ECS640U/ECS765P - Big Data Processing

COURSEWORK

Analysis of Ethereum Transactions and Smart Contracts

Name: Palaniappan Balasubramanian

Student ID: 220196730

PART A - Time Analysis (25%)

Create a bar plot showing the number of transactions occurring every month between the start and end of the dataset.

Create a bar plot showing the average value of transaction in each month between the start and end of the dataset.

Note: As the dataset spans multiple years and you are aggregating together all transactions in the same month, make sure to include the year in your analysis.

Note: Once the raw results have been processed within Spark you may create your bar plot in any software of your choice (excel, python, R, etc.)

Input file: transactions.csv

Task 1: Create a bar plot showing the number of transactions occurring every month between the start and end of the dataset.

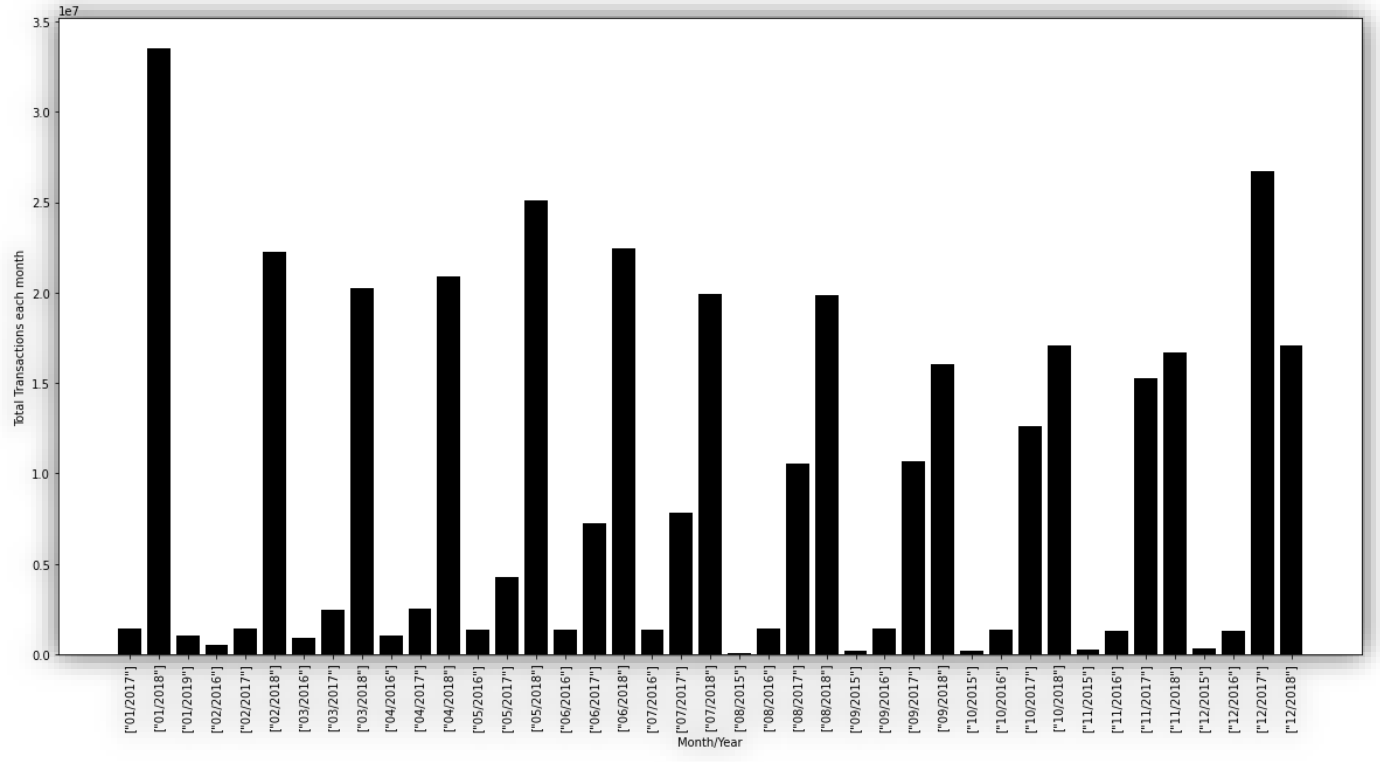
Source code attached: t1.py
PartAj1.ipynb

Explanation:
In order to do the first task, I get the time and the total number of transactions each month. So I mapped the date and the count from the transaction dataset to get the number of transactions every month and reduced by reduceByKey function to get the total number of transactions each month. The output is saved in a text file called “total_transactions.txt”. Then the text file is converted into csv to represent the total number of transactions occurring each month using google colab (PartAj1.ipynb file).

Execution Command: ccc create spark t1.py -d -s

Output file: transactions_total.txt

Bar Plot: Total Transactions vs Time



Task 2: Create a bar plot showing the average value of transaction in each month between the start and end of the dataset.

Source code attached: t2.py

PartAj2.ipynb

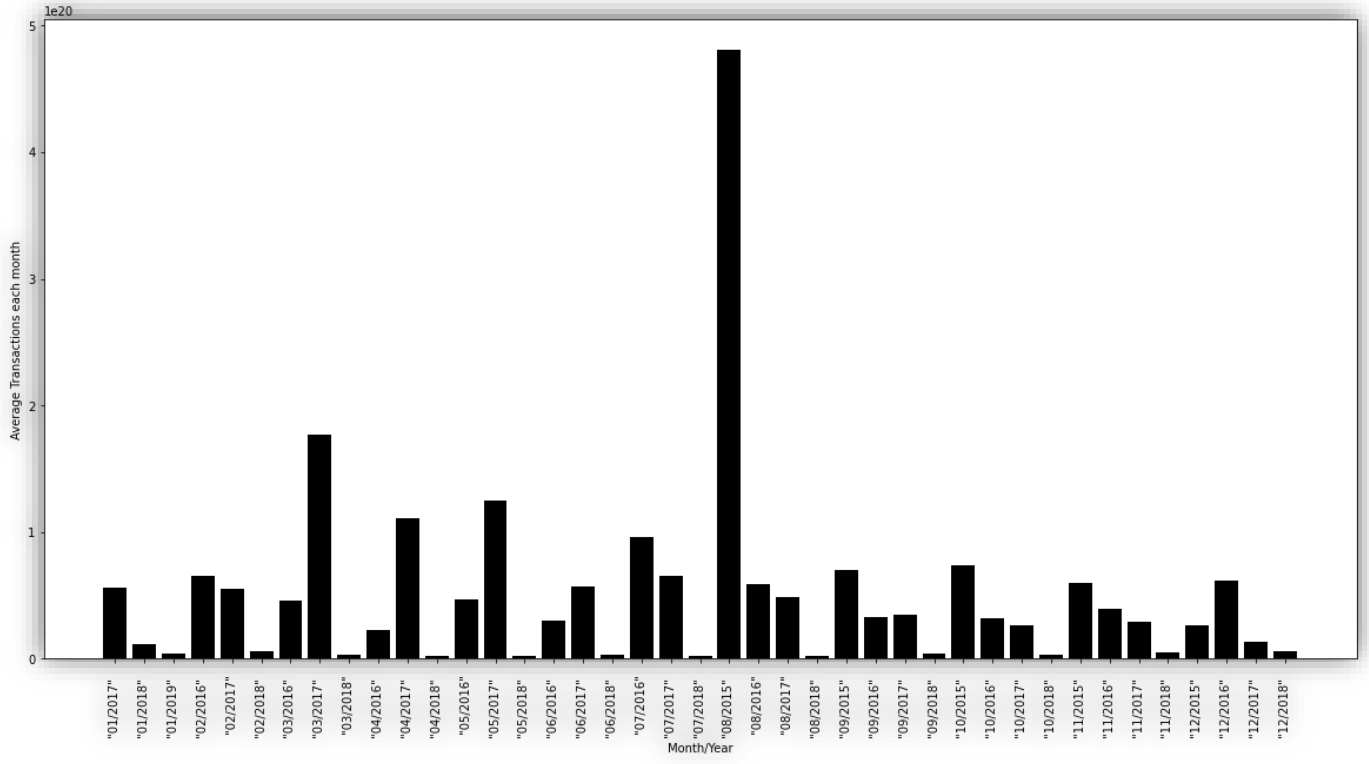
Explanation:

In order to do the second task, I get the time and the average value of transactions each month. So I mapped the date, the value and the count from the transaction dataset to get the values and number of transactions every month and reduced by reduceByKey function to get the total values and total number of transactions each month. Then the data is mapped with the date and the total value is divided by the total number of transactions to get the average values of transactions. The output is saved in a text file called “transactions_avg.txt”. Then the text file is converted into csv to represent the average value of transactions occurring each month using google colab (PartAj2.ipynb file).

Execution Command: ccc create spark t2.py -d –s

Output file: transactions_avg.txt

Bar Plot: Average Transactions vs Time



PART B - Top Ten Most Popular Services (25%)

Evaluate the top 10 smart contracts by total Ether received. You will need to join **address** field in the contracts dataset to the **to_address** in the transactions dataset to determine how much ether a contract has received.

Input file: transactions.csv
Contracts.csv

Source code attached: partb.py

Explanation:

Contracts and transactions dataset is read to get started with this part. The to_address and the value is mapped from transactions dataset and the address and the count is mapped from contracts dataset. Both the datasets are joined together with the to_address and address using .join() function. Then the addresses and the values are mapped together. The top 10 smart contracts are found using takeOrdered function. The output is saved in a text file called "top10_smart_contracts.txt".

Execution Command: ccc create spark partb.py -d -s

Output file: top10_smart_contracts.txt

Output:

RANK	ADDRESS	ETHEREUM VALUE
1.	"0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444"	84155363699941767867374641
2.	"0x7727e5113d1d161373623e5f49fd568b4f543a9e"	45627128512915344587749920
3.	"0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef"	42552989136413198919298969
4.	"0xbfc39b6f805a9e40e77291aff27aee3c96915bdd"	21104195138093660050000000
5.	"0xe94b04a0fed112f3664e45adb2b8915693dd5ff3"	15543077635263742254719409
6.	"0xabbb6bebfa05aa13e908eaa492bd7a8343760477"	10719485945628946136524680
7.	"0x341e790174e3a4d35b65fdc067b6b5634a61caea"	8379000751917755624057500
8.	"0x58ae42a38d6b33a1e31492b60465fa80da595755"	2902709187105736532863818
9.	"0xc7c7f6660102e9a1fee1390df5c76ea5a5572ed3"	1238086114520042000000000
10.	"0xe28e72fcf78647adce1f1252f240bbfaebd63bcc"	1172426432515823142714582

PART C - Top Ten Most Active Miners (10%)

Evaluate the top 10 miners by the size of the blocks mined. This is simpler as it does not require a join. You will first have to aggregate **blocks** to see how much each miner has been involved in. You will want to aggregate **size** for addresses in the **miner** field. This will be similar to the wordcount that we saw in Lab 1 and Lab 2. You can add each value from the reducer to a list and then sort the list to obtain the most active miners.

Input file: blocks.csv

Source code attached: partc.py

Explanation:

This part requires only blocks dataset. The miner and the size is mapped together and reduced by reduceByKey function to get the block size. The top 10 miners are found using takeOrdered and saved in a text file called “top10_miners.txt”.

Execution Command: ccc create spark partc.py -d -s

Output file: top10_miners.txt

Output:

RANK	MINER	BLOCK SIZE
1.	"0xea674fdde714fd979de3edf0f56aa9716b898ec8"	17453393724
2.	"0x829bd824b016326a401d083b33d092293333a830"	12310472526
3.	"0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c"	8825710065
4.	"0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5"	8451574409
5.	"0xb2930b35844a230f00e51431acae96fe543a0347"	6614130661
6.	"0x2a65aca4d5fc5b5c859090a6c34d164135398226"	3173096011
7.	"0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb"	1152847020
8.	"0x4bb96091ee9d802ed039c4d1a5f6216f90f81b01"	1134151226
9.	"0x1e9939daaad6924ad004c2560e90804164900341"	1080436358
10.	"0x61c808d82a3ac53231750dadc13c777b59310bd9"	692942577

PART D - Data exploration (40%)

Scam Analysis

1. **Popular Scams:** Utilising the provided scam dataset, what is the most lucrative form of scam? How does this change throughout time, and does this correlate with certain known scams going offline/inactive? To obtain the marks for this category you should provide the id of the most lucrative scam and a graph showing how the ether received has changed over time for the dataset. (20/40)

Input file: transactions.csv
scams.json

Source code attached: convert.py
popscams.py
ether_vs_time_plot.csv

Explanation:

Firstly, we have to convert the scams.json to csv file. This was successfully done using convert.py file. Now the scams.csv file has identity, name, url, coin, category, subcategory, index, and status as its parameters. Now, in order to access the file to obtain the required results, the scams.csv file is uploaded to the data repository bucket.

Secondly, we have to get the IDs of most lucrative forms of scam. To do this, the scams.csv and transactions.csv are read. The index, Id, and category is mapped from scams.csv and the address and ether value is mapped from transactions.csv. Both the mapped datasets are joined using addresses with .join() function. Then, the function is reduced by reduceByKey function to get the total ether profited and mapped with the Id and scam type as key and the total ether profited as the value. To get the most lucrative forms of scam, the takeOrdered function is used to get the top 15 most lucrative scams. The output is stored in “most_lucrative_scams.txt” file.

Finally, to get how the total ether received changed over time, we use the transactions.csv and scams.csv. The index and category is mapped from scams.csv and the address, date and the value is mapped from transactions.csv. Similar to the previous part, both the datasets are joined together with .join() function. Then the date and scam type is mapped as key and ether value as value. Then, the value is reduced by reduceByKey function to get the total ether received for each month. The output is saved in “ether_vs_time.txt” file. I used excel to plot the total ether received over time (ether_vs_time_plot.csv).

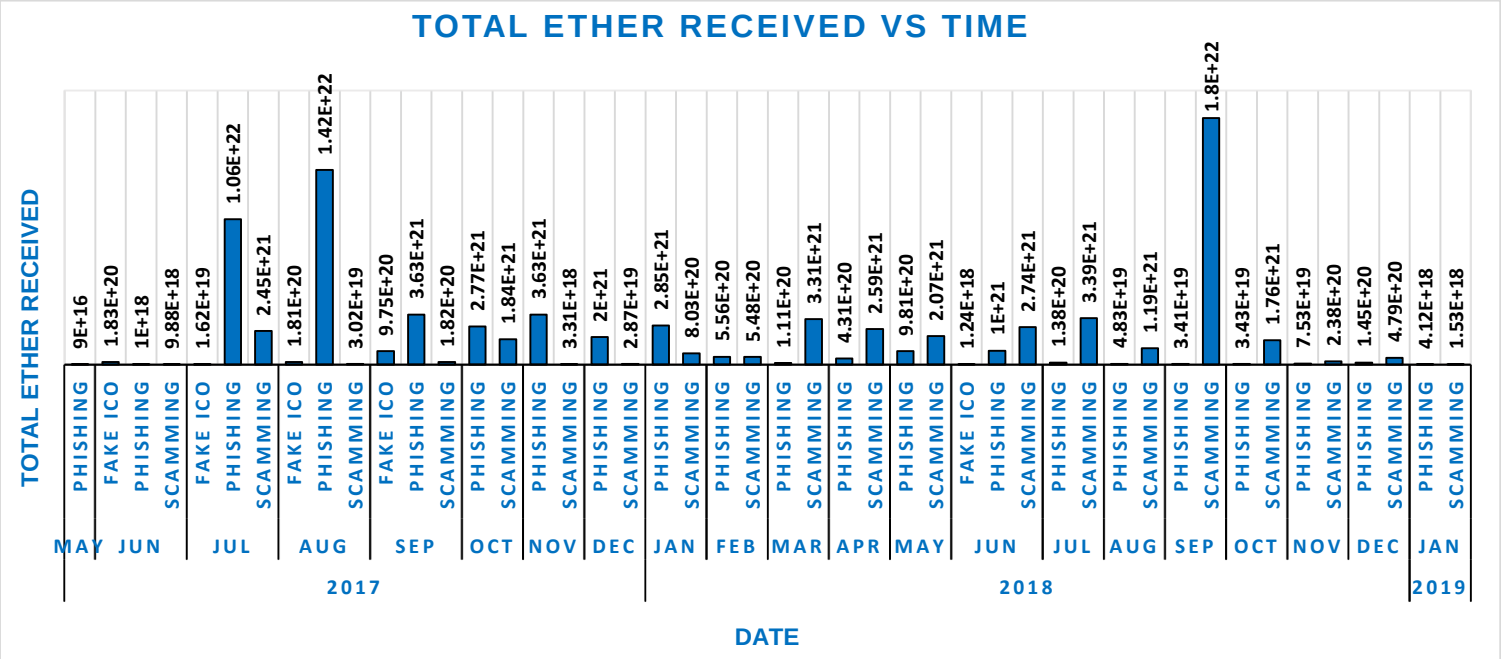
Execution Command: python3 convert.py (to convert scams.json to csv)
ccc create spark popscams.py -d -s

Output file: scams.csv
most_lucrative_scams.txt
ether_vs_time.txt

Output:

Rank	Scam ID	Scam Type	Total Ether Profited
1.	5622	Scamming	1.6709083588072808e+22
2.	2135	Phishing	6.583972305381559e+21
3.	90	Phishing	5.972589629102411e+21
4.	2258	Phishing	3.462807524703738e+21
5.	2137	Phishing	3.389914242537183e+21
6.	2132	Scamming	2.428074787748575e+21
7.	88	Phishing	2.0677508920135265e+21
8.	2358	Scamming	1.8351766714814893e+21
9.	2556	Phishing	1.803046574264181e+21
10.	1200	Phishing	1.63057741913309e+21
11.	2181	Phishing	1.1639041282770013e+21
12.	41	Fake ICO	1.1513030257909173e+21
13.	5820	Scamming	1.1339734671862086e+21
14.	86	Phishing	8.944561496957756e+20
15.	2193	Phishing	8.827100174717214e+20

Plot: Total Ether Received VS Time



2. **Wash Trading:** Wash trading is defined as "Entering into, or purporting to enter into, transactions to give the appearance that purchases and sales have been made, without incurring market risk or changing the trader's market position" Unregulated exchanges use these to fake up to 70% of their trading volume? Which addresses are involved in wash trading? Which trader has the highest volume of wash trades? How certain are you of your result? More information can be found at https://dl.acm.org /doi/pdf/10.1145/ 3442381.3449824. One way to attempt this is by using Directed Acyclic Graphs (DAGs). Keep in mind if that if you try to load the entire dataset as a graph you may run into memory problems on the cluster so you will need to filter the dataset somewhat before attempting this. This is part of the challenge. Other approaches such as measuring ether balance over time are also possible but you will need to discuss accuracy concerns. Marks will be awarded in a scale relative to the sophistication of your solution. If you can detect simple wash trading between two participants that is worth 20 marks. If you can detect more complicated wash trading between three or four participants that is worth 30 marks. If you can detect wash trading between an arbitrary number of participants that is worth full marks. (40/40)

Input file: transactions.csv

Source code attached: washtrade.py

Explanation:

There is plethora of scenarios for wash trading. One of the most common wash trading is self-trading where the trading happens between same addresses. The reference for this type of wash trading is from the paper https://arxiv.org/pdf/2102.07001.pdf. One can clearly see from the paper in 6.1 Wash Trading Structures, "On EtherDelta, the vast majority of wash trades are performed as self-trades, where an account is able to trade with itself".

To find the users who self-traded, the from_address and to_address from transactions.csv is taken into account. A data frame is created with from_address, to_address, value, and date. Then, to get the rows with same column values, the data frame is filtered if the from_address and to_address are equal with the help of the library pyspark.sql.functions. The from_address and to_address is mapped as the key and the ether value as the value from the dataframe created and reduced by reduceByKey function to get the total ether value that each address have obtained using self-trades. TakeOrdered function is used to get the top 10 addresses that performed self-trading. The output is saved in "top10_washtrade.txt" file.

Execution Command: ccc create spark washtrade.py -d -s

Output file: top10_washtrade.txt

Output:

RANK	SELF TRADE ADDRESSES	TOTAL VALUE
1.	"0x02459d2ea9a008342d8685dae79d213f14a87d43"	1.9548531332493176e+25
2.	"0x32362fbfff69b9d31f3aae04faa56f0edee94b1d"	5.295490520134208e+24
3.	"0x0c5437b0b6906321cca17af681d59baf60afe7d6"	2.3771525723546656e+24
4.	"0xdb6fd484cfa46eeeb73c71edee823e4812f9e2e1"	4.1549736829070815e+23
5.	"0xd24400ae8bfebb18ca49be86258a3c749cf46853"	2.2700012958e+23
6.	"0x5b76fbe76325b970dbfac763d5224ef999af9e86"	7.873327492788825e+22
7.	"0xdd3e4522bdd3ec68bc5ff272bf2c64b9957d9563"	5.7901756850756706e+22
8.	"0x005864ea59b094db9ed88c05ffba3d3a3410592b"	3.7199e+22
9.	"0x4739928c37159f55689981b10524a62397a65d77"	3.023899895e+22
10.	"0xb8326d2827b4cf33247c4512b72382f4c1190710"	2.4572e+22

Conclusion: From the results obtained, the wash trader with highest volume of self-trades is with the address **0x02459d2ea9a008342d8685dae79d213f14a87d43** and with the value **1.9548531332493176e+25**.

Miscellaneous Analysis

Gas Guzzlers: For any transaction on Ethereum a user must supply gas. How has gas price changed over time? Have contracts become more complicated, requiring more gas, or less so? How does this correlate with your results seen within Part B. To obtain these marks you should provide a graph showing how gas price has changed over time, a graph showing how gas_used for contract transactions has changed over time and identify if the most popular contracts use more or less than the average gas_used. (15/40)

Input file: transactions.csv

Source code attached: gg.py
PartD_gas_price.ipynb
PartD_gas_used.ipynb

Explanation:

This part has two tasks, one is to get the average gas price change over time and the other one is to get the average gas used over time. For these tasks, transactions.csv and contracts.csv datasets are taken into account.

To get the average gas price change over time, the date is mapped as the key and the gas price and the count is mapped as the value from the transactions.csv dataset. Then the mapped function is reduced using reduceByKey function to get the total gas price and total count. In order to get the average gas price, the total gas price is divided by the total count and it is mapped to the date to get the average gas price change each month. The output is saved in “avg_gas.txt” file. Then the data is plotted using google colab (PartD_gas_price.ipynb).

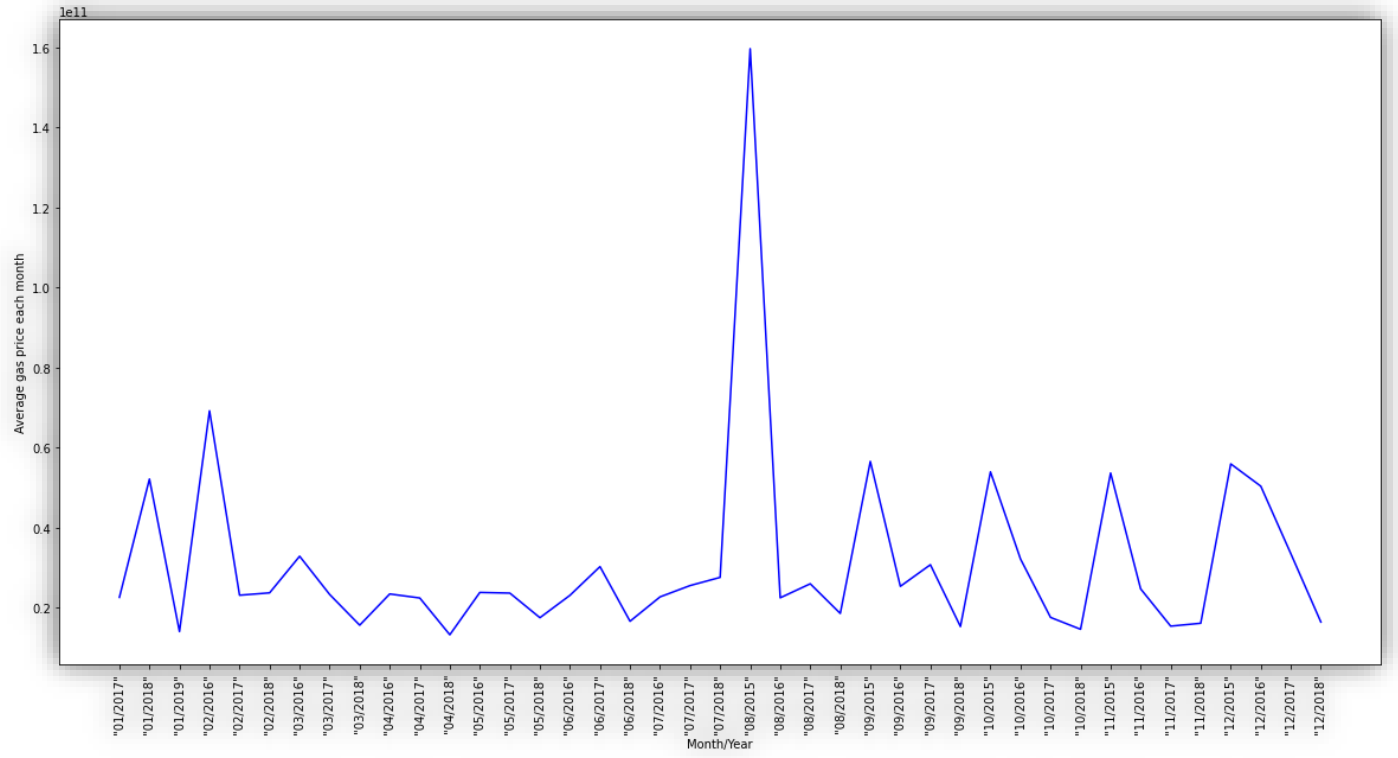
To get the average gas used over time, the to_address is mapped as the key and the date and the gas is mapped as the value from transactions.csv and the address is mapped as key and count as the value from contract.csv. Both the datasets are joined using .join() function. After joining, the date is mapped as key and the gas used and count as the value. Then the mapped function is reduced using reduceByKey function to get the total gas used and total count. In order to get the average gas price, the total gas used is divided by the total count and it is mapped to the date to get the average gas used each month. The output is saved in “gasused.txt” file. Then the data is plotted using google colab (PartD_gas_used.ipynb).

Execution Command: ccc create spark partc.py -d -s

Output file: avg_gasprice.txt
avg_gasused.txt

Output:

Plot: Average Gas Price vs Time



Plot: Average Gas Used vs Time