CS5234 – Final Project Report

Question 1 (60%)

Pick one of the functions among those you implemented for the final project and answer the following questions. If you worked on the project jointly with another student, you and your partner must choose different functions to discuss.

Q1. Include the Python code for your chosen function, and briefly describe its implementation highlighting any challenges and pitfalls you had to deal with. (20%).

Ans.

Python Code

```
def convert_to_weighted_network(rdd, drange=None):
    if (drange != None):
        rdd = rdd.filter(lambda x : (x[2] > drange[0]) and (x[2] < drange[1]))
    rdd10 = rdd.map(lambda x : ((x[0], x[1]), x[2]))
    rdd11 = rdd10.groupByKey().mapValues(len)
    rdd12 = rdd11.map(lambda x : (x[0][0], x[0][1], x[1]))
    return rdd12
```

The Problem statement for my chosen function is as follows :

Write a function convert_to_weighted_network() that takes one required argument rdd, which is an RDD that complies with the output format of the function extract_email_network() specified in Question 1, and one optional argument drange, which is a pair (d1, d2) of datetime objects with a default value

of None. The function returns an RDD consisting of distinct triples (o, d, w) such that all of the following constraints hold:

• (o, d, t) is an element of the input RDD for some timestamp t;

• if drange is not None, then w is the number of edges (o', d', t) in the input RDD such that (o ' , d') = (o, d) and drange[0] ≤ t ≤ drange[1];

• if drange is None, then w is the number of edges (o', d', t) in the input RDD such that (o ' , d') = (o, d).

Note that to make the datetime components of drange comparable to the datetime objects stored in the input RDD, each of them must be instantiated by providing timezone.utc as the value of the tzinfo parameter to its constructor. E.g., datetime(2000,9,1,tzinfo=timezone.utc) will create a datetime object encapsulating the time 1/9/2000 00:00 UTC

Explanation of the Python Code

- So accordingly, we define a function convert_to_weighted_network( ) as instructed, which takes the following 2 input arguments :
    1. An rdd, generated from the code used to solve question 1. For the reference of the user, the input rdd contains records in the format:

       Sender Email Id: The sender who sent the email.
       Receiver Email Id: The Id which received the email.
       Timestamp: When the corresponding mail was sent.

       It is important to note that each mail may have multiple records correspond to that mail in the input rdd, as each mail may have multiple receivers and our input rdd records 1 receiver-sender pair per receiver.

    2. An optional argument of date range, where d1 is the starting date of the range and d2 is the ending date of the range. When this argument is provided, our operations should be limited to records which falls between provided date range. Otherwise, we simply

report all the records provided in the input rdd after processing them in our function.

- if (drange != None):
  rdd = rdd.filter(lambda x : (x[2] > drange[0]) and (x[2] < drange[1]))

  The "if" block checks whether our optional argument is present in the input or not. Provided the date range is included in the input argument, we **filter** the input rdd according to the date range. Any included records should have date NEWER OR SAME as date 1 in the date range and should be OLDER OR SAME as date 2 in the date range.
  Example : if drange is ('1-JAN-2020', '1-JAN-2022'), then we are only concerned with emails which correspond to dates between those 2 provided dates, including the dates themselves.

  rdd10 = rdd.map(lambda x : ((x[0], x[1]), x[2]))
- The map function creates a tuple of the input rdd's Sender and Receiver pair and uses this tuple as a key to map the new rdd10.

  rdd11 = rdd10.groupByKey().mapValues(len)
- We group the rdd10 by their key, the Sender-Receiver pair and take a count of how many such records are present. The count is taken by mapping values with respect to keys and getting the length of the resulting list of values.

  rdd12 = rdd11.map(lambda x : (x[0][0], x[0][1], x[1]))
- Rdd12 simply flattens the resulting structure of Rdd11 by remapping all the values at the same level of structure, i.e., whereas in rdd11, the output would have been ((Sender, Receiver),Count), in rdd12 the output is (Sender, Receiver, Count). This is the desired format of output for Question 2.

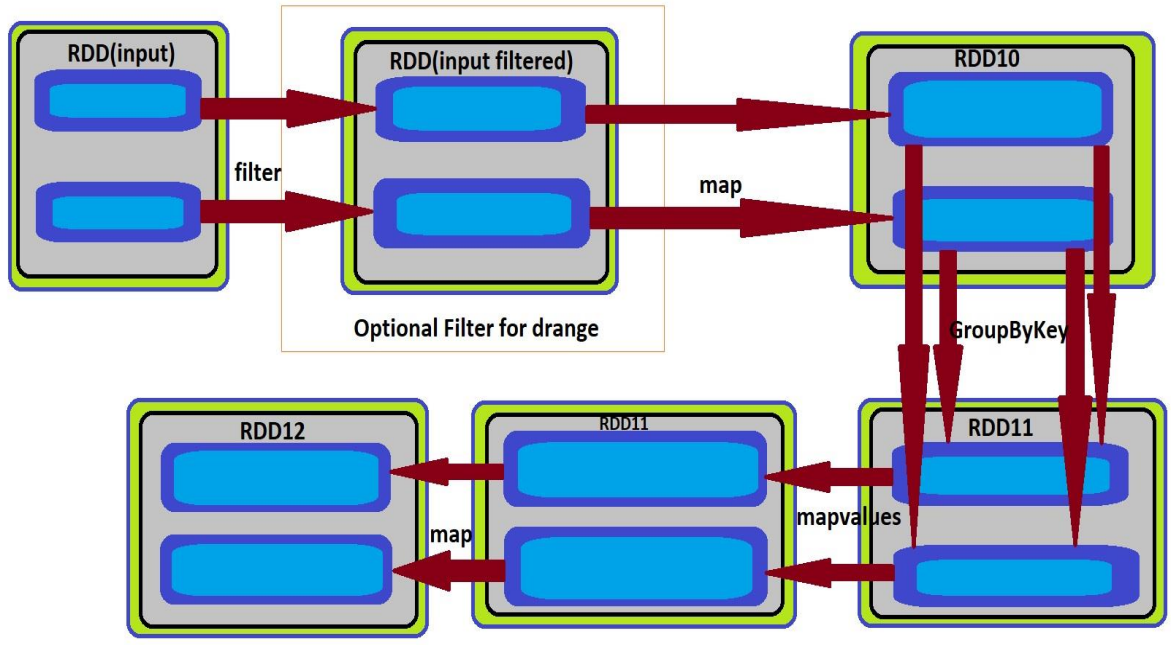
  return rdd12
- Simply return the resulting rdd12 as the output of the function convert_to_weighted_network().

Challenges and Pitfalls while implementing the code.

The code implemented is the most simplified version of code that I could come up with to solve the provided problem statement. The probable ways to count how many mails were present for a particular pair can be counted in myriad of ways in Python. Before we came up with the final version of the code reported here, we were experimenting with a few ways to implement this step. While there were other codes that achieved the same thing (e.g., **countByValue**()) , we felt the final version of the code is the most simple and efficient way to return the count without turning the resulting rdd into too complicated structures. Since that meant we could flatten the resulting rdd11 most efficiently, we went ahead with taking count by grouping with respect to (Sender, Receiver) and then taking the length of the records containing said (Sender, Receiver) pair.

Q2. Assume the count() action is called on the RDD returned by the function you have chosen. Draw a lineage graph (DAG) capturing the series of transformations triggered by the call. Your diagram should include dependencies between individual RDD partitions. You can assume that all RDDs have exactly two partitions. (30%)

Ans.



Q3. Are there any narrow dependencies in the lineage graph you have drawn in Question 1.2? Give one example of a narrow dependency if it is present. (5%)

Ans.

From the DAG, it is evident that most of the operations are Narrow Dependencies. For example, the map from rdd11 to rdd12 is an example of a narrow dependency present in the DAG.

Q4. Are there any wide dependencies in the lineage graph you have drawn in Question 1.2? Give one example of a wide dependency if it is present. (5%)

Ans.

From the DAG, we can identify only **one** Wide Dependency, which is represented by the groupBy() function responsible for transforming rdd10 to rdd11.

Question 2 (40%)

Many complex networks induced by either human interaction (e.g., friendship or follower graphs in online social networks, or World-Wide Web) or natural phenomena (e.g., protein-to-protein interaction networks) are known to be scale-free, i.e., the degree distribution in such networks follows a power law.

$$p(k) \sim k^{(-\alpha)}, \alpha > 1, (1)$$

where k is a non-negative integer, and p(k) is the probability that a node has the degree k.

Observe that (1) implies that a power law distribution will look roughly as a straight line if graphed on a log-log scale with the line's slope being equal to its exponent α.

A remarkable feature of a power law distribution is that all its moments above 1 are infinite, which implies that fluctuations around the mean can be arbitrarily high. This means that when we randomly choose a node in a scale-free network, we do not know what to expect: The selected node's degree could be tiny or arbitrarily large. In particular, this property predicts the emergence of hubs, i.e., nodes whose degrees can become disproportionally large (think e.g., of the number of people following a celebrity account on Twitter).

Pick a consecutive 12-month period contained within the range from January 2000 to March 2002 (inclusively). Use the functions you implemented for Questions 1 and 2 of the final project to extract a slice of the weighted network contained within the period you chose. Analyse its properties by answering the questions below. Note that if you worked on the final project jointly with another student, you and your partner must choose different slices to analyse.

Q1. The number of connections originating at or attracted by nodes in a scale-free network follows an 80/20 rule, that is, roughly 20% of the nodes are either origins or destinations of 80% of all edges in the network. (These 20% of nodes are, in fact, the hubs.)1 Use the code you developed for Questions 3 or 4 of the final project to compute the total weighted degree of all edges originating at (respectively attracted by) the 20% highest out-degree (respectively, in-degree) nodes in the network slice you selected. Briefly describe the methodology you used and your findings. Does the 80/20 rule indeed apply to your chosen network slice? You may use visualisations to support your conclusions. (10%)

Ans.

My chosen slice starts from 2000, August to 2001 August.

In the chosen 12 months period, the top nodes with the highest originating edges happen to be :

```
t4_out.collect()
```

```
[(28812, 'jeff.dasovich@enron.com'),
 (15604, 'cheryl.johnson@enron.com'),
 (13636, 'rhonda.denton@enron.com'),
 (12629, 'ginger.dernehl@enron.com'),
 (11545, 'pete.davis@enron.com'),
 (11258, 'susan.mara@enron.com'),
 (10781, 'veronica.espinoza@enron.com'),
 (9742, 'lorna.brennan@enron.com'),
 (6835, 'tana.jones@enron.com'),
 (4933, 'outlook.team@enron.com'),
 (4778, 'miyung.buster@enron.com'),
 (4614, 'kay.mann@enron.com'),
 (4050, 'stephanie.panus@enron.com'),
 (3817, 'janette.elbertson@enron.com'),
 (3514, 'mary.hain@enron.com'),
 (3423, 'alan.comnes@enron.com'),
 (3215, 'christi.nicolay@enron.com'),
 (3196, 'leonardo.pacheco@enron.com'),
 (3092, 'simone.la@enron.com'),
```

etc.

Similarly, the top nodes with the highest attracting edges happen to be:

```
t4_in.collect()
```

```
(2617, 'sara.shackleton@enron.com'),
(2378, 'paul.kaufman@enron.com'),
(2373, 'louise.kitchen@enron.com'),
(2198, 'mark.taylor@enron.com'),
(1952, 'tim.belden@enron.com'),
(1883, 'gerald.nemec@enron.com'),
(1824, 'john.lavorato@enron.com'),
(1710, 'sally.beck@enron.com'),
(1703, 'kate.symes@enron.com'),
(1683, 'sandra.mccubbin@enron.com'),
(1637, 'harry.kingerski@enron.com'),
(1615, 'alan.comnes@enron.com'),
(1598, 'karen.denne@enron.com'),
(1533, 'kay.mann@enron.com'),
(1467, 'vince.kaminski@enron.com'),
(1440, 'mark.guzman@enron.com'),
(1387, 'joe.hartsoe@enron.com'),
(1315, 'sarah.novosel@enron.com'),
(1280, 'linda.robertson@enron.com'),
(1278, 'william.bradford@enron.com'),
```

etc.

Methodology Used

The function explained in answer to question 1 produces a rdd of the format (Sender, Receiver, number of messages). From this result we can generate separate nodes and their edge counts for messages originating at those nodes and messages attracted to those nodes.

The python code is presented in the following diagram for the function that achieves to extract a tuple of (No of Edges, Node Name) from the function explained in question 1. This function does it for the outgoing edges, and with a simple modification to the code presented, i.e., changing the element called in the lambda functions in rdd13 and rdd14, we can have a function that gets similar output for incoming edges.

```
# Q3.1: replace pass with your code
def get_out_degrees(rdd):
    rdd13 = rdd.map(lambda x : (x[0], x[2]))
    rdd14 = rdd.map(lambda x: (x[1], 0))
    rdd15 = rdd13.union(rdd14)

    rdd16 = rdd15.groupByKey().mapValues(sum)
    rdd17 = rdd16.map(lambda x : (x[1], x[0])).sortBy(lambda x : (x[0],x[1]),ascending = False
    return rdd17
```

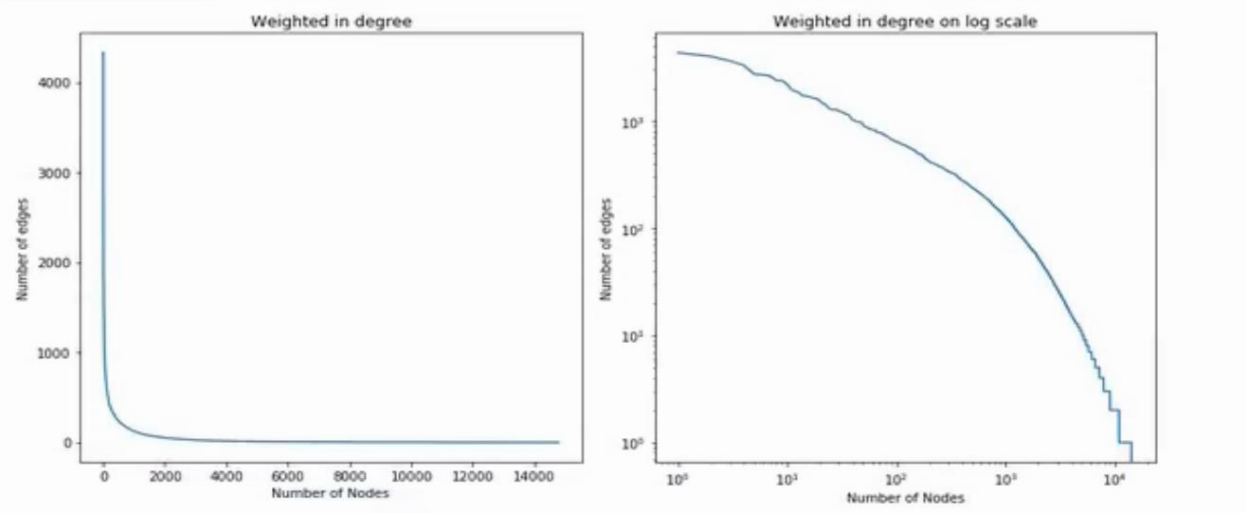
Using the function explained in Question 1 and a date range of August 2000 to August 2001, I extracted a network slice.

The following is the python code that extracts said network slice.

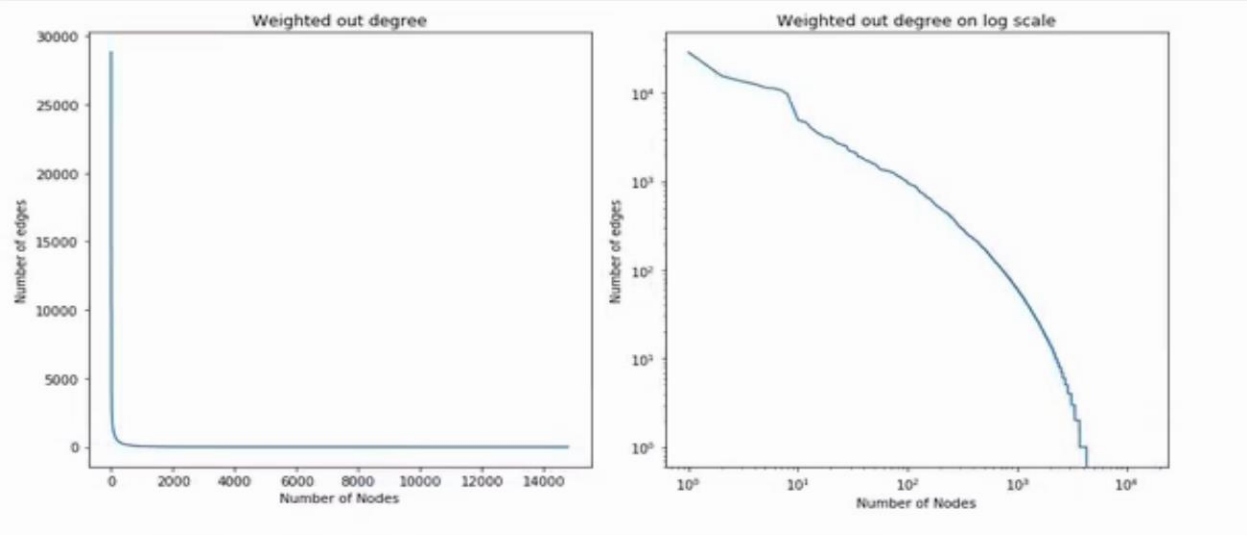
t3 = convert_to_weighted_network(t2,(datetime.datetime(2000, 8, 28, 6, 57, tzinfo=datetime.timezone(datetime.timedelta(-1, 61200))),datetime.datetime(2001, 8, 28, 6, 57, tzinfo=datetime.timezone(datetime.timedelta(-1, 61200)))))

Upon this slice, I called the function showed in the image above to generate the weighted network edges list explained in the opening part of this answer.

To better represent how many edges are generated only by the top minority note, I am including 4 graphs, 2 for incoming and 2 for outgoing edges, that shows most of the total number of edges are generated by roughly the top 20% of the Nodes.



The above image plots edges generated vs number of nodes generating the edges in a linear and logarithmic scale, respectively. Here it is clearly visible that out of 14000+ nodes, the most nodes only hold less than 200 edges. However, the top minority are responsible for holding more than 4000+ edges, hence the "**rich get richer**" rule holds.



The same holds for the number of attracted edges vs the number of nodes those edges are bound to. From the above chart, it is clearly visible that most of the nodes are attracted to top 20% of the total nodes in the network. Again, this holds the "**rich get richer**" motif.

Does the 80/20 rule still hold?

```
Total Outgoing Edges generated by Top 20% Nodes :  529622
Total Outgoing Edges present in the network :  532216
Percentage of Edges generated by Top 20% Nodes :  0.9951260390518135
================================================================================
Total Incoming Edges generated by Top 20% Nodes :  469550
Total Incoming Edges present in the network :  532216
Percentage of Edges generated by Top 20% Nodes :  0.8822545733311287
```

It is evident that **for outgoing edges, the top 20% of nodes are responsible for generating 99.5% of the edges**.

For Incoming edges, that number subsides to about 88.22% of the total incoming edges.

The code used to generate the result is as follows :

```
# calculating how many nodes fall in the top 20%
c1 = round(t4_in.count()/5)
c2 = round(t4_out.count()/5)

top_out = t4_out.take(c1)
top_in = t4_in.take(c2)

top_out_count = t4_out.map(lambda x: x[0]).take(c1)
total1 = t4_out.map(lambda x: x[0]).sum()
print('Total Outgoing Edges generated by Top 20% Nodes : ',sum(top_out_count))
print('Total Outgoing Edges present in the network : ', total1)
print('Percentage of Edges generated by Top 20% Nodes : ', sum(top_out_count)/total1)

print ('================================================================================')

top_in_count = t4_in.map(lambda x: x[0]).take(c2)
total2 = t4_in.map(lambda x: x[0]).sum()
print('Total Incoming Edges generated by Top 20% Nodes : ',sum(top_in_count))
print('Total Incoming Edges present in the network : ', total2)
print('Percentage of Edges generated by Top 20% Nodes : ', sum(top_in_count)/total2)
```

Here, we simply calculate the count of 20% nodes in c1 and c2. Then we take the top slice of total network using said indexing. Then the percentage is calculated using edges generated by top 20%/ total edges present in the network.

Q2. Another interesting property of the scale-free networks (colloquially known as ”the rich get richer”) is that the maximum node degree kmax is directly proportional to the number of nodes in the network. Use the functions you implemented for Questions 2 and 3 of the final project to compute kmax (for both in and out degrees) and the number of nodes for the sub-slices containing the Emails sent within the first n months, where $1 \leq n \leq 12$ of your chosen network slice. Briefly describe the methodology you used and your findings. Does kmax for either in or out degrees (or both) indeed grow linearly with the number of nodes in your chosen network slice? You may use visualisations to support your conclusions. (10%)

Ans.

Python Code to get the kmax for the in and out degrees:

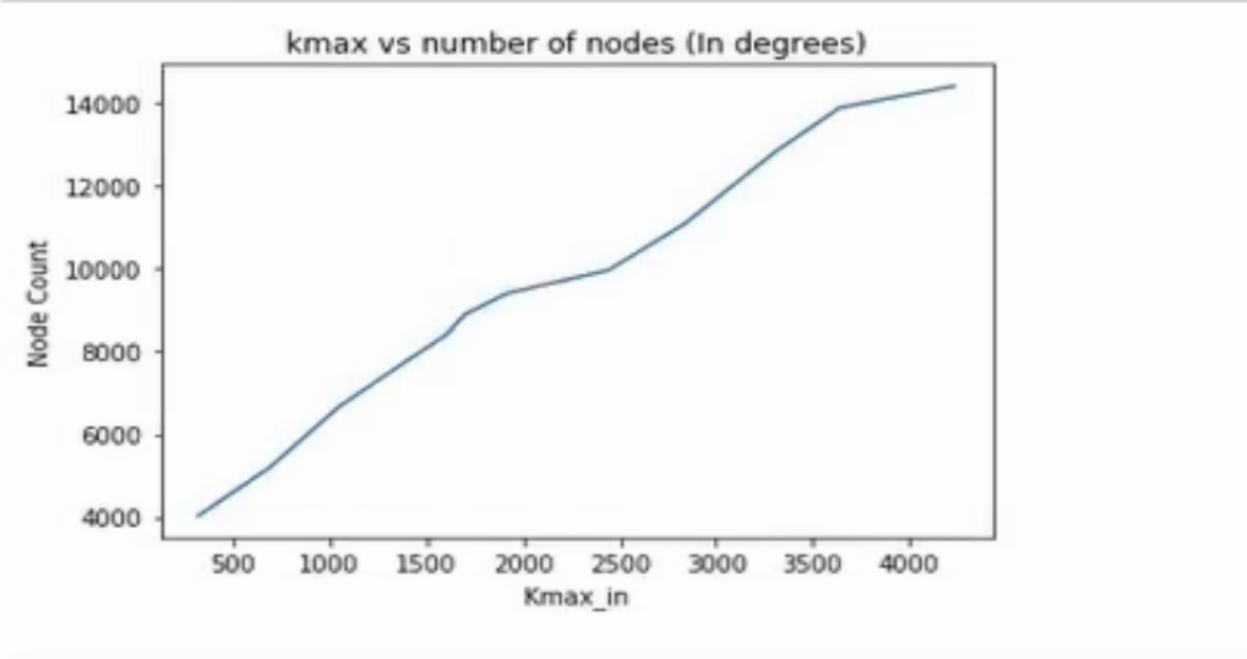
```
from datetime import timezone, datetime
```

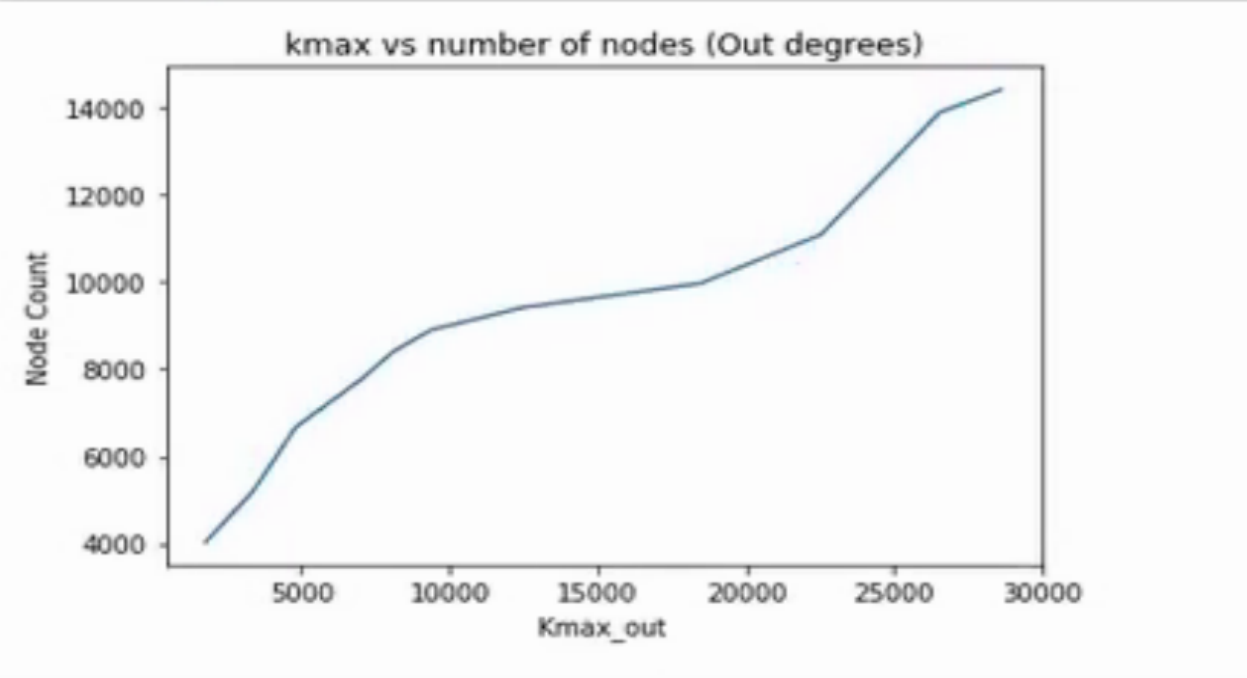
```
n = 9
y = 2000
kmax_in = []
kmax_out = []
nodes_count_in = []
nodes_count_out = []
month = []
for x in range(1,13):
    if n > 12:
        n = 1
        y += 1
    curr_network_slice = convert_to_weighted_network(t2, (datetime(2000,8,1,tzinfo=timezone.utc), datetime(y,n,1,tzinf

    rdd_in = get_in_degrees(curr_network_slice)
    rdd_out = get_out_degrees(curr_network_slice)

    kmax_in.append(rdd_in.collect()[0][0])
    kmax_out.append(rdd_out.collect()[0][0])

    nodes_count_in.append(len(rdd_in.collect()))
    nodes_count_out.append(len(rdd_out.collect()))

    month.append(n)
    n = n+1
```

1. Since all the necessary functions are already available to generate the output, we just have to loop through the months, and call the convert_to_weighted_network () function to extract the slice of the network for the period : starting date to current month in the loop, all the

way from current month = next month from the starting month to current month = 12 months from the starting month.

2. For the extracted network slice, we obtain the in and out degrees, using the get_in_degrees () and get_out_degrees () functions respectively. Since the functions return sorted outputs based on the number of edges, we get the nodes with the highest degree as the first element of the output for the function.
3. Once we have the nodes with the highest degree, it is just a matter of saving the total no of nodes present in that slice and kmax in separate lists for in degrees and out degrees respectively.
4. Finally, we simply plot kmax vs total no of nodes present in network for months 1 through 12 of the chosen period. The plots are included below.



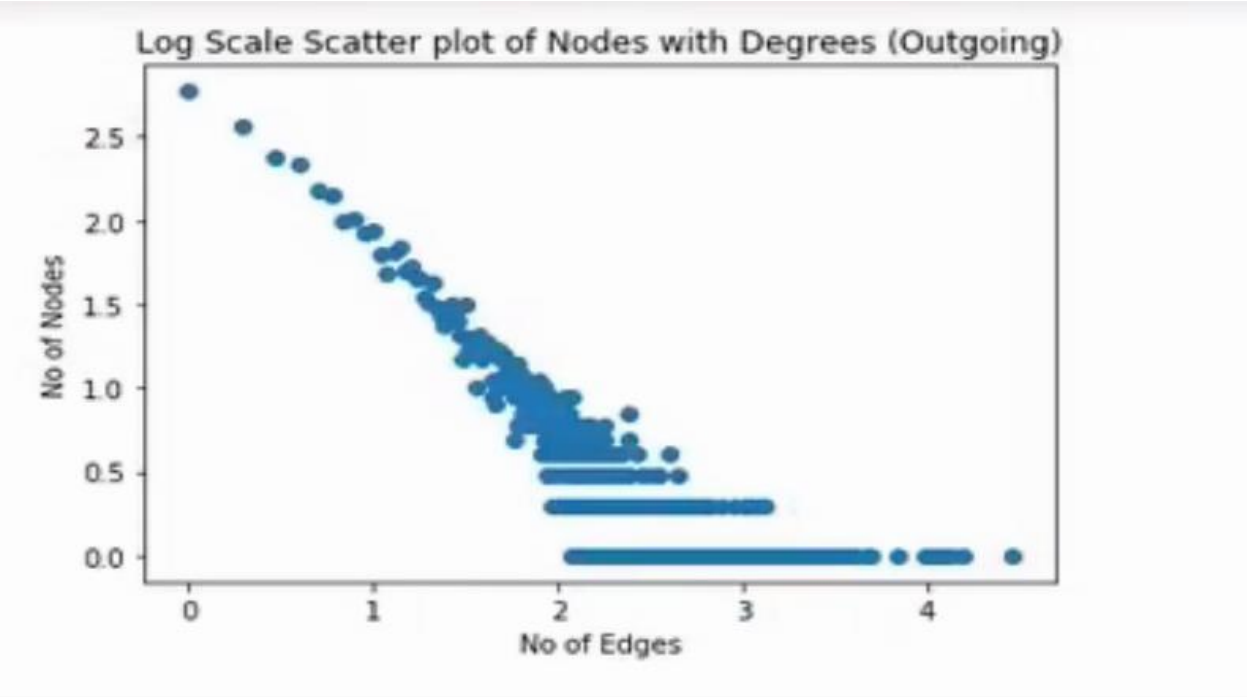


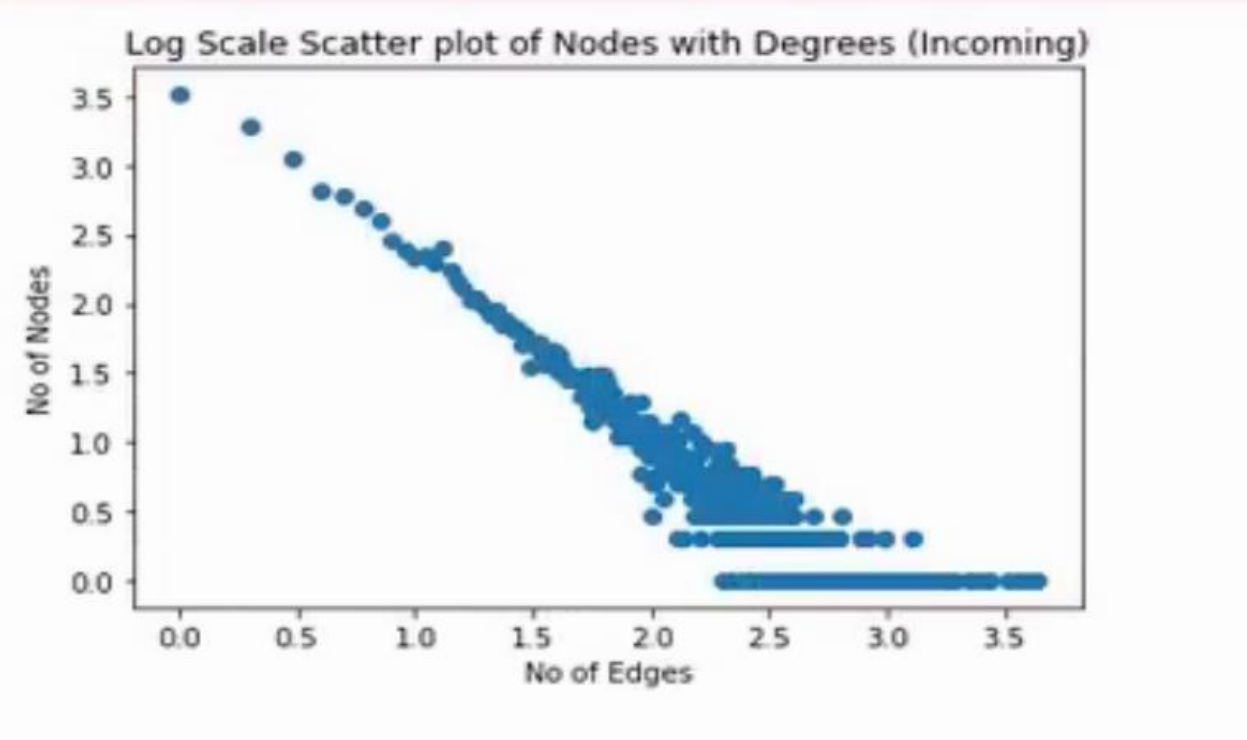
From the graphs it is evident, that kmax does indeed grow almost linearly in the initial few months period, but for a few particular months, the number of nodes in the network increases at a steady rate, while kmax increase at a relatively dampened rate. Excluding those cases, as a general rule, we can say that kmax increases linearly with the increase of nodes in the network.

Q3. Use the functions you implemented for Question 4 of the final project to compute the in and out degree distributions for your chosen network slice. Graph the distributions you obtained on a log-log scale and determine whether a straight line can be fit through the points. Briefly describe the methodology you used and your findings. Discuss whether your chosen network slice is indeed scale-free in terms of either it's in or out degrees (or both), and if so, give the value of the power law exponent α (or its range) for the relevant degree distribution. (20%)

Ans.

As instructed in the question, I have plotted the log v log scale graph for the function output of Q4.





From the graphs, it is clear that straight line can be fit through the points, however it is not going to exactly encompass most of the points.

Python code used to generate the plots:

```
x_in = t5_in.map(lambda x: x[0]).collect()
y_in = t5_in.map(lambda x: x[1]).collect()


plt.scatter(np.log10(x_in), np.log10(y_in))
plt.xlabel('No of Edges')
plt.ylabel('No of Nodes')
plt.title ('Log Scale Scatter plot of Nodes with Degrees (Incoming)')
plt.show()
```

Where t5_in is :

```
t5_in.collect()
 (18, 110),
 (19, 100),
 (20, 89),
 (21, 82),
 (22, 90),
 (23, 70),
 (24, 78),
 (25, 71),
 (26, 65),
```

etc.

For the outgoing edges, just a simple modification in the code is sufficient.

```
x_out = t5_out.map(lambda x: x[0]).collect()
y_out = t5_out.map(lambda x: x[1]).collect()


plt.scatter(np.log10(x_out), np.log10(y_out))
plt.xlabel('No of Edges')
plt.ylabel('No of Nodes')
plt.title ('Log Scale Scatter plot of Nodes with Degrees (Outgoing)')
plt.show()
```

Where t5_out is :

```
t5_out.collect()
```

```
[(0, 10502),
 (1, 592),
 (2, 360),
 (3, 238),
 (4, 214),
 (5, 152),
 (6, 143),
 (7, 98),
 (8, 101),
 (9, 83),
 (10, 87),
 (11, 63),
 (12, 48),
 (13, 64),
```

etc.

Is our network truly scale-free ?

A network can be considered scale-free in terms of its degree distribution if it follows a power-law distribution, which means that the probability of a node having k number of connections follows a mathematical function of the form:

$P(k) \sim k^{(-\alpha)}$

where P(k) is the probability of a node having k connections, and α is a constant exponent that typically lies between 2 and 3.

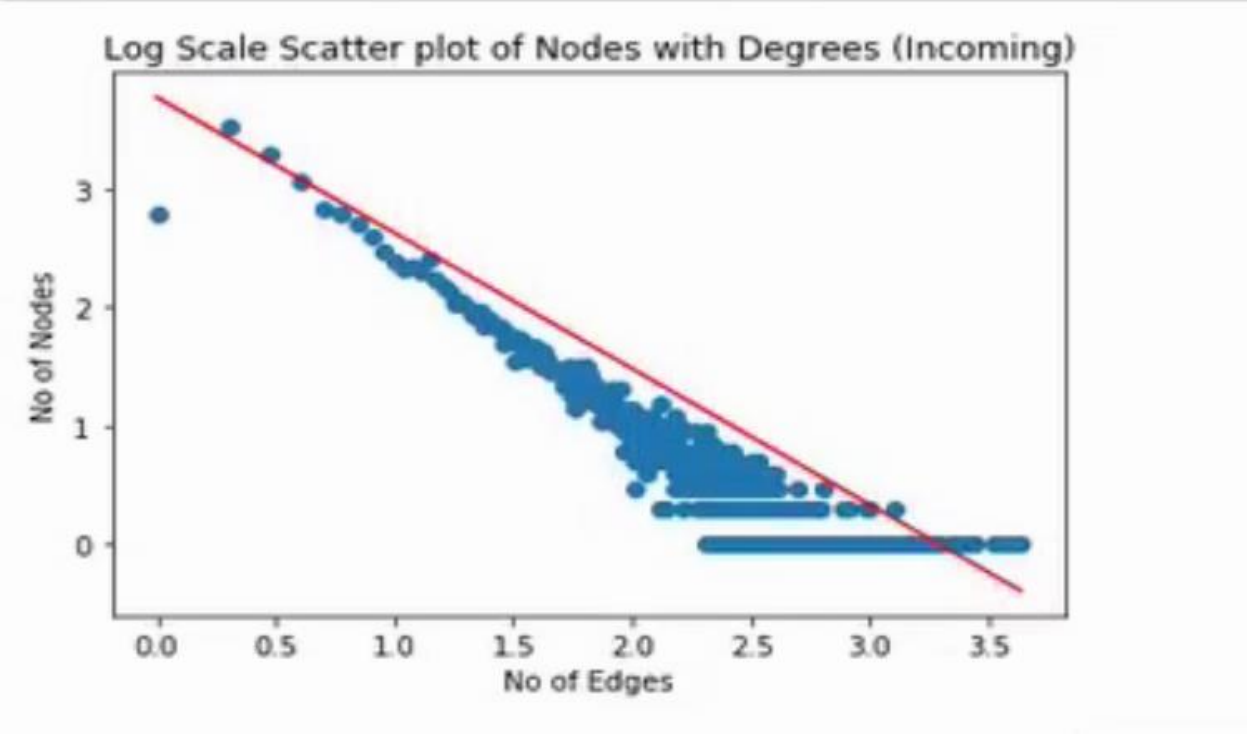
In other words, scale-free networks have a few highly connected nodes (called "hubs") and many nodes with only a few connections. This pattern is characteristic of many real-world networks, such as social networks, biological networks, and the internet.

To determine if a network is scale-free, one can plot the degree distribution on a log-log scale and look for a straight line, which would indicate a power-law distribution. However, it's important to note that not all networks follow a power-law distribution, and the presence of hubs alone is not sufficient to conclude that a network is scale-free. Other factors, such as the network's growth mechanism and its underlying topology, can also play a role.

FOR THE INCOMING EDGES :

```
x_in = t5_in.map(lambda x: x[0]).collect()
y_in = t5_in.map(lambda x: x[1]).collect()

x_in_mod = np.log10(np.array(x_in)+1)
y_in_mod = np.log10(np.array(y_in))
m,c = np.polyfit(x_in_mod,y_in_mod,1)


plt.scatter(x_in_mod, y_in_mod)
plt.plot(x_in_mod,m*x_in_mod+c,color ='red')
plt.xlabel('No of Edges')
plt.ylabel('No of Nodes')
plt.title ('Log Scale Scatter plot of Nodes with Degrees (Incoming)')
plt.show()
```

Here, we are adding 1 to each value in the x_in list to avoid any NaN or divide by zero issue in our polynomial straight line fit.



From the graph, the slope is clearly almost equal to 1, since xmax ~ ymax ~ 3.5 in Log v Log graph.

The exact value of alpha is given by :

α = 1.14

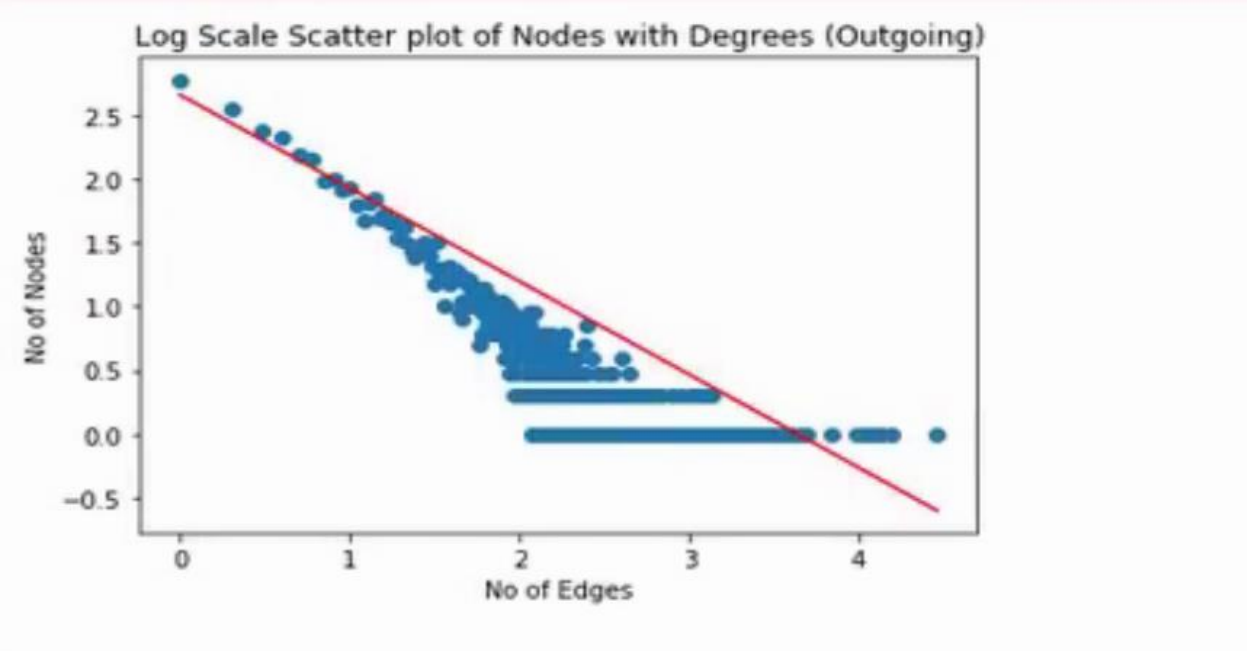
```
print (m,c)
```

```
-1.14316346105 3.76603780043
```

FOR THE OUTGOING EDGES :

```
x_out = t5_out.map(lambda x: x[0]).collect()
y_out = t5_out.map(lambda x: x[1]).collect()

x_out_mod = np.log10(np.array(x_out)+1)
y_out_mod = np.log10(np.array(y_out))
m,c = np.polyfit(x_out_mod,y_out_mod,1)


plt.scatter(np.log10(x_out), np.log10(y_out))
plt.plot(x_out_mod,m*x_out_mod+c,color ='red')
plt.xlabel('No of Edges')
plt.ylabel('No of Nodes')
plt.title ('Log Scale Scatter plot of Nodes with Degrees (Outgoing)')
plt.show()
```

Same logic as before, adding 1 to avoid errors in polynomial fit.



However, from this graph, we can see that the slope will not be greater than -1. In fact, the value of alpha is given by :

α = .73

```
print (m,c)
-0.731929238132 2.6596232339
```

Since our plot determines that our network is not truly scale-free for our outgoing network, we cannot calculate a sensible value of alpha which is larger than 1. It also holds that for our outgoing network, instead of 80/20 rule, it was more like 99/20 rule, as in the network is even more skewed for the top minority than usual scale-free networks.

For the incoming network, since the percentage is more like 88/20, the rule alpha > 1 holds, and we can see that the incoming network behaves more like a true scale free network.