ETHEREUM ANALYSIS

A)

Part A Time Analysis

The first task is to write a map-reduce program for Part A. I was able to aggregate the key of the field Timestamp and a value of one. This job shows the number of transactions that occurs every month between the start and the end of the dataset. I converted the timestamp into months and years. so, I yield both month and year plus a count of one and here are the results. I then plotted a bar chart of the results below. The program aggregates together all transactions in the same month and year and this is my analysis.

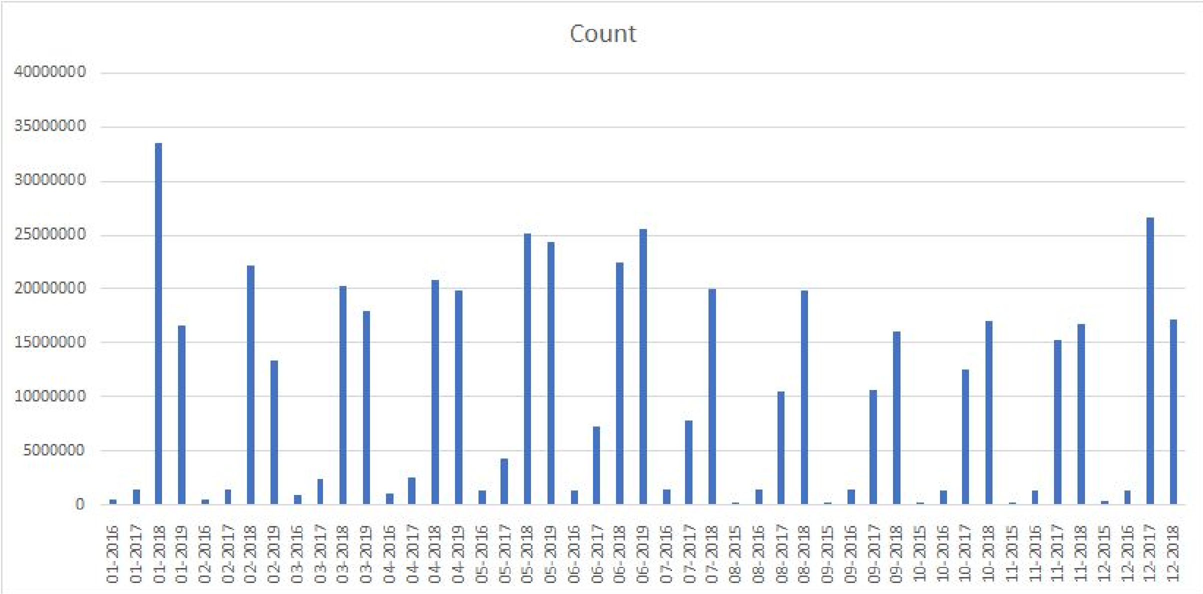
Map reduce program for part A

```
from mrjob.job import MRJob
import time

class partA(MRJob):

    def mapper(self, _, line):
        try:
            fields = line.split(',')
            if len(fields) == 7 :
                time_epoch = int(fields[6])
                day = time.strftime("%m-%Y",time.gmtime(time_epoch)) #returns the month and year
                yield (day, 1)
        except:
            pass
            #do nothing

    def combiner(self, day, counts):
        yield (day, sum(counts))

    def reducer(self, day, counts):
        yield (day, sum(counts))


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

Output from the program into a CSV file:

DD-YY	Count
01-2016	404816
01-2017	1409664
01-2018	33504270
01-2019	16569597
02-2016	520040
02-2017	1410048
02-2018	22231978
02-2019	13413899
03-2016	917170
03-2017	2426471
03-2018	20261862
03-2019	18029582
04-2016	1023096
04-2017	2539966
04-2018	20876642
04-2019	19830158
05-2016	1346796
05-2017	4245516
05-2018	25105717
05-2019	24332475
06-2016	1351536

Plotted bar chat:



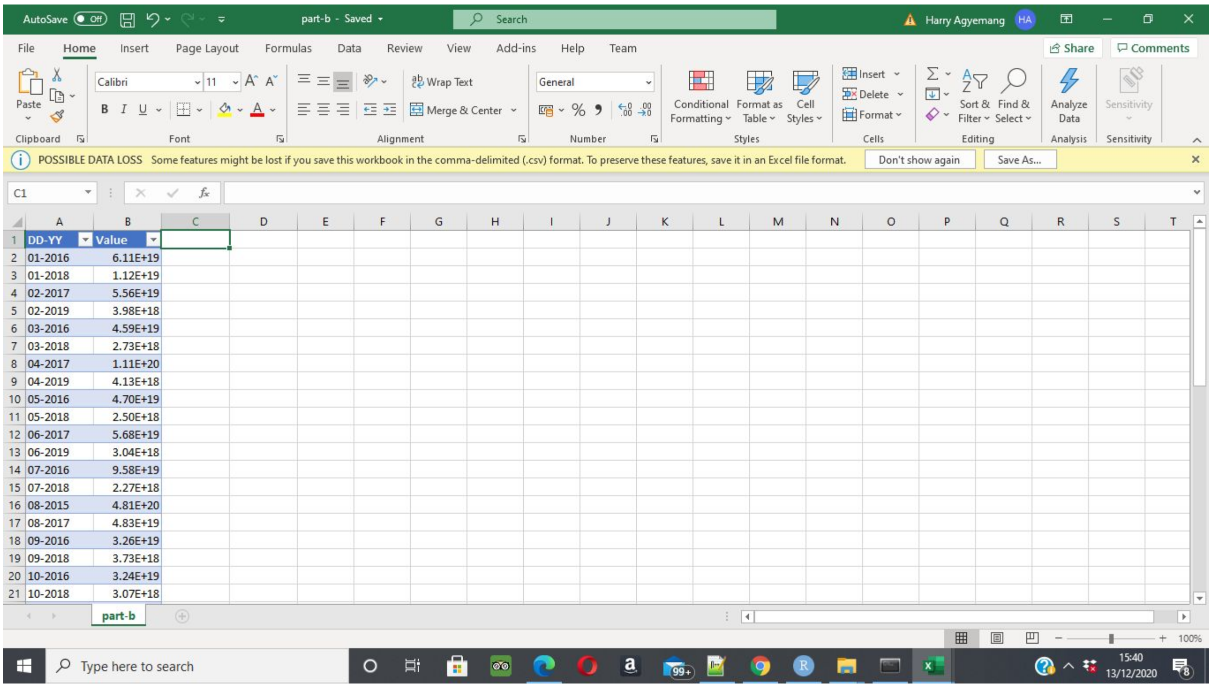
Part A2

For this part of the map-reduce program, I had to calculate the average value for each month. First I aggregated both timestamp and value. For the timestamp, I included each month and year between the start and end of the dataset. I yielded a count of 1. I calculated the total and count in the combiner then I divided the reduced which gave me this output.

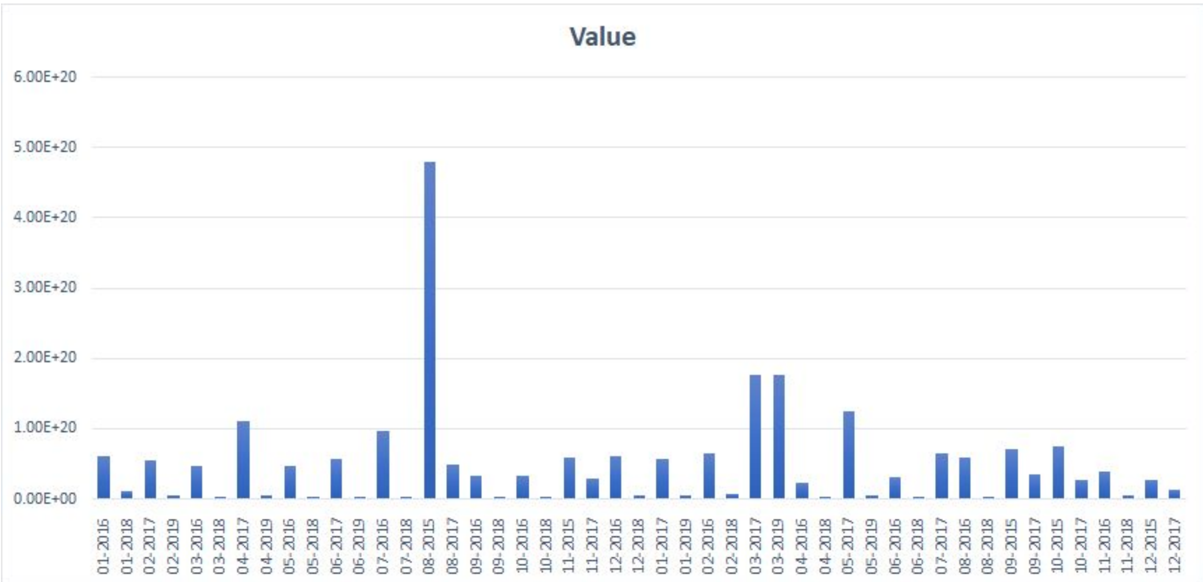
Map reduce program for Part A2

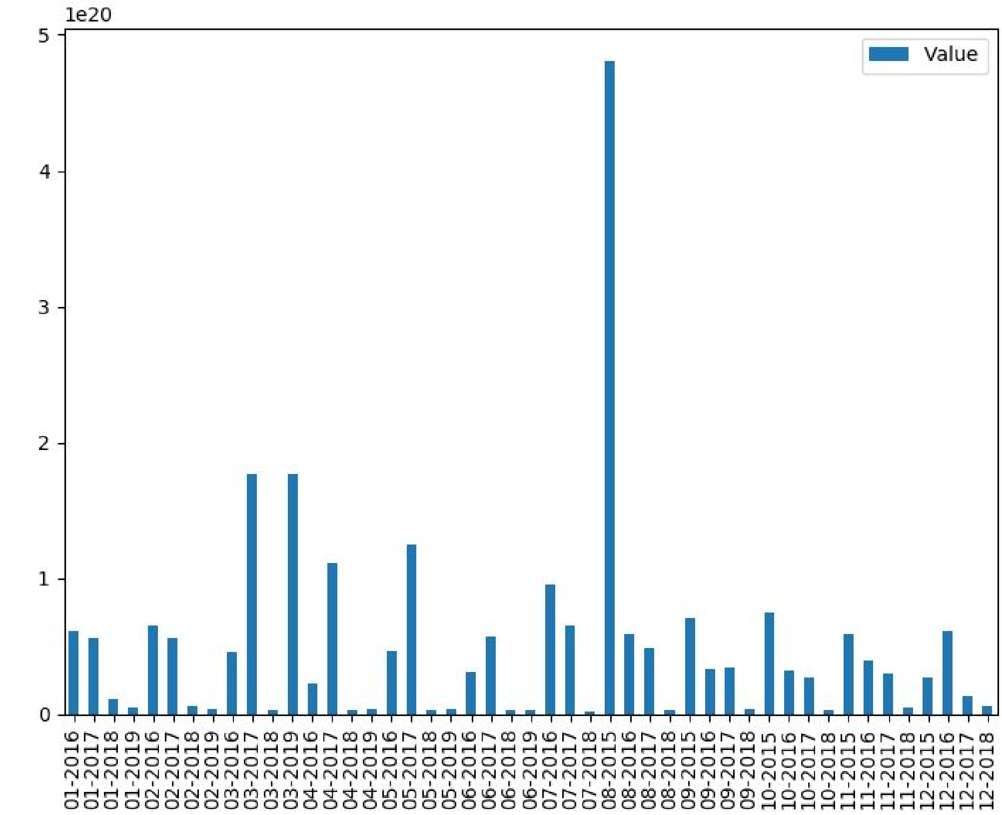
```
from mrjob.job import MRJob
import time

class partA1(MRJob):

    def mapper(self, _, line):
        try:
            fields = line.split(',')
            if len(fields) == 7 :
                value = int(fields[3])
                time_epoch = int(fields[6])
                day = time.strftime("%m-%Y",time.gmtime(time_epoch))
                yield (day, (value,1))

        except:
            pass

    def combiner(self, day, values):
        count = 0
        total = 0
        for value in values:
            count += value[1]
            total += value[0]
        yield (day, (total, count))


    def reducer(self, day, values):
        count = 0
        total = 0
        for value in values:
            count += value[1]
            total += value[0]
        yield (day, total/count)


if __name__ == '__main__':
    partA1.run()
```

Output from the program into a CSV file:



Plotted bar chat:





B)

Part B1

I wrote a map reduce program which does the first job of part b. The goal was to aggregate the address and value from the transaction dataset. The only problem was I had to sort the values as it kept yielding the value 0, so I managed to put a filter to stop this happening. This leaves any values of 0 out of the output.

Map reduce program for B1

```python
from mrjob.job import MRJob

class partB1(MRJob):
    def mapper(self, _, line):
        try:
            fields = line.split(',')
            if len(fields) == 7 :
                toAddress = fields[2]
                value = int(fields[3])
                if value == 0:
                    pass
                else:
                    yield (toAddress,value)
        except:
            pass
            #do nothing
    def combiner(self, toAddress, counts):
        yield (toAddress, sum(counts))

    def reducer(self,toAddress, counts):
        yield (toAddress, sum(counts))

if __name__ == '__main__':
    partB1.JOBCONF= {'mapreduce.job.reduces': '10' }
    partB1.run()
```

The output was saved onto a tsv file.

"\"0x0000000000000000000000000000000000000001\""	1369282904733190138
"\"0x0000000000000000000000000000000000000003\""	1693486824927768122
"\"0x0000000000000000000000000000000000000005\""	1722267911686475579
"\"0x0000000000000000000000000000000000000007\""	1
"\"0x0000000000000000000000000000000000000009\""	1
"\"0x000000000000000000000000000000000000000a\""	1000000000000001
"\"0x000000000000000000000000000000000000000c\""	5122790699342116 9
"\"0x000000000000000000000000000000000000000e\""	1
"\"0x0000000000000000000000000000000000000027\""	6616907518533786
"\"0x000000000000000000000000000000000000002a\""	4200000000000000
"\"0x00000000000000000000000000000000000000b1\""	1200000000000002
"\"0x00000000000000000000000000000000000000fe\""	4000000000000000
"\"0x000000000000000000000000000000000000041b\""	1000000000000000
"\"0x0000000000000000000000000000000000000438\""	1000000000000
"\"0x0000000000000000000000000000000000000eee\""	55000000000000064
"\"0x0000000000000000000000000000000000001231\""	5663200000000000
"\"0x0000000000000000000000000000000000001822\""	7000000000000000
"\"0x0000000000000000000000000000000000003105\""	4234750000000000
"\"0x0000000000000000000000000000000000004b3d\""	564395000000000000
"\"0x0000000000000000000000000000000000004b61\""	3000000000000000
"\"0x000000000000000000000000000000000000521c\""	100000000000000000
"\"0x000000000000000000000000000000000000537d\""	6000000000000000
"\"0x0000000000000000000000000000000000005392\""	7627067000000000
"\"0x00000000000000000000000000000000000053fc\""	200000000000000000
"\"0x0000000000000000000000000000000000005435\""	9500000000000000
"\"0x0000000000000000000000000000000000005666\""	20000000000000000
"\"0x00000000000000000000000000000000000056e9\""	100000000000000
"\"0x000000000000000000000000000000000000576e\""	4911784000000000
"\"0x0000000000000000000000000000000000005857\""	100000000000000000
"\"0x00000000000000000000000000000000000058aa\""	46309000000000000
"\"0x000000000000000000000000000000000000598c\""	1070679000000000
"\"0x00000000000000000000000000000000000059cd\""	166590208554880
"\"0x0000000000000000000000000000000000005a69\""	131682700000000000
"\"0x0000000000000000000000000000000000005a8e\""	49500000000000000
"\"0x0000000000000000000000000000000000005c01\""	1000000000000000000
"\"0x0000000000000000000000000000000000007f77\""	58088508000000000
"\"0x000000000000000000000000000000000000abd6\""	2650000000000000
"\"0x000000000000000000000000000000000000b470\""	765205000000000
"\"0x000000000000000000000000000000000000b71e\""	1000000000000000
"\"0x000000000000000000000000000000000001b77d\""	537361900000000
"\"0x000000000000000000000000000000000001dead\""	1010000000000000
"\"0x000000000000000000000000000000000001f327\""	7738140000000000
"\"0x000000000000000000000000000000000001fcba\""	4427680000000000

Part B2

Here I continued to write a map-reduce program but this was a bit different. Previously I aggregated the address and values from the transaction. But for this job I want to join it into the contracts. The first thing was to get the transactions dataset as files because I outputted the first job onto a \t files. Then I aggregated the contacts dataset. After obtaining the aggregate of the transaction the next step was to perform a reparation join between the aggregate from job 1 and contracts dataset. I completed a join of the to_address field from the output of Job 1 with the address field of contracts. For 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.

Map reduce program for the joining

```python
from mrjob.job import MRJob

class partB2(MRJob):
    def mapper(self, _, line):
        try:
            if(len(line.split('\t'))==2):
                fields=line.split('\t')
                join_key = fields[0].strip('"\"\\\"')
                join_value= int(fields[1])

                yield (join_key, join_value)

            elif len(line.split(',')) == 5:
             fields = line.split(',')
             join_key = fields[0].strip('"\"\\\"')
             yield(join_key,(join_val,2))

        except:
            pass
            #do nothing
    def reducer_sum(self, company, values):
        years = []
        sector = 0

        for value in values:
            if value[1]==1:
                sector=value[0]
            elif value[1]==2:
               years.append(value[0])
        if sector > 0 and len(years) != 0:
            yield (company, sector)

if __name__ == '__main__':
    partB2.JOBCONF= {'mapreduce.job.reduces': '10' }
    partB2.run()
```

Output onto a tsv file:

Part B3.

Lastly I wrote a map reduce job to find the top ten values within the dataset. I used the already now filtered join to find the top values in the value column. The third job will take as input the now filtered address aggregates and sort these via a top ten reducer. The goal is to find top 10 smart contracts by total Ether received.

Map reduce program

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

class partB3(MRJob):
    def mapper(self, _, line):
        try:
            if(len(line.split('\t'))==2):
                fields=line.split('\t')
                join_key = fields[0].strip('"')
                join_value= int(fields[1])
                if join_value!=0:
                    yield (None,(join_key, join_value))


        except:
            pass
            #do nothing
    def combiner(self, _, values):
        sorted_values = sorted(values, reverse = True, key = lambda tup:tup[1])
        i = 0
        for value in sorted_values:
            yield ("top", value)
            i += 1
            if i >= 10:
                break

    def reducer(self, _, values):
        sorted_values = sorted(values, reverse = True, key = lambda tup:tup[1])
        i = 0
        for value in sorted_values:
            yield ("{} - {} ".format(value[0],value[1]),None)

            i += 1
            if i >= 10:
                break

if __name__ == '__main__':
    partB3.JOBCONF= {'mapreduce.job.reduces': '10' }
    partB3.run()
```

Output from the program:

```
STDERR: 20/12/11 11:41:28 DEBUG util.NativeCodeLoader: Loaded the native-hadoop library
"0xaa1a6e3e6ef20060f7f8d8c835d2d22fd5116444 - 84155100809965865822726776 "      null
"0x3f5ce5fbfe3e9af3971dd833d26ba9b5c936f0be - 58343313022529151499090724 "      null
"0xfa52274dd61e1643d2205169732f29114bc240b3 - 45787484483189352986478805 "      null
"0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef - 43170356092262468919298969 "      null
"0x876eabf441b2ee5b5b0554fd502a8e0600950cfa - 40157678878619363035574179 "      null
"0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8 - 27068921582019542499882877 "      null
"0xbfc39b6f805a9e40e77291aff27aee3c96915bdd - 21104195138093660050000000 "      null
"0xc257274276a4e539741ca11b590b9447b26a8051 - 15678312284007170625031927 "      null
"0xe94b04a0fed112f3664e45adb2b8915693dd5ff3 - 15562398956802112254719409 "      null
"0x5e032243d507c743b061ef021e2ec7fcc6d3ab89 - 11829169343782923136145277 "      null
Removing HDFS temp directory hdfs:///user/ha004/tmp/mrjob/part-b3.ha004.20201211.11393
8050...
```

C)

Part C.

I wrote a program which was to yield the top ten values by size in the block dataset. Here I was able to aggregate the miner and size of the blocks and use a sort function to find the top 10 by size. I first aggregate blocks to see how much each miner has been involved in. The goal was to sort the list to obtain the most active miners.

Map reduce program- top 10 miners by the size of the blocks mined

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

class partC(MRJob):


    def mapper(self, _, line):
        try:
            fields = line.split(',')
            if len(fields) == 9:
                miner = fields[2]
                size = int(fields[4])
                yield (miner,size)
        except:
            pass

    def steps(self):
        return [
            MRStep(
                mapper=self.mapper, combiner=self.combine_counts,
                reducer=self.reducer_sum_counts
            ),
            MRStep(
                reducer=self.reduce_sort_counts
            )
        ]

    def combine_counts(self, miner, counts):
        yield miner, sum(counts)

    def reducer_sum_counts(self, key, values):
        yield None, (sum(values), key)

    def reduce_sort_counts(self, _, size_counts):
        i = 0
        for count, key in sorted(size_counts, reverse=True):
            yield (key,int(count))

            i += 1
            if i >= 10:
                break

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

Output from map reduce

```
"0xea674fdde714fd979de3edf0f56aa9716b898ec8"    23989401188
"0x829bd824b016326a401d083b33d092293333a830"    15010222714
"0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c"    13978859941
"0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5"    10998145387
"0xb2930b35844a230f00e51431acae96fe543a0347"    7842595276
"0x2a65aca4d5fc5b5c859090a6c34d164135398226"    3628875680
"0x4bb96091ee9d802ed039c4d1a5f6216f90f81b01"    1221833144
"0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb"    1152472379
"0x1e9939daaad6924ad004c2560e90804164900341"    1080301927
"0x61c808d82a3ac53231750dadc13c777b59310bd9"    692942577
Removing HDFS temp directory hdfs:///user/ha004/tmp/mrjob/part-c.ha004.20201211.101058.124618...
(lousie) bash-4.2$ python part-c.py -r hadoop hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/blocks
```

Application ID:

Part A- python part-a.py -r hadoop hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/transactions, http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1607539937312_1834/
Part A2- python part-b.py -r hadoop hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/transactions, http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1607539937312_1838/
Part C- python part-c.py -r hadoop hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/blocks, http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1607539937312_1849/
Part B1-python part-b1.py -r hadoop hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/transactions > out4.tsv,http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1607539937312_1897/
Part B2- python part-b2.py -r hadoop hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/contracts /homes/ha004/out4.tsv > Outputb3.tsv , http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1607539937312_1924/
PartB3-python part-b3.py -r hadoop hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/contracts /homes/ha004/Outputb3.tsv , http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1607539937312_1935/

D)
Miscellaneous Analysis
1) **Fork the Chain.**

Cryptocurrency is changing the world. Forks occur when the user base decides that the fundamentals about the cryptocurrency need to change. They are often predicted by large price fluctuations and are very controversial. A fork is a change in the blockchain's protocol that the software uses to decide whether a transaction is valid or not. A soft fork is any change that is backwards compatible. For example, when a soft fork takes place old nodes still work and recognizes new transactions as being valid. But for blocks that are minded, that will be considered as invalid by the updated nodes created. For it to be successful soft forks require the majority of the network's hash power. There is a risk of it being a small chain later becoming a hard fork. A hard fork is any change that breaks the backward compatibility. It has nodes that run the old software and it will see if any new transactions are considered as invalid. For it to be valid the chains would need to update. If for example, the community decides that they want to consider using the old rules then the chain will ultimately split into two separate currencies. A hard fork requires majority support from the coin holders that have a connection to the coin network. For a hard dork to be adopted, there must be a sufficient number of nodes needed to update the newest version of the protocol. For any nodes that don't choose to update will be unable to use the new blockchain. For any changes to happen a majority of the community needs to agree before any fundamental changes can be implemented as they can have huge consequences on the risk of a hard break. The uncertainty of a hard fork has created an effect on digital asset prices. For example, even if there is an initial dip, the long term impact of the hard fork will likely be positive due to the update improvements for scalability and cost. For example, their price has increased as the DAO gained momentum. Despite the potentially major changes the price of ethereum has remained relatively stable. (Commodity.com. 2020)

The effect it had on price and general usage
Ethereum blockchain has a hard fork and is used to apply a series of updates. One change that happened was the converting from a proof of work to a proof of stake algorithm. After the fork of the new ethereum, the USD had a steady rise while the ETC flattered. The changing in ethereum's architecture had opened up new uses and markets for it.

Double the tokens

Depending on how the fork is structured there sometimes is a change to double on tokens for example increase in the size of the block can happen and owners can end up with duplicate numbers after the fork. This can cause investors to buy even more tokens and cause prices to ride. The idea is that the added value of the new tokens will cause a price drop of the original tokens caused by the fork. Investors pay attention to the market due to the huge investment opportunities. They will be aware of the upcoming fork and be prepared to take advantage of the situation. Feeding frenzy can happen when buyers try to get their hands on a particular coin. For example, with BTC it saw a rise in value and reached all-time high prices.

The price of ethereum level is between 300 to 400 dollar the profitability is higher giving miners a strong incentive to sell at a higher price. The nature of mining is changing and more people are joining. People are picking up mid-range GPUs to mine ethereum. For this, the large quantity of people doing this and the more profitability of mining means the miners are less likely to sell coins and this causes a wait on sales until the prices rise again.

For example, a whale who knows that a fork is about to happen. This will result in him trying to get new coins for every origin coin they have. This gives them a strong incentive to increase the stake in the parent token. It causes them to buy every token available to find. Their large size means that they can up the price of the parent currency higher when leading up to the fork as whales and dolphins buy everything they can. (OpenLedger DEX. 2020)

Whales have a big effect on the value of tokens if they proceed to dump both new token and parent token on every exchange; this can cause the value of both fork and parent token to crash in value. Over time they become stabilise as traders use their profits to purchase more currency. Another example of price changes is ethereum as they had a dramatic change after DAO hack and ethereum classic contentious hard fork. Because of this, they were viewed as detrimental to the main chain.

For example, things change when Bitcoin cash forks from bitcoin and finding the best way to increase the number of transactions. One solution was proposed, e.g. a larger block size. Based on ethereum DAO hack fork

Fork dates

September 8th 2015. The ice age was the first unplanned fork for ethereum blockchain, which provided security and ped updates to the network.

Atlantis- September 2019 Atlantis hard fork event required all software users to upgrade their clients which were used to stay with the current network. This included better security, stability and high network performance for a high volume of traffic.

Homestead- March 15th 2016 is when homestead happened. This is considered as phase 2 of ethereum development. This included critical updates, the removal of centralisation on the network which allows users to hold and exchange with ethereum and to write and deploy smart contacts. (Viens, A., 2020)

2) **Gas Guzzlers.**

Ethereum transaction fees increase when the network is busier. This is caused by more people making transactions for example sending tokens, trading and depositing their assets. With the price of ethereum has risen over the past few years the average transaction costs 40 cents and takes 1 minute to process which is high. Overtime miners will be accepting lower gas prices to keep the price around 50 cents per transaction.

Gas refers to the fee to price value to successfully conduct a transaction. Price is a small fraction of the currency. Gas is used to allocate resources of the ethereum virtual machine. One benefit of smart contracts is they can be automated payments without the need for notices or collection expenses. Smart contacts can have a massive risk for example the possibility that contact hacking or a programming error could happen. With the security of blockchains, it isn't likely to happen and is possible the cause of a coding error. Before a smart contract executed on a blockchain, the payment of the transaction fee must be added to the chain and executed. The more complex the smark contact which is based on the transaction steps performed affects the gas that must be paid to execute the smart contract. Gas is important as it presents overly complex smart contacts from overwhelming the ethereum virtual machine. This also depends on the unit of ethereum and its market value. Which means miners can decide to increase or decrease the use of gas according to its need but also avoid situations in which an increase in the price of ethereum causes a change to gas prices. (Reddit. 2020)

From part b I was able to identify the smart contacts from transactions and they came with high values meaning a lot of these transactions were large. I completed a top ten smart contacts for ethereum received and found the highest amount.

Price forecasting

Dataset:https://datahub.io/cryptocurrency/ethereum#resource-ethereum

My evaluation:
After finding this dataset I utilised spark mllib to build a price forecasting model and train this on the dataset. The dataset has dates from when it was started till now. My goal was to predict the pricing and I have had some success modeling. I can give good results for dates in the middle but poor ones for extreme values

Here are the results from my price forecasting completed in pyspark.

```
[1] import pandas as pd
    import numpy as np
    import matplotlib.pyplot as plt
    import time
    import datetime
```

```
[2] !pip install pyspark
    from pyspark.sql import SparkSession
    spark = SparkSession.builder.getOrCreate()
```

```
Collecting pyspark
  Downloading https://files.pythonhosted.org/packages/f0/26/198fc8c0b98580f617cb03cb298c6056587b8f0447e20fa40c5b634ced77/pyspark-3.0.1.tar.gz (204.2MB)
     |████████████████████████████████| 204.2MB 64kB/s
Collecting py4j==0.10.9
  Downloading https://files.pythonhosted.org/packages/9e/b6/6a4fb90cd235dc8e265a6a2067f2a2c99f0d91787f06aca4bcf7c23f3f80/py4j-0.10.9-py2.py3-none-any.whl (198kB)
     |████████████████████████████████| 204kB 31.5MB/s
Building wheels for collected packages: pyspark
  Building wheel for pyspark (setup.py) ... done
  Created wheel for pyspark: filename=pyspark-3.0.1-py2.py3-none-any.whl size=204612243 sha256=6c6b414d2732fc0853d264236f1cbd3e425b749b70804588db2253730797d556
  Stored in directory: /root/.cache/pip/wheels/5e/bd/07/031766ca628adec8435bb40f0bd83bb676ce65ff4007f8e73f
Successfully built pyspark
Installing collected packages: py4j, pyspark
Successfully installed py4j-0.10.9 pyspark-3.0.1
```

```
[11] data = spark.read.csv("ethereum_csv.csv",header=True, inferSchema=True)
```

```
data.show()
```

date	txVolume(USD)	adjustedTxVolume(USD)	txCount	marketcap(USD)	price(USD)	exchangeVolume(USD)	generatedCoins	fees	activeAddresses	medianTxValue
2015-07-30	null	null	null	null	null	null	39311.09375	null	null	
2015-07-31	null	null	null	null	null	null	36191.71875	null	null	
2015-08-01	null	null	null	null	null	null	27702.1875	null	null	
2015-08-02	null	null	null	null	null	null	28227.1875	null	null	
2015-08-03	null	null	null	null	null	null	27976.71875	null	null	
2015-08-04	null	null	null	null	null	null	28478.4375	null	null	
2015-08-05	null	null	null	null	null	null	27528.59375	null	null	
2015-08-06	null	null	null	null	null	null	27075.46875	null	null	
2015-08-07	5684487.22712	5684487.22712	2050	0.0	2.83	164329.0	27437.65625	37.3184135113	1086	41.61171
2015-08-08	3.78996817885E7	3.78996817885E7	2881	1.67911008E8	2.79	674188.0	27943.4375	68.0999700719	826	
2015-08-09	823605.517746	823605.517746	1329	4.26376E7	0.706136	532170.0	27178.28125	14.0989497032	732	7.
2015-08-10	1194050.4877	1194050.4877	2037	4.313E7	0.713989	405283.0	27817.34375	31.1651444384	1001	0.3597844
2015-08-11	1053129.87257	1053129.87257	4963	4.27965E7	0.708087	1463100.0	28027.8125	11.311448586	2342	0.0340063
2015-08-12	797474.189141	797474.189141	2036	6.40184E7	1.06	2150620.0	27370.9375	32.5869924044	906	7.634961
2015-08-13	2182919.67454	2182919.67454	2842	7.39354E7	1.22	4068680.0	28268.125	24.9291691099	1274	9.94756
2015-08-14	4162001.56084	4162001.56084	3174	1.09594E8	1.81	4637030.0	31106.71875	14.3548341128	1594	18.75014
2015-08-15	1.39024105328E7	1.39024105328E7	2284	1.0916E8	1.8	2554360.0	28512.65625	8.83659302805	1313	
2015-08-16	2898479.71124	2898479.71124	2440	1.02028E8	1.68	3550790.0	27094.53125	6.67723792491	1202	12.00702
2015-08-17	5383055.46816	5383055.46816	2512	9.5819696E7	1.58	1942830.0	28118.90625	7.64281183963	1249	11.69836
2015-08-18	2409105.86945	2409105.86945	2494	8.81432E7	1.22	1485680.0	26162.8125	11.9060713402	1294	7.823492

```
[18] df1 = data.dropna()
```

```
[19] df1.show()
```

date	txVolume(USD)	adjustedTxVolume(USD)	txCount	marketcap(USD)	price(USD)	exchangeVolume(USD)	generatedCoins	fees	activeAddresses	medianTxValue
2015-08-07	5684487.22712	5684487.22712	2050	0.0	2.83	164329.0	27437.65625	37.3184135113	1086	41.61171
2015-08-08	3.78996817885E7	3.78996817885E7	2881	1.67911008E8	2.79	674188.0	27943.4375	68.0999700719	826	
2015-08-09	823605.517746	823605.517746	1329	4.26376E7	0.706136	532170.0	27178.28125	14.0989497032	732	7.
2015-08-10	1194050.4877	1194050.4877	2037	4.313E7	0.713989	405283.0	27817.34375	31.1651444384	1001	0.3597844
2015-08-11	1053129.87257	1053129.87257	4963	4.27965E7	0.708087	1463100.0	28027.8125	11.311448586	2342	0.0340063
2015-08-12	797474.189141	797474.189141	2036	6.40184E7	1.06	2150620.0	27370.9375	32.5869924044	906	7.634961
2015-08-13	2182919.67454	2182919.67454	2842	7.39354E7	1.22	4068680.0	28268.125	24.9291691099	1274	9.94756
2015-08-14	4162001.56084	4162001.56084	3174	1.09594E8	1.81	4637030.0	31106.71875	14.3548341128	1594	18.75014
2015-08-15	1.39024105328E7	1.39024105328E7	2284	1.0916E8	1.8	2554360.0	28512.65625	8.83659302805	1313	
2015-08-16	2898479.71124	2898479.71124	2440	1.02028E8	1.68	3550790.0	27094.53125	6.67723792491	1202	12.00702
2015-08-17	5383055.46816	5383055.46816	2512	9.5819696E7	1.58	1942830.0	28118.90625	7.64281183963	1249	11.69836
2015-08-18	2409105.86945	2409105.86945	2494	8.81432E7	1.22	1485680.0	26162.8125	11.9060713402	1294	7.823492
2015-08-19	1691424.30403	1691424.30403	3246	8.4663904E7	1.17	1486240.0	27282.03125	15.4309506695	1350	6.212439
2015-08-20	592806.204465	592806.204465	2303	9.0809E7	1.25	2843760.0	21057.03125	6.85556694271	1167	6.24843
2015-08-21	894127.676986	894127.676986	3919	1.07266E8	1.48	2020970.0	26690.9375	7.85504172471	1425	
2015-08-22	874020.858748	874020.858748	3579	1.01404E8	1.4	948310.0	27550.46875	42.5471718296	1414	5.7972
2015-08-23	697311.417317	697311.417317	4190	9.9894496E7	1.38	1589300.0	26255.625	8.0497676729	1570	5.135572
2015-08-24	630283.148711	630283.148711	4432	9.77916E7	1.35	924920.0	27333.28125	8.8762854198	1436	4.035544
2015-08-25	505362.445352	505362.445352	4487	8.9321904E7	1.23	1307180.0	26736.09375	8.25975582511	1580	4.19665
2015-08-26	508937.45783	508937.45783	4156	8.23844E7	1.13	1056750.0	26137.96875	7.44772927856	1527	3.15800

```
[26] feature_columns = data.columns[1:-1] # here we omit the final column
     from pyspark.ml.feature import VectorAssembler
     assembler = VectorAssembler(inputCols=feature_columns,outputCol="features")
```

```
[27] from pyspark.sql.functions import col, unix_timestamp, to_date,to_timestamp


     df1 = df1.withColumn('date', to_timestamp(unix_timestamp(col('date'), 'yyyy-MM-dd').cast("timestamp")))
```

```
[28] df1 = df1.withColumn('date', (unix_timestamp(col('date'), 'yyyy-MM-dd').cast("int")))
```

```
[29] df1.show()
```

```
[29] df1.show()
```

date	txVolume(USD)	adjustedTxVolume(USD)	txCount	marketcap(USD)	price(USD)	exchangeVolume(USD)	generatedCoins	fees	activeAddresses	medianTxValue
1438905600	5684487.22712	5684487.22712	2050	0.0	2.83	164329.0	27437.65625	37.3184135113	1086	41.61171
1438992000	3.78996817885E7	3.78996817885E7	2881	1.67911008E8	2.79	674188.0	27943.4375	68.0999700719	826	
1439078400	823605.517746	823605.517746	1329	4.26376E7	0.706136	532170.0	27178.28125	14.0989497032	732	7.
1439164800	1194050.4877	1194050.4877	2037	4.313E7	0.713989	405283.0	27817.34375	31.1651444384	1001	0.3597844
1439251200	1053129.87257	1053129.87257	4963	4.27965E7	0.708087	1463100.0	28027.8125	11.311448586	2342	0.03400636
1439337600	797474.189141	797474.189141	2036	6.40184E7	1.06	2150620.0	27370.9375	32.5869924044	906	7.634961
1439424000	2182919.67454	2182919.67454	2842	7.39354E7	1.22	4068680.0	28268.125	24.9291691099	1274	9.94756
1439510400	4162001.56084	4162001.56084	3174	1.09594E8	1.81	4637030.0	31106.71875	14.3548341128	1594	18.75014
1439596800	1.39024105328E7	1.39024105328E7	2284	1.0916E8	1.8	2554360.0	28512.65625	8.83659302805	1313	
1439683200	2898479.71124	2898479.71124	2440	1.02028E8	1.68	3550790.0	27094.53125	6.67723792491	1202	12.00702
1439769600	5383055.46816	5383055.46816	2512	9.5819696E7	1.58	1942830.0	28118.90625	7.64281183963	1249	11.69836
1439856000	2409105.86945	2409105.86945	2494	8.81432E7	1.22	1485680.0	26162.8125	11.9060713402	1294	7.823492
1439942400	1691424.30403	1691424.30403	3246	8.4663904E7	1.17	1486240.0	27282.03125	15.4309506695	1350	6.212439
1440028800	592806.204465	592806.204465	2303	9.0809E7	1.25	2843760.0	21057.03125	6.85556694271	1167	6.24843
1440115200	894127.676986	894127.676986	3919	1.07266E8	1.48	2020970.0	26690.9375	7.85504172471	1425	
1440201600	874020.858748	874020.858748	3579	1.01404E8	1.4	948310.0	27550.46875	42.5471718296	1414	5.7972
1440288000	697311.417317	697311.417317	4190	9.9894496E7	1.38	1589300.0	26255.625	8.0497676729	1570	5.135572
1440374400	630283.148711	630283.148711	4432	9.77916E7	1.35	924920.0	27333.28125	8.8762854198	1436	4.035544
1440460800	505362.445352	505362.445352	4487	8.9321904E7	1.23	1307180.0	26736.09375	8.25975582511	1580	4.19665
1440547200	508937.45783	508937.45783	4156	8.23844E7	1.13	1056750.0	26137.96875	7.44772927856	1527	3.15800

only showing top 20 rows

```
[30] data_2 = assembler.transform(df1)
```

```
[31] train, test = data_2.randomSplit([0.7, 0.3])
```

```
[32] from pyspark.ml.regression import LinearRegression
     algo = LinearRegression(featuresCol="features", labelCol="price(USD)")
```

```
[33] model = algo.fit(train)
```

```
[34] evaluation_summary = model.evaluate(test)
```

```
[35] evaluation_summary.meanAbsoluteError

     1.3145307132991313e-12
```

```
[36] evaluation_summary.rootMeanSquaredError

     2.046047940721807e-12
```

```
[37] evaluation_summary.r2

     1.0
```

```
[38] predictions = model.transform(test)
```

```
[39] predictions.select(predictions.columns[13:]).show() # here I am filtering out some columns just for the figure to fit
```

paymentCount	blockSize	blockCount	features	prediction
1973	3508878	5256	[3.78996817885E7,...	2.7900000000000515
2486	3519008	5286	[2182919.67454,21...	1.2199999999994329
3147	3392043	5192	[1691424.30403,16...	1.1699999999992443
3809	3396674	5088	[894127.676986,89...	1.4799999999989433
4025	3411809	5013	[697311.417317,69...	1.3799999999989223
4296	3464583	5097	[505362.445352,50...	1.229999999998985
4040	3345268	5003	[508937.45783,508...	1.129999999998905
4883	2740989	3577	[374923.646468,37...	1.3499999999977117
5094	3651888	5003	[249301.081565,24...	1.209999999998938
5247	3740946	4851	[252012.348897,25...	0.940565999998791
5679	3660494	4956	[523583.14265,523...	0.9419769999988741
5102	3622311	4808	[268368.722163,26...	0.9068649999987428
4938	3730754	5078	[332401.248822,33...	0.8496029999990211
6194	3860006	5210	[425540.887368,42...	0.7343069999989957
6586	4063481	5152	[191066.731219,19...	0.6095009999988731
5849	3795610	5073	[95953.5974372,95...	0.6274609999988541
6229	4053860	5148	[172115.376764,17...	0.6345149999988883
5625	3842397	4990	[245668.720604,24...	0.48962899999879866
5687	3984632	5171	[53939.4236085,53...	0.5396809999989673
5615	3757017	4908	[443466.658196,44...	0.6197429999987856

References

Viens, A., 2020. *Mapping The Most Important Ethereum Forks*. [online] Visual Capitalist. Available at: <https://www.visualcapitalist.com/mapping-major-ethereum-forks/> [Accessed 14 December 2020].

OpenLedger DEX. 2020. *How Forks Impact The Price Of Cryptocurrency*. [online] Available at: <https://dex.openledger.io/how-forks-impact-the-price-of-cryptocurrency/> [Accessed 14 December 2020].

Commodity.com. 2020. *What Are Forks And How Do They Impact The Price Of Cryptocurrency? - Commodity.Com*. [online] Available at: <https://commodity.com/cryptocurrency/what-are-forks/#Hard_Fork_or_Soft_Fork_8211_Remember_That_Your_Capital_is_at_Risk> [Accessed 14 December 2020].

reddit. 2020. *How Will AVG Gas Price Change Over Time?*. [online] Available at: <https://www.reddit.com/r/ethereum/comments/7lfmv5/how_will_avg_gas_price_change_over_time/> [Accessed 14 December 2020].