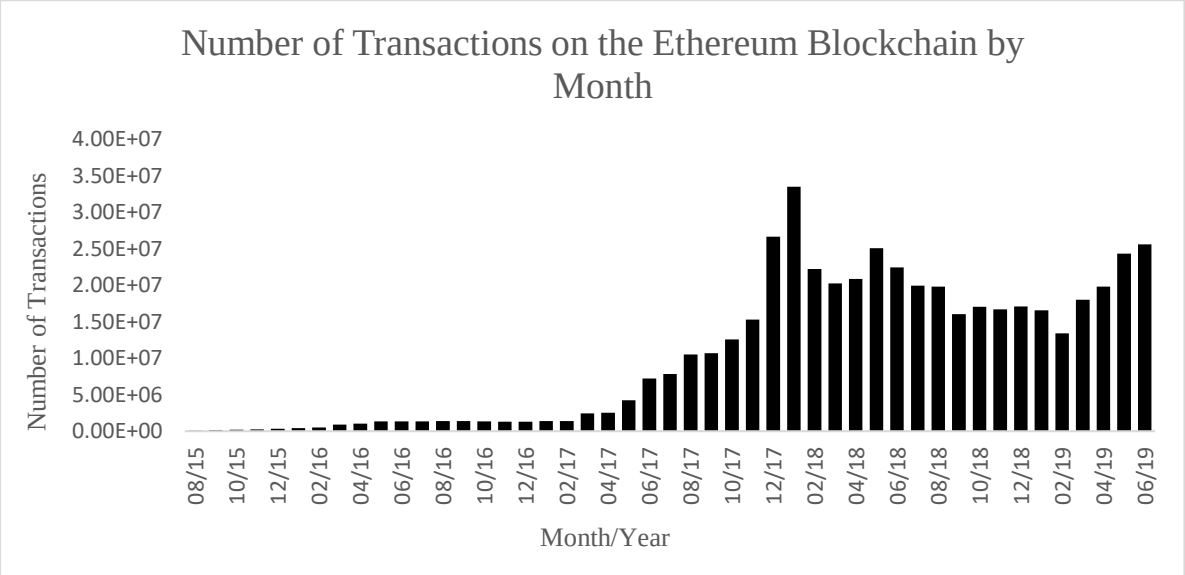
ECS765P Ethereum Analysis Coursework

Philip Wurzner
210930632

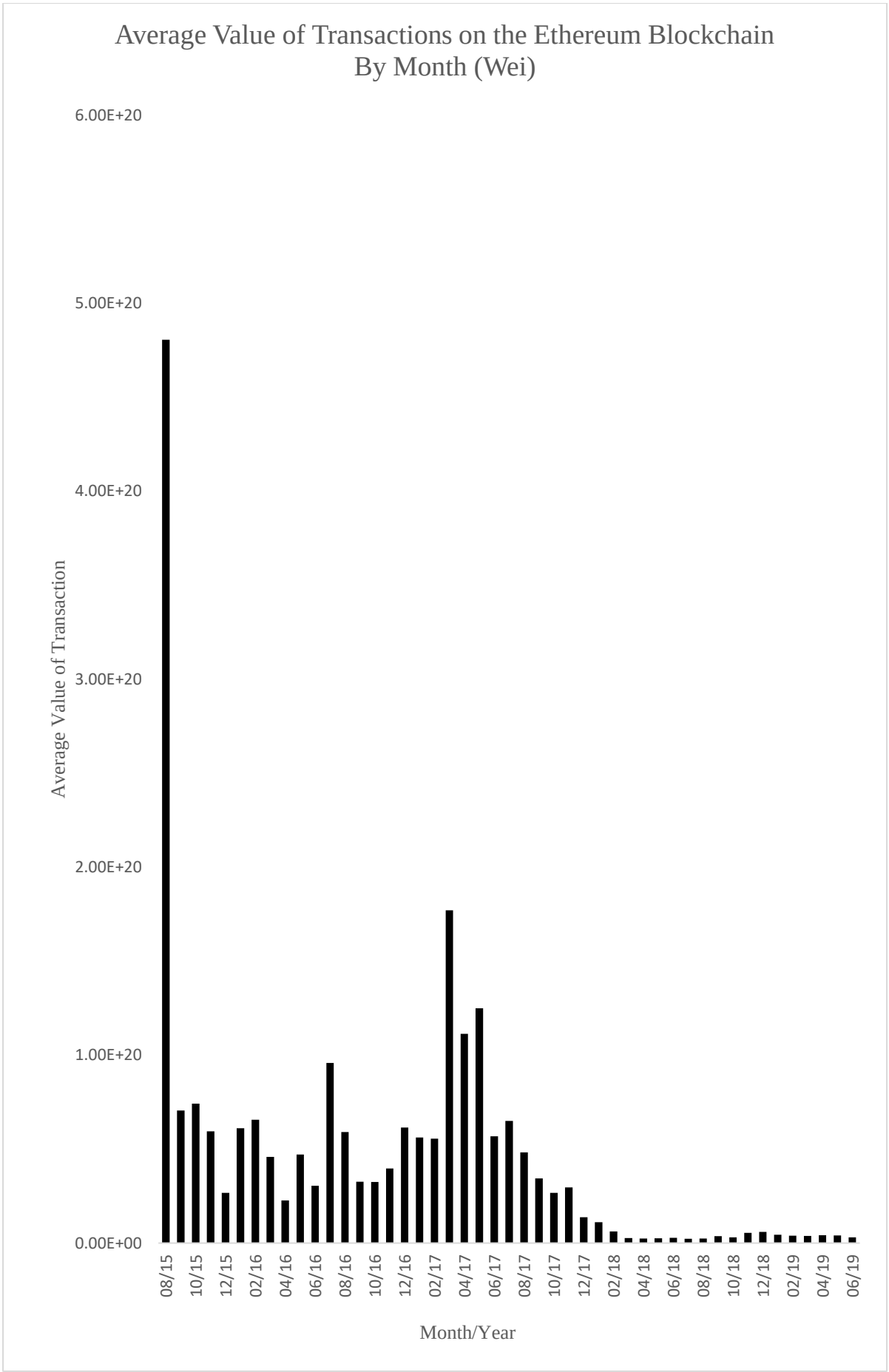
PART A

The first task was simply to create bar plots showing two key features: the average number of transactions happening on the blockchain each month and the average value of said transactions. To find these values, MapReduce was used and values from the "Transactions" database on the HDFS was queried. To find the number of transactions that occurred, the mapper in **Part_A_1.py** simply collected the year and month, and then mapped these with a value of 1. Every instance of 1 was then summated in the reducer process to find the total amount of transactions. Figure 1 displays the result of the Part_A_1.py. The output file can also be found in the attached folder and GitHub. As can be seen, the number of transactions rose rapidly around mid-2017, and then corresponded with how the Ethereum price was performing.



The average value of each transaction was calculated in a similar manner in the file **Part_A_2.py**. The mapper now also attached the value of each transaction to its output, however. The reducer then sets up variables for the total amount of transactions (*count*) and the total value transacted (total_value) in the month. Each transaction found in the month then iterates over a loop that adds their count and transaction variable. This transaction variable is then divided by the number of transactions to get the average transaction value for the month. The result can be seen on the next page.

The mean value of transactions was incredibly high in the earlier phases of Ethereum and experienced a peak in early 2017 but has been declining since. This may suggest that lower amounts of money are being traded, perhaps as a result of increased interest in crypto by non-institutional investors, who tend to invest smaller sums of money, as well as the relative supply and demand curve adjusting to make wei more valuable in terms of Fiat money.


Average Value of Transactions on the Ethereum Blockchain
By Month (Wei)
6.00E+20
5.00E+20
4.00E+20
3.00E+20
2.00E+20
1.00E+20
0.00E+00
Average Value of Transaction
08/15
10/15
12/15
02/16
04/16
06/16
08/16
10/16
12/16
02/17
04/17
06/17
08/17
10/17
12/17
02/18
04/18
06/18
08/18
10/18
12/18
02/19
04/19
06/19
Month/Year

PART B

For part B the top 10 highest value smart contracts were identified. This was done using an MRJob approach with MapReduce. The first program, **Part_B_1.py**, iterates through the transaction dataset and aggregates each receiving adress with the total amount of Wei that has been sent to it. The resulting output (containing fake results) would look something like the table shown below.

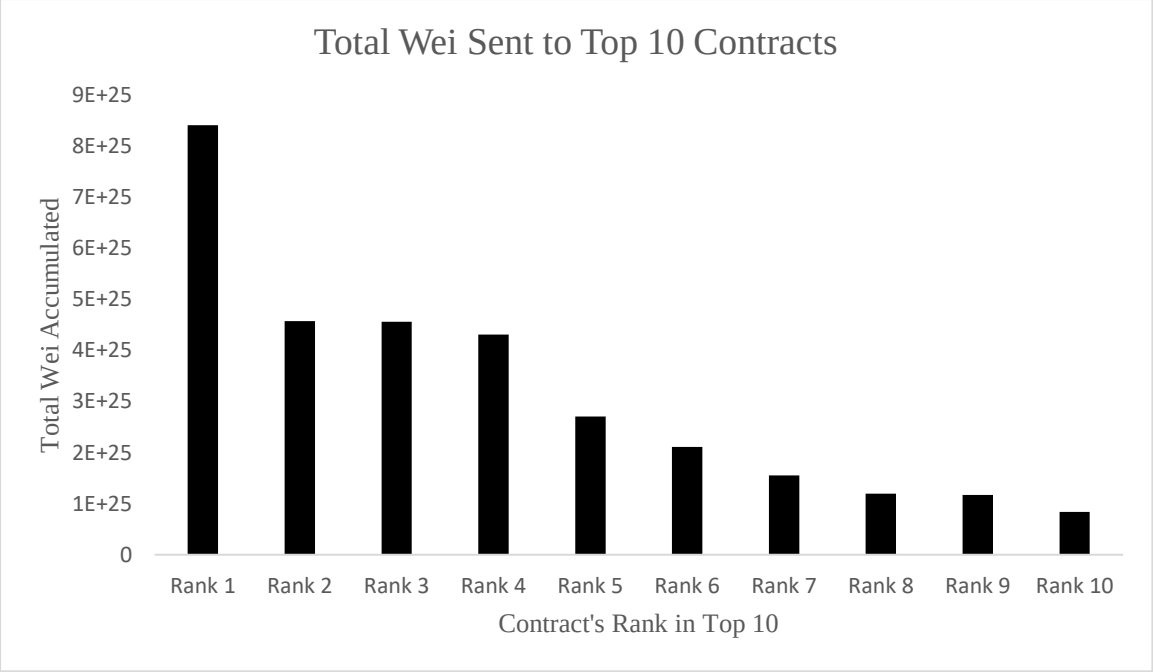
Destination address	Total value sent to address
"0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444"	1234444444444
"0x0001a3c6be5a99c2fd89eb00199cf426d20d5acd"	40000000000
"0x0002325fcaaac6ebf1254a626589147bde1a2394"	1200000000
…	…

The next program, **Part_B_2.py**, requires two inputs and preforms a repartition join operation. The program joins the addresses, with their associated values from part 1, with verified Smart Contract addresses from the Contract database. This in turn creates an output very similar to the one created in the first part, but now only contracts are present in the output.

Verified Contract address	Total value sent to contract
"0x0001a3c6be5a99c2fd89eb00199cf426d20d5acd"	250000000000000000
"0x0001ad65c82974d70889d74246e33e391e2d7903"	24545119820000000000
"0x00028c3455500c4458b1a153c536645aaad5b6d0"	1153574200000000000
…	…

The third and final program, **Part_B_3.py**, sorts and filters the repartition joined results based on the total value sent to each smart contract. Below the top 10 most valuable contracts can be seen in both table and figure form.

Rank	Top 10 Addresses	Total value sent to contract
1	"0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444"	84155100809965865822726776
2	"0xfa52274dd61e1643d2205169732f29114bc240b3"	45787484483189352986478805
3	"0x7727e5113d1d161373623e5f49fd568b4f543a9e"	45620624001350712557268573
4	"0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef"	43170356092262468919298969
5	"0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8"	27068921582019542499882877
6	"0xbfc39b6f805a9e40e77291aff27aee3c96915bdd"	21104195138093660050000000
7	"0xe94b04a0fed112f3664e45adb2b8915693dd5ff3"	15562398956802112254719409
8	"0xbb9bc244d798123fde783fcc1c72d3bb8c189413"	11983608729202893846818681
9	"0xabbb6bebfa05aa13e908eaa492bd7a8343760477"	11706457177940895521770404
10	"0x341e790174e3a4d35b65fdc067b6b5634a61caea"	837900075191775562405750 0



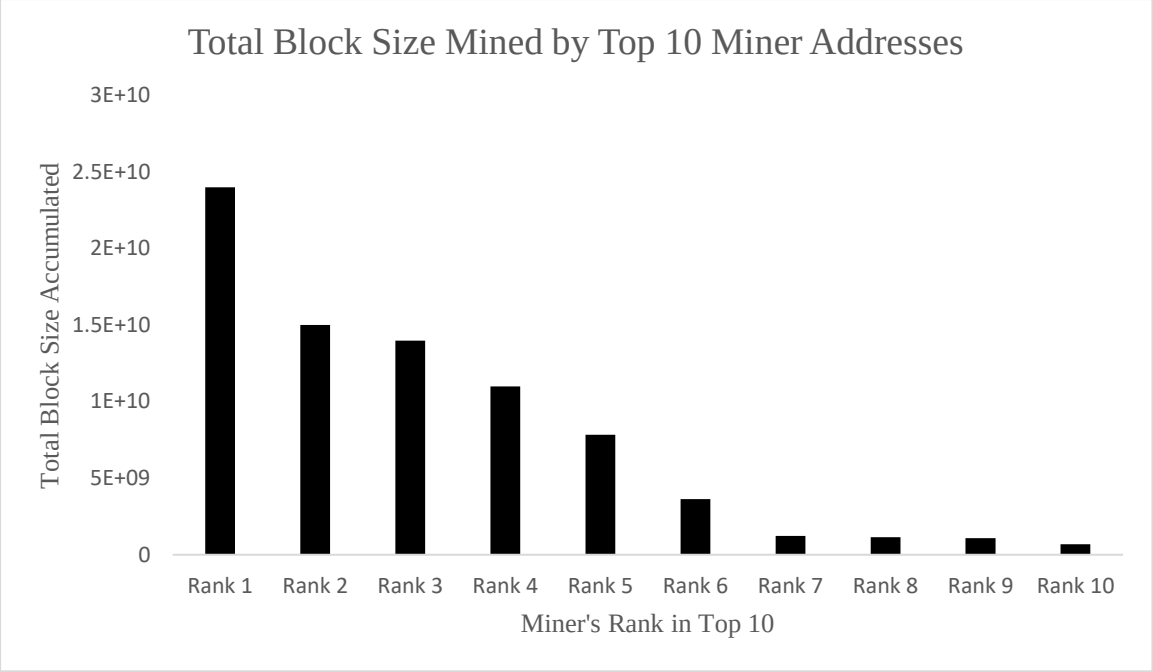
PART C

For part C the top 10 miners, sorted by total block size mined, had to be found. The method used to do so was MapReduce. The first program, **Part_C_1.py**, used the blocks dataset and simply combined the miner's adress with the aggregated size of each block mined by the address. Below an example output for the first program can be seen.

Miner Adress	Size of the Block
"0x000083ceb2317f5755be7a745e3c4be7ba396877"	2362
"0x000354c8e89af5a4ea0be126176a14848678a391"	2197
…	…

The second program, **Part_C_2.py,** then took the aggregated list of miners and sorted /filtered them based on the total size of the blocks they have mined. Below the table and figure with results can be seen.

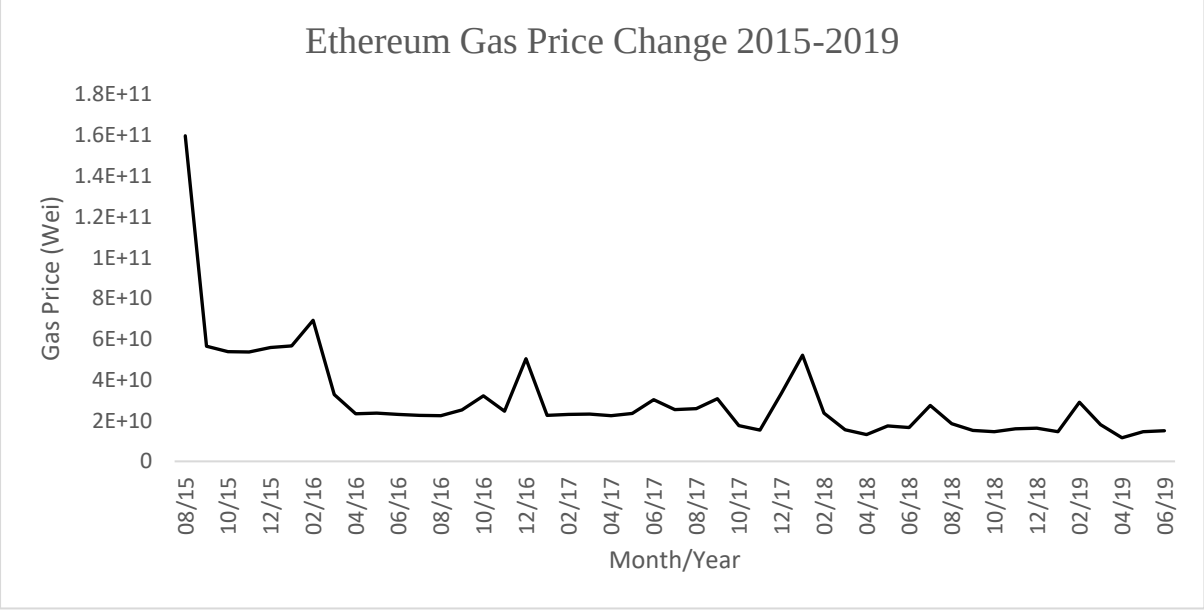
Top 10 Miner Addresses	Total Block Size Mined
"0xea674fdde714fd979de3edf0f56aa9716b898ec8"	23989401188
"0x829bd824b016326a401d083b33d092293333a830"	15010222714
"0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c"	13978859941
"0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5"	10998145387
"0xb2930b35844a230f00e51431acae96fe543a0347"	7842595276
"0x2a65aca4d5fc5b5c859090a6c34d164135398226"	3628875680
"0x4bb96091ee9d802ed039c4d1a5f6216f90f81b01"	1221833144
"0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb"	1152472379
"0x1e9939daaad6924ad004c2560e90804164900341"	1080301927
"0x61c808d82a3ac53231750dadc13c777b59310bd9"	692942577



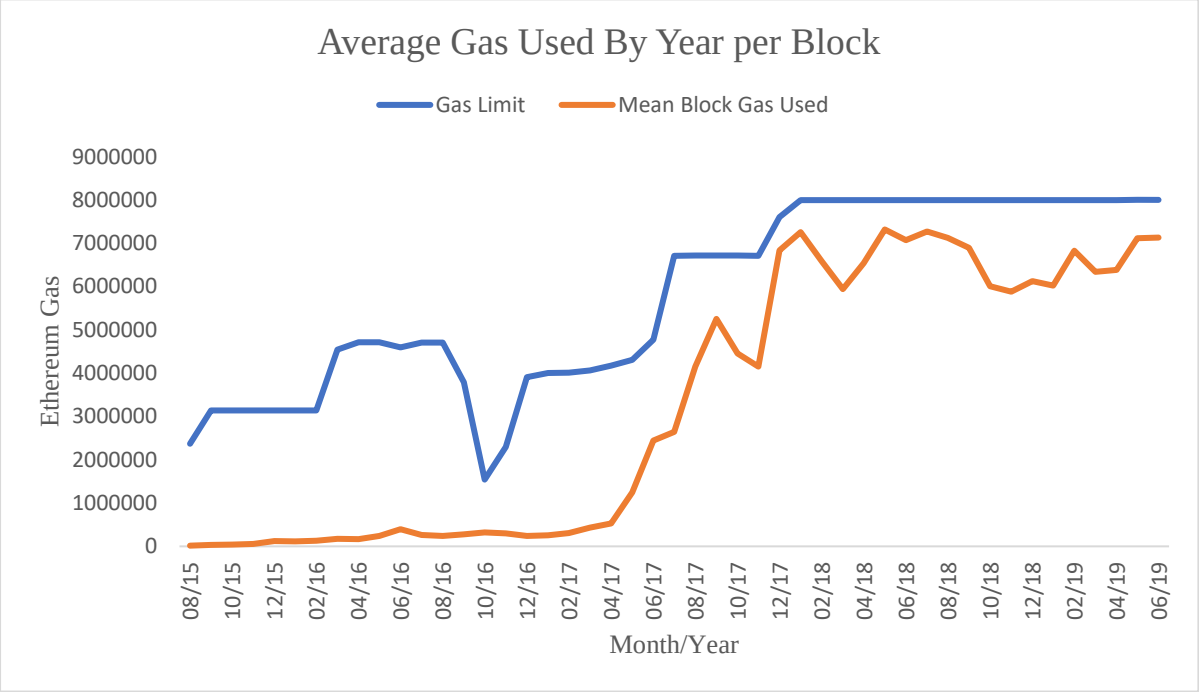
Part D

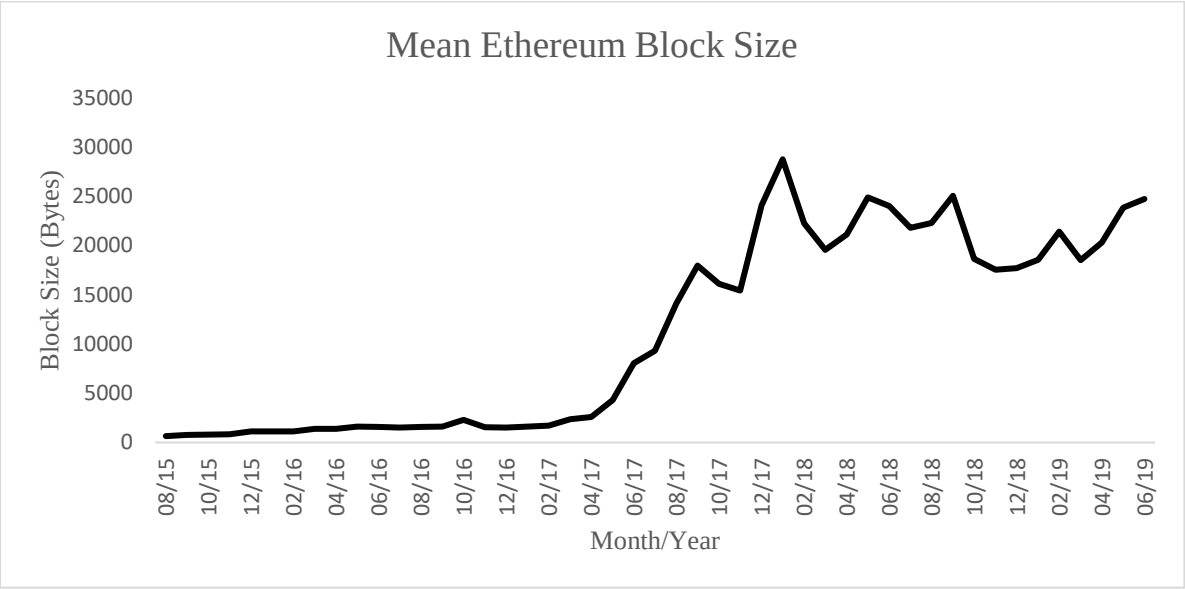
GAS GUZZLERS

The Gas Guzzler's question asked for three key pieces of information. The first of these was how the Ethereum gas prices have changed over the years. Using MapReduce, the **Part_D_Gas_1.py** program queried the transaction database and mapped the date in month and years to the price of gas, and then reduced the results by averaging the total aggregated cost of gas over month using a count variable. Below is the resulting graph. It can clearly be seen that gas prices have fallen dramatically since 2015 and still sees occasional upticks.

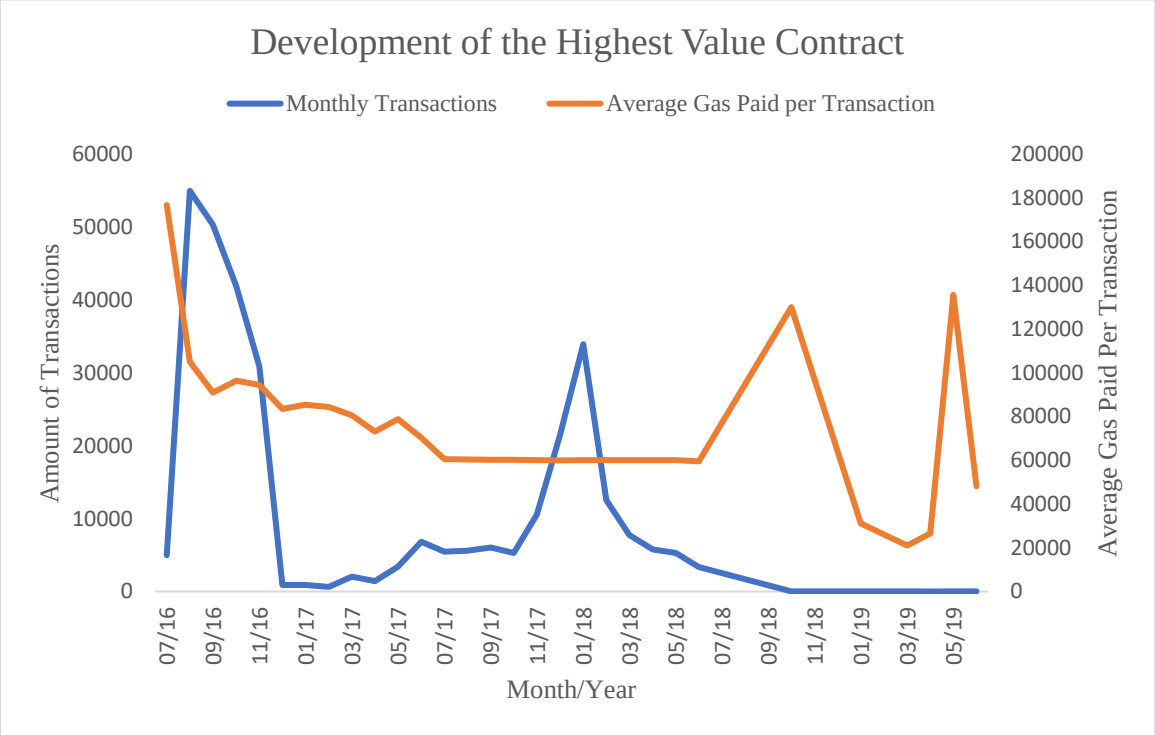


The second requested piece of information, provided by the **Part_D_Gas_2.py** program, was whether contracts have become more complicated, and require more gas as time has moved on. By querying the block dataset, the average block size and block gas used per month have been retrieved. The mean block gas used has rose steadily in early 2017, until the Gas limit was reached and has plateaued since. The mean block size has also increased along with it, clearly demonstrating the increasing complexity of the smart contracts associated with these blocks.



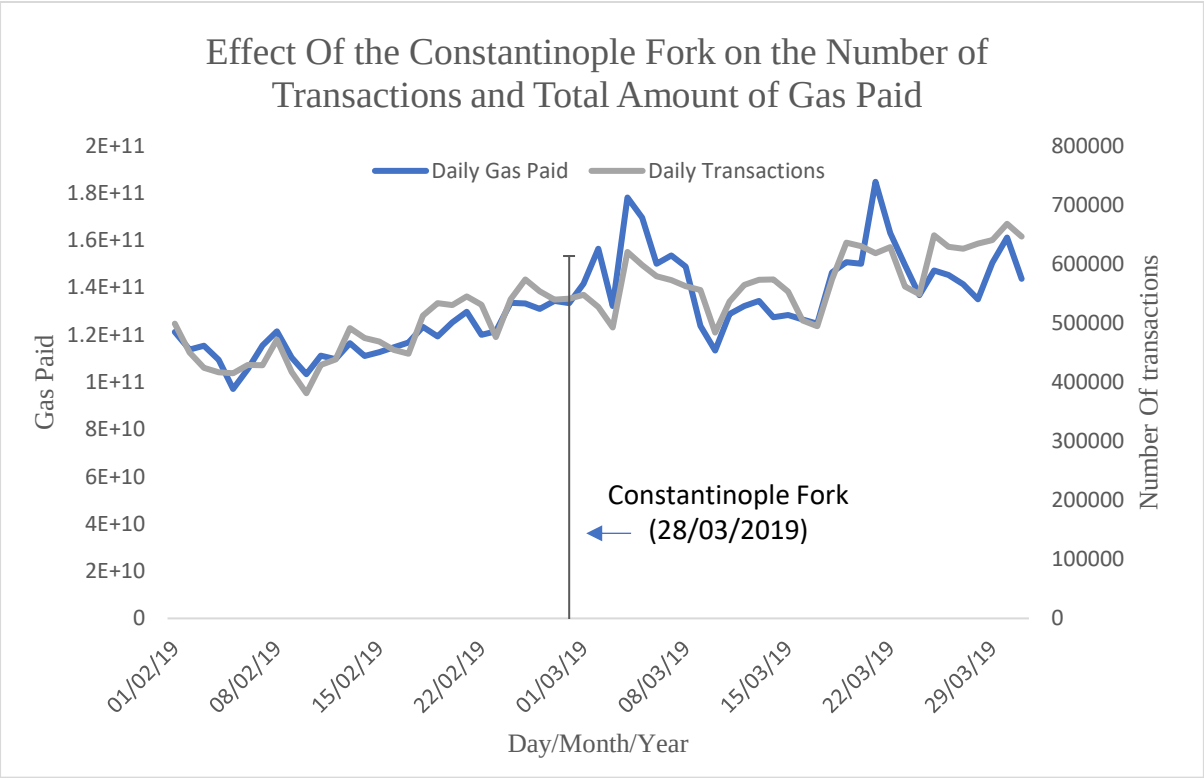


Finally, the third information request was showing how the complexity of one of the top 10 contracts has changed over time. Using MapReduce in the Part_D_Gas_3.py program, the data below was collected to show has the most valuable smart contract, 0xaa1a6e3e6…, has preformed since 2016. The number of transactions spiked in 2016 and 2018, but the average gas paid per transaction (complexity) has not correlated strongly with these spikes in transaction amounts. While it may appear that the gas fees paid by the service have increased since mid-2018 It should be noted that the transaction amounts are very low in that timespan, and therefore not particularly representative of a real trend in the smart contract.



Fork Analysis

The fork analysis was done on the Constantinople fork, which occurred in late February 2019. This fork provided stability and optimised gas costs, which is why **Part_D_Fork.py** looks into gas related cost and user data from around the time the fork was implemented. The program works by querying data from the transactions database, and filters out results which did not take place in 2019 between February and March.



As can be expected from a relatively small update, the fork did not have a substantial impact on either the use of Ethereum, nor the total amount of daily gas paid. The Constantinople fork's gas optimisation specifically targeted certain actions in the Ethereum Virtual Machine (EVM) and it can clearly be seen that these optimisations did not trickle down to most users, otherwise a divergence between the total amount of transactions and total amount of gas paid would be seen on the graph.