and concern the years from 2000 to 2015. Many different factors that influence Life Expectancy have been considered and they will be explained below.

- **Status:** Whether a country is Developed or Developing. A developing country has not yet achieved a significant degree of industrialization relative to their populations and has a low standard of living while a developed country has a sophisticated economy and an average income per resident (*bdc, n.d.*)
- **Adult Mortality:** What is illustrated is the probability of dying between 15 and 60 years per 1000 population for both sexes.
- **Infant Deaths:** The number of infant deaths per 1000 population (infant is usually applied for children under 2 years old)
- **Alcohol:** The consumption per person, over the age of 15 measured in liters
- **Percentage Expenditure:** Money each country spends on health per citizen as a percentage of GDP.
- **Hepatitis B:** Hepatitis B immunization coverage as a percentage among infants.
- **Measles:** Number of reported cases per 1000 population
- **BMI:** The average of Body Mass Index of the whole population ( $BMI = \frac{weight(kg)}{height^2(meters)}$ )
- **Under-five deaths:** The number of a child dying before reaching 5 years of age
- **Polio:** Polio immunization coverage as a percentage among infants
- **Total expenditure:** The percentage of government expenditure on health-related to total government expenditure
- **Diphtheria:** Diphtheria immunization coverage as a percentage among infants
- **HIV/AIDS:** Deaths per 1000 live births for years 0-4
- **GDP:** Gross Domestic Product in USD
- **Thinness 1-19 years:** Prevalence of thinness in percentage among 10-19-year-old
- **Thinness 5-9 years:** Prevalence of thinness in percentage among 5-9-year-old
- **Income composition of resources:** Human Development Index in terms of Income composition of resources. rates from 0-1
- **Schooling:** Years of school attendance

# Data Exploratory

Exploring the dataset is a very important step for our analysis because through it we can have an overall idea of what we are having to deal with. The reason behind the exploration is to try and find some useful insights that they will be proven to be helpful in the cleaning process, and to make the dataset much easier to handle. Therefore, we will also use some of our obtained info later in the cleaning process of the data so as to check that everything has been done correctly and to be sure that our final dataset is ready for the Machine Learning techniques to be applied.

## 1. Loading the CSV file

The first step is to import our CSV to the Jupyter Notebook environment to try and explore the data before the cleaning.

We do that by creating a SparkSession and a DataFrame with our CSV. InferSchema will make sure that the types of our variables from the CSV will be kept as they are, but later we will have to make a change to have a better look at some values. We can check that we have successfully created a SparkSession in Figure.9.

```
In [1]: # start SparkSession

from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("Data Exploratory").getOrCreate()

In [2]: # create a DataFrame

df = spark.read.csv(path="Life_Expectancy_Data.csv",
 header=True,
 inferSchema=True)
```

Figure 8. Creating SparkSession and Dataframe

```
In [3]: spark
Out[3]: SparkSession - in-memory
SparkContext

Spark UI
Version
v3.0.1
Master
local[*]
AppName
Big Data Coursework
```

Figure 9. SparkSession

## 2. Different insights from our dataset

We are getting as many details as we can from our dataset so as to use all of this information in later steps. In other words, we will be looking for general insights and imperfections in our data frame

- I. We can easily see the top rows of our CSV file to be sure that the importing was successful and to check how our data looks.

```
[3]: # see the first 5 inputs of my dataset, just for checking
df.show(5)
```

| Country     | Year | Status     | Life expectancy | Adult Mortality | infant deaths | Alcohol | percentage expenditure | Hepatitis B | Measles | BMI  | under-five deaths | Polio | Total expenditure | Diphtheria | HIV/AIDS | GDP        | Population  | thinness 1-19 years | thinness 5-9 years | Income composition of resources | Schooling |
|-------------|------|------------|-----------------|-----------------|---------------|---------|------------------------|-------------|---------|------|-------------------|-------|-------------------|------------|----------|------------|-------------|---------------------|--------------------|---------------------------------|-----------|
| Afghanistan | 2015 | Developing | 65.0            | 263             | 62            | 0.01    | 71.27962362            | 65          | 1154    | 19.1 | 83                | 6     | 8.16              | 65         | 0.1      | 584.25921  | 3.3736494E7 | 17.2                | 17.3               | 0.479                           | 10.1      |
| Afghanistan | 2014 | Developing | 59.9            | 271             | 64            | 0.01    | 73.52358168            | 62          | 492     | 18.6 | 86                | 58    | 8.18              | 62         | 0.1      | 612.696514 | 327582.0    | 17.5                | 17.5               | 0.476                           | 10.0      |
| Afghanistan | 2013 | Developing | 59.9            | 268             | 66            | 0.01    | 73.21924272            | 64          | 430     | 18.1 | 89                | 62    | 8.13              | 64         | 0.1      | 631.744976 | 3.1731688E7 | 17.7                | 17.7               | 0.47                            | 9.9       |
| Afghanistan | 2012 | Developing | 59.5            | 272             | 69            | 0.01    | 78.1842153             | 67          | 2787    | 17.6 | 93                | 67    | 8.52              | 67         | 0.1      | 669.959    | 3696958.0   | 17.9                | 18.0               | 0.463                           | 9.8       |
| Afghanistan | 2011 | Developing | 59.2            | 275             | 71            | 0.01    | 7.097108703            | 68          | 3013    | 17.2 | 97                | 68    | 7.87              | 68         | 0.1      | 63.537231  | 2978599.0   | 18.2                | 18.2               | 0.454                           | 9.5       |

only showing top 5 rows

Figure 10. Checking the dataset



- II. By printing the schema we can check the types of our variables. It seems that all of them have the type that they are supposed to have. For example, 'Country' and 'Status' are strings because they are alphabetical values, 'Life Expectancy' is double because it contains decimals and 'Year' is an integer. Also, we can see that our values could be null and for that reason, we will check and deal with them in the cleaning part.

```
[4]: # print the schema of the Dataframe so as to check the types
df.printSchema()

root
|-- Country: string (nullable = true)
|-- Year: integer (nullable = true)
|-- Status: string (nullable = true)
|-- Life expectancy : double (nullable = true)
|-- Adult Mortality: integer (nullable = true)
|-- infant deaths: integer (nullable = true)
|-- Alcohol: double (nullable = true)
|-- percentage expenditure: double (nullable = true)
|-- Hepatitis B: integer (nullable = true)
|-- Measles : integer (nullable = true)
|-- BMI : double (nullable = true)
|-- under-five deaths : integer (nullable = true)
|-- Polio: integer (nullable = true)
|-- Total expenditure: double (nullable = true)
|-- Diphtheria : integer (nullable = true)
|-- HIV/AIDS: double (nullable = true)
|-- GDP: double (nullable = true)
|-- Population: double (nullable = true)
|-- thinness 1-19 years: double (nullable = true)
|-- thinness 5-9 years: double (nullable = true)
|-- Income composition of resources: double (nullable = true)
|-- Schooling: double (nullable = true)
```

Figure 11. Checking the types of variables

- III. We can easily check the number of rows and the number of columns by using the command from Figure 12. As we can see our dataset consists of 2938 rows and 22 columns. Some of the columns and the rows will not be used in the analysis, so the numbers from Figure 12 will be used later to check our dataset after the cleaning.

```
[5]: # check the number of rows and the number of columns of my dataset
print(df.count(), len(df.columns))

2938 22
```

Figure 12. Checking the number of rows and columns

# Data Cleaning

Cleaning our data will help us because we will get rid of some variables that are not ideal for further analysis, we will spot and erase some missing values and in general, we will keep only the data that will provide us with some useful information about our coursework.

## ▪ Renaming the columns (Shannon, 2018)

Our initial action will be to change the names of the columns mostly for convenience but also to be sure that there is no column with hidden areas and type errors.

To do that we first create a list that contains the names that we want to replace (Figure.13). and then we replace our Dataframe with the one that we have swapped the name of our columns. Finally, we check that our dataset consists of the new headers (Figure.14).

```
[6]: # create a list of names for renaming the columns

newcols = ['Country', 'Year', 'Status', 'Life_Expectancy', 'Adult_Mortality',
 'Infant_Deaths', 'Alcohol', 'Percentage_Expenditure', 'Hepatitis_B',
 'Measles', 'BMI', '<5 years deaths', 'Polio', 'Total_Expenditure',
 'Diphtheria', 'HIV', 'GDP', 'Population',
 '1-19 Thinness', '5-9 Thinness', 'Income_Resources', 'Schooling']
```

Figure 13. Creating a list with the new column names

```
In [7]: # replace the old column names

for i,j in zip(df.columns,newcols):
 df=df.withColumnRenamed(i,j)

In [8]: # check the new name columns

df.show(5)
```

|      | Country     | Year | Status     | Life_Expectancy | Adult_Mortality | Infant_Deaths | Alcohol | Percentage_Expenditure | Hepatitis_B | Measles | BMI  | <5 years deaths | Polio | Total_Expenditure | Diphtheria | HIV | GDP        | Population  | 1-19 Thinness | 5-9 Thinness | Income_Resources | Schooling |
|------|-------------|------|------------|-----------------|-----------------|---------------|---------|------------------------|-------------|---------|------|-----------------|-------|-------------------|------------|-----|------------|-------------|---------------|--------------|------------------|-----------|
| 65   | Afghanistan | 2015 | Developing | 65.0            | 263             | 62            | 0.01    | 71.27962362            | 17.2        | 1154    | 19.1 | 83              | 6     | 8.16              | 65         | 0.1 | 584.25921  | 3.3736494E7 |               |              |                  |           |
| 17.3 |             |      |            | 0.479           | 10.1            |               |         |                        |             |         |      |                 |       |                   |            |     |            |             |               |              |                  |           |
| 62   | Afghanistan | 2014 | Developing | 59.9            | 271             | 64            | 0.01    | 73.52358168            | 17.5        | 492     | 18.6 | 86              | 58    | 8.18              | 62         | 0.1 | 612.696514 | 327582.0    |               |              |                  |           |
| 17.5 |             |      |            | 0.476           | 10.0            |               |         |                        |             |         |      |                 |       |                   |            |     |            |             |               |              |                  |           |
| 64   | Afghanistan | 2013 | Developing | 59.9            | 268             | 66            | 0.01    | 73.21924272            | 17.7        | 430     | 18.1 | 89              | 62    | 8.13              | 64         | 0.1 | 631.744976 | 3.1731688E7 |               |              |                  |           |
| 17.7 |             |      |            | 0.47            | 9.9             |               |         |                        |             |         |      |                 |       |                   |            |     |            |             |               |              |                  |           |
| 67   | Afghanistan | 2012 | Developing | 59.5            | 272             | 69            | 0.01    | 78.1842153             | 17.9        | 2787    | 17.6 | 93              | 67    | 8.52              | 67         | 0.1 | 669.959    | 3696958.0   |               |              |                  |           |
| 18.0 |             |      |            | 0.463           | 9.8             |               |         |                        |             |         |      |                 |       |                   |            |     |            |             |               |              |                  |           |
| 68   | Afghanistan | 2011 | Developing | 59.2            | 275             | 71            | 0.01    | 7.097108703            | 18.2        | 3013    | 17.2 | 97              | 68    | 7.87              | 68         | 0.1 | 63.537231  | 2978599.0   |               |              |                  |           |
| 18.2 |             |      |            | 0.454           | 9.5             |               |         |                        |             |         |      |                 |       |                   |            |     |            |             |               |              |                  |           |

only showing top 5 rows

Figure 14. Replacing the old column names

## ▪ Dealing with NaN and Null values

NaN and Null values can cause many problems in the analysis because they may lead to incorrect results or even failures. NaN stands for Not-a-Number and is a non-numeric variable that cannot be defined because a specific action cannot provide a valid result. A null value means that this particular spot is empty, and no data has been documented, is not 0 but is an empty space.

To handle these kinds of values originally we are checking our whole dataset for NaN and Null variables and we keep a record of the columns and the numbers of them each column has. (Figure.15 & Table.1)

Some of the columns were excluded from further analysis based on the number of the null values and some of them were removed based on the personal preference for analysing some particular aspects of the dataset (Figure.16). (\*)

```
n [9]: # check my dataset for NaN variables

from pyspark.sql.functions import isnan, when, count, col

df.select(
 [count(when(isnan(i),i)).alias(i) for i in df.columns]
).show()
```

| Country | Year | Status | Life_Expectancy | Adult_Mortality | Infant_Deaths | Alcohol | Percentage_Expenditure | Hepatitis_B | Meas | les | BMI | <5 years deaths | Polio | Total_Expenditure | Diphtheria | HIV | GDP | Population | 1-19 Thinness | 5-9 Thinness | Income_Reso | urces | Schooling |
|---------|------|--------|-----------------|-----------------|---------------|---------|------------------------|-------------|------|-----|-----|-----------------|-------|-------------------|------------|-----|-----|------------|---------------|--------------|-------------|-------|-----------|
| 0       | 0    | 0      | 0               | 0               | 0             | 0       | 0                      | 0           | 0    | 0   | 0   | 0               | 0     | 0                 | 0          | 0   | 0   | 0          | 0             | 0            | 0           | 0     | 0         |

```
n [10]: # check my dataset for Null variables

df.select(
 [count(when(isnan(i) | col(i).isNull(),i)).alias(i) for i in df.columns]
).show()
```

| Country | Year | Status | Life_Expectancy | Adult_Mortality | Infant_Deaths | Alcohol | Percentage_Expenditure | Hepatitis_B | Meas | les | BMI | <5 years deaths | Polio | Total_Expenditure | Diphtheria | HIV | GDP | Population | 1-19 Thinness | 5-9 Thinness | Income_Reso | urces | Schooling |
|---------|------|--------|-----------------|-----------------|---------------|---------|------------------------|-------------|------|-----|-----|-----------------|-------|-------------------|------------|-----|-----|------------|---------------|--------------|-------------|-------|-----------|
| 0       | 34   | 0      | 0               | 0               | 19            | 10      | 226                    | 10          | 19   | 0   | 448 | 194             | 652   | 34                | 0          | 34  | 553 | 167        | 163           |              |             |       |           |

Figure 15. Checking for NaN and Null variables

After the searching some Null values were found for the following fields:

|                   |     |
|-------------------|-----|
| Life_Expectancy   | 10  |
| Adult_Mortality   | 10  |
| Alcohol           | 194 |
| Hepatitis_B       | 553 |
| BMI               | 34  |
| Polio             | 19  |
| Total_Expenditure | 226 |
| Diphtheria        | 19  |
| GDP               | 448 |
| Population        | 652 |
| 1-19 Thinness     | 34  |
| 5-9 Thinness      | 34  |
| Income-Resources  | 167 |
| Schooling         | 163 |

Table1. Number of Null Values in columns (columns with zero Null values are ignored)

```
!]: # Remove the columns that I will not use in my Analysis

df = df.drop ("Alcohol", "Percentage_Expenditure", "Hepatitis_B",
 "BMI", "<5 years deaths", "Polio",
 "Total_Expenditure", "Diphtheria",
 "GDP", "1-19 Thinness", "5-9 Thinness",
 "Income_Resources", "Schooling")
```

Figure 16. Erasing the column that they will not be used in further analysis

So now we have our new data frame which is illustrated in Figure.17 and seems almost ready to be handled but still needs some further cleaning.

```
[3]: # check the new Dataset

df.show()
```

| Country     | Year | Status     | Life_Expectancy | Adult_Mortality | Infant_Deaths | Measles | Diphtheria | HIV | Population  |
|-------------|------|------------|-----------------|-----------------|---------------|---------|------------|-----|-------------|
| Afghanistan | 2015 | Developing | 65.0            | 263             | 62            | 1154    | 65         | 0.1 | 3.3736494E7 |
| Afghanistan | 2014 | Developing | 59.9            | 271             | 64            | 492     | 62         | 0.1 | 327582.0    |
| Afghanistan | 2013 | Developing | 59.9            | 268             | 66            | 430     | 64         | 0.1 | 3.1731688E7 |
| Afghanistan | 2012 | Developing | 59.5            | 272             | 69            | 2787    | 67         | 0.1 | 3696958.0   |
| Afghanistan | 2011 | Developing | 59.2            | 275             | 71            | 3013    | 68         | 0.1 | 2978599.0   |
| Afghanistan | 2010 | Developing | 58.8            | 279             | 74            | 1989    | 66         | 0.1 | 2883167.0   |
| Afghanistan | 2009 | Developing | 58.6            | 281             | 77            | 2861    | 63         | 0.1 | 284331.0    |
| Afghanistan | 2008 | Developing | 58.1            | 287             | 80            | 1599    | 64         | 0.1 | 2729431.0   |
| Afghanistan | 2007 | Developing | 57.5            | 295             | 82            | 1141    | 63         | 0.1 | 2.6616792E7 |
| Afghanistan | 2006 | Developing | 57.3            | 295             | 84            | 1990    | 58         | 0.1 | 2589345.0   |
| Afghanistan | 2005 | Developing | 57.3            | 291             | 85            | 1296    | 58         | 0.1 | 257798.0    |
| Afghanistan | 2004 | Developing | 57.0            | 293             | 87            | 466     | 5          | 0.1 | 2.4118979E7 |
| Afghanistan | 2003 | Developing | 56.7            | 295             | 87            | 798     | 41         | 0.1 | 2364851.0   |
| Afghanistan | 2002 | Developing | 56.2            | 3               | 88            | 2486    | 36         | 0.1 | 2.1979923E7 |
| Afghanistan | 2001 | Developing | 55.3            | 316             | 88            | 8762    | 33         | 0.1 | 2966463.0   |
| Afghanistan | 2000 | Developing | 54.8            | 321             | 88            | 6532    | 24         | 0.1 | 293756.0    |
| Albania     | 2015 | Developing | 77.8            | 74              | 0             | 0       | 99         | 0.1 | 28873.0     |
| Albania     | 2014 | Developing | 77.5            | 8               | 0             | 0       | 98         | 0.1 | 288914.0    |
| Albania     | 2013 | Developing | 77.2            | 84              | 0             | 0       | 99         | 0.1 | 289592.0    |
| Albania     | 2012 | Developing | 76.9            | 86              | 0             | 9       | 99         | 0.1 | 2941.0      |

Figure 17. Checking the new dataset after dropping some columns

### ▪ Changing data type

It can easily be seen from Figure.17 that something should be done for the 'Population' column because based on the double type we get this Scientific Notation which may be a future problem. So we are changing the type from double to decimal (Figure.18). (*Mariusz, 2016*)

```
] : # change the scientific notation in 'Population' column

from pyspark.sql.types import StructType
from pyspark.sql.types import *

df = df.withColumn('Population', df.Population.cast(DecimalType(18, 2)))
```

Figure 18. Changing the type of 'Population'

In Figure.19 it is clear that the type changed from 'double' to 'decimal' and for that reason, we can see the whole numbers.

```
15]: # check that the change in 'Population' applied

df.show(5)
df.printSchema()
```

| Country     | Year | Status     | Life_Expectancy | Adult_Mortality | Infant_Deaths | Measles | Diphtheria | HIV | Population  |
|-------------|------|------------|-----------------|-----------------|---------------|---------|------------|-----|-------------|
| Afghanistan | 2015 | Developing | 65.0            | 263             | 62            | 1154    | 65         | 0.1 | 33736494.00 |
| Afghanistan | 2014 | Developing | 59.9            | 271             | 64            | 492     | 62         | 0.1 | 327582.00   |
| Afghanistan | 2013 | Developing | 59.9            | 268             | 66            | 430     | 64         | 0.1 | 31731688.00 |
| Afghanistan | 2012 | Developing | 59.5            | 272             | 69            | 2787    | 67         | 0.1 | 3696958.00  |
| Afghanistan | 2011 | Developing | 59.2            | 275             | 71            | 3013    | 68         | 0.1 | 2978599.00  |

only showing top 5 rows

```
root
|-- Country: string (nullable = true)
|-- Year: integer (nullable = true)
|-- Status: string (nullable = true)
|-- Life_Expectancy: double (nullable = true)
|-- Adult_Mortality: integer (nullable = true)
|-- Infant_Deaths: integer (nullable = true)
|-- Measles: integer (nullable = true)
|-- Diphtheria: integer (nullable = true)
|-- HIV: double (nullable = true)
|-- Population: decimal(18,2) (nullable = true)
```

Figure 19. Checking that the type of 'Population' changed

Now it's time to check again our new dataset for any Null values. It is clear from Figure.20 that there are still some empty spots. We will continue with some further cleaning and see if we can get rid out of some of them.

```
17]: # check again these particular columns for Null values

df.select(
 [count(when(isnan(i) | col(i).isNull(),i)).alias(i) for i in df.columns]
).show()
```

| Country | Year | Status | Life_Expectancy | Adult_Mortality | Infant_Deaths | Measles | Diphtheria | HIV | Population |
|---------|------|--------|-----------------|-----------------|---------------|---------|------------|-----|------------|
|         | 0    | 0      | 0               | 10              | 10            | 0       | 0          | 19  | 0          |
|         |      |        |                 |                 |               |         |            |     | 652        |

Figure 20. Checking again for Null values in the filtered dataset

### ▪ Removing rows

After removing some columns it is very important to keep ONLY the countries that have all their values between the years 2000-2015 and exclude those who do not. Therefore, one way to figure out which countries should be removed is to count how many times the name of a country is mentioned in our dataset.

If a country is mentioned 16 times it means that we have some data for the years between 2000-2015, but if it is mentioned fewer times than 16 this means that we miss some entries. We can easily check that by firstly creating the Dataframe 'df\_count' which will count the times a country is mentioned (Figure.21). (**Rana, 2018**)

```
17]: # count how many times a country is mentioned (16 times means that we have values for all years between 2000-2015)

import pyspark.sql.functions as func

df_count = df.groupBy('Country').count().select(func.col("Country").alias("unfullfilled_countries"),
 func.col("count").alias("Counter"))

18]: df_count.show()
```

| unfullfilled_countries | Counter |
|------------------------|---------|
| Côte d'Ivoire          | 16      |
| Chad                   | 16      |
| Micronesia (Feder...   | 16      |
| Paraguay               | 16      |
| Yemen                  | 16      |
| Senegal                | 16      |
| Cabo Verde             | 16      |
| Sweden                 | 16      |
| Kiribati               | 16      |
| Republic of Korea      | 16      |
| Guyana                 | 16      |
| Eritrea                | 16      |
| Philippines            | 16      |
| Djibouti               | 16      |
| Tonga                  | 16      |
| Malaysia               | 16      |
| Singapore              | 16      |
| Fiji                   | 16      |
| Turkey                 | 16      |
| Malawi                 | 16      |

only showing top 20 rows

Figure 21. Creating a data frame which contains the number of appearances for a country

After creating 'df\_count' we can sort it in ascending order to easily identify the country names that are mentioned less than 16 times. So, now we can easily see that countries such as San Marino and Nauru are only mentioned once so there is not enough information for them between 2000-2015.

That's why we can exclude them from our analysis. We remove these particular rows by filtering our data frame. (Figure.22)

```
19]: # sort the df_count so as to find which countries do not have all their values between 2000-2015
df_count_sorted = df_count.orderBy('Counter', ascending=True)
df_count_sorted.show()
```

```
+-----+-----+
|unfullfilled_countries|Counter|
+-----+-----+
Saint Kitts and N...	1
San Marino	1
Nauru	1
Cook Islands	1
Dominica	1
Palau	1
Tuvalu	1
Marshall Islands	1
Monaco	1
Niue	1
Paraguay	16
Yemen	16
Cabo Verde	16
Cambodia	16
Senegal	16
Kiribati	16
Malaysia	16
Singapore	16
Guyana	16
Turkey	16
+-----+-----+
only showing top 20 rows
```

```
20]: # remove the 10 countries with counter = 1
from pyspark.sql import functions as F
df = df.filter(~F.col('Country').isin(['Monaco', 'Dominica', 'Nauru',
 'Cook Islands', 'Palau', 'San Marino',
 'Tuvalu', 'Marshall Islands',
 'Saint Kitts and Nevis', 'Niue']))
```

Figure 22. Sorting the df\_count data frame and erasing the countries with not enough info

We can check that the rows have been successfully removed by just comparing the number of current rows with the ones from the beginning (Figure12).

In Figure.23 we can confirm that 10 rows have been dropped successfully.

```
21]: # check that the rows have been removed (10 rows have been removed)
print(df.count(), len(df.columns))

2928 10
```

Figure 23. Checking the new number of rows for the filtered dataset

- **Replacing null values with 0**

As a final step before the analysis, we will replace the Null values with 0 and we will proceed to apply Machine Learning techniques for our clean dataset (Figure.24).

```
[22]: # fill the null values with 0 so as to continue my analysis
```

```
df = df.na.fill(0)
```

```
[23]: # check again for null values
```

```
df.select(
 [count(when(isnan(i) | col(i).isNull(),i)).alias(i) for i in df.columns]
).show()
```

| Country | Year | Status | Life_Expectancy | Adult_Mortality | Infant_Deaths | Measles | Diphtheria | HIV | Population |
|---------|------|--------|-----------------|-----------------|---------------|---------|------------|-----|------------|
| 0       | 0    | 0      | 0               | 0               | 0             | 0       | 0          | 0   | 0          |

Figure 24. Replacing the Null values

- **Finding outliers** (Packt Video, 2018)

An outlier is an observation that seems to be different from the others. This might have happened because of a mistake or because of an error in our calculations. Either way, they are not ideal for our analysis because they can mislead to wrong results.

For our coursework, we are using the Interquartile Range rule or IQR method to identify our outliers (PurpleMath, n.d.).

The IQR is defined as the subtraction between 1<sup>st</sup> and 3<sup>rd</sup> quartiles (Q1 and Q3). Q1 is the median of the smallest values of our variables and Q3 is the median of the largest values of our variables. In that way, IQR can help us identify the values that are outside the boundaries of our inputs. We set our boundaries to be any number above  $IQR + 1.5$  or below  $IQR - 1.5$  and if so is flagged as an outlier. (Figure.25)

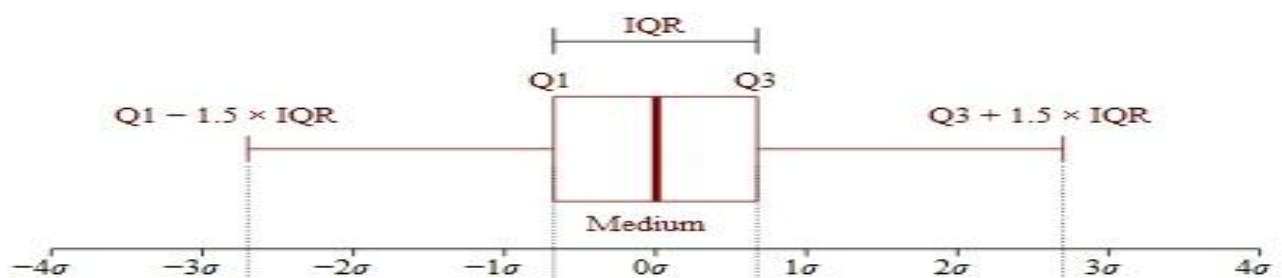


Figure 25. IQR (Jhguch, 2011)



Foremost, we are creating a list with the columns we want to check if they have outliers and then we set the bounds and the IQR (Figure. 26).

```
.]: # preparing the columns that we will check for outliers

outlcol = ['Life_Expectancy', 'Adult_Mortality', 'Infant_Deaths',
 'Measles', 'Diphtheria', 'HIV', 'Population']
bounds = {}

.]: # making the boundries for applying the IQR method

for col in outlcol:
 quantiles = df.approxQuantile(col, [0.25, 0.75], 0.05)
 IQR = quantiles[1] - quantiles[0]
 bounds[col] = [quantiles[0] - 1.5 * IQR, quantiles[1] + 1.5 * IQR]
```

Figure 26. Calculating the bound values for the columns will check for outliers

We can check the boundaries that have been created for each column and then we are flagging the outliers by setting each value which does not belong in there to be marked as an outlier (Figure. 27).

```
6]: # these are the boundries for a value being an outlier

bounds

6]: {'Life_Expectancy': [63.099999999999994, 54.500000000000003, 91.35000000000001],
 'Adult_Mortality': [77.0, 695.0, 425.5],
 'Infant_Deaths': [-1.0, 85.0, 42.5],
 'Measles': [-1.0, 1205.0, 602.5],
 'Diphtheria': [78.0, 85.0, 121.5],
 'HIV': [-0.9, 2.0, 1.1],
 'Population': [15562.0, 17796840.0, 8913983.0]}

9]: # we can now find the outliers

Outliers = df.select(*['Country'] + [(df[c] < bounds[c][0]) |
 (df[c] > bounds[c][1]).alias(c + 'outl') for c in outlcol])
```

Figure 27. Checking the bounds and finding the outliers

Every value in our data frame that is illustrated as 'true' is an outlier (Figure.28).

```
checking the outliers
Outliers.show(5)
```

| Country     | Life_Expectancyoutl | Adult_Mortalityoutl | Infant_Deathsooutl | Measlesoutl | Diphtheriaoutl | HIVoutl | Populatiooutl |
|-------------|---------------------|---------------------|--------------------|-------------|----------------|---------|---------------|
| Afghanistan | true                | false               | false              | false       | true           | false   |               |
| Afghanistan | true                | false               | false              | false       | true           | false   |               |
| Afghanistan | true                | false               | false              | false       | true           | false   |               |
| Afghanistan | true                | false               | false              | true        | true           | false   |               |
| Afghanistan | true                | false               | false              | true        | true           | false   |               |

only showing top 5 rows

Figure 28. Checking the outliers in our dataset

By counting the number of rows with outliers we can realize that every row of our data frame consists of at least 1 outlier. In that case, we will continue our analysis without removing any of them and we will be knowing that the results will be based on some not so reliable values. \*

```
: # realizing that the whole dataset contains at least one outlier in each row
print(Outliers.count(), len(Outliers.columns))

2928 8
```

Figure 29. Checking the number of rows with outliers

# Machine Learning

Machine Learning gives the computer the ability to learn without being explicitly programmed. We train our machine so as to make some decisions based on some given features giving us the ability not to worry about how to get the results. We just feed the machine with some inputs and let it figure out the best way to handle them and reach the results. There are two kinds of ML algorithms: *Supervised Learning* and *Unsupervised Learning*.

*Supervised learning* is mostly used for predictions. You give your machine some “train” data and let it find a way to produce the results. Subsequently, we can use this model so as to get some results for future events.

*Unsupervised learning* is mostly used for categorizing variables and the machine can handle unlabelled data since it tries to distinguish part of the dataset into groups.

In this coursework, **Linear Regression** is being used, which is a Supervised Learning method. Linear Regression is ideal for measuring the relationship between variables. The idea behind how Linear Regression works can be explained with the help of Figure.30. Axis Y represents the Dependent variable that we want to predict, for us is ‘Life\_Expectancy’ and X represents the Independent variables which are the points coming from the Polynomial:  $Y = A_0 + A_1x_1 + A_2x_2 + \dots + A_nx_n$ , where  $A_i$  is the features of each column in each row. So Linear Regression tries to find the best possible line that fits our model based on the train data we give to our machine.

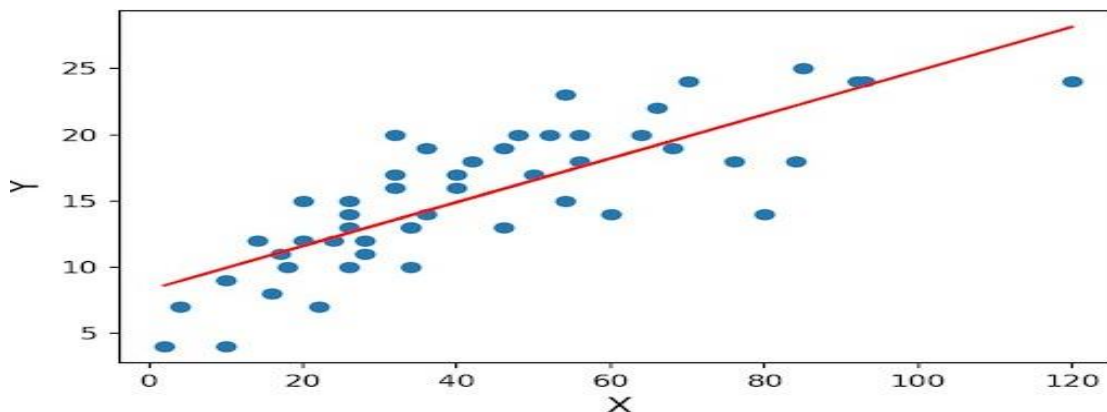


Figure 30. Linear Regression Model

In our case, we are giving our machine some trained data which contain the ‘Life\_Expectancy’ and the features of the rest of the columns and based on them our computer will build a model to make predictions for ‘Life\_Expectancy’. Afterwards, we can use train data and see if the predictions for this dataset are accurate and at which level.

Firstly we will get the **correlation** matrix between variables to see how strong is the relationship between them and see to what degree each column depends on the other.

## 1. Splitting the Dataframe

For our analysis, we will split our data frame into 2 parts based on the 'Status' of the countries so as to get some insights based on whether a country is Developed or Developing. Also, this procedure will help us later in the visualization part. (Figure.31)

```
1]: # categorizing the countries based on their status so as to do a ML Analysis for each
```

```
df_developed = df.filter(df.Status == 'Developed')
df_developed = df_developed.drop('Status')
df_developing = df.filter(df.Status == 'Developing')
df_developing = df_developing.drop('Status')
```

```
2]: df_developed.show(2)
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
| Country|Year|Life_Expectancy|Adult_Mortality|Infant_Deaths|Measles|Diphtheria|HIV| Population|
+-----+-----+-----+-----+-----+-----+-----+-----+
|Australia|2015| 82.8| 59| 1| 74| 93|0.1|23789338.00|
|Australia|2014| 82.7| 6| 1| 340| 92|0.1| 2346694.00|
+-----+-----+-----+-----+-----+-----+-----+-----+
```

only showing top 2 rows

```
3]: df_developing.show(2)
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
| Country|Year|Life_Expectancy|Adult_Mortality|Infant_Deaths|Measles|Diphtheria|HIV| Population|
+-----+-----+-----+-----+-----+-----+-----+-----+
|Afghanistan|2015| 65.0| 263| 62| 1154| 65|0.1|33736494.00|
|Afghanistan|2014| 59.9| 271| 64| 492| 62|0.1| 327582.00|
+-----+-----+-----+-----+-----+-----+-----+-----+
```

only showing top 2 rows

Figure 31. Splitting the Dataframe into Developed and Developing

## 2. Correlation

We initially check for the Developed countries (Figure.32) and then we follow the same steps for the Developing (Figure.33)

The closest the correlation value is to 1 or -1 the strongest the relationship between our values. For example in our data frame for Developed countries, we can see that some values have a positive correlation and some others have negative. The same pattern can be detected also for Developing countries. We mostly care about the correlation between 'Life\_Expectancy' with the other variables.\*

```
34]: # delete the 'string' column to get the correlation between variables
df_developed_corr = df_developed.drop('Country')

35]: from pyspark.ml.stat import Correlation
from pyspark.ml.feature import VectorAssembler

convert to vector column first

vector_col = "corr_features"
assembler = VectorAssembler(inputCols=df_developed_corr.columns, outputCol=vector_col)
df_vector = assembler.transform(df_developed_corr).select(vector_col)

get correlation matrix

matrix = Correlation.corr(df_vector, vector_col)

36]: matrix.collect()[0][0]["pearson({})".format(vector_col)].values

seems that there is a problem with HIV

36]: array([1. , 0.33272528, -0.14605756, -0.04129371, -0.08805252,
 0.13764474, nan, -0.00757979, 0.33272528, 1. ,
 -0.48548879, -0.05476379, 0.03780051, -0.01960463, nan,
 0.08083224, -0.14605756, -0.48548879, 1. , -0.04810856,
 -0.03649289, 0.01037815, nan, -0.02777954, -0.04129371,
 -0.05476379, -0.04810856, 1. , 0.05148005, 0.03458013,
 nan, 0.01373322, -0.08805252, 0.03780051, -0.03649289,
 0.05148005, 1. , 0.00881759, nan, 0.07640627,
 0.13764474, -0.01960463, 0.01037815, 0.03458013, 0.00881759,
 1. , nan, 0.05070005, nan, nan,
 nan, nan, nan, nan, 1. ,
 nan, -0.00757979, 0.08083224, -0.02777954, 0.01373322,
 0.07640627, 0.05070005, nan, 1.])
```

Figure 32. Correlation Matrix for Developed countries

```
1: #####
#2nd we work for the DEVELOPING countries#
#####

1: # delete the 'string' column to get the correlation between variables

df_developing_corr = df_developing.drop('Country')

1: # convert to vector column first

vector_col = "corr_features"
assembler = VectorAssembler(inputCols=df_developing_corr.columns, outputCol=vector_col)
df_vector = assembler.transform(df_developing_corr).select(vector_col)

get correlation matrix

matrix = Correlation.corr(df_vector, vector_col)

1: matrix.collect()[0][0]["pearson({})".format(vector_col)].values

1: array([1. , 0.18716873, -0.08147983, -0.04009469, -0.08699704,
 0.1559432 , -0.1545175 , 0.01663989, 0.18716873, 1. ,
 -0.66083581, -0.1664737 , -0.14178805, 0.45862753, -0.57059635,
 -0.01563596, -0.08147983, -0.66083581, 1. , 0.04665096,
 0.00790441, -0.23988393, 0.51556572, -0.01707059, -0.04009469,
 -0.1664737 , 0.04665096, 1. , 0.49908615, -0.15004743,
 0.00828713, 0.55448271, -0.08699704, -0.14178805, 0.00790441,
 0.49908615, 1. , -0.12221298, 0.01949687, 0.23661987,
 0.1559432 , 0.45862753, -0.23988393, -0.15004743, -0.12221298,
 1. , -0.13620609, -0.0235473 , -0.1545175 , -0.57059635,
 0.51556572, 0.00828713, 0.01949687, -0.13620609, 1. ,
 -0.02202188, 0.01663989, -0.01563596, -0.01707059, 0.55448271,
 0.23661987, -0.0235473 , -0.02202188, 1.])
```

Figure 33. Correlation Matrix for Developing countries

### 3. Linear Regression (Li, 2018)

- I. First, we start by creating the vectors which we will use for our Linear Regression Model. We want to predict the 'Life\_Expectancy' based on the features of the columns we have kept from the cleaning process. (Figure.34) \*

```

8]: # Preparing for ML in developed countries
We want to predict Life_Expectancy by using the other features

vectorAssembler = VectorAssembler(inputCols = ['Year', 'Life_Expectancy', 'Adult Mortality',
 'Measles', 'Diphtheria', 'HIV', 'Population', 'Infant_Deaths',
 'Developed_Features'], outputCol = 'Developed_Features')
vdf_developed = vectorAssembler.transform(df_developed_corr)
vdf_developed = vdf_developed.select(['Developed_Features', 'Life_Expectancy'])
vdf_developed.show(3)

```

| Developed_Features    | Life_Expectancy |
|-----------------------|-----------------|
| [2015.0,82.8,59.0...] | 82.8            |
| [2014.0,82.7,6.0,...] | 82.7            |
| [2013.0,82.5,61.0...] | 82.5            |

only showing top 3 rows

Figure 34. Creating Vector Assembler and Preparing for LR

- II. After that, we are preparing the *train* and the *test* data frames by doing a random split of our data. We will use 70% of our data so as to train our model and the rest 30% as a way to test the prediction that our machine will make. (Figure.35). \*

```

9]: # we split the developed vector into train and test so as to check the model

splits = vdf_developed.randomSplit([0.7, 0.3])
df_train = splits[0]
df_test = splits[1]

```

```

9]: # We are preparing for applying LinearRegression

from pyspark.ml.regression import LinearRegression

lr = LinearRegression(featuresCol = 'Developed_Features', labelCol='Life_Expectancy', maxIter=10, regParam=0.3, elasticNetParam=0.0)
lr_model = lr.fit(df_train)

print("Coefficients: " + str(lr_model.coefficients))
print("Intercept: " + str(lr_model.intercept))

```

Coefficients: [0.0,0.9253444349364341,0.0,0.0,0.0,0.0,0.0,0.0,0.0]

Intercept: 5.917882887147626

Figure 35. Split data frame into Train and Test

III. Before we apply Linear Regression we can check some metrics so as to see the efficiency of our model.

- *Root Mean Square Error (RMSE)* is a metric that indicates how good are the predictions for our model. Evaluates the error by comparing the actual value with the prediction of our machine based on the model so in our case the model is quite accurate since the RMSE is only 29,4% (Figure.36).
- $r^2$  is a metric that indicates how close are the data to the regression line. The higher the  $r^2$  the better the model fits your data and in our case is very high. (Figure.36).

```
41]: # We check RMSE and r^2
trainingSummary = lr_model.summary
print("RMSE: %f" % trainingSummary.rootMeanSquaredError)
print("r2: %f" % trainingSummary.r2)

RMSE: 0.294123
r2: 0.994427
```

```
42]: # Check the train dataframe
df_train.describe().show()
```

|        | summary           | Life_Expectancy |
|--------|-------------------|-----------------|
| count  |                   | 363             |
| mean   | 79.26914600550957 |                 |
| stddev | 3.945174839889555 |                 |
| min    | 69.9              |                 |
| max    | 89.0              |                 |

```
43]: # check the test dataframe
df_test.show(2)
```

|                       | Developed_Features | Life_Expectancy |
|-----------------------|--------------------|-----------------|
| [2000.0,71.6,2.0,...] |                    | 71.6            |
| [2000.0,76.0,122....] |                    | 76.0            |

only showing top 2 rows

Figure 36. Checking RMSE and  $r^2$

IV. Finally, we are ready to test our model by applying the ML analysis to our test data.

```
[4]: # we apply Linear Regression for developed countries

lr_predictions = lr_model.transform(df_test)

lr_predictions.select("prediction","Life_Expectancy","Developed_Features").show(5)
from pyspark.ml.evaluation import RegressionEvaluator

lr_evaluator = RegressionEvaluator(predictionCol="prediction", \
 labelCol="Life_Expectancy",metricName="r2")

print("R Squared (R2) on test data = %g" % lr_evaluator.evaluate(lr_predictions))

+-----+-----+-----+
| prediction|Life_Expectancy| Developed_Features|
+-----+-----+-----+
72.17254442859631	71.6	[2000.0,71.6,2.0,...
76.24405994231662	76.0	[2000.0,76.0,122....
77.16940437725306	77.0	[2000.0,77.0,175....
79.57529990808779	79.6	[2000.0,79.6,73.0...
79.66783435158143	79.7	[2000.0,79.7,74.0...
+-----+-----+-----+
only showing top 5 rows

R Squared (R2) on test data = 0.994404
```

```
[5]: # We get the RMSE

test_result = lr_model.evaluate(df_test)
print("Root Mean Squared Error (RMSE) on test data = %g" % test_result.rootMeanSquaredError)

Root Mean Squared Error (RMSE) on test data = 0.291036
```

Figure 37. Applying Linear Regression

```
] : # We check the predictions for our dataset

predictions_developed = lr_model.transform(df_test)
predictions_developed.show(5)

+-----+-----+-----+
| Developed_Features|Life_Expectancy| prediction|
+-----+-----+-----+
[2000.0,71.6,2.0,...	71.6	72.17254442859631
[2000.0,76.0,122....	76.0	76.24405994231662
[2000.0,77.0,175....	77.0	77.16940437725306
[2000.0,79.6,73.0...	79.6	79.57529990808779
[2000.0,79.7,74.0...	79.7	79.66783435158143
+-----+-----+-----+
only showing top 5 rows
```

Figure 38. Checking the prediction in our test data frame

We can see that the predictions of our model are quite accurate.



## V. Now we just follow the same steps for the Developing countries (Figures.39 - 41)

```

.): # Preparing for ML in developing countries

vectorAssembler = VectorAssembler(inputCols = ['Year', 'Life_Expectancy', 'Adult_Mortality',
 'Measles', 'Diphtheria', 'HIV', 'Population', 'Infant_Deaths',
 'Developing_Features'], outputCol = 'Developing_Features')
vdf_developing = vectorAssembler.transform(df_developing_corr)
vdf_developing = vdf_developing.select(['Developing_Features', 'Life_Expectancy'])
vdf_developing.show(3)

+-----+-----+
| Developing_Features|Life_Expectancy|
+-----+-----+
[2015.0,65.0,263....	65.0
[2014.0,59.9,271....	59.9
[2013.0,59.9,268....	59.9
+-----+-----+
only showing top 3 rows

.): splits = vdf_developing.randomSplit([0.7, 0.3])
df_train = splits[0]
df_test = splits[1]

.): lr = LinearRegression(featuresCol = 'Developing_Features', labelCol='Life_Expectancy', maxIter=10, regParam=0.3, el
lr_model = lr.fit(df_train)

print("Coefficients: " + str(lr_model.coefficients))
print("Intercept: " + str(lr_model.intercept))

Coefficients: [0.0,0.9672582828378288,0.0,0.0,0.0,0.0,0.0,0.0,0.0]
Intercept: 2.1872471169907755

```

Figure 39. Preparing for LR in Developing countries

```

.4]: trainingSummary = lr_model.summary
print("RMSE: %f" % trainingSummary.rootMeanSquaredError)
print("r2: %f" % trainingSummary.r2)

RMSE: 0.298120
r2: 0.998928

.5]: df_train.describe().show()

+-----+-----+
|summary| Life_Expectancy|
+-----+-----+
count	1663
mean	66.80306674684306
stddev	9.107939910325658
min	36.3
max	89.0
+-----+-----+

.6]: df_test.show(2)

+-----+-----+
| Developing_Features|Life_Expectancy|
+-----+-----+
|[2000.0,46.0,49.0...| 46.0|
|[2000.0,46.0,665....| 46.0|
+-----+-----+
only showing top 2 rows

```

Figure 40. Checking for MRSE and  $r^2$

```
7]: lr_predictions = lr_model.transform(df_test)
lr_predictions.select("prediction", "Life_Expectancy", "Developing_Features").show(5)

lr_evaluator = RegressionEvaluator(predictionCol="prediction", \
 labelCol="Life_Expectancy", metricName="r2")
print("R Squared (R2) on test data = %g" % lr_evaluator.evaluate(lr_predictions))

+-----+-----+-----+
| prediction|Life_Expectancy| Developing_Features|
+-----+-----+-----+
46.6811281275309	46.0	[2000.0,46.0,49.0...
46.6811281275309	46.0	[2000.0,46.0,665...
47.2614830972336	46.6	[2000.0,46.6,554...
47.74511223865251	47.1	[2000.0,47.1,45.0...
48.42219303663899	47.8	[2000.0,47.8,647...
+-----+-----+-----+
only showing top 5 rows

R Squared (R2) on test data = 0.998914

8]: test_result = lr_model.evaluate(df_test)
print("Root Mean Squared Error (RMSE) on test data = %g" % test_result.rootMeanSquaredError)

Root Mean Squared Error (RMSE) on test data = 0.287946

9]: predictions_developing = lr_model.transform(df_test)
predictions_developing.show(5)

+-----+-----+-----+
| Developing_Features|Life_Expectancy| prediction|
+-----+-----+-----+
[2000.0,46.0,49.0...	46.0	46.6811281275309
[2000.0,46.0,665...	46.0	46.6811281275309
[2000.0,46.6,554...	46.6	47.2614830972336
[2000.0,47.1,45.0...	47.1	47.74511223865251
[2000.0,47.8,647...	47.8	48.42219303663899
+-----+-----+-----+
only showing top 5 rows
```

Figure 41. Checking for prediction of our model in the test data frame

In Figure.42 we can check that for both Developed and Developing countries the model is accurate and the predictions are good.

```
]: predictions_developed.show(5)
predictions_developing.show(5)

+-----+-----+-----+
| Developed_Features|Life_Expectancy| prediction|
+-----+-----+-----+
[2000.0,71.6,2.0,...	71.6	72.17254442859631
[2000.0,76.0,122...	76.0	76.24405994231662
[2000.0,77.0,175...	77.0	77.16940437725306
[2000.0,79.6,73.0...	79.6	79.57529990808779
[2000.0,79.7,74.0...	79.7	79.66783435158143
+-----+-----+-----+
only showing top 5 rows

+-----+-----+-----+
| Developing_Features|Life_Expectancy| prediction|
+-----+-----+-----+
[2000.0,46.0,49.0...	46.0	46.6811281275309
[2000.0,46.0,665...	46.0	46.6811281275309
[2000.0,46.6,554...	46.6	47.2614830972336
[2000.0,47.1,45.0...	47.1	47.74511223865251
[2000.0,47.8,647...	47.8	48.42219303663899
+-----+-----+-----+
only showing top 5 rows
```

Figure 42. The prediction tables of our analysis

## Visualization

Once we have extracted the data frames that we created, with the aim of Tableau we can gain some interesting insights for the Life Expectancy and other variables of the countries based on their status and make easily some forecasts for the next years.

- In Figures 43 & 44 we can see the average Life Expectancy for the years 2000-2015 across the countries.

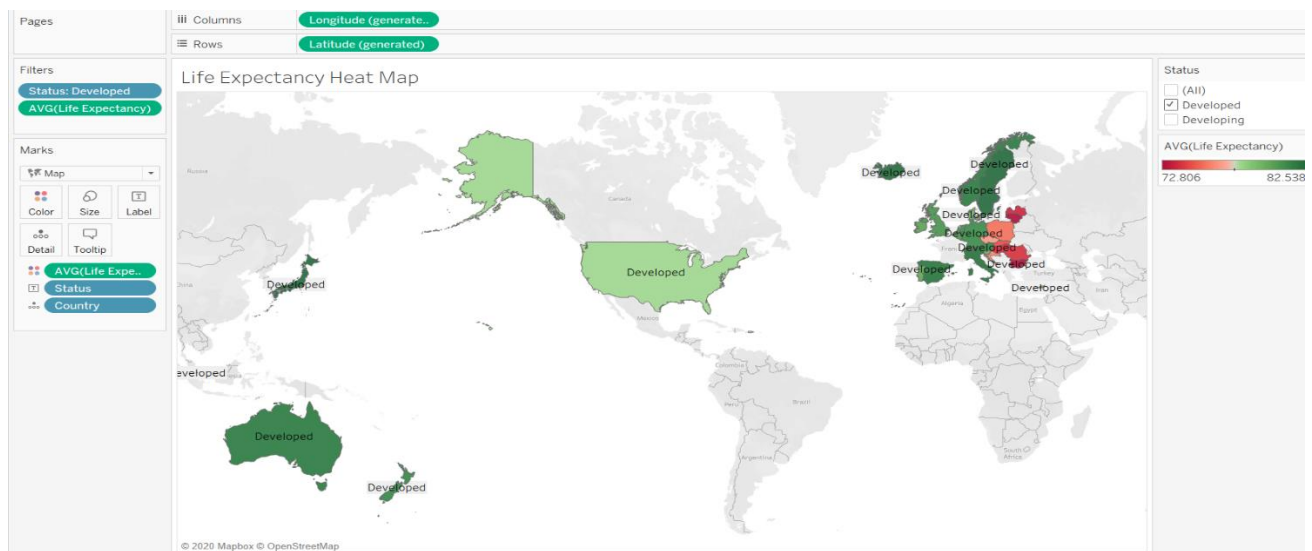


Figure 43. Life Expectancy for Developed countries (Heat Map)

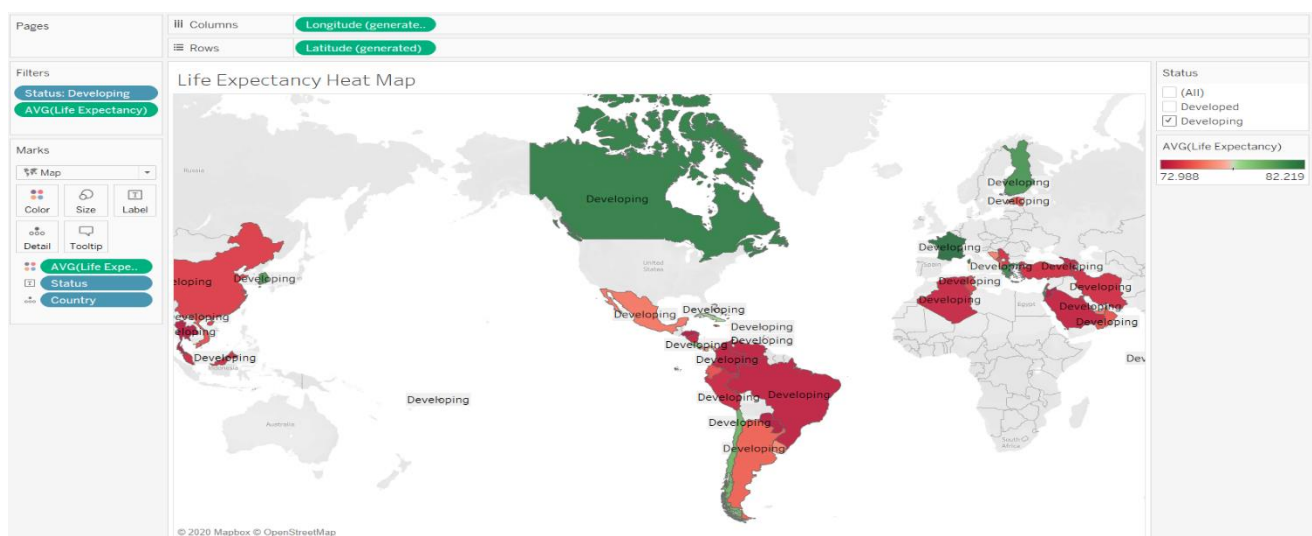


Figure 44. Life Expectancy for Developing countries (Heat Map)

- In the table below (Figure.45) we can see the number of HIV cases being counted over the years between 2000-2015 for Developing countries.



Figure 45. HIV (SUM) over the years for Developing countries (Table)

- In Figure 46 we have a bubble chart that demonstrates the SUM of Adult Mortality for all the Developed countries.  
Tableau gives you the ability to visualize in dissimilar shapes the different results based on the numbers making it much easier for being obtainable in a presentation.

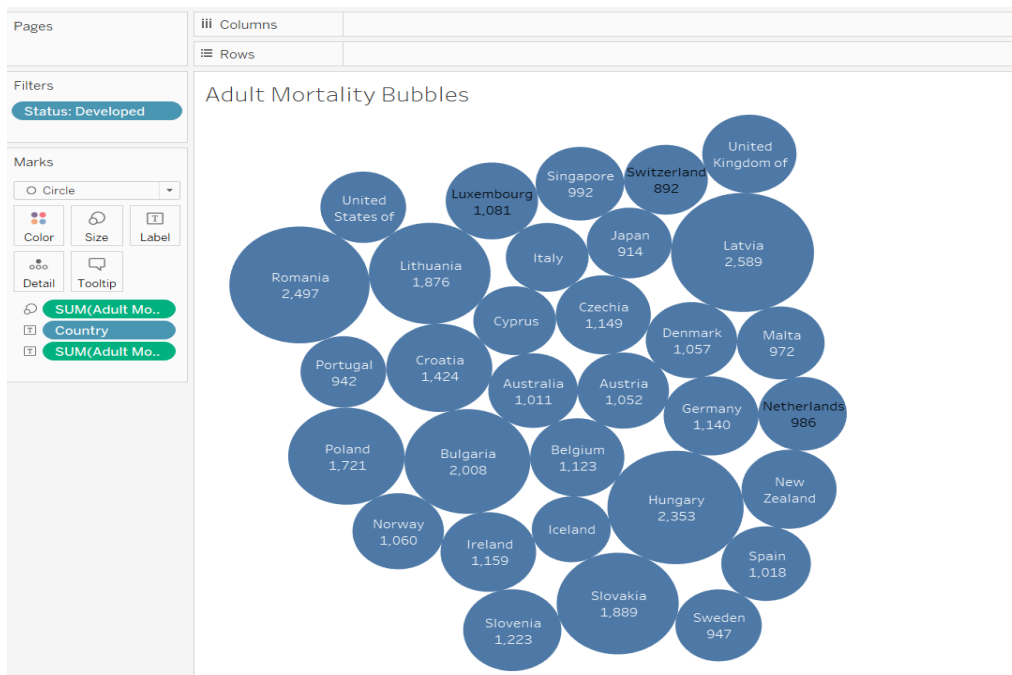


Figure 46. Adult Mortality for Developed countries (Bubble Chart)

- We can also have a mixed graph to compare different attributes. So, in Figure.47 we can compare the alteration of Adult Mortality with the equivalent of the Population over the years for the Developed countries.



Figure 47. Adult Mortality compared to Population for Developed countries (Line Chart)

- Tableau can create a forecast based on the data so we can easily visualize a prediction for the Average Life Expectancy of the countries. (Figure. 48)

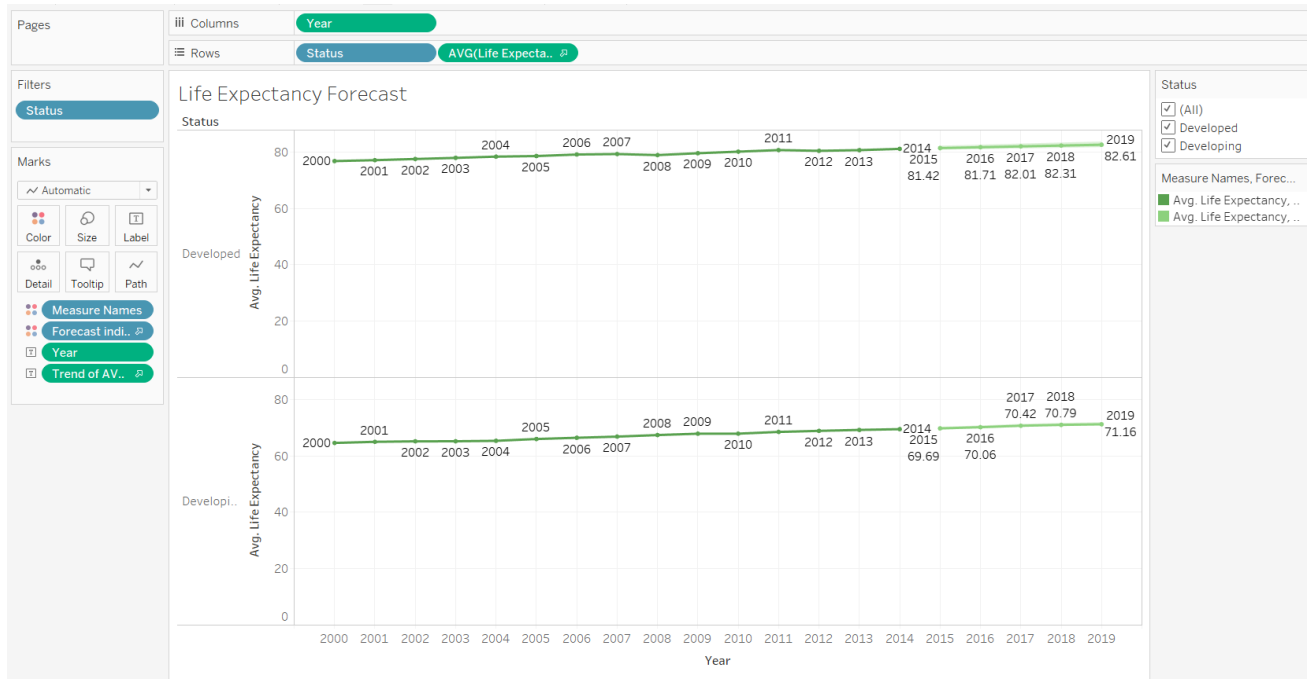


Figure 48. Life Expectancy forecast for the next 5 years

## Conclusion

- Our model seems to have high accuracy for future predictions which means that if we feed it with some new features our machine will be able to predict the 'Life Expectancy' for them also. In both data frames, the accuracy of the model is almost 70% (RMSE = ~29%) and the predictions on the test data indicate that is well fitted.
- From the Heat Map in Figure. 43 we can see that the Western Developed countries tend to have a larger 'Life Expectancy' than those to the East. Similarly, in Developing countries, those being to the North seem to have bigger 'Life Expectancy'.
- Using Tableau we can get a forecast for the Average of 'Life Expectancy' until 2019 and we can see that for both Developed and Developing countries the Average Life Expectancy is going to increase. Until the end of 2019, we have respectively a ~ 1,5% increase for the Developed countries and a ~1,1% increase for the Developing.

## Discussion – Personal Notes (\*)

- In *Data Cleaning >> part 2: "Dealing with NaN and Null values"* the removing of the columns was mostly based on the number of missing values and personal preference. Some columns could have been used as well but because of lack of experience, I decided to handle fewer data and try to focus more on the steps of the cleaning and analysis.
- In *Data Cleaning >> part 6: "Finding Outliers"* the bounds for the outliers were based on the whole dataset but it would better to be done independently for each country. That is why every row had an outlier in our analysis. I tried to find a way to apply the outlier for each name of the country, but I couldn't find a proper way to do it. With the help of Tableau I managed to find outliers for each country and year (Figure. 46) so I could extract them as a different data frame and then drop them from the original, but I preferred to use only PySpark for my data processing and analysis.

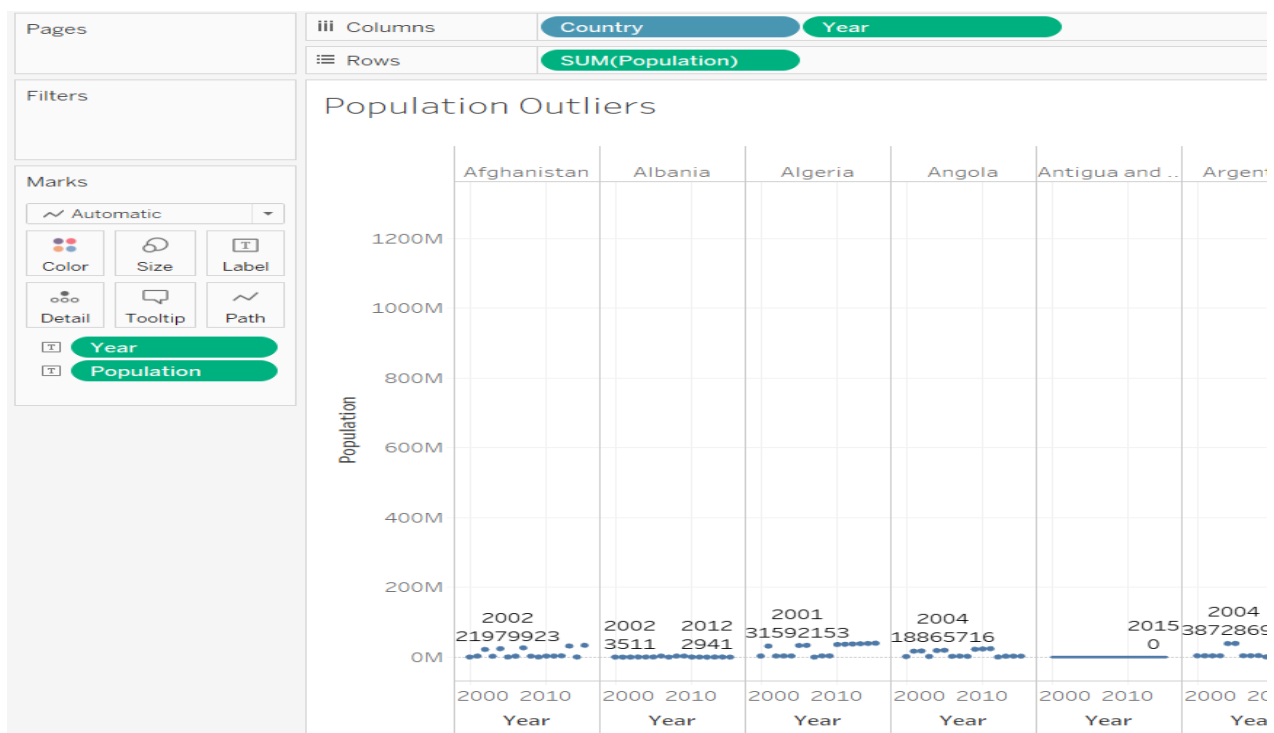


Figure 49. Example of finding outliers for with the help of Tableau

- In *Machine Learning >> Part 2: "Correlation"* it seems that correlation with the variable of HIV for 'Developed' countries returns *nan* and that indicates no independence at all with the other variables. However, in Developing countries, there is a small correlation for the same attribute. This might be because of the cleaning process and the 0 values we replaced for continuing our analysis.

- In Machine Learning >> Part 3: “Linear Regression” we couldn’t take into consideration the name of the countries because for our model we couldn’t use string values. One solution would be to flag each country with a number but I wasn’t sure that this would be the best option mostly because the numbers would not help in the building of the model because they just represent a name and have nothing to do with the features we wanted to take into consideration for the analysis.
- In *Machine Learning* >> Part 3: “Linear Regression” even though our model makes a good prediction the  $r^2$  metric is almost 1 which indicates that our model may have been overfitted, so in that case might not be ideal for future predictions.
- In Machine Learning >> Part 3 : “Linear Regression the data was split randomly into *train* and *test*. I tried to figure a way to split it in a way so that I could keep the years steady and have an overall prediction over the years in order but I couldn’t find a way that would work for the Linear Regression because if the split is not random we can’t expect our machine to learn based on the given features.



## References

1. Shannon, G., 2018. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/34077353/how-to-change-dataframe-column-names-in-pyspark>
2. Apache Software Foundation, 2020. *Apache Hadoop*. [online] Hadoop.apache.org. Available at: <https://hadoop.apache.org/>
3. Apache Spark, 2020. *Apache Spark™ - Unified Analytics Engine For Big Data*. [online] Spark.apache.org. Available at: <https://spark.apache.org/>
4. **Installing Pyspark & Jupyter** Karthikeyan Mohanra, n.d. *Apache SPARK Using Jupyter In LINUX : Installation And Setup*. [online] Medium. Available at: <https://medium.com/python-in-plain-english/apache-spark-using-jupyter-in-linux-installation-and-setup-b2cacc6c7701>
5. **Sudo 777 command** Damien, 2012. *Chmod 777: What Does It Really Mean? - Make Tech Easier*. [online] Make Tech Easier. Available at: <https://www.maketecheasier.com/file-permissions-what-does-chmod-777-means/>
6. bdc, n.d. [online] Available at: <https://www.bdc.ca/en/articles-tools/entrepreneur-toolkit/templates-business-guides/glossary/pages/developed-country.aspx>
7. Mariusz, 2016. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/40206592/how-to-turn-off-scientific-notation-in-pyspark>
8. Rana, V., 2018. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/53650430/pyspark-counting-number-of-occurrences-of-each-distinct-values>
9. PurpleMath, n.d. [ebook] PurpleMath. Available at: <https://www.purplemath.com/modules/boxwhisk3.htm>
10. Packt Video, 2018. *Pyspark For Beginners: Checking For Duplicates, Missing Observations, And Outliers | Packtpub.Com*. [video] Available at: [https://www.youtube.com/watch?v=wXx58-mDOKI&list=PLFDnsVr7cOJBr18gVCF2XByfeJr-W3BNL&index=7&ab\\_channel=PacktVideo](https://www.youtube.com/watch?v=wXx58-mDOKI&list=PLFDnsVr7cOJBr18gVCF2XByfeJr-W3BNL&index=7&ab_channel=PacktVideo)
11. **Linear Regression** Li, S., 2018. *Building A Linear Regression With Pyspark And Mllib*. [online] Medium. Available at: <https://towardsdatascience.com/building-a-linear-regression-with-pyspark-and-mllib-d065c3ba246a>
12. Jhguch, 2011. *The Interquartile Range*. [image] Available at: <https://www.statisticshowto.com/calculators/interquartile-range-calculator/>