ECS765P - Big Data Processing

Report for Analysis of Ethereum Transactions and Smart Contracts

Neeraj Yadav

ID:220708603

<u>PART A. TIME ANALYSIS (25%)</u>

Bar plot for the number of transactions occurring every month between the start and end of the dataset.

Input file : transactions.csv

Attached source code : total.py, plota1.ipynb

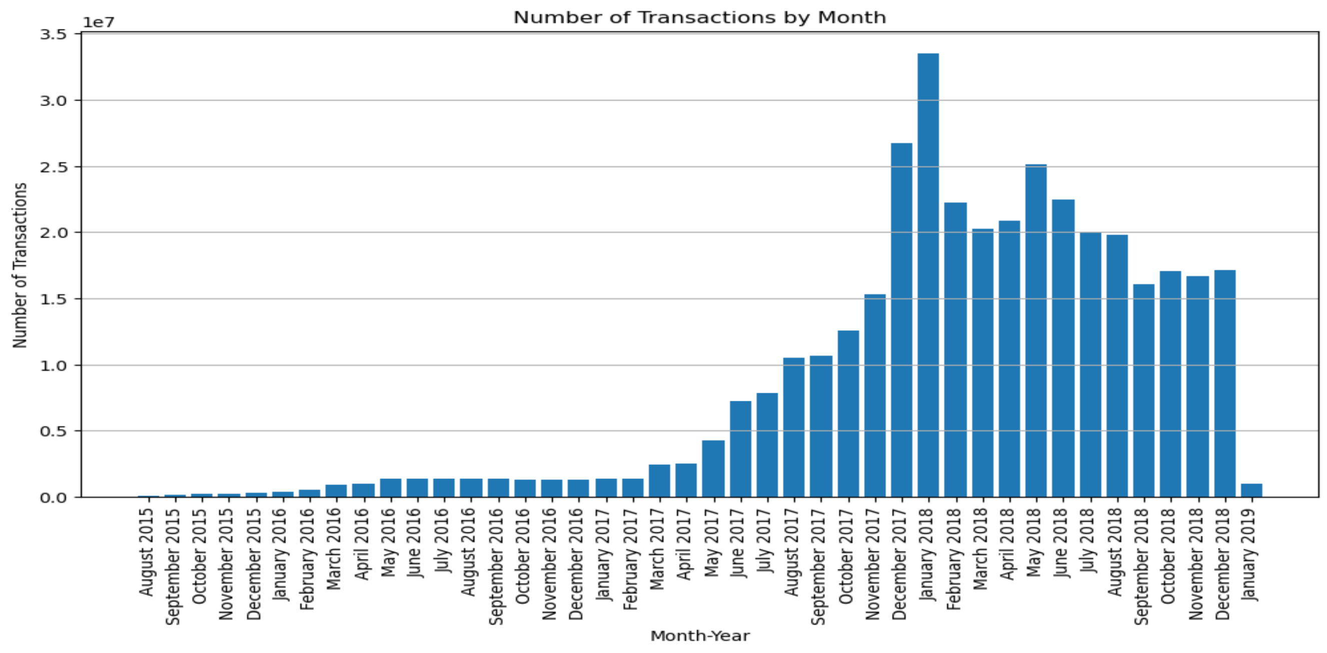
Commands used: ccc create spark total.py -d -s (run spark)

ccc method bucket cp -r bkt: (output copy)

Output file: total_transactions.txt

The script total.py uses SparkSession to read Ethereum transaction data from the CSV file in an S3 bucket, remove invalid transactions, compute the total number of transactions per month, and store the result as a JSON file in another S3 bucket. Additionally, it displays the result on the console. The code starts by creating a SparkSession and defining a function to verify the validity of a transaction line. It then reads environmental variables and configures the Hadoop configuration object for S3 access. After that, it uses the SparkContext's textFile method to read the transaction data from S3, remove invalid transactions using the is_valid_transaction function, and calculates the total number of transactions per month by mapping the transactions to (month/year, 1) pairs and reducing the pairs by key. The script then uses the boto3 S3 resource object to write the total number of transactions per month to S3 as a JSON file with a timestamp in the filename. It also prints the result to the console. Finally, the script ends by stopping the SparkSession.

Then python code in plota1.pynb uses the matplotlib library to create a bar plot of the number of transactions by month and year present in the total_transactions.txt file. The data is sorted by month and year, and the month and year are extracted from the date string. The resulting bar plot is then displayed.



Bar plot showing the average value of transaction in each month between the start and end of the dataset

Attached source code : average.py, plota2.ipynb

Commands used: ccc create spark average.py -d -s (spark run)

ccc method bucket cp -r bkt: (output copy)

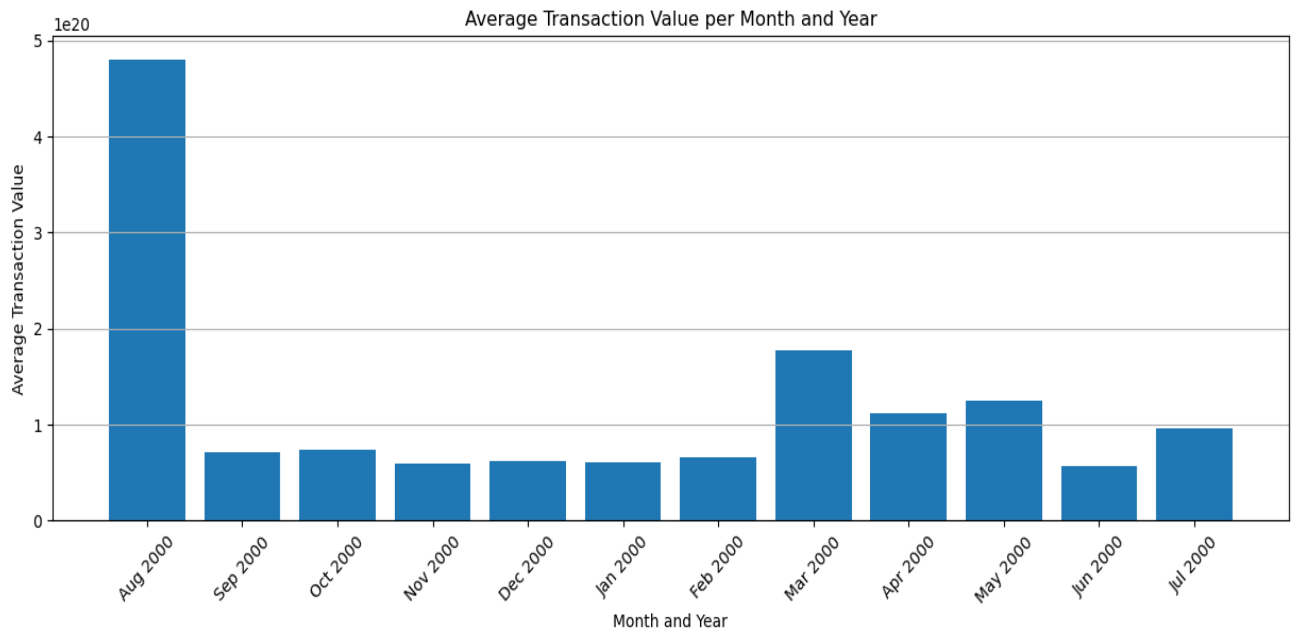
Output file: total_transactions.txt

The script average.py loads transaction data from S3 bucket, processes it to calculate the average transaction value per month, and saves the result to another S3 bucket. The code works as :

1. First, the code imports necessary modules including boto3 (a Python library for interacting with Amazon Web Services), pyspark.sql, datetime, and time.
2. Next, the code defines a function check_transaction(line) to check if a line in the transaction data is valid. The function checks if the line has 15 fields and if the 7th and 11th fields can be converted to float and int, respectively.
3. Then, the code defines a function map_transaction(line) to transform a valid line into a (month, (value, 1)) tuple. The function extracts the value and timestamp from the line, converts the timestamp to a month (in the format of "mm/YYYY"), and returns the tuple.

4. After that, the code defines a function reduce_transaction(t1, t2) to reduce tuples by month. The function takes two tuples of the form (value, count) and returns a new tuple with the sum of values and counts.
5. The code then creates a Spark session, and reads S3 bucket details from environment variables.
6. The code configures Hadoop to access S3 by setting various properties to the Hadoop configuration object.
7. The code loads transaction data from S3 as an RDD (Resilient Distributed Dataset) using Spark's textFile() method.
8. The code filters out invalid transactions using the check_transaction() function, and maps valid transactions to (month, (value, 1)) tuples using the map_transaction() function.
9. The code reduces tuples by month using the reduce_transaction() function, and calculates the average transaction value per month using Spark's mapValues() method and a lambda function.
10. The code converts the RDD to a list of strings, and saves the result to S3 using the boto3 client's put_object() method. The file name includes the current date and time.
11. Finally, the code stops the Spark session.

Finally, the plota2.ipynb uses the data in total_transactions.txt containing dates and transaction values and imports the matplotlib.pyplot library. A loop is then used to divide the data into months, years, and values. The average transaction values per month and year are kept in a dictionary. By dividing the total transaction values by the number of transactions for each month and year, the average transaction values are determined and then arranged in ascending order. The plt.bar() method is then used to plot the sorted months and values as a bar chart, with the month and year on the x-axis and the average transaction value on the y-axis. The plt.show() function is used to display the plot after saving it as a file.



PART B. TOP TEN MOST POPULAR SERVICES (25%)

Input file : transactions.csv , contracts.csv

Attached source code : topten.py

Commands used: ccc create spark average.py -d -s (spark run)

ccc method bucket cp -r bkt: (output copy)

Output file: top10_smart_contracts2.txt

The script topten.py reads data from the two CSV files stored in an S3 bucket, filters and processes the data, and then saves the result in another S3 bucket. The script uses the pyspark.sql module to create a SparkSession, which is then used to read the CSV files and perform various data transformations. Specifically, it filters out rows where certain columns have missing values, selects certain columns from the transactions and contracts dataframes, joins them on the "address" column, groups the resulting dataframe by "address" and computes the sum of "value" and the count of "contractAddress" for each group. It then filters out rows where "total_value" or "contract_count" is null, sorts the resulting dataframe in descending order by "total_value", takes the top 10 rows, and saves them to an S3 bucket in JSON format.The script also sets certain Hadoop configuration properties to enable Spark to read data from and write data to S3. Finally, it uses the Boto3 library to create an S3 client and save the results to the specified S3 bucket. The output is presented in the top10_smart_contracts2.txt and are sorted here in the table.

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%)

Input file : blocks.csv

Attached source code : active.py

Commands used: ccc create spark active.py -d -s(spark run)

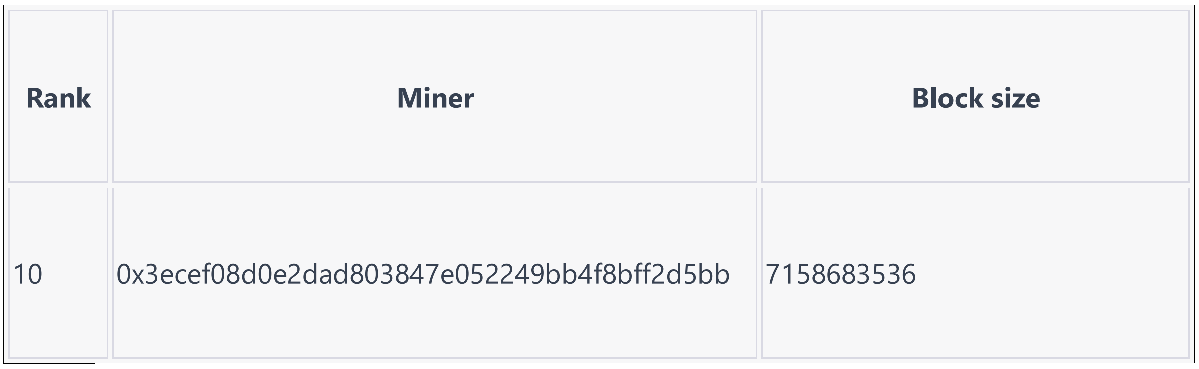
ccc method bucket cp -r bkt: (output copy)

Output file: active_miners.txt

The script active.py performs the following tasks to find the top 10 miners of the Ethereum blockchain based on the size of their blocks.:

1. Create a Spark session
2. Define two functions: **is_valid_block_data** and **get_block_features** to filter out irrelevant data and extract relevant features from the Ethereum blockchain data.
3. Set up access credentials and endpoint URLs for connecting to an Amazon S3 bucket where the Ethereum blockchain data is stored.
4. Read the blockchain data from the S3 bucket using the **textFile** method.
5. Filter out irrelevant data using the **is_valid_block_data** function and **filter** method.
6. Extract relevant features using the **get_block_features** function and **map** method.
7. Reduce the data by key (miner) and sum block sizes using the **reduceByKey** method.
8. Get the top 10 miners by block size using the **takeOrdered** method.
9. Write the results to an S3 bucket in JSON format using the **json.dumps** method and **put** method. The results are shown in the active_miners.txt and are sorted here in the table.

Rank	Miner	Block size
1	0xea674fdde714fd979de3edf0f56aa9716b898ec8	115564318921
2	0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c	65704646587
3	0x829bd824b016326a401d083b33d092293333a830	56642548682
4	0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5	25793793591
5	0x1ad91ee08f21be3de0ba2ba6918e714da6b45836	17465705776
6	0x00192fb10df37c9fb26829eb2cc623cd1bf599e8	10206344028
7	0x04668ec2f57cc15c381b461b9fedab5d451c8f7f	10071546636
8	0xb2930b35844a230f00e51431acae96fe543a0347	9176722960
9	0x7f101fe45e6649a6fb8f3f8b43ed03d353f2b90c	7825764182



Rank	Miner	Block size
10	0x3ecef08d0e2dad803847e052249bb4f8bff2d5bb	7158683536

PART D. DATA EXPLORATION (40%)

SCAM ANALYSIS

POPULAR SCAMS

Input file : transactions.csv , scams.json

Attached source code : jsontocsv.py, scams.py, ether_time.ipynb

Commands used: ccc create spark jsontocsv.py -d -s (spark run)

ccc create spark scams.py -d -s (spark run)

ccc method bucket cp -r bkt: (output copy)

Output file: scams.csv, lucrative_scams.txt, time_data.txt

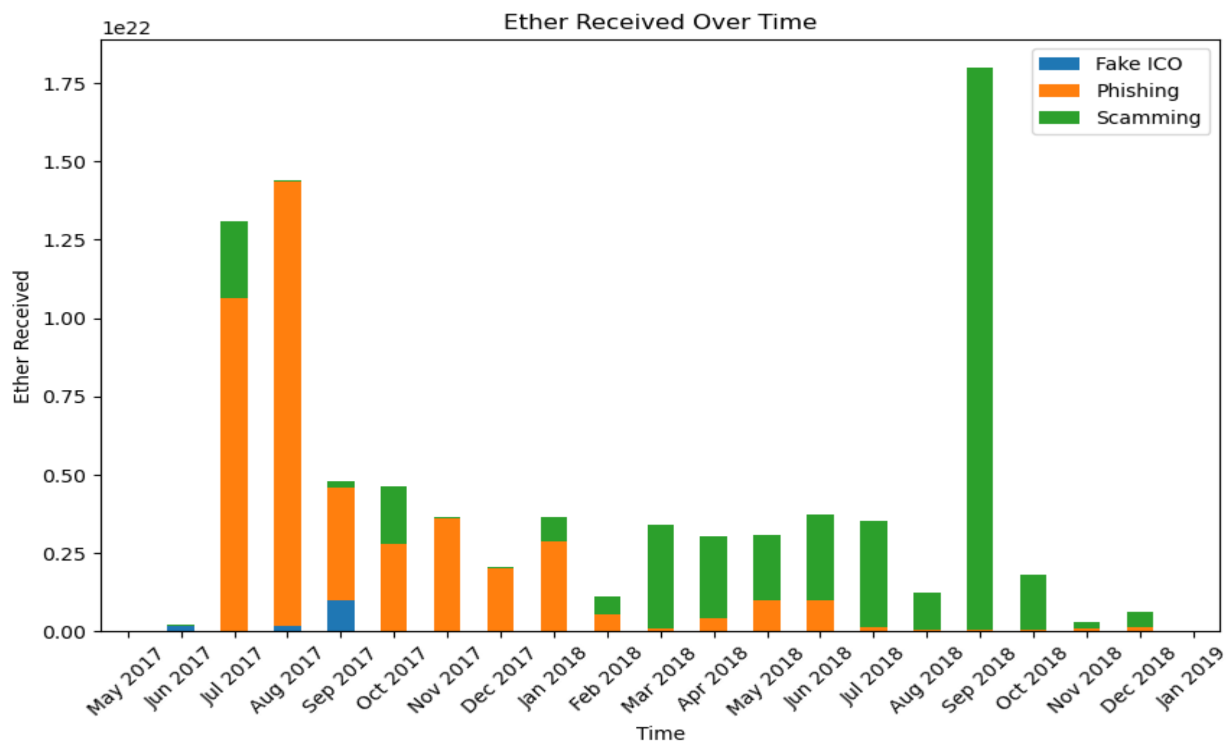
1. Scams.json was converted to a csv file called scams.csv, which contains various parameters such as identity, name, url, coin, category, subcategory, index, and status. The conversion process was done successfully using the convert.py file, and the resulting file was uploaded to the data repository bucket.

2. The goal was to obtain the IDs of the most lucrative scams. This was done by reading the scams.csv and transactions.csv files, mapping the necessary fields such as index, Id, category, address, and ether value, and then joining the datasets based on

addresses using .join() function. The resulting dataset was then reduced by reduceByKey function to obtain the total ether profited, which was then mapped with the Id and scam type as key and the total ether profited as the value. Finally, the takeOrdered function was used to get the top 15 most lucrative scams, which were stored in the “most_lucrative_scams.txt” file and are sorted in the table .

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

3. To visualize the change in total ether received over time, the transactions.csv and scams.csv files were used. The necessary fields such as index, category, address, date, and value were mapped from these files, and the datasets were joined using .join() function. The resulting dataset was then reduced by reduceByKey function to obtain the total ether received for each month. The output was saved in “time_data.txt” file, and the data was plotted using matplotlib in python.



Miscellaneous Analysis

GAS GUZZLERS

Input file : transactions.csv

Attached source code : gas_guz.py, av_gasused.ipynb, av_gaspriced.ipynb

Commands used: ccc create spark gas_guz.py -d -s (spark run)

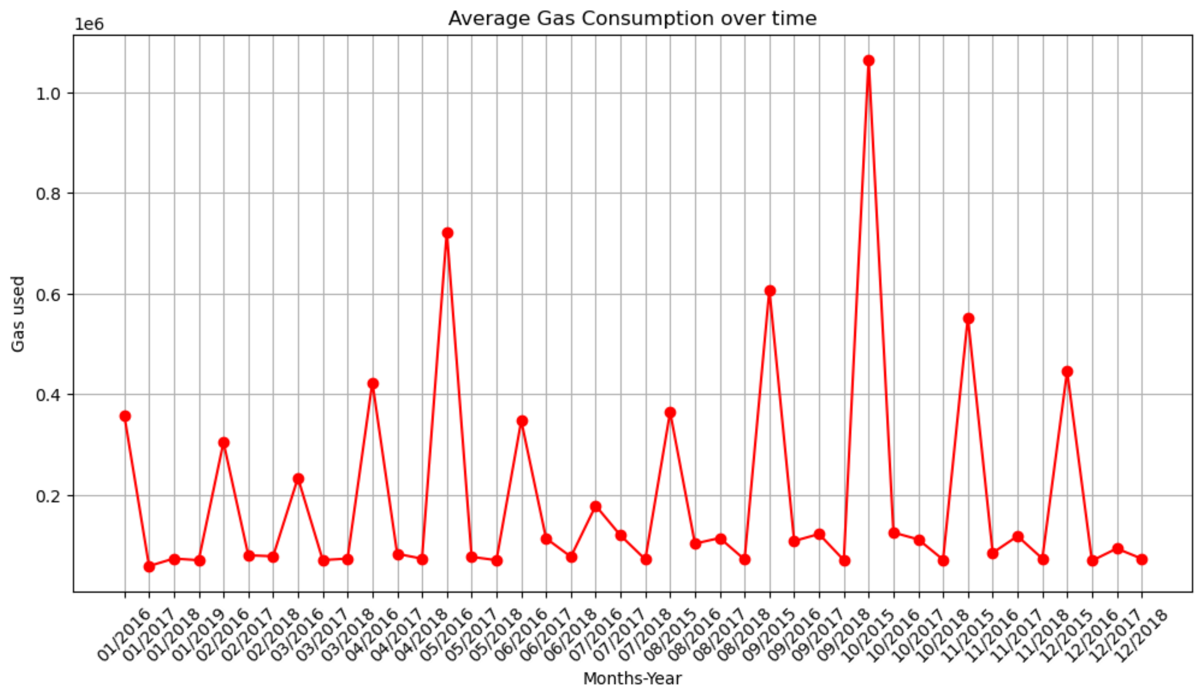
ccc method bucket cp -r bkt: (output copy)

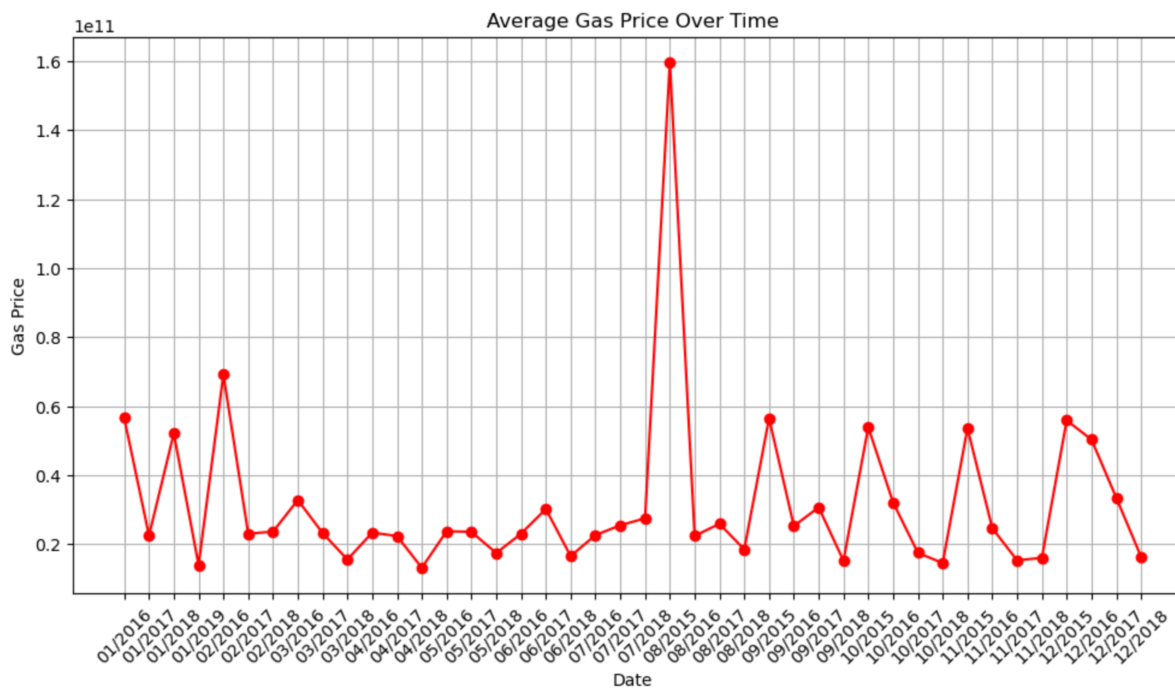
Output file: av_gasused.txt, av_gasprice.txt

In summary, the task involves two parts:

1. To calculate the average gas price change over time, transactions.csv and contracts.csv datasets are used. The date is mapped as the key and the gas price and count are mapped as the value from transactions.csv. Then the mapped function is reduced using reduceByKey function to get the total gas price and total count. 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 a file called “avg_gas.txt” and the data is plotted using matplotlib in av_gasused.ipynb

2. To calculate the average gas used over time, transactions.csv and contracts.csv datasets are used. The to_address is mapped as the key and the date and the gas are mapped as the value from transactions.csv, and the address is mapped as key and count as the value from contract.csv. Both 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. 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 a file called “gasused.txt”. The data is plotted using matplotlib in av_gaspriced.ipynb.





Based on the given information, it is clear that the average gas price on Ethereum is not constant and varies over time. The data shows that there are significant fluctuations in the gas price from year to year, with some periods of high prices and some periods of low prices.

It is also noteworthy that there is an overall downward trend in the average gas price from 2015 to 2018. This suggests that, on average, the cost of executing transactions on the Ethereum network has decreased over time. However, it is important to note that there are still periods of higher gas prices within this overall downward trend.

The data also highlights some interesting patterns within each year. For example, in 2016, the gas price fluctuated within a broad range, while in 2017, the range was narrower, but with two peaks around June and September. In 2018, there was a gradual decrease in gas prices throughout the year, with the lowest point being reached in December.

Similarly the average gas consumption graph indicates that the average gas used on the Ethereum network fluctuates over time, without showing a clear overall trend. The data suggests that gas used values have varied significantly within each year. In 2015, there was a spike in gas used values in October, followed by a decrease toward the end of the year. In 2016, the range of gas used values was narrower, with the highest value in May and the lowest in August. In 2017, gas used values increased almost linearly throughout the year, with the highest value in September and the lowest in January. In 2018, gas used values had a general upward trend from January to September, with the highest value in April and the lowest in October.

Overall, the data suggests that the cost of executing transactions on the Ethereum network is not fixed and can be influenced by a variety of factors, such as network congestion and demand for gas.