ECS765P / Big Data Processing Coursework - 2023

Analysis of Ethereum Transactions and Smart Contracts

Student Details:
Name: Farheen Dairkee
Student ID: 220843155

Dataset Overview:

The dataset provided is of an Ethereum network that contains transactions, blocks and contracts information. An additional dataset for scam analysis has been provided.

Introductory Steps:

All codes have been written in Python using the pyspark library. The libraries imported are given below:

```
import sys, string
import os
import socket
import time
import operator
import boto3
import json
from pyspark.sql import SparkSession
from datetime import datetime
from time import gmtime, strftime
```

Part A. Time Analysis (25%)

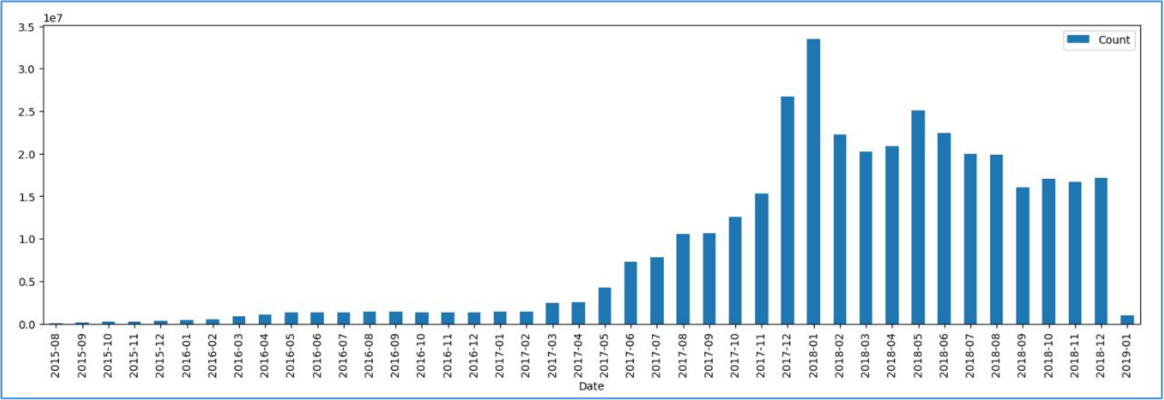
Create a bar plot showing the number of transactions occurring every month between the start and end of the dataset.

Approach Explained:
1. The transaction csv file is read using textFile function.
2. Any lines with unreliable or missing timestamps are filtered out.
3. The timestamp is converted to year-month. The count of transactions for every year and month is created using ‘map’ (this applies a given function to each element of an RDD and returns new RDD).
4. The transaction count is reduced to its sum as per year-month using ‘reduceByKey’ (this function combines values for each key in an RDD using a specified reduced function) and ‘add’ from operator library.

Spark Code Snippet:

```
transactions = spark.sparkContext.textFile("s3a://" + s3_data_repository_bucket + "/ECS765/ethereum-parvulus/transactions.csv")
clean_transactions = transactions.filter(good_line_t)
transaction_count = clean_transactions.map(lambda b: (time.strftime("%Y-%m",time.gmtime(int(b.split(',')[11]))),1))
trans_records=transaction_count.count()
transaction_count=transaction_count.reduceByKey(operator.add)
print(transaction_count.take(2))
#[["2017-06", 7244657], ["2018-03", 20261862]]
```

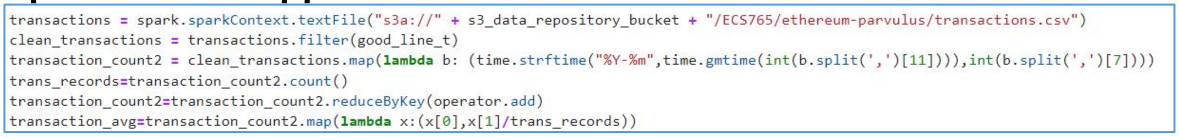
Plot created - Number of Transactions vs Time:



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

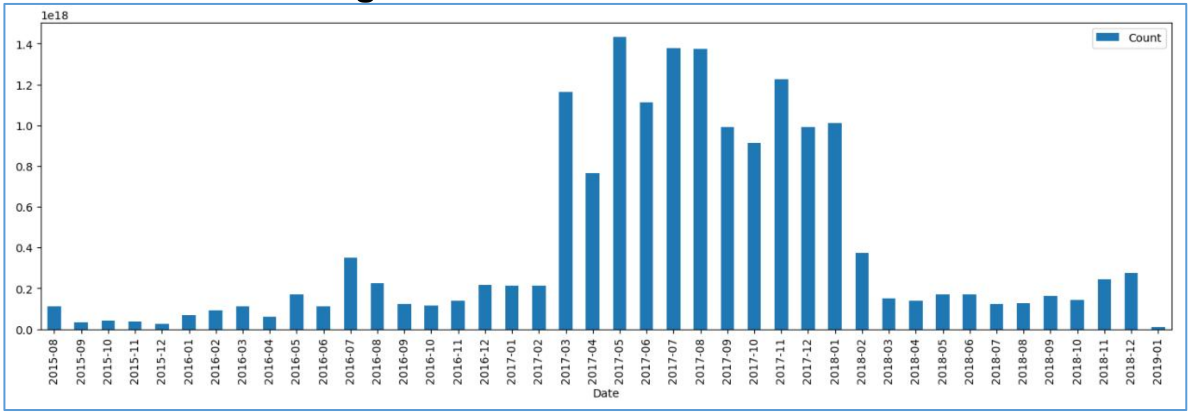
Approach Explained:
1. The transaction csv file is read using textFile function.
2. Any lines with unreliable or missing timestamps and ether value are filtered out.
3. The timestamp is converted to year-month. Element wise timestamp and ether value is created for each row using ‘map’ (this applies a given function to each element of an RDD and returns new RDD).
4. The ether value is reduced to its sum as per year-month using ‘reduceByKey’ (this function combines values for each key in an RDD using a specified reduced function) and ‘add’ from operator library.

Spark Code Snippet:



```
transactions = spark.sparkContext.textFile("s3a://" + s3_data_repository_bucket + "/ECS765/ethereum-parvulus/transactions.csv")
clean_transactions = transactions.filter(good_line_t)
transaction_count2 = clean_transactions.map(lambda b: (time.strftime("%Y-%m",time.gmtime(int(b.split(',')[11]))),int(b.split(',')[7])))
trans_records=transaction_count2.count()
transaction_count2=transaction_count2.reduceByKey(operator.add)
transaction_avg=transaction_count2.map(lambda x:(x[0],x[1]/trans_records))
```

Plot created - Average Ether Value in transaction vs Time:



Part B. Top Ten Most Popular Services (25%)

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.

Approach Explained:
1. The address and ether value is read from transaction csv file and any lines with unreliable or missing data in given columns is removed.
2. A new RDD with tuples of address and ether value is created using the old RDD where map function is applied to each element.
3. Then the contracts csv file is read from which contract address is required. A RDD with tuple of contract and integer (this integer has no significance in the code, it is only used because map function requires a tuple) is created.
4. The transaction information is joined with contract information with address as key using ‘join’ function (combines two RDDs based on a common key and returns a new RDD with tuples of the key and corresponding values from both RDDs, where the key exists in both RDDs).
5. The final tuple received looks like:
('0x782c4adfab128f9d9475d3403e575d635c95e333', (6100320400000000000, 1))

6. From the above, a new RDD is created where address and ether value is taken.
7. Then a ‘reduceByKey’ function is applied to reduce each ether value to its sum as per contract address.
8. Finally, the aggregate ether values are sorted to find top 10 contracts with highest total ether value.

Spark Code Snippet:

```
    transactions = spark.sparkContext.textFile("s3a://" + s3_data_repository_bucket + "/ECS765/ethereum-parvulus/transactions.csv")
    clean_transactions = transactions.filter(good_line_t)

    trans_to_address=clean_transactions.map(lambda l: (l.split(',')[6],int(l.split(',')[7])))
    trans_to_address=trans_to_address.reduceByKey(operator.add)
    print(trans_to_address.take(2))
    #[('0x2cf3cf3c9fd3dccd6fa96e495046b74fafbdf838', 6888977220000000000), ('0x01d1d9b03e7315a9594c5682bc9d17b3e5a824bf', 46287694167912)]

    contracts = spark.sparkContext.textFile("s3a://" + s3_data_repository_bucket + "/ECS765/ethereum-parvulus/contracts.csv")
    clean_contracts = contracts.filter(good_line_c)
    contract_address=clean_contracts.map(lambda l:(l.split(',')[0],1))
    Ether_Data=trans_to_address.join(contract_address)

    print(Ether_Data.take(2))
    #[('0x782c4adfab128f9d9475d3403e575d635c95e333', (6100320400000000000, 1)), ('0xe9396ba86bf3fa883e62e3ba43784414624cbc33', (3200000000000000,
1))]

    final_Ether_Data=Ether_Data.map(lambda x: (x[0],x[1][0]))
    top10=final_Ether_Data.takeOrdered(10, key=lambda x: -x[1])
```

Output:

```
[('0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444', 84155363699941767867374641), ('0x7727e5113d1d161373623e5f49fd568b4f543a9e',
45627128512915344587749920), ('0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef', 42552989136413198919298969),
('0xfa52274dd61e1643d2205169732f29114bc240b3', 40546128459947291326220872), ('0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8',
24543161734499779571163970), ('0xbfc39b6f805a9e40e77291aff27aee3c96915bdd', 21104195138093660050000000),
('0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', 15543077635263742254719409), ('0xbb9bc244d798123fde783fcc1c72d3bb8c189413',
11983608729102893846818681), ('0xabbb6bebfa05aa13e908eaa492bd7a8343760477', 10719485945628946136524680),
('0x341e790174e3a4d35b65fdc067b6b5634a61caea', 8379000751917755624057500)]
```

Part C. Top Ten Most Active Miners (10%)

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.

Approach Explained:
1. The blocks csv file is read. Required columns are miner which contains address and size (of blocks mined).
2. Any Lines with unreliable or missing data in above fields is removed.
3. An RDD is created with ‘map’ function which contains tuples of miner and the size.
4. The size of blocks mined is reduced to its sum using ‘reduceByKey’ function and then sorted using ‘sortBy’ (sorts an RDD in ascending or descending order based on a specified key function and returns a new RDD).

Spark Code Snippet:

```
blocks=spark.sparkContext.textFile("s3a://" + s3_data_repository_bucket + "/ECS765/ethereum-parvulus/blocks.csv")
clean_blocks=blocks.filter(good_line_b)
sorted_counts = clean_blocks.map(lambda l : (l.split(',')[9],int(l.split(',')[12])))
counts=sorted_counts.reduceByKey(lambda a,b : a+b).sortBy(lambda x: x[1], False)
```

Output:

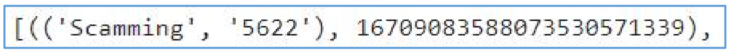
```
[["0xea674fdde714fd979de3edf0f56aa9716b898ec8", 17453393724], ["0x829bd824b016326a401d083b33d092293333a830", 12310472526],
["0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c", 8825710065], ["0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5", 8451574409],
["0xb2930b35844a230f00e51431acae96fe543a0347", 6614130661],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

Part D. Data exploration (40%)

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%)

Approach Explained:
1. From the transaction csv file, address, ether value and timestamp is read after filtering out any unreliable or missing data.
2. The RDD from transaction contains address, ether value, timestamp and integer 1 (to calculate count of transactions later).
3. The ether value is used to find the most lucrative scam and the address is used as the key to join with the scam dataset.
4. The scam dataset is given as a json file which is converted to csv file. In the csv file, relevant columns are scam addresses (which are multiple and are split into each row of the csv file), scam ID and scam category.
5. The scam csv file is then read. An RDD is created with the address as key of th etuple and ID and scam category as rest of the elements using 'map' function.
6. The transaction RDD and scam RDD is joined with the common key as address to create a new RDD which is used for the two purposes below:

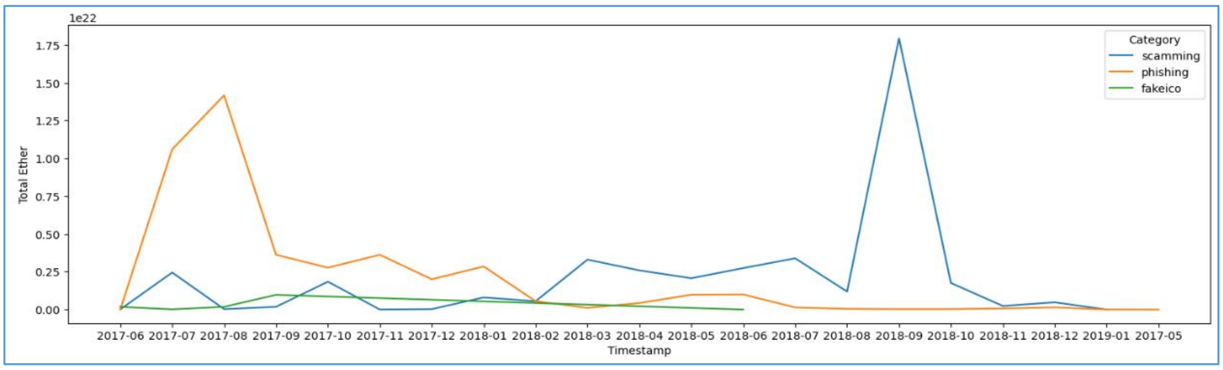
> To find the most lucrative scam and the ID associated with it.
7. A new RDD is created with 'map' which contains scam category, ID and ether value.
8. The ether value is reduced to sum with 'reduceByKey' for each scam category and ID.
9. The final output contains 100 scams with highest values, of which the first one is:



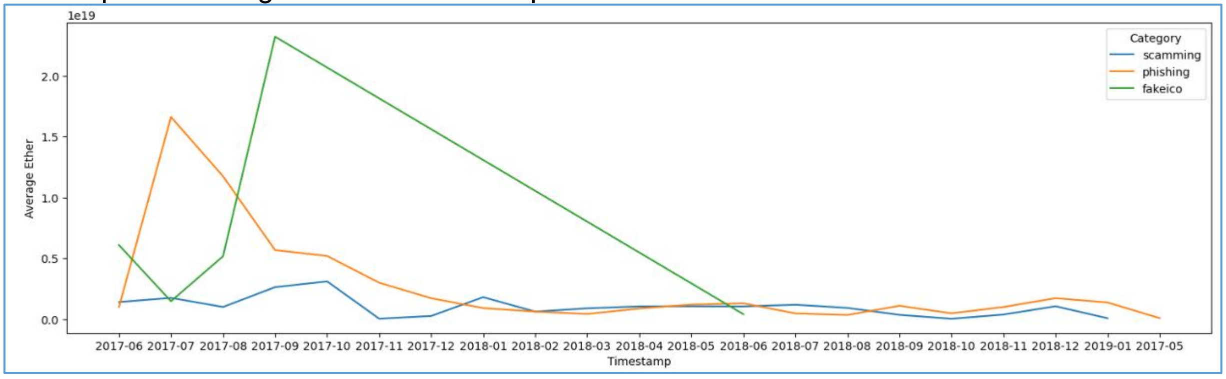
```
[(('Scamming', '5622'), 16709083588073530571339),
```

Therefore, the most lucrative scam is "**Scamming**".
The ID being associated with scamming is "**5622**" .

> To plot how ether received has changed over time for the dataset.
10. A new RDD is created with 'map' which contains scam category, timestamp, ether value and integer 1.
11. Both sum of ether value and count is calculated using 'reduceByKey'.
12. The final RDD contains scam category, timestamp, total ether value, count of transactions.
13. The above values will be saved in a text file which is used to plot the graphs.
14. The text file is parsed and the total ether value and count of transactions is used to calculate average ether value.
15. Graph for total ether vs timestamp is given below:



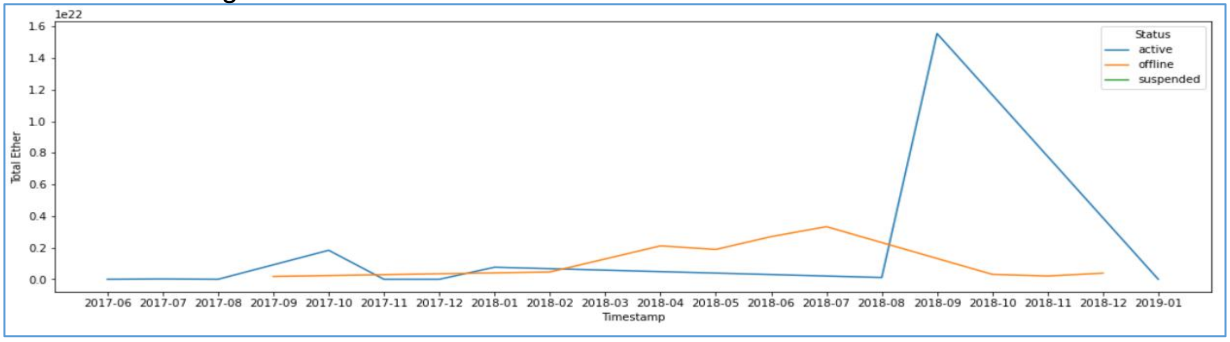
16. Graph for average ether vs timestamp:



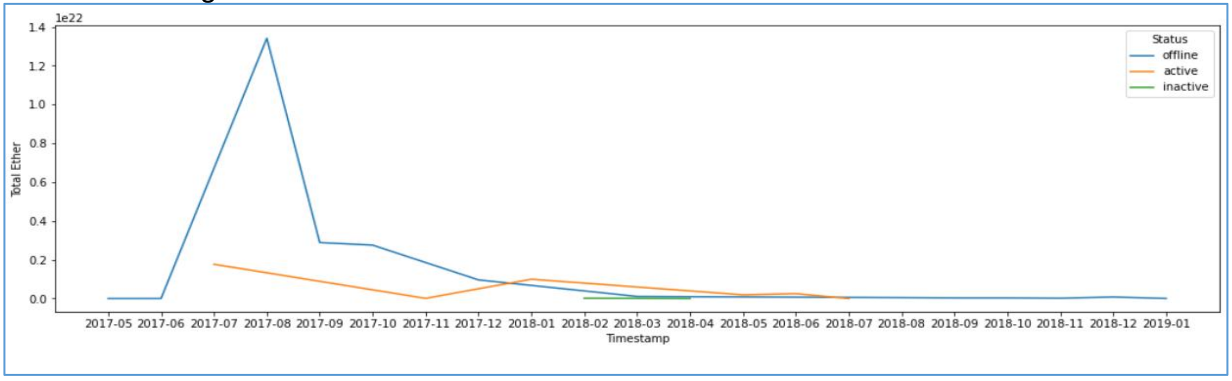
> To correlate the scams and their status with the total ether received.

17. An RDD is created using 'join' with key as address and contains the status of scams and scam category from the scams csv along with the ether value and timestamp from the transactions csv. The ether value is then reduced to its sum using 'reduceByKey'. Below are the graphs received.

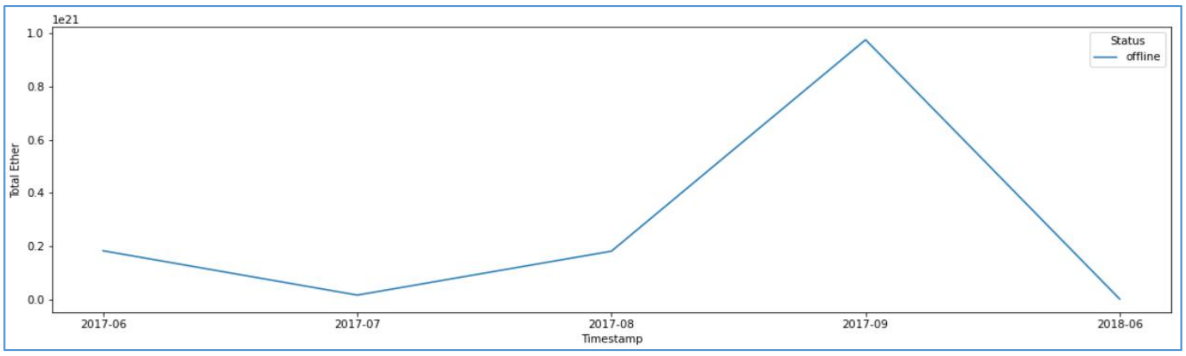
18. For scamming:



19. For Phishing:



20. For Fake ICO:

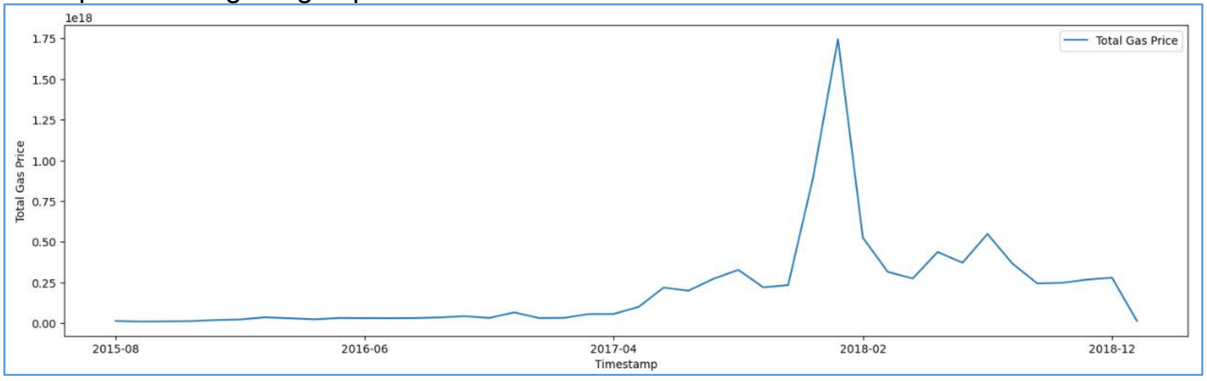


Conclusion:
The most lucrative scam is “Scamming: and as can be seen from the above plots, the total ether received in scam “Scamming” has peaks throughout with the highest near late 2018, whereas ether received in “Phishing” has reduced with time and ether received in “fake ICO” is only until mid 2018. While studying the status of the scams, it can be found that during the “offline” status of the scams the ether received is maximum as compared to the rest of the statuses.

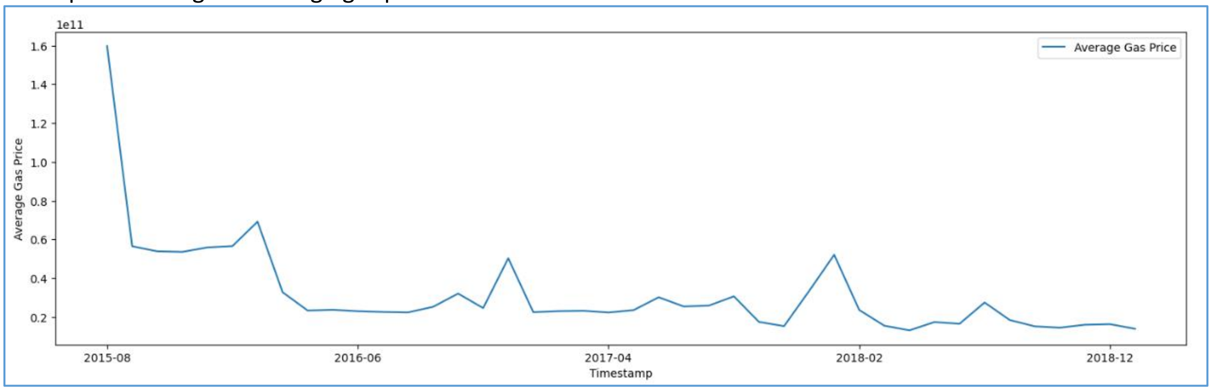
Gas Guzzlers
For any transaction on Ethereum a user must supply gas. How has gas price changed over time? Have contracts become more complicated, requiring more gas, or less so? How does this correlate with your results seen within Part B. To obtain these marks you should provide a graph showing how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

Approach Explained:
> To find how gas price has changed over time:
1. From the transaction file, fetch timestamp and gas price while filtering out missing values.
2. An RDD will be created from ‘map’ which contains timestamp, gas price and integer 1 (to calculate average gas price later).
3. For each timestamp, the total gas price and total number of records will be calculated using the ‘reduceByKey’.
4. Then to create a file with required format, a new RDD is created with timestamp, total gas price and average gas price for that timestamp.

5. Graph for change in gas price over time:

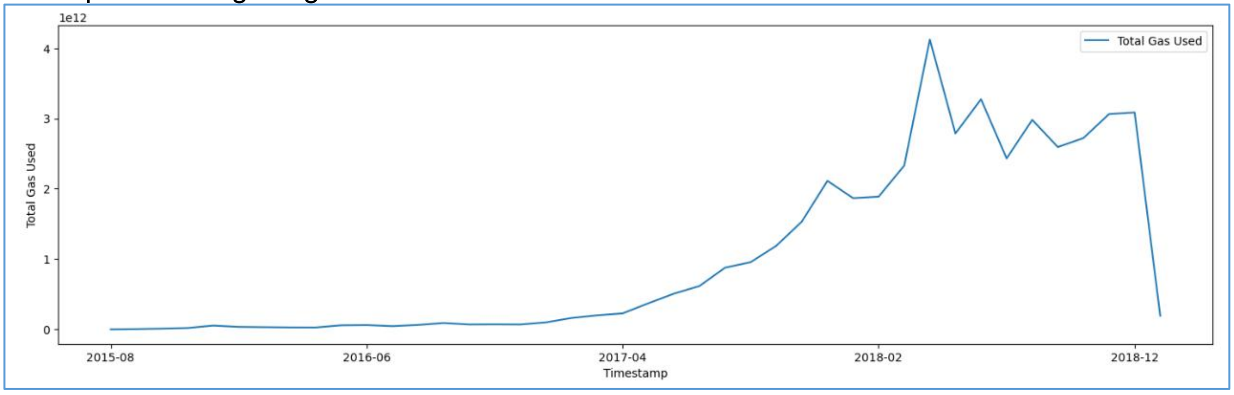


6. Graph for change in average gas price over time:

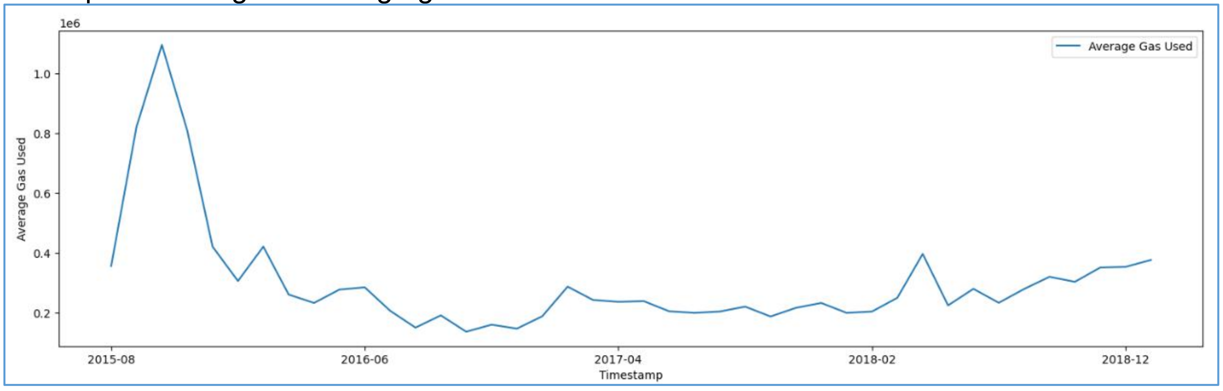


> To understand how gas used has changed over time for contracts:

1. From the transaction file, fetch timestamp, gas and address and remove any missing values or unreliable data.
2. From the contracts file, fetch address.
3. From both files, separate RDD were created using 'map'.
4. These were then joined using address as the common key so that only addresses of contracts are considered
5. Contract address was then ignored and total gas and average gas is calculated using for each timestamp.
6. Graph for change in gas used over time for contracts:

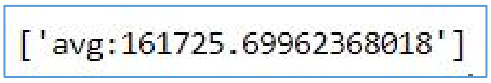


7. Graph for change in average gas used over time for contracts:



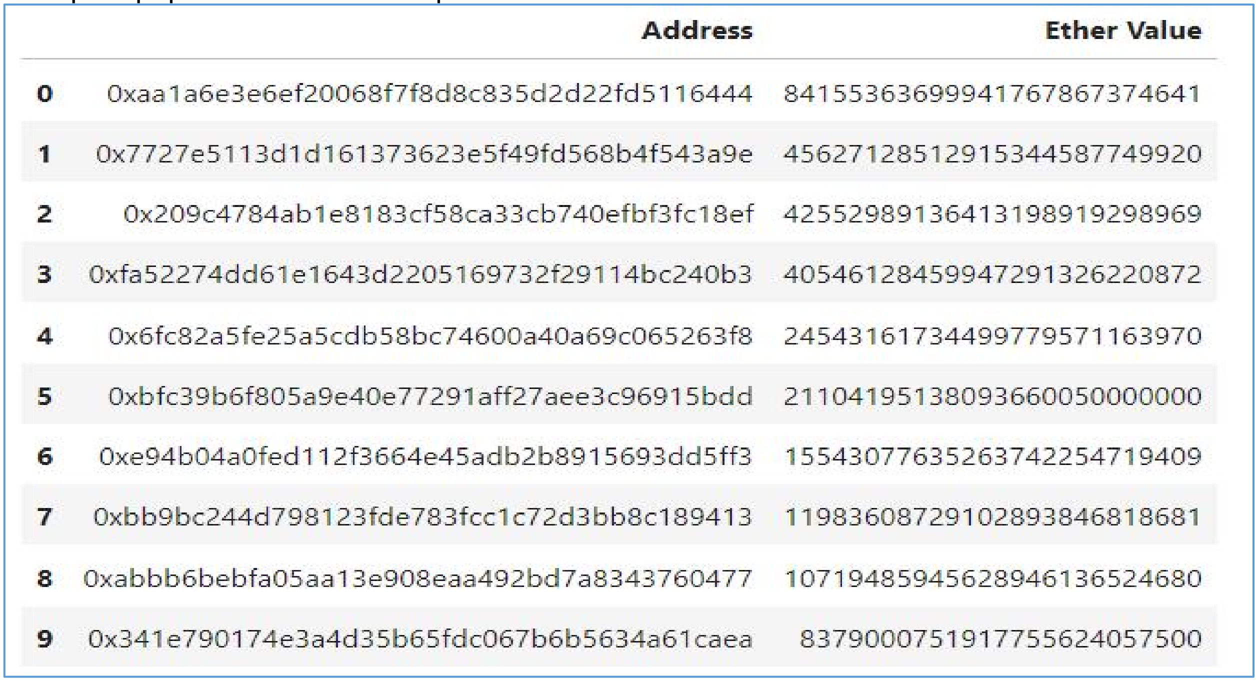
> To identify if most popular contracts extracted from Part B use more or less than average gas:

1. From the transaction file, fetch timestamp, gas and address and remove any missing values or unreliable data.
2. From the contracts file, fetch address.
3. From both files, separate RDD were created using ‘map’.
4. These were then joined using address as the common key so that only addresses of contracts are considered.
5. For each contract address, total gas and average gas is calculated and compared with total average gas that has been calculated separately and given below:



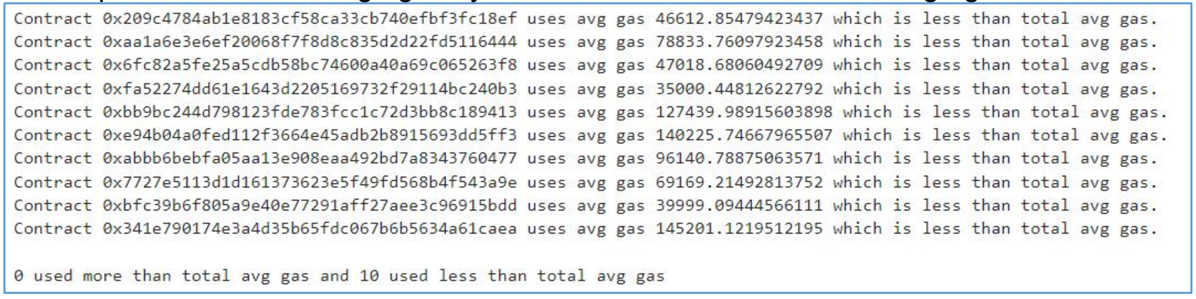
```
['avg:161725.69962368018']
```

6. Top 10 popular contracts from part B:



	Address	Ether Value
0	0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444	84155363699941767867374641
1	0x7727e5113d1d161373623e5f49fd568b4f543a9e	45627128512915344587749920
2	0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef	42552989136413198919298969
3	0xfa52274dd61e1643d2205169732f29114bc240b3	40546128459947291326220872
4	0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8	24543161734499779571163970
5	0xbfc39b6f805a9e40e77291aff27aee3c96915bdd	21104195138093660050000000
6	0xe94b04a0fed112f3664e45adb2b8915693dd5ff3	15543077635263742254719409
7	0xbb9bc244d798123fde783fcc1c72d3bb8c189413	11983608729102893846818681
8	0xabbb6bebfa05aa13e908eaa492bd7a8343760477	10719485945628946136524680
9	0x341e790174e3a4d35b65fdc067b6b5634a61caea	8379000751917755624057500

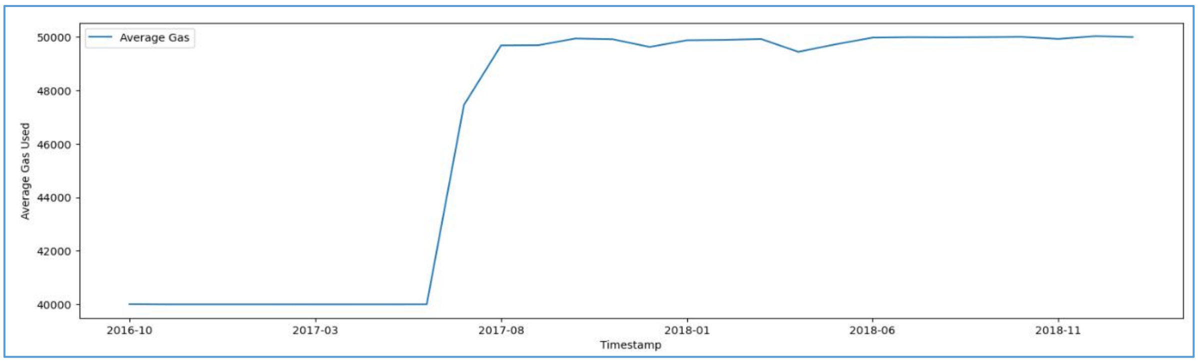
7. Comparison of the average gas by these contracts vs the total average gas.



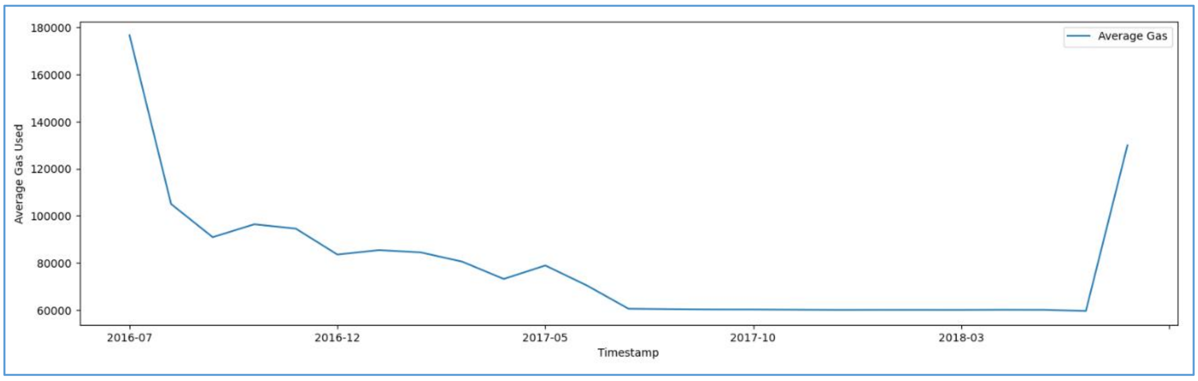
```
Contract 0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef uses avg gas 46612.85479423437 which is less than total avg gas.
Contract 0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444 uses avg gas 78833.76097923458 which is less than total avg gas.
Contract 0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8 uses avg gas 47018.68060492709 which is less than total avg gas.
Contract 0xfa52274dd61e1643d2205169732f29114bc240b3 uses avg gas 35000.44812622792 which is less than total avg gas.
Contract 0xbb9bc244d798123fde783fcc1c72d3bb8c189413 uses avg gas 127439.98915603898 which is less than total avg gas.
Contract 0xe94b04a0fed112f3664e45adb2b8915693dd5ff3 uses avg gas 140225.74667965507 which is less than total avg gas.
Contract 0xabbb6bebfa05aa13e908eaa492bd7a8343760477 uses avg gas 96140.78875063571 which is less than total avg gas.
Contract 0x7727e5113d1d161373623e5f49fd568b4f543a9e uses avg gas 69169.21492813752 which is less than total avg gas.
Contract 0xbfc39b6f805a9e40e77291aff27aee3c96915bdd uses avg gas 39999.09444566111 which is less than total avg gas.
Contract 0x341e790174e3a4d35b65fdc067b6b5634a61caea uses avg gas 145201.1219512195 which is less than total avg gas.

0 used more than total avg gas and 10 used less than total avg gas
```

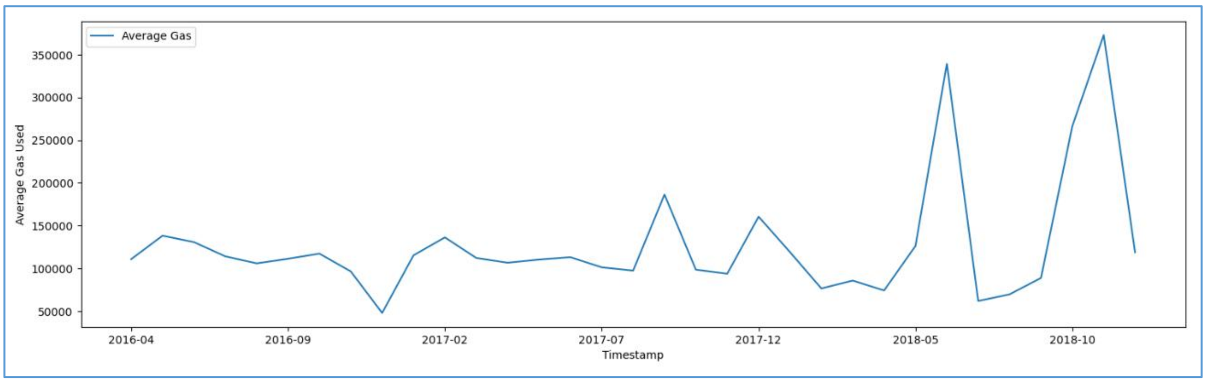
8. Plot of average gas used by contract over time:
0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef



9. Plot of average gas used by contract over time:
0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444



10. Plot of average gas used by contract over time:
0xbb9bc244d798123fde783fcc1c72d3bb8c189413



Conclusion:
The total gas price has peaked in 2018 but the average gas price has remained stable during the time denoting that number of transactions were high during 2018. The total gas used for contracts has increased considerably from 2017 to reaching highest in 2018 before falling again. Since more number of contracts were using gas during the year 2018, the average gas used is stable during the time. Comparing the average gas used by top 10 miners from Part B of the coursework, we can see that their gas used is generally less than total average gas. I have picked three contract addresses randomly to see their average gas used over time and as understood from previous statement it can be noticed that two of three have used more gas in the year 2018.