Part A. Time Analysis

Calculate the number of transactions occurring every month between the start and end of the dataset

Python file: PartA_1.py

```
from mrjob.job import MRJob
import re
import time
import datetime

#This line declares the class Lab1, that extends the MRJob format.
class PartA(MRJob):
    def mapper(self, _, line):
        try:
            fields = line.split(",")
            if (len(fields)==7):
                time_epoch = int(fields[6])
                #returns month in appropriate format
                month = time.strftime("%m",time.gmtime(time_epoch))
                #returns year in appropriate format
                year = time.strftime("%Y",time.gmtime(time_epoch))
                yearMonth = (year,month)
                yield (yearMonth, 1)
        except:
            pass
    def combiner(self, month, count):
        yield (month, sum(count))

    def reducer(self, month, count):
        yield (month, sum(count))

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

Explanation

- In mapper the lines are split by comma: **fields = line.split(",")**.
- Filter out any bad lines while reading the transactions file: **if len(fields)==7:**
- block_timestamp field is defined as time_epoch, which is used to process the month and year values.
- The Unix timestamp is converted to appropriate format using time function.
- Map reduce job with key monthYear (month and year) and value 1 from transactions dataset is used to aggregate together all the transactions in the same month, including year.
- A combiner is used to calculate the sum of transaction counts for each month present in each mapper.
- Combiner improves the aggregation performance of the job.
- Reducer is the same as combiner.
- The output shows the total tansaction count per month for each year.

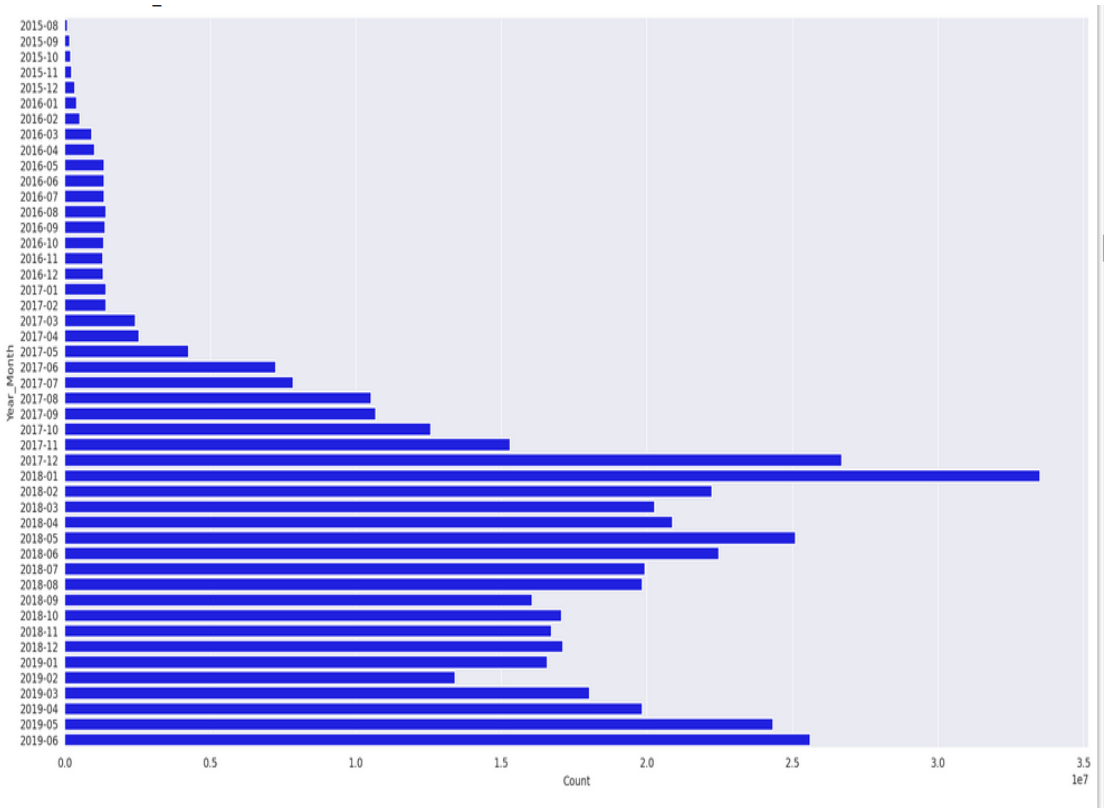
Job ID:

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1607539937312_0326/

Bar plot graph:

The output data from Part_A file is used to plot the below graph. The graph shows the changes in the total tansaction count per month.

- From Aug 2015 to Jan 2016 the transaction count is very low.
- From Feb 2016 to Feb 2017 there is a small increase in transactions.
- From Mar 2017 to Jan 2018 there is a major increase in the volume of transactions with Jan 2018 being the top increase with 3,35,04,270 transactions.
- From Jan 2018 to June 2019 there is a fluctuation in the number of transactions.



Calculate the average value of transaction in each month between the start and end of the dataset

Python file: PartA_2.py

```
from mrjob.job import MRJob
import re
import time
import datetime

class PartA_2(MRJob):
    def mapper(self, _, line):
        try:
            fields = line.split(",")
            if (len(fields)==7):
                time_epoch = int(fields[6])
                #returns month in appropriate format
                month = time.strftime("%m",time.gmtime(time_epoch))
                #returns year in appropriate format
                year = time.strftime("%Y",time.gmtime(time_epoch))
                yearMonth = (year,month)
                value = int(fields[3])
                yield (yearMonth, (value, 1))
        except:
            pass

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

    def reducer(self, feature, values):
        count = 0
        total = 0
        for value in values:
            count += value[1]
            total += value[0]
        #Average value per month
        yield (feature, total/count)

if __name__ == '__main__':
    PartA_2.run()
```

Explanation

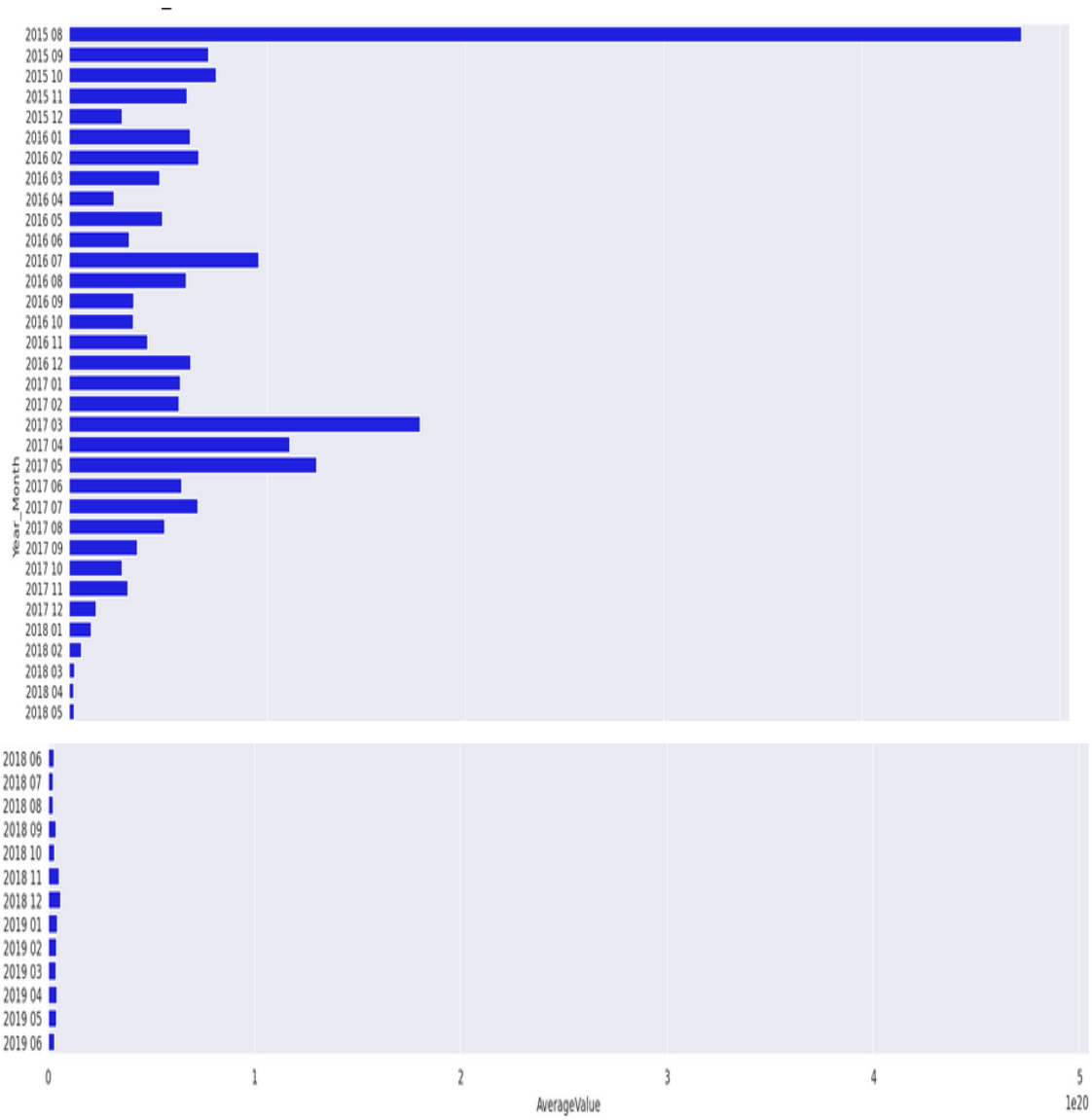
- In mapper the lines are split by comma: **fields = line.split(",")**.
- Filter out any bad lines while reading the transactions file: **if len(fields)==7:**
- block_timestamp field is defined as time_epoch, which is used to process the month and year values.
- The Unix timestamp is converted to appropriate format using time function.
- The value field in transactions dataset is defined as value.
- The key in mapper is month and year and values are value and count 1 to count the number of occurrences.
- The key in combiner is the formatted timestamp (year and month) and values are value field and count. A for loop is used to calculate the sum of value and its number of occurrences per month.

- In reducer the same for loop is implemented and the yielded values are formatted timestamp and the average of value of transactions per month. The average is calculated by dividing the sum of value with its total number of occurrences.
- The result is the average of value per month for each year.

Job ID:
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1604349877093_2898/

Bar plot graph:

Below graph shows the average value of transactions over time:



- At Aug 2015 the average value of transactions is very high.
- From Sep 2015 to Feb 2017 the average value is going up and down.

- From Mar 2017 to June 2019, there is a major decrease in the average value of transactions.

Part B. Top Ten Most Popular Services

Job 1 - Initial Aggregation

Python file: PartB_1.py

```
from mrjob.job import MRJob
import re

#aggregate transactions
class partB1(MRJob):

    def mapper(self, _, line):
        try:
            fields = line.split(",")
            if (len(fields)==7):
                toAddress = fields[2]
                value = int(fields[3])
                #yield only non-zero values
                if value == 0:
                    pass
                else:
                    yield(toAddress,value)
        except:
            pass

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

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

if __name__ == '__main__':
    partB1.run()
```

Explanation:

- In mapper the lines are split by comma: **fields = line.split(",")**.
- Filter out any bad lines while reading the transactions file: **if len(fields)==7:**
- If statement to filter zero values in the values field and yield only key and values that are not zero. By this way we save memory and improve the execution performance of the map-reduce job.
- A combiner is used to calculate the sum of values for each to_address from each mapper and improve the performance of the map-reduce job.
- Reducer is the does the same with combiner.
- The result shows us how much each address within the user space has been involved in.

Job ID:

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1604349877093_4820/

Job 2 - Joining transactions/contracts and filtering

Python file: PartB_2.py

```
from mrjob.job import MRJob

class partB2(MRJob):

    def mapper(self, _, line):
        try:
            if len(line.split('\t'))==2:#if it reads the output of job1
                fields = line.split('\t')#lines are splitted by tab
                toAddress = fields[0]
                toAddress = toAddress[1:-1]#remove double quotes ("")
                tValue = int(fields[1])
                yield (toAddress,(tValue,1))#identifier is 1

            elif len(line.split(','))==5:#if it reads from the contracts dataset
                fields = line.split(',')#lines are splitted by comma
                address = fields[0]
                cValue = int(fields[3])
                yield (address,(cValue,2))#identifier is 2
        except:
            pass

    def reducer(self, addresses, values):
        blockNumber = 0
        counts = 0
        for i in values:
            if i[1]==1:#if it reads from the output of job1
                counts = i[0]#gets the value
            elif i[1]==2:#if it reads from the contracts dataset
                blockNumber = i[0]#gets the value

        if blockNumber > 0 and counts > 0:#valid contract should be bigger than 0
            yield(addresses,counts)

if __name__ == '__main__':
    partB2.run()
```

Explanation:

- This map-reduce job performs repartition join between contracts dataset and the merged output file from job 1.
- The two input files are differentiated by checking the number of fields of each file. 2 fields for job1 output file and 5 fields for contracts dataset. Also, the output of job1 is tab separated and the contracts dataset is comma separated.
- In case that the mapper identifies the input as the output of job1 we specify key as toAddress and value as sum of values (in Wei).
- We filter smart contract address by ensuring that reducer takes records only if both keys match (address from both dataset match).
- A for loop is used in the reducer to identify and aggregate the values of each dataset

- Finally, the reducer yields smart contract address as key and aggregated value as value. So, the output of the reducer is the the aggregated values of all smart contracts.

Job ID:

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1604349877093_4607/

This is a sample of the output of Job 2:

```
"0x0000000000b3f879cb30fe243b4dfee438691c04"    21000000000000005
"0x0000000000c90bc353314b6911180ed7e06019a9"    165090823807118895905
"0x000000000a3c7c86345a35a0e97a4bb4370a8dd9"    65100000000000000000
"0x000000001e29fcd9b1469a7954dc65ff254fffc0"    70100000000000000000
"0x000000002bb43c83ece652d161ad0fa862129a2c"    19000000000000000
"0x00000000a857d2cb72574491795fbed7ceac3095"    100000000000000000000
"0x00000000aca7ba47149da8a2f0ce02d140ecfa12"    65100000000000000000
"0x0000010d23ccfee520c3fb5a5ba9679cb9d83cbe"    32100000000000000
"0x00008055826f5781cfe162b27e44521749a88f35"    300292030000000000
"0x000081b3b28e1f0f2b1011037cc32667feb61c58"    48600000000000000
"0x00009ccbed893482507888d4b357c13bb8aac4d0"    295167891800000000000
"0x0000b3c60adf7cce979886e09ad7c5c063e8364e"    56089659915611634 82
"0x0000ec7af931889dff9a7914757b8ee36b534c81"    53966541100000000000
"0x0000f450492c31a4f6512a197c183c046e2adc86"    900000000000000000
"0x0000fc09a954ae4c66da7ae21bd4ed0389cbeebf"    2940000000000000000
"0x00017f8810b9996099457594b6b707c0f84617f2"    133360000000000000
"0x000180e8ca31a690ba5b772667938ea9da12d492"    635420000000000000
"0x0001848946cb542fc4ef7e80eef0448672bf8e36"    1538699230000000000
"0x00019ad131efb14b3431d9f4f68db5f5922bf1b9"    28941202800000000000
"0x0001a3c6be5a99c2fd89eb00199cf426d20d5acd"    542000000000000000
"0x0001ab13f42c1f1be321ec92abdea822f6673090"    19498195700000000000
"0x0001adf843ffff7fdb5c6791df5bc56cf0ca574f"    245451198200000000000
"0x0001b871ad660424120a9df84529039104248a57"    95623320000000000
"0x0001fa39b9e6e096ca71c23c9e6c17f873207198"    50107815919616310
```

Job 3 - Top Ten

```
from mrjob.job import MRJob

class partB3(MRJob):

    def mapper(self, _, line):
        try:
            if len(line.split('\t'))==2:#mapper reads the output from job2
                fields = line.split('\t')
                address = fields[0]
                address = address[1:-1]#remove double quotes("")
                count = int(fields[1])
                yield (None,(address,count))
        except:
            pass
    #sort the top 10 values
    def combiner(self,_, values):
        sortedValues = sorted(values, reverse = True, key = lambda tup:tup[1])
        counter = 0
        for value in sortedValues:
            yield("top", value)
            counter += 1
            if counter >= 10:
                break
    #sort the top 10 values and yield them
    def reducer(self,_, values):
        sortedValues = sorted(values, reverse = True, key = lambda tup:tup[1])
        counter = 0
        for value in sortedValues:
            yield(counter, ("{} - {}".format(value[0], value[1])))
            counter += 1
            if counter >= 10:
                break

if __name__ == '__main__':
    partB3.run()
```

Explanation:

- Map reduce job to filter top 10 smart contracts from Job 2 output.
- The number of fields of the file should be 2. The lines are splitted by tab, because the input file is tab separated.
- Yield None as key and the address and sum of aggregated counts as value. Key is None because we need to sort the values based on the aggregated count.
- The combiner is used to sort the values in descending order by specifying reverse = True. This will give us the top values first. A for loop is used to iterate through the sorted values and yield the first top ten in descending order.
- The reducer performs the same operation with the combiner. Finally, the for loop is terminated if the iteration count reaches 10 and the reducer shows the top 10 values.

Job ID:

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1604349877093_6317/

Top ten output (Job 3):

Below image shows the top ten smart contracts:

```
0       "0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444 - 841551008099658658227267776"
1       "0xfa52274dd61e1643d2205169732f29114bc240b3 - 45787484483189352986478805"
2       "0x7727e5113d1d161373623e5f49fd568b4f543a9e - 45620624001350712557268573"
3       "0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef - 43170356092262468919298969"
4       "0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8 - 27068921582019542499882877"
5       "0xbfc39b6f805a9e40e77291aff27aee3c96915bdd - 21104195138093660050000000"
6       "0xe94b04a0fed112f3664e45adb2b8915693dd5ff3 - 15562398956802112254719409"
7       "0xbb9bc244d798123fde783fcc1c72d3bb8c189413 - 11983608729202893846818681"
8       "0xabbb6bebfa05aa13e908eaa492bd7a8343760477 - 11706457177940895521770404"
9       "0x341e790174e3a4d35b65fdc067b6b5634a61caea - 837900075191775562405750 0"
```

PART C. TOP TEN MOST ACTIVE MINERS

Evaluate the top 10 miners by the size of the blocks mined

I have splitted this part into two different jobs, for better management.

Job 1

Python file: PartC_1.py

```
from mrjob.job import MRJob

#aggregate blocks
class partC1(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
#aggregate size in the miner field
    def combiner(self,miner,value):
        yield (miner, sum(value))
#aggregate size in the miner field
    def reducer(self,miner,value):
        yield (miner, sum(value))

if __name__ == '__main__':
    partC1.run()
```

Explanation:

- In mapper the lines are split by comma: **fields = line.split(‘,’)**.
- Filter out any bad lines while reading the block dataset: **if len(fields) ==9:**

- Map reduce job with key miner and value size from blocks dataset is used to aggregate together all the transactions in the same month, including year.
- A combiner is used to calculate the sum of size values for each miner present in each mapper.
- Combiner improves the aggregation performance of the job.
- Reducer is the same as combiner.
- The output shows the total size value per miner.

Job ID:
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1604349877093_6293/

Job 2

Python file: PartC_2.py

```
from mrjob.job import MRJob
#aggregate top 10 values
class partC2(MRJob):

    def mapper(self, _, line):
        try:
            if len(line.split('\t'))==2:#read job1 output file
                fields = line.split('\t')
                miner = fields[0]
                miner = miner[1:-1]#remove double quotes
                size = int(fields[1])
                yield (None,(miner,size))
        except:
            pass
#sort values and show top ten most active miners
    def reducer(self,_, values):
        sortedValues = sorted(values, reverse = True, key = lambda tup:tup[1])#sort values in descending order
        counter = 0
        for value in sortedValues:
            yield(counter, ("{} - {}".format(value[0], value[1])))
            counter += 1
            if counter >= 10:
                break

if __name__ == '__main__':
    partC2.run()
```

Explanation:

- Map reduce job to filter top 10 most active miners from Job 1 output.
- The number of fields of the file should be 2. The lines are splitted by tab, because the input file is tab separated.
- Yield None as key and address and sum of aggregated counts as value. Key is None because we need to sort the values based on the aggregated count.
- The combiner is used to sort the values in descending order by specifying reverse = True. This will give us the top values first. A for loop is used to iterate through the sorted values and yield the first top ten in descending order.

- The reducer performs the same operation with the combiner. Finally, the for loop is terminated if the iteration count reaches 10 and the reducer shows the top 10 values.

Job ID:

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1604349877093_6302/

Top ten most active miners output (Job 2):

```
0       "0xea674fdde714fd979de3edf0f56aa9716b898ec8 - 23989401188"
1       "0x829bd824b016326a401d083b33d092293333a830 - 15010222714"
2       "0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c - 13978859941"
3       "0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5 - 10998145387"
4       "0xb2930b35844a230f00e51431acae96fe543a0347 - 7842595276"
5       "0x2a65aca4d5fc5b5c859090a6c34d164135398226 - 3628875680"
6       "0x4bb96091ee9d802ed039c4d1a5f6216f90f81b01 - 1221833144"
7       "0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb - 1152472379"
8       "0x1e9939daaad6924ad004c2560e90804164900341 - 1080301927"
9       "0x61c808d82a3ac53231750dadc13c777b59310bd9 - 692942577"
```

Part D

Scam Analysis

Convert scams.json to csv

Firstly, I converted scams.json to csv format in order to access the address, category and status field. The scams.csv dataset is uploaded again on HDFS and stored in 'input' directory:

Python file: JsonToCSV.py

```
from mrjob.job import MRJob
import json
#convert json to csv
class ScamToCSV(MRJob):

    def mapper(self, _, line):
        fields = line.split("},")
        count=0
        for i in range(len(fields)):
            scam = fields[i].split(",")
            if count is 0:
                address = scam[1][11:54][0:-1]
                category = scam[5][12:-1]
                status = scam[10][10:-1]
                print("{},{},{}".format(address,category,status))
                count+=1
            else:
                address2 = scam[0][1:].split("\"", 1)[0]
                for i in range(len(scam)):
                    if scam[i].startswith("\"category\""):
                        category2=scam[i][12:-1]
                    elif scam[i].startswith("\"status\""):
                        status2= scam[i].split("\"", 3)[3].split("\"", 1)[0]
                print("{},{},{}".format(address2,category2,status2))
if __name__ == '__main__':
    ScamToCSV.run()
```

Find the most lucrative scam and how does it change throughout time

Python file: scamAnalysis.py

```
import pyspark
import time

sc = pyspark.SparkContext()
#function to filter out bad lines from transactions dataset
def good_lines_transactions(line):
    try:
        fields = line.split(',')
        if len(fields)!=7:
            return False
        int(fields[6])
        int(fields[3])
        return True
    except:
        return False
#read transactions dataset
transactions = sc.textFile('/data/ethereum/transactions')
#use good_lines_transactions function to filter out bad lines
transactions_good = transactions.filter(good_lines_transactions)
#map function to map to_address field as key and block_timestamp field and value as values from transactions
rdd1_transactions = transactions_good.map(lambda t: (t.split(',')[2], (int(t.split(',')[6]), int(t.split(',')[3]))))
#read scams dataset
scams = sc.textFile('/user/xg001/input/convert.csv')
#map function to map the address and category fields from scams dataset
rdd2_scams = scams.map(lambda s: (s.split(',')[0], s.split(',')[1]))
#join operation for transactions and scams datasets
joined_datasets = rdd1_transactions.join(rdd2_scams)
#map category as key and value as values
lucrativeScams = joined_datasets.map(lambda a: (a[1][1], a[1][0][1]))
#reduceByKey function to aggregate value
sum_lucrativeScams = lucrativeScams.reduceByKey(lambda a,b: (a+b)).sortByKey()
#save the result with category as key and aggregated value as values
sum_lucrativeScams.saveAsTextFile('lucrativeScams')
#map category as key and block_timestamp and value as values
timeseries = joined_datasets.map(lambda b: ((b[1][1], time.strftime("%m-%Y",time.gmtime(b[1][0][0]))), b[1][0][1]))
#reduceByKey to aggregate values
sum_timeseries = timeseries.reduceByKey(lambda a,b: (a+b)).sortByKey()
#save the result with category, and timestamp along with the aggregated values.
sum_timeseries.saveAsTextFile('scamTimeseries')
```

Explanation:

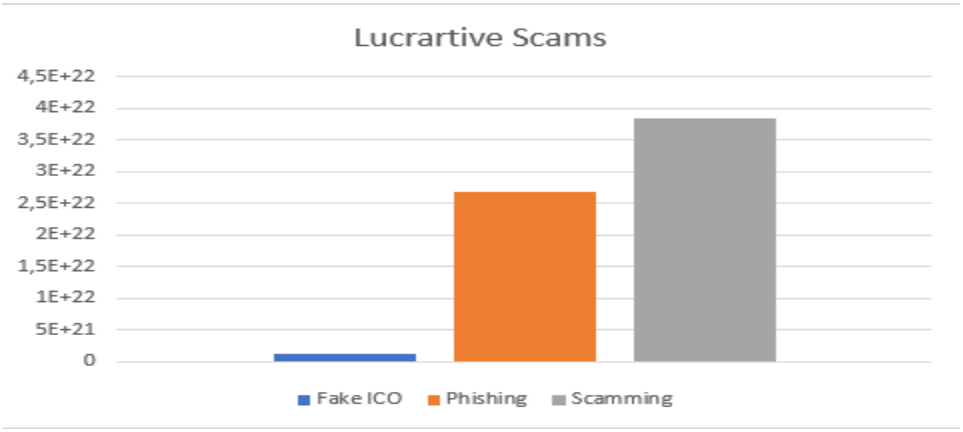
- Find the most lucrative form of scam and how they change throughout time by running a Spark Job.
- Write good_transactions function to filter any malformed lines from transactions dataset and convert values and block_timestamp to integers.
- Read transactions dataset and filter any bad lines using good_transactions function. Map the to_address as key and values and block_timestamp as values using lambda function.
- After that, read the converted scams.csv dataset. Use lambda function to map address as key and category as values.
- Perform join operation for the transactions and scams datasets. From the joined datasets, we map category as key and value as values.

- Perform reducebyKey operation and use lambda function to aggregate the values. The output of this operation is the scam categories along with the sum of values and it is stored in HDFS as lucrativeScams.
- Then, in order to find the timeseries for each scam, from the joined dataset we map the scam category as key and timestamp and value as values. The timestamp is converted to the appropriate format.
- Use reduceByKey function to aggregate the values.
- The output of the function is the scam category, the year and month of the scam and the sum of values for that month. This output is stored to HDFS as timeseries.

Output of lucrativeScams:

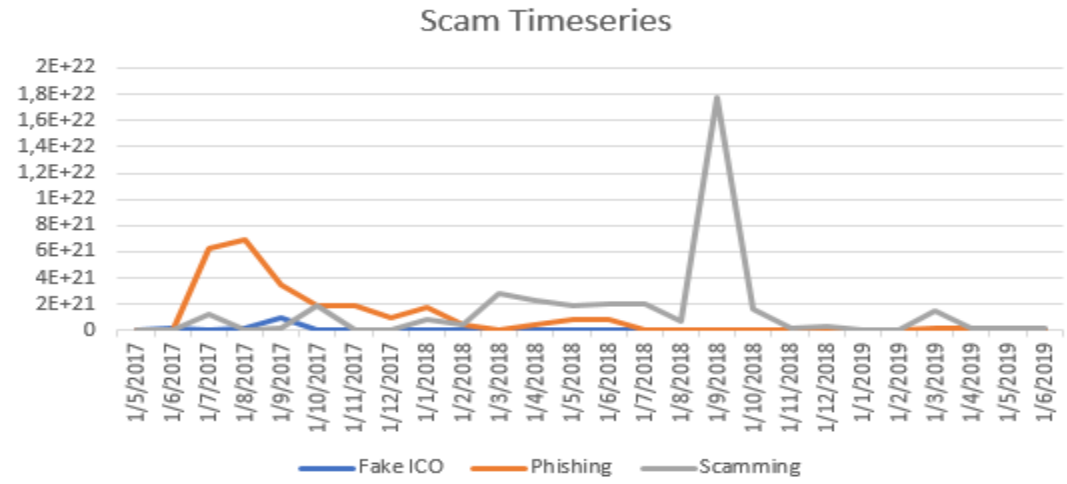
The values show how much value was transferred for each form of scam. Based on the output values, the most lucrative form of scam is 'Scamming'.

```
(u'Fake ICO', 135645756688962997967 8L)
(u'Phishing', 26927757396110618476458L)
(u'Scamming', 38407781260421703730344L)
```



Timeseries analysis of scams:

The below graph is plotted based on the output of 'scamTimeseries' file.



The above graph shows that the peak of the most lucrative scam 'Scamming' was on September of 2018.

Correlation with a known scam:

According to the chief economist at Chainalysis, Philip Gradwell, the classic ponzi scheme was one of the most important scams of 2018. Specifically, some scammers sent emails to Ether owners asking them to invest money and guaranteed to them that they would take back a good profit. As it might be expected, the return was merely derived from other individuals who contributed to the pot, not from any actual investment (Brewster, 2019).

Miscellaneous Analysis

Comparative evaluation

Reimplement Part B using Spark

Python file: compEval.py

```
import pyspark

sc = pyspark.SparkContext()
#function to filter out bad lines from transactions dataset
def good_lines_transactions(line):
    try:
        fields = line.split(',')
        if len(fields)!=7:
            return False
        int(fields[3])
        return True
    except:
        return False
#function to filter out bad lines from contracts dataset
def good_lines_contracts(line):
    try:
        fields = line.split(',')
        if len(fields)!=5:
            return False
        return True
    except:
        return False
#read transactions dataset
transactions = sc.textFile('/data/ethereum/transactions')
#use good_lines_transactions function to filter out bad lines
transactions_good = transactions.filter(good_lines_transactions)
#map function to map to_address field as key and value as values from transactions
rdd1_transactions = transactions_good.map(lambda t: (t.split(',')[2], int(t.split(',')[3])))
#reduceByKey function to aggregate value
outJob1 = rdd1_transactions.reduceByKey(lambda a,b: (a+b)).sortByKey()
```

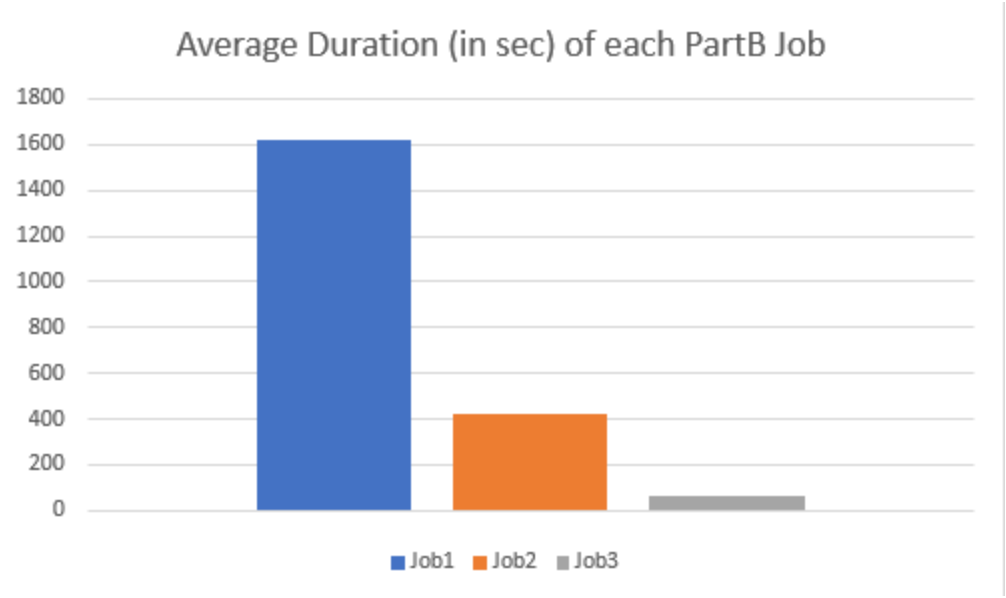
```
#read contracts dataset
contracts = sc.textFile('/data/ethereum/contracts')
#use good_lines_contracts function to filter out bad lines
contracts_good = contracts.filter(good_lines_contracts)
#map function to map address field as key and block_number as values from contracts
rdd2_contracts = contracts_good.map(lambda c: (c.split(',')[0], c.split(',')[3]))
#join operation for transactions and contracts datasets
joined_datasets = outJob1.join(rdd2_contracts)
#filter top 10 smart contracts from the joined_datasets
topTen = joined_datasets.takeOrdered(10, key = lambda x:-x[1][0])

counter = 0

for value in topTen:

    counter += 1

    print(counter, "{} {}".format(value[0], value[1][0]))
```

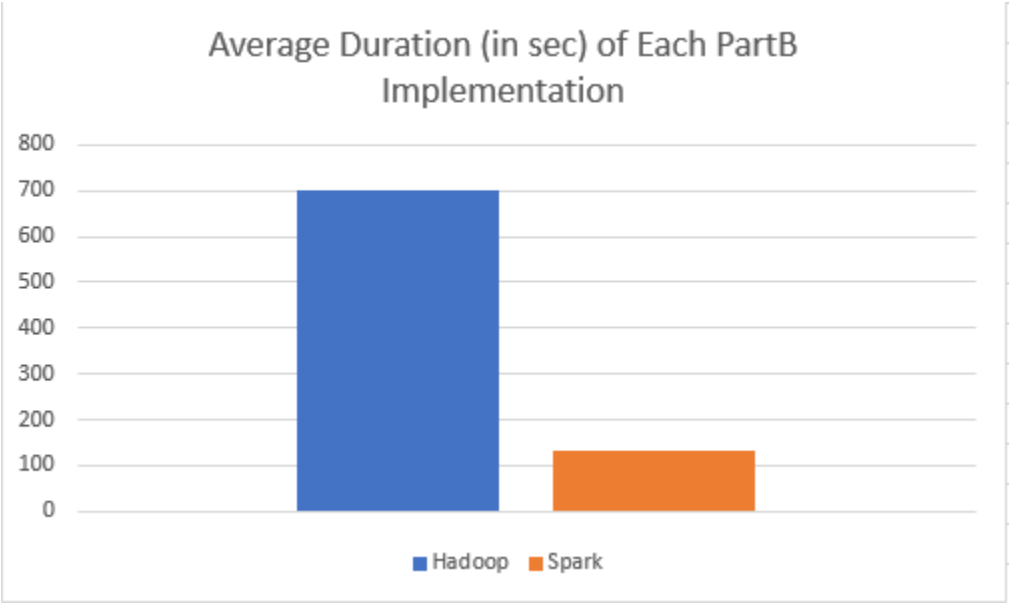
Explanation:

- Reimplement Part B (find top 10 smart contracts) running a Spark Job.
- Write good_transactions function to filter any malformed lines from transactions dataset and convert values field to integer.
- Write good_contracts function to filter any malformed lines from contracts dataset.
- Read transactions dataset and use map function. Using lambda function, map the to_address field as key and value field as values.
- Use reduceByKey lambda function to aggregate transaction values.
- Read contracts dataset and filter out any bad lines using good_contracts function. Using lambda function, map address field as key and block number as values.
- Perform join operation between outJob1 and rdd2_contracts.
- Use takeOrdered function to sort and filter only the top 10 values. Use lambda function 'x:-x[1][0]' to sort the values in descending order.
- Print the output (top 10 smart contracts) at the console.

Time Comparison:

In order to evaluate the performance between Hadoop and Spark implementation of Part B, I had to run each Hadoop job and Spark job twice and report the average duration time. For the first graph I found the average duration time between the first and the second run of each Hadoop job. For the second graph, I found the average duration time of Hadoop implementation (Job1, Job2, Job3) versus Spark implementation.





Conclusion:

- As it is shown in the graphs, Spark performs much faster than Hadoop for the implementation of Part B.
- Spark supports in-memory processing, while Hadoop MapReduce has to read from and write to a disk.
- In an in-memory processing approach data is loaded in memory before computation and kept in memory during successive steps.

- If the task is iterative like PartB, Spark is better than Hadoop. Spark's RDDs enable multiple map operations in memory, whereas Hadoop has to write interim results to the disk.
- Although, it is possible to implement multiple map and reduce steps in one Hadoop job, but the amount of shuffle and sort that occurs throughout join operation is very large that would reduce the performance of the job.
- These are the reasons why a Spark Job is preferred for the implementation of Part B.

Hadoop Job IDs:

Job1(1):
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1606730688641_6281/

Job1(2):
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1606730688641_6331/

Job2(1):
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1606730688641_6419/

Job2(2):
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1606730688641_6429/

Job3(1):
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1606730688641_7101/

Job3(2):
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1606730688641_7110/

Spark Job IDs:

*None

Gas Guzzlers

Calculate how the gas price has changed over time

Python file: GasPrice.py

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

class GasPrice(MRJob):
    #mapper yields formatted block_timestamp as key and gasPrice field and 1 as values
    def mapper(self, _, line):
        try:
            fields = line.split(",")
            if (len(fields)==7): #read from transactions dataset
                gasPrice = int(fields[5])
                time_epoch = int(fields[6])
                month = time.strftime("%m",time.gmtime(time_epoch))
                year = time.strftime("%Y",time.gmtime(time_epoch))
                yearMonth = (year,month)
                yield (yearMonth, (gasPrice, 1))
        except:
            pass
    #combiner yields formatted block_timestamp as key and total gasPrice and counter as values
    def combiner(self, feature, values):
        count = 0
        total = 0
        for value in values:
            count += value[1]
            total += value[0]
        yield (feature, (total,count))
    #reducer yields formatted block_timestamp as key and average gasPrice as values
    def reducer(self, feature, values):
        count = 0
        total = 0
        for value in values:
            count += value[1]
            total += value[0]
        yield (feature, total/count)

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

Explanation:

- Map-reduce job to calculate the change of gas_price over the years. In order to perform a timeseries analysis, we must find the average value of gas_price per month each year.
- In mapper we convert the block_timestamp field to the appropriate format and gas_price to integer. Then, we yield converted block_timestamp as key and gas_price and 1 as values. Count 1 is yielded to count the number of occurrences that will be used in reducer to calculate the average.
- In combiner, a for loop is used to aggregate the sum of gas_price values and the number of occurrences per month of each year and yield the formatted block_timestamp as key and total gas price and count as values.
- The reducer does the same with combiner, but it yields the formatted block_timestamp as key and the average gas_price by calculated by dividing the total gas_price with the total number of occurrences. The output is the average of gas price per month of each year.

Job ID:

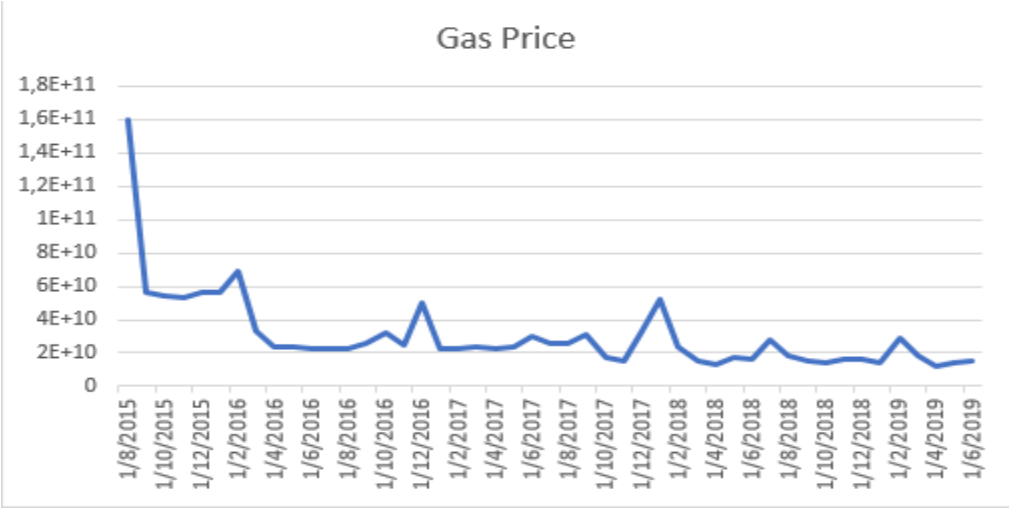
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1607539937312_0507/

Timeseries analysis of gas price:

Excel does not recognize float values, so I converted GasPrice column to integer using pandas.

```
import pandas as pd

df = pd.read_csv('GasPriceAvg.txt')

df['GasPrice'] = df['GasPrice'].astype(int)
df.to_csv('newGasPriceAvg.txt')
```

The below graph depicts the change of gas price over time.



Calculate the complexity of the contracts over time

Job 1

File: TopGasJoin.py

```
import pyspark

sc = pyspark.SparkContext()
#function to filter out bad lines from transactions dataset
def good_lines_transactions(line):
    try:
        fields = line.split(',')
        if len(fields)!=7:
            return False
        int(fields[4])
        int(fields[6])
        return True
    except:
        return False
#function to filter out bad lines from top 10 dataset
def good_lines_top10(line):
    try:
        fields = line.split(',')
        if len(fields)!=2:
            return False
        int(fields[1])
        return True
    except:
        return False
#read transactions dataset
transactions = sc.textFile('/data/ethereum/transactions')
#use good_lines_transactions function to filter out bad lines
transactions_good = transactions.filter(good_lines_transactions)
#map function to map to_address field as key and gas and block_timestamp fields as values from transactions
rdd1_transactions = transactions_good.map(lambda t: (t.split(',')[2], (int(t.split(',')[4]), int(t.split(',')[6]))))
#read top 10 smart contracts dataset
top10 = sc.textFile('/user/xg001/input/smart.txt')
#use good_lines_top10 function to filter out bad lines
top10_good = top10.filter(good_lines_top10)
#map function to map address field as key and value as values from top 10
rdd2_top10 = top10_good.map(lambda s: (s.split(',')[0], int(s.split(',')[1])))
#join operation for transactions and top 10 smart contracts datasets
joined_datasets = rdd1_transactions.join(rdd2_top10)
#store output at HDFS
joined_datasets.saveAsTextFile("topGasJob1")
```

Explanation:

- Perform join operation on top 10 smart contracts dataset (from PartB) and transactions dataset, using Spark.
- Write good_transactions function to filter any malformed lines from transactions dataset and convert values field to integer.
- Write good_contracts function to filter any malformed lines from contracts dataset.
- Read transactions dataset and filter any bad lines using good_transactions function
- Perform map operation to transactions dataset using lambda function. Map to_address filed as key and gas and block_timestamp field as values.
- Read top10 smart contracts dataset and filter any bad lines using good_top10 function
- Perform map operation to top 10 dataset using lambda function. Map address field as key and aggregated value as values
- Then, perform join operation and join the two mapped datasets. Address is the key and gas, block_timestamp and aggregated value are the values.
- The result is stored in the HDFS.

Join output:

Below there is a sample of the join output.

```
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', ((135546, 1511941959), 15562398956802112254719409L))
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', ((135546, 1511941959), 15562398956802112254719409L))
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', ((135546, 1511943737), 15562398956802112254719409L))
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', ((135546, 1511943737), 15562398956802112254719409L))
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', ((135546, 1511943737), 15562398956802112254719409L))
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', ((135546, 1511941996), 15562398956802112254719409L))
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', ((135546, 1511941996), 15562398956802112254719409L))
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', ((135546, 1511941996), 15562398956802112254719409L))
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', ((135546, 1511943918), 15562398956802112254719409L))
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', ((135546, 1511943918), 15562398956802112254719409L))
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', ((135546, 1511943918), 15562398956802112254719409L))
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', ((135546, 1511943918), 15562398956802112254719409L))
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', ((135546, 1511943918), 15562398956802112254719409L))
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', ((135546, 1511943918), 15562398956802112254719409L))
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', ((135546, 1511943918), 15562398956802112254719409L))
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', ((135546, 1511943918), 15562398956802112254719409L))
```

Job 2

File: TopGasJob2.py

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

class GasTop(MRJob):
    #mapper yields to_address, formatted block_timestamp as key and gas field and 1 as values
    def mapper(self, _, line):
        try:
            fields = line.split(",")
            to_address = fields[0]
            time_epoch = int(fields[2][:-1])#remove extra brackets
            gas = int(fields[1][3:])
            month = time.strftime("%m",time.gmtime(time_epoch))
            year = time.strftime("%Y",time.gmtime(time_epoch))
            yearMonth = (year,month)
            yield ((to_address, yearMonth), (gas, 1))
        except:
            pass
    #combiner yields to_address, formatted block_timestamp as key and total gas, counter as values
    def combiner(self, feature, values):
        count = 0
        total = 0
        for value in values:
            count += value[1]
            total += value[0]
        yield (feature, (total,count))
    #reducer yields to_address, formatted block_timestamp as key and average gas as values
    def reducer(self, feature, values):
        count = 0
        total = 0
        for value in values:
            count += value[1]
            total += value[0]
        yield (feature, total/count)

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

Explanation:

- Map-reduce job to calculate the average gas that each smart contract has consumed over time.
- In mapper, we read the output from previous job (gasJoin), convert the block_timestamp field to the appropriate format (year and month) and declare gas

field as integer. Then, yield to_address, block_timestamp as key and gas, count 1 as values. Count 1 is yielded to count the number of occurrences that will be used in reducer to calculate the average.

- In combiner, a for loop is used to aggregate the sum of gas values and the number of occurrences per month of each year and yield to_address, the formatted block_timestamp as key, and total gas, count as values.
- The reducer does the same with combiner, but it yields the to_address, formatted block_timestamp as key and the average gas by dividing the total gas with the total number of occurrences. The output is the average gas of each one of the top smart contracts over its time.

Job ID:

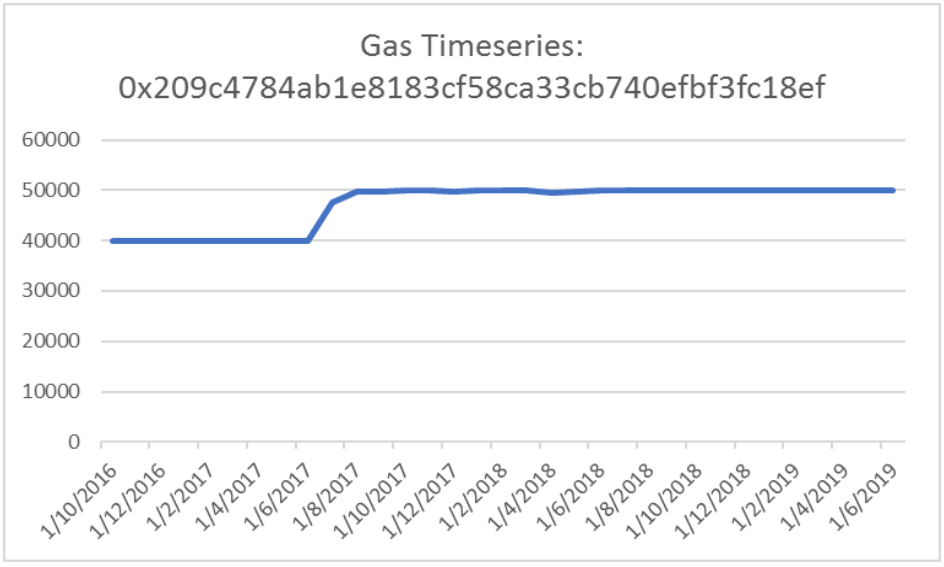
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1607539937312_2179/

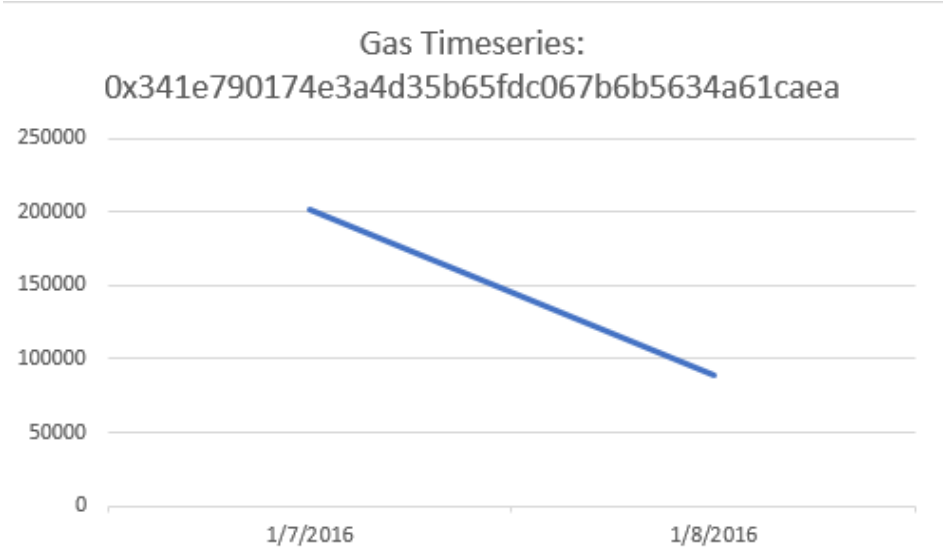
Timeseries analysis:

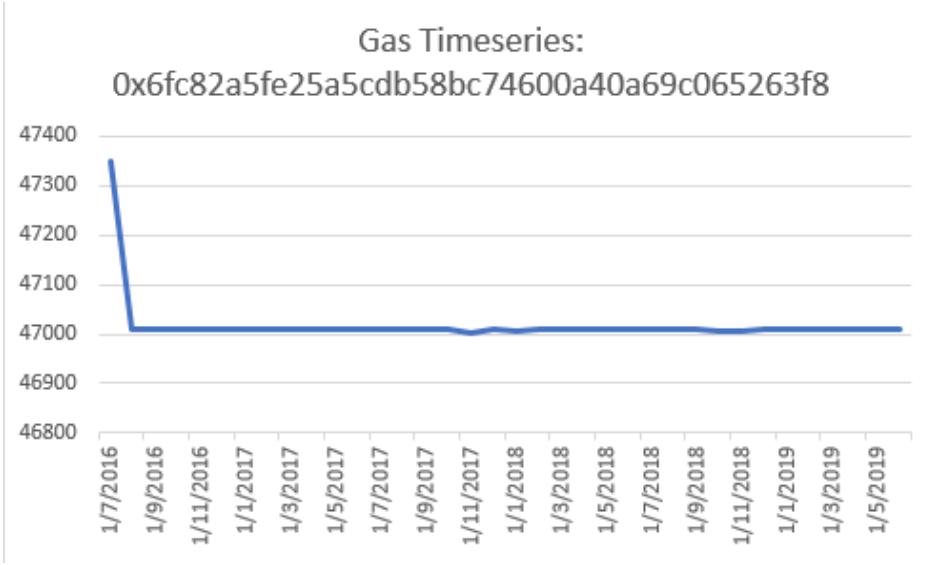
Excel does not recognize float values, so I converted gas column to integer using pandas.

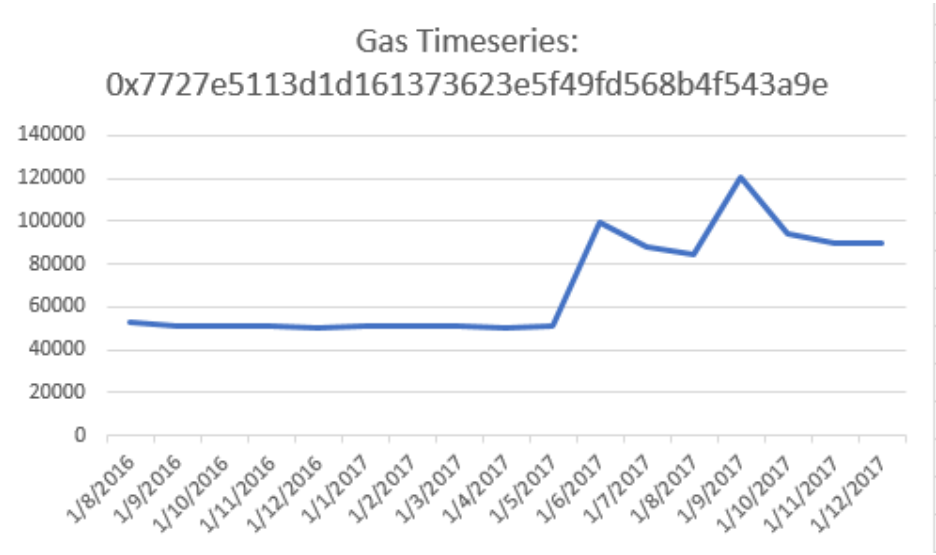
```
import pandas as pd

df = pd.read_csv('topGasOut.txt')

df['Gas'] = df['Gas'].astype(int)
df.to_csv('newTopGasOut.txt')
```

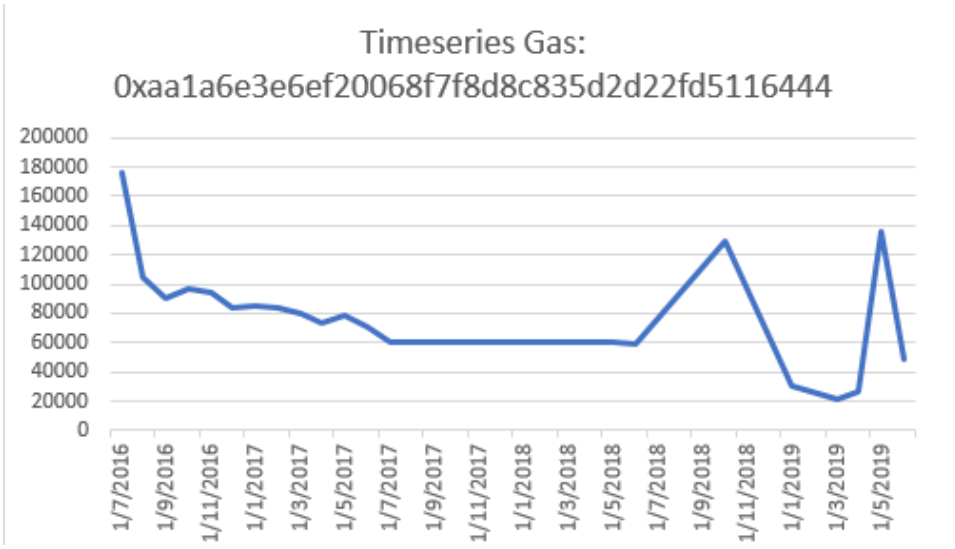
Below timeseries graphs present the average gas consumed by each one of the top 10 smart contracts over its time.

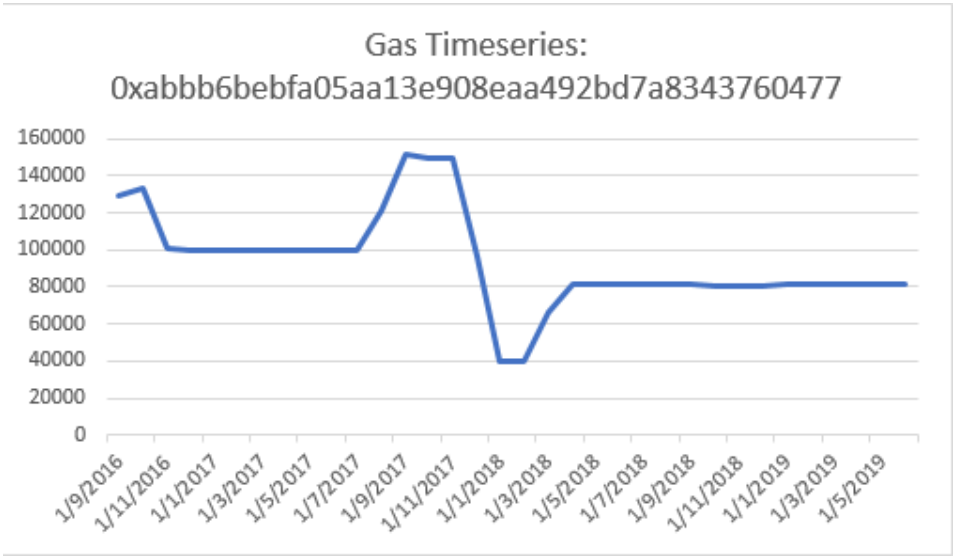


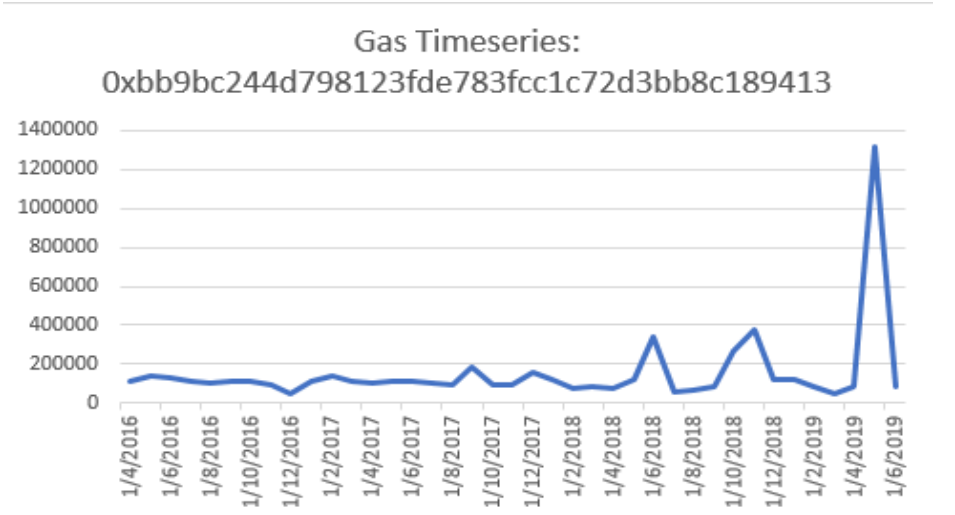

Gas Timeseries:
0x341e790174e3a4d35b65fdc067b6b5634a61caea
250000
200000
150000
100000
50000
0
1/7/2016
1/8/2016

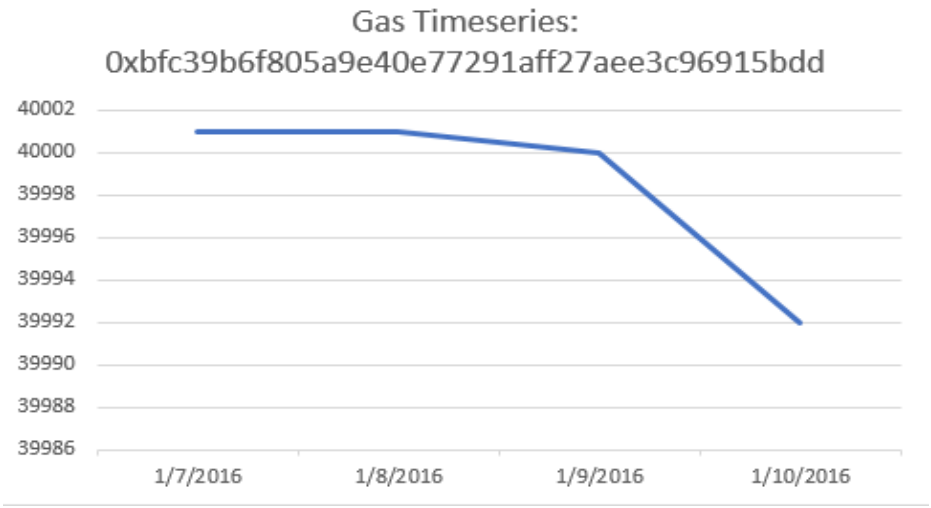

Gas Timeseries:
0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8
47400
47300
47200
47100
47000
46900
46800
1/7/2016
1/9/2016
1/11/2016
1/1/2017
1/3/2017
1/5/2017
1/7/2017
1/9/2017
1/11/2017
1/1/2018
1/3/2018
1/5/2018
1/7/2018
1/9/2018
1/11/2018
1/1/2019
1/3/2019
1/5/2019

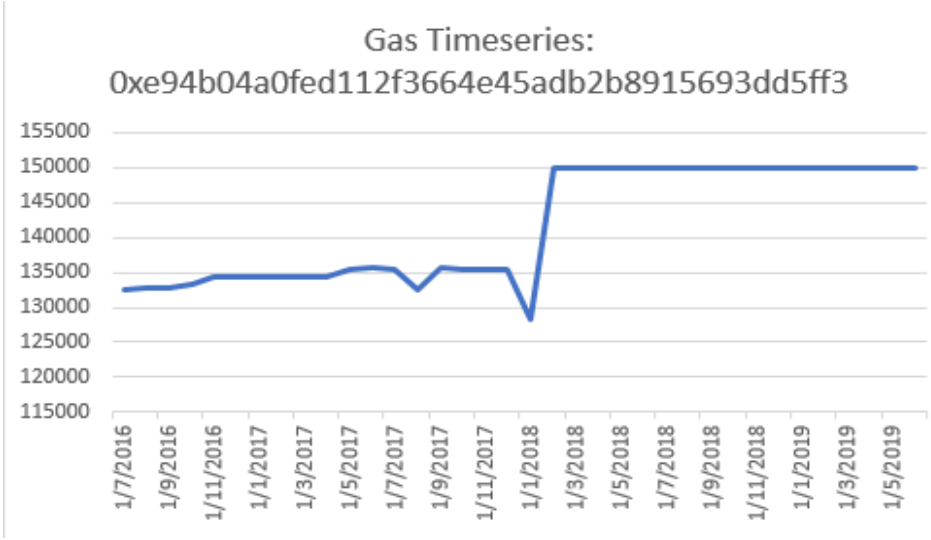

Gas Timeseries:
0x7727e5113d1d161373623e5f49fd568b4f543a9e
140000
120000
100000
80000
60000
40000
20000
0
1/8/2016
1/9/2016
1/10/2016
1/11/2016
1/12/2016
1/1/2017
1/2/2017
1/3/2017
1/4/2017
1/5/2017
1/6/2017
1/7/2017
1/8/2017
1/9/2017
1/10/2017
1/11/2017
1/12/2017

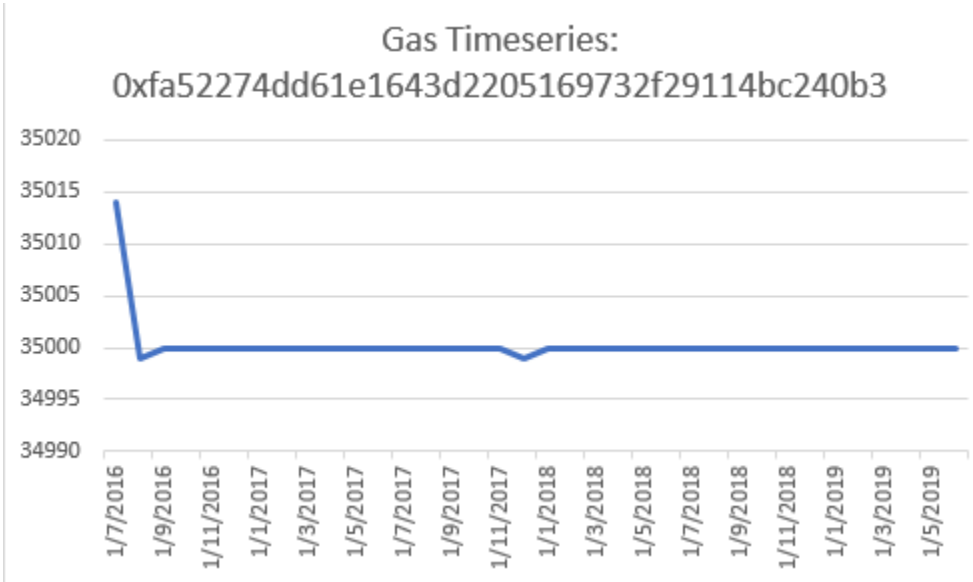

Timeseries Gas:
0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444
200000
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
1/7/2016
1/9/2016
1/11/2016
1/1/2017
1/3/2017
1/5/2017
1/7/2017
1/9/2017
1/11/2017
1/1/2018
1/3/2018
1/5/2018
1/7/2018
1/9/2018
1/11/2018
1/1/2019
1/3/2019
1/5/2019


Gas Timeseries:
0xabbb6bebfa05aa13e908eaa492bd7a8343760477
160000
140000
120000
100000
80000
60000
40000
20000
0
1/9/2016
1/11/2016
1/1/2017
1/3/2017
1/5/2017
1/7/2017
1/9/2017
1/11/2017
1/1/2018
1/3/2018
1/5/2018
1/7/2018
1/9/2018
1/11/2018
1/1/2019
1/3/2019
1/5/2019


Gas Timeseries:
0xbb9bc244d798123fde783fcc1c72d3bb8c189413
1400000
1200000
1000000
800000
600000
400000
200000
0
1/4/2016
1/6/2016
1/8/2016
1/10/2016
1/12/2016
1/2/2017
1/4/2017
1/6/2017
1/8/2017
1/10/2017
1/12/2017
1/2/2018
1/4/2018
1/6/2018
1/8/2018
1/10/2018
1/12/2018
1/2/2019
1/4/2019
1/6/2019


Gas Timeseries:
0xbfc39b6f805a9e40e77291aff27aee3c96915bdd
40002
40000
39998
39996
39994
39992
39990
39988
39986
1/7/2016
1/8/2016
1/9/2016
1/10/2016


Gas Timeseries:
0xe94b04a0fed112f3664e45adb2b8915693dd5ff3
155000
150000
145000
140000
135000
130000
125000
120000
115000
1/7/2016
1/9/2016
1/11/2016
1/1/2017
1/3/2017
1/5/2017
1/7/2017
1/9/2017
1/11/2017
1/1/2018
1/3/2018
1/5/2018
1/7/2018
1/9/2018
1/11/2018
1/1/2019
1/3/2019
1/5/2019


Gas Timeseries:
0xfa52274dd61e1643d2205169732f29114bc240b3
35020
35015
35010
35005
35000
34995
34990
1/7/2016
1/9/2016
1/11/2016
1/1/2017
1/3/2017
1/5/2017
1/7/2017
1/9/2017
1/11/2017
1/1/2018
1/3/2018
1/5/2018
1/7/2018
1/9/2018
1/11/2018
1/1/2019
1/3/2019
1/5/2019

Conclusion:

Looking at the above graphs, it is concluded that there is not a specific trend. Some smart contracts require more gas over time, making them more complicated, but some other require less, making them less complicated. There are also some smart contracts that have been going up and down over time.

Notes

*There were not Job IDs, when I was running Spark jobs.

References

Brewster, T., 2019. *Ether Cryptocurrency Scammers Made $36 Million In 2018 -- Double Their 2017 Winnings*. [online] Forbes. Available at: <https://www.forbes.com/sites/thomasbrewster/2019/01/23/ether-scammers-made-36-million-in-2018double-their-2017-winnings/?sh=7f2e83bc2c16> [Accessed 05 December 2020].