big_data_exploration

May 21, 2022

1 Projet BIG DATA : Partie Exploration

POIRON Alex 23492 et THIL Tom 23034

[1]:
```
from pyspark.sql import SparkSession
from pyspark.sql.types import *

#Manipulation to not confuse max() function from Python and max() function from␣
 ↪Pyspark
to_exclude = ['max', 'sum']
from pyspark.sql.functions import *
for name in to_exclude:
    del globals()[name]

from pyspark.sql.functions import max as max_
from pyspark.sql.functions import sum as sum_

#Correlation imports
from pyspark.ml.stat import Correlation
from pyspark.ml.feature import VectorAssembler

#Visualization
import matplotlib.pyplot as plt
from matplotlib import cm
import numpy as np
from matplotlib import rcParams

import itertools
```

[2]:
```
spark_application_name = "Projet"
spark = (SparkSession.builder.appName(spark_application_name).getOrCreate())
```

```
22/05/21 15:50:12 WARN Utils: Your hostname, alex-ASUS resolves to a loopback
address: 127.0.1.1; using 192.168.1.44 instead (on interface enx4ce1734b82ca)
22/05/21 15:50:12 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:/opt/spark/jars/spark-unsafe_2.12-3.2.1.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
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use
setLogLevel(newLevel).
22/05/21 15:50:13 WARN NativeCodeLoader: Unable to load native-hadoop library
for your platform… using builtin-java classes where applicable
```

[3]:
```
amazon = spark.read.csv("./stocks_data/AMAZON.csv")
amazon.printSchema()
amazon.select('*').limit(10).show()
print("Number of rows =", amazon.count())
```

```
root
 |-- _c0: string (nullable = true)
 |-- _c1: string (nullable = true)
 |-- _c2: string (nullable = true)
 |-- _c3: string (nullable = true)
 |-- _c4: string (nullable = true)
 |-- _c5: string (nullable = true)
 |-- _c6: string (nullable = true)
 |-- _c7: string (nullable = true)

+----------+-----------------+-----------------+-----------------+-------------
---+-------+-----------------+------------+
|       _c0|              _c1|              _c2|              _c3|
_c4|    _c5|              _c6|         _c7|
+----------+-----------------+-----------------+-----------------+-------------
---+-------+-----------------+------------+
|      Date|             High|              Low|             Open|
Close| Volume|        Adj Close|company_name|
|2017-01-03| 758.760009765625|747.7000122070312|757.9199829101562|753.6699829101
562|3521100|753.6699829101562|      AMAZON|
|2017-01-04|759.6799926757812|754.2000122070312|758.3900146484375|757.1799926757
812|2510500|757.1799926757812|      AMAZON|
|2017-01-05|782.4000244140625|
760.260009765625|761.5499877929688|780.4500122070312|5830100|780.4500122070312|
AMAZON|
|2017-01-06|799.4400024414062|  778.47998046875|782.3599853515625|
795.989990234375|5986200| 795.989990234375|      AMAZON|
|2017-01-09|  801.77001953125|  791.77001953125|
798.0|796.9199829101562|3446100|796.9199829101562|      AMAZON|
|2017-01-10|            798.0|789.5399780273438|796.5999755859375|795.9000244140
625|2558400|795.9000244140625|      AMAZON|
|2017-01-11|            799.5| 789.510009765625|793.6599731445312|
799.02001953125|2992800|  799.02001953125|      AMAZON|
|2017-01-12|814.1300048828125|
799.5|800.3099975585938|813.6400146484375|4873900|813.6400146484375|
AMAZON|
|2017-01-13|821.6500244140625|811.4000244140625|814.3200073242188|817.1400146484
375|3791900|817.1400146484375|      AMAZON|
+----------+-----------------+-----------------+-----------------+--------------
---+-------+-----------------+------------+

Number of rows = 988
```

Il nous faut changer le schema egalement de ces dataframe pour que nous puissions mieux nous servir des donnees.

[4]:
```python
stocks_columns = [StructField("Date",TimestampType()),␣
 ↪StructField("High",FloatType()), StructField("Low",FloatType()),␣
 ↪StructField("Open",FloatType()),StructField("Close",FloatType()),␣
 ↪StructField("Volume",FloatType()), StructField("Adj Close",FloatType()),␣
 ↪StructField("company_name",StringType())]
new_schema = StructType(stocks_columns)
```

On remarque que l'on peut faire une fonction générique qui peut être appelée à chaque fois. Cela serait beucoup plus rapide si à chaque fois en plus de load le dataset, on souhaite avoir quelques informations dessus. C'est ce que nous allons faire maintenant.

[5]:
```python
def read_infos(path):
    """
        :param path : related path to the file
        :return the new DataFrame created
    """
    df_with_null = spark.read.csv(path, new_schema) #Creating the df with the␣
 ↪new schema, we have a first null row
    df = spark.createDataFrame(df_with_null.tail(df_with_null.count()-1),␣
 ↪df_with_null.schema) #With this line we delete it
    df.printSchema()
    df.show(40)
    print("Number of rows =", df.count())
    return df
```

1.0.1 Date period

Nous allons maintenant nous intéresser à la période qui s'écoule entre chaque observation de chaque DataFrame. Pour ce faire, nous allons faire une fonction qui en prenant un DataFrame en paramètre, retournera le type de période écartant chaque observation. (jour le jour, quelques jours, semaines, …)

```
[6]: def date_period(df):
         """
             :param df the DataFrame
             :return a string that correspond to the period in the DataFrame
         """
         diff = []
         name = df.first()["company_name"]
         dates = df.select('Date').rdd.flatMap(lambda x: x).collect()
         #Day case
         for i in range(0, len(dates)-1):
             diff.append(dates[i+1].day - dates[i].day)

         period = max(diff, key=diff.count)
         if period == 1 :
             return "For " + name + ", it's a day period"

         #Month and year case
         diff = []
         for i in range(0, len(dates)-1):
             diff.append(dates[i+1].month - dates[i].month)
         period = max(diff, key=diff.count)

         if period == 1 :
             return "For " + name + ", it's a month period"
         else:
             return "For " + name + ", its a year period"
```

1.0.2 Descriptive Statistics

Fonction permettant d'avoir certaines statistiques sur chaque colonne d'un dataframe (min, max, moyenne, variance)

```
[7]: def stats(df):
         """
             :param df : DataFrame
             :return Print mean, min, max, sd for each column of the DataFrame
         """
         columns = ["High", "Low", "Open", "Close", "Volume", "Adj Close"]

         for colname in columns:
             print("Stats for :", colname)
             df.agg(mean(df[colname]), min(df[colname]), max_(df[colname]),␣
      ↪stddev(df[colname])).show()
```

1.0.3 Check missing values

Fonction qui permet de verifier s'il y a des valeurs nulles dans chaque colonnes du DataFrame donnee en parametre.

[8]:
```python
def check_missing(df):
    """
        :param df : DataFrame
        :return : Show NULL values
    """
    df.select(*[
    (
        count(when((isnan(c) | col(c).isNull()), c)) if t not in "timestamp"
        else count(when(col(c).isNull(), c))
    ).alias(c)
    for c, t in df.dtypes if c in df.columns
]).show()
```

1.0.4 Correlation

Pour chaque DataFrame, nous allons étudier les corrélations entre les données. Pour ce faire, nous allons changer le type des donées en **Vector** puis appliquer dessus une fonction du package **ML** de pyspark pour avoir la matrice de corrélation. Nous crééons donc une fonction générique qui prend en paramètre un DataFrame et qui retourne sa matrice de corrélation associée.

[9]:
```python
def get_corr_matrix(df):
    """
        :param df : DataFrame
        :return the correlation matrix of the DataFrame
    """
    vect_col = "corr_data"
    df = df.drop('Date','company_name') #Only numeric values
    assembler = VectorAssembler(inputCols=df.columns, outputCol=vect_col)
    df_vector = assembler.transform(df).select(vect_col)
    print(df.columns)

    #Transform dataframe matrix in list by getting its values
    matrix = Correlation.corr(df_vector, vect_col).collect()[0][0].toArray().
 ↪tolist()
    return matrix
```

1.1 Questions on the data

- What is the average of the opening and closing prices for each stock price and for different time periods (week, month, year)

On créé une fonction qui calcule la moyenne des prix selon une période précise. Pour nous aider, on créée également une fonction auxiliaire qui permet de construire une liste correspondant à la bonne temporalité demandée.

[10]:
```
def get_list_period(df, w=False, m=False, y=False):
    """
        :param
            - df : DataFrame
            - w : Boolean True if we want a week period, False otherwise
            - m : Boolean True if we want a month period, False otherwise
            - y : Boolean True if we want a year period, False otherwise

        :return : List containing date values corresponding to the period.
            - [2017,2018,...] for years
            - [1, 2, 3, 4, ..., 12] for months
            - [1, 2, ..., 52] for weeks
    """
    list = []
    if w:
        dates = df.select(weekofyear(df['Date'])).collect()
        for i in range(len(dates)):
            if dates[i][0] not in list:
                list.append(dates[i][0])
    else:
        dates = df.select('Date').collect() #convert column to list
        for date in dates:
            if y:
                if date[0].year not in list:
                    list.append(date[0].year)
            else:
                if date[0].month not in list:
                    list.append(date[0].month)
    list.sort()
    return list
```

[11]:
```
def average_price(df, w=False, m=False, y=False):
    """
        :param
            - df : DataFrame
            - w : Boolean True if we want a week period, False otherwise
            - m : Boolean True if we want a month period, False otherwise
            - y : Boolean True if we want a year period, False otherwise

        :return : Show average for opening and closing price calculated␣
 ↪following the period choose
            - if year -> [2017, 2018, ...] -> average open and close price for␣
 ↪2017, 2018, ...
            - if month -> [1, 2, ...] -> average open and clos eprice for 1, 2,␣
 ↪3, ...
            - if week -> [1, 2, ..., 52] -> average open and clos eprice for 1,␣
 ↪2, 3, ..., 52
```

```
    """
    list_period = get_list_period(df, w, m, y)
    for i in list_period:
        #Filter the Dataframe following the period choose
        if y: df_filtered = df.filter(year(df['Date']) == i)
        elif m: df_filtered = df.filter(month(df['Date']) == i)
        else: df_filtered = df.filter(weekofyear(df['Date']) == i)

        print("Average open price and close price in", i)
        df_filtered.agg(mean(df_filtered['Open']), mean(df_filtered['Close'])).
↪show()
```

- How do the stock prices change day to day and month to month (may be you can create new columns to save those calculations)

```
[12]: def plot_evolution_stock_prices(df, d=False,m=False):
    """
        :param
            - df : DataFrame
            - w : Boolean True if we want a week period, False otherwise
            - m : Boolean True if we want a month period, False otherwise

        :return Plotting evolution for each price for each DataFrame following␣
↪a
                specific period(each month or day to day).
                For each DataFrame, we have a subplot with all prices except␣
↪Volume price and another
                subplot with Volume price. This separation is caused by the␣
↪values of Volume price
                that are higher than other prices
    """
    #Construct list of y axis that is period scale (year, week or month)
    period = []
    if d:
        dates = df.select(dayofyear(df['Date'])).collect()
        for i in range(len(dates)):
                period.append(dates[i][0])
    else:
        dates = df.select('Date').collect() #convert column to list
        for date in dates:
            if date[0].month not in period:
                period.append(date[0].month)

    if m: period.sort() #sort for months not for days

    #Lists for all prices except volume
    avg_open = []
    avg_close = []
    avg_high = []
    avg_low = []
    avg_adj_close = []
    prices = [
        (avg_open, "Open"),
        (avg_close, "Close"),
        (avg_high, "High"),
        (avg_low, "Low"),
        (avg_adj_close, "Adj CLose")
        ]

    avg_volume = []

    #Construct lists
    for i in period:
        if m: df_filtered = df.filter(month(df['Date']) == i)
        else: df_filtered = df.filter(dayofyear(df['Date']) == i)

        for (price, name_price) in prices:
            price.append(df_filtered.agg(mean(df_filtered[name_price])).
↪collect()[0][0])

        avg_volume.append(df_filtered.agg(mean(df_filtered['Volume'])).
↪collect()[0][0])


    #PLotting
    name = df.first()['company_name']
    x = np.arange(len(period))
    figure, axis = plt.subplots(1, 2, figsize=(10,7))

    #First plot for al prices except volume due to the diffence in values
    for (price, name_price) in prices:
        axis[0].plot(x, price, label=name_price)

    axis[0].set_title('Average by prices except Volume one for ' + name)
    axis[0].set_xlabel("Period of time")
    axis[0].set_ylabel('Average for all prices')
    axis[0].legend()

    #2d plot for volume prices
    axis[1].plot(x, avg_volume, label="Volume price")
    axis[1].set_title('Average for Volume price for ' + name)
    axis[1].legend()
    axis[1].set_xlabel('Period of time')
    plt.show()
```

- Based on the opening and closing price, calculate the daily return of each stock

```
[13]: def daily_return(df):
          """
              :param df : Dataframe
              :return add in a column the daily return calculated day to day
          """
          df.withColumn("evolution_price", df['Close'] - df['Open']).show()
```

- What are the stocks with the highest daily return

```
[14]: def highest_daily_return():
          """
              :return : the highest daily return
          """
          maxi = 0
          for df in LIST_DF:
              tmp = df.agg(max_(df['Close'] - df['Open'])).collect()[0][0]
              name_tmp = df.first()['company_name']
              if (tmp > maxi):
                  maxi = tmp
                  name = name_tmp

          print("Highest daily return is :",maxi ,"found in", name)
```

- Calculate the average daily return for different periods (week, month, and year)

```
[15]: def avg_daily_return(df, w=False, m=False, y=False):
          """
              :param
                  - df : DataFrame
                  - w : Boolean True if we want a week period, False otherwise
                  - m : Boolean True if we want a month period, False otherwise
                  - y : Boolean True if we want a year period, False otherwise

              :return Average daily return calculated following the perdio choose
          """
          list = []
          if w:
              dates = df_zoom.select(weekofyear(df_zoom['Date'])).collect()
              for i in range(len(dates)):
                  if dates[i][0] not in list:
                      list.append(dates[i][0])
          else:
              dates = df.select('Date').collect() #convert column to list
```

```
        for date in dates:
            if y:
                if date[0].year not in list:
                    list.append(date[0].year)
            else:
                if date[0].month not in list:
                    list.append(date[0].month)

    list.sort()

    for i in list:
        if y: df_filtered = df.filter(year(df['Date']) == i)
        elif m: df_filtered = df.filter(month(df['Date']) == i)
        else: df_filtered = df.filter(weekofyear(df['Date']) == i)
        print("Average daily return in", i)
        df_filtered.agg(mean(df_filtered['Close']- df_filtered['Open'])).show()
```

1.2 Main class

Maintenant que nous avons toutes nos fonctions, nous pouvons faire une classe main qui les appelera toute