ECS765P - Big Data Processing
Coursework - Analysis of Ethereum Transactions and Smart Contracts
Pranjali Hande – 220707639

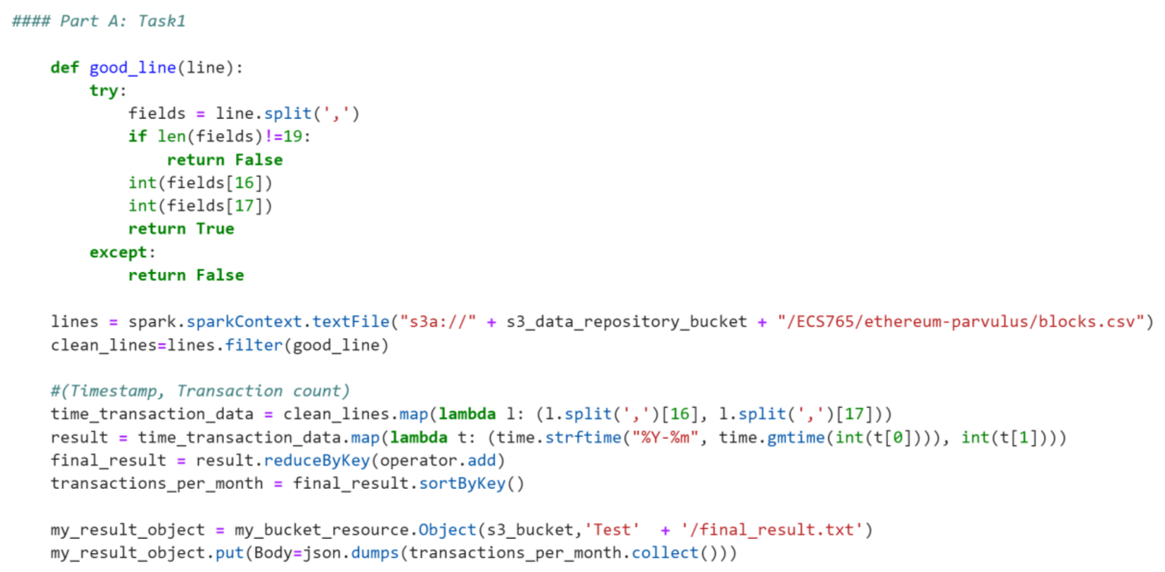
Part A. Time Analysis (25%)

Task 1: Create a bar plot showing the number of transactions occurring every month between the start and end of the dataset.

Approach:

- The blocks dataset contains the transaction count with the time of the transaction.
- The timestamp and the transaction count are mapped from the dataset using the map function.
- With the second map function, the timestamp is used as a key, and the transaction count has been used as a value pair.
- This data is then reduced using the 'reduceByKey' method, which collects all the distinct key values (unique month and year) with the total number of transactions that occurred each month and year. This resulting data is then sorted according to the key, i.e., timestamp (Year-Month)
- The resulting sorted data is saved in the '/final_result.txt file'. This file plots a graph using Matplotlib where sorted data by year-month and the count of transactions represents the height of each bar in the plot.
- Note: The same data can be fetched through the transaction's dataset. I have tried both ways and found the same result.

Code Snippet:



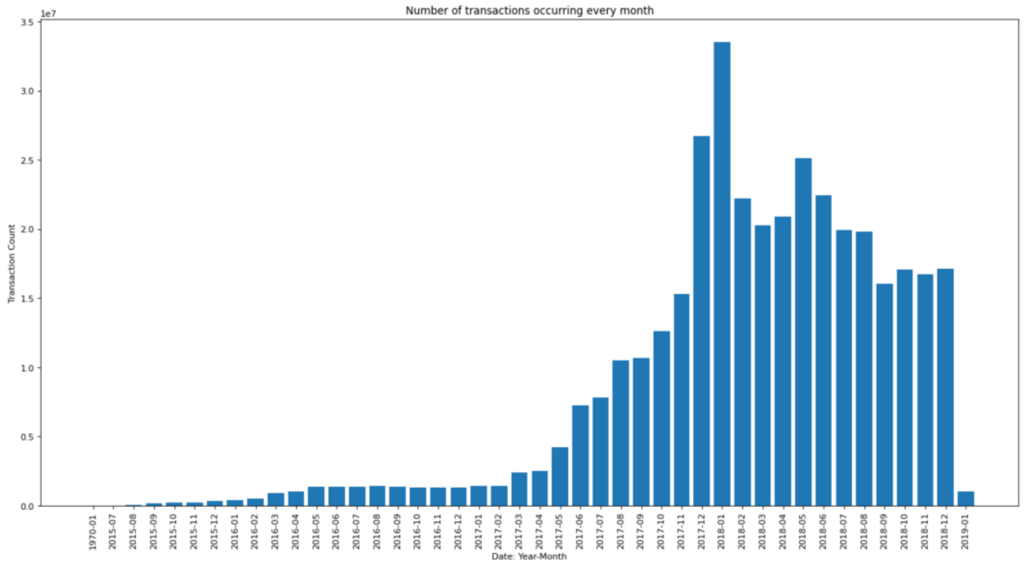
```
#### Part A: Task1

    def good_line(line):
        try:
            fields = line.split(',')
            if len(fields)!=19:
                return False
            int(fields[16])
            int(fields[17])
            return True
        except:
            return False

    lines = spark.sparkContext.textFile("s3a://" + s3_data_repository_bucket + "/ECS765/ethereum-parvulus/blocks.csv")
    clean_lines=lines.filter(good_line)

    #(Timestamp, Transaction count)
    time_transaction_data = clean_lines.map(lambda l: (l.split(',')[16], l.split(',')[17]))
    result = time_transaction_data.map(lambda t: (time.strftime("%Y-%m", time.gmtime(int(t[0]))), int(t[1])))
    final_result = result.reduceByKey(operator.add)
    transactions_per_month = final_result.sortByKey()

    my_result_object = my_bucket_resource.Object(s3_bucket,'Test'  + '/final_result.txt')
    my_result_object.put(Body=json.dumps(transactions_per_month.collect()))
```

Output:

```
[["1970-01", 0], ["2015-07", 0], ["2015-08", 85609], ["2015-09", 173805], ["2015-10", 205045], ["2015-11", 234733], ["2015-12", 347092], ["2016-01", 404816], ["2016-02",
520040], ["2016-03", 917170], ["2016-04", 1023096], ["2016-05", 1346796], ["2016-06", 1351536], ["2016-07", 1356907], ["2016-08", 1405743], ["2016-09", 1387412], ["2016-
10", 1329847], ["2016-11", 1301586], ["2016-12", 1316131], ["2017-01", 1409664], ["2017-02", 1410048], ["2017-03", 2426471], ["2017-04", 2539966], ["2017-05", 4245516],
["2017-06", 7244657], ["2017-07", 7835875], ["2017-08", 10523178], ["2017-09", 10679242], ["2017-10", 12602063], ["2017-11", 15292269], ["2017-12", 26732085], ["2018-
01", 33504270], ["2018-02", 22231978], ["2018-03", 20261862], ["2018-04", 20876642], ["2018-05", 25105717], ["2018-06", 22471788], ["2018-07", 19937033], ["2018-08",
19842059], ["2018-09", 16056742], ["2018-10", 17056926], ["2018-11", 16713911], ["2018-12", 17107601], ["2019-01", 1002431]]
```

Bar Plot:

Implementation of this graph can be found in part_A.ipynb file.



Observations:

The number of transactions increases suddenly from a very low value and reaches its highest in 2018-01. After this, the transaction count has been decreasing again.

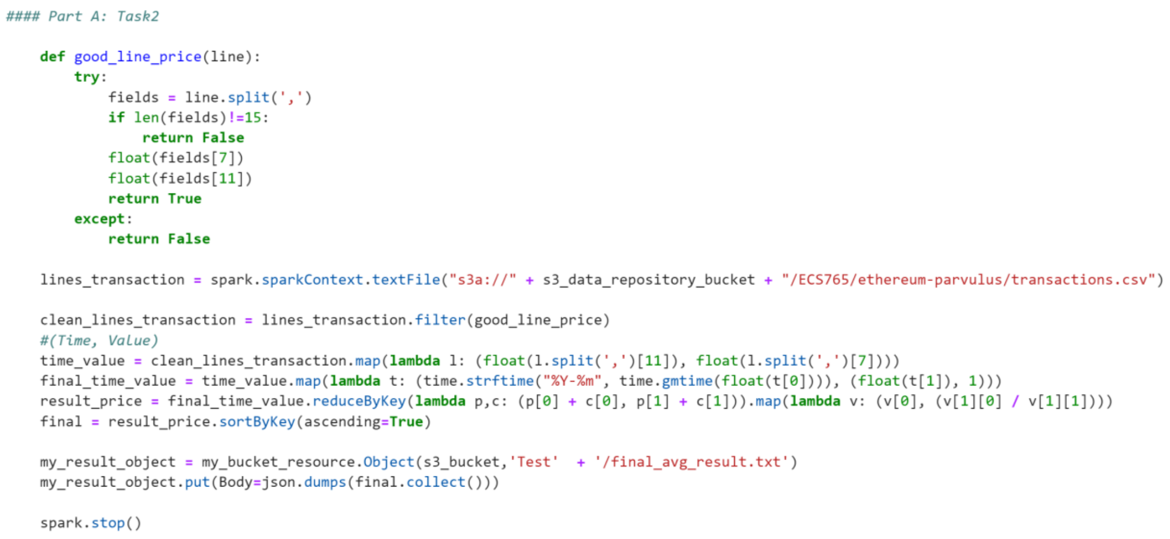
Task 2: Create a bar plot showing the average value of transactions in each month between the start and end of the dataset.

Approach:

- In order to get the average value of transactions in each month, data from the transaction's dataset is used.
- Firstly, by using the map function, we are extracting the value for each transaction and the timestamp for each transaction. (Timestamp, Transaction value)
- For the second mapper, we are using a timestamp with the unique month in the dataset as a key and transaction value and its count as a value. (Timestamp, (Transaction value,1)).

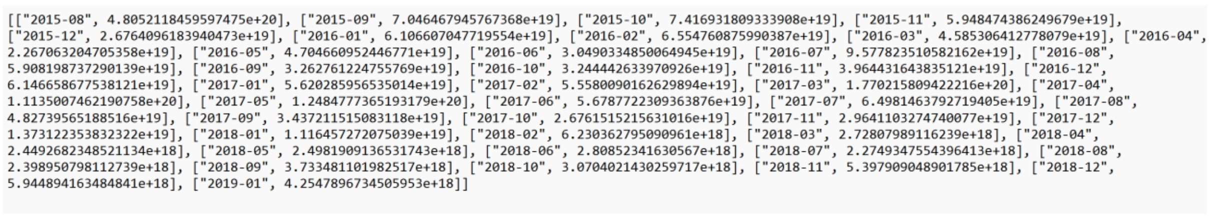
- The average is then calculated by reducing the number of key instances with their associated values by summing them up. Here we had 2 values which are, transaction value and count. The average is calculated by dividing the sum of values by the sum of the transaction count.
- The resulting data is then sorted by key using the ‘sortByKey’ method.
- This sorted data is copied to file ‘/final_avg_result.txt’ to plot the graph.
- The graph is plotted using Matplotlib where the sorted data by Year-Month and plotted against the average transaction count.

Code Snippet:



```
#### Part A: Task2

    def good_line_price(line):
        try:
            fields = line.split(',')
            if len(fields)!=15:
                return False
            float(fields[7])
            float(fields[11])
            return True
        except:
            return False

    lines_transaction = spark.sparkContext.textFile("s3a://" + s3_data_repository_bucket + "/ECS765/ethereum-parvulus/transactions.csv")

    clean_lines_transaction = lines_transaction.filter(good_line_price)
    #(Time, Value)
    time_value = clean_lines_transaction.map(lambda l: (float(l.split(',')[11]), float(l.split(',')[7])))
    final_time_value = time_value.map(lambda t: (time.strftime("%Y-%m", time.gmtime(float(t[0]))), (float(t[1]), 1)))
    result_price = final_time_value.reduceByKey(lambda p,c: (p[0] + c[0], p[1] + c[1])).map(lambda v: (v[0], (v[1][0] / v[1][1])))
    final = result_price.sortByKey(ascending=True)

    my_result_object = my_bucket_resource.Object(s3_bucket,'Test'  + '/final_avg_result.txt')
    my_result_object.put(Body=json.dumps(final.collect()))

    spark.stop()
```

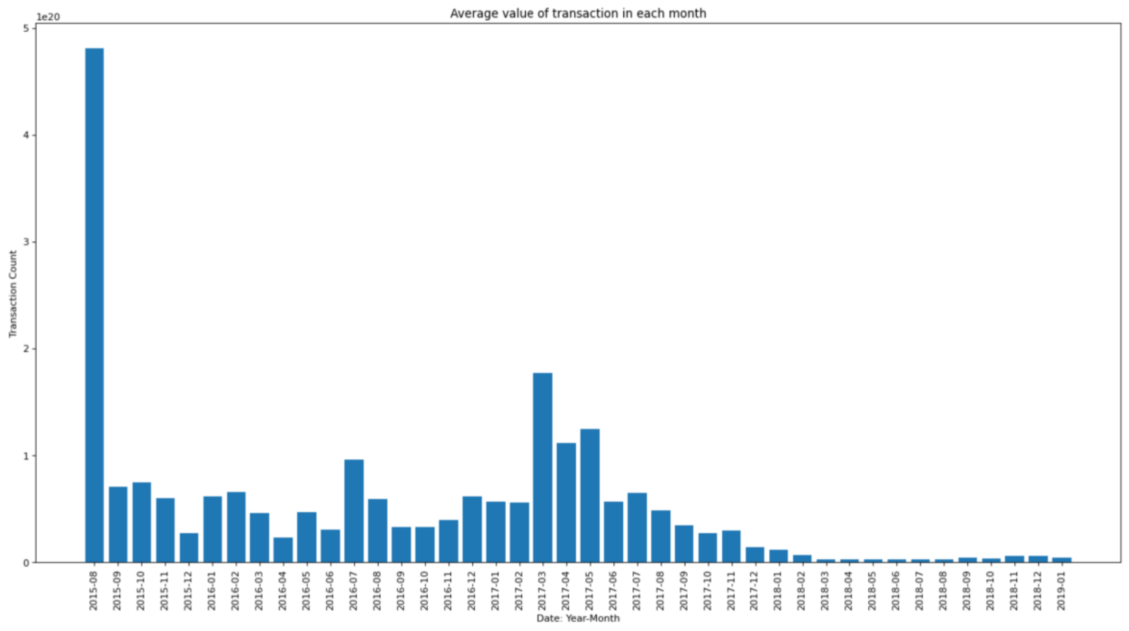
Output:



```
[["2015-08", 4.8052118459597475e+20], ["2015-09", 7.046467945767368e+19], ["2015-10", 7.416931809333908e+19], ["2015-11", 5.948474386249679e+19],
["2015-12", 2.6764096183940473e+19], ["2016-01", 6.106607047719554e+19], ["2016-02", 6.554760875990387e+19], ["2016-03", 4.585306412778079e+19], ["2016-04",
2.267063204705358e+19], ["2016-05", 4.704660952446771e+19], ["2016-06", 3.0490334850064945e+19], ["2016-07", 9.577823510582162e+19], ["2016-08",
5.908198737290139e+19], ["2016-09", 3.262761224755769e+19], ["2016-10", 3.244442633970926e+19], ["2016-11", 3.964431643835121e+19], ["2016-12",
6.146658677538121e+19], ["2017-01", 5.620285956535014e+19], ["2017-02", 5.5580090162629894e+19], ["2017-03", 1.770215809422216e+20], ["2017-04",
1.1135007462190758e+20], ["2017-05", 1.2484777365193179e+20], ["2017-06", 5.6787722309363876e+19], ["2017-07", 6.4981463792719405e+19], ["2017-08",
4.82739565188516e+19], ["2017-09", 3.437211515083118e+19], ["2017-10", 2.6761515215631016e+19], ["2017-11", 2.9641103274740077e+19], ["2017-12",
1.373122353832322e+19], ["2018-01", 1.116457272075039e+19], ["2018-02", 6.230362795090961e+18], ["2018-03", 2.72807989116239e+18], ["2018-04",
2.4492682348521134e+18], ["2018-05", 2.4981909136531743e+18], ["2018-06", 2.80852341630567e+18], ["2018-07", 2.2749347554396413e+18], ["2018-08",
2.398950798112739e+18], ["2018-09", 3.733481101982517e+18], ["2018-10", 3.0704021430259717e+18], ["2018-11", 5.397909048901785e+18], ["2018-12",
5.944894163484841e+18], ["2019-01", 4.2547896734505953e+18]]
```

Bar Plot:

Implementation of this graph can be found in part_A.ipynb file.



Observations:

From the above bar plot, we can infer that; Number of average transactions was higher in the Year 2015-08. After which they have dropped drastically to a very low value and continue too same. After the 2018 year, we can see a very negligible average transaction count.

Part B. Top Ten Most Popular Services (25%)

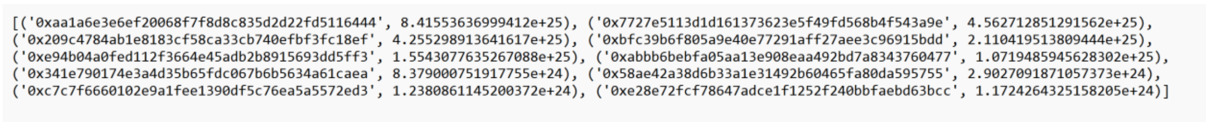
Evaluate the top 10 smart contracts by total Ether received. You will need to join **address** field in the contracts dataset to the **to_address** in the transactions dataset to determine how much ether a contract has received.

Approach:

- To evaluate the top 10 smart contracts by total ether received we need the value of ether which is present in the Transaction dataset. Due to this, we need to join the contracts dataset with the transaction dataset.
- First, the 'address' column data is extracted from the contract dataset such as (Address, 1) using the map method.
- From the Transaction dataset, 'To address' and 'Value' (Value transferred in Wei) is extracted using the map method.
- The join operation is then performed on these two datasets over the address column as the key. As a result of join operation, we received data with the address and their ether value.
- This data is then reduced to sum all the values of ether received for each address.

- In order to get the top 10 smart contracts, this data is ordered according to the value of ether received by using ‘takeOrdered’ method. This method is provided with ‘10’ as the first input as we need the top 10 contracts and then order descending to get those 10 contracts that have the highest ether value received.
- These top 10 contracts are saved in a text file named ‘part_B_top_10.txt’. The details of the file are as per below image:

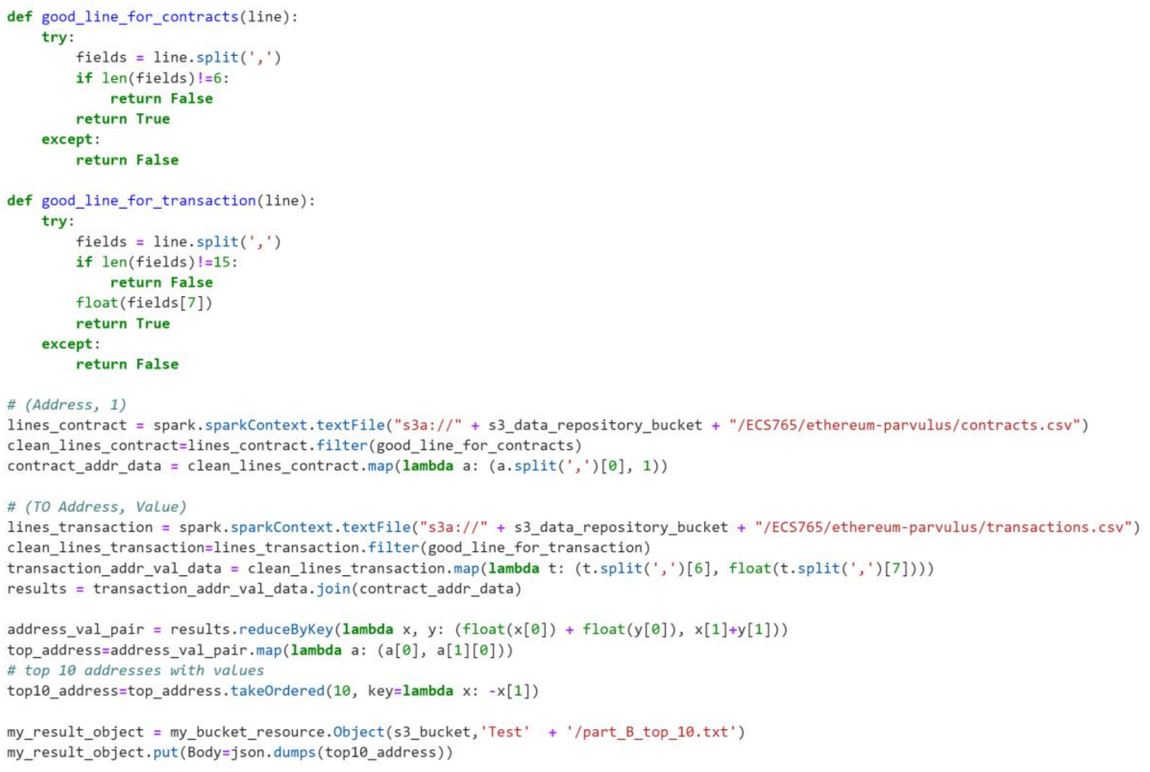
Output:



```
[('0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444', 8.41553636999412e+25), ('0x7727e5113d1d161373623e5f49fd568b4f543a9e', 4.562712851291562e+25),
('0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef', 4.255298913641617e+25), ('0xbfc39b6f805a9e40e77291aff27aee3c96915bdd', 2.110419513809444e+25),
('0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', 1.5543077635267088e+25), ('0xabbb6bebfa05aa13e908eaa492bd7a8343760477', 1.0719485945628302e+25),
('0x341e790174e3a4d35b65fdc067b6b5634a61caea', 8.379000751917755e+24), ('0x58ae42a38d6b33a1e31492b60465fa80da595755', 2.9027091871057373e+24),
('0xc7c7f6660102e9a1fee1390df5c76ea5a5572ed3', 1.2380861145200372e+24), ('0xe28e72fcf78647adce1f1252f240bbfaebd63bcc', 1.1724264325158205e+24)]
```

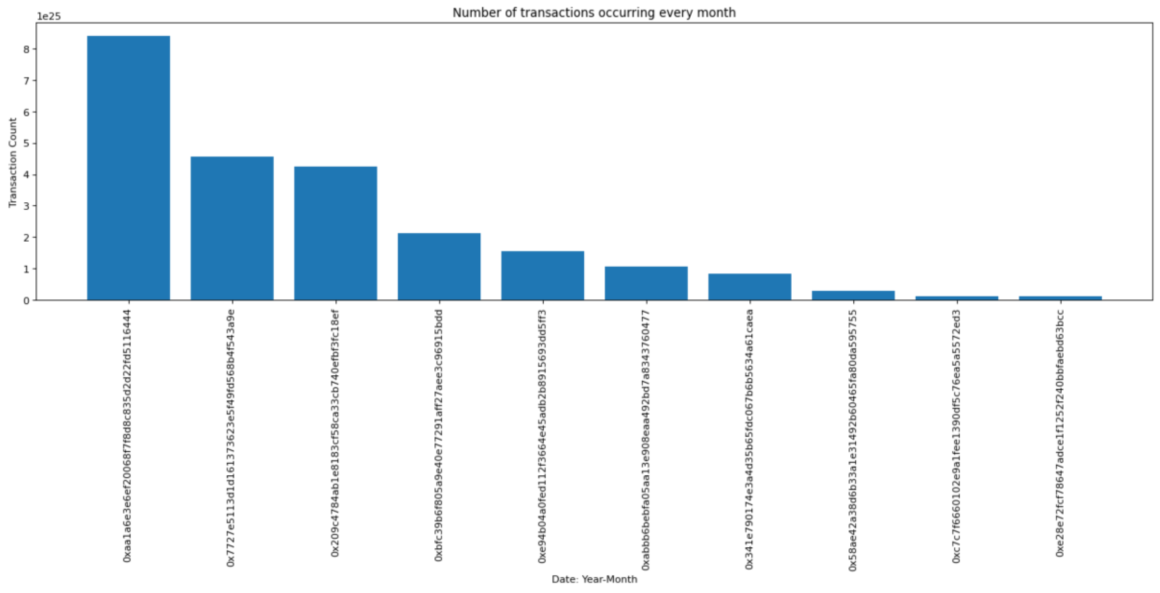
Figure 1. Top Ten Most Popular Services

Code Snippet:



```python
def good_line_for_contracts(line):
    try:
        fields = line.split(',')
        if len(fields)!=6:
            return False
        return True
    except:
        return False

def good_line_for_transaction(line):
    try:
        fields = line.split(',')
        if len(fields)!=15:
            return False
        float(fields[7])
        return True
    except:
        return False

# (Address, 1)
lines_contract = spark.sparkContext.textFile("s3a://" + s3_data_repository_bucket + "/ECS765/ethereum-parvulus/contracts.csv")
clean_lines_contract=lines_contract.filter(good_line_for_contracts)
contract_addr_data = clean_lines_contract.map(lambda a: (a.split(',')[0], 1))

# (TO Address, Value)
lines_transaction = spark.sparkContext.textFile("s3a://" + s3_data_repository_bucket + "/ECS765/ethereum-parvulus/transactions.csv")
clean_lines_transaction=lines_transaction.filter(good_line_for_transaction)
transaction_addr_val_data = clean_lines_transaction.map(lambda t: (t.split(',')[6], float(t.split(',')[7])))
results = transaction_addr_val_data.join(contract_addr_data)

address_val_pair = results.reduceByKey(lambda x, y: (float(x[0]) + float(y[0]), x[1]+y[1]))
top_address=address_val_pair.map(lambda a: (a[0], a[1][0]))
# top 10 addresses with values
top10_address=top_address.takeOrdered(10, key=lambda x: -x[1])

my_result_object = my_bucket_resource.Object(s3_bucket,'Test'  + '/part_B_top_10.txt')
my_result_object.put(Body=json.dumps(top10_address))
```

Graph Plot:



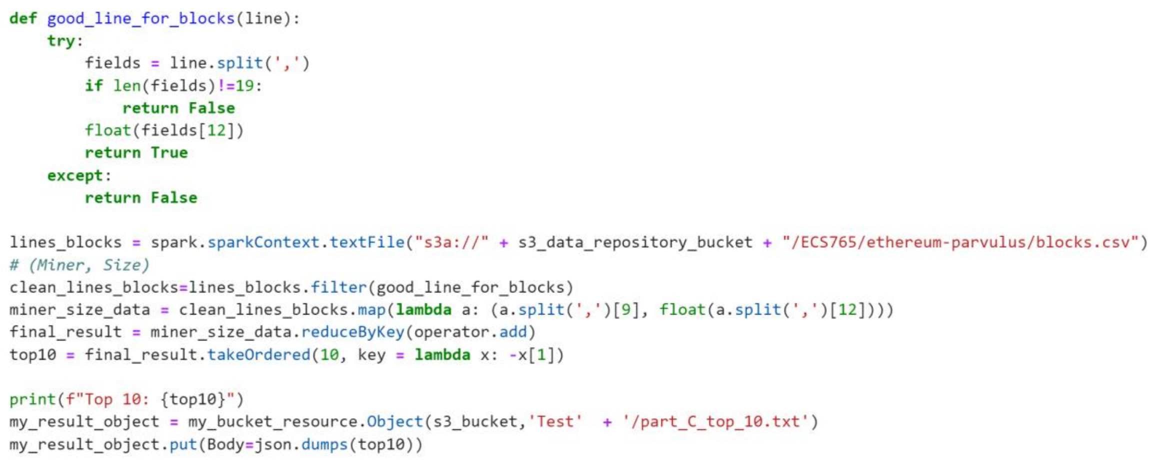
Part C. Top Ten Most Active Miners (10%)

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.

Approach:

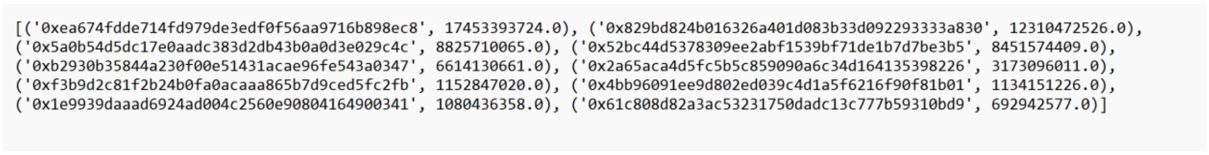
- To get the most active miners by the size of blocks mined, we need to extract the Miner (The address of the beneficiary to whom the mining rewards were given) and the size (The size of this block in bytes) of the block from the blocks.csv file.
- We got the miner and their activity by mapping each block size to the miner as a key in the given blocks dataset. Such as, (Miner, Size).
- This data is reduced to a number of key instances by summing the size value for each miner, to get the block size mined by each miner.
- The resulting data is sorted according to the size value and provides us top 10 active miners.
- These top 10 most active miners are saved in a text file named ‘part_C_top_10.txt’. The details of the file are as per below output image.

Code Snippet:



```
def good_line_for_blocks(line):
    try:
        fields = line.split(',')
        if len(fields)!=19:
            return False
        float(fields[12])
        return True
    except:
        return False

lines_blocks = spark.sparkContext.textFile("s3a://" + s3_data_repository_bucket + "/ECS765/ethereum-parvulus/blocks.csv")
# (Miner, Size)
clean_lines_blocks=lines_blocks.filter(good_line_for_blocks)
miner_size_data = clean_lines_blocks.map(lambda a: (a.split(',')[9], float(a.split(',')[12])))
final_result = miner_size_data.reduceByKey(operator.add)
top10 = final_result.takeOrdered(10, key = lambda x: -x[1])

print(f"Top 10: {top10}")
my_result_object = my_bucket_resource.Object(s3_bucket,'Test'  + '/part_C_top_10.txt')
my_result_object.put(Body=json.dumps(top10))
```

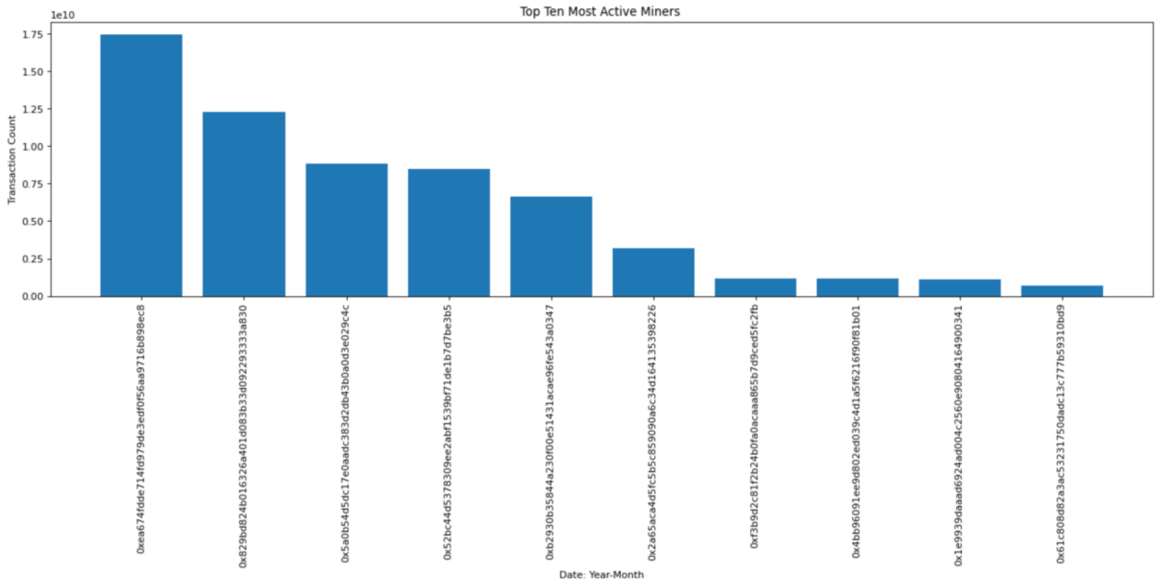
Output:



```
[('0xea674fdde714fd979de3edf0f56aa9716b898ec8', 17453393724.0), ('0x829bd824b016326a401d083b33d092293333a830', 12310472526.0),
('0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c', 8825710065.0), ('0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5', 8451574409.0),
('0xb2930b35844a230f00e51431acae96fe543a0347', 6614130661.0), ('0x2a65aca4d5fc5b5c859090a6c34d164135398226', 3173096011.0),
('0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb', 1152847020.0), ('0x4bb96091ee9d802ed039c4d1a5f6216f90f81b01', 1134151226.0),
('0x1e9939daaad6924ad004c2560e90804164900341', 1080436358.0), ('0x61c808d82a3ac53231750dadc13c777b59310bd9', 692942577.0)]
```

Figure 2. Top 10 most active miners

Graph Plot:



Part D. Data exploration (40%)

Scam Analysis:

1. Popular Scams:

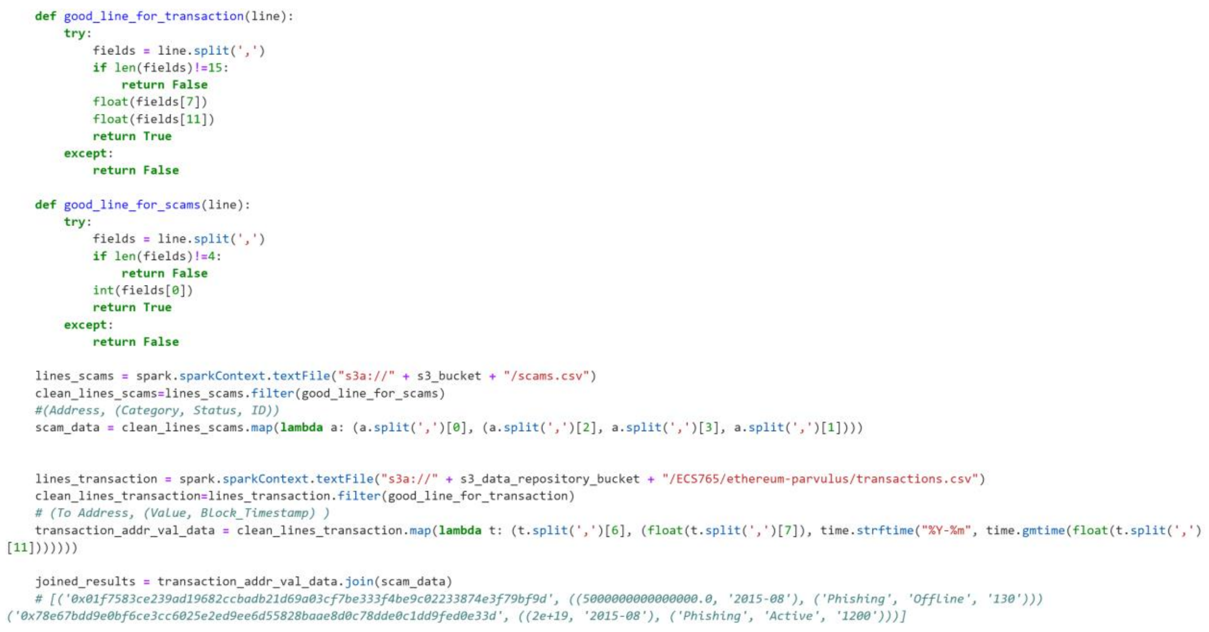
Utilising the provided scam dataset, what is the most lucrative form of scam? How does this change throughout time, and does this correlate with certain known scams going offline/inactive? To obtain the marks for this category you should provide the id of the most lucrative scam and a graph showing how the ether received has changed over time for the dataset. (20%/40%)

Approach:

To achieve this, we needed data from the scams.json file. I have converted the given scams.json file to a scams.csv file by expanding the list of addresses to a single address. This converted scams.csv file is then copied to my local bucket. This file has the same data as the provided scams.csv.

- First, Address, Category, status, and ID from the scams.csv are extracted using the map method with 'Address' as the key and others as values.
  E.g. ('0x00e01a648ff41346cdeb873182383333d2184dd1', ('Phishing', 'Offline', '130'))
- To Address, Value, Block_Timestamp are extracted from the transaction.csv.
  E.g. ('0x6baf48e1c966d16559ce2dddb616ffa72004851e', (5000000000000000.0, '2015-08'))
- These two datasets are then joined with Address as a key and all others as a value. Such as, (Address, (Value, Timestamp), (Category, status, ID))
  E.g. ('0x01f7583ce239ad19682ccbadb21d69a03cf7be333f4be9c02233874e3f79bf9d', ((5000000000000000.0, '2015-08'), ('Phishing', 'Offline', '130')))

Code Snippet:

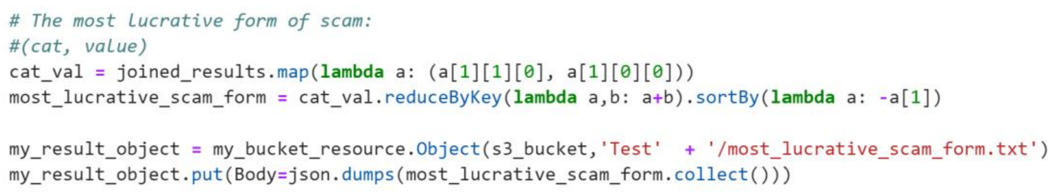


```
def good_line_for_transaction(line):
    try:
        fields = line.split(',')
        if len(fields)!=15:
            return False
        float(fields[7])
        float(fields[11])
        return True
    except:
        return False

def good_line_for_scams(line):
    try:
        fields = line.split(',')
        if len(fields)!=4:
            return False
        int(fields[0])
        return True
    except:
        return False

lines_scams = spark.sparkContext.textFile("s3a://" + s3_bucket + "/scams.csv")
clean_lines_scams=lines_scams.filter(good_line_for_scams)
#(Address, (Category, Status, ID))
scam_data = clean_lines_scams.map(lambda a: (a.split(',')[0], (a.split(',')[2], a.split(',')[3], a.split(',')[1])))


lines_transaction = spark.sparkContext.textFile("s3a://" + s3_data_repository_bucket + "/ECS765/ethereum-parvulus/transactions.csv")
clean_lines_transaction=lines_transaction.filter(good_line_for_transaction)
# (To Address, (Value, Block_Timestamp) )
transaction_addr_val_data = clean_lines_transaction.map(lambda t: (t.split(',')[6], (float(t.split(',')[7]), time.strftime("%Y-%m", time.gmtime(float(t.split(',')
[11]))))))

joined_results = transaction_addr_val_data.join(scam_data)
# [('0x01f7583ce239ad19682ccbadb21d69a03cf7be333f4be9c02233874e3f79bf9d', ((5000000000000000.0, '2015-08'), ('Phishing', 'Offline', '130')))
('0x78e67bdd9e0bf6ce3cc6025e2ed9ee6d55828baae8d0c78dde0c1dd9fed0e33d', ((2e+19, '2015-08'), ('Phishing', 'Active', '1200')))]
```

A. What is the most lucrative form of scam?

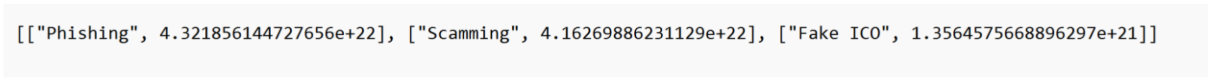
- To get the 'The most lucrative form of scam', the category and the values are mapped with the category being key. This is done with the help of the map method.
- The data is then reduced by key by summing up these values for each unique category. This data is sorted with the value to get the most lucrative form of scam.
- This data is saved in a file 'most_lucrative_scam_form.txt'.

Code Snippet:



```
# The most lucrative form of scam:
#(cat, value)
cat_val = joined_results.map(lambda a: (a[1][1][0], a[1][0][0]))
most_lucrative_scam_form = cat_val.reduceByKey(lambda a,b: a+b).sortBy(lambda a: -a[1])

my_result_object = my_bucket_resource.Object(s3_bucket,'Test'  + '/most_lucrative_scam_form.txt')
my_result_object.put(Body=json.dumps(most_lucrative_scam_form.collect()))
```

Output



```
[["Phishing", 4.321856144727656e+22], ["Scamming", 4.16269886231129e+22], ["Fake ICO", 1.3564575668896297e+21]]
```

Figure 3. The most lucrative form of scam

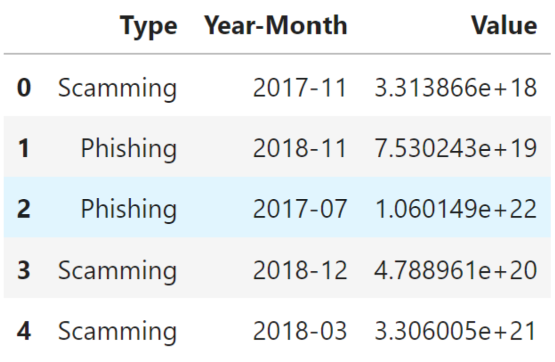
The most lucrative form of scam?

Phishing: 4.321856144727656e+22

B. How does this change throughout time:

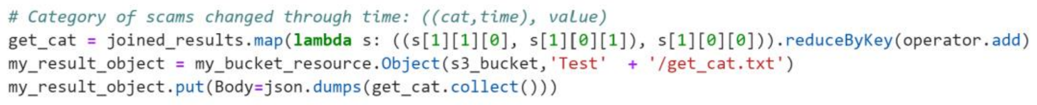
- To understand how the most lucrative scam changes through time, we need to collect the category and Value and understand how it changes with time.
- The mapper is used to extract this data with category and time as a key and transaction value as value, such as ((category, time), value).
- This transaction value is then summed up for each unique category for a given timestamp. This is then reduced by using reduceByKey.
- The resulting data is saved in the file, get_cat.txt. This data is then plotted with a timestamp on X-axis and a value on Y-axis.

Output Sample:



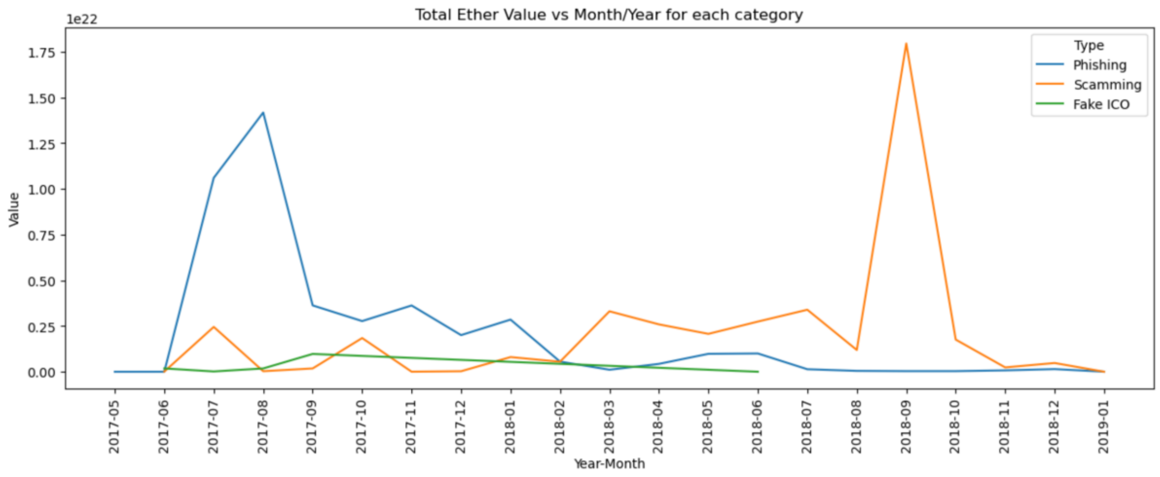
	Type	Year-Month	Value
0	Scamming	2017-11	3.313866e+18
1	Phishing	2018-11	7.530243e+19
2	Phishing	2017-07	1.060149e+22
3	Scamming	2018-12	4.788961e+20
4	Scamming	2018-03	3.306005e+21

Code Snippet:



```
# Category of scams changed through time: ((cat,time), value)
get_cat = joined_results.map(lambda s: ((s[1][1][0], s[1][0][1]), s[1][0][0])).reduceByKey(operator.add)
my_result_object = my_bucket_resource.Object(s3_bucket,'Test'  + '/get_cat.txt')
my_result_object.put(Body=json.dumps(get_cat.collect()))
```

Graph plot:



Observation:

Phishing:

Ether Value has increased suddenly from 0 in 2017-06 to the highest value in 2017-08. This dropped drastically after 2017-08 and gradually decrease to 0.

Scamming:

Ether value has been gradually changing from 0 to 0.25 from 2017-06 till 2018-08. This drastically increased in 2018-09 till the highest value and decreases to 0.

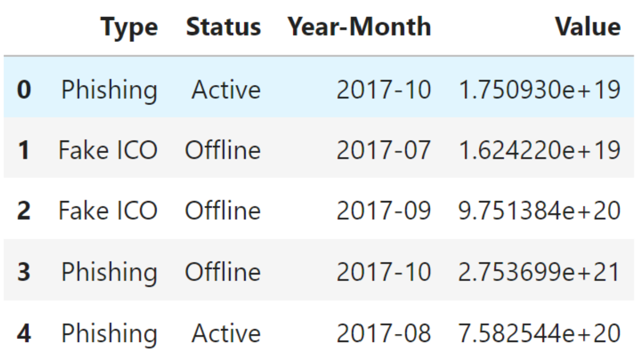
Fake ICO:

The ether value has been almost consistent with values less than 0.25 from 2017-06 till 2018-06. There is no ether that has been received after that.

C. Does this correlate with certain known scams going offline/inactive?

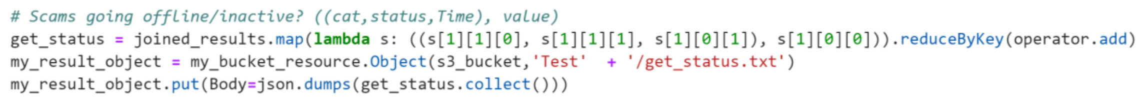
- In order to understand how the different categories, correlate with scams going offline/inactive, we need to fetch the status with category and time with respect to ether value.
- With map method, we fetch this data using Category, status, and Time as a key and ether value as a value. ((Category, Status, Time), Value)
- This transaction value is then summed up for each unique category and status for a given timestamp. This is then reduced by using reduceByKey.
- The resulting data is saved in the file, get_status.txt. This data is then plotted with a timestamp on X-axis and a value on Y-axis.

Output Sample:



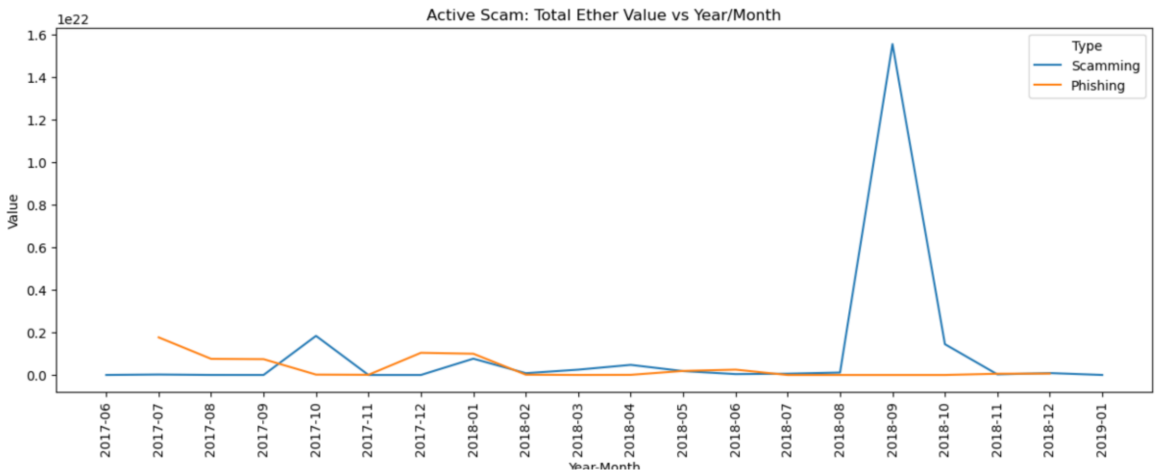
	Type	Status	Year-Month	Value
0	Phishing	Active	2017-10	1.750930e+19
1	Fake ICO	Offline	2017-07	1.624220e+19
2	Fake ICO	Offline	2017-09	9.751384e+20
3	Phishing	Offline	2017-10	2.753699e+21
4	Phishing	Active	2017-08	7.582544e+20

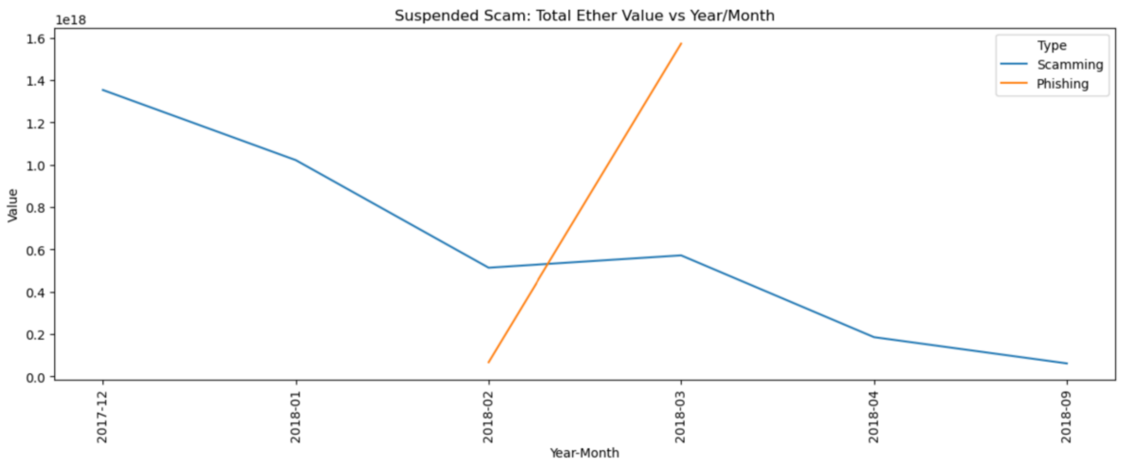
Code Snippet:

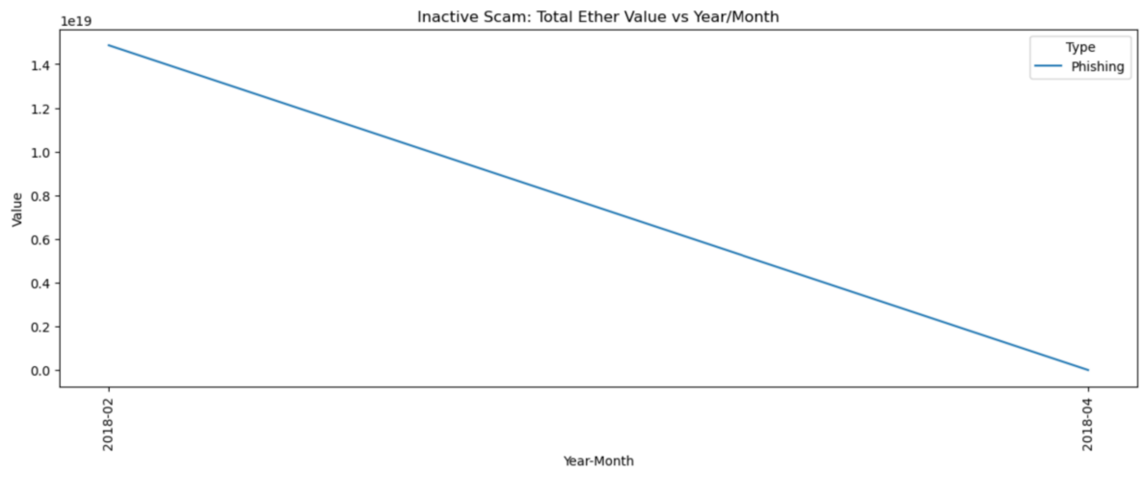


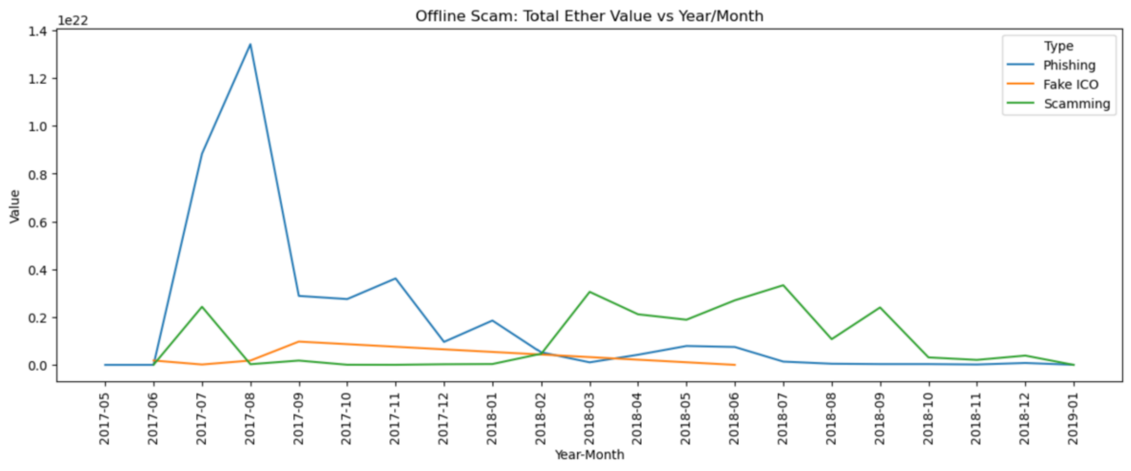
```
# Scams going offline/inactive? ((cat,status,Time), value)
get_status = joined_results.map(lambda s: ((s[1][1][0], s[1][1][1], s[1][0][1]), s[1][0][0])).reduceByKey(operator.add)
my_result_object = my_bucket_resource.Object(s3_bucket,'Test'  + '/get_status.txt')
my_result_object.put(Body=json.dumps(get_status.collect()))
```

Graph plot:



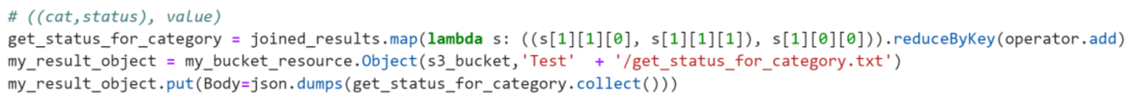






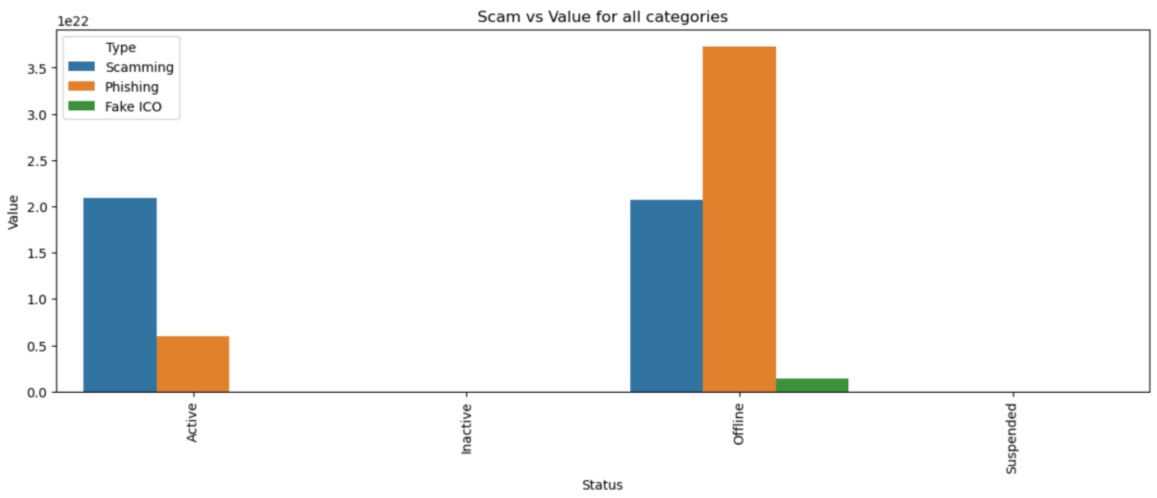
- We can also get the same values when we map the data for category and Status with respect to value. Such as ((Category, Status), Value). This will provide us sum of all the values for the unique status and category by using the reducer.

Code Snippet:



```
# ((cat,status), value)
get_status_for_category = joined_results.map(lambda s: ((s[1][1][0], s[1][1][1]), s[1][0][0])).reduceByKey(operator.add)
my_result_object = my_bucket_resource.Object(s3_bucket,'Test'  + '/get_status_for_category.txt')
my_result_object.put(Body=json.dumps(get_status_for_category.collect()))
```

Graph Plot:



Observation:

From all the above graphs plots, we can observe that for suspended and inactive scam status we don’t see all categories present. For **Active scam** status, we can observe that “Scamming” has more value than “Phishing”, For **Offline scam** status, we can infer that it has the highest value for the “Phishing” category and after that, the value reduces further for Scamming and Fake ICO.

D. The id of the most lucrative scam:

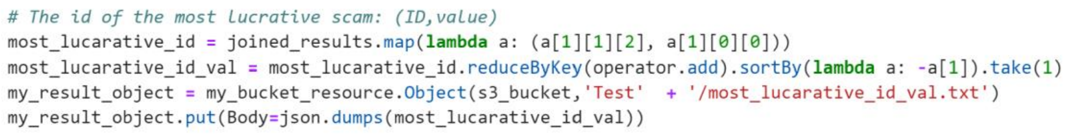
- To get the ID of the most lucrative scam, we need to extract the ID and the ether value for all transactions in the dataset.
- The map method is used to fetch the transaction ID and value, such as (ID, value), with ID as a key and ether value as a value.
- By using the reducer method (reduceByKey), we add all the values for each unique ID.
- This result data is then sorted according to the value.
- In order to get the ID of the most lucrative scam, will only fetch the first value in the dataset along with its ID.
- This data is saved in the file ‘most_lucarative_id_val.txt’.

Output:

ID of the most lucrative scam:

```
[["5622", 1.6709083588072867e+22]]
```

Code Snippet:

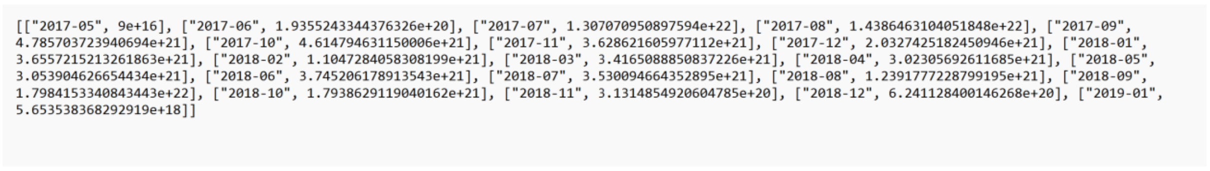


```python
# The id of the most lucrative scam: (ID,value)
most_lucarative_id = joined_results.map(lambda a: (a[1][1][2], a[1][0][0]))
most_lucarative_id_val = most_lucarative_id.reduceByKey(operator.add).sortBy(lambda a: -a[1]).take(1)
my_result_object = my_bucket_resource.Object(s3_bucket,'Test'  + '/most_lucarative_id_val.txt')
my_result_object.put(Body=json.dumps(most_lucarative_id_val))
```

E. How the ether received has changed over time?

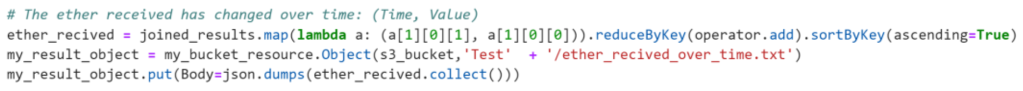
- To understand how ether received has changed over time, we need to extract the data for the Timestamp and ether value.
- By using the map method, we fetch the data for Timestamp and Ether value.
- With the help of the reducer, we sum up all the values for a particular Month-Year.
- Then we need to sort this data according to the timestamp in ascending order.
- This data is saved in ‘ether_recived_over_time.txt’ file.

Output:



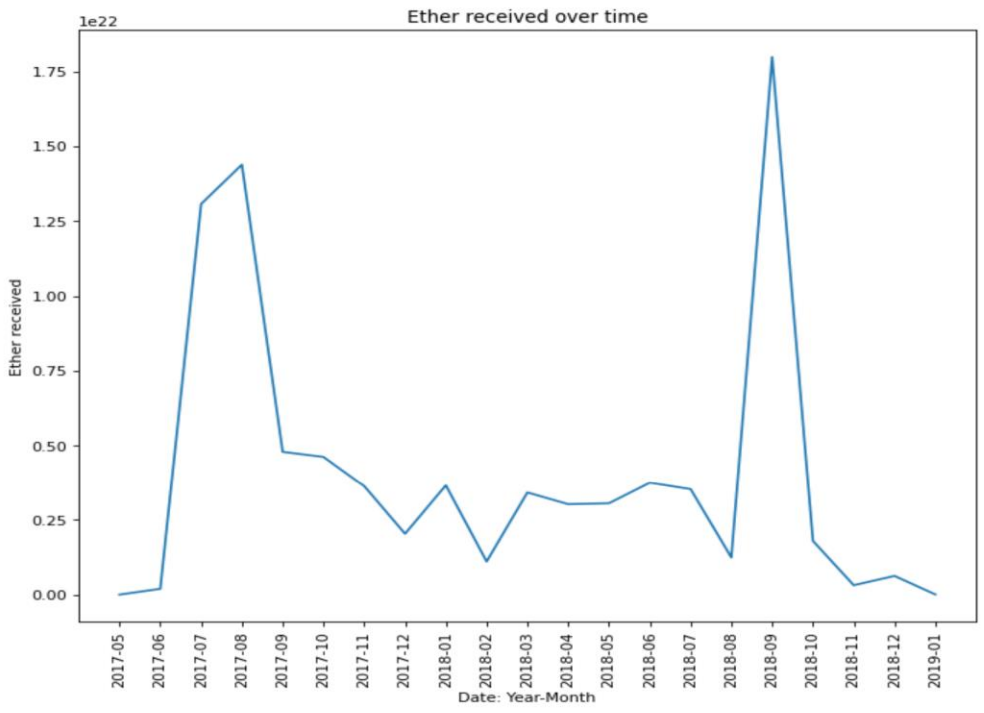
```
[["2017-05", 9e+16], ["2017-06", 1.9355243344376326e+20], ["2017-07", 1.307070950897594e+22], ["2017-08", 1.4386463104051848e+22], ["2017-09",
4.785703723940694e+21], ["2017-10", 4.614794631150006e+21], ["2017-11", 3.628621605977112e+21], ["2017-12", 2.0327425182450946e+21], ["2018-01",
3.6557215213261863e+21], ["2018-02", 1.1047284058308199e+21], ["2018-03", 3.4165088850837226e+21], ["2018-04", 3.02305692611685e+21], ["2018-05",
3.053904626654434e+21], ["2018-06", 3.745206178913543e+21], ["2018-07", 3.530094664352895e+21], ["2018-08", 1.2391777228799195e+21], ["2018-09",
1.7984153340843443e+22], ["2018-10", 1.7938629119040162e+21], ["2018-11", 3.1314854920604785e+20], ["2018-12", 6.241128400146268e+20], ["2019-01",
5.653538368292919e+18]]
```

Code Snippet:



```python
# The ether received has changed over time: (Time, Value)
ether_recived = joined_results.map(lambda a: (a[1][0][1], a[1][0][0])).reduceByKey(operator.add).sortByKey(ascending=True)
my_result_object = my_bucket_resource.Object(s3_bucket,'Test'  + '/ether_recived_over_time.txt')
my_result_object.put(Body=json.dumps(ether_recived.collect()))
```

Graph Plot:



Observation:

We can observe from the above graph that initially the ether value increased drastically till August 2017. After this, the values decreased suddenly and remain low till August 2018. Again, we can see the spike in value in September 2018 and reach its lowest again.

Part D. Data exploration (40%)

Miscellaneous Analysis

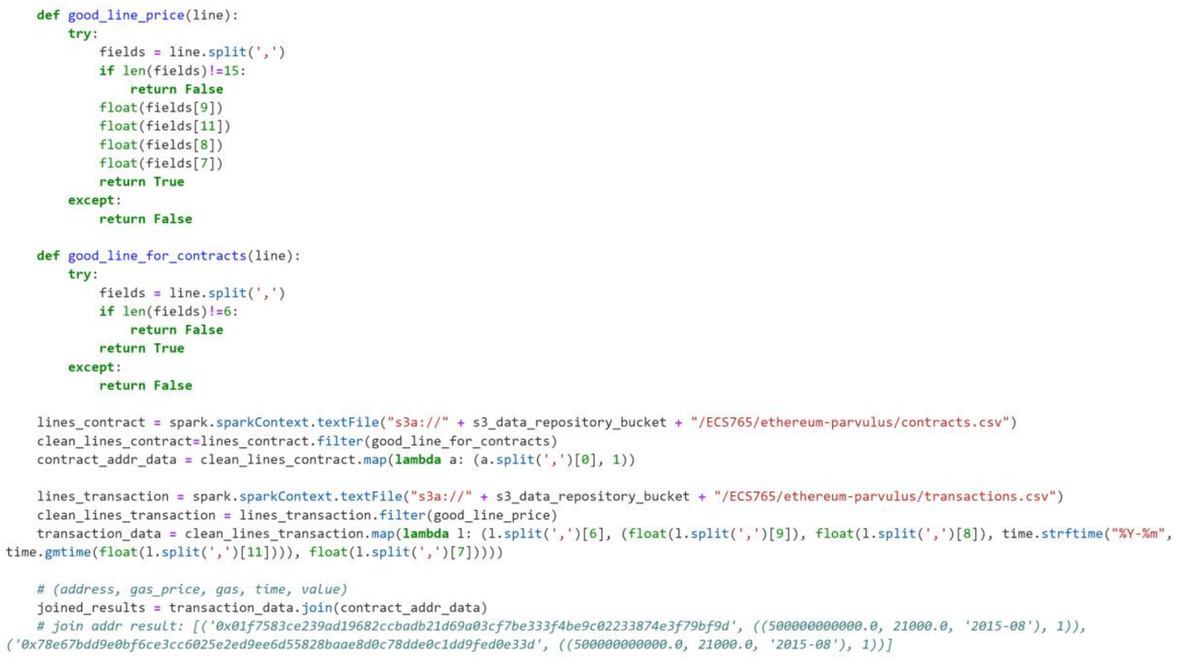
1.Gas Guzzlers:

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? How does this correlate with your results seen within Part B. To obtain these marks you should provide a graph showing how gas price has changed over time, a graph showing how gas used for contract transactions has changed over time and identify if the most popular contracts use more or less than the average gas_used. (20%/40%)

Approach:

- In order to get the details of gas guzzlers, we need to get the data from contracts.csv and Transaction.csv.
- From contracts.csv, we will get the data for: (Address, count) using the Map method.
- From Transaction.csv, we will get the data for: (Address, gas_price, gas, time, value) using the Map method.
- We will join these two columns in order to get the common address between these two datasets.

Code Snippet:

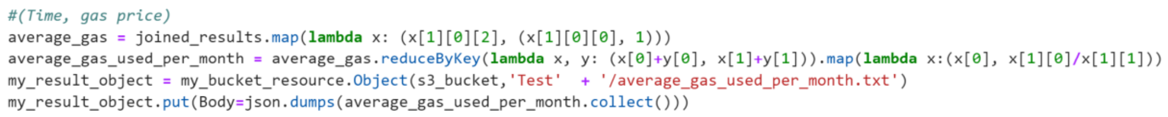


```
    def good_line_price(line):
        try:
            fields = line.split(',')
            if len(fields)!=15:
                return False
            float(fields[9])
            float(fields[11])
            float(fields[8])
            float(fields[7])
            return True
        except:
            return False

    def good_line_for_contracts(line):
        try:
            fields = line.split(',')
            if len(fields)!=6:
                return False
            return True
        except:
            return False

    lines_contract = spark.sparkContext.textFile("s3a://" + s3_data_repository_bucket + "/ECS765/ethereum-parvulus/contracts.csv")
    clean_lines_contract=lines_contract.filter(good_line_for_contracts)
    contract_addr_data = clean_lines_contract.map(lambda a: (a.split(',')[0], 1))

    lines_transaction = spark.sparkContext.textFile("s3a://" + s3_data_repository_bucket + "/ECS765/ethereum-parvulus/transactions.csv")
    clean_lines_transaction = lines_transaction.filter(good_line_price)
    transaction_data = clean_lines_transaction.map(lambda l: (l.split(',')[6], (float(l.split(',')[9]), float(l.split(',')[8]), time.strftime("%Y-%m",
time.gmtime(float(l.split(',')[11]))), float(l.split(',')[7]))))

    # (address, gas_price, gas, time, value)
    joined_results = transaction_data.join(contract_addr_data)
    # join addr result: [('0x01f7583ce239ad19682ccbadb21d69a03cf7be333f4be9c02233874e3f79bf9d', ((500000000000.0, 21000.0, '2015-08'), 1)),
('0x78e67bdd9e0bf6ce3cc6025e2ed9ee6d55828baae8d0c78dde0c1dd9fed0e33d', ((500000000000.0, 21000.0, '2015-08'), 1))]
```

A. How has gas price changed over time?

- To understand how gas price changed over time, we need to get the data for gas price over each year-month.
- To do that, we first used the map function to map the values of gas price and their count with timestamps. Here timestamp will be the key and the gas price, and its count will be the value. (Time, (Gas price, 1))
- With the help of reduce method 'reduceByKey', we will sum all the gas prices and their counts for each unique Year-Month timestamp.
- To understand how gas price changed over time, will take an average of gas price by dividing it by its count.
- This data is saved in file: '/average_gas_used_per_month.txt'.

Code Snippet:

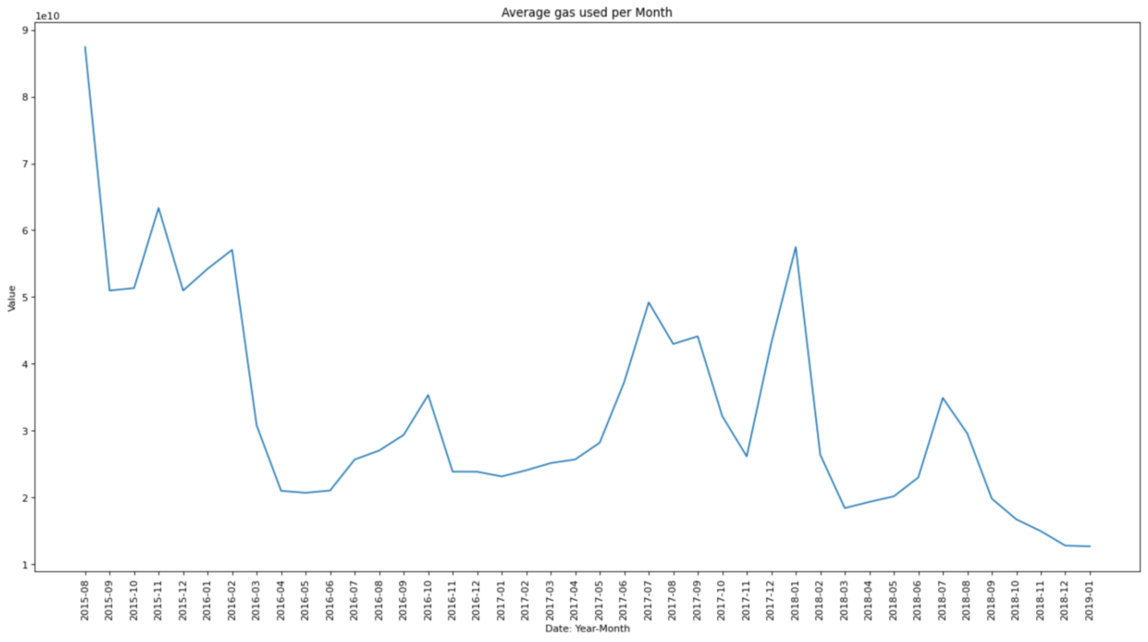


```
#(Time, gas price)
average_gas = joined_results.map(lambda x: (x[1][0][2], (x[1][0][0], 1)))
average_gas_used_per_month = average_gas.reduceByKey(lambda x, y: (x[0]+y[0], x[1]+y[1])).map(lambda x:(x[0], x[1][0]/x[1][1]))
my_result_object = my_bucket_resource.Object(s3_bucket,'Test'  + '/average_gas_used_per_month.txt')
my_result_object.put(Body=json.dumps(average_gas_used_per_month.collect()))
```

Output:

```
[["2018-04", 19327021493.842545], ["2017-06", 37260296374.11382], ["2017-01", 23140757145.28418], ["2015-09", 50970286230.00969], ["2016-09", 29355781913.194828],
["2017-07", 49202267803.197105], ["2017-12", 43092243916.64027], ["2018-11", 14949633382.839464], ["2016-04", 20982525575.672344], ["2018-12", 12784296289.616867],
["2017-10", 32178750484.971848], ["2016-12", 23842408499.962578], ["2017-04", 25685283554.467873], ["2015-11", 63333520976.56268], ["2018-01", 57483012664.48943],
["2018-05", 20155196674.953697], ["2018-06", 22986246106.07143], ["2017-09", 44113491149.92766], ["2016-07", 25672283224.261543], ["2018-02", 26389790971.973385],
["2015-08", 87416140025.80464], ["2016-11", 23854715991.613033], ["2016-02", 57063407819.05038], ["2017-05", 28191710621.506485], ["2016-03", 30798185119.328835],
["2016-05", 20692511462.501236], ["2015-10", 51339163158.0504], ["2019-01", 12680215019.414627], ["2015-12", 50966162647.0882], ["2016-01", 54225575726.05373], ["2017-
03", 25137010162.101383], ["2017-11", 26135201845.21011], ["2018-09", 19818069788.69361], ["2018-03", 18398010693.100178], ["2018-07", 34896729295.40592], ["2018-08",
29569650269.76273], ["2017-02", 24060845952.717495], ["2016-10", 35331841650.01722], ["2016-06", 21033795937.93862], ["2018-10", 16715167752.054398], ["2016-08",
27014569490.407352], ["2017-08", 42974548561.49431]]
```

Graph Plot: (How gas price has changed over time?)



Observation:

From the above graph, we can observe that the gas price has reduced over time increases. We can notice that gas price was their highest in August 2018 and continuously reduced with time and reach their lowest in January 2019.

B. Have contracts become more complicated, requiring more gas, or less so?

- In order to understand whether contracts become more complicated, we need to verify whether contracts are requiring more gas over time.
- To obtain these results, we used the map method to map Timestamp and gas used from the joined result. In this case, timestamp (Year-Month) will be the kay and Gas will be the value.
- With the help of Reduce method, we then sum all the gas used for each month.
- The resulting data is saved in the file: ‘/average_gas_used_per_month.txt’

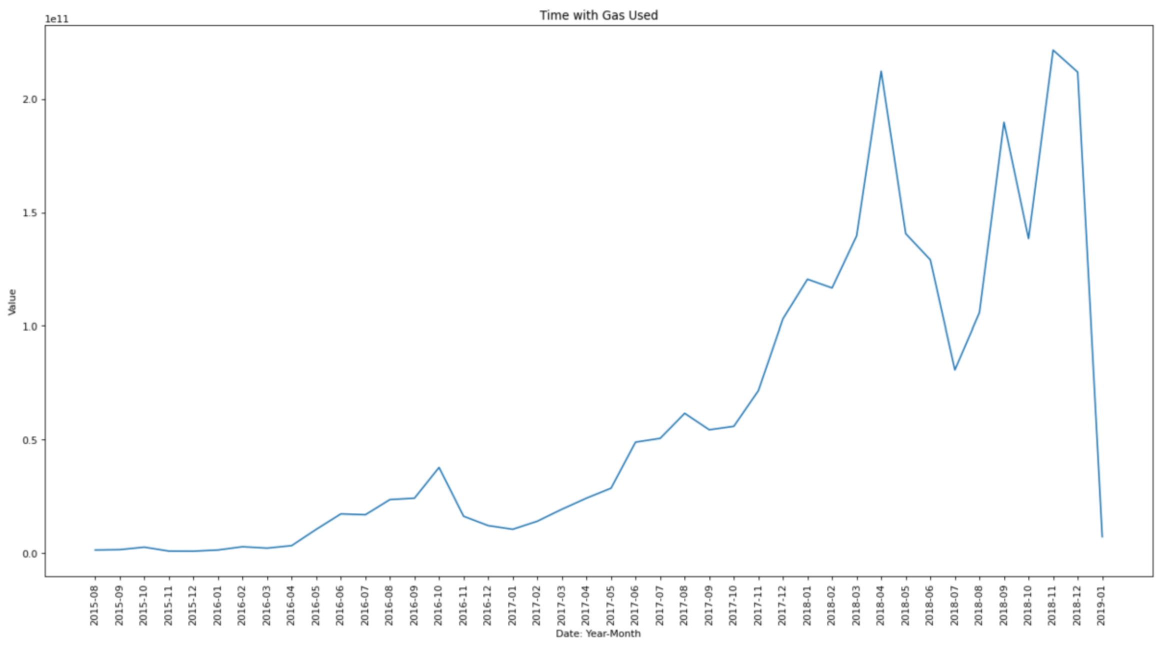
Code Snippet:

```
 #(Time, Gas)
time_with_gas_used = joined_results.map(lambda x: (x[1][0][2], x[1][0][1])).reduceByKey(operator.add)
my_result_object = my_bucket_resource.Object(s3_bucket,'Test'  + '/time_with_gas_used.txt')
my_result_object.put(Body=json.dumps(time_with_gas_used.collect()))
```

Output:

```
[["2018-04", 212180115965.0], ["2017-06", 48947512173.0], ["2017-01", 10548495524.0], ["2015-09", 1567904257.0], ["2016-09", 24252053441.0], ["2017-07", 50583880582.0],
["2017-12", 103103851168.0], ["2018-11", 221485460606.0], ["2016-04", 3331252064.0], ["2018-12", 211782713065.0], ["2017-10", 55919177677.0], ["2016-12", 12164904201.0],
["2017-04", 24254186931.0], ["2015-11", 938976373.0], ["2018-01", 120527045688.0], ["2018-05", 140592879480.0], ["2018-06", 129105739103.0], ["2017-09", 54403177130.0],
["2016-07", 16937163595.0], ["2018-02", 116685302454.0], ["2015-08", 1415324814.0], ["2016-11", 16275236784.0], ["2016-02", 2853087522.0], ["2017-05", 28612804402.0],
["2016-03", 2234018589.0], ["2016-05", 10514452171.0], ["2015-10", 2660854053.0], ["2019-01", 7255970274.0], ["2015-12", 899923777.0], ["2016-01", 1415062358.0], ["2017-
03", 19359220588.0], ["2017-11", 71605737718.0], ["2018-09", 189702922783.0], ["2018-03", 139627787856.0], ["2018-07", 80766277787.0], ["2018-08", 105853762137.0],
["2017-02", 14091875159.0], ["2016-10", 37785173037.0], ["2016-06", 17296162070.0], ["2018-10", 138426150360.0], ["2016-08", 23633048747.0], ["2017-08", 61629918609.0]]
```

Graph plot: (How gas used for contract transactions has changed over time)



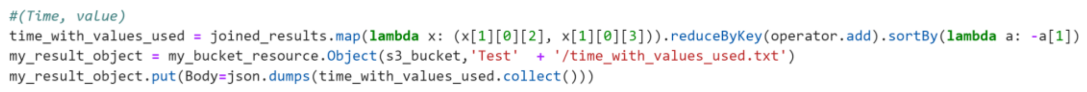
Observation:

From the above graph, we can conclude that, as time increases gas use is getting increased. That means we can infer that the contracts have become more complicated.

C. How does this correlate with your results seen within Part B.

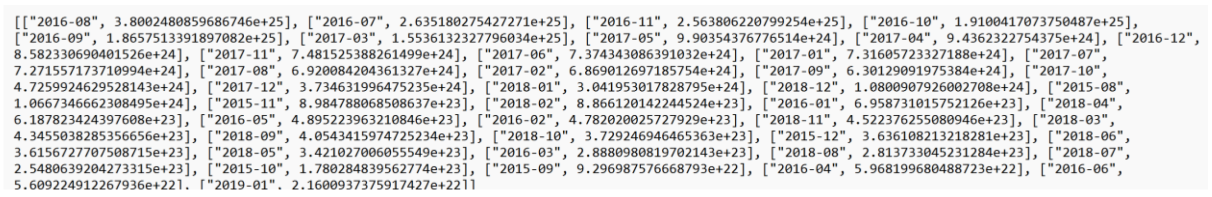
- To understand the correlation with Part B, we first need to get the data we generated in Part B.
- In part B, we got the data for the top 10 smart contracts by total Ether received. In this case, we are getting the values of ether received for all the contracts.
- To get this, we are extracting the timestamp and the value using the map method.
- Will then use reduce method to get summation for ether received for each Year-Month.
- This data will be sorted ascendingly on ether received value.
- Resulting data is saved in file: ‘/time_with_values_used.txt’

Code Snippet:



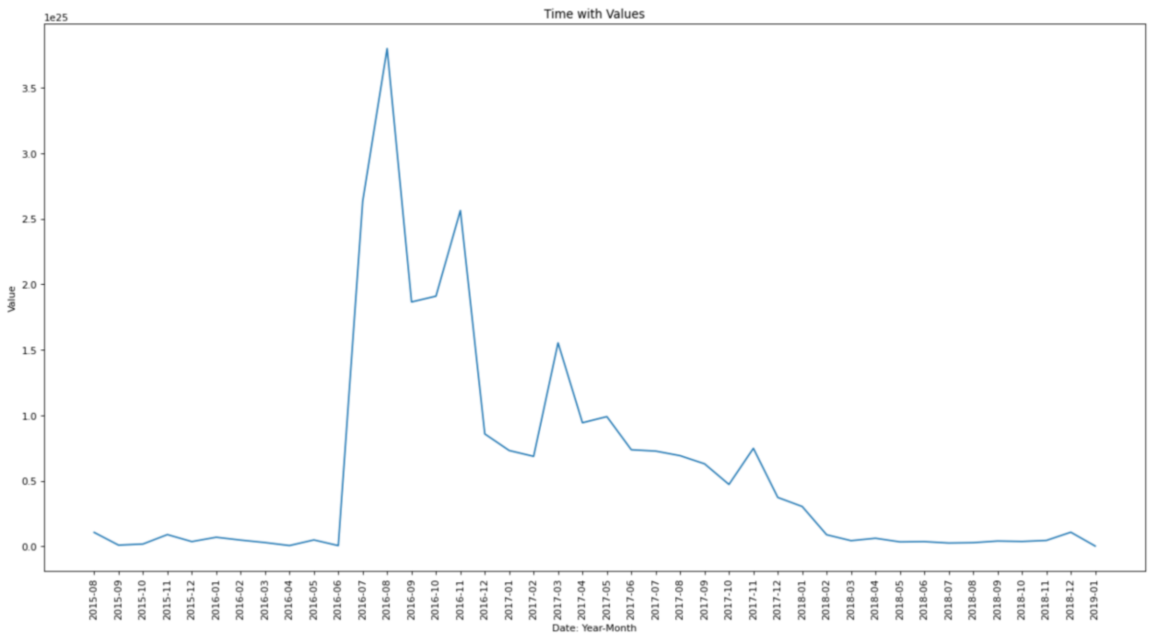
```
#(Time, value)
time_with_values_used = joined_results.map(lambda x: (x[1][0][2], x[1][0][3])).reduceByKey(operator.add).sortBy(lambda a: -a[1])
my_result_object = my_bucket_resource.Object(s3_bucket,'Test'  + '/time_with_values_used.txt')
my_result_object.put(Body=json.dumps(time_with_values_used.collect()))
```

Output:



```
[["2016-08", 3.8002480859686746e+25], ["2016-07", 2.635180275427271e+25], ["2016-11", 2.563806220799254e+25], ["2016-10", 1.9100417073750487e+25],
["2016-09", 1.8657513391897082e+25], ["2017-03", 1.5536132327796034e+25], ["2017-05", 9.90354376776514e+24], ["2017-04", 9.4362322754375e+24], ["2016-12",
8.582330690401526e+24], ["2017-11", 7.481525388261499e+24], ["2017-06", 7.374343086391032e+24], ["2017-01", 7.31605723327188e+24], ["2017-07",
7.271557173710994e+24], ["2017-08", 6.920084204361327e+24], ["2017-02", 6.869012697185754e+24], ["2017-09", 6.30129091975384e+24], ["2017-10",
4.725992462952814 3e+24], ["2017-12", 3.734631996475235e+24], ["2018-01", 3.041953017828795e+24], ["2018-12", 1.0800907926002708e+24], ["2015-08",
1.0667346662308495e+24], ["2015-11", 8.984788068508637e+23], ["2018-02", 8.866120142244524e+23], ["2016-01", 6.958731015752126e+23], ["2018-04",
6.187823424397608e+23], ["2016-05", 4.895223963210846e+23], ["2016-02", 4.782020025727929e+23], ["2018-11", 4.522376255080946e+23], ["2018-03",
4.3455038285356656e+23], ["2018-09", 4.0543415974725234e+23], ["2018-10", 3.729246946465363e+23], ["2015-12", 3.636108213218281e+23], ["2018-06",
3.6156727707508715e+23], ["2018-05", 3.421027006055549e+23], ["2016-03", 2.8880980819702143e+23], ["2018-08", 2.813733045231284e+23], ["2018-07",
2.5480639204273315e+23], ["2015-10", 1.780284839562774e+23], ["2015-09", 9.296987576668793e+22], ["2016-04", 5.968199680488723e+22], ["2016-06",
5.609224912267936e+22], ["2019-01", 2.1600937375917427e+22]]
```

Graph Plot:



Observation:

From the above plot, we can observe that, as time increases the ether received values also increase. That means we can infer the contracts getting complicated are **negatively correlated** with the ether value received.

D. Identify if the most popular contracts use more or less than the average gas_used.

- In order to identify if the most popular contracts use more or less than the average gas_used we need to get the most popular contracts and Average gas used.
- First, we will get the most popular contracts by fetching Address as a key and value, gas, and count as value. Such as, ((address, (value, gas, 1))) by using the Map method on the joined result we obtained earlier.

- With the help of reduce method, we will get the summation of all the values, gas and count for each address.
- In order to get the average gas used for each address we will further use the map method by dividing the gas by the count. (Address, (value, gas/count))
- We will sort these data with values.
- To get the top 10 contracts with the highest value first will use 'takeOrdered' on the basis of value.
- The result is saved in file: '/top10address_avg_gas_value.txt'.

Code Snippet:

```python
# (address, (value, gas, 1))
most_popular_contract_with_gas = joined_results.map(lambda l: (l[0],(l[1][0][3],l[1][0][1],1)))
most_popular_contract_with_gas_data = most_popular_contract_with_gas.reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1], x[2] + y[2]))
# (address, (value, gas/count)), sorted with value
average_gas_with_popular_contracts = most_popular_contract_with_gas_data.map(lambda x: (x[0],(x[1][0], x[1][1] / x[1][2]))).sortBy(lambda a: -a[1][0])
# Top 10 contracts with highest value first
top10_address_avg_gas_value = average_gas_with_popular_contracts.takeOrdered(10, key = lambda x: -x[1][0])
my_result_object = my_bucket_resource.Object(s3_bucket,'Test'  + '/top10address_avg_gas_value.txt')
my_result_object.put(Body=json.dumps(top10_address_avg_gas_value))
```

Output:

```
[["0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444", [8.415536369994497e+25, 83153.10930123563]], ["0x7727e5113d1d161373623e5f49fd568b4f543a9e",
[4.562712851291337e+25, 89826.72419423984]], ["0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef", [4.255298913641732e+25, 42553.82170235774]],
["0xbfc39b6f805a9e40e77291aff27aee3c96915bdd", [2.110419513809466e+25, 39999.61330880607]], ["0xe94b04a0fed112f3664e45adb2b8915693dd5ff3",
[1.554307763526529e+25, 136704.900966096]], ["0xabbb6bebfa05aa13e908eaa492bd7a8343760477", [1.071948594562895e+25, 101932.03479546004]],
["0x341e790174e3a4d35b65fdc067b6b5634a61caea", [8.379000751917755e+24, 198832.66666666666]], ["0x58ae42a38d6b33a1e31492b60465fa80da595755",
[2.9027091871057405e+24, 50212.80846597866]], ["0xc7c7f6660102e9a1fee1390df5c76ea5a5572ed3", [1.2380861145200458e+24, 33973.648482614575]],
["0xe28e72fcf78647adce1f1252f240bbfaebd63bcc", [1.172426432515822e+24, 150945.27918781727]]]
```

- To get the overall average gas used, we need to map the gas value only with its count.
- With the reducer method we will take a summation of all gas values and a summation of the count.
- To get the average gas, we need to divide the gas value by the count.
- This data is saved in file: '/total_average_gas_used.txt'

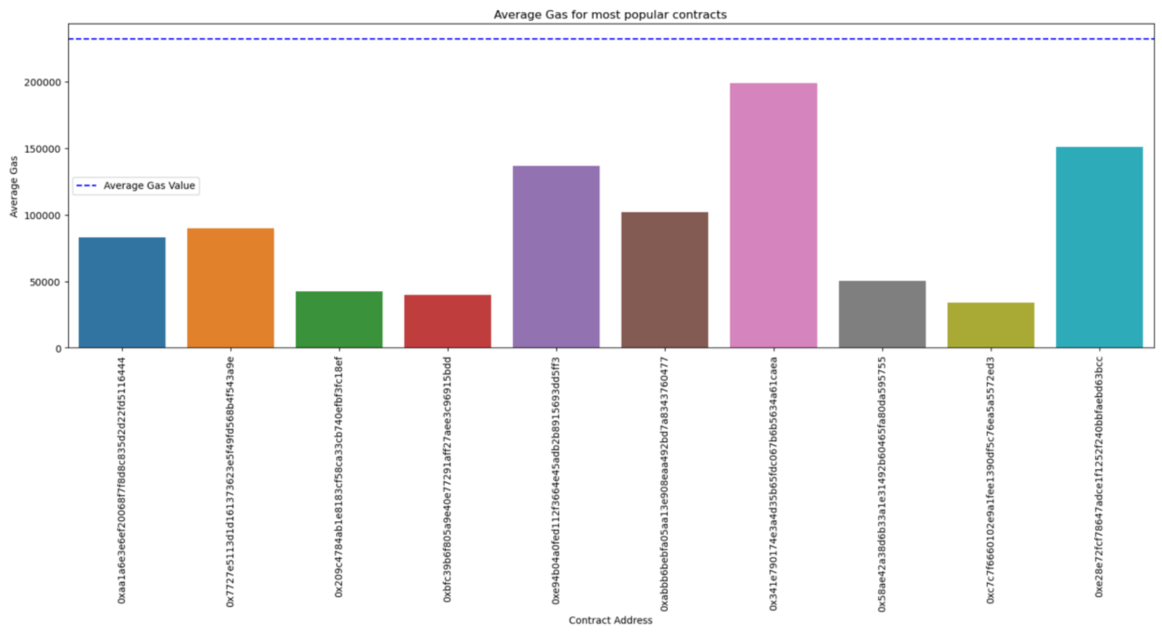
Code Snippet:

```python
#(Gas, overall avg)
average_gas = joined_results.map(lambda x: ('key', (x[1][0][1], 1)))
total_average_gas_used = average_gas.reduceByKey(lambda x, y: (x[0]+y[0], x[1]+y[1])).map(lambda x:(x[0], x[1][0]/x[1][1]))
my_result_object = my_bucket_resource.Object(s3_bucket,'Test'  + '/total_average_gas_used.txt')
my_result_object.put(Body=json.dumps(total_average_gas_used.collect()))
```

Output:

```
[["key", 231728.5239958902]]
```

Graph Plot:



Observation:

From the above plot, we can observe that the most popular contracts use less gas than the average gas_used. The blue dotted line shows the average gas used. All the top contracts use gas less than the average gas used.

2.Data Overhead:

The blocks table contains a lot of information that may not strictly be necessary for a functioning cryptocurrency e.g. logs_bloom, sha3_uncles, transactions_root, state_root, receipts_root. Analyse how much space would be saved if these columns were removed. Note that all of these values are hex_strings so you can assume that each character after the first two requires four bits (this is not the case in reality but it would be possible to implement it like this) as this will affect your calculations. (20%/40%)

Approach:

- To calculate the space saved by removing columns logs_bloom, sha3_uncles, transactions_root, state_root, receipts_root, we first need to get the space used by each of the columns.
- We have written a separate method to calculate the size of column data, (get_size).
- In this method, first we will calculate the length of column data using the ‘len’ function.
- As given in the question, we need to ignore the first 2 hex string. In order to do that, we need to subtract 2 from the length.
- As each character requires four bits, we will multiply this value by four.
- If the column data is empty i.e., length is 0, in that case, we don’t need to calculate the size. Hence, we have added the check to calculate size only when the column has data else, we can return 0.

Code Snippet:

```
def get_size(column_data):
    if len(column_data) > 0:
        size = (len(column_data)-2) * 4
        return size
    else:
        return 0
```

- This size will be then used in the map method to calculate the size of each column of data.
- The size of each column will be then summed up using the reduce method. The resulting data will have the total size used by each column.
- In order to get the total saved space, we need to add all these five values again.
- This can be done using the map method on this result.
- The total saved space is saved in file: ‘/saved_space.txt’.

Code Snippet:

```
    def good_line_for_blocks(line):
        try:
            fields = line.split(',')
            if len(fields)!=19:
                return False
            return True
        except:
            return False

    lines_contract = spark.sparkContext.textFile("s3a://" + s3_data_repository_bucket + "/ECS765/ethereum-parvulus/blocks.csv")
    clean_lines_contract=lines_contract.filter(good_line_for_blocks)

    # (logs_bloom, sha3_uncles, transactions_root, state_root, and receipts_root),
    contract_addr_data = clean_lines_contract.map(lambda a: ("Key", (get_size(a.split(',')[5]), get_size(a.split(',')[4]), get_size(a.split(',')[6]),
get_size(a.split(',')[7]), get_size(a.split(',')[8]))))

    result = contract_addr_data.reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1], x[2] + y[2], x[3] + y[3], x[4] + y[4]))
    final_result = result.map(lambda x: (x[0], (x[1][0] + x[1][1] + x[1][2] + x[1][3] + x[1][4])))

    my_result_object = my_bucket_resource.Object(s3_bucket,'Test'  + '/saved_space.txt')
    my_result_object.put(Body=json.dumps(final_result.collect()))
```

Output:

```
[["Key", 21504003276]]
```

The total bits size: 21504003276

Total size saved in GB: 2.6880004095 ~ **2.69GB**