project1_solutions

February 23, 2024

0.1 Network Science Project 1 Solutions

0.1.1 Spring 2024

0.1.2 Due: February 9th, 1:00pm GMT

Below are solutions to the Project, but by all means they are not the only solutions. In terms of code, marks were awarded for correctly working and efficient code. In terms of "explanation" questions, if you argued your point well, and comments on the data were made intuitively, marks were awarded.

0.1.3 Overview

Over the past few weeks we have been studying different graph models, with the aim of using these models to interpret real-world networks. In this project, we will compare the degree distributions of different models, and apply the G_{Np} model to a real-world dataset.

[ ]:
```
# Do not modify this cell or import any other modules
# without explicit permission.
# You should run this cell before running the code below.
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
#You may also use scipy as needed
```

0.2 Task 1: The Iteration Graph Model and Barabsai-Albert Model

0.2.1 Part 1: The iteration model (7 marks)

In Problem sheet 2, Question 1, the following model for an undirected graph was given: 1. Initial state: 2 nodes connected by 1 link. 2. Iteration 1: Replace the link between the node pair with two new nodes and four new links arranged so that the two original nodes are only connected by 2 "fully distinct" length-2 paths. Here, two paths are "fully distinct" if they have zero links in common. 3. Iteration i + 1: Apply the process for iteration 2 to each linked node pair in the graph at iteration i. So, for each linked pair of nodes in the graph at iteration i, and replace it with two new nodes and four new links so that the two 'old' nodes are connected by 2 new fully distinct length 2 paths.

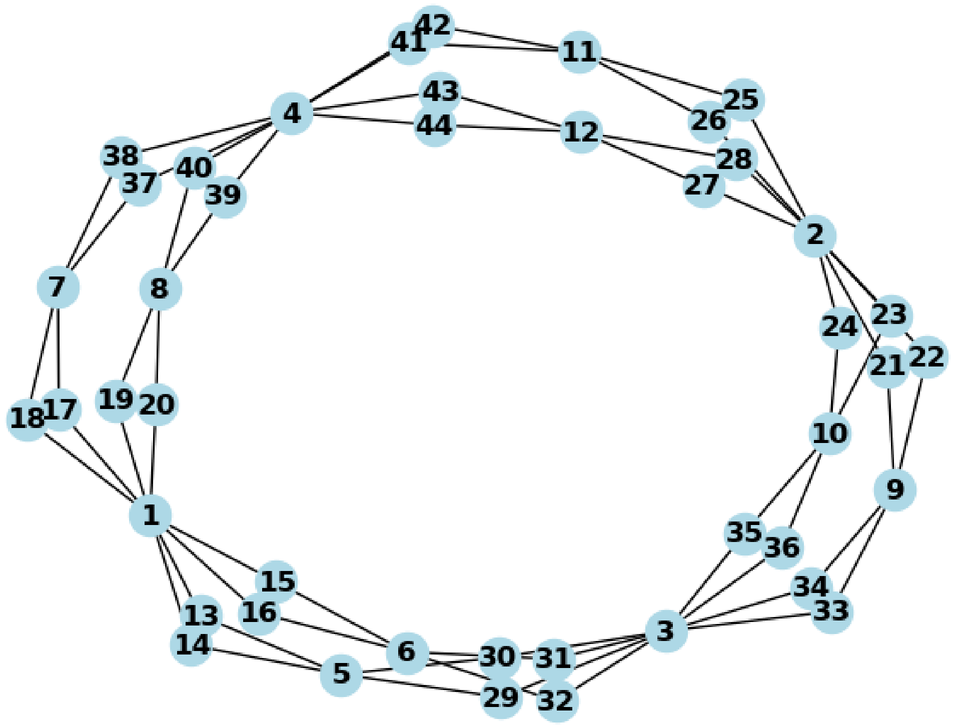
This model will now be named the *Iteration Model* for the purpose of the project.

1. Below, a function 'generate_graph' has been defined to generate a graph using the Iteration Model, for 'it' ammount of iterations. Complete the function below to *efficiently* generate the graph. Comments and some lines of code have been provided for guidance. You should think carefully about how to avoid unnecessary calculations and unnecessary loops. You may use numpy and scipy as needed. If using scipy, add the appropriate import statements to the cell below within the function. **Do not use or import any other modules for this question**. Below the function, provide a 2-3 sentence explanation of the main steps you have taken to make your code efficient.

[ ]:
```python
import networkx as nx
import numpy as np

import networkx as nx
import numpy as np

def generate_graph(it):
    """
    Generate a graph using the required model

    Input:
    it: The ammount of iterations required to generate the graph G

    Output:
    G: The graph after the desired iterations

    Please do not modify the function input or the return statement below
    """

    #The initial state
    G = nx.Graph()
    G.add_edge(1, 2)

    for p in range(it):
        # Get a list of edges in the current graph
        current_edges = list(G.edges())

        #Calculate no. of edges
        L = int(np.size(current_edges)/2)
        N = G.number_of_nodes()
        new_node2 = N

        #Process each link
        for k in range(0, L):
            # Remove the current edge
            (i,j) = current_edges[k]
            G.remove_edge(i,j)
            # Create two new nodes
            new_node1 = new_node2 + 1
            new_node2 = new_node1 + 1


            # Create required links
            G.add_edge(i, new_node1)
            G.add_edge(new_node1, j)
            G.add_edge(i, new_node2)
            G.add_edge(new_node2, j)

    return G
```

2. Draw the graph for three iterations (1 more than in the problem sheet), and verify the number of links or nodes are as expected by the theory derived in the problem sheet.

[ ]:
```
import matplotlib.pyplot as plt

# Code for calling generate_graph
# Number of iterations required
it = 3

# Create the graph
G = generate_graph(it)

# Draw the graph
nx.draw(G, with_labels=True, node_color='lightblue', font_weight='bold')
plt.show()

# Check no. of edges and nodes agree with theory
N_theory = G.number_of_nodes()
L_theory = G.number_of_edges()
if L_theory == 4**it:
    print('Number of links in graph agrees with theory')
if N_theory == 2 + 2*(4**it-1)/3:
    print('Number of nodes in graph agrees with theory')
```



```
Number of links in graph agrees with theory
Number of nodes in graph agrees with theory
```

The graph should be drawn with nodes labeled. The output from the cell above should indicate that yes the number of edges and nodes agree with the theory shown in problem sheet 2, question 1.

0.2.2 Part 2: Degree distribution and the Barabasi-Albert model (6 points)

We will study the Barabasi-Albert model in upcoming lectures. NextworkX generates a Barabasi-Albert graph when given the number of nodes 'n', and the number of edges 'm' to attach from a new node to an existing node. More information is given here https://networkx.org/documentation/stable/reference/generated/networkx.generators.random_graphs.barabasi_

Here, we will use the Barabasi-Albert model to generate a graph of N nodes, with $m = 2$, and compare the degree distribution of said graph, to the degree distribution of a graph generated from the Iteration model for 'it' iterations.

1) Develop a code below to create a well-designed figure to compare the degree distributions of a Barabasi-Albert graph for N nodes with m=2, and the degree distribution of the Iteration model. When comparing the two graphs generated by each model, the graphs should have the same amount of nodes N, and both distributions should be shown on the same plot. Produce

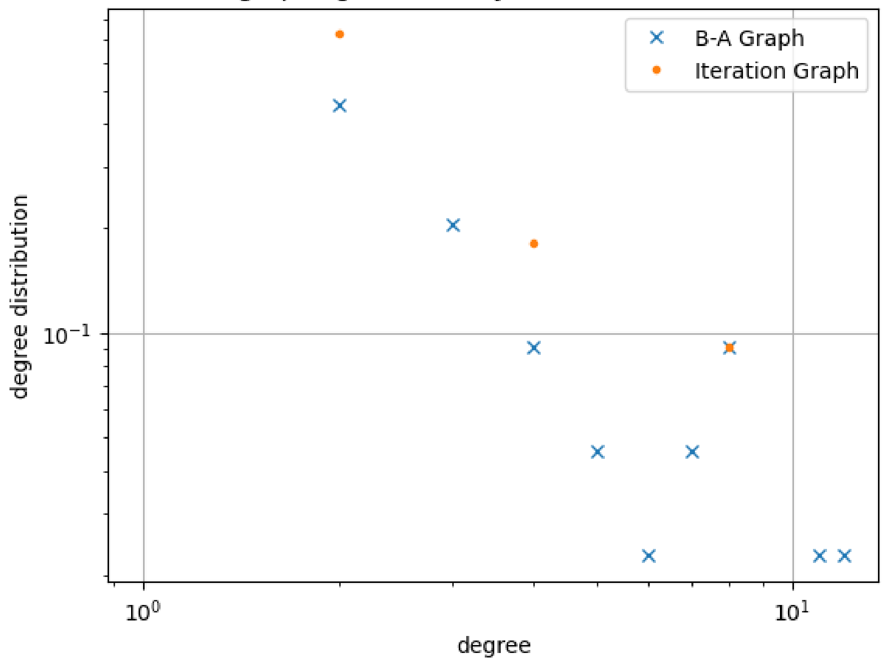
three figures in total by considering first 3 iterations (figure 1), and subsequently 4 (figure 2) and 5 (figure 3) (Note: taking the iterations too large will take a long time to run).

2) Describe the trends you see in the three figures in a short paragraph, focusing on the patterns associated with changing the amount of iterations (and therefore N), as well as differences and similarities between the two distributions.

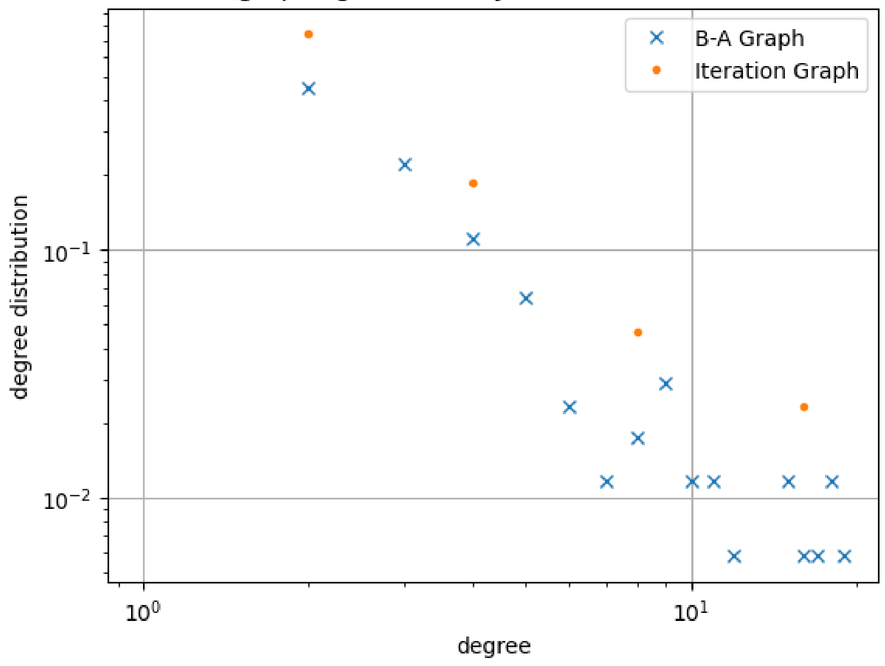
3) Does this change when m is varied? There is no need to show the figures for other m, just a comment on the patterns emerging.

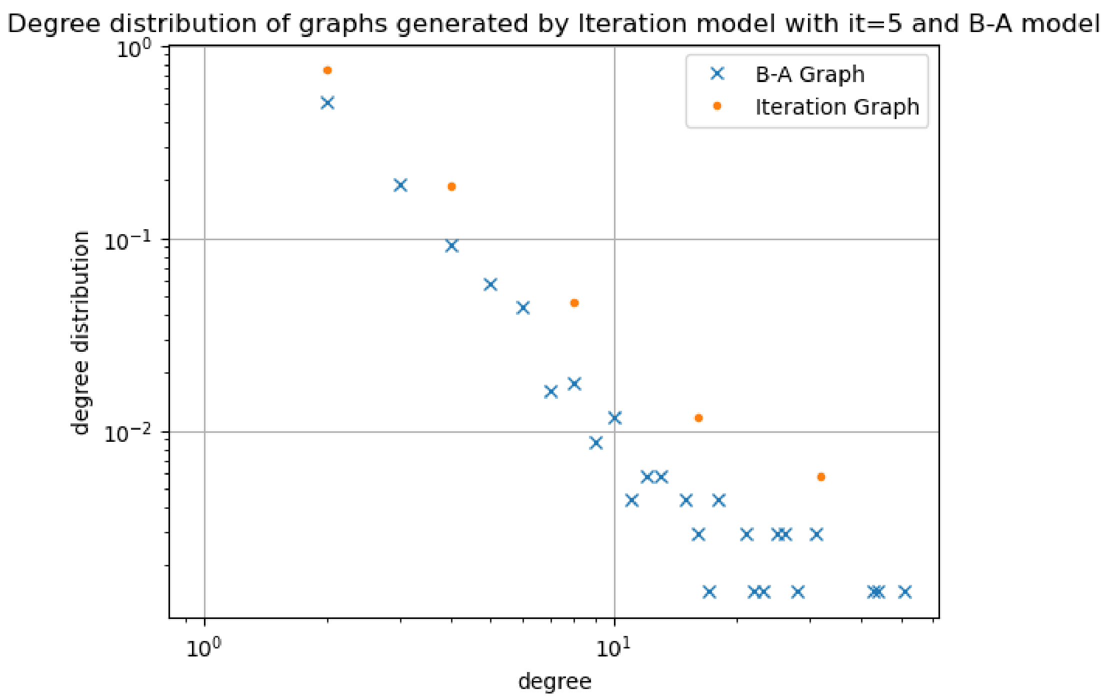
[ ]:
```
#List of no. of iterations to perform
iter = [3, 4, 5]

#Generate required figure
plt_title = ['Degree distribution of graphs generated by Iteration model with␣
 ↪it=3 and B-A model','Degree distribution of graphs generated by Iteration␣
 ↪model with it=4 and B-A model','Degree distribution of graphs generated by␣
 ↪Iteration model with it=5 and B-A model']
for i in range(3):
    it = iter[i]
    N = int(2 + 2*(4**it-1)/3)
    G_it = generate_graph(it)
    G_BA = nx.barabasi_albert_graph(N, 2)
    h_L = nx.degree_histogram(G_BA)
    h_G = nx.degree_histogram(G_it)
    plt.figure(i)
    plt.loglog(np.array(h_L)/N,'x',label='B-A Graph')
    plt.loglog(np.array(h_G)/N,'.',label='Iteration Graph')
    plt.xlabel('degree')
    plt.ylabel('degree distribution')
    plt.title(plt_title[i])
    plt.legend()
    plt.grid()

    #Check that the graphs have the same ammount of nodes
    N_Git = G_it.number_of_nodes()
    N_BA = G_BA.number_of_nodes()
    if N_Git != N_BA:
        print('Error: Number of nodes does not match')
```

Degree distribution of graphs generated by Iteration model with it=3 and B-A model



Degree distribution of graphs generated by Iteration model with it=4 and B-A model





Add explanation here

Firstly, note that loglog plots, in this instance, show the trends clearly. We can see that the nodes of highest degree in the Barabasi-Albert model consistently exhibit a higher degree than those of the Iteration model. However, for $m = 2$, as the iterations are increased, and therefore N is increased, this dispartity decreases. When $m = 2$, the models both follow a negative linear trend, at least for small and moderate degree, which becomes more apparent as N is increased. Here, we see the B-A model exhibits outliers from this trend, for example a small amount of nodes with degree 1 exist, where as the Iteration model has no nodes with degree 1. Additionally, there appears a critical degree value, k_1, where the distribution of nodes with degree $k \geq k_1$ changes very little. Finally, the iteration model does not experience a scatter of data like the B-A model, a direct result of the algorithm.

As m is increased, more nodes expierience a higher degree in the B-A model. Intuitively, this makes sense as m is defined as the number of edges to attach from a new node to an existing nodes. Thus, we are adding more edges to the problem. For example, when $m = 10$ the B-A model and Iteration model do not have the same degree distribution when the degree is less than 10. In fact, as we increase the number of iterations in the Iteration model, and therefore N, the B-A graph has no nodes with degree less than 10, where as the Iteration graph does.

0.3 Task 2: Comparing the G_{Np} model with a real-world dataset (7 points)

We have considered the G_{Np} model in Lectures and Labs. In particular, we have studied the expected values of certain model properties and derived equations for these. For this part of the project, we won't be using the G_{Np} graph generator provided by NetworkX, but rather the derivations derived in lectures for the expected number of edges and triangles of a G_{Np} graph, and the probability distribution of the G_{Np} graph. We will compare the expected values for a G_{Np} graph to the actual number of edges, triangles and degrees of nodes in a real-world data set, and produce a judgement on whether the dataset could be modeled with the G_{Np} model.

1) Below, the functions “expected_edges” and “degree_dist” have been defined in order to determine the expected number of edges for a graph generated by the G_{Np} model for N nodes and probability p, and to calculate the probability that a node has a degree k in a G_{Np} graph. Add code to complete the functions to return the expected number of edges and the probability distribution, using the theory from lectures. You will see that scipy.special.comb has been imported for use in the functions.

[ ]:
```python
from scipy.special import comb
def expected_edges(N,p):
    """Expected number of edges in G_Np graph
    Input:
    N: number of nodes
    p: probability of a link being placed between two distinct nodes

    Output:
    exp_L: expected number of edges

    """
    exp_L = comb(N,2)*p

    return exp_L

def degree_dist(N,p,k):
    """Probability that a node has a degree k in G_Np model
    Input:
    N: number of nodes
    p: probability of a link being placed between two distinct nodes
    k: degree

    Output:
    p_k: probability that a node has a degree k
    """

    p_k = (comb(N-1,k))*((p**k))*((1-p)**(N-1-k))

    return p_k


def expected_triangles(N,p):
    """Expected number of triangles in GNp graph
    """
    return comb(N,3)*p**3
```

```
def count_triangles(G):
    """Returns total number of triangles in G
    """
    t = nx.triangles(G)
    return np.sum(list(t.values()))/3
```

You have been provided with a dataset of email connections in an email network. Run the cell below to load the data. Check that there are 1133 nodes and 5451 edges in the network.

[ ]:
```
#Load data
edges_data = nx.read_edgelist('email-univ.edges')
G1 = nx.Graph(edges_data)

#Check nodes and edges are correct

N1 = G1.number_of_nodes()
N2 = G1.number_of_edges()
print(N1)
print(N2)
```

```
1133
5451
```

2)a) Using the functions previously defined in Task 2 Q1, compare the expected number of triangles and edges in a G_{Np} model. with the dataset values. You should consider three cases $p = 0.01, 0.1, 0.2$. Comment on the actual number of triangles and edges in comparison to the G_{Np} model.

[ ]:
```
#First find the number of nodes of the dataset
N1 = G1.number_of_nodes()

#Define a list with the probabilities we will consider
p_set = [0.01, 0.1, 0.2]

#The expected number of triangles for each p is
for p in p_set:
    exp_T = expected_triangles(N1, p)
    print('Expected number of triangles is', int(np.round(exp_T)), 'when p is',␣
 ↪p)

print('The amount of triangles in the data set is', count_triangles(G1) )

#Next consider the expected number of edges
for p in p_set:
    exp_L = expected_edges(N1, p)
    print('Expected edges is', int(np.round(exp_L)), 'when p is', p)

print('The amount of edges in the data set is', N2 )
```

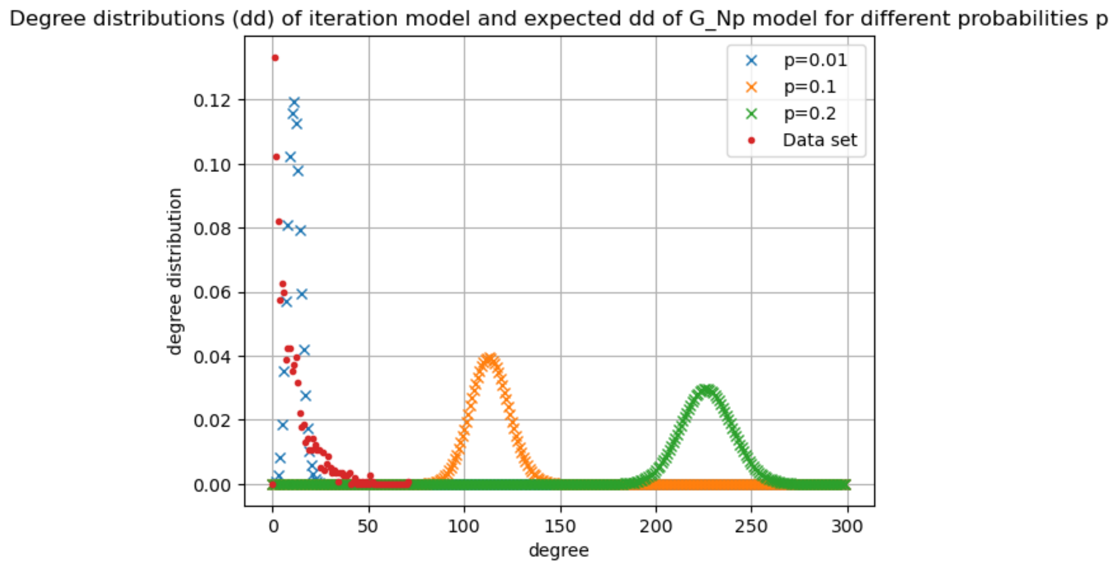
```
Expected number of triangles is 242 when p is 0.01
Expected number of triangles is 241762 when p is 0.1
Expected number of triangles is 1934094 when p is 0.2
The amount of triangles in the data set is 5343.0
Expected edges is 6413 when p is 0.01
Expected edges is 64128 when p is 0.1
Expected edges is 128256 when p is 0.2
The amount of edges in the data set is 5451
```

Add comments here

Clearly p is of high importance when comparing the data set to the model. Firstly, when considering the amount of triangles in the dataset, $p = 0.01$ has an expected number of triangles equal to 242 which is too low, whereas $p = 0.1$ expects 241762, which is much too large. Note, that even $p = 0.05$ produces expected triangles of 32064, which again is too large. Ideally, p between 0.01 and 0.05 would be needed to determine the correct number of triangles. The expected number of edges when $p = 0.01$ is 6413, and it therefore over predicts the number of edges by 20%. Yet, in comparison to the other values of p, it does a relatively good job.

2)b) First find the degree distribution for the dataset, then find the expected degree distribution for $0 \leq k \leq 300$ with a G_{Np} model when $p = 0.01, 0.1, 0.2$. Produce one figure showing this comparison for all four degree distributions for $0 \leq k \leq 300$. In a few lines describe what you see.

[ ]:
```python
#Degree distribution of data set
h_G1 = nx.degree_histogram(G1)
x = np.array(h_G1)/N1

#Find the expected degree distribution 0<k<300 and plot all distributions on␣
 ↪one figure
pk = np.zeros(300)
plt.figure()
legend_label = ['p=0.01', 'p=0.1', 'p=0.2']
for i in range(3):
    p = p_set[i]
    for k in range(300):
        pk[k] = degree_dist(N1,p,k)
    plt.plot(pk,'x', label = legend_label[i])
plt.plot(x[:300],'.',label='Data set')
plt.xlabel('degree')
plt.ylabel('degree distribution')
plt.title('Degree distributions (dd) of iteration model and expected dd of G_Np␣
 ↪model for different probabilities p')
plt.legend()
plt.grid()
```



Add comments here

Immediately we see that the dataset has no nodes with degree greater than ~ 75, and in general most nodes have small degree, with very few having degree greater than ~ 25. Consequently, the expected degree distribution when $p = 0.2$ in the model does not reflect the degree distribution of the dataset. Similarly, when $p = 0.1$. On the other hand, we could argue that $p = 0.01$ does a relatively okay job of modelling the degree distribution, in comparision to the other p we have considered. However, the distribution is skewed slightly to the right of the data set. This makes sense, since there were 20% more edges in the model for $p = 0.01$. When $p = 0.01$ there are less nodes of small degree $\sim 1 - 10$, in comparision to the data set where the degree distribution is at its maximum for this range of degrees. Yet, we see a good agreement for the distribution of nodes of degree $\sim 20 - 50$.

2c) Give your judgement on whether the G_{Np} model predicts the behaviour of the dataset accurately in a short paragraph.

As discussed in lectures, the model in general does not describe real-world networks accurately, and clearly when $p = 0.1, 0.2$ it does not work for this dataset either. However, when $p = 0.01$ it does a relatively okay job of predicting the degree distribution, considering it is such a simple model. Saying this, the expected number of triangles given by the model when $p = 0.01$ is far too low, yet the edges are too high. Therefore, by modelling the dataset with a model when $p = 0.01$, the graph is likely to be less dense than the real-world network.

0.3.1 Further guidance

- Your group should submit both a completed Jupyter notebook and a pdf version of the notebook (generated using File — Download as). If you cannot generate a pdf, try installing latex first. Each group should make a single submission. To submit your assignment, go to the **Assesments and Mark Schemes** folder on the module Blackboard page. In the folder there is another folder called **Coursework 1 Drop Box Spring 24** within this folder there are two dropbox's one for the pdf and one for a ZIP file which should include your ipynb file. To convert a file to ZIP, right click on the file and click "compress" and this will generate a ZIP. (these should be named *project1_groupx.ipynb* and *project1_groupx.pdf* where x is your group number. *Please make sure all CID numbers are written at the top of the project.

- You may use numpy, scipy, and matplotlib as needed. You may use networkx as needed. Please do not use any other packages without explicit permission.
- Marking will be based on the correctness of your work, the efficiency of your codes, and the degree to which your submission reflects a good understanding of the material covered up to the release of this assignment.
- This assignment requires sensible time-management on your part. Do not spend so much time on this assignment that it interferes substantially with your other modules. If you are concerned that your approach to the assignment may require an excessive amount of time, please get in touch with the instructor.
- Questions about the assignment should be asked in private settings. This can be a "private" question on Ed (which is distinct from "anonymous"), asking for a one-on-one meeting during office hours, or during a problem class.
- Please regularly backup your work. For example, you could keep an updated copy of your notebook on OneDrive.
- In order to assign partial credit, we need to understand what your code is doing, so please add comments to the code to help us.
- It may be helpful to initally develop your code in a Python module (outside of a function) and run it in a qtconsole (or similar Python terminal) so that you can readily access the values of the variables you are using.
- Feel free to use/modify codes that I have provided during the term.

[ ]:

```

```