ECS765P - BIG DATA PROCESSING
2022/2023 - Semester 2
Analysis of Ethereum Transactions and Smart Contracts

Lecturer: Professor Ahmed M. A. Sayed
Name: Hoang Hoa Pham
Student ID: 220454836

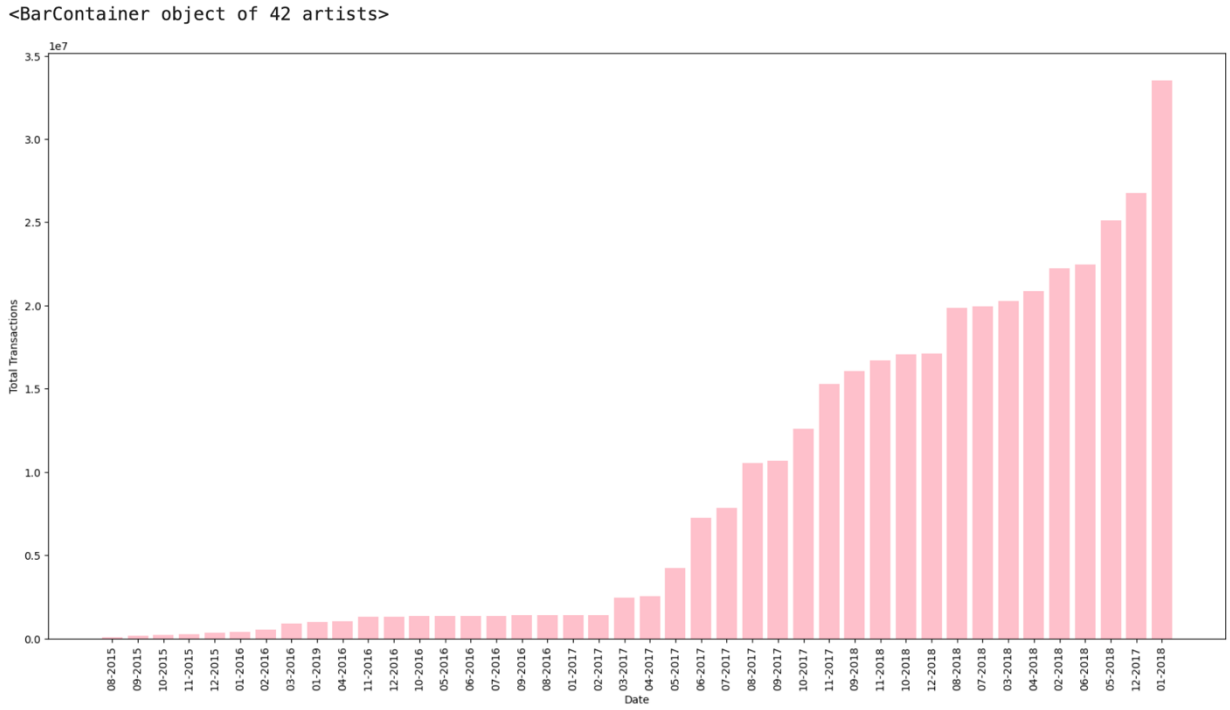
PART A. TIME ANALYSIS

Dataset: transactions.csv
Source Code: timeanalysis1.py; timeanalysis2.py; partA_1.ipynb; partA_2.ipynb
Output File: time_analysis_a.txt; time_analysis_b.txt
Command: ccc create spark timeanalysis1.py -s -d; ccc create spark timeanalysis2.py -s -d

- For both time series, I uses PySpark library to read Ethereum transactions data ("/ECS765/ethereum-parvulus/transactions.csv") from an AWS S3 bucket.
- I uses the reduceByKey function to group the transactions by month and year. Then, the spark map() function counts the number of transactions and average value of transaction according to each month from the end to the beginning of the dataset.

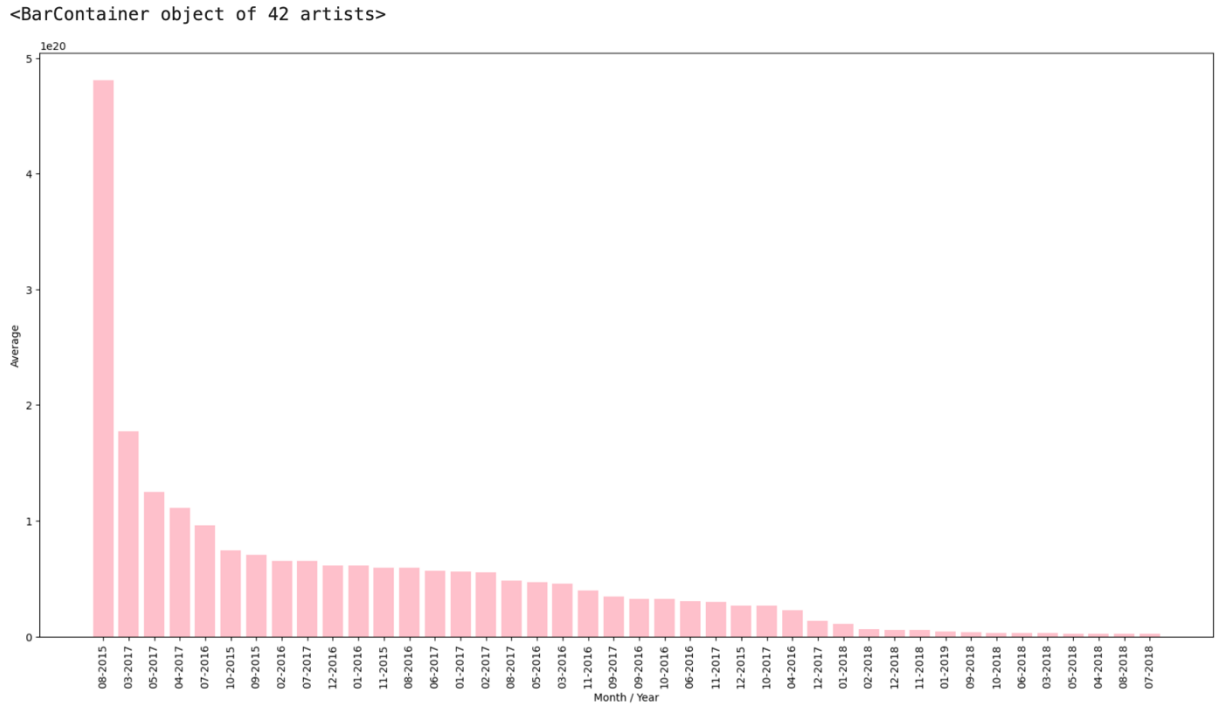
a) Number of transactions occurring every month

As we can see from the plot, the number of transactions per month seemingly to increase gradually throughout the year from 2015 to 2017. From 2017 to 2018, the number of transactions sharply increase from 2 million to peak 33 million transactions



b) Average value of transaction in each month

Contradicting to previous graph, 2015 started very high at nearly 470 million average transaction per month. From March 2017 to July 2018, the million average transaction declines significantly to less than 10 million average transactions.



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

Dataset: transactions.csv & contracts.csv
Source Code: popularservices.py
Output File: popular_service.txt
Command: ccc create spark popularservices.py -s -d

Methodology:

- I created two functions called good_line_tran() and good_line_contract() which checks the lines for the "transaction" and "contract" files. These functions will later on filter out invalid lines
- I then create top_ten_most_popular_services() funtion to retrieve the data from the "transaction" and "contract" files in the AWS S3 bucket

- The "map" function turned the valid lines into key-value pairs

- I then use “reduceByKey” to aggregate service name transaction data, following up with “join” method to consolidate with the contract data. The top 10 services will be retained by using “takeOrdered” and sorted descendently according to the Ethereum Value

Result:

	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%)

Dataset: blocks.csv

Source Code: activeminers.py
Output File: active_miners.txt
Command: ccc create spark activeminers.py -s -d

Methodology

Similarly to the previous task, I first created a function called good_line_blocks() to check the line in “blocks” data and clear out invalid lines. The blocks_mapper() function will then map the lines to the tuple in the miner and the mined block size.

I get the top 10 most active miner by creating the top_ten_most_active_miners() function to read block data from ("s3a://" + s3_data_repository_bucket + BLOCKS_FILE_PATH). takeOrdered again was used to get top 10 miners according to the mined block size in descending order

Result

	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: POPULAR SCAMS (20%/40%)

Dataset: transactions.csv, scams.json

Source Code: scamanalysis.py; partD_Scams.ipynb
Output File: lucrative.txt; corr.txt; time.txt
Command: ccc create spark scamanalysis.py -s -d

Methodology:

- The main 2 files to be retrieved from AWS S3 bucket are transactions.csv and scams.json.
- I created read_scam() function reads the scam data and returns id, address, name, category, and status fields. I then used the map function to yield the scam and transaction feature format into the following: (address, (id, category)) and (address, value); then "join" the features
- I create a function called mapper_transactions(line) "map"-ing each line in the transactions RDD to key-value pairs: the address as the key and the month-year and value as the value.
- **The top 10 scams by value** was generated from the scam_analysis() function: lucrative_scams_features (joins scam address to id and category) and lucrative_trans_feature (joins each transaction address to its value). These 2 RDDs are then "join" together into ((id, category), value). I used the 2 functions reduceByKey() and takeOrdered() to reduce the RDD according to the id and category, and then filter and sort top 10 scams by value descend order.
- I create corr_scams variable to create an RDD and then reduced using reduceByKey() function

```
corr_scams = time_trans_and_scams.map(lambda x:((x[1][1][0], x[1][1][1]), 1))
corr_scams = corr_scams.reduceByKey(operator.add)
print(corr_scams.take(100))
```

- In the Jupyter notebook , I use the seaborn package to generate heatmap using the heatmap() function

```
import seaborn as sns
# Convert the data to a pandas dataframe
df = pd.DataFrame(content, columns=['Category', 'Status', 'Counts'])

# Pivot the table to create a matrix for the heatmap
table = pd.pivot_table(df, values='Counts', index='Category', columns='Status')
print(table)
sns.heatmap(table, cmap='YlOrRd', annot=True)
plt.show()
```

Result:

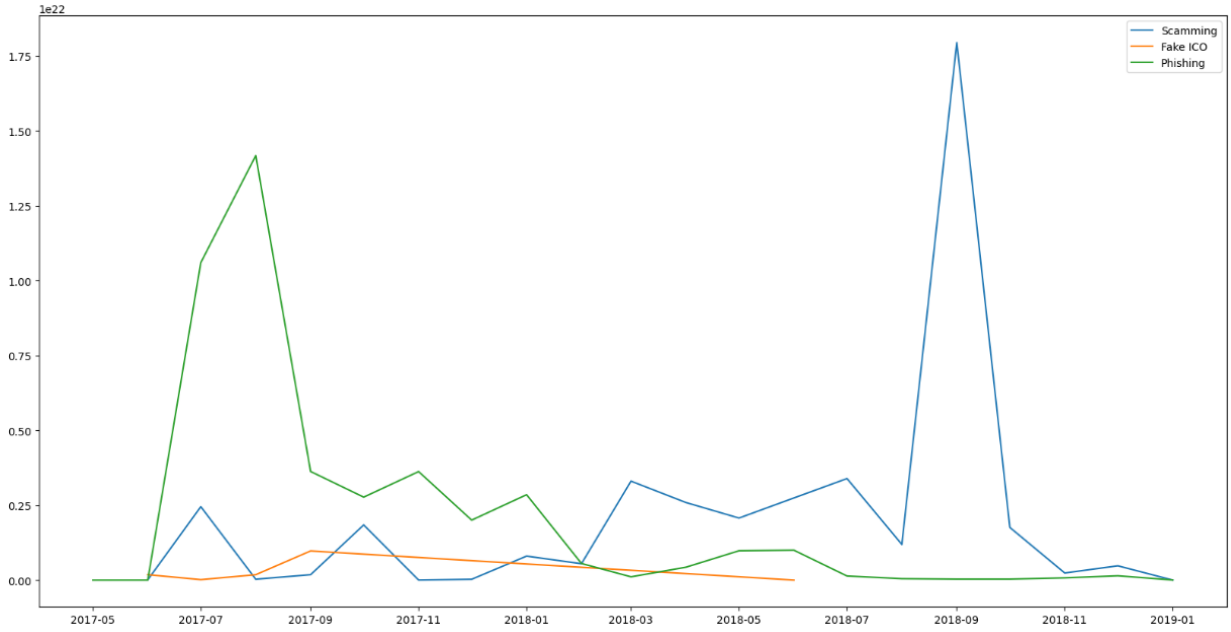
Below is the top 10 scams by value Table.

- As we can see, the most lucrative form of scam is "Scamming" with the ID 5622

	Date	Value
1	[5622, Scamming]	1.670908e+22
2	[2135, Phishing]	6.583972e+21
3	[90, Phishing]	5.972590e+21
4	[2258, Phishing]	3.462808e+21
5	[2137, Phishing]	3.389914e+21
6	[2132, Scamming]	2.428075e+21
7	[88, Phishing]	2.067751e+21
8	[2358, Scamming]	1.835177e+21
9	[2556, Phishing]	1.803047e+21
10	[1200, Phishing]	1.630577e+21

Below is the line graph demonstrating the changes throughout 2017 to 2019 of each form of scam

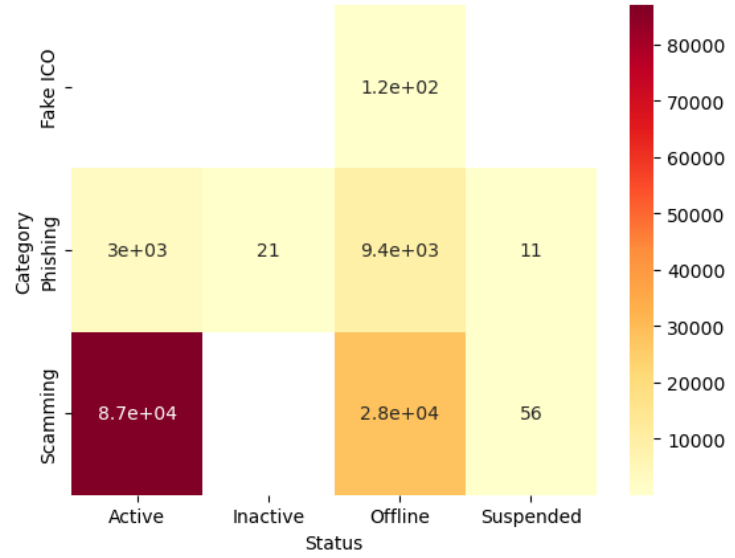
- All three form of scams start out very low in 2017
- For Scamming, the value seemingly to be increase throughout the year until it hits its peak over 1.8e+22 on 09-2018
- Contrary to Scamming, Phishing reaches its peak on 08-2017 at 1.4e+22
- In the meanwhile, Fake ICO value doesn't seem to have much significant change throughout the year ranging from 0.09e+22 to 1.8e+22



Below is the heatmap to analyze the correlation between scam forms and active status:

- For Scamming: There are a strong correlation with the Active and least correlated with Inactive
- For Fake ICO: The strongest correlation is Offline
- For Phishing: The strongest correlation is also Offline and the least correlation is Suspended

Status	Active	Inactive	Offline	Suspended
Category				
Fake ICO	NaN	NaN	121.0	NaN
Phishing	2965.0	21.0	9422.0	11.0
Scamming	86943.0	NaN	28056.0	56.0



MISCELLANEOUS ANALYSIS: GAS GUZZLERS (20%/40%)

Dataset: transactions.csv; contracts.csv
Source Code: partD_Gas_Guzzlers.ipynb; gasguzzlers.py
Output File: average_gas_price.txt; average_gas_used.txt
Command: ccc create spark gasguzzlers.py -s -d

Methodology:

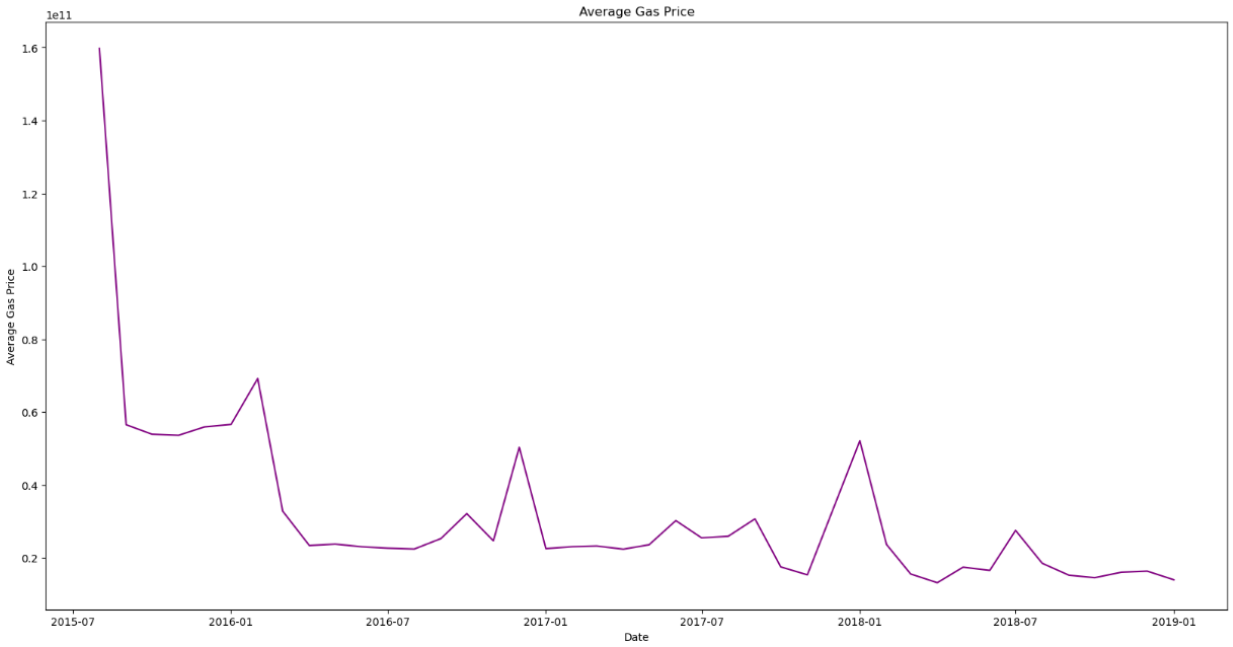
I defined the gas_guzzlers() function along with reduceByKey() function to perform both of the calculation tasks:

- In the first task, the average gas price per month is calculated by the division of the total gas price over the total number of transactions
- In the second task, I used the "join" function on the transaction data and contract data to get the average gas used by each contract address per month

Result:

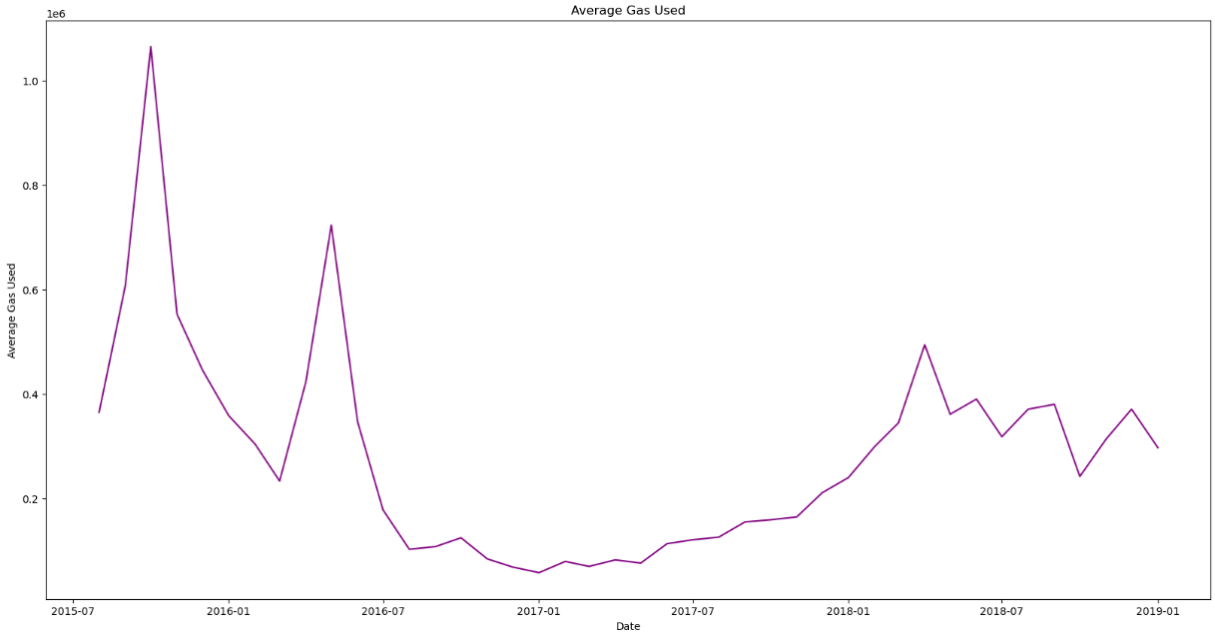
Graph showing how gas price per month has changed over time:

Average Gas Price starts at maximum value at 1.6e+11 Wei in August 2015 then declines throughout the year with the 1.4e+10 Wei by 2019



Graph showing how gas used for contract transactions:

The average gas usage is ranging pretty high during 2015 to mid-2016 period reaching its peak at approximately 1,064,341 on 10/2015. Between mid-2016 to 2018 the average gas usage seemingly to decrease significantly at the minimum of 58,527. From 2018 to 2019, the gas usage slightly increases



- By observing the graph, from PART B the most popular contracts 0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444 use more than the average gas_used