ECS765P - Big Data Processing - 2022/23
Analysis of Ethereum Transactions and Smart Contracts

PART A. Time Analysis

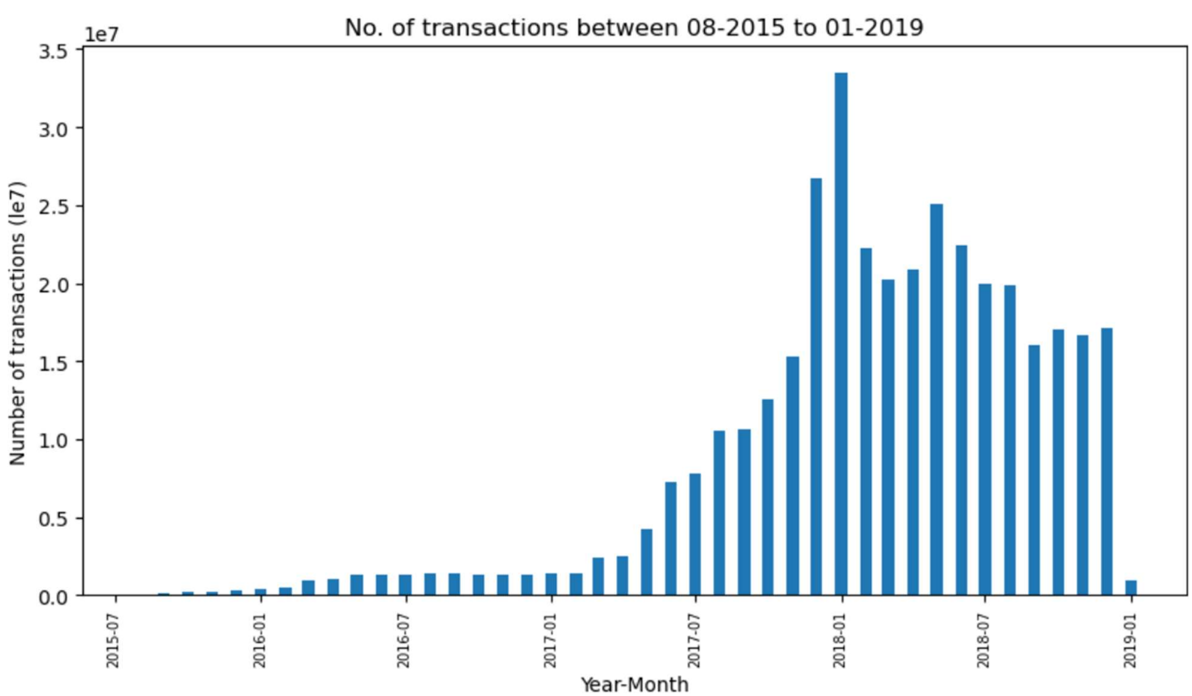
i. Create a bar plot showing the number of transactions occurring every
ii. Create a bar plot showing the average value of transactions in each month.

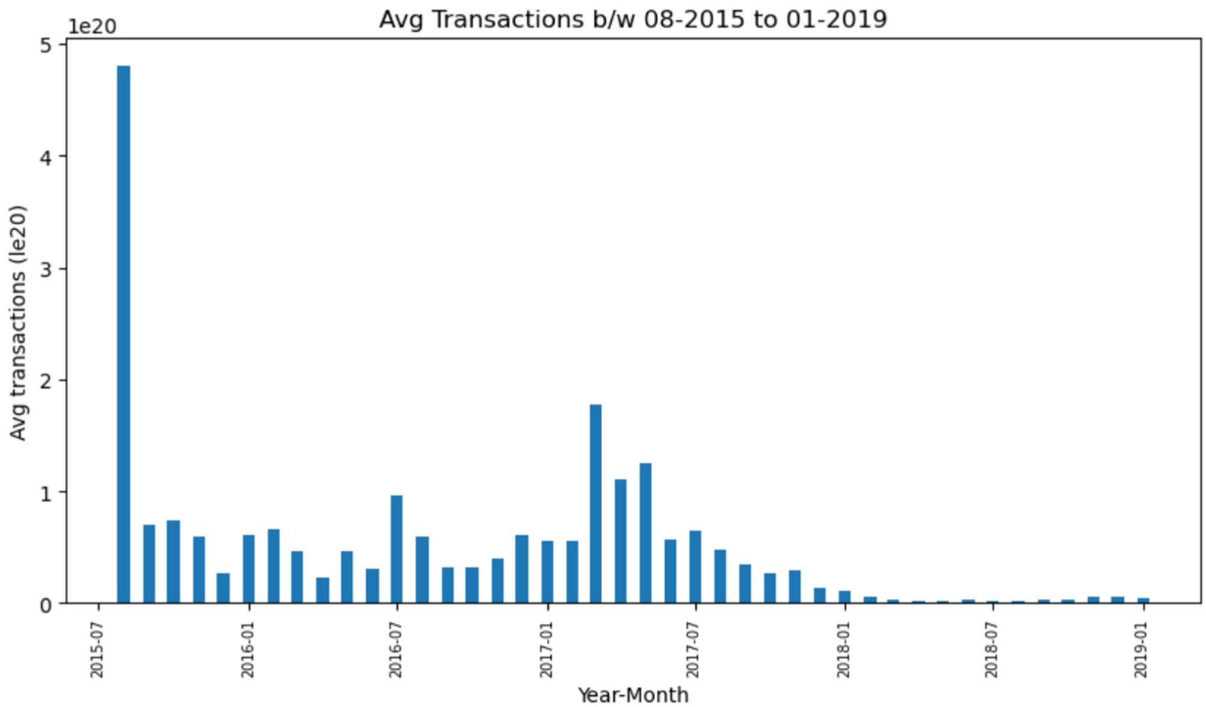
Code explanation:
For the solution to this task, we first load the transactions.csv file and pass it through good_lines filter and filter out all the records which either do not contain all 15 fields or do not contain an integer value for the block number.
Number of transactions every month- The map function is used to get the date and transaction values for that date with the block number as the key. Count each month's total number of transactions using map(date,1) and reducebykey().
Avg. value of transactions each month- Map transaction values to the corresponding month. Then use reduceByKey() to aggerate the total amount for each month. Join the above to rdds to pair the total monthly transaction value and the number of transactions with a common date key. Then calculate the average for each month using map().

Files: q1.py
q1_graphs.ipynb

Output: q1_count.csv
q1_avg.csv

Graphs:





PART B: Top Ten Most Popular Services

Evaluate the top 10 smart contracts by total Ether received.

Code explanation:

For the solution to this task, we first load the transactions.csv and contracts.csv files and pass them through trans_good_line and contract_good_line filters to filter out abnormal lines and headers (for contracts.csv we filter out the headers by checking if 'is_erc20' column value either 'True' or 'False').

Then we map 'to_address' and value from transaction and 'address' and count from contracts dataset. We get aggregate values for each address using reduceByKey on transaction rdd. Now both aggregated transaction and contract rdds are joined using .join() keeping address as the key and we find the top 10 contracts using .takeOrdered() and sorting the key in descending order.

Files: q2.py

q2_top10.csv (output)

Output:

	Address	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

Evaluate the top 10 miners by the size of the blocks mined.

Code explanation:

In this task, we first read the blocks.csv file and pass it through good_lines filter and filter out all the records which either do not contain all 19 fields or do not contain an integer value for the block number.

We then map the ‘miner’(key) and ‘size’ of the block(value) and aggregate the size using reduceByKey(). Then get the top 10 miner using .takeOrdered() with key sorted in descending order.

Files: q3.py

q3_top10.csv (output)

Output:

	Miner	Block size
1	Oxea674fdde714fd979de3edfOf56aa9716b898ec8	17453393724
2	0x829bd824b016326a401d083b33d092293333a830	12310472526
3	0x5aOb54d5dc17eOaadc383d2db43bOaOd3e029c4c	8825710065
4	0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5	8451574409
5	Oxb2930b35844a230fOOe51431acae96fe543a0347	6614130661
6	0x2a65aca4d5fc5b5c859090a6c34d164135398226	3173096011
7	Oxf3b9d2c81f2b24bOfaOacaaa865b7d9ced5fc2fb	1152847020
8	0x4bb96091ee9d802ed039c4d1a5f6216f90f81b01	1134151226
9	Oxie9939daaad6924ad004c2560e90804164900341	1080436358
10	0x61c808d82a3ac53231750dadc13c777b59310bd9	692942577

Part D. Data Exploration:

Scam Analysis:

Q1. Provide the id of the most lucrative scam

Code explanation:

In this task, we read the transactions.csv file and pass it through good_lines filter mentioned in Part A. We also need the scams.json file which we first read using sc.**textFile**() and then map it using **json.load(x).**

Next, we map transaction data to get columns (to_address, (value, timestamp))**.** To map data from the scams dataset we use **flatmap**() and for each record in scams[‘result’] we get the list of addresses associated with the scam, ID and category. We further map this rdd to go from ([add1, add2],(ID, category)) → (add1, ([add1,add2],ID, cat))**,** (add2,([add1, add2], ID, cat))**.** This way we have all the addresses mapped to their respective scam IDs.

These address keys would be further useful to join scams with transactions using **.join**(). We then create a profit_rdd which maps the ID column(key) from the join rdd to the value column. Next, we use **reduceByKey**() to sum up all the values for each ID.

Finally, we use **takeOrdered**(1,key=lambda x: -x[1]) to get the ID with the largest value as the most profitable scam.

Output: ID and Value of the most lucrative scam is: [(5622, 16709083588073530571339)]

File: q4scamid.py, q4ID.txt (output)

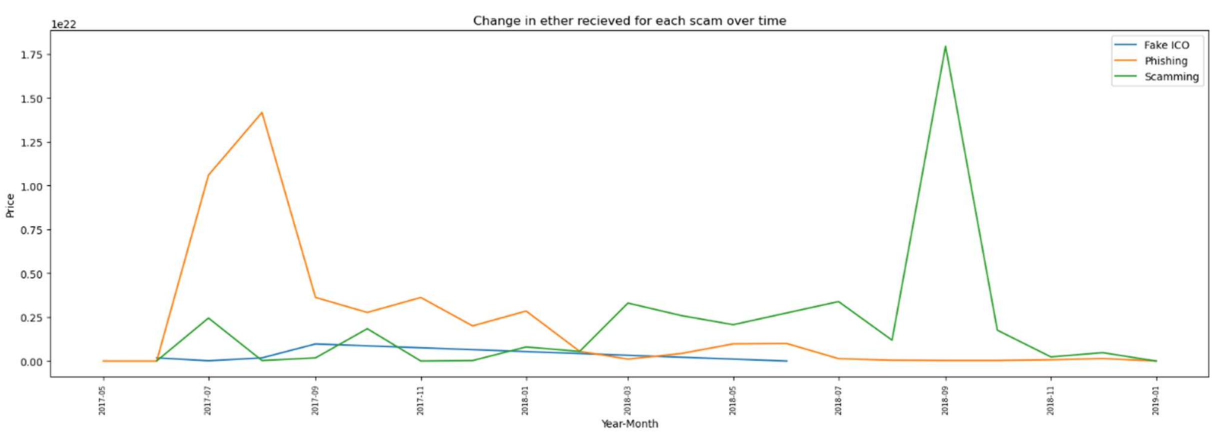
Q2. Provide a graph showing how the ether received has changed for each scam over time for the dataset.

Explanation:
For this part of the task, we follow the above same steps till joining the 2 rdds. Then we map the timestamp and category (key) to the value column $((ts, cat), value)$. Finally, we use **reduceByKey**() to sum up all the values for each category for every month of each year.
In the graph, we are plotting the change in the ether received for each category for every month of each year.

Output: q4_scam_graph.txt

File: q4scamsvstime.py
scams_graph.ipynb

Graph:



Data Overhead:

Q. Analyse how much space would be saved if logs_bloom, sha3_uncles, transactions_root, state_root, and receipts_root columns were removed.

Code explanation:
In this task, we first read the block.csv file and pass it through good_line function to filter out rows which do not have 19 fields and the first field is an integer which will eliminate the header row.
Next, we map all the space required by all unnecessary columns (logs_bloom, sha3_uncles, transactions_root, state_root, and receipts_root) from the filtered dataset to a common string as the key.
Space calculation: Since all the values of the mentioned columns are hex_strings where each character after the first two requires four bits so we calculate the length of each value subtract it by 2 (removes '0x') and then divide the length by 2 (1 char = 4 bits and 8 bits = 1 byte. Therefore, 2 characters = 1 byte) e.g. – $(len(\, x.split(',')[4]) - 2) \,/\, 2$

Output: [["Total unnecessary data consumed (Bytes): ", 2688000384.0]]

Files: q4do.py, q4_data.txt

Gas Guzzler:

Q1. provide a graph showing how gas price has changed over time.

Code explanation:

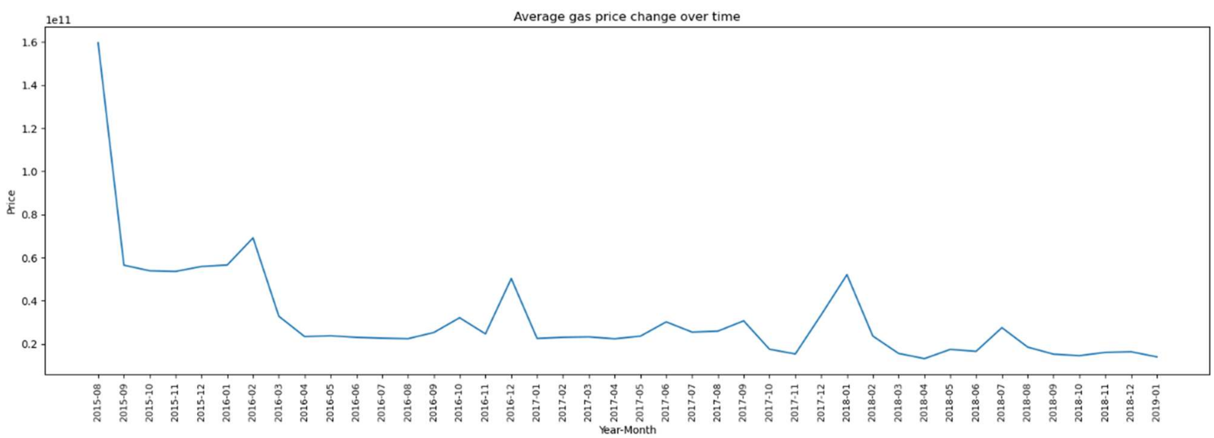
In this task, we use the transactions.csv and filter out the abnormal lines as mentioned in Part A. Then from the transaction dataset, we **map(block_timestamp, gas_price).**

To check gas price vs time data we need to find the average gas price for each month, for this, we get the count of transactions for each month by using **map(mm-yyyy, 1)** and aggregate it using **reduceByKey(mm-yyyy, monthly_count)**. Similarly, to get aggregate gas_price for each month **map(mm-yyyy, price)** and **reduceByKey(mm-yyyy, sum_price).** We then join time_count and time_price to get an rdd of the form **(mm-yyyy, (monthly_count, sum_price)**. Finally, we calculate average gas_price for each month by dividing sum_price by monthly_count and get **map(mm-yyyy, monthly_avg)**

Output file: q4gasprice.txt

Files: q4gasprice.py, AvgPriceGraph.ipynb

Graph:



Q2. Provide a graph showing how gas used for contract transactions has changed over time.

Code explanation:

In this task, we use the transactions.csv and contracts.csv files and filter out the abnormal lines as mentioned in Part B.

Then from the transaction dataset, we **map(to_address, (block_timestamp, gas))** and from the contract dataset, we **map(address, 1).**

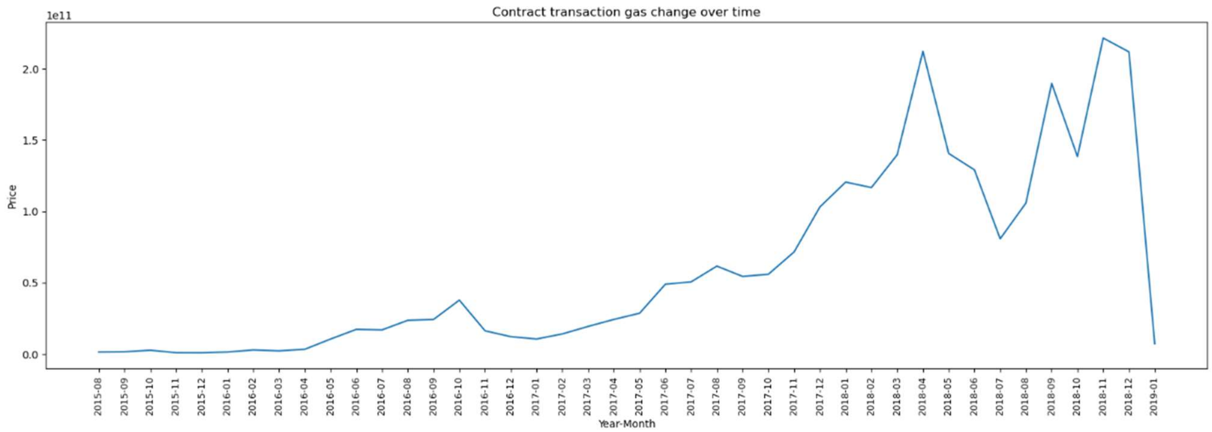
Next, we join these to datasets with the address column as the key to form **join_ds** (address, (1, (mm-yyyy, gas))).

Finally, in order to get the total monthly gas used by the contract transactions, we map the timestamp (key) from join_ds to gas i.e. (mm-yyyy, gas) and aggregate the gas used using reduceByKey(). We now have gas used for contract transactions for each month of each year.

Output: q4gasvstime.txt

Files: q4gasvstime.py
gasvstime.ipynb

Graph:



Q3. Identify if the most popular contracts use more or less than the average gas_used.

Code explanation:
For this task, we follow all the initial steps as mentioned in previous sub-question then we calculate the overall average gas used by all contracts by mapping a common string(key) to the gas used ('string', gas) by all the transaction addresses in **join_ds** from the previous subtask. Then aggregate it using **reduceByKey()** and get the transactions count using .count() on the above-mapped rdd. We calculate the average by using map for dividing aggregated gas used with the number of transactions.
Then to calculate the gas used by popular contracts, we load the top 10 contracts file generated in PART B and map it to create an rdd (address, value). Next, we join this pop_con rdd to the transaction_ds, followed by mapping addresses to their corresponding gas value. Then, we aggregate these values to get the total gas_used by each popular contract.

Overall average gas used: 231728.524
Gas used by popular contracts:

Address	Gas Used
'0x58ae42a38d6b33a1e31492b60465fa80da595755'	287066626
'0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444'	26790518214
'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3'	214899694204
'0xc7c7f6660102e9a1fee1390df5c76ea5a5572ed3'	553023050
'0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef'	74926981993
'0xbfc39b6f805a9e40e77291aff27aee3c96915bdd'	14481700000
'0xabbb6bebfa05aa13e908eaa492bd7a8343760477'	66970468113
'0x341e790174e3a4d35b65fdc067b6b5634a61caea'	8350972
'0xe28e72fcf78647adce1f1252f240bbfaebd63bcc'	89208660
'0x7727e5113d1d161373623e5f49fd568b4f543a9e'	46097278322

Observation:
All the popular contracts use more than the average amount of gas
Files: q4avggasvscon.py
top10.csv (additional input)
top10vsAvgGas.txt (log)