Yash Desai

ETHEREUM ANALYSIS COURSEWORK

PART A. TIME ANALYSIS (20%)

Create a bar plot showing the number of transactions occurring every month between the start and end of the dataset. Create a bar plot showing the average value of transactions in each month between the start and end of the dataset. **Note:** As the dataset spans multiple years and you are aggregating together all transactions in the same month, make sure to include the year in your analysis.
Note: Once the raw results have been processed within Hadoop/Spark you may create your bar plot in any software of your choice (excel, python, R, etc.)

Methodology: -

The first task in the Part A involves computing the total number of transactions per month year wise. To achieve the same, a pair of map reduce functions were used. First of all, we define a **mapper** function which reads the input file. Using the split function in python we extract all the fields (column values) that are comma separated from our csv file. Now to make sure that right data is going into our mapper, we use a condition where we check that the number of fields extracted are equal to the column values in our original dataset. After checking the same we extract the time values that is in epochs format. Now to extract months and years from that value, we use python's time library and the time.gmtime function. Finally, we yield the pairs of (year, month) along with a value '1' which will be mapped with every (year,month) pair. A **combiner** is added to reduce the volume of data transfer between our mapper and reducer which in turn increases the speed of execution. Our combiner and reducer have the exact same inputs and outputs. The **reducer** reads the year-month pairs and computes the total number of transactions by summing all the values that were mapped for each month and year.
Code:-

```python
from mrjob.job import MRJob
import time

class number_of_transactions(MRJob):

    def mapper(self, _, line):

        ''' Reading all the fields of by splitting our data using a comma (extracting fields from the csv data file) '''
        total_fields = line.split(",")
        try:
            if (len(total_fields) == 7):

                ''' Extracting the time field from our data'''
                time_value_epochs = int(total_fields[6])

                ''' using the time.gmttime method from the time library in python, extracting the month and year values '''
                year = time.strftime("%Y", time.gmtime(time_value_epochs))
                month = time.strftime("%m", time.gmtime(time_value_epochs))

                ''' yielding the mapper output as ((year,month), 1) in order to count all the number of transactions '''
                yield((year, month), 1)
        except:
            pass

    def combiner(self, keys, value):
        yield(keys, sum(value))

    ''' yielding the sum of the extracted values from the mapper year and monthwise in the reducer '''
    def reducer(self, keys, value):
        yield(keys, sum(value))

if __name__ =='__main__':
    number_of_transactions.run()
```

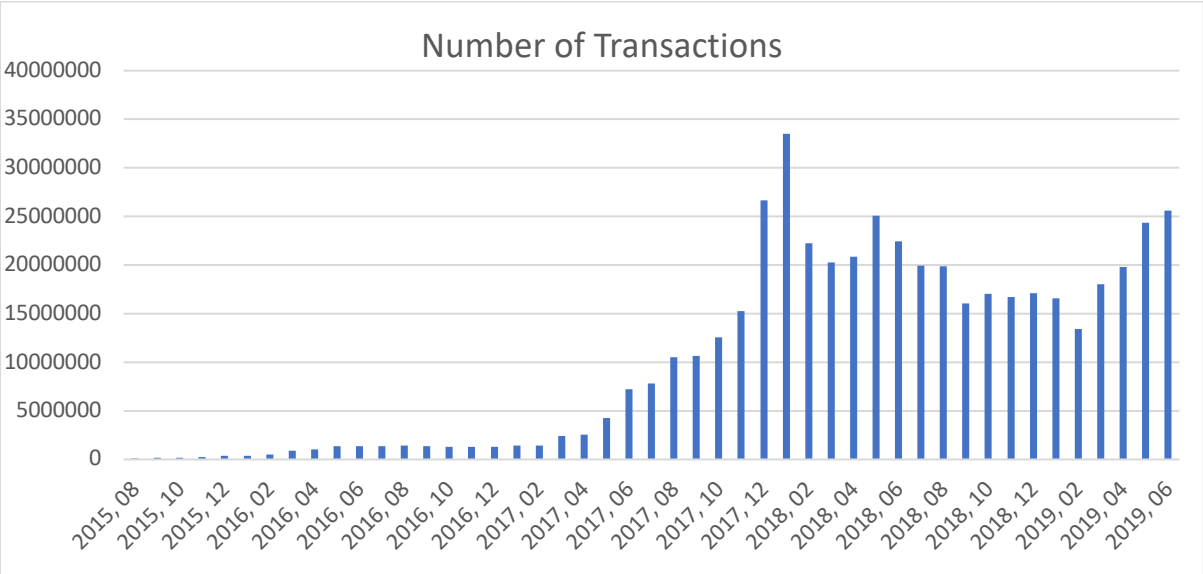
JOB ID TASK 1 PART A
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1648683650522_1187/

Output:-

```
"2015", "09"]   173805
"2015", "10"]   205045
"2015", "12"]   347092
"2016", "02"]   520040
"2016", "04"]   1023096
"2016", "06"]   1351536
"2016", "08"]   1405743
"2016", "11"]   1301586
"2017", "01"]   1409664
"2017", "03"]   2426471
"2017", "05"]   4245516
"2017", "07"]   7835875
"2017", "09"]   10672734
"2017", "10"]   12570063
"2017", "12"]   26687692
"2018", "02"]   22231978
"2018", "04"]   20876642
"2018", "06"]   22471788
"2018", "08"]   19842059
"2018", "11"]   16713911
"2019", "01"]   16569597
"2019", "03"]   18029582
"2019", "05"]   24332475
"2015", "08"]   85609
"2015", "11"]   234733
"2016", "01"]   404816
"2016", "03"]   917170
"2016", "05"]   1346796
"2016", "07"]   1356907
"2016", "09"]   1387412
"2016", "10"]   1329847
"2016", "12"]   1316131
"2017", "02"]   1410048
"2017", "04"]   2539966
"2017", "06"]   7244657
"2017", "08"]   10523178
"2017", "11"]   15292269
"2018", "01"]   33504270
"2018", "03"]   20261862
"2018", "05"]   25105717
"2018", "07"]   19937033
"2018", "09"]   16056742
"2018", "10"]   17056926
"2018", "12"]   17107601
"2019", "02"]   13413899
"2019", "04"]   19830158
"2019", "06"]   25613628
```

After we get the number of transactions per month, I've constructed a bar plot which shows the number of transactions for every month in different years. The bar plot has been constructed in MS Excel after sorting the data in ascending order chronologically.

Plot:-



Task A2: -

Methodology: -

The Task 2 of Part A asks us to yield the average transactions per month for different years instead the number of transactions that we yielded in the Task 1. Hence the mapper function works similarly but instead of yielding a '1' with the month-year pairs, the value of transactions which is the 4th field (column) in the transaction is also yielded.
The reducer in this case calculates both the number of transactions and a sum of all the transaction values. Finally, it yields (the sum of transactions/total number of transactions) which is nothing but the average value of transactions in each month between the start and the end of the transactions dataset. Further, creating a bar plot which shows the average values of transactions in each month-year pair.

Code: -

```python
from mrjob.job import MRJob
import time

class cal_average_a2(MRJob):

    def mapper(self,_,line):
        try:
            '''
            Reading all the fields of by splitting our data using a comma (extracting fields from the csv data file)
            '''
            fields = line.split(',')
            '''
            Extracting the value of transaciton field from our data
            '''
            val = float(fields[3])
            '''
            Extracting the time field from our data
            '''
            date = time.gmtime(float(fields[6]))
            '''
            yielding the mapper output as ((month,year), (1,val)) to get the number of transactions along with the transaction values
            '''
            yield ((date.tm_mon,date.tm_year), (1, val))

        except:
            pass

    '''
    Counting the number of transactions and summing all the tranaction values together monthwise
    '''
    def combiner(self,key,val):
        count = 0
        total = 0
        for v in val:
            count+= v[0]
            total+= v[1]

        yield (key,(count,total))

    def reducer(self,key,val):
        count = 0
        total = 0
        for v in val:
            count += v[0]
            total += v[1]
        '''
        Finally yielding the average values by diving the total sum of values divided by the total number of transactions
        '''
        yield (key, float(total/count))

if __name__=='__main__':
    cal_average_a2.run()

```

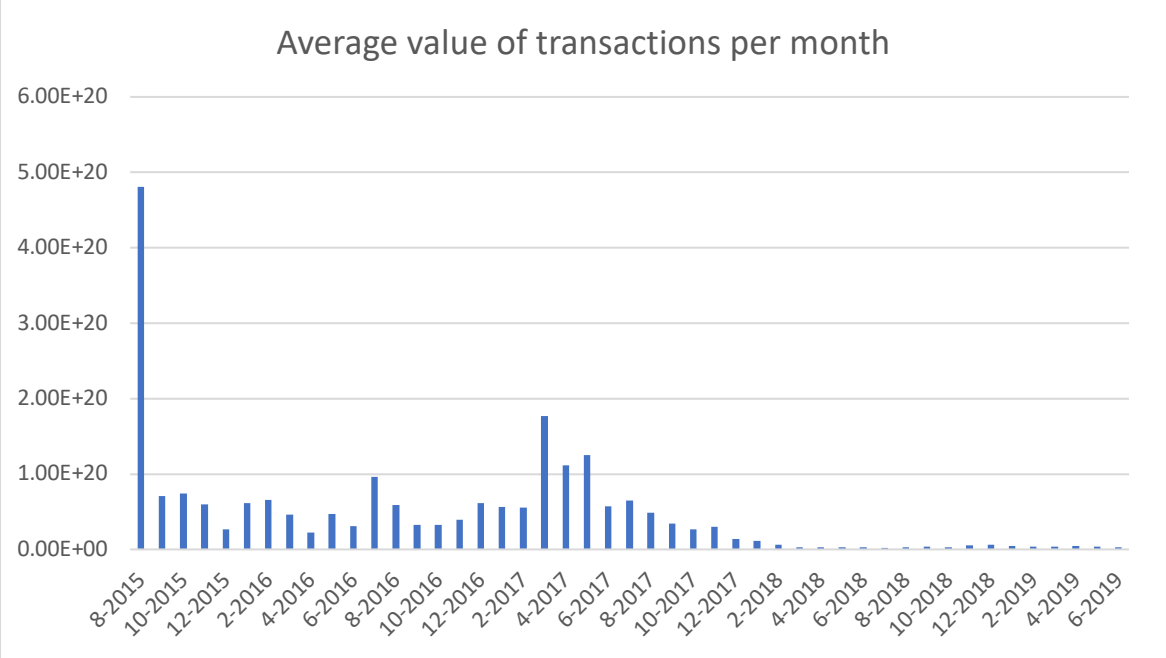
JOB ID TASK 2 PART A:-
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1648683650522_1195/

Output:-

```
[1, 2017]       5.620285956535426e+19
[1, 2019]       4.45481388890254e+18
[10, 2015]      7.416931809333608e+19
[10, 2017]      2.673691042474723e+19
[11, 2016]      3.9644316438348145e+19
[11, 2018]      5.397909048901687e+18
[12, 2015]      2.676409618393977e+19
[12, 2017]      1.3737046580113578e+19
[2, 2016]       6.554760875990558e+19
[2, 2018]       6.23036279509048e+18
[3, 2017]       1.77021580942219e+20
[3, 2019]       3.82327466867818e+18
[4, 2016]       2.2670632047051923e+19
[4, 2018]       2.449268234852297e+18
[5, 2017]       1.248477736519285e+20
[5, 2019]       4.005277417445861e+18
[6, 2016]       3.049033485006488e+19
[6, 2018]       2.808523416305672e+18
[7, 2017]       6.498146379271786e+19
[8, 2016]       5.9081987372901745e+19
[8, 2018]       2.3989507981127424e+18
[9, 2015]       7.046467945767235e+19
[9, 2017]       3.4388885218278113e+19
[1, 2016]       6.1066070477197754e+19
[1, 2018]       1.1164572720748577e+19
[10, 2016]      3.2444426339708813e+19
[10, 2018]      3.0704021430260526e+18
[11, 2015]      5.948474386250735e+19
[11, 2017]      2.9641103274740425e+19
[12, 2016]      6.146658677538237e+19
[12, 2018]      5.944894163484676e+18
[2, 2017]       5.5580090162627715e+19
[2, 2019]       3.9808452645114465e+18
[3, 2016]       4.585306412777953e+19
[3, 2018]       2.7280798911625533e+18
[4, 2017]       1.1135007462190776e+20
[4, 2019]       4.1280245320092544e+18
[5, 2016]       4.7046609524466655e+19
[5, 2018]       2.498190913653367e+18
[6, 2017]       5.678772230936551e+19
[6, 2019]       3.0362427767094016e+18
[7, 2016]       9.57782351058093e+19
[7, 2018]       2.2749347554397706e+18
[8, 2015]       4.805211845959821e+20
[8, 2017]       4.8273956518857196e+19
[9, 2016]       3.2627612247555543e+19
[9, 2018]       3.733481101982357 5e+18
```

After extracting the average value of transactions in each month-year pair for all our data, I've constructed a bar plot as shown below using MS Excel after sorting the data in ascending order chronologically.

Plot:-



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

Evaluate the top 10 smart contracts by total Ether received. An outline of the subtasks required to extract this information is provided below, focusing on a MRJob based approach. This is, however, is not the only way to complete the task, as there are several other viable ways of completing this assignment.

JOB 1 - INITIAL AGGREGATION

To workout which services are the most popular, you will first have to aggregate **transactions** to see how much each address within the user space has been involved in. You will want to aggregate **value** for addresses in the **to_address** field. This, in essence, will be similar to the wordcount problem that we saw in Lab 1 and Lab 2.

JOB 2 - JOINING TRANSACTIONS/CONTRACTS AND FILTERING

Once you have obtained this aggregate of the transactions, the next step is to perform a **repartition join** between this aggregate and **contracts** (example here and slides here). You will want to join the **to_address** field from the output of Job 1 with the **address** field of **contracts** .Secondly, in the reducer, if the address for a given aggregate from Job 1 was not present within **contracts** this should be filtered out as it is a user address and not a smart contract.

JOB 3 - TOP TEN

Finally, the third job will take as input the now filtered address aggregates and sort these via a top ten reducer, utilizing what you have learned from lab 4.

Methodology:-

The jobs 1,2 ad 3 have all been performed simultaneously using a set of 2 mappers and reducers. This has been implemented using the step function from the MRJOB library in python.

The sequence of execution has been defined in the step function as follows: -

```
[MRStep(mapper = self.mapper1, reducer=self.reducer1), MRStep(mapper = self.mapper2, reducer = self.reducer2)]
```

The functions of each mapper and reducer are stated below: -

Map/Reduce	Functions/ Jobs
mapper_1	**Job 1**
reducer_1	**Job 2**
mapper_2	**Job 3**
reducer_2	**Job 3**

Creating the **mapper1** which yields the transaction_values for their corresponding to_address fields for the transactions dataset and while checking another condition it yields the address field from the contracts dataset. I have yielded (2,1) for the contracts dataset in order to avoid overlapping values between both the datasets while the data is being mapped by the mapper.

The function of the **reducer1** is to check if the data is coming from transactions dataset or contracts dataset. I further check if the data that is coming from the transactions dataset is also present in the contracts dataset. If this condition is true, then the data is in the smart contracts and hence we append those values to the smart_contracts list after calculating their **sum** and yield the same along with their address values.

The **mapper2** maps the smart contracts addresses and values from the reducer1 (addresses present in both the datasets) and then passes it to the **reducer2** which sorts the data in descending order and yields the **top ten smart contracts.** These are the top 10 aggregate values of ether received that were present in both the transactions dataset and the contracts dataset.

Code:-

```
from mrjob.job import MRJob
from mrjob.step import MRStep

class part_b(MRJob):

    ''' creating the mapper1 which yields the transaction_values for their corresponding to_address fields for the transactions dataset and while checking
    another condition it yields the address field from the contracts dataset. I have yielded (2,1) for the contracts dataset inroder to avoid overlapping
    values between both the datasets while the data is being mapped by the mapper '''

    def mapper1(self, _, line):
        fields = line.split(',')
        try:

            if len(fields) == 7: ''' if the length of the fields is 7, then the data is coming from the transactions dataset'''
                address_t = fields[2] ''' extracting the to_address field'''
                value = int(fields[3]) '''extracting the transaction values from the transactions dataset'''
                yield address_t, (1,value) '''yielding the to_address field along with the transaction values'''


            elif len(fields) == 5: ''' if the length of the fields is 5, then the data is coming from the contracts dataset'''
                address_c = fields[0] ''' extracting the address field from the contracts dataset'''
                yield address_c, (2,1) '''yielding the address field along with a 2 to differentiate between the to_address and address fields'''
        except:
            pass

    ''' the function of the reducer1 is to check if the data is coming from transactions dataset or contracts dataset. I further check if the data
    that is coming from the transactions dataset is also present in the contracts dataset. If this condition is true then the data is considered to be in
    the smart contracts and hence we append those values to the smart_contracts list after calculating their sum and yield the same along with their address values.'''

    def reducer1(self, key, values):
        f = False
        smart_contracts = []
        for i in values:
            if i[0]==1:
                smart_contracts.append(i[1])
            elif i[0] == 2:
                f = True
        if f:
            yield key, sum(smart_contracts)

    def mapper2(self, key,value):
        yield None, (key,value)  ''' mapping the smart contracts addresses and values from the reducer1 (addresses present in both the datasets) '''

    def reducer2(self, _, keys):
        sortedval = sorted(keys, reverse = True, key = lambda x: x[1]) ''' sorting our data in the descending order of the values (aggregate) '''
        for i in sortedval[:10]: ''' yeilding the addresses and value of ether received (aggregate) of the top 10 smart contracts '''
            yield i[0], i[1]

    def steps(self):
        return [MRStep(mapper = self.mapper1, reducer=self.reducer1), MRStep(mapper = self.mapper2, reducer = self.reducer2)]

if __name__ == '__main__':
    part_b.run()

```

We have two different job ids instead of 1 because we are using 2 pairs of mappers and reducers instead of 1.

JOB ID1 PART B:-
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1648683650522_1218/
JOB ID2 PART B:-
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1648683650522_1228/

Output:-

The top ten smart contracts as per the total ether received.

```
"0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444"   84155100809965865822726776
"0xfa52274dd61e1643d2205169732f29114bc240b3"   45787484483189352986478805
"0x7727e5113d1d161373623e5f49fd568b4f543a9e"   45620624001350712557268573
"0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef"   43170356092262468919298969
"0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8"   27068921582019542499882877
"0xbfc39b6f805a9e40e77291aff27aee3c96915bdd"   21104195138093660050000000
"0xe94b04a0fed112f3664e45adb2b8915693dd5ff3"   15562398956802112254719409
"0xbb9bc244d798123fde783fcc1c72d3bb8c189413"   11983608729202893846818681
"0xabbb6bebfa05aa13e908eaa492bd7a8343760477"   11706457177940895521770404
"0x341e790174e3a4d35b65fdc067b6b5634a61caea"   83790007519177556240575 00
```

PART C. TOP TEN MOST ACTIVE MINERS (15%)

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.

Methodology:-

The Part C of the coursework has been performed using a set of 2 mappers and reducers similar to what was done in the Part B. This has been implemented using the step function from the MRJOB library in python.
The sequence of execution has been defined in the step function as follows: -
[MRStep(mapper = self.mapper1, reducer=self.reducer1), MRStep(mapper = self.mapper2, reducer = self.reducer2)]

Creating the **mapper1** which yields the *mine size* corresponding to the *miner address* for the 'blocks' dataset. First, I check if the number of columns in the dataset that I have passed are equal to 9. This means that we are using the correct dataset i.e the 'blocks' dataset which has 9 fields. The *miner* field which is the miner address can be found at the index number 2 (3rd field) whereas the size field which is the miner sizer can be found at the index number 4 (5th column). After fetching both the columns, the mapper 1 yields *(miner_address, int(mine_size)).*

The **reducer1** yields the same fields as the mapper1 but instead of the size mined for each address value it yields the sum of the ether mined for every miner.
In the **reducer2** I have sorted the value of the mine size for its corresponding miner in descending order (reverse=True). Further I have just extracted the top 10 *miner ,mine_size pairs.*

Code:-

```python
from mrjob.job import MRJob
from mrjob.step import MRStep

class top10miners(MRJob):
    '''
    The mapper1 yields the miner size for the corresponding miner address. First I check if the number of columns in the dataset that I have passed are equal to 9.
    This means that we are using the correct dataset i.e the 'blocks' dataset which has 9 fields. The miner field which is the miner address can be found at the
    index number 2 (3rd field) whereas the size field which is the miner sizer can be found at the index number 4 (5th column). After fetching both of these columns,
    the mapper 1 yields (miner_address, int(mine_size))
    '''
    def mapper1(self, _, line):
        fields = line.split(',')
        try:
            if len(fields) == 9:
                miner_address = fields[2]
                mine_size = fields[4]
                yield (miner_address, int(mine_size))

        except:
            pass

    '''
    The reducer1 yields the same fields as the mapper1 but instead of the size mined for each address value it yields the sum of the ether mined for every miner.
    '''
    def reducer1(self, key, value):
        try:
            yield(key, sum(value))

        except:
            pass

    '''
    I have used the mapper2 to yield the miner address and its total mine size together as a single tuple.
    '''

    def mapper2(self, key, sum_of_value):
        try:
            yield(None, (key,sum_of_value))
        except:
            pass

    '''
    In the reducer2 I have sorted the value of the mine size for its corresponding miner in descending order (reverse=True). Further I have just extracted
    the top 10 miner,mine_size pairs.
    '''
    def reducer2(self, _, value):
        try:
            sort_val = sorted(value, reverse = True, key = lambda x:x[1])
            for val in sort_val[:10]:
                yield(val[0],val[1])
        except:
            pass

    def steps(self):
        return [MRStep(mapper = self.mapper1, reducer=self.reducer1), MRStep(mapper = self.mapper2, reducer = self.reducer2)]

if __name__ == '__main__':
    top10miners.run()
```

JOB ID Part C:-
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1648683650522_5312/

Output:-

```
"0xea674fdde714fd979de3edf0f56aa9716b898ec8"	23989401188
"0x829bd824b016326a401d083b33d092293333a830"	15010222714
"0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c"	13978859941
"0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5"	10998145387
"0xb2930b35844a230f00e51431acae96fe543a0347"	7842595276
"0x2a65aca4d5fc5b5c859090a6c34d164135398226"	3628875680
"0x4bb96091ee9d802ed039c4d1a5f6216f90f81b01"	1221833144
"0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb"	1152472379
"0x1e9939daaad6924ad004c2560e90804164900341"	1080301927
"0x61c808d82a3ac53231750dadc13c777b59310bd9"	692942577
```

PART D. DATA EXPLORATION (40+%)

The final part of the coursework requires you to explore the data and perform some (or all) analysis of your choosing. These tasks may be completed in either MRJob or Spark or any other relevant library/framework that you deem fit. Below are some suggested ideas for analysis that could be undertaken, along with an expected grade for completing it to a *good* standard. You may attempt several (or all) of these tasks.

Scam Analysis: -

Popular Scams: Utilising the provided scam dataset, what is the most lucrative form of scam? Does this correlate with certainly known scams going offline/inactive? For the correlation, you could produce the count of how many scams for each category are active/inactive/offline/online/etc and try to correlate it with volume (value) to make conclusions on whether state plays a factor in making some scams more lucrative. Therefore, getting the volume and state of each scam, you can make a conclusion whether the most lucrative ones are ones that are online or offline or active or inactive. So for that purpose, you need to just produce a table with SCAM TYPE, STATE, VOLUME which would be enough **(15%).**

Methodology:-

In the Scam Analysis part of the coursework, I have used PySpark to calculate the most lucrative scam type and the status of the scam. First of all I pre-processed the 'scams.json' file to extract 3 fields, *miner address*, *scam type* and *scam status* and convert the same to a csv format. Now we have a new dataset with the above three fields.

To compute the most lucrative scam I use the transactions dataset and extract the 'gas' and 'values' field for every 'to_address' field. Before extracting these fields from the dataset, I perform some data cleaning on the dataset, like removing the 'value' rows with '0' values and converting the 'to_address' field to a string format. After pre-processing on the transactions dataset, I join the scams.csv dataset and the transactions dataset with the 'address' value as the key. The new dataset has the following fields: - *address, gas, value, scam type* and the *scam status.*

From this dataset I have extracted the *scam type* with their corresponding gas and value fields. To compute the most lucrative scam type I have computed the summation of the '*gas'* and the '*value'* fields. The output of this following operation is the **Output 1** of this task and is shown on the next page. It can be seen from the output that the *scam type* '**Scamming'** is the most lucrative scam with the highest gas and the mine value of '***5978937963.0'*** and '***4.4715806098440545e+22'*** respectively.

Now to further check if the most lucrative scam type are online or offline or active or inactive, we take *scam type* and *scam status* both as the key and extract the gas value for the same. I compute the sum of all the gas values and check which one of the 'scam type'
and 'scam status' pairs have the highest *gas* and the *mine value*.

Conclusion:-

- It can be observed from the **Output 2** on the next page that the scam type which was most lucrative in **Output 1** '**Scamming**' has maximum *(gas, value)* for the status '**Active**'.

- Further it can be concluded that the *(gas, value)* pair for the scam status '**Active'** and '**Offline'** are maximum followed by the scam status '**Inactive**' and '**Suspended**'. The 'Suspended' scam status has values which are significantly lower than that of the scam status '**Active**' and '**Offline'.**

- The scam type '**Face ICO**' has all the scams that are **'Offline'.**

Code:-

```
import pyspark
import json
from pyspark.sql.functions import *


def clean(line):
    try:
        fields = line.split(',')
        ''' Checking if the input dataset's every row has 7 fields (removing rows with empty fields)'''
        if len(fields) == 7:
            str(fields[2])
            ''' Removing all the rows where mine value is 0 '''
            if int(fields[3]) == 0:
                return False
            return True
    except:
        return False

''' Creating a spark context'''
sc = pyspark.SparkContext()
print(sc.applicationId)

''' Creating an SQL context from Spark Context'''
sql_context = pyspark.SQLContext(sc)

''' Reading the scam csv file (converted from the scam.json file on hadoop) using the SQL context'''
dat = sql_context.read.option('header', False).format('csv').load("scam.csv")

''' After reading the scams csv file, we convert it to spark's rdd for further spark operations'''
sc_rdd = dat.rdd.map(lambda x: (x[0],(x[1],x[2])))

''' We filter our transactions data using the above defined clean function '''
trans_data = sc.textFile('/data/ethereum/transactions').filter(clean)

''' Extracting data from the transactions dataset [2] is 'to_address' [4] is 'gas' and [3] is 'value' '''
extract_trans = trans_data.map(lambda l:  (l.split(',')[2], (float(l.split(',')[4]), float(l.split(',')[3]))))

''' joining the scams.csv dataset with our trasactions dataset '''
scam_join = extract_trans.join(sc_rdd)

''' x[1] is the key and x[1][0] is the data from transactions dataset and x[1][1] is the data from the scams dataset.
 In the below map funciton we have extracred the scam_type which is x[1][1][0], the gas value x[1][0][0] and the mine value x[1][0][1]'''
scam_extract = scam_join.map(lambda x: (x[1][1][0], (x[1][0][0], x[1][0][1])))

''' Using the below reduceByKey method we have performed the summation of all the values from the above dataset'''
scam_sum= scam_extract.reduceByKey(lambda a,b: (a[0]+b[0], a[1]+b[1]))
scam_sum.saveAsTextFile('scam_ou1')

'''
Part 2
'''

''' Extracting the scam'''
map_new = scam_join.map(lambda x: (x[1][1],  (x[1][0][0],x[1][0][1])))
reduced_new = map_new.reduceByKey(lambda a,b: (a[0]+b[0], a[1]+b[1])).sortByKey(ascending=True)
reduced_new.saveAsTextFile('scam_out2')
```

JOB ID Part D Scam Analysis:-
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1648683650522_5983

Output 1:-

```
(u'Phishing', (724922480.0, 4.372701002504078e+22))
(u'Fake ICO', (7402302.0, 1.356457566889 63e+21))
(u'Scamming', (5978937963.0, 4.4715806098440545e+22))
```

Output 2:-

```
((u'Fake ICO', u'Offline'), (7402302.0, 1.35645756688963e+21))
((u'Phishing', u'Active'), (111750772.0, 6.256455846464803e+21))
((u'Phishing', u'Inactive'), (2255249.0, 1.488677770799503e+19))
((u'Phishing', u'Offline'), (610020459.0, 3.745402749273796e+22))
((u'Phishing', u'Suspended'), (896000.0, 1.63990813e+18))
((u'Scamming', u'Active'), (4282186989.0, 2.261220527919417e+22))
((u'Scamming', u'Offline'), (1694248234.0, 2.209989065129634e+22))
((u'Scamming', u'Suspended'), (2502740.0, 3.71016795e+18))
```

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? Also, could you correlate the complexity for some of the top-10 contracts found in Part-B by observing the change over their transactions (10%)

I have divided this part of the assignment into three tasks.
Task 1: - See how the gas price has changed over time
Task 2: - Check if the contracts have become more complicated
Task 3: - Checking the complexity for some of the contracts from top 10 contracts in Part B

Task1:-

Methodology: -

In the mapper, we extract the gas value and their corresponding year-month pairs. Using this we can compute the total sum of gas price per month for every year using the reducer.
Hence our final result from the reducer is the (year-month) pairs as the keys and average gas price per month as the values. We further plot the same as a line chart using our output values in MS Excel. Line chart is used so that a clear conclusion of the trend of gas price can be formulated

Code:-

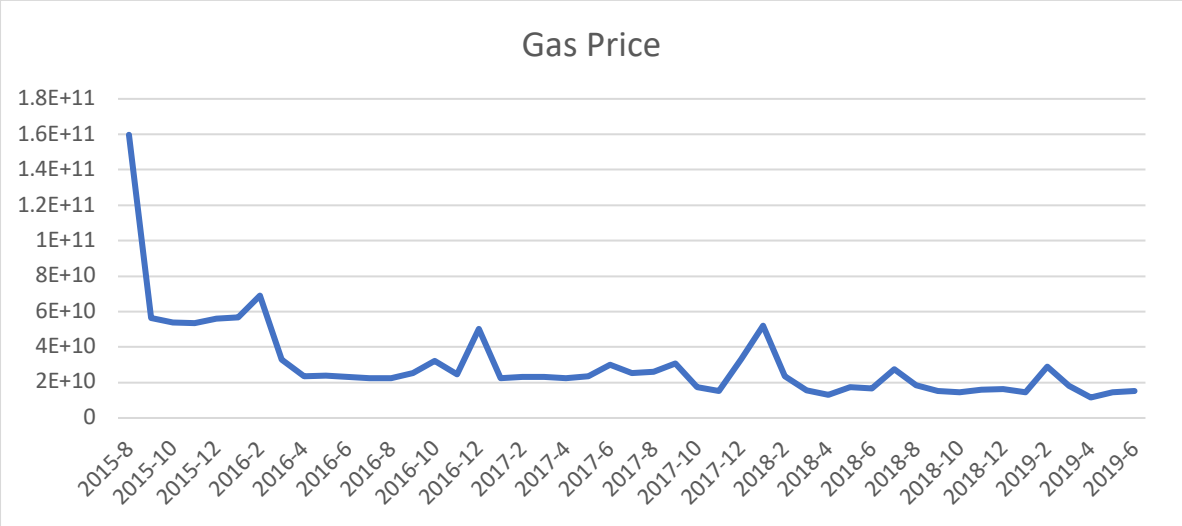
```python
from mrjob.job import MRJob
import time

'''
In the mapper, I extract the gas values for every month and year pairs in our dataset. Month and year are extracted separately and then joined togetehr and used as a key for the reducer.
'''

''' The reducer calculates the average value of gas per month by adding all the gas values and dividing it by the total number of values.
Fnially the reducer yields the average value of gas per month/year.
'''

class gas_guzzler(MRJob):
    def mapper(self, _, lines):
        try:
            fields = lines.split(",")
            if len(fields) == 7:
                gas_val = int(fields[5])
                date = time.gmtime(float(fields[6]))
                year_mon = str(date.tm_year) + '-' + str(date.tm_mon)
                yield(year_mon,gas_val)
        except:
            pass

    def reducer(self, key, values):
        price_var = 0
        c = 0

        for val in values:
            price_var = price_var + val
            c = c + 1

        yield(key, float(price_var/c))


if __name__ == '__main__':
    gas_guzzler.run()
```

JOB ID Gas Guzzler Part1: -
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1648683650522_6268/

Output:-

```
"2015-11"   53607614201.796776
"2015-8"    159744029578.03113
"2016-1"    56596270931.31685
"2016-10"   32112869584.914665
"2016-12"   50318068074.68128
"2016-3"    32797039087.356667
"2016-5"    23746277028.263245
"2016-7"    22629542449.24175
"2016-9"    25270403393.626083
"2017-11"   15312465314.693544
"2017-2"    23047230327.254303
"2017-4"    22355124545.395317
"2017-6"    30199442465.128727
"2017-8"    25905774673.990257
"2018-1"    52106060636.84502
"2018-10"   14526936383.350008
"2018-12"   16338844844.014668
"2018-3"    15549765961.743273
"2018-5"    17422505108.986416
"2018-7"    27506077453.154327
"2018-9"    15213870989.523378
"2019-2"    28940599438.14876
"2019-4"    11569797163.994152
"2019-6"    15076119023.332266
"2015-10"   53901692120.53661
"2015-12"   55899526672.35486
"2015-9"    56511301521.033226
"2016-11"   24634294365.279953
"2016-2"    69180681134.38849
"2016-4"    23361180502.721268
"2016-6"    23021251389.812134
"2016-8"    22396836435.95849
"2017-1"    22507570807.719795
"2017-10"   17506742628.97231
"2017-12"   33423472930.410748
"2017-3"    23232253600.81683
"2017-5"    23572314972.01526
"2017-7"    25460300456.232986
"2017-9"    30675106219.637596
"2018-11"   16034859008.681648
"2018-2"    23636574203.828976
"2018-4"    13153739247.92998
"2018-6"    16533308366.813036
"2018-8"    18483235826.894573
"2019-1"    14611816445.785303
"2019-3"    18083035519.522877
"2019-5"    14479858767.711182
```

Plot:-



It can be observed from the trend of the line chart that the average gas price per month has no trend as such and has been fluctuating over the time. A significant dip is observed from August to October in the year 2015, since then the price has fluctuated a lot. The line chart was plotted using our output values of year/month pairs with the average gas price month on MS Excel.

Task 2: -

Methodology: -

For the task 2 we need to check if the contracts have become more complex over the time. We use both the contracts and the blocks dataset for this task. After cleaning our datasets, we use the *contracts* dataset to extract block number along with a 1 (used in future to calculate average). Further we extract block number, difficulty, gas value and the year-month pairs from the *blocks* dataset. Now using block number as the key, we join the contracts and the block dataset. After joining the datasets, we extract year-month pairs, difficulty, gas consumed and the counter (1s). Finally, I perform a summation of all these values using a lambda function. The new data now has the following structure,

```
('2019-02', (1084082528199181398629L, 3197663942850, 419164))
(Year-month, (total difficulty, total gas consumed, counts))
```

To get the average gas price and difficulty per month for each block (which is present in both the datasets) I further divide the total gas and difficulty values with the total number of blocks. Our final output looks like this:-

```
('2015-08', (4030960805570, 360218))
(Year-month, (average difficulty, average gas consumed))
```

The output is the cleaned and line charts for difficulty and gas consumed per month have been plotted using MS Excel.

Code: -

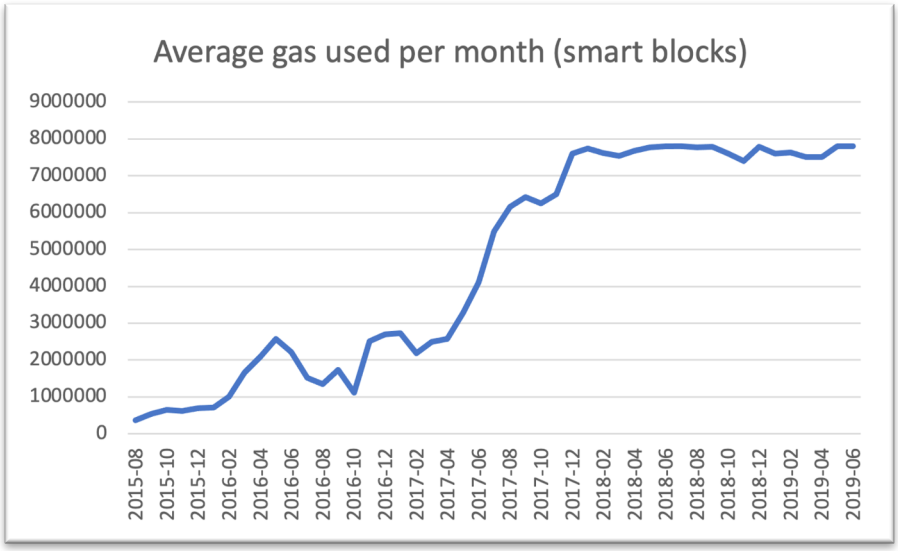
```python
import pyspark
import time

'''
creating a function to clean our contracts dataset from where we will extract the block number
'''
def clean_contracts(line):
    try:
        fields = line.split(',')
        if len(fields) != 5:
            return False
        float(fields[3])
        return True
    except:
        return False

'''
creating a function to clean our blocks dataset
'''

def clean_blocks(line):
    try:
        fields = line.split(',')
        if len(fields)!=9:
            return False
        float(fields[0]) # block number
        float(fields[6]) # gas used
        float(fields[7]) # time stamp
        return True
    except:
        return False

sc = pyspark.SparkContext() # creating spark context
print(sc.applicationId)

read_contract = sc.textFile('/data/ethereum/contracts')
contract_clean = read_contract.filter(clean_contracts) # cleaning the contracts dataset
block_id = contract_clean.map(lambda l: (l.split(',')[3], 1)) # getting block id from the contracts dataset

read_block = sc.textFile('/data/ethereum/blocks')
blocks_clean = read_block.filter(clean_blocks) # cleaning the blocks dataset using the function we created above

'''
Extracting block number, difficulty, gas used and year-month pairs from the blocks dataset
'''
block_map = blocks_clean.map(lambda l: (l.split(',')[0], (int(l.split(',')[3]),int(l.split(',')[6]), time.strftime("%Y-%m", time.gmtime(float(l.split(',')[7])))))
print(block_map.take(2))
# [(u'4776199', (1765656009004680, 2042230, '2017-12')), (u'4776200', (1765656009037448, 4385719, '2017-12'))]

 '''
 Joining the contracts and the blocks dataset using the key value as the block number.
 We get block id, along with difficulty, gas used and date which are common in both our datasets. We also have a '1' at the position x[1][1],
 from our block_id
 '''
 results = block_map.join(block_id)
 print(results.take(2))
 # [(u'7352442', ((1862016659491710, 6317667, '2019-03'), 1)), (u'7352442', ((1862016659491710, 6317667, '2019-03'), 1))]

 '''
 Extracting Year-Month pairs, difficulty, gas value and a '1' which will help us keep a count of the values
 '''
 result1 = results.map(lambda x: (x[1][0][2], (x[1][0][0], x[1][0][1], x[1][1])))
 print(result1.take(2))
 # [('2017-08', (1600249491213932, 2032838, 1)), ('2019-06', (2134143517862686, 7989075, 1))]

 '''
 Adding all the values
 Now we have total values of difficulty and gas consumed per year-month pairs along with their counts
 '''
 total_val = result1.reduceByKey(lambda a,b: (a[0]+b[0], a[1]+b[1], a[2]+b[2]))
 print(total_val.take(2))
 # [('2019-02', (1084082528199181398629L, 3197663942850, 419164)), ('2016-07', (1662703037969682192, 43543836232, 28773))]

 '''
 Dividing the difficulty and gas values by the total counts to get average if these values per year-month pairs
 '''
 avg_val = total_val.map(lambda x: (x[0], (float(x[1][0] / x[1][2]), float(x[1][1] / x[1][2])))).sortByKey(ascending=True)
 print(avg_val.take(2))
 # [('2015-08', (4030960805570, 360218)), ('2015-09', (6577868584193, 540131))]

 '''
 saving our output as outg2
 '''
 avg_val.saveAsTextFile('outg2')
 #Output is of the format YY-MM, Complexity, Avg Gas
```

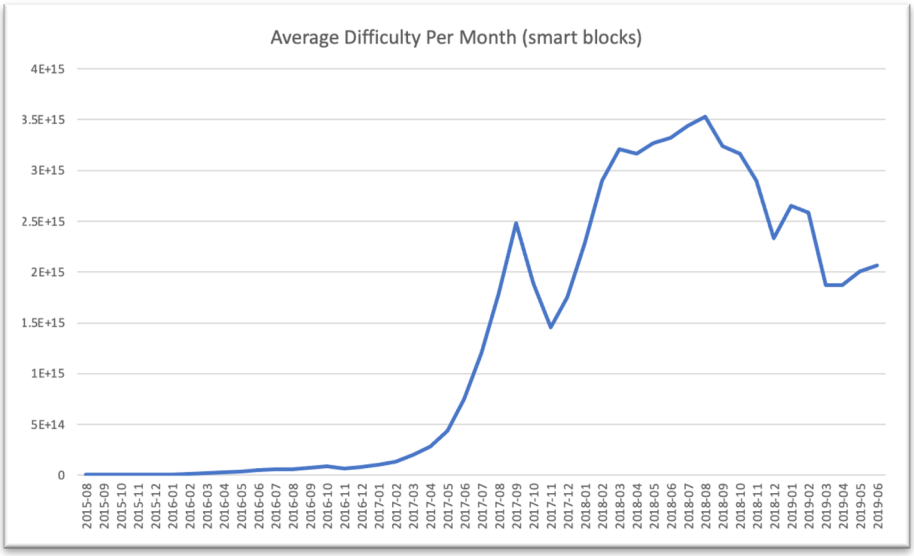
JOB ID Gas Guzzler Part2: -
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/ application_1649894236110_0701

Output: -

```
('2015-08', (4030960805570.0, 360218.0))
('2015-09', (6577868584193.0, 540131.0))
('2015-10', (6352827787297.0, 641355.0))
('2015-11', (7772752046929.0, 609518.0))
('2015-12', (8279271173937.0, 696716.0))
('2016-01', (9509622515878.0, 714548.0))
('2016-02', (12769672945336.0, 1005720.0))
('2016-03', (20066111443630.0, 1650559.0))
('2016-04', (28654861822851.0, 2097505.0))
('2016-05', (39311237605142.0, 2571039.0))
('2016-06', (49061031594690.0, 2217506.0))
('2016-07', (57786919611082.0, 1513357.0))
('2016-08', (59234534795662.0, 1345600.0))
('2016-09', (72600341869690.0, 1734328.0))
('2016-10', (87658236572825.0, 1117711.0))
('2016-11', (67812116442679.0, 2503838.0))
('2016-12', (77496935205604.0, 2699688.0))
('2017-01', (102282325084568.0, 2727947.0))
('2017-02', (130757112575887.0, 2177629.0))
('2017-03', (198044781241186.0, 2499731.0))
('2017-04', (284126014229797.0, 2568672.0))
('2017-05', (436460724525936.0, 3282000.0))
('2017-06', (752656695470601.0, 4110544.0))
('2017-07', (1200999266684616.0, 5492711.0))
('2017-08', (1790200849201658.0, 6157330.0))
('2017-09', (2481432368327849.0, 6419935.0))
('2017-10', (1885459218688474.0, 6243435.0))
('2017-11', (1457697972100907.0, 6503790.0))
('2017-12', (1753712331569158.0, 7602976.0))
('2018-01', (2281188940495787.0, 7743489.0))
('2018-02', (2895297998304151.0, 7621988.0))
('2018-03', (3208321357874551.0, 7540115.0))
('2018-04', (3163895606602080.0, 7673254.0))
('2018-05', (3271540170064034.0, 7772207.0))
('2018-06', (3322174039186059.0, 7804846.0))
('2018-07', (3438689513774835.0, 7799773.0))
('2018-08', (3525770967637325.0, 7771450.0))
('2018-09', (3239257170470205.0, 7782860.0))
('2018-10', (3168540540592017.0, 7595336.0))
('2018-11', (2893844101806584.0, 7395694.0))
('2018-12', (2335811289311401.0, 7790698.0))
('2019-01', (2650625285026055.0, 7608784.0))
('2019-02', (2586296838944139.0, 7628670.0))
('2019-03', (1871306263893841.0, 7515069.0))
('2019-04', (1875185187990232.0, 7501078.0))
('2019-05', (2006321344431032.0, 7804283.0))
('2019-06', (2064945483489737.0, 7799617.0))
```

Plots :-





It can be observed that as the difficulty to mine increases over the years, the gas price also increases with a lot of fluctuations in the between. A significant increase in difficulty to mine is observed from ***2017-02*** along with an increase in gas at the same period. Hence it can be inferred that difficulty in mining is positively correlated to the gas consumed per month.

Task 3: -

Computing average difficulty per month for the top 3 addresses in our top 10 contracts that were extracted in the Part C of this coursework.
The top three addresses that are used for this task are: -

Address 1	0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444
Address 2	0xfa52274dd61e1643d2205169732f29114bc240b3
Address 3	0x7727e5113d1d161373623e5f49fd568b4f543a9e

Methodology: -

First, I create a list of the top three addresses from our top contracts output of Part C. Now, while creating the data clean function which we apply while filtering our entire dataset, I have used a condition which will enable the function to run only on a particular address value. For this, I create a for loop which will run 3 times and take the 3 address values that I have stored in the list *top_three_contracts*. From the transactions dataset I have extracted the *block id* and *to_address* fields. Now from the blocks dataset I have extracted *block-id, year-month* pairs (converted from timestamps) and the *difficulty* fields. After extracting data from both the datasets, I have joined the datasets with the key as the *block-id.* Using a lambda function I have mapped the y*ear-month* pairs along with the *difficulty* field. Finally, I take the summation of the difficulty field to compute total difficulty per month. This entire process will run thrice, for 3 of our top contracts addresses and hence generate 3 outputs. I have plotted these on a line chart to get a clear idea of the trends.

Code:-

```python
import pyspark
import time

global topadd # creating a global address variable to store the top three contracts from the for loop and use it in the

top_three_contracts = ["0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444", "0xfa52274dd61e1643d2205169732f29114bc240b3","0x7727e5113d1d161373623e5f49fd568b4f543a9e"]

'''
creating a function to clean our transactions dataset and adding a condition which only takes a particular value of address from our list of top three contracts
'''
def filter_trans(line):
    try:
        fields = line.split(',')
        if len(fields) != 7:
            return False
        # if fields[2] in popular_contracts:
        #     return True
        if fields[2] == topadd:
            return True
    except:
        return False

'''
creating a function to clean our blocks dataset
'''

def filter_blocks(line):
    try:
        fields = line.split(',')
        if len(fields)!=9:
            return False
        float(fields[0])
        float(fields[6])
        float(fields[7])
        return True
    except:
        return False



for add in top_three_contracts:
    '''
    Creating a for loop which will extract the difficulty of mining per month for the top three contracts (the list that is created above)
    '''
    sc = pyspark.SparkContext()
    print(sc.applicationId)
    topadd = add
    trans = sc.textFile('/data/ethereum/transactions')
    clean_trans = trans.filter(trans_good)
    block_id = clean_trans.map(lambda l: (l.split(',')[0], l.split(',')[2])) # extracting block id and address from the transactions data


    blocks = sc.textFile('/data/ethereum/blocks')
    clean_blocks = blocks.filter(filter_blocks)
    '''
    extracting block-id, difficulty and year-month pairs from the blocks dataset
    '''
    block_mapped = clean_blocks.map(lambda l: (l.split(',')[0], (int(l.split(',')[3]), time.strftime("%Y-%m", time.gmtime(float(l.split(',')[7]))))))

    joined = block_mapped.join(block_id) # joining both the datasets with same block-id
    extracted = joined.map(lambda x: (x[1][0][1], x[1][0][0] )) # extracting the year-month pairs and difficulty for all those blocks

    summed = extracted.reduceByKey(lambda a,b: float(a+b)).sortByKey(ascending=True) # taking the summation of difficulty

    summed.saveAsTextFile('g3_'+topadd)
    sc.stop() # stopping the instance after every for loop iteration
```

Job IDs:-

Address:-	Job Id
0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444	http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1649894236111_1999
0xfa52274dd61e1643d2205169732f29114bc240b3	http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1649894236111_2004

0x7727e5113d1d161373623e5f49fd568b4f543a9e	http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_164989423611_2008

Output for all three addresses: -

Address 1

```
('2016-07', 2.718958970612495e+17)
('2016-08', 3.3090653484252334e+18)
('2016-09', 3.644407855239301e+18)
('2016-10', 3.7885071058745856e+18)
('2016-11', 2.042418418124033e+18)
('2016-12', 7.338033309097378e+16)
('2017-01', 9.008018607132115e+16)
('2017-02', 8.077608327164771e+16)
('2017-03', 4.002547211877932e+17)
('2017-04', 3.894301116248709e+17)
('2017-05', 1.4490512169815828e+18)
('2017-06', 5.34327808553945e+18)
('2017-07', 6.519868087058378e+18)
('2017-08', 1.008868102297838e+19)
('2017-09', 1.4819432308195158e+19)
('2017-10', 1.1263434923421493e+19)
('2017-11', 1.5588975032167318e+19)
('2017-12', 3.6429620739654926e+19)
('2018-01', 7.34655697293912e+19)
('2018-02', 3.5853841953898926e+19)
('2018-03', 2.479765650510974e+19)
('2018-04', 1.8190956842146005e+19)
('2018-05', 1.7230088406829412e+19)
('2018-06', 1.1159817736018 43e+19)
('2018-10', 2.663552566255318e+16)
('2019-01', 5016104683133135.0)
('2019-03', 3738051403883900.0)
('2019-04', 1868782580019087)
('2019-05', 5716621807119833.0)
('2019-06', 4075830573246786.0)
```

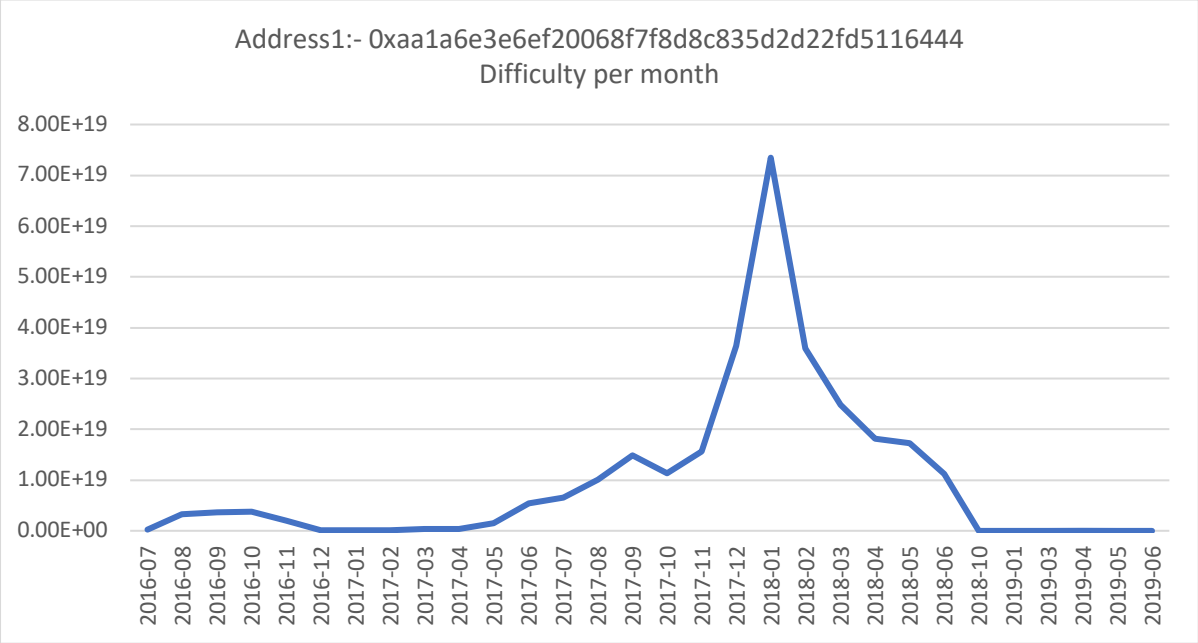
Address 2

```
('2016-07', 2.1645507440666838e+17)
('2016-08', 1.7153342950359132e+18)
('2016-09', 1.7437561047816146e+18)
('2016-10', 2.1547296303776924e+18)
('2016-11', 1.471879506542946e+18)
('2016-12', 1.6424292271350193e+18)
('2017-01', 2.3326302544017316e+18)
('2017-02', 3.0315421946356116e+18)
('2017-03', 7.513660193076873e+18)
('2017-04', 9.365151703394273e+18)
('2017-05', 2.1414999795367584e+19)
('2017-06', 5.233321085941123e+19)
('2017-07', 8.401119816057328e+19)
('2017-08', 1.2195815691977182e+20)
('2017-09', 1.2969332123853201e+20)
('2017-10', 1.039676390838899e+20)
('2017-11', 9.291891745276897e+19)
('2017-12', 1.3896930342939252e+20)
('2018-01', 1.911256077952511e+20)
('2018-02', 1.796606866909491e+20)
('2018-03', 1.632497674090881e+20)
('2018-04', 1.4750054472399 3e+20)
('2018-05', 1.6335512552493472e+20)
('2018-06', 1.4200348717856938e+20)
('2018-07', 1.4651400966468546e+20)
('2018-08', 1.3577078881636198e+20)
('2018-09', 1.2236647966917034e+20)
('2018-10', 1.1061035188616924e+20)
('2018-11', 1.0110035948677279e+20)
('2018-12', 8.280783841826737e+19)
('2019-01', 8.190083939348629e+19)
('2019-02', 7.320184632422411e+19)
('2019-03', 4.9524673348498104e+19)
('2019-04', 5.5116335960837505e+19)
('2019-05', 6.990279166612734e+19)
('2019-06', 6.663560163798499e+19)
```

Address 3

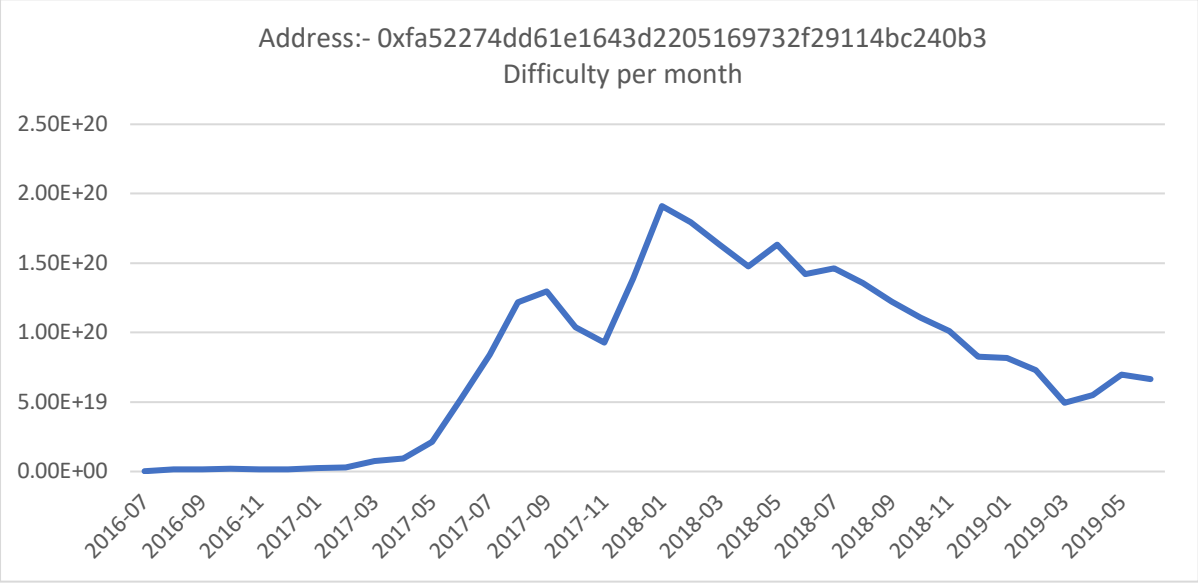
```
('2016-08', 1.575517902711486e+17)
('2016-09', 2.7099988975149875e+17)
('2016-10', 2.6239898209485373e+17)
('2016-11', 2.0231788592331693e+17)
('2016-12', 3.0402780082904966e+17)
('2017-01', 3.559905958906322e+17)
('2017-02', 5.354255372826449e+17)
('2017-03', 1.8018857184539287e+18)
('2017-04', 2.469472953763996e+18)
('2017-05', 5.386944933665667e+18)
('2017-06', 2.2094278780029293e+19)
('2017-07', 5.296304922916344e+19)
('2017-08', 1.455296459247652e+20)
('2017-09', 1.6571017174755856e+20)
('2017-10', 1.4193341425964027e+20)
('2017-11', 1.7962826535957604e+20)
('2017-12', 6.84893958069538e+19)
```

Plot for Address 1:-



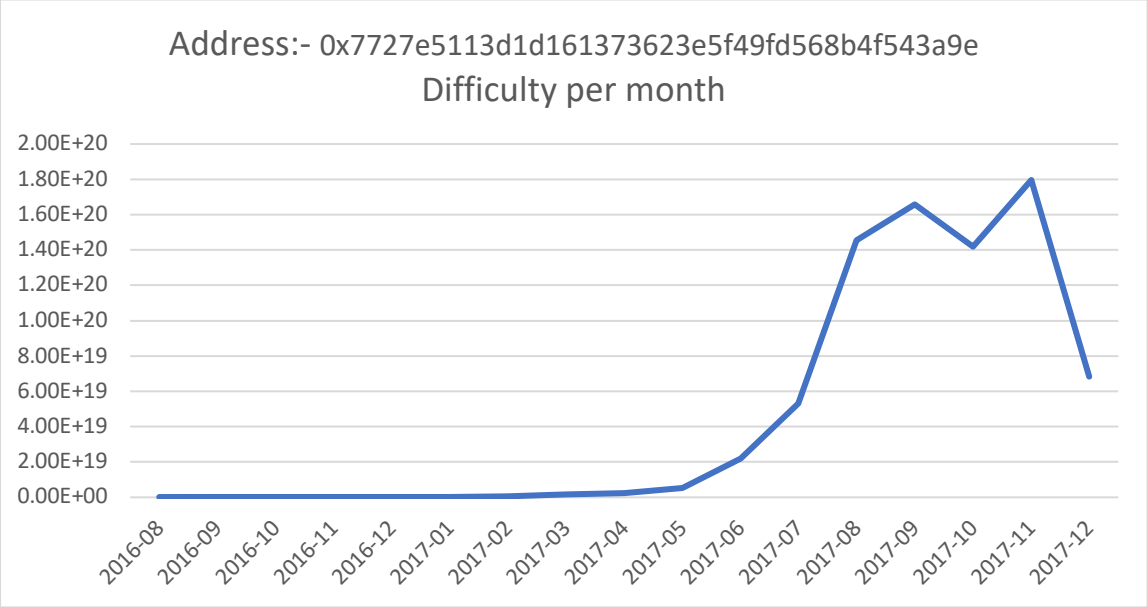
It is evident that difficulty to mine for our top contract started to rocket in December 2017. A peak was observed in January 2018 and since then the difficulty decreased gradually until October 2018. The trend for this particular address is correlated with the average difficulty per month (from Task 2 difficulty trend). The first peak in this graph is observed in August 2017, which is similar to the one observed in our average difficulty plot.

Plot for Address 2:-



Two peaks can be observed from the trend of this particular address. But only the first peak which is observed in August 2017, can be correlated to the average difficulty in August 2017 that also had a peak during this time. The second peak is observed in August 2018 after which there are a few fluctuations and then the difficulty starts to decrease gradually.

Plot for Address 3:-



For the third address in the smart contracts, the difficulty had a peak in August 2017 which again correlates with the average difficulty trend graph. The difficulty decreases gradually after November 2017.

Comparative Evaluation: -

Reimplement Part B in Spark (if your original was MRJob, or vice versa). How does it run in comparison? Keep in mind that to get representative results you will have to run the job multiple times, and report median/average results. Can you explain the reason for these results? What framework seems more appropriate for this task? (10%)

Running on Spark: -

Methodology: -

We use both the transactions and the contracts dataset exactly like we did in the Part B of our assignment, but here we have used Spark to accomplish the task rather than traditional map-reduce. We start by creating functions to clean both of our datasets. Now, from the *transactions* dataset we extract the *address* and the *values* fields and take a summation of the values for all the addresses. Hence, we have different addresses with total mine values. Now in order to check if these addresses are a part of smart contracts we check if they are present in the contract's dataset. To accomplish this, we join both the datasets with addresses as the key value. Hence, we now obtain the addresses present in both datasets (smart contracts) along with their total

mine values. Finally, to get the top 10 miners we sort our data into descending order and extract the top 10 values. Simultaneously using python's time library, I have computed the time of execution for this Spark Job. The objective of this part of the coursework is to compare the time of execution of the spark job with that of map reduce for the same problem. To achieve this I have executed this particular Spark Job thrice and calculated the average time taken for these three jobs. Similarly, I executed the Part B of this assignment again 3 times (in map reduce) and calculated the average of the execution times.

Spark Code:-

```
import pyspark
from time import *
'''
Function to clean our transactions dataset
'''
def clean_trans(line):
    try:
        fields = line.split(",")
        if len(fields) == 7:
            int(fields[3])
            return True
        else:
            return False
    except:
        return False

'''
Function to clean our contracts dataset
'''
def clean_contracts(line):
    try:
        fields = line.split(",")
        if len(fields) == 5:
            fields[0]
            return True
        else:
            return False
    except:
        return False

start = time()  # creating a time object to calculate the execution time

sc = pyspark.SparkContext()
print(sc.applicationId)
trans = sc.textFile("/data/ethereum/transactions")
transactions = trans.filter(clean_trans).map(lambda x: x.split(",")) # cleaning the transactions dataset and splitting the text file by a comma to get separate field values
features_t = trasactions.map(lambda x: (x[2], int(x[3]))) # extracting the transaction address and the transaction values
result_t = features_t.reduceByKey(lambda a,b: a+b) # summing the values

contract = sc.textFile("/data/ethereum/contracts")
contracts = contract.filter(clean_contracts).map(lambda x: x.split(",")) # cleaning the contracts dataset and splitting the text file by a comma to get separate field values
features_c = contracts.map(lambda x: (x[0], None)) # extracting the address field from the contracts dataset

result_c = result_t.join(features_c) # with address as the key, joining both the datasets (with only extracted values)

# (address, (summed_value, None))
'''
This means that we have extracted smart contarcts. (Addresses that are present in both the transactions and the contracts dataset)
'''
ten = result_c.takeOrdered(10, key = lambda x: -x[1][0]) # sorting the extracted smart contracts and their corresponsding values in descending order and extracting top 10 values
top_ten = sc.parallelize(ten).saveAsTextFile("spark_3") # saving our output file

stop = time() # finished the execution of our code and extracting total time spent in the job
print('Time:', (stop-start))
```

Output:-

```
(u'0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444', (84155100809965865822726776L, None))
(u'0xfa52274dd61e1643d2205169732f29114bc240b3', (45787484483189352986478805L, None))
(u'0x7727e5113d1d161373623e5f49fd568b4f543a9e', (45620624001350712557268573L, None))
(u'0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef', (43170356092262468919298969L, None))
(u'0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8', (27068921582019542499882877L, None))
(u'0xbfc39b6f805a9e40e77291aff27aee3c96915bdd', (21104195138093660050000000L, None))
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', (15562398956802112254719409L, None))
(u'0xbb9bc244d798123fde783fcc1c72d3bb8c189413', (11983608729202893846818681L, None))
(u'0xabbb6bebfa05aa13e908eaa492bd7a8343760477', (11706457177940895521770404L, None))
(u'0x341e790174e3a4d35b65fdc067b6b5634a61caea', (8379000751917755624057500L,  None))
```

Time Taken:-

Spark Job Number	Job Id	Time Taken (seconds)
1	http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1649894236110_1142	204.35
2	http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1649894236110_1154	246.71
3	http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1649894236110_1166	272.63
Average Time Taken:-		(204.35+246.71+272.63)/3 = **241.23**

Map Reduce Job Number	Job Id	Time Taken (seconds)
1	http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1649894236110_1265/ http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1649894236110_1400/	2433 + 204 = 2637
2	http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1649894236110_1452/ http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1649894236110_1587/	2087 + 256 = 2343
3	http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1649894236110_1617/ http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1649894236110_1786/	2541 + 219 = 2760
Average Time Taken:-		(2637+2343+2760) = **2580**

Observation: -

It is clearly evident from the average execution times of the above 3 jobs that Spark has taken significantly less amount of time to finish the jobs as compared to map reduce. The primary reason being that Spark uses system memory to read and write whereas map reduce uses HDFS

to store intermediate results. The mapper in map reduce gets data from HDFS and stores the output on system memory, which is again read by the reducer from the system memory and processed on HDFS. This read-write process is the reason that map reduce is almost **10x** slower than spark in our case. Average run time to get top 10 miners in map reduce (for 3 programs) was **2580 seconds** compared to just **241.23 seconds** in spark.

Fork the Chain: -

There have been several forks of Ethereum in the past. Identify one or more of these and see what effect it had on price and general usage. For example, did a price surge/plummet occur, and who profited most from this? (**10%**)

Methodology: -

First I have extracted the details (block id and fork date) of two forks called **Tangerine Whistle** and **Byzantium**. To study the effect of fork, I have used the gas price of all the block numbers in the block time frame of 7 days. This means we have gas prices and addresses for every block timestamps that are 7 days before and 7 days after the fork date.
I create a list with two elements which basically has the fork date and the fork name. Now I run a for loop in this list and extract the addresses and gas prices for all the blocks whose time stamp falls in 7-day range of the fork date. Further I map these values using the date as the key, take a summation for all the values on that day and then dividing it with the number of transactions to get the average gas price for a particular day. This loop will run twice and compute results for both the forks in our *forks* list.

Code:-

```
import pyspark
import datetime
import time

blocks = [2463000,4370000] # block number of the forks

forks = [ [datetime.datetime(2016,10,18),'Tangerine whistle'], [datetime.datetime(2017,10,16),'Byzantium'] ] # date of fork and name of fork

'''
Tangerine whistle
Oct-18-2016 01:19:31 PM +UTC
#2463000
'''

'''
Byzantium

Oct-16-2017 05:22:11 AM +UTC
Block number: 4,370,000
ETH price: $334.23 USD
ethereum.org on waybackmachine

'''
def clean_trans(line):
    try:
        fields = line.split(',')
        block_ts = datetime.datetime.fromtimestamp(int(fields[6])) # extracting block timestamp
        date_trans = time.gmtime(float(fields[6]))
        add = fields[2]
        gas_val = int(fields[5])

        check_fork = False # will toggle this true when the block time is 7 days before or after the fork date.

        for fork in forks:
            start_date = fork[0] + datetime.timedelta(-7)
            end_date = fork[0] + datetime.timedelta(7)
            print(start_date, end_date)

            if block_ts > start_date and block_ts < end_date: # check if
                fork_name = fork[1]
                print(fork_name)
                check_fork = True
                break
        if check_fork:
            return True

    except Exception as e:
        print(str(e))
        return False

```

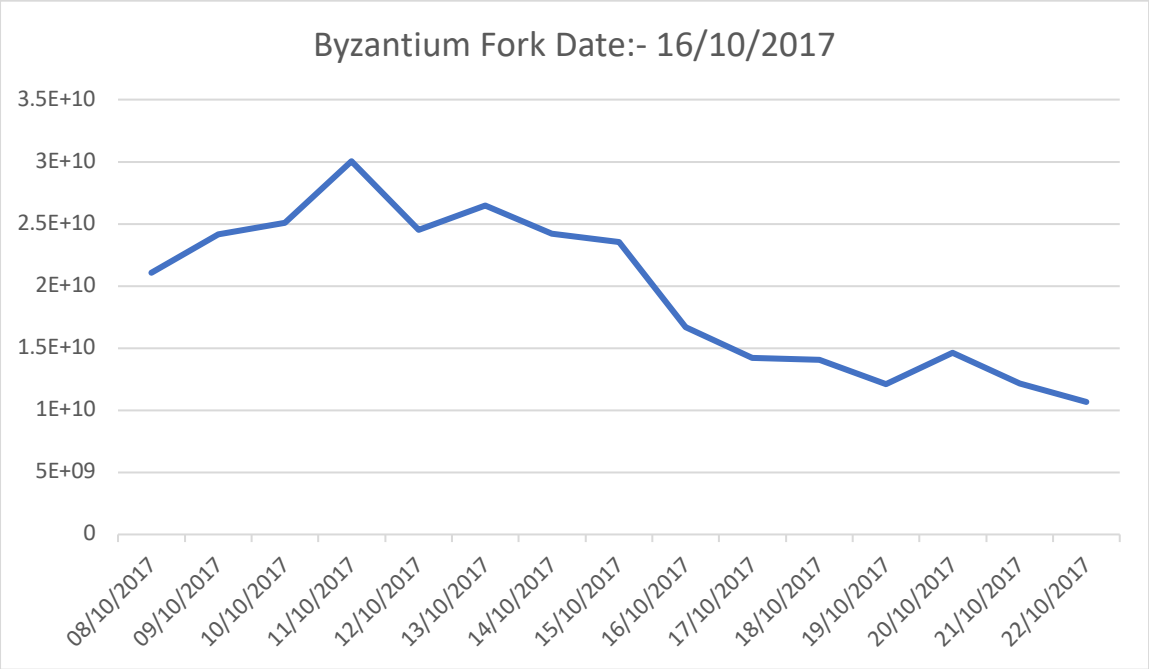
```python

def map_data(line):

    try:
        fields = line.split(',')
        block_ts = datetime.datetime.fromtimestamp(int(fields[6]))
        date_trans = time.gmtime(float(fields[6]))
        add = fields[2]
        # tvalue = int(fields[3])
        gas_val = int(fields[5])
        print(block_ts)

        check_fork = False

        for fork in forks:
            start_date = fork[0] + datetime.timedelta(-7)
            end_date = fork[0] + datetime.timedelta(7)
            print(start_date, end_date)

            if block_ts > start_date and block_ts < end_date:
                fork_name = fork[1]
                print(fork_name)
                check_fork = True

        if check_fork:
            datekey = str(date_trans.tm_year)+'-'+str(date_trans.tm_mon)+'-'+str(date_trans.tm_mday)
            return((datekey, add ), (gas_val,fork_name, 1))

    except Exception as e:
        print(str(e))

sc = pyspark.SparkContext()
print(sc.applicationId)

transactions = sc.textFile('/data/ethereum/transactions').filter(clean_trans) # extracts only those transactions whose block time stamp is in the range of +- 7 days of the fork date

map_trans = transactions.map(map_data) # maps data with date as the key and returns address, gas value, fork name and a '1' to count the addresses
print(map_trans.take(2))
# [(('2016-10-15', u'0x873144b9c49332a7fad5a07efb37c4c565e4fbd1'), (70153218482, 'Tangerine whistle', 1))]
get_gas = map_trans.map(lambda x: ((x[1][1], x[0][0]), (x[1][0], x[1][2]))) #
print(get_gas.take(2))
# [(('Tangerine whistle', '2016-10-15'), (70153218482, 1)), (('Tangerine whistle', '2016-10-15'), (63255441070, 1))]
# [(('Tangerine whistle', '2016-10-23'), (961528182055074, 35364)), (('Byzantium', '2017-10-12'), (8908169232568344, 363411))]

sum_gas = get_gas.reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1])) # summing the gas values for all the transactions of each block ids
print(sum_gas.take(2))

sort_map = sum_gas.map(lambda x: (x[0], float(x[1][0] / x[1][1]))).sortByKey(ascending = True) # sorting the values in ascending order

sort_map.saveAsTextFile('fork_out')
```

JOB ID Fork the chain: -
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1649894236110_3502

Output: -

```
(('Byzantium', '2017-10-10'), 25058292801.0)
(('Byzantium', '2017-10-11'), 30039329444.0)
(('Byzantium', '2017-10-12'), 24512657108.0)
(('Byzantium', '2017-10-13'), 26489325565.0)
(('Byzantium', '2017-10-14'), 24196653845.0)
(('Byzantium', '2017-10-15'), 23514174126.0)
(('Byzantium', '2017-10-16'), 16678649825.0)
(('Byzantium', '2017-10-17'), 14194850634.0)
(('Byzantium', '2017-10-18'), 14081137033.0)
(('Byzantium', '2017-10-19'), 12100849868.0)
(('Byzantium', '2017-10-20'), 14610296315.0)
(('Byzantium', '2017-10-21'), 12128683877.0)
(('Byzantium', '2017-10-22'), 10666208189.0)
(('Byzantium', '2017-10-8'), 21051671916.0)
(('Byzantium', '2017-10-9'), 24174030452.0)
(('Tangerine whistle', '2016-10-10'), 22952362759.0)
(('Tangerine whistle', '2016-10-11'), 24289033632.0)
(('Tangerine whistle', '2016-10-12'), 26037096488.0)
(('Tangerine whistle', '2016-10-13'), 74508549154.0)
(('Tangerine whistle', '2016-10-14'), 47491822158.0)
(('Tangerine whistle', '2016-10-15'), 67021626921.0)
(('Tangerine whistle', '2016-10-16'), 64860315135.0)
(('Tangerine whistle', '2016-10-17'), 62917625993.0)
(('Tangerine whistle', '2016-10-18'), 56109104952.0)
(('Tangerine whistle', '2016-10-19'), 26726320741.0)
(('Tangerine whistle', '2016-10-20'), 28388678071.0)
(('Tangerine whistle', '2016-10-21'), 27327186755.0)
(('Tangerine whistle', '2016-10-22'), 25568419660.0)
(('Tangerine whistle', '2016-10-23'), 27189463354.0)
(('Tangerine whistle', '2016-10-24'), 26806621407.0)
```

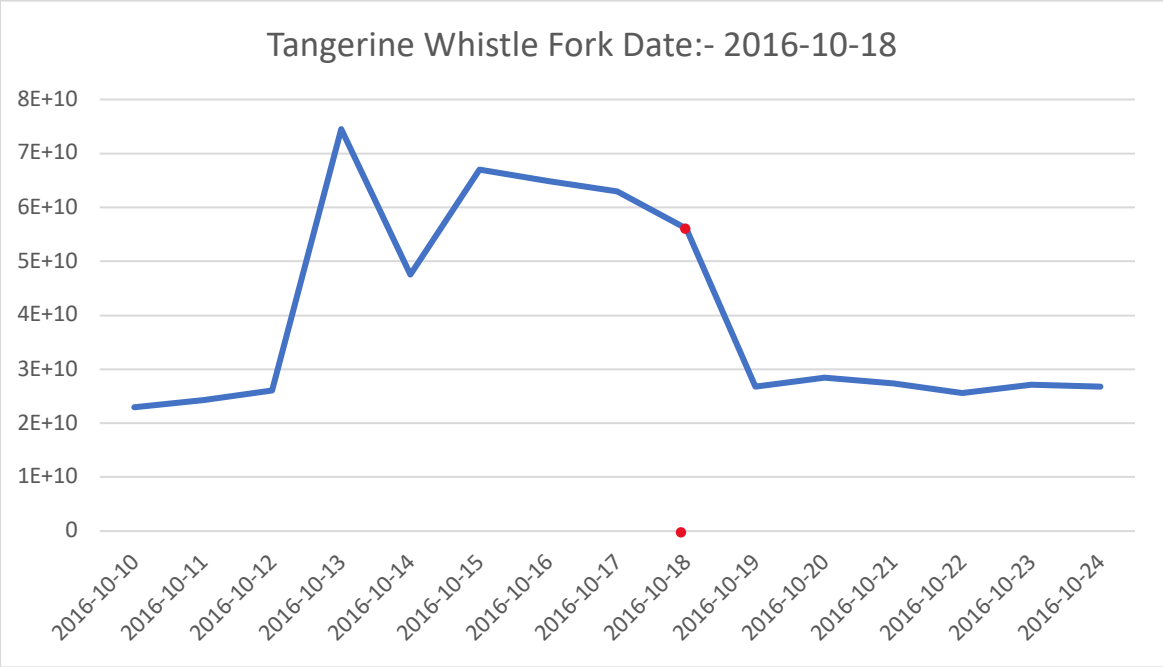
Byzantium: -



Inference:-

It is evident that the average gas price per day decreased after the day of fork. In fact the price started to decrease just a day before the fork date. Further the price keeps on decreasing for 3 days, after which it starts to rise again. It can be inferred that this fork was profitable for the addresses who made the greatest number of sales before the fork date. (as the price decreased after the fork). The ones who purchased (*to_address*) fields after the fork date, can also be profitable but that again depends whether they sell after the price reaches back to normal.

Tangerine Whistle: -



Inference: -

It is evident that the average gas price per day decreased after the day of fork similar to that of the **Byzantium** fork. The price falls significantly on the day of the fork after which it becomes constant. It can be inferred that this fork was profitable for the addresses who made the greatest number of sales before one or two days of the fork date (as the price decreased after the fork). The ones who purchased (*to_address*) fields after the fork date (purchasing at a low price), can also be profitable but that again depends if they sell after the price rises again.