Coursework : Analysis of Ethereum Transactions & Smart Contracts

Big Data Processing : ECS765P

Natisha Mallick : 220867092

PART A : TIME ANALYSIS

Task1: Create a bar plot showing the no. of transactions occurring every month between start and end of the dataset.

Input file : transactions.csv
Code file: partatransactions.py

To obtain the time and overall quantity of transactions per month, the map function is used to retrieve the transaction count per month by mapping the data and the transaction count from the transactions.csv dataset. The reduceByKey function is applied to calculate the total number of transactions for each month.

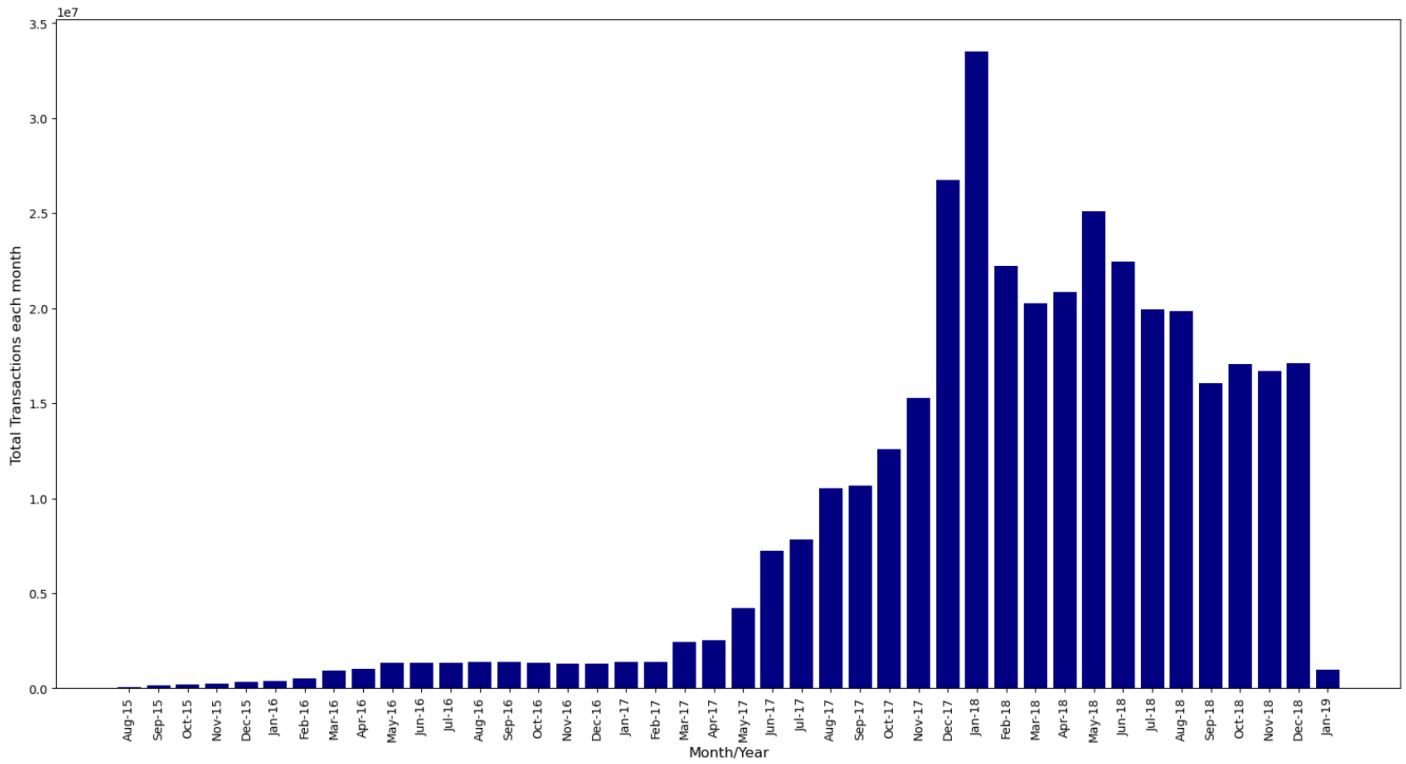
In the terminal the following code is used to run the partatransactions.py file for the output using the command-

ccc create spark partatransactions.py -s -e 10

The result is saved in the following way-
Output folder: parta_transactions_total
Text file: transaction_total.txt

Next the .txt file is converted into csv and a simple bar chart has been plotted using matplotlib.

txt to csv file: ConvertingTxt2CSV.ipynb | transactions_total.csv
Visualization file: Viz_PartA_transactions.ipynb



Task2: Create a bar plot showing the average value of transaction in each month between the start and end of the dataset.

Input file: transactions.csv
Code file: partaavgval.py

To obtain the time and average number of transactions per month, the map function is used to retrieve the values and transaction count per month by mapping the data, value and count from the transactions.csv dataset. The reduceByKey function is then applied to calculate the total number of values and total number of transactions for each month. These values are then mapped again with the data and total value, which is divided by the total number of transactions to obtain the average value of transactions.

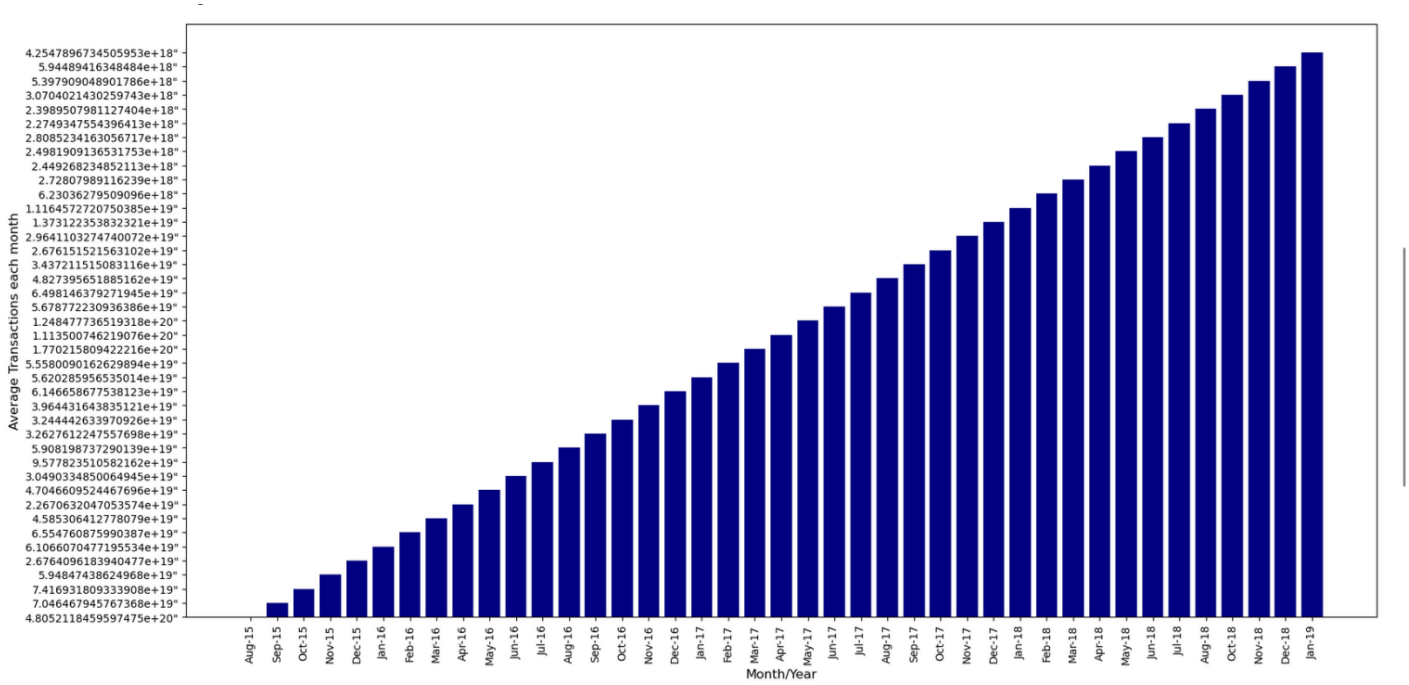
In the terminal the following command is run for the output-

ccc create spark partaavgval.py -s -e 10

The result is saved in the following way-
Output folder: parta_avgvalues
Text file: average_values.txt

Next the .txt file is converted into csv and a simple bar chart has been plotted using matplotlib.

txt to csv file: ConvertingTxt2CSV.ipynb | new_average_values.csv
Visualization file: Viz_PartA_AvgValues.ipynb



PART B : Top Ten Most Popular Services

Task: 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
Code file: top10popular.py

For this task both contracts & transactions dataset is used. Using the map function the to_address and value is taken from transactions dataset and the address & count from contacts dataset.
Using the .join() function both the datasets are joined with the to_address and address. Again the addresses & the values are mapped and using the takeOrdered function the top 10 smart contracts are found.

In the terminal the following command is run for the output-

ccc create spark top10popular.py -s -e 10

The result is saved in the following way-
Output folder: partboutput
Text file: top10_smart_contracts.txt

Next the .txt file is converted into csv and the top 10 smart contracts are displayed.

txt to csv file: ConvertingTxt2CSV.ipynb | top10_smart_contracts.csv

	Address	Ether Value
1	0xaa1a6e3e6ef2006…	8415536369994176…
2	0x7727e5113d1d161…	4562712851291534…
3	0x209c4784ab1e818…	4255298913641319…
4	0xbfc39b6f805a9e40…	2110419513809366…
5	0xe94b04a0fed112f3…	1554307763526374…
6	0xabbb6bebfa05aa1…	1071948594562894…
7	0x341e790174e3a4d…	8379000751917755…
8	0x58ae42a38d6b33a…	2902709187105736…
9	0xc7c7f6660102e9a1…	1238086114520042…
10	0xe28e72fcf78647ad…	1172426432515823…

PART C : Top Ten Most Active Miners

Task: 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 like 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
Code file: activeminers.py

This task uses the blocks.csv file and applies the map function to the miner and size parameters. The reduceByKey function is then used to obtain the block size. Using the takeOrdered function the top 10 miners are found.

In the terminal the following command is run for the output-

ccc create spark activeminers.py -s -e 10

The result is saved in the following way-
Output folder: partcoutput
Text file: top10_miners.txt

Next the .txt file is converted into csv and the top 10 miners are displayed.

txt to csv file: ConvertingTxt2CSV.ipynb | top10_miners.csv

	Miner	Size
1	0xea674fdde714fd97...	17453393724
2	0x829bd824b016326...	12310472526
3	0x5a0b54d5dc17e0a...	8825710065
4	0x52bc44d5378309e...	8451574409
5	0xb2930b35844a230...	6614130661
6	0x2a65aca4d5fc5b5c...	3173096011
7	0xf3b9d2c81f2b24b0f...	1152847020
8	0x4bb96091ee9d802...	1134151226
9	0x1e9939daaad6924...	1080436358
10	0x61c808d82a3ac53...	692942577

PART D : SCAM ANALYSIS

Task 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.

Most Lucrative Scam –

Input file: transactions.csv & scams.csv
Code file: popularscam.py

To identify the most profitable types of scams- both scams.csv and transactions.csv are used. From the scams.csv, index, ID and category are extracted, while from transactions.csv – the map function is used to map the address and ether value. The .join() function is used to join both the datasets based on addresses. Next, the reduceByKey function is applied to aggregate the total ether profits for each scam type- with the ID and scam type serving as the key and total ether profit as the value. Using the takeOrdered function- the top 15 most lucrative scams are found.

The result is saved in the following way-
Output folder: partdpopularscam_lucrative
Text file: most_lucrative_scams.txt

Next the .txt file is converted into csv to get the result.
txt to csv file: ConvertingTxt2CSV.ipynb | mostlucrativescams.csv

The most lucrative form of scam is Scamming with ID 5622 and a score of 1.670908358807289e+22 as can be seen from the below table and the output CSV file – mostlucrativescams.csv

	ID	Type	Score
1	5622	Scamming	1.670908358807289e+22
2	2135	Phishing	6.583972305381555e+21
3	90	Phishing	5.972589629102424e+21
4	2258	Phishing	3.46280752470374e+21
5	2137	Phishing	3.389914242537183e+21
6	2132	Scamming	2.428074787748576e+21
7	88	Phishing	2.067750892013526e+21
8	2358	Scamming	1.8351766714814877e+21
9	2556	Phishing	1.8030465742641803e+21
10	1200	Phishing	1.6305774191330895e+21
11	2181	Phishing	1.1639041282770013e+21
12	41	Fake ICO	1.1513030257909171e+21
13	5820	Scamming	1.1339734671862094e+21
14	86	Phishing	8.944561496957758e+20
15	2193	Phishing	8.827100174717206e+20

Ether vs Time -

Input file: scams.json, transactions.csv
Code file: ethervstime.py

The scams.json file is loaded into the s3 bucket. Json.loads() function is used to convert the json strings into python dictionaries. Using the map function the values are extracted with the result key. .flatmap() function is used to flat the address values in the tuples into a list of tuples containing individual addressed associated with its associated id, status and category. The map function is used again to extract to_address and values of the transactions. The .join() function is used to join on the basis of to_address resulting in tuples containing id, status, category and value of each transaction associated with the known scam address. reduceByKey is used to aggregate the total transaction value for each combination of category and month/year.

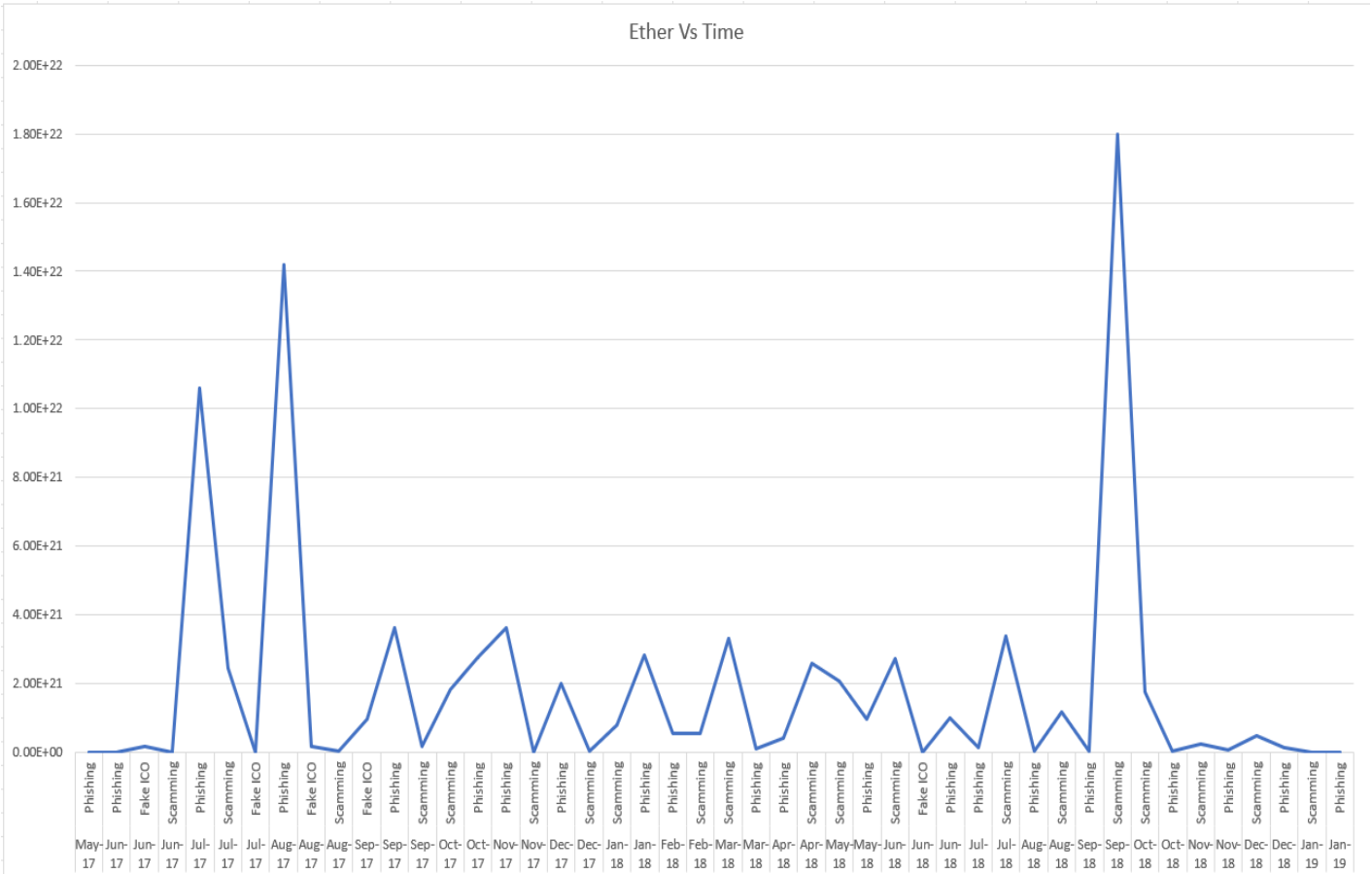
In the terminal the following command is run for the output-

ccc create spark ethervstime.py -s -e 10

The result is saved in the following way-
Output folder: partdpopularscam_ethervstime
Text file: ethervstime.txt

Next the .txt file is converted into csv to get the result.

txt to csv file: ConvertingTxt2CSV.ipynb
Visualization: ethervstime.csv is used in excel for the change over time visualization.



Observation: As seen in the above visualization it can be observed that from July 2017 till January 2018 Phishing was the most popular form of scam- with its highest peak in August 2017, although Scamming became the most popular form of Scam July 2018 onwards with its peak in September 2018.

Task 2: Wash Trading: Which addresses are involved in wash trading? Which trader has the highest volume of wash trades? How certain are you of your result?

Input file: transactions.csv
Code file: washtrading.py

To identify the users involved in self-trading- the from_address and to_address from transactions.csv are examined. A new dataframe is created containing from_address, to_address, value and date.
With pyspark.sql.functions library, the data frame is filtered to retain only those rows where the from_address and to_address values match. These are then used as the key along with the ether value serving as the value. These key-value pairs are then reduced by the reduceByKey function to find the total ether value received by each address through self-trading. Using TakeOrdered function the top 10 addresses engaged in self-trading are found.

In the terminal the following command is run for the output-

ccc create spark washtrading.py -s -e 10

The result is saved in the following way-
Output folder: partdwashtrading
Text file: top10_washtrade.txt

Next the .txt file is converted into csv to get the result.

txt to csv file: ConvertingTxt2CSV.ipynb | top10_washtrade.csv

	FromAddress_ToAddress	Ether_Value
1	['0x02459d2ea9a008342d8685dae79d213f14a87d43', '0x02459d2ea9a008342d8685dae79d213f14a87d43']	1.9548531332493185e+25
2	['0x32362fbfff69b9d31f3aae04faa56f0edee94b1d', '0x32362fbfff69b9d31f3aae04faa56f0edee94b1d']	5.295490520134209e+24
3	['0x0c5437b0b6906321cca17af681d59baf60afe7d6', '0x0c5437b0b6906321cca17af681d59baf60afe7d6']	2.3771525723546656e+24
4	['0xdb6fd484cfa46eeeb73c71edee823e4812f9e2e1', '0xdb6fd484cfa46eeeb73c71edee823e4812f9e2e1']	4.1549736829070815e+23
5	['0xd24400ae8bfebb18ca49be86258a3c749cf46853', '0xd24400ae8bfebb18ca49be86258a3c749cf46853']	2.2700012958e+23
6	['0x5b76fbe76325b970dbfac763d5224ef999af9e86', '0x5b76fbe76325b970dbfac763d5224ef999af9e86']	7.873327492788825e+22
7	['0xdd3e4522bdd3ec68bc5ff272bf2c64b9957d9563', '0xdd3e4522bdd3ec68bc5ff272bf2c64b9957d9563']	5.790175685075673e+22
8	['0x005864ea59b094db9ed88c05ffba3d3a3410592b', '0x005864ea59b094db9ed88c05ffba3d3a3410592b']	3.7199e+22
9	['0x4739928c37159f55689981b10524a62397a65d77', '0x4739928c37159f55689981b10524a62397a65d77']	3.023899895e+22
10	['0xb8326d2827b4cf33247c4512b72382f4c1190710', '0xb8326d2827b4cf33247c4512b72382f4c1190710']	2.4572e+22

The trader with highest volume of wash trades is from address -
0x02459d2ea9a008342d8685dae79d213f14a87d43 with ether value at 1.9548531332493185e+25

As can be seen from the address column in the table- it is pretty evident that the wash-trades between 0x02459d2ea9a008342d8685dae79d213f14a87d4 and 0x02459d2ea9a008342d8685dae79d213f14a87d4 ie. with itself are the highest.

PART D : MISCELLANOUS ANALYSIS

Task 1: Gas Guzzlers: 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

Input file: transactions.csv
Code file: gasguzzler.py

a. Average Gas Price Per Month:

To determine the average change in gas prices over time, the map function is used to map the date as the key and the gas price & count as the value from the transactions.csv. Then the reduceByKey function is used to aggregate the total price and count. Finally, to calculate the average gas price, the total has price is divided by the total count and the result is mapped to the corresponding date to obtain the monthly average gas price change.

In the terminal the following command is run for the output-

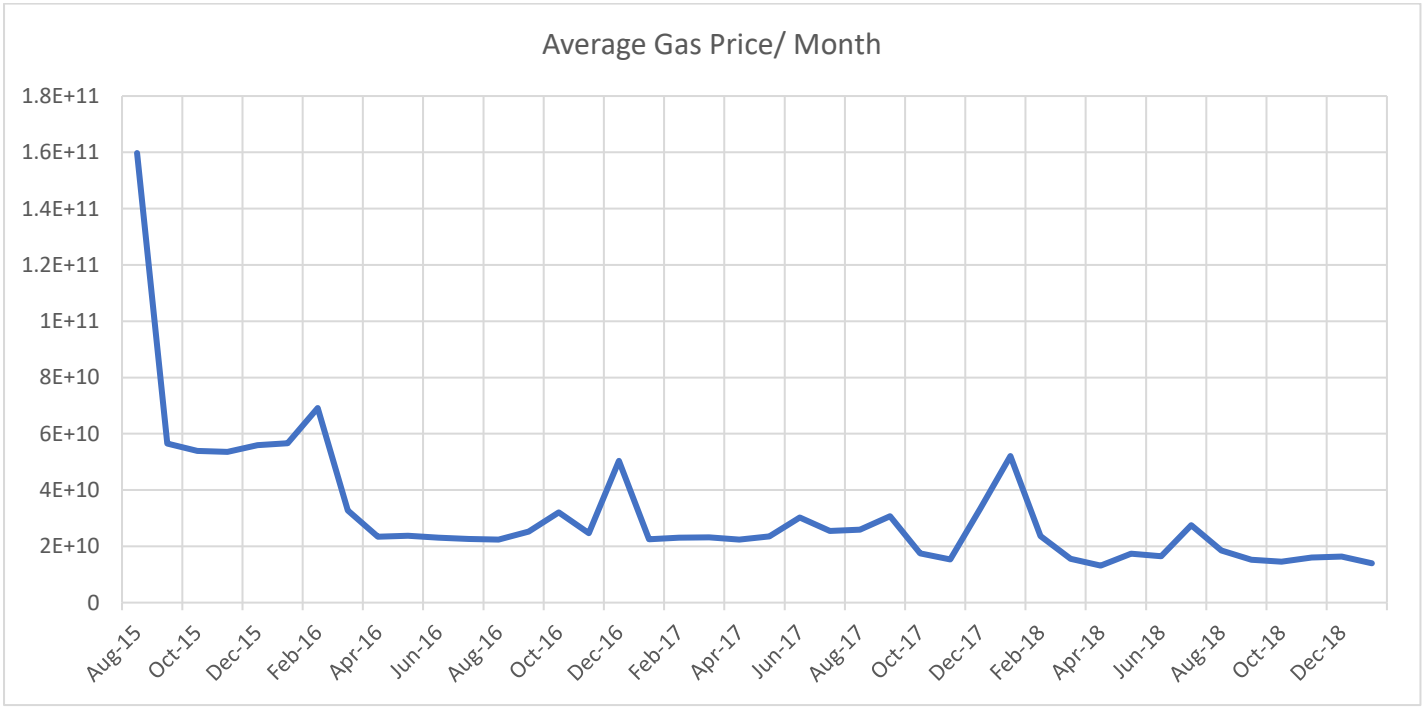
ccc create spark gasguzzler.py -s -e 10

The result is saved in the following way-
Output folder: partdgasguzzler
Text file: avg_gasprice.txt

Next the .txt file is converted into csv to get the result.

txt to csv file: ConvertingTxt2CSV.ipynb | avg_gasprice.csv

Visualization: avg_gasprice.csv is used in excel to create the visualization.



As can be seen from the above visualization, there has been a decline in the average gas price over the years. With the highest price in August 2015 and the lowest towards the end of December 2018.

b. Average Gas Used Per Month:

To determine the average gas used over time, the map function is used to map the to_address as the key and the date and gas as the value from the transactions.csv. From the contracts.csv, the address is mapped as key and count as the value. Using the .join() function both datasets are joined. Then the date from the joined dataset is mapped as the key and the gas usage & count as the value. reduceByKey function is used to aggregate the total gas usage and count. Finally, to calculate the average gas used per month, the total gas usage is divided by the total count and the result is mapped to the corresponding date to obtain the monthly gas average.

In the terminal the following command is run for the output-

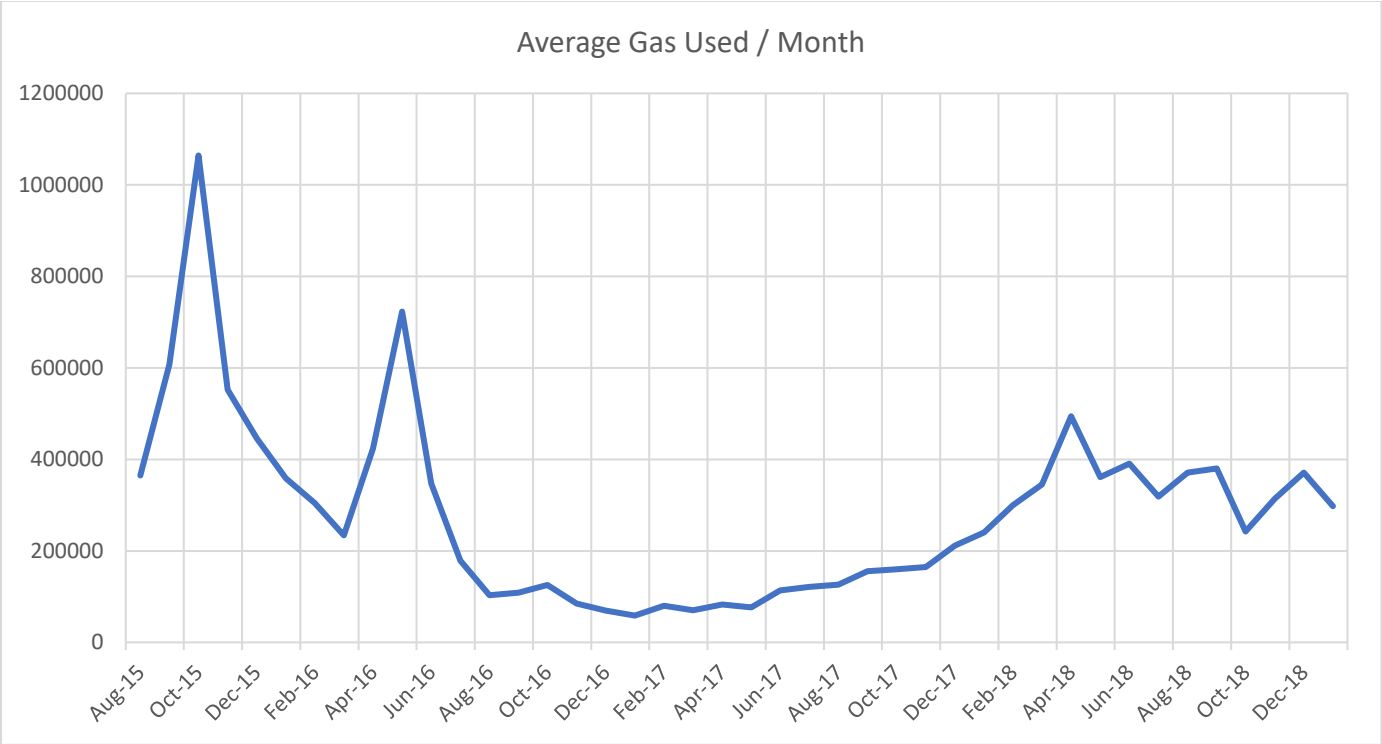
ccc create spark gasguzzler.py -s -e 10

The result is saved in the following way-
Output folder: partdgasguzzler
Text file: avg_gasused.txt

Next the .txt file is converted into csv to get the result.

txt to csv file: ConvertingTxt2CSV.ipynb | avg_gasused.csv

Visualization: avg_gasused.csv is used in excel to create the visualization.



As can be seen from the above visualization, the average gas used was at it's peak in October 2015 and started decline till April 2016. It peaked again in May 2016 but continued to decline over several months.
It only again started rising from June 2017 and saw a small peak again in May 2018 after which it is remained steady.

Task 2: Data Overhead: The blocks table contains a lot of information that may not strictly be necessary for a functioning cryptocurrency e.g. logs_bloom, sha3_uncles, transactions_root, state_root, receipts_root. Analyse how much space would be saved if these columns were removed.

Input file: Blocks.csv
Code file: overhead.py

The blocks.csv is read and the mean length of the columns logs_bloom, sha3_uncles, transactions_root, state_root and receipts_root is calculated, and the average length is converted into bytes from characters. This is used to calculate the space saved in each of these columns in bytes by subtracting the size of the column represented as text from size of column represented as bytes. The total space savings and the percentage of saved space is stored in data frames.

In the terminal the following command is run for the output-

ccc create spark overhead.py -s -e 10

The result is saved in the following way-
Output folder: overhead
Text file: overhead_data_file.txt , percentage_saved_file.txt

txt to csv file: ConvertingTxt2CSV.ipynb | overhead_data.csv , percentage_saved_file.csv

	column_name	size_bytes
1	logs_bloom	1792000256
2	sha3_uncles	224000032
3	transactions_root	224000032
4	state_root	224000032
5	receipts_root	224000032

	column_name	space_saved	percentage_saved
1	number	16944447.5	0.465805239
2	hash	224000032	6.157792303
3	parent_hash	224000032	6.157792303
4	nonce	56000008	1.539448076
5	sha3_uncles	224000032	6.157792303
6	logs_bloom	1792000256	49.26233842
7	transactions_root	224000032	6.157792303
8	state_root	224000032	6.157792303
9	receipts_root	224000032	6.157792303
10	miner	140000020	3.848620189
11	difficulty	45092015.5	1.239585832
12	total_difficulty	65700399	1.806113185
13	size	7078047	0.194576505
14	extra_data	114808596	3.156104409
15	gas_limit	17414528.5	0.478727831
16	gas_used	11477951	0.315530483
17	timestamp	27999999.5	0.769723914
18	transaction_count	-848591.5	-0.023327899
19	base_fee_per_gas	0	0