ECS765P - BIG DATA PROCESSING

Analysis of Ethereum Transactions and Smart Contracts

NITISH KRISHNA SADHU

EC22098

220283937

PART – A:

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

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

Code:

# The following code filters for good transactions.

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

- The *good_line* function takes a line which is of string data type as input from the csv file.
- It splits the input line with a comma as the separator and gives a list called fields.
- The *if* clause checks if the length of the list is anything other than 15, if yes, it returns false.
- The next two lines tries to convert fields[11] and fields[7] to *int* data type, if they cannot, the code inside *except* gets executed which is returning false.
- If the conversion to *int* is successful, the function returns True.

# The following code counts the Number of transactions per month.

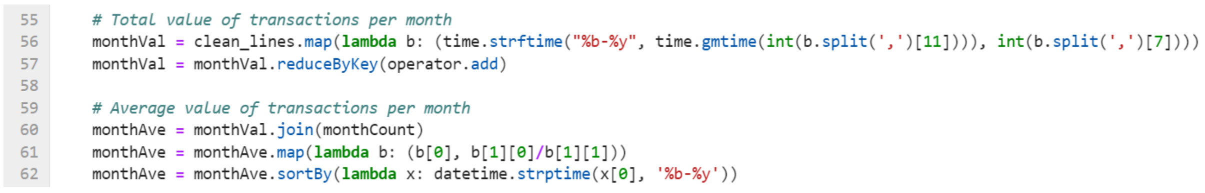
```python
lines = spark.sparkContext.textFile("s3a://" + s3_data_repository_bucket + "/ECS765/ethereum-parvulus/transactions.csv")
clean_lines = lines.filter(good_line)
```

- The above code reads data from the transactions.csv file to the lines RDD.
- We use filter to apply the *good_lines* function to the lines, the filtered output is assigned to the clean_lines RDD.

```python
# Number of transactions per month
monthCount = clean_lines.map(lambda b: (time.strftime("%b-%y", time.gmtime(int(b.split(',')[11]))), 1))
monthCount = monthCount.reduceByKey(operator.add)
monthCount = monthCount.sortBy(lambda x: datetime.strptime(x[0], '%b-%y'))
```

- The first line of the above code is a map, which is applied on the clean_lines and outputs a key-value pair, where the key is the date in the month-year format using the timestamp and the values is set as 1.
- The second line is a reducer which reduces by adding the value in the key-value pair based on the key.
- The third line sorts the key-value pairs based on the key.

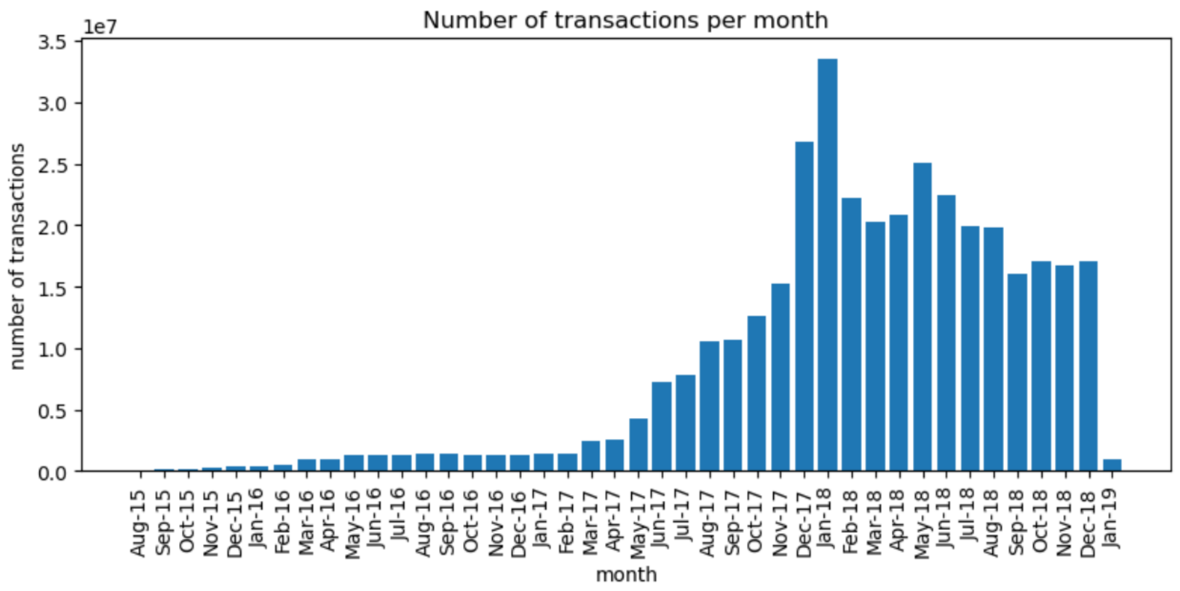
# The following code calculates the Average value of transactions per month



```python
# Total value of transactions per month
monthVal = clean_lines.map(lambda b: (time.strftime("%b-%y", time.gmtime(int(b.split(',')[11]))), int(b.split(',')[7])))
monthVal = monthVal.reduceByKey(operator.add)

# Average value of transactions per month
monthAve = monthVal.join(monthCount)
monthAve = monthAve.map(lambda b: (b[0], b[1][0]/b[1][1]))
monthAve = monthAve.sortBy(lambda x: datetime.strptime(x[0], '%b-%y'))
```

- The first line is a mapper whose output is a key-value pair where the key is the date in the format month-year calculated from timestamp, value is the value of the transaction.
- The second line is a reducer which reduces by adding the values based on the key.
- The next line joins the monthVal RDD and the monthCount RDD.
- Then comes the mapper which outputs a key-value pair where the key is a date in the format month-year, and the value is calculated by dividing the sum of the value of all the transactions divided by the total number of transactions.
- The next line sorts the RDD monthAve based on the key.

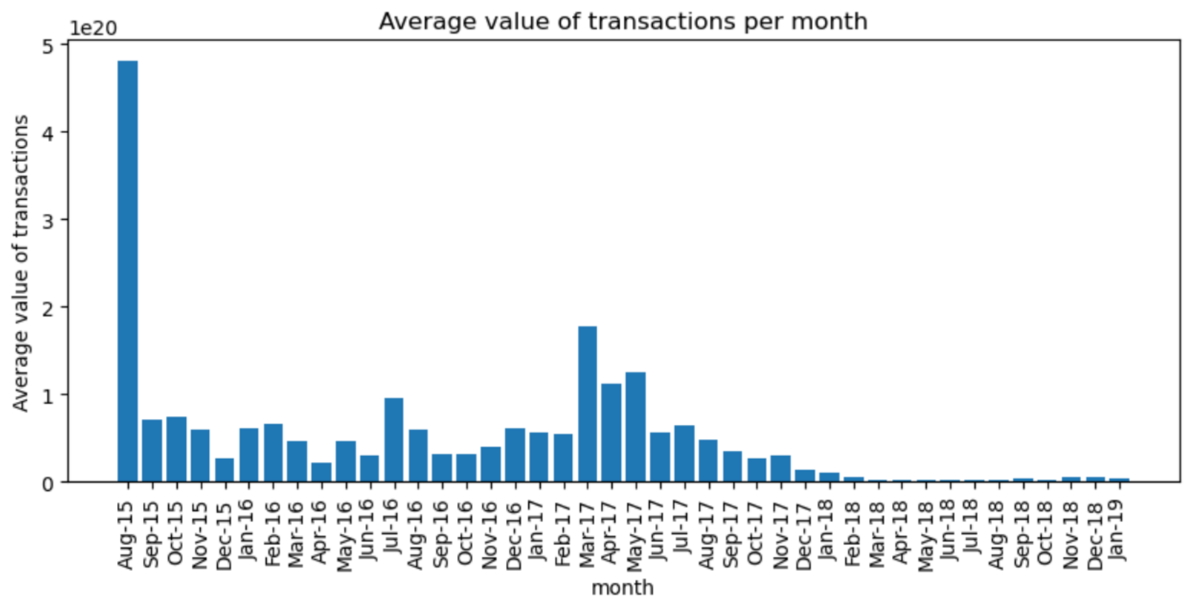
# Number of transactions per month:



Explanation:

The number of transactions remain almost consistent from August ’15 to February ’17, after which it raises almost linearly and reaches its peak in January ’18, then it decreases gradually till December ’18 while attaining a local peak in May ’18.

# Average value of transactions per month:



The Average value of transactions per month achieves its highest value in August ’15. It drastically drops in September ’15 an remains almost consistent till February ’17 and attains a local maximum in March ’17 and then gradually decreases till February ’18 after which it remains consistent till December ’18.

Part – B:

25

Aim: To evaluate the top 10 smart contracts by total Ether received.

Code:

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

- The above block of code is the function called good_trans which filters for the complete transaction records in the table.
- In the try block:
    - In the first line we split the string in to a list with ‘ , ’ as the separator.
    - The second line has the if statement which returns *false* if the length of the list is not equal to 15.
    - The next line tries to convert the element at index ‘7’ of the list into float. If the conversion is successful, the code returns *true* else it breaks out of the try and goes into the except block.

- In the except block:
    - The code returns *false*.

```python
lines = spark.sparkContext.textFile("s3a://" + s3_data_repository_bucket + "/ECS765/ethereum-parvulus/transactions.csv")
clean_lines=lines.filter(good_trans)
transValue = clean_lines.map(lambda t: (t.split(",")[6], t.split(",")[7]))
print(transValue.take(1))
```

- The first line of the above code block reads the *transactions.csv* file from the repository.
- The second line filters for the good transactions by using the *good_trans* function.
- The third line is a mapper which outputs key-value pairs with to_address as the key and the value transferred in Wei as the value.
- The final line is a print statement which prints only one value from transValue RDD. This line consists of *take* which is an action, it triggers the execution of the transformations on the transValue RDD that comes before it.

```python
    contracts = spark.sparkContext.textFile("s3a://" + s3_data_repository_bucket + "/ECS765/ethereum-parvulus/contracts.csv")

    bytecode = contracts.map(lambda b: (b.split(",")[0], b.split(",")[1]))
    print(bytecode.take(1))
```

- The first line of the above code block reads the *contracts.csv* file from the repository.
- The second line is a mapper which outputs key-value pairs with address of the contract as the key and the bytecode of the contract as the value.
- The final line is a print statement which prints only one value from bytecode RDD. This line consists of *take* which is an action, it triggers the execution of the transformations on the bytecode RDD that comes before it.

```python
contract_wei = transValue.join(bytecode)

contract_wei = contract_wei.map(lambda w: (w[1][1], int(w[1][0])))
contract_wei = contract_wei.reduceByKey(operator.add)

top10 = contract_wei.takeOrdered(10, lambda x : -x[1])

```

- The first line of the above code block joins the transValue and bytecode RDDs based on the key which is the address of the contract.
- The second line is a mapper which outputs key-value pairs with the bytecode of the contract as the key and the value of the ether received in Wei as the value.
- The next line is a reducer which adds the value based on the key.
- The takeOrdered function gives the top10 contracts with the most ether received

Note: *The bytecode is too large to include in the report, so the address of the contract is included below, the actual output is provided as a separate text file.*

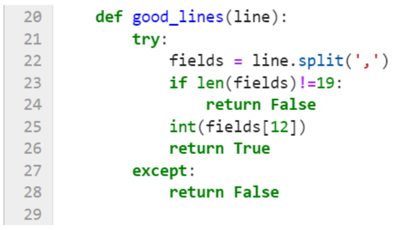
The address of the top 10 contracts that received the most ether.

["0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444", 84155363699941767867374641],
["0x7727e5113d1d161373623e5f49fd568b4f543a9e", 45627128512915344587749920],
["0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef", 42552989136413198919298969],
["0xfa52274dd61e1643d2205169732f29114bc240b3", 40546128459947291326220872],
["0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8", 24543161734499779571163970],
["0xbfc39b6f805a9e40e77291aff27aee3c96915bdd", 21104195138093660050000000],
["0xe94b04a0fed112f3664e45adb2b8915693dd5ff3", 15543077635263742254719409],
["0xbb9bc244d798123fde783fcc1c72d3bb8c189413", 11983608729102893846818681],
["0xabbb6bebfa05aa13e908eaa492bd7a8343760477", 10719485945628946136524680],
["0x341e790174e3a4d35b65fdc067b6b5634a61caea", 8379000751917755624057500]

Part-C:

Aim: To evaluate the top 10 miners by the size of the blocks mined.

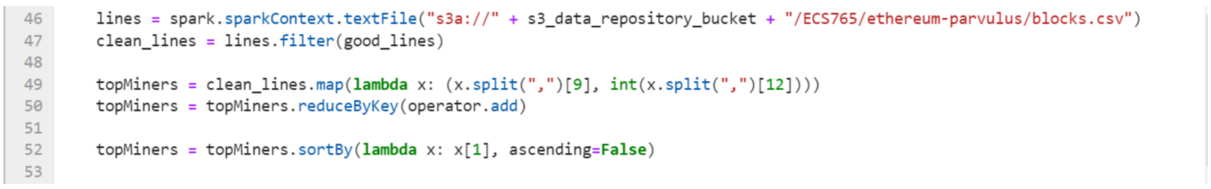
Code:



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

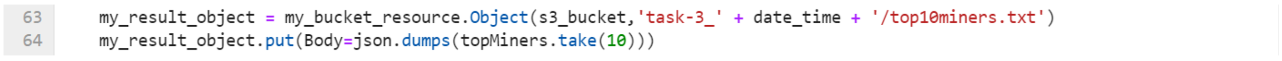
- The above block of code is the function called good_lines which filters for the complete blocks records in the table.
- In the try block:
    - In the first line we split the string in to a list with ' , ' as the separator.
    - The second line has the if statement which returns *false* if the length of the list is not equal to 19.
    - The next line tries to convert the element at index '12' of the list into int. If the conversion is successful, the code returns *true* else it breaks out of the try and goes into the except block.

- In the except block:
    - The code returns *false*.



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

topMiners = clean_lines.map(lambda x: (x.split(",")[9], int(x.split(",")[12])))
topMiners = topMiners.reduceByKey(operator.add)

topMiners = topMiners.sortBy(lambda x: x[1], ascending=False)

```

- The first line of the code reads the *blocks.csv* from the repository.
- The next line filters for the complete records of the blocks using the *good_lines* function.
- The third line is a mapper which outputs a key-value pair where the key is miner, and the value is the size of the block.
- The next line is a reducer which adds the values based on the key.
- The final line is a sortBy which sort the topMiners RDD in descending order based on the value.



```
my_result_object = my_bucket_resource.Object(s3_bucket,'task-3_' + date_time + '/top10miners.txt')
my_result_object.put(Body=json.dumps(topMiners.take(10)))
```

- The second line has take action, which takes the first 10 items of the RDD and writes them to the output.

<u>Output</u>:

Top 10 miners by the size of the blocks mined.

```
["0xea674fdde714fd979de3edf0f56aa9716b898ec8", 17453393724],
["0x829bd824b016326a401d083b33d092293333a830", 12310472526],
["0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c", 8825710065],
["0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5", 8451574409],
["0xb2930b35844a230f00e51431acae96fe543a0347", 6614130661],
["0x2a65aca4d5fc5b5c859090a6c34d164135398226", 3173096011],
["0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb", 1152847020],
["0x4bb96091ee9d802ed039c4d1a5f6216f90f81b01", 1134151226],
["0x1e9939daaad6924ad004c2560e90804164900341", 1080436358],
["0x61c808d82a3ac53231750dadc13c777b59310bd9", 692942577]
```

Part-D:

1.Gas Guzzlers: 17

Part-1:

Aim: To find how the gas price has changed over time.

Code:

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

- The above block of code is the function called good_lines which filters for the complete transaction records in the table.
- In the try block:
    - In the first line we split the string in to a list with ' , ' as the separator.
    - The second line has the if statement which returns *false* if the length of the list is not equal to 15.
    - The next three lines tries to convert the item at indices '8','9','11' of the list into *int*. If the conversion is successful, the code returns *true* else it breaks out of the try and goes into the except block.

- In the except block:
    - The code returns *false*.

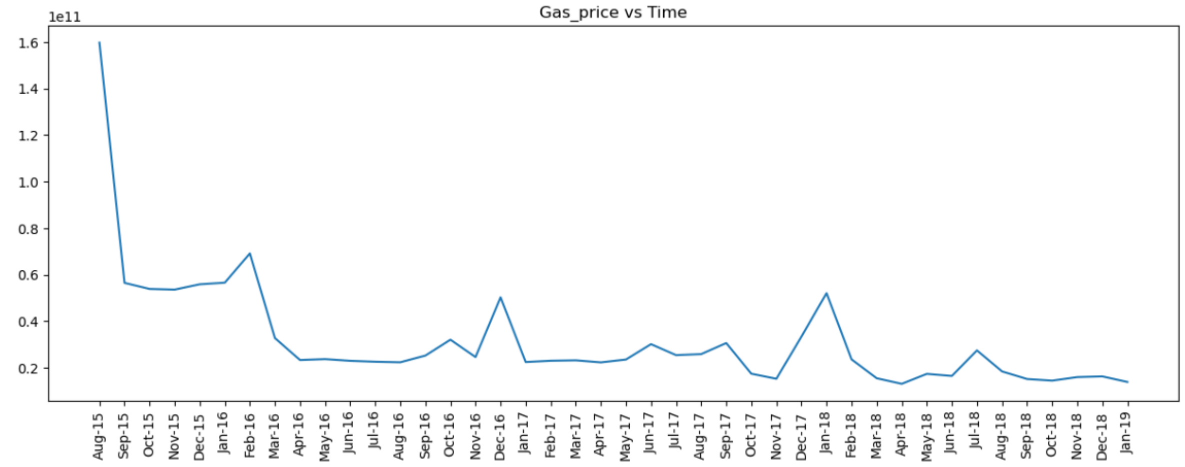
```python
    transactions = spark.sparkContext.textFile("s3a://" + s3_data_repository_bucket + "/ECS765/ethereum-parvulus/transactions.csv")
    clean_transactions = transactions.filter(good_transaction)
```

- In the code above, the first line reads the transactions.csv from the repository
- The second line filters for the good transactions by using the *good_transaction* function.

```python
    # Change of gas required over time
    gas = clean_transactions.map(lambda g: (time.strftime("%b-%y",time.gmtime(int(g.split(",")[11]))), (int(g.split(",")[9]), 1)))
    gas = gas.reduceByKey(lambda a,b: (a[0]+b[0], a[1]+b[1]))
    gas = gas.map(lambda g: (g[0], g[1][0]/g[1][1]))

    gas = gas.sortBy(lambda g : datetime.strptime(g[0], '%b-%y'))

```

- The first line of the above code is a mapper which gives out the output as a key-value pair, with the key value being the month-year and the value being a tuple whose first element is the gas, and the second element is number 1, which is used to count the number of transactions in the reducer step.
- The next line is a reducer which adds the values based on the key and the index.
- The third line is a mapper which outputs a key-value pair where the key is the month-year and the value is the average gas-price per month obtained by dividing the total gas and the number of transactions obtained from the previous step.
- The final line of the code is a sortBy which sorts on the basis of the key i.e., the date.

# Change of Gas-Price over time:



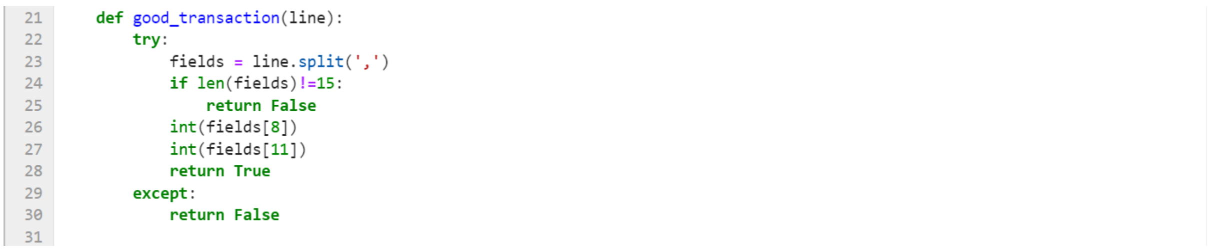
Explanation:

It can be seen from the graph that the gas-price decreases over time. Gas-price starts at 1.6e11 in Aug-15 to around 0.1e11 in Jan-15 with local peaks in Feb-16, Dec-16 and Jan-18.

Part-2:

Aim: To see if the contracts have become less or more complicated over time i.e., requiring more gas or less.

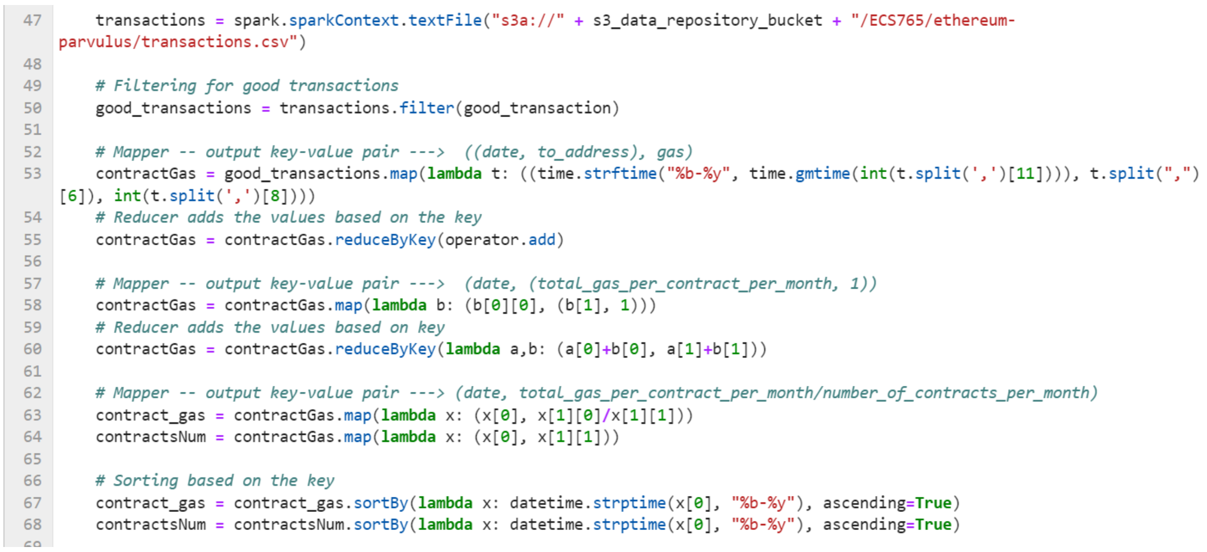
Code:



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

- The above block of code is the function called good_lines which filters for the complete transaction records in the table.
- In the try block:
  - In the first line we split the string in to a list with ‘ , ’ as the separator.
  - The second line has the if statement which returns *false* if the length of the list is not equal to 15.
  - The next three lines tries to convert the item at indices ‘8’ ,’11’ of the list into *int*. If the conversion is successful, the code returns *true* else it breaks out of the try and goes into the except block.

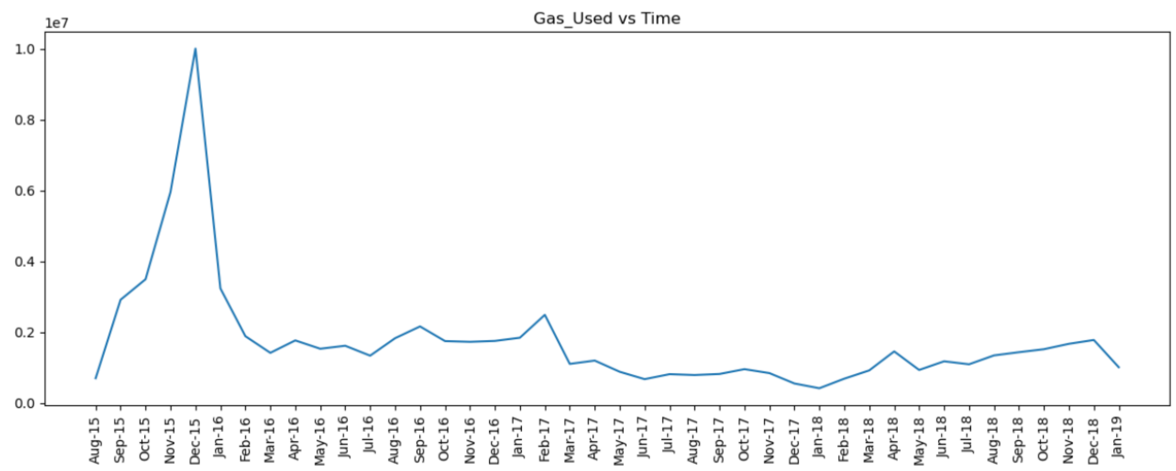
- In the except block:
  - The code returns *false*.



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

    # Filtering for good transactions
    good_transactions = transactions.filter(good_transaction)

    # Mapper -- output key-value pair --->  ((date, to_address), gas)
    contractGas = good_transactions.map(lambda t: ((time.strftime("%b-%y", time.gmtime(int(t.split(',')[11]))), t.split(",")[6]), int(t.split(',')[8])))
    # Reducer adds the values based on the key
    contractGas = contractGas.reduceByKey(operator.add)

    # Mapper -- output key-value pair --->  (date, (total_gas_per_contract_per_month, 1))
    contractGas = contractGas.map(lambda b: (b[0][0], (b[1], 1)))
    # Reducer adds the values based on key
    contractGas = contractGas.reduceByKey(lambda a,b: (a[0]+b[0], a[1]+b[1]))

    # Mapper -- output key-value pair ---> (date, total_gas_per_contract_per_month/number_of_contracts_per_month)
    contract_gas = contractGas.map(lambda x: (x[0], x[1][0]/x[1][1]))
    contractsNum = contractGas.map(lambda x: (x[0], x[1][1]))

    # Sorting based on the key
    contract_gas = contract_gas.sortBy(lambda x: datetime.strptime(x[0], "%b-%y"), ascending=True)
    contractsNum = contractsNum.sortBy(lambda x: datetime.strptime(x[0], "%b-%y"), ascending=True)
```

- The first line of the code reads the transactions.csv file from the repository
- The second line filters for complete records of the transactions using the good_transaction function.
- The next line is a mapper which outputs a key-value pair with the (month-year, to_address) as key and the gas provided by the sender as the value.
- The next line reduces by adding the values based on key.
- The next line is a mapper which outputs the key-value pair (month-year, (sum of gas, 1)) which is reduced in the next line based on the key.
- The next line is a mapper which computes the average value of the gas transferred based on the key followed by a mapper which outputs the number of contracts in a month.
- The final two lines sorts both outputs (average gas value and number of contracts in a month) based on the key.

# Average gas used per contract per month overtime



Explanation:

Gas used has increased from Aug-15 to Dec-15 when the usage reached its peak then it plummeted and stabilized in Feb-16, where it remained between 0 and 0.2e7 till Jan-19.

Did contracts become more or less complicated?

From the above graphs (Gas_Used vs Time and Gas_Price vs Time) we can see that the Gas used has peaked on Dec-15 and decreased and remained stable from Feb-16 till Jan-19. Coming to the gas_price, it slightly decreased over time, from this we can infer that the contracts required less gas over time which resulted in slight decrease in gas-price due to low demand(which can be seen from gas_usage graph).

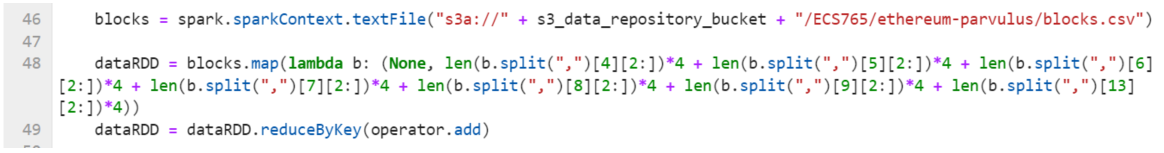
The top 10 contracts with most gas usage used way more than the average gas used.

15

Data Overhead:

Aim: To analyse how much space would be saved if unnecessary columns were removed.

Code:



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

    dataRDD = blocks.map(lambda b: (None, len(b.split(",")[4][2:])*4 + len(b.split(",")[5][2:])*4 + len(b.split(",")[6]
[2:])*4 + len(b.split(",")[7][2:])*4 + len(b.split(",")[8][2:])*4 + len(b.split(",")[9][2:])*4 + len(b.split(",")[13]
[2:])*4))
    dataRDD = dataRDD.reduceByKey(operator.add)
```

- The first step of the above code block reads the *blocks.csv* from the repository.
- The second step is a mapper which outputs key-value pairs where the key is **None,** the value is the sum of the lengths of all the items in unnecessary columns where the length is calculated excluding the first two characters as they are not part of the hex

numbers but denotes that the number is hex. The lengths are multiplied by 4(each character occupies 4 bytes), all these computed values are added to give the value in the key-value pair.
- The next line is a reducer which adds the values based on the keys.

Output:

```
[[null, 23542472248]]
```

- A total of *23542472248* bytes can be saved by removing the unnecessary columns.