Michael Lovell

160552382

ECS640U – Coursework

# Bitcoin Analysis using Big Data jobs

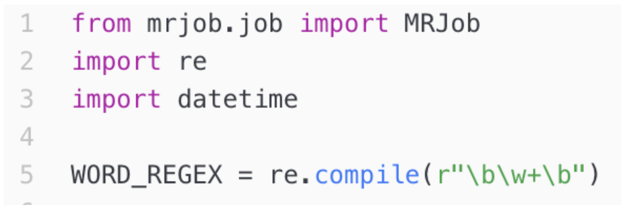
Submitted before 3/12/2018

# Part A – Time Analysis

For the initial part of the coursework, I decided to use a simple Map/Reduce MRJob to summarise a count as value for how many times each month/year set as the key appears. This would be a simple way to see the distribution and the popularity of bitcoin transactions over the period of 2009 – 2014. As the transactions file is a fairly large data set and a simple count is required, map/reduce is more than sufficient to process this, especially with its support for key/value pairs. This could have been done using Spark, in a way which emulates Map/Reduce and its key/value pairs, but I have decided to use a mixture of Map/Reduce and Spark throughout this coursework for variety.

## Job Setup

The initial job set up relies on splitting the Transactions CSV file into lines to be parallelised and used by the individual mappers. Before doing so, I have imported the libraries needed for the job to run properly, especially the datetime library, which is needed to format the time format from Unix EPOCH time to a more readable format. The same regular expression has been used from the labs before running the map methods.



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

WORD_REGEX = re.compile(r"\b\w+\b")
```

Figure 1: Initial library imports and regular expression needed

## Mapper Method

For each 128mb block of lines within the Transaction file, a mapper will be created which will further split the line into individual fields of tx_hash, blockhash, time, tx_in_count and tx_out_count. The important field being time, which would be used to find the month and year for each transaction occurred within the bitcoin blockchain. It is needed to format the time from Unix time (UNIX Epoch time) into a more understandable format of month and year.

The time field would become the key, where a count of how many times this month and year appear in the transaction file being the value. This key/value pair will be sent to the reducer via the combiner

```
6
7   class partA(MRJob):
8
9       def mapper(self, _, line):
10          fields = line.split(",")
11          try:
12              if(len(fields)==5):
13                  time_epoch = int(fields[2])
14                  date = datetime.datetime.fromtimestamp(time_epoch)
15                  monthYear = date.strftime("%Y-%m")
16                  yield (monthYear, 1)
17              else:
18                  yield ("malformed", 1)
19          except:
20              yield ("malformed", 1)
```

Figure 2: Mapper code, yielding month/year as key and a value for counting

As shown in figure 2, at line 10 the lines are split into fields with the length of fields for each line to be tested on line 12 to see if there are any malformed lines of data. Line 13 shows the field 2 of the array which is the time in Unix epoch time, being assigned to a variable which is then calculated into a date on line 14. This date is calculated from the standard UNIX date of 01-01-1970. Each date has the month and year extracted from it as shown on line 15.

This now allows for the month and year string combination to be yielded as a key, with a value to allow for a count of how many times that month/year combination appears in the transaction file.

Malformed is yielded at lines 18 and 19 if the data is not as expected.

## Combiner Method

As the transaction file has lots of data within it, to make the job more efficient and generally faster, a combiner is needed to be implemented.

The combiner I have implemented exists to summarise the count of values for each key from its specific Mapper, to save work on the reducer receiving large amounts of data via the shuffle and sort process. Once the reducer receives the key based on its hashed value, its counts will be easier to combine compared to the reducer receiving individual counts without a combiner method in-between.

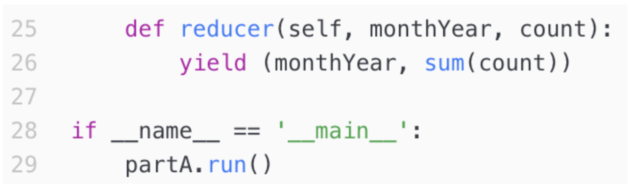
```

    def combiner(self, monthYear, count):
        yield(monthYear, sum(count))
```

Figure 3: Combiner used for efficiency and reducing job time

## Reducer Method

The reducer receives values based on the hash of the key sent from the Mapper/Combiner based on the month and year format of the original time field. Its job is to create a simple sum of all values for each month/year pair and yield the total count of transactions for that period. This data would then allow for a chart to be created showing the trend of bitcoin transactions over the period of month and year between 2009 and 2014. The main method is also shown.



```
    def reducer(self, monthYear, count):
        yield (monthYear, sum(count))

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

Figure 4: Reducer method to yield to output file plus main method

## Results

After running the job and yielding to an output file, I have sorted the results by date and plotted the chart below. It shows a common trend of transaction growth over the period of 5 years observed. This is understandable as Bitcoin has become more popular over time.

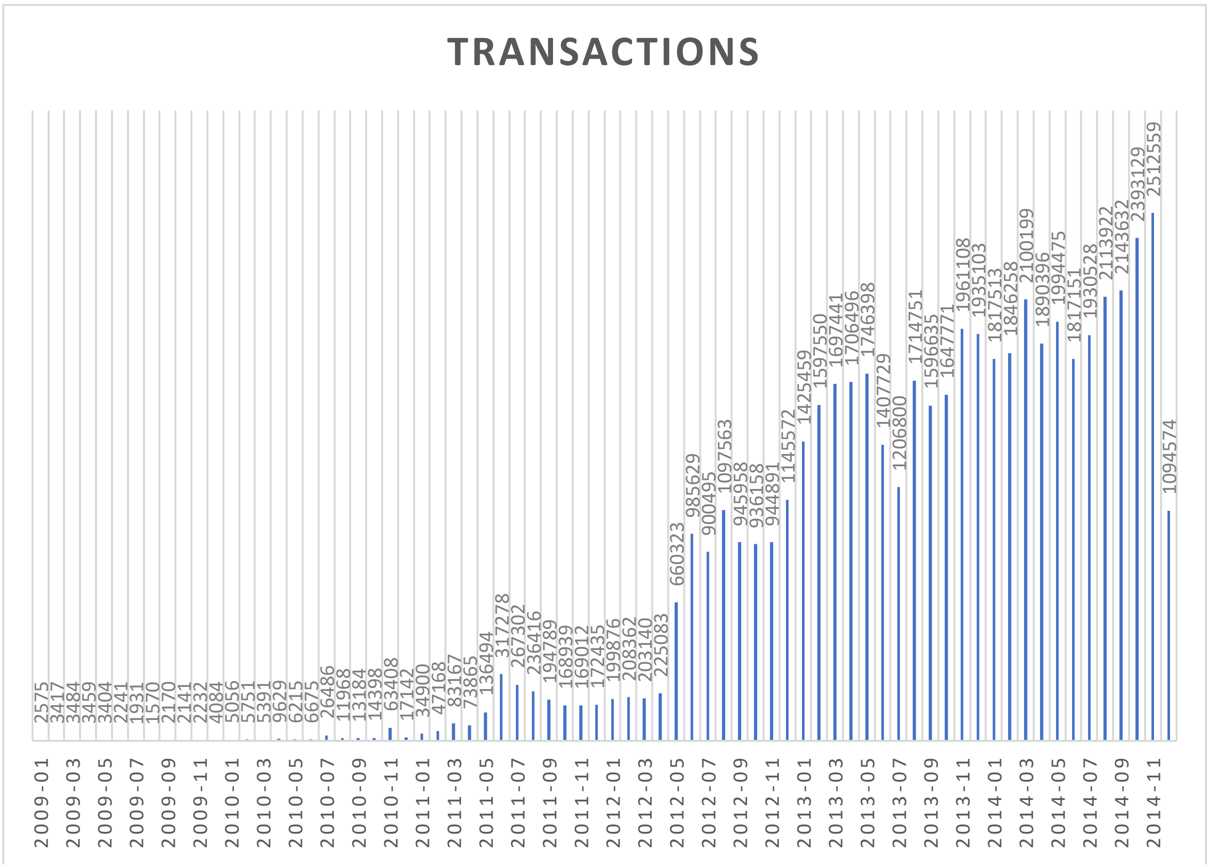


Table 1: Histogram of transaction amounts for month and year

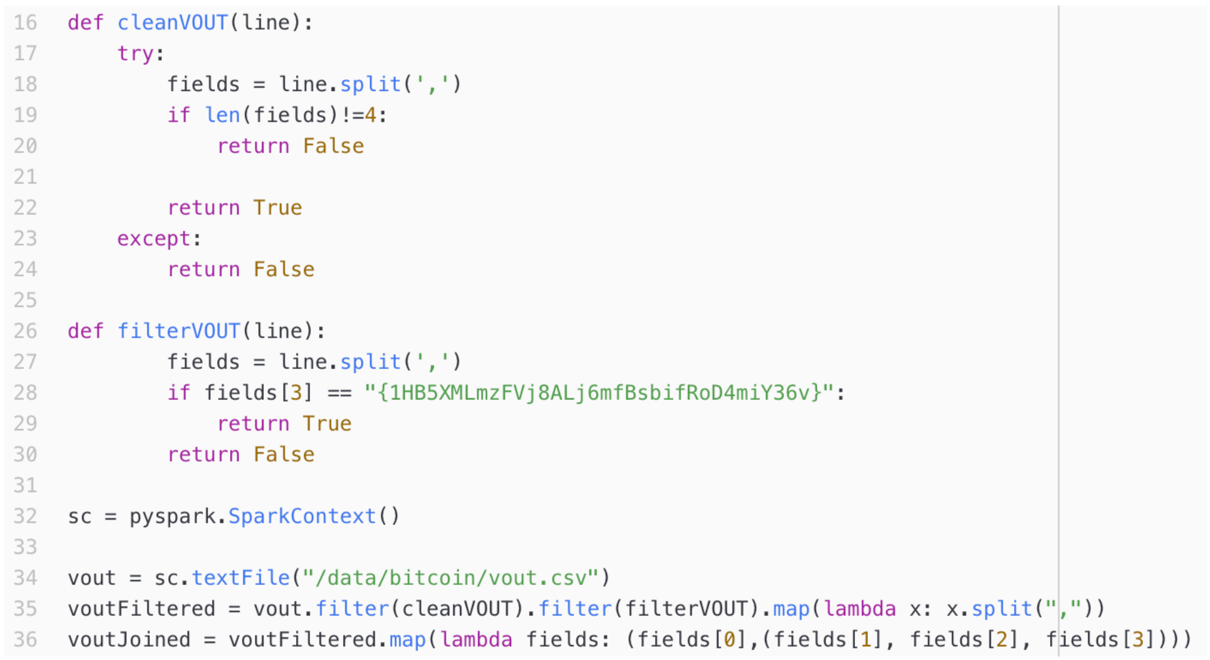
## Part B – Top Ten Donors

To find the top ten donors to the WikiLeaks bitcoin wallet address, I have decided to use Spark, for its handling of iterative algorithms and it's handling and reuse of data in memory. This type of job would be very inefficient to have completed within map/reduce, having to save each job for each iteration to HDFS then starting up a new job with the previous ones outputted data.

This part handles filtering and multiple joins when handling both the vout.csv and vin.csv data files. Multiple transformation and RDD's are created to handle this task, as explained in more detail below;

### Initial filtering and join

The initial transformation and RDD would be to filter the vout.csv data file. This would be filtered so that only the WikiLeaks bitcoin wallet data would be created into a new RDD to be joined with the vin.csv data set.



```
16  def cleanVOUT(line):
17      try:
18          fields = line.split(',')
19          if len(fields)!=4:
20              return False
21
22          return True
23      except:
24          return False
25
26  def filterVOUT(line):
27          fields = line.split(',')
28          if fields[3] == "{1HB5XMLmzFVj8ALj6mfBsbifRoD4miY36v}":
29              return True
30          return False
31
32  sc = pyspark.SparkContext()
33
34  vout = sc.textFile("/data/bitcoin/vout.csv")
35  voutFiltered = vout.filter(cleanVOUT).filter(filterVOUT).map(lambda x: x.split(","))
36  voutJoined = voutFiltered.map(lambda fields: (fields[0],(fields[1], fields[2], fields[3])))
```

Figure 5: Cleaning, filtering and mapping vout.csv file

Before filtering the vout.csv data set, I ran a function which would clean it. This would check whether any malformed lines existed, not equal to the four fields expected. This would work by splitting the line given and simply looking at the length of the field array created returning a Boolean of true or false depending on meeting this condition. This is shown at line 16-24 for cleaning and the call to the function at line 35 after importing the file as shown at line 34.

Once the dataset is cleaned, it can be filtered using another function which checks whether the third field is equal to the desired WikiLeaks bitcoin wallet address, this condenses the dataset significantly for it to be joined with the vin.csv data. Once both of these have been run on the dataset, it can be mapped with the four fields it contains. The filter function is shown at line 26 and called at line 35 after the initial cleaning filter has run. The filtered vout.csv file is mapped on line 36 ready to be joined.

The vin.csv data set is also ran through a similar cleaning function checking for its field length to be equal to three this is shown at line 6-14 and called from line 39. Anything other than this, is filtered out due to its malformed length. The data is then mapped using its three fields to be joined with the vout.csv file, this is shown at line 40 for the mapping and line 42 for the join.

```
38  vin = sc.textFile("/data/bitcoin/vin.csv")
39  vinFiltered = vin.filter(cleanVIN).map(lambda y: y.split(","))
40  vinJoined = vinFiltered.map(lambda fields: (fields[0],(fields[1], fields[2])))
41
42  firstJoin = voutJoined.join(vinJoined)
```

Figure 6: Importing, filtering, mapping and joining vin.csv file

```
6   def cleanVIN(line):
7       try:
8           fields = line.split(',')
9           if len(fields)!=3:
10              return False
11
12          return True
13      except:
14          return False
```

Figure 7: Cleaning vin.csv malformed lines

## Additional filtering and join

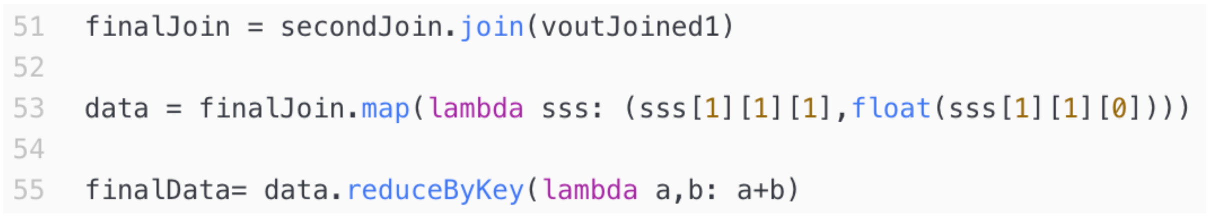
After the first filter and join is complete, another cleaning filter of vout.csv is done with it mapped again but this time without the WikiLeaks filtering applied. This is shown at line 45, with the data mapped by the four fields on line 46. The data for the additional join is shown at line 48, being mapped with the tuple data created. The code on line 48 & 49 was originally one line but split over two for an easier and more readable screenshot.

```
45  voutFiltered1 = vout.filter(cleanVOUT).map(lambda x: x.split(","))
46  voutJoined1 = voutFiltered1.map(lambda fields: ((fields[0], fields[2]), (fields[1], fields[3])))
47
48  secondJoin = firstJoin.map(lambda secondJoin: ((secondJoin[1][1][0], secondJoin[1][1][1]),
49  (secondJoin[0], secondJoin[1][0][0], secondJoin[1][0][1], secondJoin[1][0][2])))
```

Figure 8: Calling the vin clean filter, mapping by fields and mapping again for another join

The new join is shown on line 51, combining the data from the first join with the data from line 46.

Once the data is joined it is mapped again as shown at line 53 with the reduceByKey transformation shown at line 55. The reduce by key yielding the publicKey (wallet) and value.

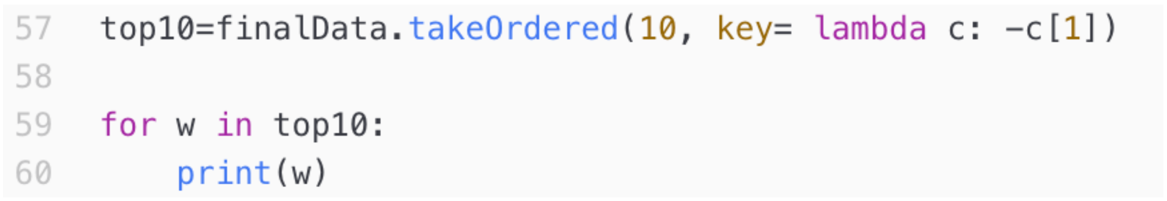


```
51  finalJoin = secondJoin.join(voutJoined1)
52
53  data = finalJoin.map(lambda sss: (sss[1][1][1],float(sss[1][1][0])))
54
55  finalData= data.reduceByKey(lambda a,b: a+b)
```

Figure 9: Second join at line 51, joined data being mapped at line 53 and reduced by key at line 55

## Top 10 sorted donations

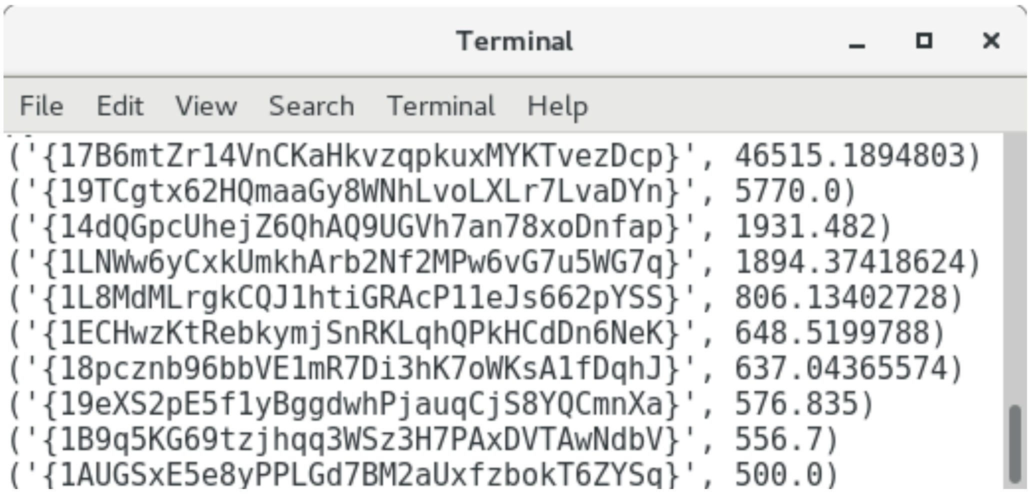
The data is then sorted by the top 10 donation amounts in bitcoin, which is then printed to terminal in descending order.



```
57  top10=finalData.takeOrdered(10, key= lambda c: -c[1])
58
59  for w in top10:
60      print(w)
```

Figure 10: Sorting the data into top 10 on line 57, printing to terminal at line 60

The final sorted amount by top donation is shown below in figure 11. At the time the transfer was made of the top donation (13/09/2012), one bitcoin was the equivalent of $11.20, which equated to £0.6192. The value of 46,515 bitcoins at the time of the transaction was made was £322,583. In todays value, this would equate to £150,337,393 which is an increase of 452x since the original donation was made.



```
('{17B6mtZr14VnCKaHkvzqpkuxMYKTvezDcp}', 46515.1894803)
('{19TCgtx62HQmaaGy8WNhLvoLXLr7LvaDYn}', 5770.0)
('{14dQGpcUhejZ6QhAQ9UGVh7an78xoDnfap}', 1931.482)
('{1LNWw6yCxkUmkhArb2Nf2MPw6vG7u5WG7q}', 1894.37418624)
('{1L8MdMLrgkCQJ1htiGRAcP11eJs662pYSS}', 806.13402728)
('{1ECHwzKtRebkymjSnRKLqhQPkHCdDn6NeK}', 648.5199788)
('{18pcznb96bbVE1mR7Di3hK7oWKsA1fDqhJ}', 637.04365574)
('{19eXS2pE5f1yBggdwhPjauqCjS8YQCmnXa}', 576.835)
('{1B9q5KG69tzjhqq3WSz3H7PAxDVTAwNdbV}', 556.7)
('{1AUGSxE5e8yPPLGd7BM2aUxfzbokT6ZYSq}', 500.0)
```

Figure 11: Top ten donations to the WikiLeaks wallet address