Ethereum Analysis

December 14, 2020

Student ID: 180428302
Name: David von Huth
Module: Big Data Processing ECS640U

In this coursework, snippets of code have been included in the report for explanation purposes, with the full source code available in a different folder (in the same zipped submission file).

partA

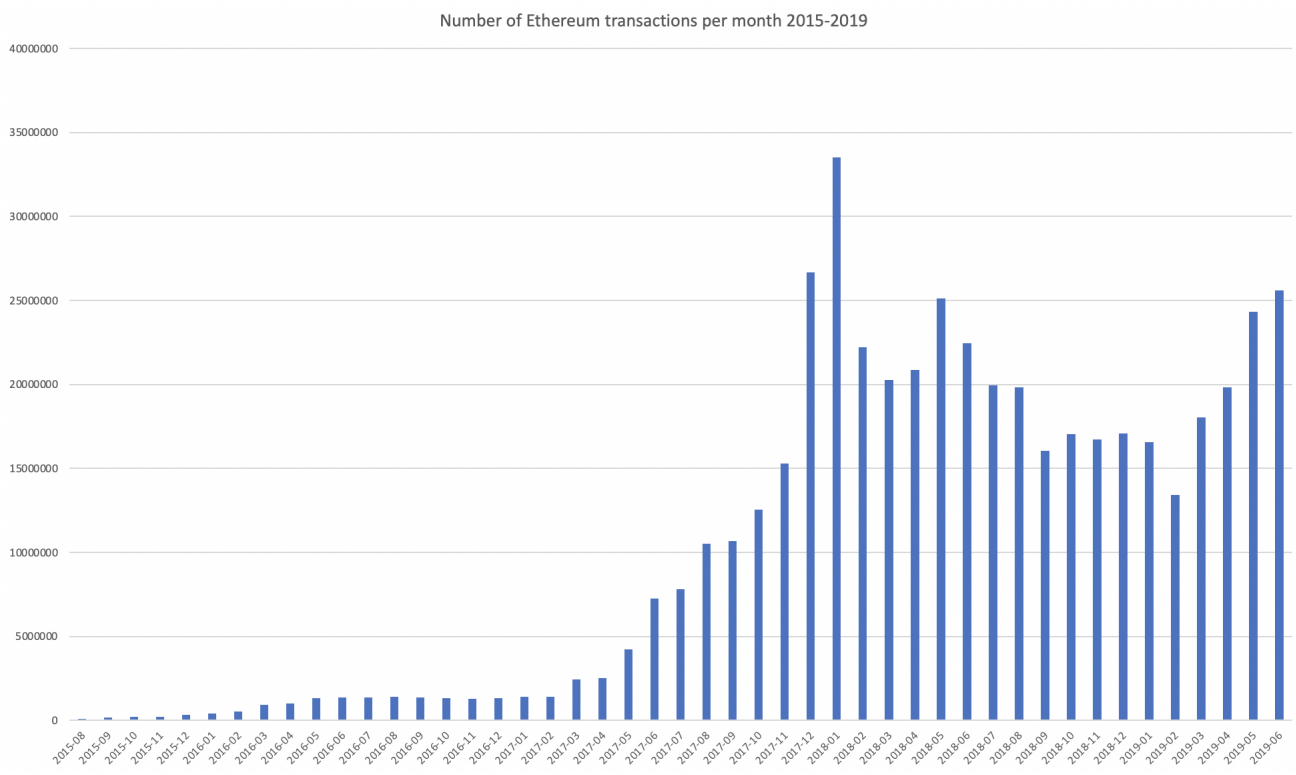
Number of transactions per month

JOB-ID: job_1606730688641_4013

The below code was run for finding the number of transactions per month:

```
class time_analysis_nr_trans(MRJob):

    def mapper(self, _, line):

        try:
            fields = line.split(',')
            timestamp = int(fields[6]) #Month and year
            date = datetime.utcfromtimestamp(timestamp).strftime('%Y-%m')

            yield(date,1)

        except:
            pass



    def reducer(self, date, values):
        yield(date, sum(values))




if __name__ == '__main__':
    time_analysis_nr_trans.run()
```

The following graph could thereafter be constructed:



The output values from the MapReduce job were unsorted initially. The values were loaded into Microsoft Excel, where they were first sorted according to date, and then graphed.

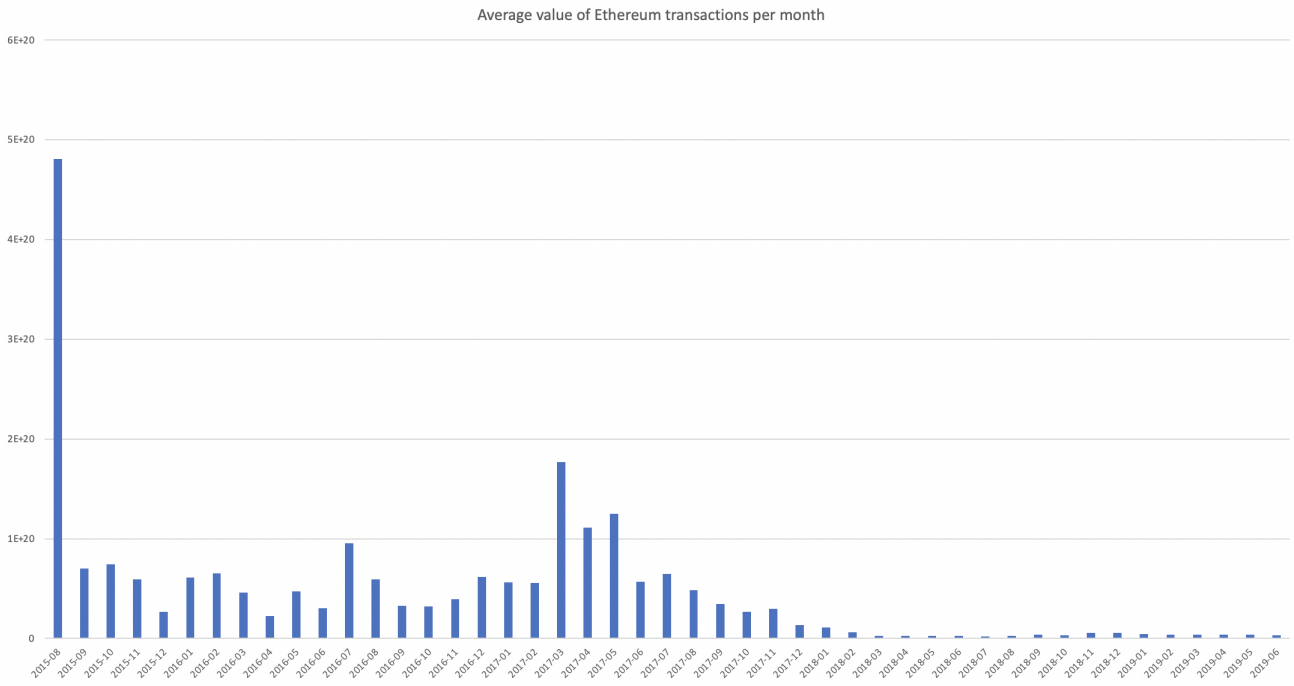
Average value of transactions per month

JOB-ID: job_1606730688641_4015

The below code was run for finding the average value of transactions per month:

```
class time_analysis_avg_value(MRJob):

    def mapper(self, _, line):

        try:
            fields = line.split(',')
            timestamp = int(fields[6]) #Month and year
            date = datetime.utcfromtimestamp(timestamp).strftime('%Y-%m')
            value = int(fields[3])

            yield(date,value)

        except:
            pass



    def reducer(self, date, values):
        iterator = list(values)
        average = int(sum(iterator)/len(iterator))
        yield(date, average)


if __name__ == '__main__':
    time_analysis_avg_value.run()
```

Excel was also used for sorting the MapReduce output values, as well as graphing.



partB

JOB-1 ID: job_1606730688641_4025
JOB-2 ID: job_1606730688641_4037

The following code was run for this task:

```
class top_ten_popularServices(MRJob):
    def mapper1(self, _,line):
        try:
            if (len(line.split(',')) == 5): #CONTRACTS
                fields = line.split(',')
                c_address = fields[0]
                yield(c_address, (None,1))

            elif(len(line.split(',')) == 7): #TRANSACTIONS,
                fields = line.split(',')
                address = fields[2]
                amount = int(fields[3])
                yield(address, (amount, 2))
        except:
            pass

    def reducer1(self, address, values):
            amount = 0
            c_address = None

            for value in values:
                if value[1] == 1:
                    c_address = address
                elif value[1] == 2:
                    amount += value[0]

            if c_address != None:
                yield(None, (c_address, amount))

    def reducer2(self, _, values):
            sorted_values = sorted(values, reverse=True, key=lambda x: x[1])
            top_ten = sorted_values[0:10]

            for contract in top_ten:
                yield("{} - {}".format(contract[0], contract[1]), None)

    def steps(self):
        return [MRStep(mapper=self.mapper1,
                       reducer=self.reducer1),
                MRStep(reducer=self.reducer2)]

if __name__ == '__main__':
    top_ten_popularServices.run()
```

The program works by performing a repartition join between the contract data-set and the transaction data-set. By considering the address of each contract (from the contract data-set), as well as the amount of ether that each contract-address received (from the transactions data-set), the program calculates the total amount of ether that each contract-address has received. The contracts are then sorted, with the top ten displayed below:

```
"0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444 - 841551008099658658227267776"
"0xfa52274dd61e1643d2205169732f29114bc240b3 - 45787484483189352986478805"
"0x7727e5113d1d161373623e5f49fd568b4f543a9e - 45620624001350712557268573"
"0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef - 43170356092262468919298969"
"0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8 - 27068921582019542499882877"
"0xbfc39b6f805a9e40e77291aff27aee3c96915bdd - 21104195138093660050000000"
"0xe94b04a0fed112f3664e45adb2b8915693dd5ff3 - 15562398956802112254719409"
"0xbb9bc244d798123fde783fcc1c72d3bb8c189413 - 11983608729202893846818681"
"0xabbb6bebfa05aa13e908eaa492bd7a8343760477 - 11706457177940895521770404"
"0x341e790174e3a4d35b65fdc067b6b5634a61caea - 837900075191775562405750O"
```

partC

JOB-1 ID: job_1606730688641_4044
JOB-2 ID: job_1606730688641_4045

The following code was run for this task:

```
class top_ten_miners(MRJob):

    def mapper1(self, _, line):

        try:
            fields = line.split(',')
            miner = fields[2]
            size = int(fields[4])
            yield(miner, size)

        except:
            pass



    def reducer1(self, miner, sizes):
        yield(None, (miner,sum(sizes)))

    def reducer2(self, _, values):
        sorted_values = sorted(values, reverse=True, key = lambda x: x[1])
        top_ten = sorted_values[0:10]

        for miner in top_ten:
            yield("{} - {}".format(miner[0], miner[1]), None)

    def steps(self):
        return [MRStep(mapper=self.mapper1,
                       reducer=self.reducer1),
                MRStep(reducer=self.reducer2)]



if __name__ == '__main__':
    top_ten_miners.run()
```

The program works by splitting the task into two jobs. The "blocks" data-set was used, where the size of each block was aggregated for the particular miner that mined that block. The summed size values for each miner is then sorted, with the top ten miners displayed below:

```
"0xea674fdde714fd979de3edf0f56aa9716b898ec8 - 23989401188"
"0x829bd824b016326a401d083b33d092293333a830 - 15010222714"
"0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c - 13978859941"
"0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5 - 10998145387"
"0xb2930b35844a230f00e51431acae96fe543a0347 - 7842595276"
"0x2a65aca4d5fc5b5c859090a6c34d164135398226 - 3628875680"
"0x4bb96091ee9d802ed039c4d1a5f6216f90f81b01 - 1221833144"
"0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb - 1152472379"
"0x1e9939daaad6924ad004c2560e90804164900341 - 1080301927"
"0x61c808d82a3ac53231750dadc13c777b59310bd9 - 692942577"
```

partD

Scam Analysis

JOB-1 ID: job_1607539937312_6466
JOB-1 ID: job_1607539937312_6487

A single typo was found in the provided scams.json file, where one specific scam had the label "Scam" as opposed to "Scamming". It was decided to change this to "Scamming", as it was assumed to be a mistake.

The below MapReduce code was run for this task:

```
class scam_analysis(MRJob):
    def mapper1(self, _,line):
        try:
            if(len(line.split(',')) == 7):   #TRANSACTIONS,
            smart contract
                fields = line.split(',')
                address = fields[2]
                amount = int(fields[3])
                yield(address, (amount, 1))

            else:                            #SCAMS
                fields = line.split(',')
                sc_category = fields[0]
                sc_addresses = fields[1:]

                for address in sc_addresses:
                    yield(address, (sc_category,2))
        except:
            pass

    def reducer1(self, address, values):

            amount = 0
            category = None

            for value in values:
                if value[1] == 1:
                    amount += value[0]
                elif value[1] == 2:
                    category = value[0]

            if category != None:
                yield(category, amount)


    def reducer2(self, category, values):
            yield(category, sum(values))

    def steps(self):
        return [MRStep(mapper=self.mapper1,
                       reducer=self.reducer1),
                MRStep(reducer=self.reducer2)]

if __name__ == '__main__':
    scam_analysis.run()
```

The jobs above aimed to find the scam category which proved most lucrative for scammers. It was measured by looking at the total accumulated sum that each scammer address had received, summing up for each of the labelled categories. The results can be seen below, where it is clear that the general "Scamming" category prevailed:

```
"Scamming"      383361628624448966503 97
"Fake ICO"      135645756688962997967 8
"Phishing"      269993757940874255564 05
```

Price forecasting

The MLlib of Spark was used to create a Linear Regression model. Below are some important parts of the code and outputs from the training of the model.

First, each line of the input file is "cleaned" so as to filter away any rows that may contain errors.For each row, the two columns of interest are then extracted, namely the Ethereum *Closing Price* as well as the *Date*. For our regression model, the date will server as the independent variable, while the price will serve as the dependent variable. The two variables are then used to form the columns of a dataframe, utilizing the Spark transformation operation *map* to achieve this. Furthermore, the date is changed to a unix-timestamped type in order to yield a numeric value that regression can be performed on. The original date column is afterwards dropped.

```
sc = pyspark.SparkContext()
spark = SparkSession(sc)

def good_line(line):
    try:
        fields = line.split(',')
        if len(fields)!=6:
            return False

        float(fields[2])
        return True

    except:
        return False



lines = sc.textFile('./ETH_USD_2015-08-09_2020-11-01-CoinDesk.csv')
clean_lines = lines.filter(good_line)


#Extracting only dates and prices
df = clean_lines.map(lambda line: Row(Date = line.split(',')[1],
                                      Closing_price = line.split(',')[2])).toDF()

#Casting the price to FloatType
df = df.withColumn('Closing_price', df['Closing_price'].cast(FloatType()))
df = df.withColumn("timestamps", unix_timestamp('Date','yyyy-MM-dd'))
df = df.drop('Date')
```

Furthermore, another map transformation is performed, creating dense vectors that the final ML model will be able to interpret. The date timestamps are now under the column name "features". It is common practice to normalize independent features, which is further done on the timestamped data:

```
input_data = df.rdd.map(lambda x: (x[0], DenseVector(x[1:])))
df = spark.createDataFrame(input_data, ["Closing_price", "features"])
standardScaler = StandardScaler(inputCol="features", outputCol="dates_scaled")
scaler = standardScaler.fit(df)
scaled_df = scaler.transform(df)
```

yielding the below dataframe:

```
+------------------+-------------+--------------------+
|     Closing_price|     features|        dates_scaled|
+------------------+-------------+--------------------+
|0.9090459942817688|[1.4390748E9]||[29.902079658735666]|
|0.6923210024833679|[1.4391612E9]||[29.903874936981463]|
|0.6680669784545898|[1.4392476E9]| [29.90567021522726]|
| 0.850151002407074| [1.439334E9]||[29.907465493473058]|
|  1.26602303981781|[1.4394204E9]||[29.909260771718852]|
|1.9514600038528442|[1.4395068E9]| [29.91105604996465]|
|1.5912189483642578|[1.4395932E9]||[29.912851328210447]|
|  1.69370698928833|[1.4396796E9]||[29.914646606456245]|
|1.4232439994812012| [1.439766E9]||[29.916441884702042]|
|1.1995949745178223|[1.4398524E9]| [29.91823716294784]|
|1.1828370094299316|[1.4399388E9]||[29.920032441193637]|
|   1.2795490026474|[1.4400252E9]||[29.921827719439435]|
|1.4474619626998901|[1.4401116E9]| [29.92362299768523]|
|1.3665599822998047| [1.440198E9]||[29.925418275931026]|
|1.3208719491958618|[1.4402844E9]||[29.927213554176824]|
| 1.374640941619873|[1.4403708E9]| [29.92900883242262]|
| 1.209496021270752|[1.4404572E9]| [29.93080411066842]|
|1.0545769929885864|[1.4405436E9]||[29.932599388914216]|
|  1.18259596824646|  [1.44063E9]||[29.934394667160014]|
|1.1549899578094482|[1.4407164E9]| [29.93618994540581]|
+------------------+-------------+--------------------+
only showing top 20 rows

root
 |-- Closing_price: double (nullable = true)
 |-- features: vector (nullable = true)
```

Some summary information on the target variable is then printed:

```
+-------+------------------+
|summary|      Closing_price|
+-------+------------------+
|  count|              1872|
|   mean| 209.15359582732884|
| stddev| 227.46975113886927|
|    min|0.42839398980140686|
|    max|    1405.2099609375|
+-------+------------------+
```

With the final regression model yielding the following Root Mean Squared Error and R2-Score:

```
Coefficient:  [1.7021431876249117e-06]
Intercept:  -2379.886743511116
RMSE:  215.12785579963915
R2-Score:  0.12757000647780414
```

Performance is sub-optimal, which is likely due to the lack of hyper-parameter tuning in the initilization of the model. More complex models exist, for instance Random Forests, which likely would yield better results. Online sources were used for the completion of the above task [1]

References

[1] K. Willems, Apache Spark Tutorial: ML with PySpark, (28/07/2017), source = "https://www.datacamp.com/community/spark-tutorial-machine-learningmodel"