Big Data Processing (ECS765P)
Coursework – Analysis of Ethereum Transactions and Smart Contracts
Name – Sneh Patel – 220661364

Part A. Time Analysis

parta1:

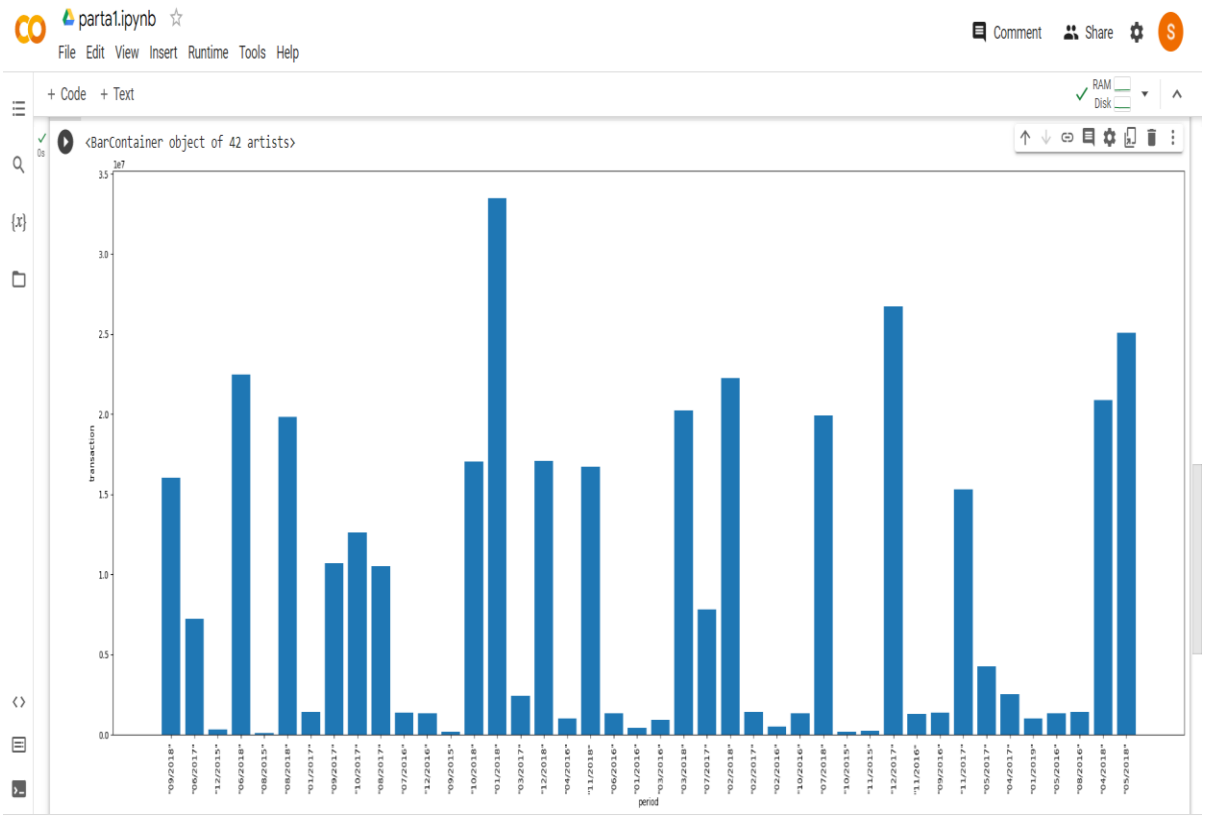
Create a bar plot showing the number of transactions occurring every month between the start and end of the dataset.

- In the start of the code, I have defined a function good_line that takes a single argument, line and splits it by commas and checks if the resulting list has 15 field. If yes, then returns true and if no then returns false.
- Then transaction.csv file is read and I create a clean_lines variable using filter method of lines that are not considered good.
- Then parta1 variable maps rdd to a tuple of string of month and year. From timestamp column it is converted to good format.
- Then monthly_transaction variable is created by calling reduceByKey method on the rdd which groups together tuples with same key and keeps adding. Then code print first 100 elements. Output is a list of tuple containing month and year with count of the number of transaction in that time.

  This is the output textfile.

[["09/2018", 16056742], ["06/2017", 7244657], ["12/2015", 347092], ["06/2018", 22471788], ["08/2015", 85609], ["08/2018", 19842059], ["01/2017", 1409664], ["09/2017", 10679242], ["10/2017", 12602063], ["08/2017", 10523178], ["07/2016", 1356907], ["12/2016", 1316131], ["09/2015", 173805], ["10/2018", 17056926], ["01/2018", 33504270], ["03/2017", 2426471], ["12/2018", 17107601], ["04/2016", 1023096], ["11/2018", 16713911], ["06/2016", 1351536], ["01/2016", 404816], ["03/2016", 917170], ["03/2018", 20261862], ["07/2017", 7835875], ["02/2018", 22231978], ["02/2017", 1410048], ["02/2016", 520040], ["10/2016", 1329847], ["07/2018", 19937033], ["10/2015", 205045], ["11/2015", 234733], ["12/2017", 26732085], ["11/2016", 1301586], ["09/2016", 1387412], ["11/2017", 15292269], ["05/2017", 4245516], ["04/2017", 2539966], ["01/2019", 1002431], ["05/2016", 1346796], ["08/2016", 1405743], ["04/2018", 20876642], ["05/2018", 25105717]]

  This is the output graph.



Parta2:

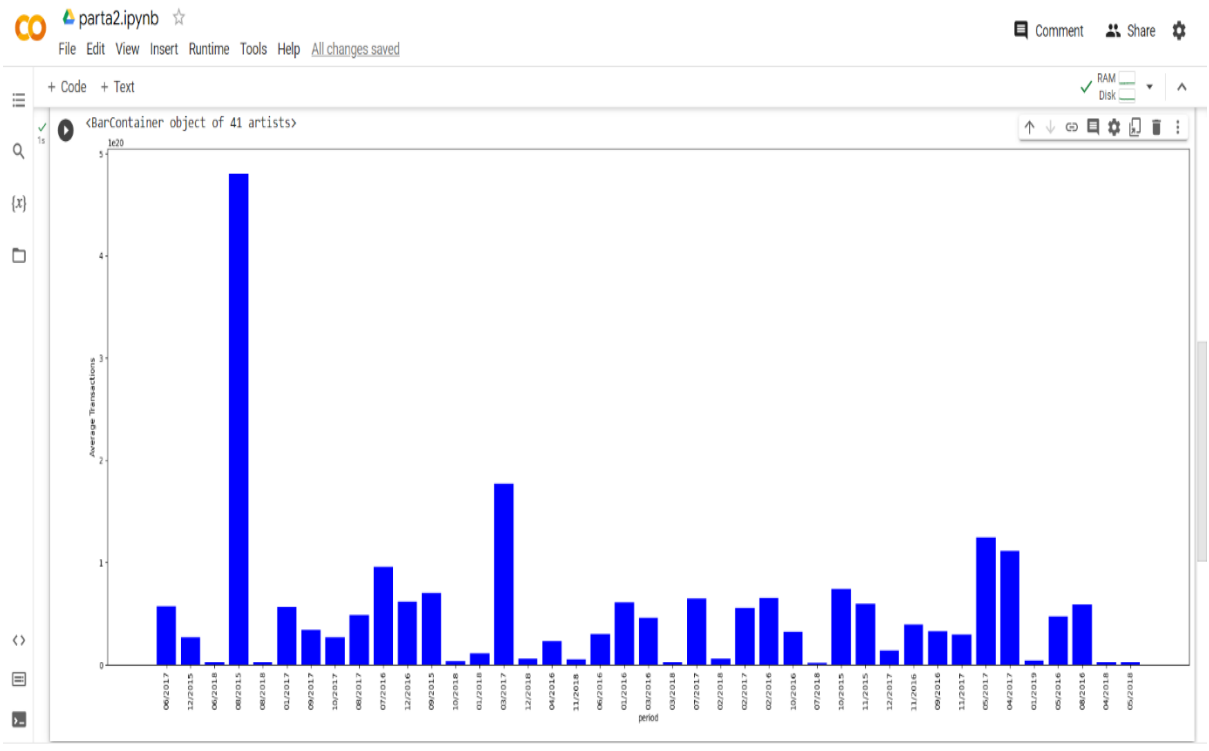
Create a bar plot showing the average value of transaction in each month between the start and end of the dataset.

- Here at the start, there is good_line function checks if the transaction.csv is valid. If line has 15 fields then line 7 and 11 is given float and integer respectively and returns true, otherwise false.
- Then I read the csv file in clean_lines rdd after filtering. After that map_transaction function is given to each line and extracts timestamp and value first and then converts timestamp to month and year and returns that as key and tuple of transaction value and count of 1. Then aggregation by month is done.
- Then date_vals rdd sums transaction values and counts by month by using reduce operation. This results reduce rdd with month and year as key and tuple with sums of values and counts as the value.
- Final part of code calculates average transaction value per month by dividing the counted sum values and results in avg_transactions rdd which then stores the result in out variable.

  This is the output textfile.

```
09/2018,3.7334811019825 2e+18
06/2017,5.6787722309363925e+19
12/2015,2.6764096183940473e+19
06/2018,2.8085234163056 75e+18
08/2015,4.8052118459597475e+20
08/2018,2.3989507981127 4e+18
01/2017,5.6202859565350 14e+19
09/2017,3.4372115150831 16e+19
10/2017,2.6761515215631016e+19
08/2017,4.8273956518851 63e+19
07/2016,9.5778235105821 62e+19
12/2016,6.1466586775381 21e+19
09/2015,7.0464679457673 68e+19
10/2018,3.0704021430259 7e+18
01/2018,1.1164572720750387e+19
03/2017,1.7702158094222 16e+20
12/2018,5.9448941634848 35e+18
04/2016,2.2670632047053 58e+19
11/2018,5.3979090489017 85e+18
06/2016,3.0490334850064945e+19
01/2016,6.1066070477195 54e+19
03/2016,4.5853064127780 79e+19
03/2018,2.7280798911623895e+18
07/2017,6.4981463792719405e+19
02/2018,6.2303627950909 59e+18
02/2017,5.5580090162629894e+19
02/2016,6.5547608759903 87e+19
10/2016,3.2444426339709256e+19
07/2018,2.2749347554396408e+18
10/2015,7.4169318093339 08e+19
11/2015,5.9484743862496 79e+19
12/2017,1.3731223538323202e+19
11/2016,3.9644316438351 21e+19
09/2016,3.2627612247557 69e+19
11/2017,2.9641103274740072e+19
05/2017,1.2484777365193179e+20
04/2017,1.1135007462190758e+20
01/2019,4.2547896734505 96e+18
05/2016,4.7046609524467 7e+19
08/2016,5.9081987372901 39e+19
```

This is output graph.



Part B. Top Ten Most Popular Services

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.

- First I read transaction.csv and contracts.csv. Then I filter those transaction and contracts rdds to remove those not valid records where good_transactions and good_contracts is defined and is given 15 and 6 field respectively.
- Then from both dataset, I extract and aggregate those to get total value of transaction per contract address. Then new rdd is created using good_transaction and extracts data inside. After that, new rdd total_values_in_address applies reduce operation to sum those values to each address. Then same things goes with contracts dataset.
- The total_values_per_contract rdd performs a join between The total_values_in_transaction and The total_values_in_contract to get combine values per contract address.
- This rdd is mapped to new rdd with contract address as the key and total value as value. On The total_values_per_contract rdd I give takeOrdered function to get top 10 contract addresses with highest total values.

  This is output textfile.

```
[('0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444', 8415536369994167867374641), ('0x7727e5113d1d161373623e5f49fd568b4f543a9e', 4562712851291534458774992O),
('0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef', 42552989136413198919298969), ('0xbfc39b6f805a9e40e77291aff27aee3c96915bdd', 2110419513809366005000000O),
('0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', 1554307763526374225471940O), ('0xabbb6bebfa05aa13e908eaa492bd7a8343760477', 1071948594562894613652468O),
('0x341e790174e3a4d35b65fdc067b6b5634a61caea', 8379000751917755624057500), ('0x58ae42a38d6b33a1e31492b60465fa80da595755', 2902709187105736532863818),
('0xc7c7f6660102e9a1fee1390df5c76ea5a5572ed3', 1238086114520042000000000), ('0xe28e72fcf78647adce1f1252f240bbfaebd63bcc', 1172426432515823142714582)]
```

Part C. Top Ten Most Active Miners

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.

- First, block_schema rdd is created using blocks.csv. Then i apply filter which checks if lines are 19 and second field is not equal to hash. Which ultimately filters out unvalid data.
- Then new rdd mapping_miner_and_block_size extracts miner and block size information from the filtered lines. It will have minor address as key and block size as value.
- We apply reduce operation on mapping_miner_and_block_size to get total block size mined by each miner. Then reduceKey function is applied on the rdd to add block size for each miner.
- I apply takeOrdered action on size_of_miner_block rdd to get top 10 miners with highest block size. Output list will have miner addresses with their total block size.

  This is output textfile.


```
[["0xea674fdde714fd979de3edf0f56aa9716b898ec8", 17453393724], ["0x829bd824b016326a401d083b33d092293333a830", 12310472526], ["0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c", 8825710065],
["0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5", 8451574409], ["0xb2930b35844a230f00e51431acae96fe543a0347", 6614130661], ["0x2a65aca4d5fc5b5c859090a6c34d164135398226", 3173096011],
["0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb", 1152847020], ["0x4bb96091ee9d802ed039c4d1a5f6216f90f81b01", 1134151226], ["0x1e9939daaad6924ad004c2560e90804164900341", 1080436358],
["0x61c808d82a3ac53231750dadc13c777b59310bd9", 692942577]]
```


Part D. Data exploration

The final part of the coursework requires you to explore the data and perform some analysis of your choosing. Below are some suggested ideas for analysis which could be undertaken, along with an expected grade for completing it to a *good* standard. You may attempt several of these tasks or undertake your own.

Scam Analysis

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 catergory 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%)

- For scam analysis, as we were given scams.json file, I first converted that json file to csv file with the help of some python coding websites and made a csv with less columns. But couldn't read and implement that file with the result.
- I have uploaded my tried code which I can say the steps of approach from my side. This code reads in two datasets, transactions.csv and scmas.json, and performs some data cleaning and processing on them using PySpark. After the scams.csv file didn't gave output, I tried it directly on json file as well.

- Firstly, it defines a dictionary category_map to map certain categories in the scams dataset to a general category of "Scam". Then, two functions is_valid_transaction and is_valid_scam are defined to filter out invalid lines in datasets.
- The extract_transaction_fields function is used to extract relevant fields from the transactions dataset, and returns a tuple containing the sender and receiver addresses and the amount of Ether transferred.
- Similarly, the extract_scam_fields function is used to extract relevant fields from the scams dataset, and returns a tuple containing the start and end addresses and the coin involved in the scam.
- Finally, the main data processing is performed using PySpark's rdd transformations. The filter method is used to remove invalid lines from the datasets, and then the map method is used to extract the relevant fields from each dataset. The resulting rdds containing the relevant fields are then used for further processing, such as joining and aggregation, to generate the desired output. That was my attempt but it did not give output.

- But only test1 file gave output which I have attached in zip file but not sure with the result.
- This code processes data related to Ethereum transactions and scams. It reads in two data files , and applies data cleaning and filtering to extract relevant information. The code then joins the two datasets and performs aggregation to obtain information about popular scams and Ether usage over time.
- The two functions are used to filter out lines of data that do not meet certain criteria. The RDDs are created by applying the functions. Then another rdd is created by mapping the to a tuple of scam address and a tuple of scam ID and category. The RDD is created by mapping those RDD to a tuple of transaction address and transaction value.
- The joins RDD is created by joining filter on the address field. The RDD is created by mapping the joins RDD to a tuple of scam ID and category and transaction value. The popular scams RDD is created by reducing it by scam ID and category and summing the transaction values. Finally, the rdd is created by taking the top 10 scams by transaction value.
- The next RDD is created by mapping the previous RDD to a tuple of scam address and date. The new RDD is created by mapping the filter to a tuple of transaction address and a tuple of date and transaction value. The joins1 rdd is created by joining filters on the address field. The RDD is created by mapping the RDD to a tuple of date and scam ID and category and transaction value. The final RDD is created by reducing by date and scam ID and category and summing the transaction values.

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%)

- I have done this problem in 2 different logic. First one is top100_washtrade.py. In this, as the difficulty of question is high and require abstract thinking, I went with spark dataframes which are higher level APIs.
- The check_transaction function is defined to check whether a given line of the CSV file contains valid transaction data.
- Then the transacions variable is created to read in the transaction.csv file. The trans variable is created by filtering out any invalid transactions using the check_transactions function. After that trans_map variable is created by mapping the trans RDD to a new RDD that only contains relevant transaction data.
- A Spark DataFrame is created from the trans_map RDD, with column names defined by the naming columns list. Then a window function is defined to calculate lagged values for each partition.
- The DataFrame is grouped by block number, to address, and value; total value, count, max timestamp, and count by partition were then calculated.
- The DataFrame is filtered for wash trades, which are defined as transactions with more than one occurrence in the same block or transactions with the same from_address and to_address.
- The trans_rdd RDD is created by mapping the filtered DataFrame to a tuple containing the block number, to address, and value.
- At the end, output RDD is created by reducing the trans_rdd by key, summing the transaction values for each key.
- The top100 variable is created by taking the top 100 values in the output RDD, sorted in descending order of transaction value. These represent the top 100 potential wash traders.

  This is output textfile.

- Now, onto the second file which I have used different approach but with same high level API spark dataframes.
- Read the "transactions.csv" file from the specified S3 bucket into a Spark DataFrame.
- I grouped the DataFrame by "from_address", "to_address", "value", "gas", "gas_price", "block_timestamp".
- Then Aggregate the data to compute the count of transactions for each group, average gas price, and previous to_address using the agg() method.
- I applied a filter to keep only the rows where one or more of the following conditions are met: transaction count is greater than 1, gas price is greater than 1.5 times the average gas price, value is 0 and gas is greater than 21000 and gas price is greater than 1,000,000,000, previous to_address is null or not equal to the current to_address.
- Selected the "from_address", "to_address", and "value" columns from the filtered DataFrame.
- After that I transformed the resulting DataFrame into rdd and extract the first 50 records to a list called "wash_trade_list".

  This is output textfile.

Contract Types

Identifying Contract Types: The identification and classification of smart contracts can help us to better understand the behavior of smart contracts and figure out vulnerabilities, such as confirming fraud contracts. By identifying features of different contracts such as number of transactions, number of unique outflow addresses, Ether balance and others we can identify different types of contracts. How many different types can you identify? What is the most popular type of contract? More information can be found at https://www.sciencedirect.com/science/article/pii/S0306457320309547#bib0019. Please note that you are not required to use the types defined in this paper. Partial marks will be awarded for feature extraction and analysis. Classifying contracts based upon ERC20 or ERC721 only will be awarded minimal marks e.g. 10. I expect something more sophisticated for this. (40%/40%)

- My contract types file did not give me output as well. I have attached code of my try.
- This code loads two RDDs, "transactions_rdd" and "contracts_rdd", from two CSV files stored in an S3 bucket. It then processes these RDDs to create a new RDD that contains key-value pairs with the contract address as the key and various features related to the contract as the value.
- The "transactions_rdd" is split into individual fields and transformed into key-value pairs with the "to_address" field as the key. The "contracts_rdd" is similarly transformed into key-value pairs with the "address" field as the key. The two RDDs are then joined on their respective keys to create a new RDD with the contract address as the key and various features as the value.

- The features extracted from the joined RDD include the total number of transactions, the number of transactions involving contract creation or interaction, the total ether transferred, the contract type (either "Contract Creation" or "Contract Interaction"), and the time difference between the contract creation and the last transaction involving the contract.
- The new RDD is then reduced by key using aggregation functions to obtain the desired features. Contracts with zero ether held are filtered out, and the resulting RDD is transformed into a list of tuples containing the contract address, ether held, and contract type. The final output is a list of all contracts with non-zero ether held, sorted by contract type.

Miscellaneous Analysis

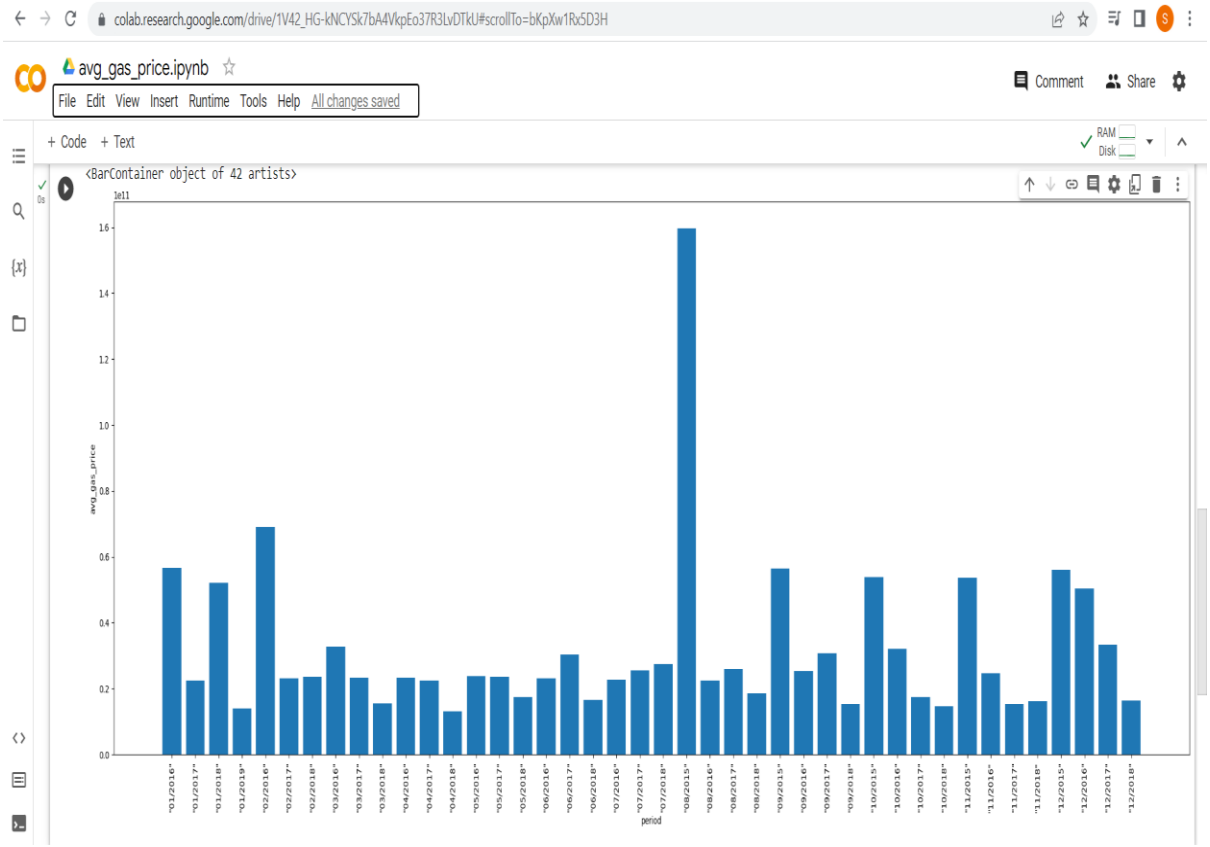
Gas Guzzlers: For any transaction on Ethereum a user must supply gas. How has gas price changed over time? Have contracts become more complicated, requiring more gas, or less so? How does this correlate with your results seen within Part B. To obtain these marks you should provide a graph showing how 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. (20%/40%)

- For this problem also, the code is divided into 2 files. The first one is for avg_gas_price.
- We first read the transaction.csv data from S3 and create an RDD.
- Then I define a function to filter out invalid transactions based on the number of fields and the validity of two specific fields. After that we filter the rdd to remove invalid transactions using the function defined in above.
- We then define a function to map each transaction to a tuple of (month, (gas_price, 1)). Mapping each transaction in the filtered RDD using the function defined as map_transactions to create an RDD of (month, (gas_price, 1)) tuples.
- Then reducing the RDD by key to get the sum of gas prices and the count of transactions for each month and mapping each month to a tuple of (month, average_gas_price) by dividing the sum of gas prices by the count of transactions for each month.
- At the end, I Sort the rdd by month in ascending order and Collect the results as a list of tuples containing the month as the first element and the average gas price for that month as the second element.

This is first output textfile for avg_gas_price.

[["01/2016", 56596270931.31685], ["01/2017", 22507570807.719795], ["01/2018", 52106060636.845055], ["01/2019", 13954460713.077589], ["02/2016", 69180681134.38847], ["02/2017", 23047230327.254303], ["02/2018", 23636574203.828987], ["03/2016", 32797039087.35667], ["03/2017", 23232253600.81683], ["03/2018", 15549765961.743269], ["04/2016", 23361180502.721268], ["04/2017", 22355124545.395313], ["04/2018", 13153739247.92998], ["05/2016", 23746277028.26325], ["05/2017", 23572314972.01526], ["05/2018", 17422505108.986416], ["06/2016", 23021251389.81214], ["06/2017", 30199442465.128727], ["06/2018", 16533308366.813042], ["07/2016", 22629542449.24175], ["07/2017", 25460300456.232975], ["07/2018", 27506077453.154324], ["08/2015", 159744029578.0331], ["08/2016", 22396836435.958485], ["08/2017", 25905774673.99024], ["08/2018", 18483235826.89456], ["09/2015", 56511301521.033226], ["09/2016", 25270403393.62608], ["09/2017", 30675032016.988663], ["09/2018", 15213870989.52338], ["10/2015", 53901692120.53661], ["10/2016", 32112869584.91466], ["10/2017", 17498286426.76892], ["10/2018", 14526936383.350012], ["11/2015", 53607614201.796776], ["11/2016", 24634294365.279953], ["11/2017", 15312465314.693556], ["11/2018", 16034859008.681646], ["12/2015", 55899526672.35486], ["12/2016", 50318068074.686455], ["12/2017", 33439362876.108326], ["12/2018", 16338844844.014643]]

This is output graph.



- Now moving onto the second file which has avg_gas_used_year.
- First I read the transactions and contracts datasets into dataframes using Spark. Because using high level API makes job quite easy.
- Then select the relevant fields from the transactions and contracts dataframes to be used in the subsequent operations.
- Define a function to map the transactions dataframe into the required format, which includes the "to_address", "gas", and "block_timestamp" fields from the

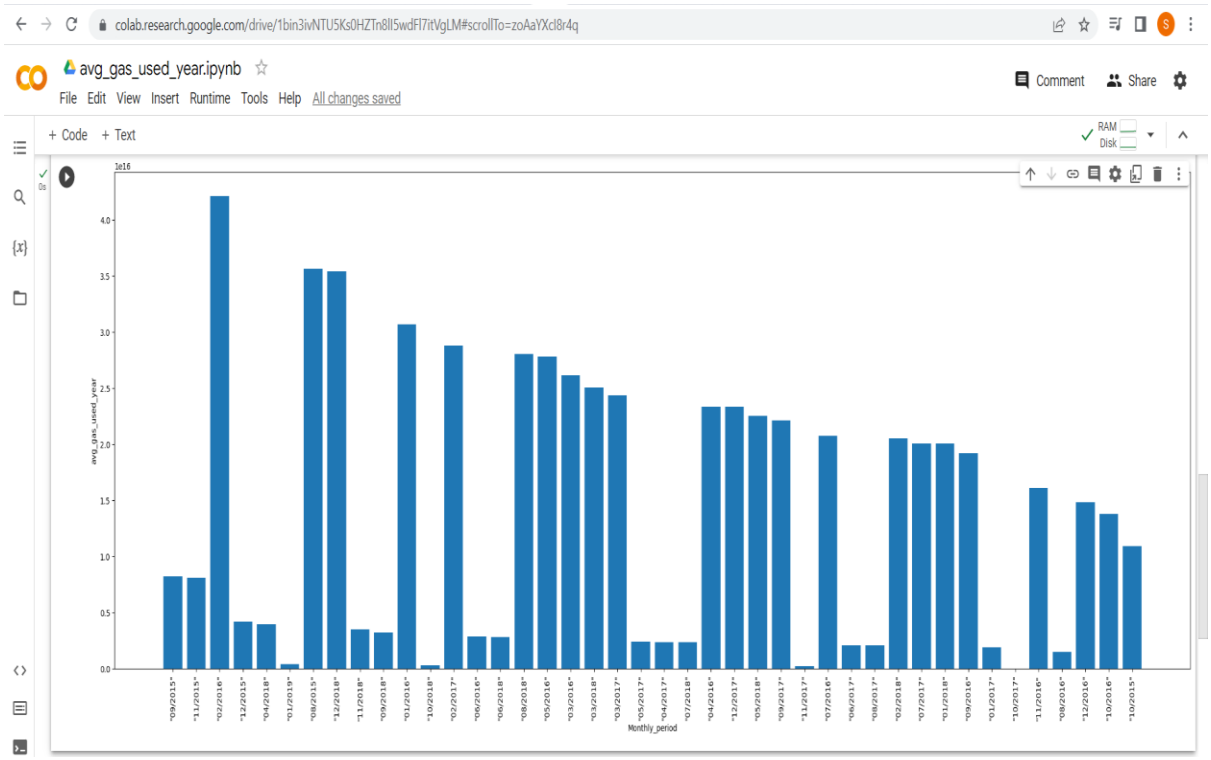
original dataframe, as well as the date of the transaction in the format "mm/yyyy".

- Now I map the transactions dataframe using the previously defined mapping function to obtain an rdd of tuples where each tuple contains the "to_address" and the gas used in that transaction.
- Then again define a function to map the contracts dataframe into the required format, which includes only the contract addresses from the original dataframe, which maps the contracts dataframe using the previously defined mapping function to obtain an RDD of tuples where each tuple contains the contract address and a value of 1.
- Join the transactions rdd with the contracts rdd based on the "to_address" and "address" fields respectively, to obtain an rdd of tuples where each tuple contains the "to_address", the gas used in the transaction, and a value of 1.
- Now a function will be introduced to map the joined rdd into the required format, which includes the date of the transaction and a tuple containing the gas used in the transaction and a value of 1.
- We reduce the joined rddusing the previously defined mapping function to obtain an rdd of tuples where each tuple contains the date of the transaction and a tuple containing the sum of gas used and the count of transactions for that date.
- Then define a function to map the reduced RDD into the required format, which includes the date of the transaction and the average gas used in transactions for that date.
- At last map the reduced RDD using the previously defined mapping function to obtain an RDD of tuples where each tuple contains the date of the transaction and the average gas used in transactions for that date and finally collect and sort the final RDD by the average gas used in descending order.

This is the output text file.


```
[["09/2015", "821011.3419611065"], ["11/2015", "808819.8470650569"], ["02/2016", "421546.35267500667"], ["12/2015", "421430.0369266747"], ["04/2018", "397139.0777938786"], ["01/2019",
"377000.769162696"], ["08/2015", "356364.30898021307"], ["12/2018", "354023.42463293206"], ["11/2018", "351709.9608263743"], ["09/2018", "320847.6191174533"], ["01/2016",
"306853.88528227125"], ["10/2018", "304058.511235191"], ["02/2017", "288033.05409124907"], ["06/2016", "285748.3862790856"], ["06/2018", "280741.0498537269"], ["08/2018",
"280252.74579720566"], ["05/2016", "278264.25591171347"], ["03/2016", "261626.41243942833"], ["03/2018", "250838.30657591383"], ["03/2017", "243614.15575359698"], ["05/2017",
"239714.1703375609"], ["04/2017", "237381.5255083979"], ["07/2018", "234143.8078040237"], ["04/2016", "233613.63115644714"], ["12/2017", "233579.24725262943"], ["05/2018",
"225455.22331318035"], ["09/2017", "221447.61487522215"], ["11/2017", "217321.880537949"], ["07/2016", "207590.73473255133"], ["06/2017", "205877.5735886294"], ["08/2017",
"205029.6725001102"], ["02/2018", "204948.09528605614"], ["07/2017", "200610.43534558726"], ["01/2018", "200588.84111940436"], ["09/2016", "192223.63495248306"], ["01/2017",
"189320.4329184991"], ["10/2017", "188448.11841054"], ["11/2016", "161155.52778544786"], ["08/2016", "151401.4865178056"], ["12/2016", "148029.41172236484"], ["10/2016",
"137672.74968342367"], ["10/2015", "1094350.5834600439"]]
```


This is the graph output.



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. Note that all of these values are hex_strings so you can assume that each character after the first two requires four bits (this is not the case in reality but it would be possible to implement it like this) as this will affect your calculations. (20%/40%)

- So the flow of the code is that first we load the blocks dataset into a PySpark DataFrame.
- The we define a list of hex columns in the DataFrame that are in hexadecimal format and need to be converted to a readable format. Which we calculate the size of each hex column in bytes by iterating over the hex columns, selecting them in the DataFrame, and applying the length() function to each column.
- The length() function calculates the number of characters in each string column. Since each character in a hexadecimal string represents 4 bits, we divide the length of each hex column by 4 to get the size in bytes.
- Then I sum the sizes of all hex columns to get the total size of the hex columns in bytes and convert the total size of hex columns from bytes to megabytes.
- Then again calculate the size of the remaining columns in the DataFrame. We first create a list of columns that are not hex columns by subtracting the hex columns list from the list of all columns in the DataFrame. We then select each non-hex column in the DataFrame, apply the length() function, and sum

the resulting sizes using an rdd and convert the total size of the remaining columns from bytes to megabytes.

- Now calculate the percentage of space that would be saved by removing the hex columns. We divide the total size of the hex columns by the sum of the total size of hex columns and the total size of the remaining columns, and multiply by 100 to get a percentage.
- At last, print the total size of hex columns to be removed, total size of remaining columns, and the percentage of space saved by removing hex columns.

This is the output textfile of remaining_size.

```
45.670411109924316
```

This is the output textfile of space_saved.

```
0.00040614677352983407
```

This is the output textfile of total_hex_size.

```
0.00018548965454101562
```