ECS765

BIG DATA PROCESSING

Andreas Iacovou

180971990

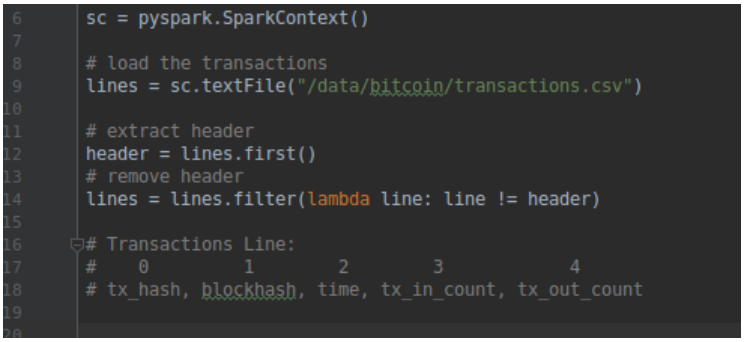
MSc Software Engineering FT

THIS COURSEWORK WAS SUBMITTED BY THE ORIGINAL DEADLINE OF MONDAY 3RD OF DECEMBER

PART A. TIME ANALYSIS

Create a bar plot showing the number of transactions which occurred every month between the start and end of the dataset. What do you notice about the overall trend in the utilization of bitcoin?

This task was implemented in pyspark. First, the data was loaded from the HDFS into an RDD. The first row of the RDD was removed, since it contained the header of our dataset.



```
sc = pyspark.SparkContext()

# load the transactions
lines = sc.textFile("/data/bitcoin/transactions.csv")

# extract header
header = lines.first()
# remove header
lines = lines.filter(lambda line: line != header)

# Transactions Line:
#    0         1        2        3            4
# tx_hash, blockhash, time, tx_in_count, tx_out_count
```

The next step was to then filter out the bad lines of our dataset, as well as the lines that contained transactions that happened outside our timeframe, if any.

Then, mapping over the RDD, the pair (date, 1) was yielded; where date is a string created by converting the timestamp field into the format “YYYY-MM”.

In the reducer phase, the values of each date where summed, therefore getting the number of transactions for each month of each year in our dataset.

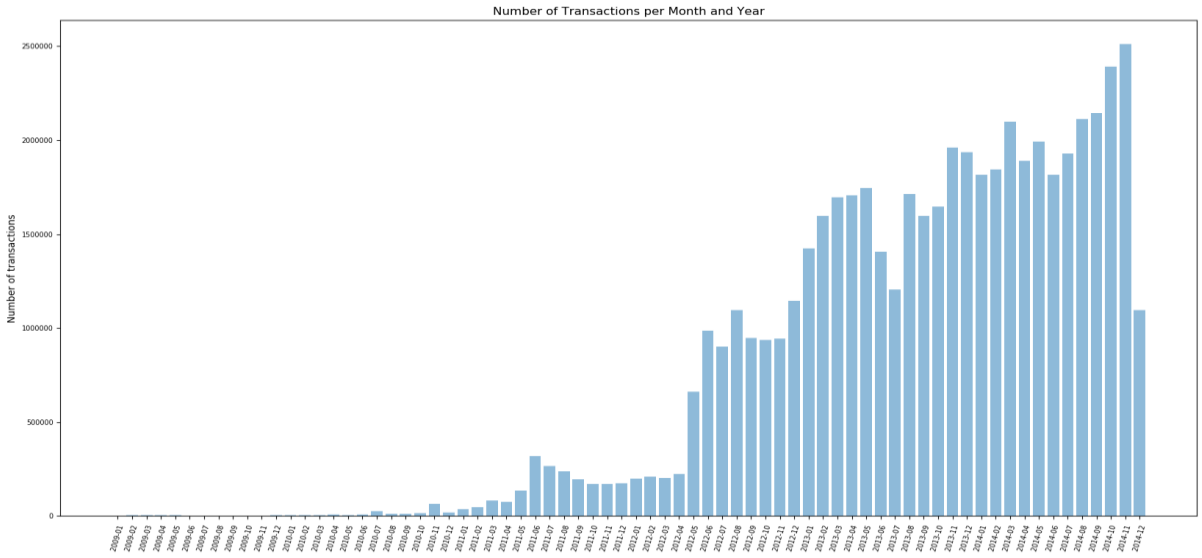
Lastly, the results were sorted by the earliest date.

```
    # filter out bad lines
    # filter out transactions outside our timeframe
    # map through the transactions and yield ("YYYY-MM", 1)
    # get the sum of the values for each key in the reducer
    # sort by earliest date
    filtered_transactions = lines.filter(is_good_line) \
        .filter(in_time) \
        .map(lambda line: (unix_to_date(line.split(',')[2]), 1)) \
        .reduceByKey(lambda x, y: (x+y)) \
        .sortBy(lambda x: x[0])
```

To create the plot, 2 arrays were created. All dates were pushed in one array while the number of transactions of each date were pushed into another. Using the library matplotlib the bar plot was then created, where the dates are on the x axis and the number of transactions on the y axis.

```
    months = []
    num_of_transactions = []

    # save the results in a text file
    filtered_transactions.saveAsTextFile('outBitcoin/TimeAnalysis')

    # collect the results to create a plot
    for pair in filtered_transactions.collect():
        print(pair)
        myString = "{} ".format(pair[0])
        months.append(myString)
        num_of_transactions.append(pair[1])

    y_pos = np.arange(len(months))

    # create a plot with the dates on the x axis and number of transactions on the y axis
    plt.bar(y_pos, num_of_transactions, align='center', alpha=0.5)
    plt.xticks(y_pos, months)
    plt.xticks(rotation=75, size=7)
    plt.ylabel('Number of transactions')
    plt.title('Number of Transactions per Month and Year')
    plt.tight_layout()
    plt.show()
```

The resulting plot is the following:



Although the letters are too small in this picture to be able to view the plot properly, the plot can also be found in a .png file in the path PartA/TimeAnalysisPlot.png

From this plot, one can clearly draw some conclusions about the utilization of bitcoin through time. In January of 2009 for example, it is obvious that the number of bitcoin transactions are close to 0, while in November of 2014 that number exceeded 2500000. One can therefore talk of an exponential growth in bitcoin utilization in just a period of 5 years. Something interesting though, would also be the sudden drop of the number of bitcoin transactions in December of 2014, with almost half the number of transactions of the previous month.

PART B. TOP TEN DONORS:

In this part, pyspark was used once again.

First, the two datasets were loaded into RDDs. The vout dataset was filtered to keep only the lines with public key equal to the one of Wikileaks, and both datasets were filtered from bad lines.

An RDD dictionary was created, named v_out_dict by mapping through the vout dataset and yielding (hash, {}).

Then, by mapping through the vin dataset, (txid,(value,n)) was yielded, to remove the unnecessary fields and have as the first field the key on which the join will be.

```
44      vin_lines = sc.textFile("/data/bitcoin/vin.csv")
45      vout_lines_first = sc.textFile("/data/bitcoin/vout.csv")
46
47      vout_lines = vout_lines_first.filter(is_to_wikileaks)
48
49      vin_lines = vin_lines.filter(is_vin_good_line)
50
51      # VIN file:
52      #  0       1      2
53      # txid, tx_hash, vout
54
55      # VOUT File:
56      #  0      1    2      3
57      # hash, value, n, publicKey
58
59      v_out_dict = vout_lines.map(lambda line: (line.split(',')[0], {}))
60
61      vin_lines = vin_lines.map(lambda x: (x.split(',')[0], (x.split(',')[1], x.split(',')[2])))
62
63      joined_1 = vin_lines.join(v_out_dict)
64
65      joined_1.saveAsTextFile("outBitcoin/TopTen/joined1")
66
67      joined_1 = joined_1.map(lambda x: (x[0], x[1][0][0], x[1][0][1]))
68
69      joined_1.saveAsTextFile('outBitcoin/TopTen/joined1_mapped')
70      #
71      # # Joined 1
72      # #   0       1      2
73      # # tx_id, tx_hash, vout
```

After that the two RDDs were joined, resulting in a new dataset, named joined1, with the following format: tx_id, tx_hash, vout

```
('39d09797204e71a42697439d4870ea9f1726637a5024a1d36aed7dc5d2bf29c3', (('aaed17b9f1d6ad8091afbed8cbd2e2bd015adf9da9fd1f357dfcf1dele3325ee', '0'), {}))
('399adec8c2eb89605dc0db01e23e0d4bc15298337fe1f0025c265fc9769efc28', (('b12113ad02120afd0f2059340b4d111fce890fdf3f1caf1869ad0379b8708abd', '0'), {}))
```

To perform the second join, we created another RDD to have the role of a dictionary, by mapping through joined_1 and yielding (tx_hash,vout), {}), to remove the unnecessary field tx_id and to make tx_hash and vout to be the key (composed of multiple fields) to perform the 2nd join on.

The original vout dataset (with all the wallets) was also mapped then and the fields yielded were ((hash, n), (n, publicKey)). The key (hash,n) is once again the key for the join, while the pair (n, publicKey) is our value.

```
joined_1_dict = joined_1.map(lambda x: ((x[1], x[2]), {}))

vout_lines = vout_lines_first

vout_lines = vout_lines.filter(is_vout_good_line)

vout_lines = vout_lines.map(lambda line: ((line.split(',')[0], line.split(',')[2]), (line.split(',')[1], line.split(',')[3])))

joined_2 = joined_1_dict.join(vout_lines)

joined_2.saveAsTextFile('outBitcoin/TopTen/joined2')
```

After performing the 2nd join we get a dataset in the following format: (tx_hash, vout), ({},(value, publicKey))

```
'6934e6160850be81d3a9e837c07dc18e20beb697e73ec', '1'), ({}, ('0.1', '{1P6CfAkPx73fBkmNVdrmLiMWCLVcFnFEzr}')))
'b1056653ae294cf98eed425eb49d9ec789298a282c061', '1'), ({}, ('10.11', '{1DooAJYazyZSx1sVZ3ophDRXc3xhRQ6oFz}')))
```

After obtaining the results the next step was to map through them and yield (publicKey, float(value)), since we want to group by the publicKey field. In a reducer the values of the value field for each publicKey are summed, obtaining the total number of bitcoins for each publicKey.

The results are then sorted by descending order based on the value field and the top 10 are taken.

```
joined_2 = joined_2.map(lambda x: (x[1][1][1], float(x[1][1][0])))

unsortedResults = joined_2.reduceByKey(lambda x, y: x+y)

top10 = unsortedResults.takeOrdered(10, lambda x: -x[1])


for x in top10:
    print(x)

top10 = sc.parallelize(top10)

top10.saveAsTextFile("outBitcoin/TopTen/Topten")
```

The final results are the following:

```
('{17B6mtZr14VnCKaHkvzqpkuxMYKTvezDcp}', 46515.1894803)
('{19TCgtx62HQmaaGy8WNhLvoLXLr7LvaDYn}', 5770.0)
('{14dQGpcUhejZ6QhAQ9UGVh7an78xoDnfap}', 1931.482)
('{1LNWw6yCxkUmkhArb2Nf2MPw6vG7u5WG7q}', 1894.37418624)
('{1L8MdMLrgkCQJ1htiGRAcP11eJs662pYSS}', 806.13402728)
('{1ECHwzKtRebkymjSnRKLqhQPkHCdDn6NeK}', 648.5199788)
('{18pcznb96bbVE1mR7Di3hK7oWKsA1fDqhJ}', 637.04365574)
('{19eXS2pE5f1yBggdwhPjauqCjS8YQCmnXa}', 576.835)
('{1B9q5KG69tzjhqq3WSz3H7PAxDVTAwNdbV}', 556.7)
('{1AUGSxE5e8yPPLGd7BM2aUxfzbokT6ZYSq}', 500.0)
```

Unfortunately, it is impossible to find who these wallets belong to just by knowing their publicKey.

Blockchain will keep a record of each transaction ever to have occurred in the network, however only the following details about the transfer are made publicly available:

- Amount of cryptocurrency sent
- Sender's address
- Receiver's address
- Date of transfer
- Charged fees
- Number of confirmations

Finding out how much was being donated in pounds is not an easy task, since each transaction took place at a different time and/or date. As already seen, the price of bitcoin fluctuates both widely and wildly. Therefore, to calculate the amount in British pounds, one would have to look at the exact price of bitcoin in pounds, at the specific date and time that each transaction took place.

To do this we would have to join the results of the 2nd join with the transactions dataset based on the transaction hash. Therefore, for each transaction in the results dataset we would now also have the time of each transaction. Ideally, the format would now look like this:

(tx_hash, vout, value, publicKey, time)

After that, we would need to have another dataset that contained the price of bitcoin in pounds through time. That way we would join the 2 datasets based on the time field and get the following format:

(time, tx_hash, vout, value, bitcoinPriceInPounds, publicKey)

By multiplying the value field with the bitcoinPriceInPounds we would get:

(time, tx_hash, vout, value, value_in_pounds, publicKey)

After that, we would map through the data and yield ((publicKey), (value_in_pounds))

We would then carry on doing the same thing as before, summing all the value_in_pounds values for each publicKey, sorting and taking the top 10 results.

For simplicity reasons however, if we skip this process and just decide to calculate the value in pounds using the price of bitcoin today then we would get:

publicKey	Bitcoins	Value in Pounds
17B6mtZr14VnCKaHkvzqpkuxMYKTvezDcp	46515.1894803	150,107,799.69
19TCgtx62HQmaaGy8WNhLvoLXLr7LvaDYn	5770.0	18,620,197.27
14dQGpcUhejZ6QhAQ9UGVh7an78xoDnfap	1931.482	6,233,028.75
1LNWw6yCxkUmkhArb2Nf2MPw6vG7u5WG7q	1894.37418624	6,095,375.48
1L8MdMLrgkCQJ1htiGRAcP11eJs662pYSS	806.13402728	2,593,832.63
1ECHwzKtRebkymjSnRKLqhQPkHCdDn6NeK	648.5199788	2,086,690.58
18pcznb96bbVE1mR7Di3hK7oWKsA1fDqhJ	637.04365574	2,049,764.14
19eXS2pE5f1yBggdwhPjauqCjS8YQCmnXa	576.835	1,856,035,591.58
1B9q5KG69tzjhqq3WSz3H7PAxDVTAwNdbV	556.7	1,791,146.07
1AUGSxE5e8yPPLGd7BM2aUxfzbokT6ZYSq	500.0	1,608,717.50

PART C. DATA EXPLORATION:

Find a dataset containing the prices of bitcoin throughout time (Many sites exist with such information). Utilizing the Spark MlLib library, combine this, characteristics extracted from the data (volume of bitcoins sold per day, transactions per pay), and any other external sources into a model for price forecasting. How far into the future from our subset end date (September 2013) is your model accurate? How does the volatility of bitcoin prices affect the effectiveness of your model?

For the purposes of this part, pyspark was used once again.

A dataset was found, containing bitcoin prices though time, from 01/01/2012 to 11/11/2018. Due to the fact that the file is very large (200MB) QMPlus says that the zip folder is too big. Therefore, only a sample of this file is included in the submitted zip folder.

The file was loaded into a spark dataframe that has the following format:

Timestamp,Open,High,Low,Close,Volume_(BTC),Volume_(Currency),Weighted_Price

- Timestamp: the timestamp of the prices
- Open: the price of bitcoin at the opening of the time interval
- High: the highest price of bitcoin during the time interval
- Low: the lowest price of bitcoin during the time interval
- Close: the price of bitcoin at the closing of the time interval
- Volume_(BTC): the volume of bitcoins exchanged
- Volume_(Currency): the volume of bitcoins exchanged, in US dollars
- Weighted_Price: Volume_(Currency)/Volume_(BTC)

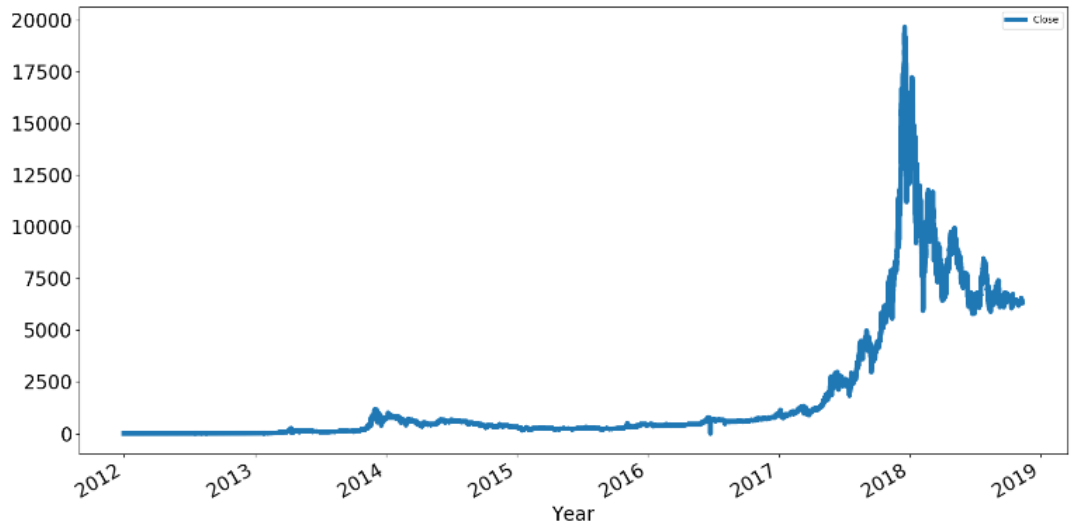
As seen above, our dataset contains all the relevant fields that we would get by joining with the files provided in our coursework. Hence it was decided to skip joining our datasets, as such a thing would also imply a decrease in our data, since the new dataset starts from 2012 and the coursework datasets end at 2014.

Consequently, the first step as always, was to remove the bad and null lines. Our dataset, was also not consistent with the number of lines and bitcoin prices for each date. For example, there could be 3 lines with data about the prices of bitcoin on 01/01/2012 (so 3 different prices for each day) and 1 line with prices for 02/01/2012, so just 1 price.

To solve this problem, it was decided that the best approach would be to convert the timestamp field into date and then group together all lines with the same date, by calculating daily averages for each field. (e.g. Date, Open_Daily_Avg, High_Daily_Avg, …)

```
bit_df = sql.read.format('com.databricks.spark.csv').options(header='true', inferschema='true').load('input/bitstamp.csv')


bit_df = bit_df.na.drop()

# bit_df = bit_df.limit(10000000)
#

bit_df = bit_df.withColumnRenamed('Volume_(BTC)', 'Volume_BTC').withColumnRenamed('Volume_(Currency)', 'Volume_Currency')


bit_df = bit_df.groupBy(from_unixtime("Timestamp", "yyyy-MM-dd").alias("Date")).agg(mean('Open').alias('Open'),
    mean('High').alias('High'), mean('Low').alias('Low'), mean('Close').alias('Close'),
    mean('Volume_BTC').alias('Volume_BTC'), mean('Volume_Currency').alias('Volume_Currency'), mean("Weighted_Price").alias("Weighted_Price"))

```

A good way of going forward, was to try and visualize the data that had to be predicted. Therefore, a plot was created, visualizing the Close price of bitcoin throughout the dataset.



As can be seen in the above plot, although bitcoin prices experienced a relatively smooth rise in the first couple of years, that was followed by an exponential rise of the prices, wild and wide fluctuations and an exponential drop. Therefore, the extreme difficulty of accurately predicting bitcoin prices is easily understandable just by looking at this plot.

To move on with the prediction, a very basic assumption had to be made. That we're predicting the price in a black box environment. We do not use other sources of knowledge such as news, Twitter feeds, and others to predict how the market would react to them, which is a very basic factor in the change of bitcoin prices. For simplicity reasons the only data we use is price and volume.

It was then decided, that the way to go about predicting future bitcoin prices, would be to try and predict the price of bitcoin 30 days into the future. Hence, we created a new column in our dataframe called Price_After_Month. For each row and therefore date (since each row has a unique date) in our dataframe, we grabbed the closing price of the date 30 days ahead and made it the Price_After_Month of current date and row.

```
48
49  w = Window().partitionBy().orderBy(col("Date"))
50
51  bit_df = bit_df.withColumn("Price_After_Month", lag("Close", -30, 'NaN').over(w))
52
53  bit_df = bit_df.na.drop()
54
```

```
1  Date      ,     Open      ,     High      ,     Low       ,     Close     ,Volume_(BTC)     ,Volume_(Currency) ,  Weighted_Price ,Price_After_Month
2  2012-01-01,4.806666666666667,4.806666666666667,4.806666666666667,4.806666666666667,7.200666666666667,35.259719999999994,4.806666666666667,5.619024390243903
3  .
4  .
5  .
6  2012-01-31,5.611463414634148,5.619024390243903,5.598536585365855,5.619024390243903,2.6952032521951224,15.587281248292681,5.6140454556560995,5.002666666666667
```

As seen in the picture above, the Price_After_Month value of 01-01-2012 is equal to the Close value of 31-01-2012.

The next step is the model training for the prediction. First, the data of the dataframe need to be vectorized. Using a VectorAssembler() all of the columns except the date and Price_After_Month columns are vectorized and a new column named features is created.

The dataset is then split into train and test data; 70% being train data and 30% being test data.

A linear regression model is then trained for the prediction.

```
vectorAssembler = VectorAssembler(inputCols=['Open', 'High', 'Low', 'Close', 'Volume_BTC', 'Volume_Currency', 'Weighted_Price'], outputCol='features')
# vectorAssembler = VectorAssembler(inputCols=['DayMeanPrice'], outputCol='features')
vbit_df = vectorAssembler.transform(bit_df)

splits = vbit_df.randomSplit([0.7, 0.3])
train_df = splits[0]
test_df = splits[1]

lr = LinearRegression(featuresCol='features', labelCol='Price_After_Month', maxIter=1000, regParam=0.3, elasticNetParam=0.8)
lr_model = lr.fit(train_df)
print("Coefficients: " + str(lr_model.coefficients))
print("Intercept: " + str(lr_model.intercept))

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

Lastly, we print some valuable metrics for our train data and perform the prediction on the test data, evaluate the model and print some valuable metrics again.

```
lr_predictions = lr_model.transform(test_df)
lr_predictions.select("prediction", "Price_After_Month", "features").show(5)
lr_evaluator = RegressionEvaluator(predictionCol="prediction", labelCol="Price_After_Month", metricName="r2")
print("R Squared (R2) on test data = %g" % lr_evaluator.evaluate(lr_predictions))

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

First 10 rows of prediction:

```
Date       ,    prediction    ,Price_After_Month
2011-12-31,137.53277523765485,5.544999999999999
2012-01-01,223.62115048204618,5.619024390243903
2012-01-02,162.7222882495672,5.78875
2012-01-07,260.9814883067307,5.52625
2012-01-14,253.49686844184262,5.648181818181818
2012-01-17,244.65390762781826,4.575588235294118
2012-01-21,178.31978721942448,4.560714285714287
2012-01-22,209.92390539486402,4.483437500000001
2012-02-08,189.82384058735,5.011666666666668
```

Last 10 rows of prediction:

```
2018-09-12,5880.635140262816,6200.056210045651
2018-09-13,6092.29921370494,6195.578832891246
2018-09-17,6003.614227961239,6439.336028146999
2018-09-18,5905.402266115898,6427.678450244705
2018-09-22,6308.630093154443,6405.119255404312
2018-09-27,6130.815686629534,6406.077371727761
2018-10-01,6173.949927813033,6277.983404423378
2018-10-02,6126.927330894477,6311.7777797356775
2018-10-06,6094.200318436699,6404.265751211624
2018-10-10,6133.448406240196,6365.227866666656
```

As obvious from the above pictures, our model is much more accurate on late prediction than predictions on earlier dates. This may be due to both the nature of the linear regression model as well as the nature of bitcoin and volatility of its price. During the first couple of years of our dataset, it is obvious from the previous plot, that the rise in bitcoin prices didn't follow the same exponential rise of later years. Since our linear regression model was trained on the whole train set, including dates with exponential growth in bitcoin prices, that could perhaps be the reason why the predictions on early dates show higher prices than the real ones.

Lastly, evaluating the model on the test data, based on the mean squared error and the unadjusted coefficient of determination (r^2) we get:

```
R Squared (R2) on test data = 0.836656
Root Mean Squared Error (RMSE) on test data = 1296.43
```