Queen Mary
University of London

Specific Learning Differences Cover Note

Student ID: 190076421

Guidelines for markers assessing coursework and examinations of students diagnosed with Specific Learning Differences (SpLDs).

As far as the learning outcomes for the module allow, examiners are asked to mark exam/essay scripts sympathetically, ignoring the types of errors that students with SpLDs make and to focus on content and the student's understanding of the subject.

SpLDs may affect student performance in written work in the following ways:

- Spelling, grammar and punctuation may be less accurate than expected
- Organisation of ideas may be confused, affecting the overall structure of written work, despite including appropriate and interesting content
- Proof reading may be weak with some errors undetected, particularly homophones and homonyms which can avoid spell checkers

Under examination conditions, these difficulties are likely to be exacerbated, particularly towards the end of scripts.

When marking, please ensure the following guidance is adhered to:

- Scan the text for content to gain an overview of the work, to avoid making a judgement based on the structure alone or any minor spelling and grammar errors
- Do not comment solely on spelling and grammar issues, SpLD students are usually aware of this already. If you feel this is a major problem which has affected the sense of the writing, then please encourage the student to use their study skills support from DDS to help with this.
- Include positive and constructive comments amongst the feedback so that students can acknowledge their strengths but work with specialist study skills tutors on developing new strategies for areas of difficulty.
- Use clear English providing specific examples of what is right or wrong, to ensure that the student knows what to improve upon for future work

Colleagues in Schools and Institutes are asked to encourage students with specific learning differences to access the support provided by the Disability and Dyslexia Service if they have any concerns about a students work.

For more information regarding marking guidelines see the Institutional Marking Practices for Dyslexic Students on the DDS webpage.

Disability and Dyslexia Service
http://www.dds.qmul.ac.uk/

Disability and Dyslexia Service
Room 3.06, Francis Bancroft Building
Queen Mary University of London
Mile End Road
London
E1 4NS

T: +44 (0) 20 7882 2756
F: +44 (0) 20 7882 5223
E: dds@qmul.ac.uk
W: www.dds.qmul.ac.uk

Alteration or misuse of this document will result in disciplinary action

PART A. TIME ANALYSIS

JOB 1 AGGREGATION

Code: BD > A > BDA1.py

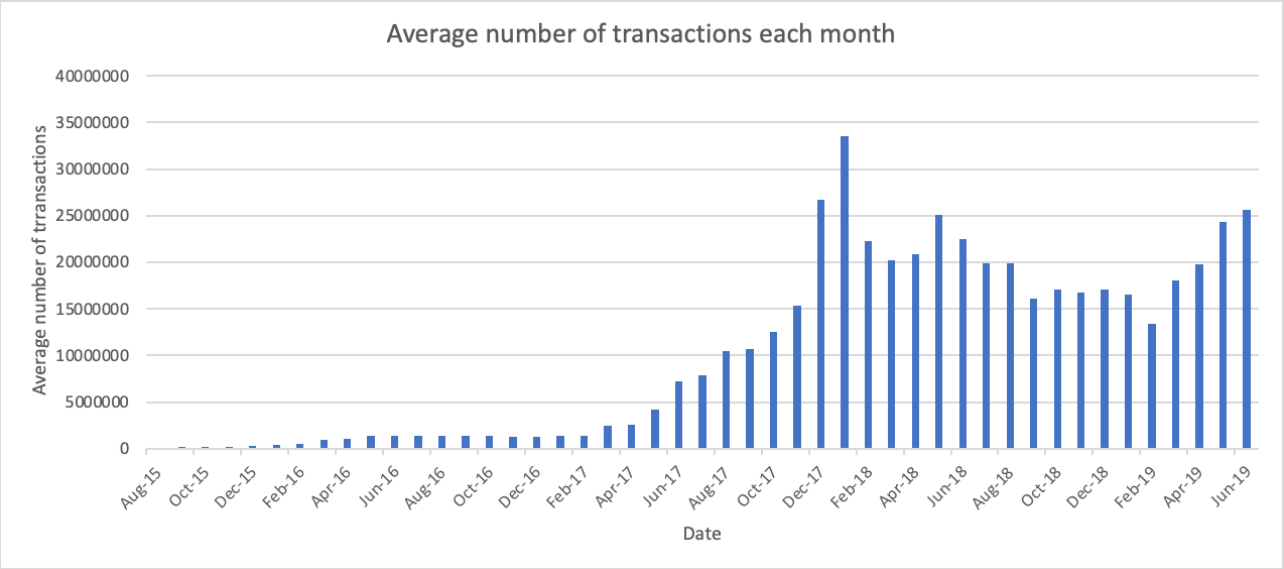
The input data is the transactions file.

Inside the mapper, we yield the time from epoch converted to month year using python module datetime and 1 since each line is one transaction.

We introduce a combiner to reduce the amount of data sent to the cluster. This is done by summing all values with the same key at the particular node and then yielding it, thus reducing the time it takes to complete the job.

Then the reducer adds all values with the same key using pythons built-in function sum and yields the month year pair with the total number of transactions during that month.

RESULTS



JOB 2 CALCULATING AVERAGE

Code: BD > A > BDA2.py

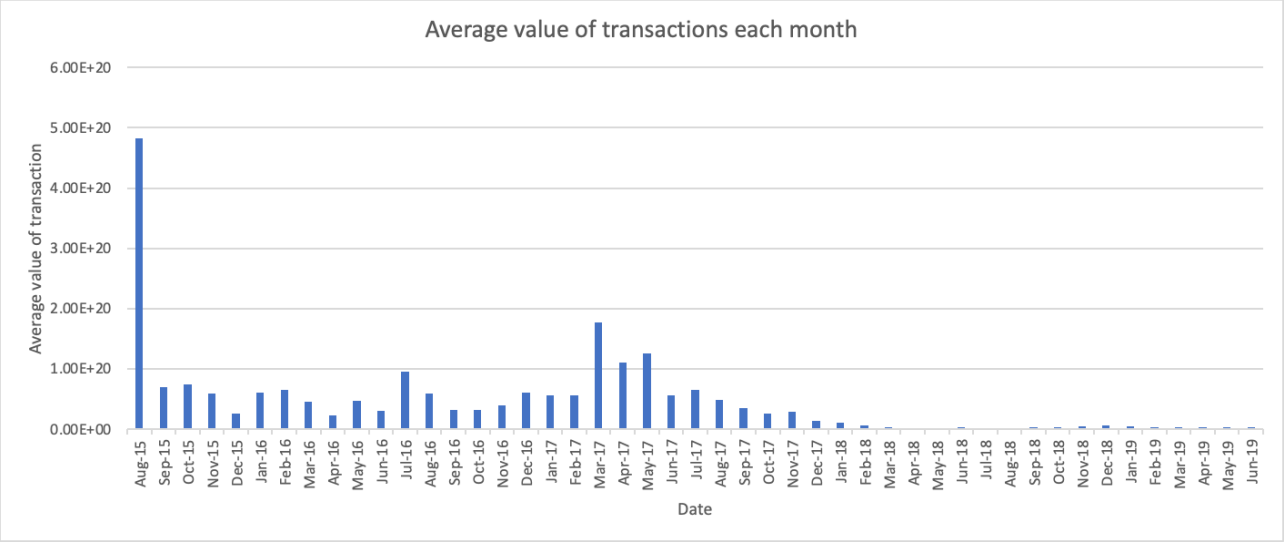
The input file is transactions.

In the mapper, we yield the month year pair and the value.

In the combiner, we yield the key (month year pair) and tuple. The first value inside the tuple is the number of transactions at the particular node during the particular month and year. The second value inside the tuple is the number of transactions during that time.

In the reducer, we can finally calculate the average for the particular month year pair by adding all the totals calculated by the reducer and dividing them by the total number of the transactions in that month and year. We then yield the month year pair and the average number of transactions per month and year.

RESULTS



PART B. TOP 10 MOST POPULAR SERVICES

JOB 1 - INITIAL AGGREGATION

Code: BD > B > BDB1.py

The input file is transactions.

In the mapper, we yield to_address and the value that got parsed to an integer.

Then in the combiner, we yield to_address and the sum of all values at the node associated with the to_address.

Then in the reducer, we calculate the total of all the sums from the reducer for each to_address, and we yield it.

JOB 2 - JOINING TRANSACTIONS/CONTRACTS AND FILTERING

Code: BD > B > BDB2.py

The input files are the output of job 1 and contracts.

In the mapper, if the line is from the job 1 output, we yield to_address as key, and tuple, “T” to indicate it is a transaction and the total value of the transactions for that address. If the line is from contracts, we yield address and tuple with one value “C” to indicate it is the contract.

In the reducer, we check that it is a contact for the given address, and there have been transactions to that address. If the number of transactions for the given address is non-zero and the address exists in contracts, we yield to_address and total value.

JOB 3 - TOP 10

Code BD > B > BDB3.py

The input file is the output of job 2.

The mapper yields None as a key (to ensure all values end up in the same reducer at once), and tuple to_address and the total value of the transactions for that address.

In the reducer, we sort all the values in descending order based on the total value and yield first ten elements in the array. We yield to_address and its corresponding total value.

RESULTS

to_address	total value
0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444	84155100809965865822726776
0xfa52274dd61e1643d2205169732f29114bc240b3	45787484483189352986478805
0x7727e5113d1d161373623e5f49fd568b4f543a9e	45620624001350712557268573
0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef	43170356092262468919298969
0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8	27068921582019542499882877
0xbfc39b6f805a9e40e77291aff27aee3c96915bdd	21104195138093660050000000
0xe94b04a0fed112f3664e45adb2b8915693dd5ff3	15562398956802112254719409
0xbb9bc244d798123fde783fcc1c72d3bb8c189413	11983608729202893846818681
0xabbb6bebfa05aa13e908eaa492bd7a8343760477	11706457177940895521770404
0x341e790174e3a4d35b65fdc067b6b5634a61caea	8379000751917755624057500

PART C. TOP TEN MOST ACTIVE MINERS

JOB 1. INITIAL AGGREGATION

Code: BD > C > BDC1.py

The input file is blocks.

The mapper yields miner and size as integer.

The reducer yields miner and the total size.

JOB 2. TOP TEN

Code: BD > C > BDC2.py

The input file it output of job 1.

The mapper yields None as key so all values go to one reducer at once so we can determine top ten most active miners. The value yielded is a tuple, miner and total size of block.

In the reducer, we sort all the values in descending order based on the total block size and yield first ten elements in the array. We yield to_address and its corresponding total value. We yield miner and the corresponding block size.

RESULTS

miner	total block size
0xea674fdde714fd979de3edf0f56aa9716b898ec8	23989401188
0x829bd824b016326a401d083b33d092293333a830	15010222714
0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c	13978859941
0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5	10998145387
0xb2930b35844a230f00e51431acae96fe543a0347	7842595276
0x2a65aca4d5fc5b5c859090a6c34d164135398226	3628875680
0x4bb96091ee9d802ed039c4d1a5f6216f90f81b01	1221833144
0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb	1152472379
0x1e9939daaad6924ad004c2560e90804164900341	1080301927
0x61c808d82a3ac53231750dadc13c777b59310bd9	692942577

PART D. DATA EXPLORATION

SCAM ANALYSIS

1.POPULAR SCAMS

PARSING SCAMS.JSON TO SCAMS.CSV

I modified the provided parser to extract id, category, status and addresses.

JOB 1 - JOINING TRANSACTIONS/SCAMS AND FILTERING

Code: BD > D > scams > BDD1.py

The input files are transactions and scams.

In the mapper, if the line is from the transactions, we yield to_address as key and value as tuple, value and “T” to indicate it is the transaction. If the line is from the scams, for each address we yield address as key and value as a tuple, category and “S” to indicate it is a scam.

In the reducer, we calculate the total that given address received through transactions. We also have two boolean variables to verify that the given transaction address is a scam and there was a transaction to the scam address. If the condition gets satisfied, we update the value of the corresponding boolean variable to True. We also need to determine what scam category the given address corresponds to. Each address can correspond only to one category of scams, so whenever we have a value from the scams dataset, we store the category. If both of our boolean variables are True, we yield the category and the total value sent to that address.

JOB 2 - AGGREGATION

Code: BD > D > scams > BDD2.py

The input file is output of job 1.

In the mapper we yield category and the value as integer.

In the combiner we yield the category as sum of values at that node for that category.

In the reducer yield category and total sum of all values for that category.

RESULTS

category	total value transferred (in Wei)
Scamming	3821810270167372638200
Fake ICO	135645756688962997967
Phishing	2519502571635775159088

The most lucrative type of scam is Scamming, then Phishing and then Fake ICO.

JOB 3 - JOINING TRANSACTIONS/SCAMS A FILTERING

Code: BD > D > scams > BDD3.py

The input files are transactions and scams.

In the mapper, if we have a line from transactions, we yield to_address as key and tuple with three values, date, “T” to indicate it is a transaction and value. If we have a line from scams, for each address, we yield address as key and value as a tuple, category and “S” to indicate it is a scam.

In the reducer for each transaction sending money to a scam, we yield the scam category as key and tuple, date and value. We have two boolean variables initially set to False to check whether the transaction is a scam and whether there was a transaction to a scam address. We also have an array to store information that we will later yield. If the value is from a transaction, we append the date and value to it and set one of the boolean variables to True. If we have a value from scams, we set the category of the scam and set the other variable to True. After we iterate through all values, if both boolean variables are set to True, we yield all the values from the array.

JOB 4 - AGGREGATION

Code: BD > D > scams > BDD3.py

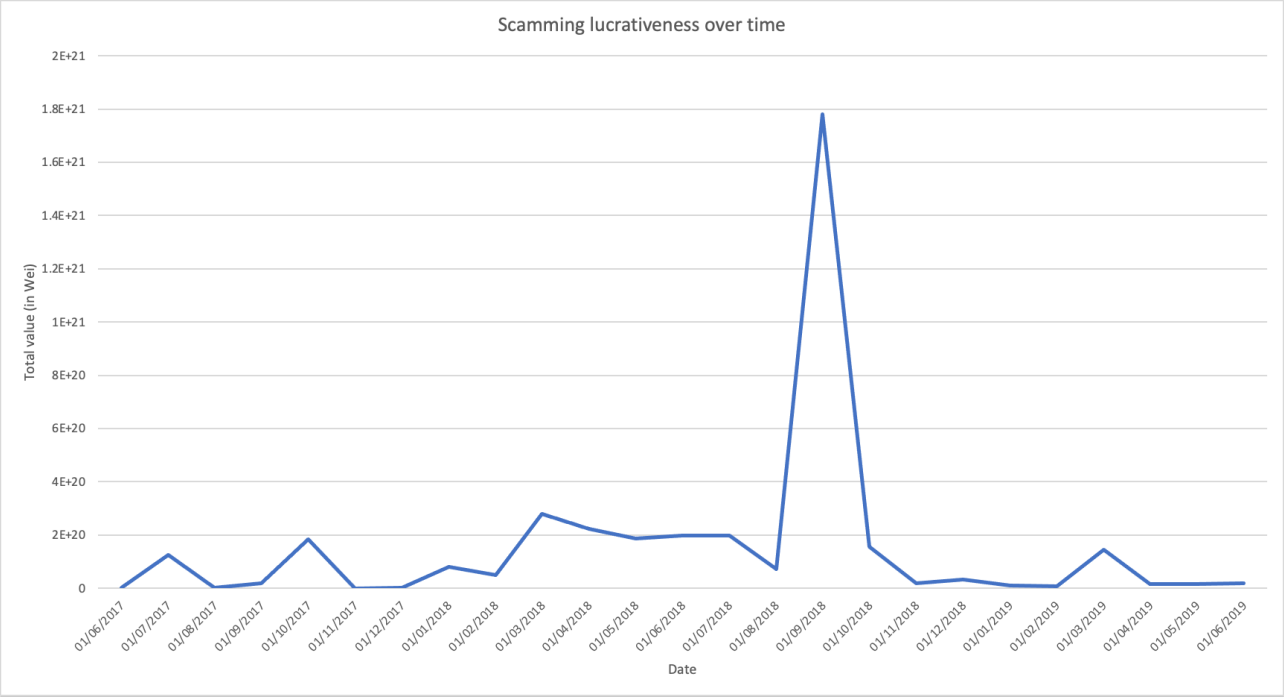
The input fie is output of job 3.

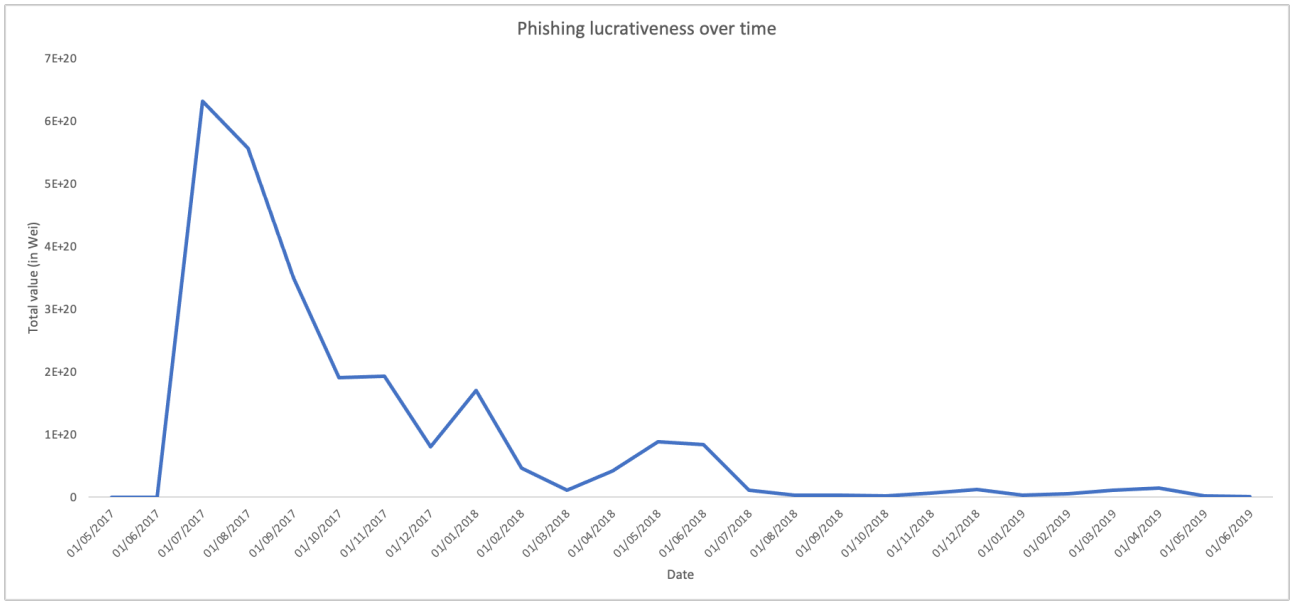
In the mapper we yield concatenation of category and date. The value yielded is value transferred to the scam address.

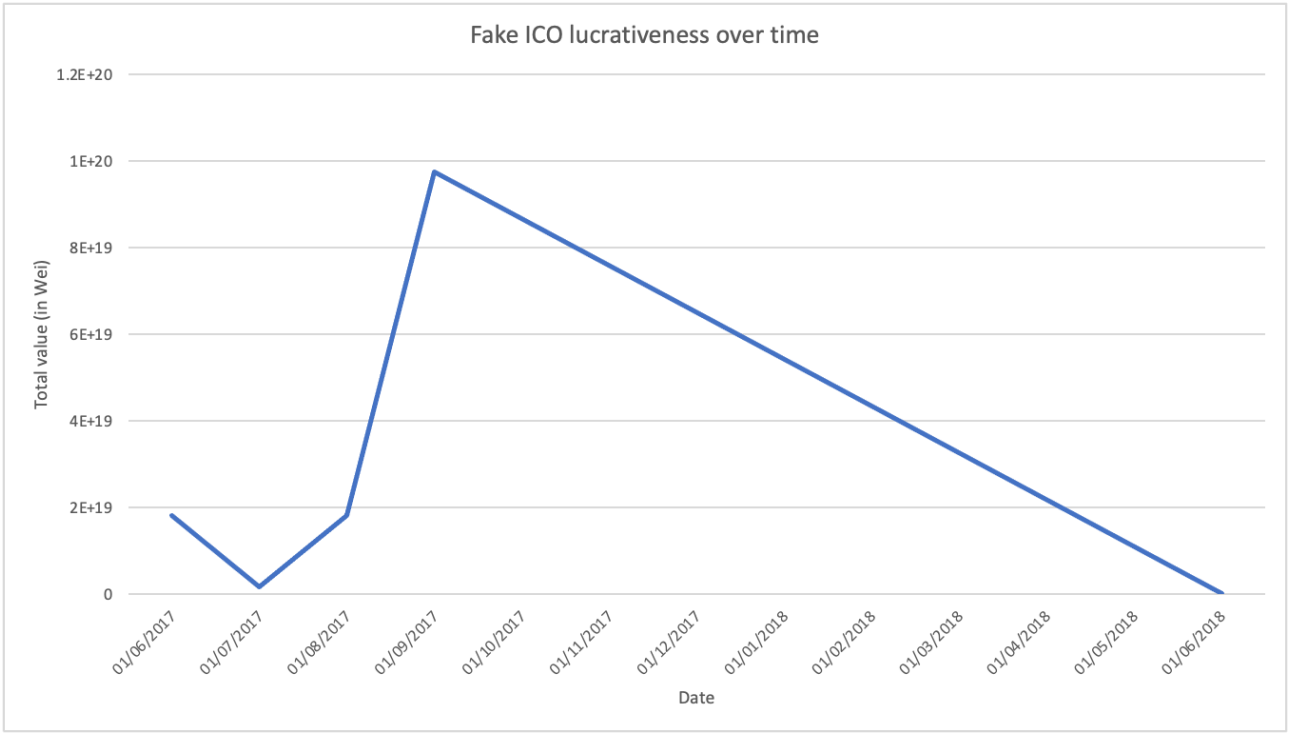
In combiner we yield the key and sum of values for that key at that node.

In reducer we yield key and total sum of values.

RESULTS







The most lucrative scam in 2017 was Phishing. Then at the beginning of 2018, Scamming overtook it when Phishing's lucrativeness dropped very low in March 2018. Then it bounced back up a bit, but its lucrativeness was again very low in July 2018. Scamming remained the most lucrative scam until November 2018, when it's lucrativeness became very low. After 2018 it received a tiny amount of Wei. There was a spike in March 2019, but it dropped again the very next month.

Fake ICO was never the most lucrative scam. In general, it received a tiny amount of Wei compared to the other two scams. It started gaining its popularity in August 2018 and reached its peak in September 2018, but then it started dropping. Even at its peak, it is pretty minor compared to the other two scams.

MISCELLANEOUS ANALYSIS

2. Gas Guzzlers

JOB 1 AVERAGE PRICE OF GAS PER DAY

Code: BD > D > gas > BDD1.py

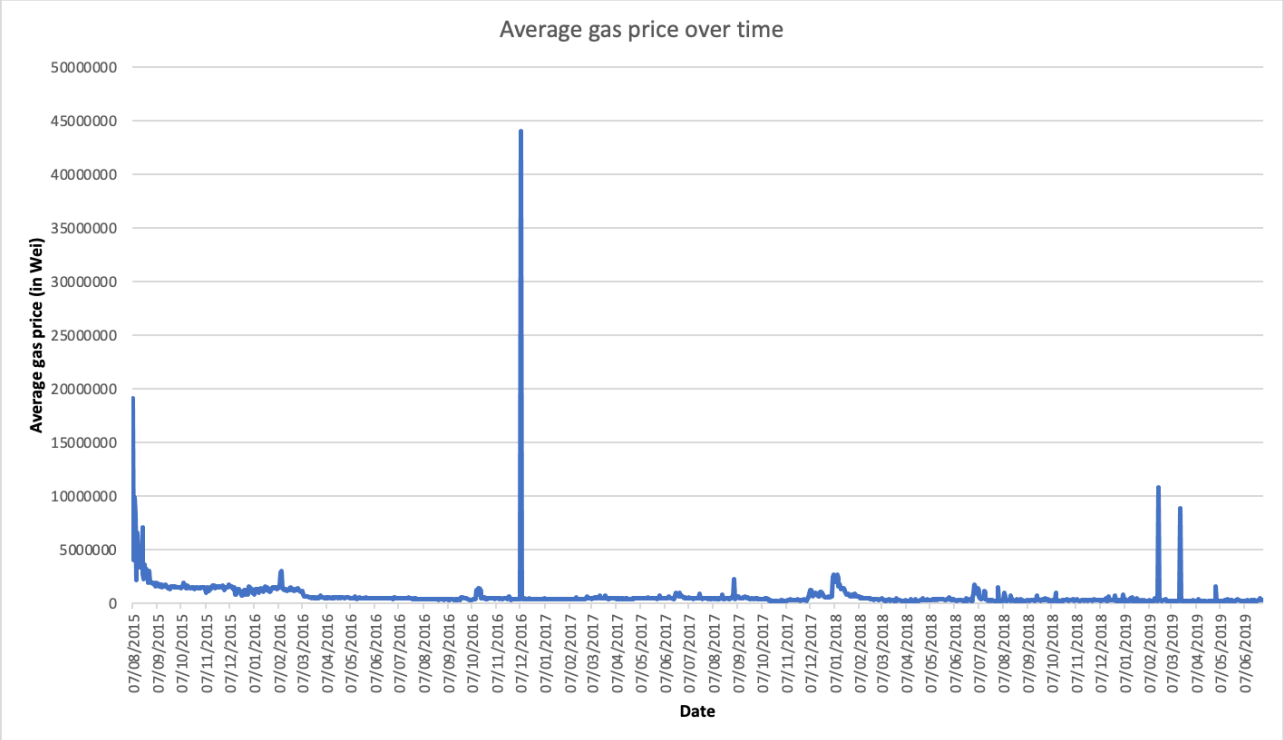
The input file is transactions.

In the mapper, we yield date as key and quotient of gas_price and gas (the price of 1 gas).

In the combiner, we yield date as key and tuple as values, a total of sums and number of values added.

In the reducer, we calculate the average and yield date and the average price of 1 gas for the given day.

RESULTS



Initially, the price gas price was relatively high; however, within a month, it dropped significantly. Then it continued to drop with a few minor spikes periodically over the next few months. In December 2016, it spiked up, and it reached its all-time high; however, it quickly dropped back. There was a small spike for a short time in September 2017. There was a small spike for a bit longer in December 2017, but it started dropping in January 2018. There was a series of small spikes between July 2018 and October 2018. There were a few tiny spikes between December 2018 and February. In February 2019, there was a more significant spike, but it quickly dropped down. However, there was another one in March 2019 and another much smaller one at the end of April 2019.

JOB 2 - JOINING TRANSACTIONS/CONTRACTS

Code: BD > D > gas > BDD2.py

The input files are transactions and contracts.

In the mapper, if the line is from transactions, we yield key as to_address and tuple containing two values date and gas. If the line is from contracts, we yield address as key and tuple containing one value “C” that works as an identifier.

We yield date and gas in the reducer, assuming a transaction is a contract. To achieve this, we iterate through values. If we have a value from contracts, we set the boolean variable to True. If we have value from transactions, we append the values to the array. After we iterate through all of the values, we then check the value of our boolean variable. If it is False, we yield nothing. If it is True, we yield all elements inside the array, date and gas.

JOB 3 - CALCULATING AVERAGE GAS NEEDED PER CONTRACT

Code: BD > D > gas > BDD3.py

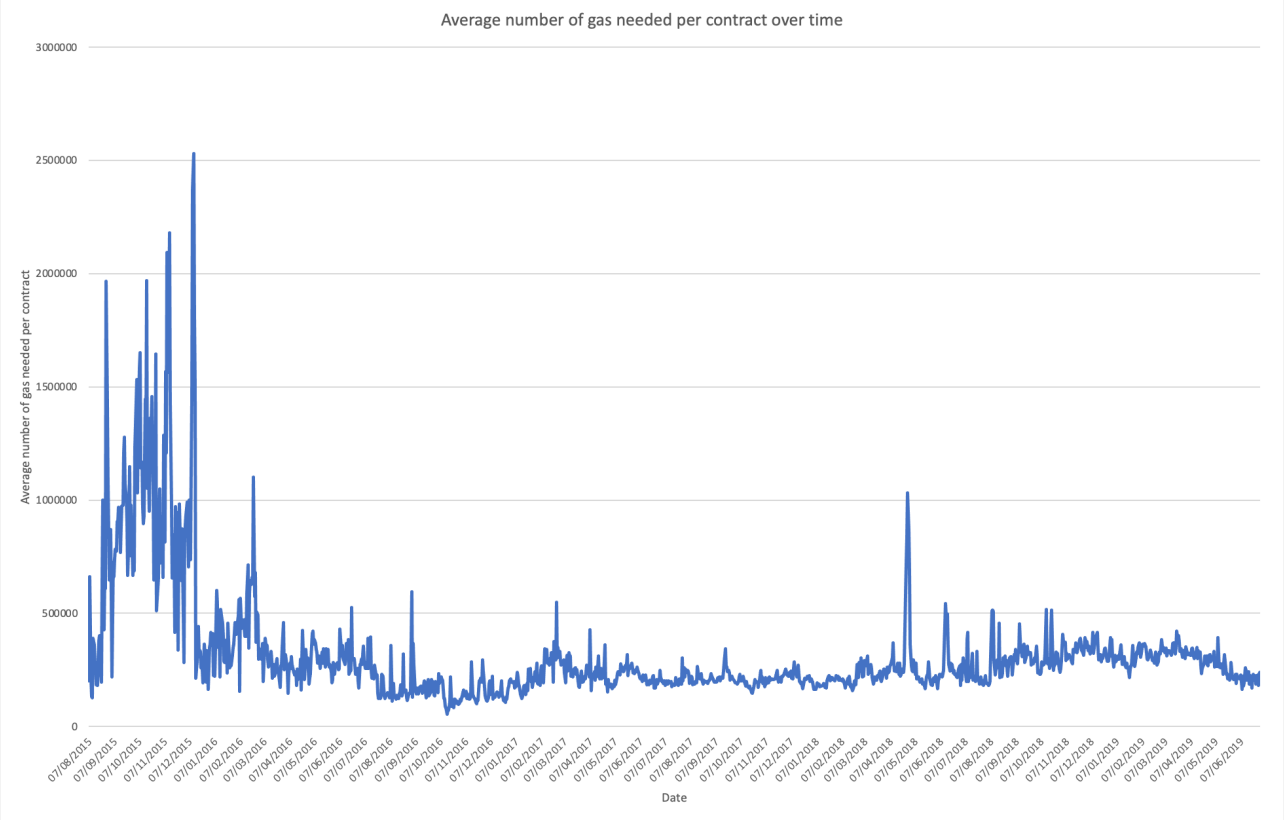
The input file is output of job 2.

In the mapper we yield the date as key and the gas as value.

In the combiner we yield date as key and tuple of values, sum of gas and number of values.

In the reducer we yield the date as key and the average number of gas per contract.

RESULTS



Initially, the number of gas needed to contract was relatively low; however, it quickly spiked up and reached its all-time peak in December 2016. Then it rapidly drops down and goes slightly back up for a short amount of time between January 2016 and February 2016, and then it drops by a lot again. The number of gas keeps oscillating; however, it generally decreases, reaching its lowest point in October 2016. Then the trend changes, and the number of gas needed per contract increases. There is another more significant spike in May 2018, but it drops down very quickly once again. After that, there are a few more notable but still relatively small spikes in June 2018, July 2018, August 2018 and November 2018. Then the number of gas needed keys increased; however, in April 2019, the trend changed again, and it started dropping. Overall the contracts with the time required less gas than initially.

JOB 4 - JOINING TRANSACTIONS/OUTPUT OF PART B

Code: BD > D > gas > BDB4.py

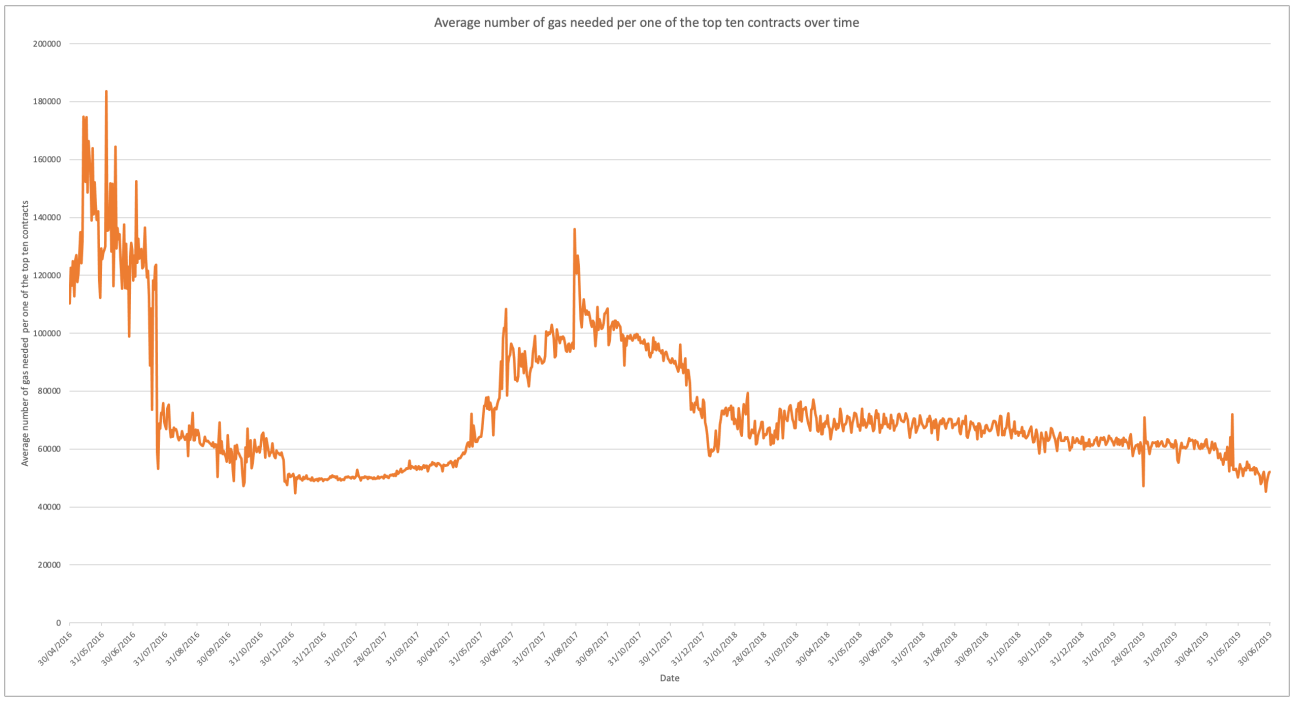
This job is exactly the same as job 2. The only difference is that now the input is transactions and output of part B, as a result we had to adapt the mapper to yield correct field however it still yields the same values.

JOB 5 - CALCULATING AVERAGE GAS NEEDED PER TOP TEN CONTRACT

Code: BD > D > gas > BDB3.py

For this job we reused the code from job 3. The only difference is the input file is now output of job 4.

RESULTS



The graph initially follows the same trend as the graph for all contracts. It quickly grew and reached its all-time peak in June 2016. Then it rapidly dropped in July 2016. Then it continues to drop until November 2016, but this decrease is more gradual than before. Then it slowly starts rising until April 2017. After that date, it started growing more rapidly and reached a peak in August 2018. The price rapidly drops until December 2017. Then it slightly spikes up and drops more gradually till the end of the dataset. Till the end, the gas amount starts dropping more rapidly. What is noticeable is that the average gas needed per top ten contracts is lower than the average gas needed for all contracts, which probably contributed to their popularity as less means they are, in general, cheaper.

3. COMPERATIVE EVALUATION

Code: BD > D > spark > spark.py

To run the code five times, we put a for loop around the method that runs the Spark job.
We want to measure the time, so to do that, before we call the method to run the Spark job, we call time.time() and store it in a variable. Whenever the spark job finishes, we call time.time() again and calculate the difference. Then we append it to the array, which will be printed after for loops finishes.

The spark job starts by checking which lines in transactions are good (not corrupted) and then makes the same thing on contracts. Then it splits each line in transactions and merges values for each key. Then we perform the join; however, we only do the join if the transaction is in contracts. Lastly, we get the top ten values. In order to see the top ten values, we then call for loop to print out the ten values.

	Hadoop Map/Reduce	Spark
Run 1	26.56	3.54
Run 2	28.76	2.94
Run 3	25.94	3.11
Run 4	27.52	2.87
Run 5	28.37	3.05
Average	27.43	3.10

Hadoop Map/Reduce based on the average values is 8.85 times slower than Spark.

The reason behind it is that Hadoop reads and writes to a hard disk while Spark reads and writes data to the RAM (random access memory), and the hard disk is much slower than the RAM.

Based on the data gathered, Spark is much more appropriate for this task as the average time to complete the job is significantly lower.