ITCS 8010: Machine Learning with Graphs and Large Networks (Fall 2020)

Problem Set 2

Abdullah Al Raqibul Islam

The purpose of these exercises is to explore the `Signed` and `Disease Spreading` network. To get more understanding about the topics, we further implemented a couple of graph analysis algotithms. For example, to analyze the deviation of users behavior of a `Signed` network, we computed the `generative` and `receptive` surprise. For `Disease Spreading` network, we implemented the `SIR Model` to understand the likelihood of an epidemic in a given network. We further test our implementations on the provided datasets.

For this homework, I installed the `NetworkX` [2] network analysis package. The details about the installation of `NetworkX` can be found in [3]. Here is the software packages I used in this assignment:

- Python 3.7.6
- NetworkX 2.5
- Numpy
- Matplotlib

1. Analyzing the Signed networks [50 points]

Online networks are very useful in analyzing the social theory of structural balance and status inequality.

- Reading: Signed Networks in Social Media
- Reading: Predicting Positive and Negative Links in Online Social Networks

I have downloaded the Slashdot dataset and conduct analysis to answer the following questions:

1. Compute the number of triangles in the network.
   - **Answer:**
     - Number of cycles: 391205
     - Number of triangles: 22936
2. Report the fraction of balanced triangles and unbalanced triangles. (assume network is undirected; if there is a sign for each direction, randomly pick one.)
   - **Answer:**
     - Number of balanced triangles: 19238
     - Number of unbalanced triangles: 3698
3. Compare the frequency of signed triads in real and "shuffled" networks (refer slides) (assume network is undirected; if there is a sign for each direction, randomly pick one.)
   - **Answer:**
     - Edge-list size with negative sign: 114959
     - Number of cycles: 391205
     - Number of triad: 22936
     - $(+++)$ Original Vs. Shuffeled: 14605 - 9605
     - $(--+)$ Original Vs. Shuffeled: 4633 - 3347
     - $(+--)$ Original Vs. Shuffeled: 2918 - 9576
     - $(---)$ Original Vs. Shuffeled: 780 - 408

Is Balanced?	Triad	Slashdot	
		Real Network	Shuffled Network
Balanced	+++	0.64	0.42
	--+	0.20	0.15
Unbalanced	+--	0.13	0.42
	---	0.03	0.02

Table 1: Compare frequencies of signed triads in real and "shuffled" signs.

4. Compute "Gen. Surprise" (assume directed signed networks) for each of the 16 types.
   - **Answer:**

Type	Generative Surprise
Type-1	388.25
Type-2	-3.74
Type-3	388.54
Type-4	-4.08
Type-5	6.46
Type-6	4.66
Type-7	6.17
Type-8	2.28
Type-9	405.85
Type-10	4.59
Type-11	104.56
Type-12	-2.70
Type-13	3.60
Type-14	26.36
Type-15	-2.60
Type-16	-1.84

Table 2: "Generative Surprise" on Slashdot dataset

5. Rewrite the formula for "Rec. Surprise" using the idea introduced in "Gen. Surprise".
   - **Answer:**

`Generative baseline` (p_g) represents the fraction of positive feedback $(+)$ given by a user. On the other hand, `receptive baseline` (p_r) the fraction of positive feedback $(+)$ received by a user. In this context, `surprise` represents the behavioural deviation of two users (A/B) from baseline w.r.t. another user (X).

The formula for `Generative Surprise` of context X is:

$$s_g(X) = \frac{k - \sum_{i=1}^{n} p_g(A_i)}{\sqrt{\sum_{i=1}^{n} p_g(A_i) * (1 - p_g(A_i))}}$$

Here,

- $p_g(A_i)$ represents generative baseline of A_i
- Context X represents all the (A, B, X) triads, i.e., $(A_1, B_1|X_1)$,..., $(A_n, B_n|X_n)$
- k is the number of triad instances where X closed with a plus edges

Receptive surprise is similar, just use $p_r(A_i)$. So, the formula for `Receptive Surprise` of context X is:

$$s_r(X) = \frac{k - \sum_{i=1}^{n} p_r(A_i)}{\sqrt{\sum_{i=1}^{n} p_r(A_i) * (1 - p_r(A_i))}}$$

Here, $p_r(A_i)$ represents receptive baseline of A_i.

1. Compute "Rec. Surprise" for all each of the 16 types.
   - **Answer:**

Type	Receptive Surprise
Type-1	401.58
Type-2	-5.02
Type-3	400.85
Type-4	-5.10
Type-5	4.77
Type-6	3.14
Type-7	5.65
Type-8	0.55
Type-9	402.80
Type-10	4.31
Type-11	102.21
Type-12	-2.17
Type-13	5.95
Type-14	26.76
Type-15	-0.87
Type-16	-0.58

Table 3: "Receptive Surprise" on Slashdot dataset

In [111…

```
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
import collections as collec
import math
import random
from random import randrange
%matplotlib inline
```

Reading `Slashdot` graph as an weighted undirected graph:

In [162…

```
G_slash_undir = nx.read_edgelist("data/soc-sign-Slashdot.txt", nodetype=int, data=(("s
```

Check basic graph properties:

In [31]:

```
print("Number of nodes: {}".format(nx.number_of_nodes(G_slash_undir)))
print("Number of edges: {}".format(nx.number_of_edges(G_slash_undir)))
```

```
Number of nodes: 77350
Number of edges: 468554
```

1. Compute the number of triangles in the network.

In [29]:

```
cycle_list = nx.cycle_basis(G_slash_undir)

print("Number of cycles: {}".format(len(cycle_list)))

triangle_count = 0
for cycle in cycle_list:
    if len(cycle) == 3:
        triangle_count += 1

print("Number of triangles: {}".format(triangle_count))
```

```
Number of cycles: 391205
Number of triangles: 22936
```

2. Report the fraction of balanced triangles and unbalanced triangles. (assume network is undirected; if there is a sign for each direction, randomly pick one.)

In [163…

```
cycle_list = nx.cycle_basis(G_slash_undir)
triad_count = 0

print("Number of cycles: {}".format(len(cycle_list)))

fff_tri_count = 0
eef_tri_count = 0
ffe_tri_count = 0
eee_tri_count = 0
for cycle in cycle_list:
    if len(cycle) == 3:
        triad_count += 1

        # print(cycle)
        sum_sign = G_slash_undir[cycle[0]][cycle[1]]['sign'] + G_slash_undir[cycle[1]
        # print(G_slash_undir[cycle[0]][cycle[1]]['sign'])
        # print(G_slash_undir[cycle[1]][cycle[2]]['sign'])
        # print(G_slash_undir[cycle[2]][cycle[0]]['sign'])
        if sum_sign == 3:
            fff_tri_count += 1
        if sum_sign == -1:
            eef_tri_count += 1
        if sum_sign == 1:
            ffe_tri_count += 1
        if sum_sign == -3:
            eee_tri_count += 1


print("Number of triads: {}".format(triad_count))
print("Number of balanced triangles: {}".format(fff_tri_count + eef_tri_count))
print("Number of unbalanced triangles: {}".format(ffe_tri_count + eee_tri_count))
```

```
Number of cycles: 391205
Number of triad: 22936
Number of balanced triangles: 19238
Number of unbalanced triangles: 3698
```

3. Compare the frequency of signed triads in real and "shuffled" networks (refer slides) (assume network is undirected; if there is a sign for each direction, randomly pick one.)

In [164…

```
G_slash_undir_shuffle = nx.read_edgelist("data/soc-sign-Slashdot.txt", nodetype=int, d
num_nodes = nx.number_of_nodes(G_slash_undir_shuffle)

#print("Edges: {}".format(G_slash_undir_shuffle.edges(data=True)))

# selecting negative edges
edge_list = [(u,v) for u,v,e in G_slash_undir_shuffle.edges(data=True) if e['sign'] ==

random.seed()
print("Edge-list size with negative sign: {}".format(len(edge_list)))
#print(edge_list)
for edge in edge_list:
    shuffeled = False
    while shuffeled == False:
        u = random.randint(0, num_nodes)
        v = random.randint(0, num_nodes)
        #print(u)
        #print(v)
        if G_slash_undir_shuffle.has_edge(u, v) and G_slash_undir_shuffle[u][v]['sign
            shuffeled = True
            G_slash_undir_shuffle[u][v]['sign'] = -1
            G_slash_undir_shuffle[edge[0]][edge[1]]['sign'] = 1

print("Shuffeled the edge signs!")

cycle_list = nx.cycle_basis(G_slash_undir_shuffle)
triad_count = 0

print("Number of cycles: {}".format(len(cycle_list)))

s_fff_tri_count = 0
s_eef_tri_count = 0
s_ffe_tri_count = 0
s_eee_tri_count = 0
for cycle in cycle_list:
    if len(cycle) == 3:
        triad_count += 1

        # print(cycle)
        sum_sign = G_slash_undir_shuffle[cycle[0]][cycle[1]]['sign'] + G_slash_undir_s
        # print(G_slash_undir[cycle[0]][cycle[1]]['sign'])
        # print(G_slash_undir[cycle[1]][cycle[2]]['sign'])
        # print(G_slash_undir[cycle[2]][cycle[0]]['sign'])
        if sum_sign == 3:
            s_fff_tri_count += 1
        if sum_sign == -1:
            s_eef_tri_count += 1
        if sum_sign == 1:
            s_ffe_tri_count += 1
        if sum_sign == -3:
            s_eee_tri_count += 1


print("Number of triads: {}".format(triad_count))

print("fff Original Vs. Shuffeled: {} - {}".format(fff_tri_count, s_fff_tri_count))
print("eef Original Vs. Shuffeled: {} - {}".format(eef_tri_count, s_eef_tri_count))
print("ffe Original Vs. Shuffeled: {} - {}".format(ffe_tri_count, s_ffe_tri_count))
print("eee Original Vs. Shuffeled: {} - {}".format(eee_tri_count, s_eee_tri_count))
```

```
Edge-list size with negative sign: 114959
Shuffeled the edge signs!
Number of cycles: 391205
Number of triads: 22936
fff Original Vs. Shuffeled: 14605 - 9161
eef Original Vs. Shuffeled: 4633 - 3521
ffe Original Vs. Shuffeled: 2918 - 9798
eee Original Vs. Shuffeled: 780 - 456
```

Reading `Slashdot` graph as an weighted directed graph:

In [167…

```
G_slash_dir = nx.read_edgelist("data/soc-sign-Slashdot.txt", nodetype=int, data=(("si
```

Check basic graph properties:

In [168…

```
print("Number of nodes: {}".format(nx.number_of_nodes(G_slash_dir)))
print("Number of edges: {}".format(nx.number_of_edges(G_slash_dir)))
```

```
Number of nodes: 77350
Number of edges: 516575
```

4. Compute "Gen. Surprise" (assume directed signed networks) for each of the 16 types

In [169…

```
# G_slash_dir = nx.read_edgelist("data/tmp.txt", nodetype=int, data=(("sign", int),),

# store K value for each type
K = {1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6: 0, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0,

# store P_g value for each type
P_g = {1: 0.0, 2: 0.0, 3: 0.0, 4: 0.0, 5: 0.0, 6: 0.0, 7: 0.0, 8: 0.0, 9: 0.0, 10: 0.0

# store Denominator value for each type's S_g formula, needs to make square-root over
Denominator = {1: 0.0, 2: 0.0, 3: 0.0, 4: 0.0, 5: 0.0, 6: 0.0, 7: 0.0, 8: 0.0, 9: 0.0,


"""
    Triad type matcher on graph "G_slash_dir".
    This function assumes there is a directed edge from node "a" to node "b".

    attributes
    ==========
    a            node-id
    b            node-id
    x            node-id
"""
def triad_type_matcher(a, b, x):
    ret = []
    # print("a: {}, b: {}, x: {}".format(a, b, x))
    if G_slash_dir.has_edge(a, x) and G_slash_dir.has_edge(x, b):
        # print("a->x: {}, x->b: {}".format(G_slash_dir[a][x]['sign'], G_slash_dir[x]
        if G_slash_dir[a][x]['sign'] == 1 and G_slash_dir[x][b]['sign'] == 1:
            ret.append(1)
        elif G_slash_dir[a][x]['sign'] == 1 and G_slash_dir[x][b]['sign'] == -1:
            ret.append(2)
        elif G_slash_dir[a][x]['sign'] == -1 and G_slash_dir[x][b]['sign'] == 1:
            ret.append(5)
        elif G_slash_dir[a][x]['sign'] == -1 and G_slash_dir[x][b]['sign'] == -1:
            ret.append(6)

    if G_slash_dir.has_edge(a, x) and G_slash_dir.has_edge(b, x):
        # print("a->x: {}, b->x: {}".format(G_slash_dir[a][x]['sign'], G_slash_dir[b]
        if G_slash_dir[a][x]['sign'] == 1 and G_slash_dir[b][x]['sign'] == 1:
            ret.append(3)
        elif G_slash_dir[a][x]['sign'] == 1 and G_slash_dir[b][x]['sign'] == -1:
            ret.append(4)
        elif G_slash_dir[a][x]['sign'] == -1 and G_slash_dir[b][x]['sign'] == 1:
            ret.append(7)
        elif G_slash_dir[a][x]['sign'] == -1 and G_slash_dir[b][x]['sign'] == -1:
            ret.append(8)

    if G_slash_dir.has_edge(x, a) and G_slash_dir.has_edge(x, b):
        # print("x->a: {}, x->b: {}".format(G_slash_dir[x][a]['sign'], G_slash_dir[x]
        if G_slash_dir[x][a]['sign'] == 1 and G_slash_dir[x][b]['sign'] == 1:
            ret.append(9)
        elif G_slash_dir[x][a]['sign'] == 1 and G_slash_dir[x][b]['sign'] == -1:
            ret.append(10)
        elif G_slash_dir[x][a]['sign'] == -1 and G_slash_dir[x][b]['sign'] == 1:
            ret.append(13)
        elif G_slash_dir[x][a]['sign'] == -1 and G_slash_dir[x][b]['sign'] == -1:
            ret.append(14)

    if G_slash_dir.has_edge(x, a) and G_slash_dir.has_edge(b, x):
        # print("x->a: {}, b->x: {}".format(G_slash_dir[x][a]['sign'], G_slash_dir[b]
        if G_slash_dir[x][a]['sign'] == 1 and G_slash_dir[b][x]['sign'] == 1:
            ret.append(11)
        elif G_slash_dir[x][a]['sign'] == 1 and G_slash_dir[b][x]['sign'] == -1:
            ret.append(12)
        elif G_slash_dir[x][a]['sign'] == -1 and G_slash_dir[b][x]['sign'] == 1:
            ret.append(15)
        elif G_slash_dir[x][a]['sign'] == -1 and G_slash_dir[b][x]['sign'] == -1:
            ret.append(16)

    return ret


"""
    Triad type matcher on graph "G_slash_dir".
    This function assumes there is a directed edge from node "a" to node "b".

    attributes
    ==========
    a            node-id
    b            node-id
    x            node-id
"""
def update_dictionary(a, b, x, type_):
    if G_slash_dir[a][b]['sign'] == 1:
        K[type_] += 1

    out_deg = G_slash_dir.out_degree(a)
    pos_out_deg = len([(u,v) for u,v,e in G_slash_dir.out_edges(a, data=True) if e['s
    # print("Node-id: {}, Pos-OutEdges: {}".format(a, [(u,v) for u,v,e in G_slash_dir
    # print("Pos-OutDeg: {}, OutDeg: {}".format(pos_out_deg, out_deg))

    P_g[type_] += (pos_out_deg/float(out_deg))
    Denominator[type_] += ((pos_out_deg/float(out_deg)) * (1 - (pos_out_deg/float(out_


def print_gen_surprise():
    for type_ in range(1, 17):
        gen_surp = 0.0
        if Denominator[type_] != 0.0:
            gen_surp = K[type_] - P_g[type_] / math.sqrt(Denominator[type_])
        # else:
            # print("Type-{} is not covered. K: {}, P_g: {}, Denominator: {}".format(

        print("Generative Surprise for type-{}: {}".format(type_, gen_surp))


def calculate_gen_surprise():
    triad_gen = nx.all_triads(G_slash_dir)
    triad_nums = 0

    for triad in triad_gen:
        triad_nodes = list(triad.nodes())
        # print(type(triad_nodes))
        # print(triad_nodes)

        triad_nums += 1
        if triad_nums % 100000 == 0:
            print("Finished {} triads.".format(triad_nums))

        if triad_nums >= 10000000:
            break

        if G_slash_dir.has_edge(triad_nodes[0], triad_nodes[1]):
            types = triad_type_matcher(triad_nodes[0], triad_nodes[1], triad_nodes[2]
            # print("a: {} to b: {} got: {}".format(triad_nodes[0], triad_nodes[1], t
            for type_ in types:
                update_dictionary(triad_nodes[0], triad_nodes[1], triad_nodes[2], type

        if G_slash_dir.has_edge(triad_nodes[1], triad_nodes[0]):
            types = triad_type_matcher(triad_nodes[1], triad_nodes[0], triad_nodes[2]
            # print("a: {} to b: {} got: {}".format(triad_nodes[1], triad_nodes[0], t
            for type_ in types:
                update_dictionary(triad_nodes[1], triad_nodes[0], triad_nodes[2], type

        if G_slash_dir.has_edge(triad_nodes[1], triad_nodes[2]):
            types = triad_type_matcher(triad_nodes[1], triad_nodes[2], triad_nodes[0]
            # print("a: {} to b: {} got: {}".format(triad_nodes[1], triad_nodes[2], t
            for type_ in types:
                update_dictionary(triad_nodes[1], triad_nodes[2], triad_nodes[0], type

        if G_slash_dir.has_edge(triad_nodes[2], triad_nodes[1]):
            types = triad_type_matcher(triad_nodes[2], triad_nodes[1], triad_nodes[0]
            # print("a: {} to b: {} got: {}".format(triad_nodes[2], triad_nodes[1], t
            for type_ in types:
                update_dictionary(triad_nodes[2], triad_nodes[1], triad_nodes[0], type

        if G_slash_dir.has_edge(triad_nodes[0], triad_nodes[2]):
            types = triad_type_matcher(triad_nodes[0], triad_nodes[2], triad_nodes[1]
            # print("a: {} to b: {} got: {}".format(triad_nodes[0], triad_nodes[2], t
            for type_ in types:
                update_dictionary(triad_nodes[0], triad_nodes[2], triad_nodes[1], type

        if G_slash_dir.has_edge(triad_nodes[2], triad_nodes[0]):
            types = triad_type_matcher(triad_nodes[2], triad_nodes[0], triad_nodes[1]
            # print("a: {} to b: {} got: {}".format(triad_nodes[2], triad_nodes[0], t
            for type_ in types:
                update_dictionary(triad_nodes[2], triad_nodes[0], triad_nodes[1], type


calculate_gen_surprise()
print_gen_surprise()
```

```
Finished 100000 triads.
Finished 200000 triads.
Finished 300000 triads.
Finished 400000 triads.
Finished 500000 triads.
Finished 600000 triads.
Finished 700000 triads.
Finished 800000 triads.
Finished 900000 triads.
Finished 1000000 triads.
Finished 1100000 triads.
Finished 1200000 triads.
Finished 1300000 triads.
Finished 1400000 triads.
Finished 1500000 triads.
Finished 1600000 triads.
Finished 1700000 triads.
Finished 1800000 triads.
Finished 1900000 triads.
Finished 2000000 triads.
Finished 2100000 triads.
Finished 2200000 triads.
Finished 2300000 triads.
Finished 2400000 triads.
Finished 2500000 triads.
Finished 2600000 triads.
Finished 2700000 triads.
Finished 2800000 triads.
Finished 2900000 triads.
Finished 3000000 triads.
Finished 3100000 triads.
Finished 3200000 triads.
Finished 3300000 triads.
Finished 3400000 triads.
Finished 3500000 triads.
Finished 3600000 triads.
Finished 3700000 triads.
Finished 3800000 triads.
Finished 3900000 triads.
Finished 4000000 triads.
Finished 4100000 triads.
Finished 4200000 triads.
Finished 4300000 triads.
Finished 4400000 triads.
Finished 4500000 triads.
Finished 4600000 triads.
Finished 4700000 triads.
Finished 4800000 triads.
Finished 4900000 triads.
Finished 5000000 triads.
Finished 5100000 triads.
Finished 5200000 triads.
Finished 5300000 triads.
Finished 5400000 triads.
Finished 5500000 triads.
Finished 5600000 triads.
Finished 5700000 triads.
Finished 5800000 triads.
Finished 5900000 triads.
Finished 6000000 triads.
Finished 6100000 triads.
Finished 6200000 triads.
Finished 6300000 triads.
Finished 6400000 triads.
Finished 6500000 triads.
Finished 6600000 triads.
Finished 6700000 triads.
Finished 6800000 triads.
Finished 6900000 triads.
Finished 7000000 triads.
Finished 7100000 triads.
Finished 7200000 triads.
Finished 7300000 triads.
Finished 7400000 triads.
Finished 7500000 triads.
Finished 7600000 triads.
Finished 7700000 triads.
Finished 7800000 triads.
Finished 7900000 triads.
Finished 8000000 triads.
Finished 8100000 triads.
Finished 8200000 triads.
Finished 8300000 triads.
Finished 8400000 triads.
Finished 8500000 triads.
Finished 8600000 triads.
Finished 8700000 triads.
Finished 8800000 triads.
Finished 8900000 triads.
Finished 9000000 triads.
Finished 9100000 triads.
Finished 9200000 triads.
Finished 9300000 triads.
Finished 9400000 triads.
Finished 9500000 triads.
Finished 9600000 triads.
Finished 9700000 triads.
Finished 9800000 triads.
Finished 9900000 triads.
Finished 10000000 triads.
Generative Surprise for type-1: 388.2544289596636
Generative Surprise for type-2: -3.7381487074464657
Generative Surprise for type-3: 388.535100067563
Generative Surprise for type-4: -4.075290160413643
Generative Surprise for type-5: 6.458610990238941
Generative Surprise for type-6: 4.6586460718737115
Generative Surprise for type-7: 6.168911263570949
Generative Surprise for type-8: 2.2838291882649155
Generative Surprise for type-9: 405.8509139060184
Generative Surprise for type-10: 4.590167410845922
Generative Surprise for type-11: 104.55614116177377
Generative Surprise for type-12: -2.697895351530641
Generative Surprise for type-13: 3.5997589892227975
Generative Surprise for type-14: 26.361778962203214
Generative Surprise for type-15: -2.6092547143287756
Generative Surprise for type-16: -1.8458440619593413
```

5. Rewrite the formula for "Rec. Surprise" using the idea introduced in "Gen. Surprise"

`Generative baseline` (p_g) represents the fraction of positive feedback $(+)$ given by a user. On the other hand, `receptive baseline` (p_r) the fraction of positive feedback $(+)$ received by a user. In this context, `surprise` represents the behavioural deviation of two users (A/B) from baseline w.r.t. another user (X).

The formula for `Generative Surprise` of context X is:

$$s_g(X) = \frac{k - \sum_{i=1}^{n} p_g(A_i)}{\sqrt{\sum_{i=1}^{n} p_g(A_i) * (1 - p_g(A_i))}}$$

Here,

- $p_g(A_i)$ represents generative baseline of A_i
- Context X represents all the (A, B, X) triads, i.e., $(A_1, B_1|X_1)$,..., $(A_n, B_n|X_n)$
- k is the number of triad instances where X closed with a plus edges

Receptive surprise is similar, just use $p_r(A_i)$. So, the formula for `Receptive Surprise` of context X is:

$$s_r(X) = \frac{k - \sum_{i=1}^{n} p_r(A_i)}{\sqrt{\sum_{i=1}^{n} p_r(A_i) * (1 - p_r(A_i))}}$$

Here, $p_r(A_i)$ represents receptive baseline of A_i.

6. Compute "Rec. Surprise" for all each of the 16 types.

In [172…

```
# G_slash_dir = nx.read_edgelist("data/tmp.txt", nodetype=int, data=(("sign", int),),

# store K value for each type
K_r = {1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6: 0, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13:

# store P_r value for each type
P_r = {1: 0.0, 2: 0.0, 3: 0.0, 4: 0.0, 5: 0.0, 6: 0.0, 7: 0.0, 8: 0.0, 9: 0.0, 10: 0.0

# store Denominator value for each type's S_g formula, needs to make square-root over
Denominator_r = {1: 0.0, 2: 0.0, 3: 0.0, 4: 0.0, 5: 0.0, 6: 0.0, 7: 0.0, 8: 0.0, 9: 0


"""
    Triad type matcher on graph "G_slash_dir".
    This function assumes there is a directed edge from node "a" to node "b".

    attributes
    ==========
    a            node-id
    b            node-id
    x            node-id
"""
def update_rec_dictionary(a, b, x, type_):
    if G_slash_dir[a][b]['sign'] == 1:
        K_r[type_] += 1

    in_deg = G_slash_dir.in_degree(a)
    pos_in_deg = len([(u,v) for u,v,e in G_slash_dir.in_edges(a, data=True) if e['sign
    # print("Node-id: {}, Pos-InEdges: {}".format(a, [(u,v) for u,v,e in G_slash_dir.
    # print("Pos-InDeg: {}, InDeg: {}".format(pos_in_deg, in_deg))

    if in_deg != 0:
        P_r[type_] += (pos_in_deg/float(in_deg))
        Denominator_r[type_] += ((pos_in_deg/float(in_deg)) * (1 - (pos_in_deg/float(i


def print_rec_surprise():
    for type_ in range(1, 17):
        rec_surp = 0.0
        if Denominator_r[type_] != 0.0:
            rec_surp = K_r[type_] - P_r[type_] / math.sqrt(Denominator_r[type_])
        # else:
            # print("Type-{} is not covered. K: {}, P_r: {}, Denominator: {}".format(

        print("Receptive Surprise for type-{}: {}".format(type_, rec_surp))


def calculate_rec_surprise():
    triad_gen = nx.all_triads(G_slash_dir)
    triad_nums = 0

    for triad in triad_gen:
        triad_nodes = list(triad.nodes())
        # print(type(triad_nodes))
        # print(triad_nodes)

        triad_nums += 1
        if triad_nums % 100000 == 0:
            print("Finished {} triads.".format(triad_nums))

        if triad_nums >= 10000000:
            break

        if G_slash_dir.has_edge(triad_nodes[0], triad_nodes[1]):
            types = triad_type_matcher(triad_nodes[0], triad_nodes[1], triad_nodes[2]
            # print("a: {} to b: {} got: {}".format(triad_nodes[0], triad_nodes[1], t
            for type_ in types:
                update_rec_dictionary(triad_nodes[0], triad_nodes[1], triad_nodes[2],

        if G_slash_dir.has_edge(triad_nodes[1], triad_nodes[0]):
            types = triad_type_matcher(triad_nodes[1], triad_nodes[0], triad_nodes[2]
            # print("a: {} to b: {} got: {}".format(triad_nodes[1], triad_nodes[0], t
            for type_ in types:
                update_rec_dictionary(triad_nodes[1], triad_nodes[0], triad_nodes[2],

        if G_slash_dir.has_edge(triad_nodes[1], triad_nodes[2]):
            types = triad_type_matcher(triad_nodes[1], triad_nodes[2], triad_nodes[0]
            # print("a: {} to b: {} got: {}".format(triad_nodes[1], triad_nodes[2], t
            for type_ in types:
                update_rec_dictionary(triad_nodes[1], triad_nodes[2], triad_nodes[0],

        if G_slash_dir.has_edge(triad_nodes[2], triad_nodes[1]):
            types = triad_type_matcher(triad_nodes[2], triad_nodes[1], triad_nodes[0]
            # print("a: {} to b: {} got: {}".format(triad_nodes[2], triad_nodes[1], t
            for type_ in types:
                update_rec_dictionary(triad_nodes[2], triad_nodes[1], triad_nodes[0],

        if G_slash_dir.has_edge(triad_nodes[0], triad_nodes[2]):
            types = triad_type_matcher(triad_nodes[0], triad_nodes[2], triad_nodes[1]
            # print("a: {} to b: {} got: {}".format(triad_nodes[0], triad_nodes[2], t
            for type_ in types:
                update_rec_dictionary(triad_nodes[0], triad_nodes[2], triad_nodes[1],

        if G_slash_dir.has_edge(triad_nodes[2], triad_nodes[0]):
            types = triad_type_matcher(triad_nodes[2], triad_nodes[0], triad_nodes[1]
            # print("a: {} to b: {} got: {}".format(triad_nodes[2], triad_nodes[0], t
            for type_ in types:
                update_rec_dictionary(triad_nodes[2], triad_nodes[0], triad_nodes[1],


calculate_rec_surprise()
print_rec_surprise()
```

```
Finished 100000 triads.
Finished 200000 triads.
Finished 300000 triads.
Finished 400000 triads.
Finished 500000 triads.
Finished 600000 triads.
Finished 700000 triads.
Finished 800000 triads.
Finished 900000 triads.
Finished 1000000 triads.
Finished 1100000 triads.
Finished 1200000 triads.
Finished 1300000 triads.
Finished 1400000 triads.
Finished 1500000 triads.
Finished 1600000 triads.
Finished 1700000 triads.
Finished 1800000 triads.
Finished 1900000 triads.
Finished 2000000 triads.
Finished 2100000 triads.
Finished 2200000 triads.
Finished 2300000 triads.
Finished 2400000 triads.
Finished 2500000 triads.
Finished 2600000 triads.
Finished 2700000 triads.
Finished 2800000 triads.
Finished 2900000 triads.
Finished 3000000 triads.
Finished 3100000 triads.
Finished 3200000 triads.
Finished 3300000 triads.
Finished 3400000 triads.
Finished 3500000 triads.
Finished 3600000 triads.
Finished 3700000 triads.
Finished 3800000 triads.
Finished 3900000 triads.
Finished 4000000 triads.
Finished 4100000 triads.
Finished 4200000 triads.
Finished 4300000 triads.
Finished 4400000 triads.
Finished 4500000 triads.
Finished 4600000 triads.
Finished 4700000 triads.
Finished 4800000 triads.
Finished 4900000 triads.
Finished 5000000 triads.
Finished 5100000 triads.
Finished 5200000 triads.
Finished 5300000 triads.
Finished 5400000 triads.
Finished 5500000 triads.
Finished 5600000 triads.
Finished 5700000 triads.
Finished 5800000 triads.
Finished 5900000 triads.
Finished 6000000 triads.
Finished 6100000 triads.
Finished 6200000 triads.
Finished 6300000 triads.
Finished 6400000 triads.
Finished 6500000 triads.
Finished 6600000 triads.
Finished 6700000 triads.
Finished 6800000 triads.
Finished 6900000 triads.
Finished 7000000 triads.
Finished 7100000 triads.
Finished 7200000 triads.
Finished 7300000 triads.
Finished 7400000 triads.
Finished 7500000 triads.
Finished 7600000 triads.
Finished 7700000 triads.
Finished 7800000 triads.
Finished 7900000 triads.
Finished 8000000 triads.
Finished 8100000 triads.
Finished 8200000 triads.
Finished 8300000 triads.
Finished 8400000 triads.
Finished 8500000 triads.
Finished 8600000 triads.
Finished 8700000 triads.
Finished 8800000 triads.
Finished 8900000 triads.
Finished 9000000 triads.
Finished 9100000 triads.
Finished 9200000 triads.
Finished 9300000 triads.
Finished 9400000 triads.
Finished 9500000 triads.
Finished 9600000 triads.
Finished 9700000 triads.
Finished 9800000 triads.
Finished 9900000 triads.
Finished 10000000 triads.
Receptive Surprise for type-1: 401.58089932026814
Receptive Surprise for type-2: -5.023151620791333
Receptive Surprise for type-3: 400.85028478746676
Receptive Surprise for type-4: -5.104071038418393
Receptive Surprise for type-5: 4.768974916383906
Receptive Surprise for type-6: 3.1388705471413596
Receptive Surprise for type-7: 5.64533625931791
Receptive Surprise for type-8: 0.5515864018597609
Receptive Surprise for type-9: 402.79645487980036
Receptive Surprise for type-10: 4.3128843698129025
Receptive Surprise for type-11: 102.2056085236446
Receptive Surprise for type-12: -2.169578575373137
Receptive Surprise for type-13: 5.950058176973219
Receptive Surprise for type-14: 26.76190072871994
Receptive Surprise for type-15: -0.8723142731789988
Receptive Surprise for type-16: -0.5875556839666229
```

2. The SIR Model of Disease Spreading [50 points]

In this question, we will explore how varying the set of initially infected nodes in the SIR model can affect how a contagion spreads through a network.

For the `2005 Graph Drawing` conference, a data set was provided of the IMDB movie database. We will use a reduced version of this dataset, which derived all actor-actor collaboration edges where the actors co-starred in at least 2 movies together between 1995 and 2004. The following files are in the `data` directory:

- imdb_actor_edges.tsv
- imdb_actors_keys.tsv

We will be comparing our results to two other null models, the `Erdos–Renyi` graph and the `Preferential Attachment` graph, with the same number of nodes and expected degree. The following files are in the `data` directory:

- SIR_erdos_renyi.txt
- SIR_preferential_attachment.txt

Under the SIR model, every node can be either `susceptible`, `infected`, or `recovered` and every node starts as either `susceptible` or `infected`. Every infected neighbor of a susceptible node infects the `susceptible node` with probability β, and `infected nodes` can recover with probability δ. `Recovered nodes` are no longer susceptible and cannot be infected again. In the problem statement, $Algorithm 1$ describes for pseudo-code of this process.

1. For a node with d neighbors, we need to find the probability of getting infected in a given round.
2. Need to implement the SIR model (described above) and run 100 simulations with $\beta = 0.05$ and $\delta = 0.5$ for each of the three graphs (e.g. imdb, erdos-renyi, and preferential attachment). Initialize the infected set with a single node chosen uniformly at random. Record the total percentage of nodes that became `infected` in each simulation. Note that a simulation ends when there are no more infected nodes; the total percentage of nodes that became infected at some point is thus the number of `recovered nodes` at the end of your simulation divided by the total number of nodes in the network.

Some simulations may die out very quickly as not able to create an epidemic in the netwotk. While others may become epidemics and infect a large proportion of the networks, and thus may need a longer simulation time. For all three graphs if the proportion of simulations that infected at least 50 of the network; we will consider these events as `epidemics`. To compare the likelihood of an epidemic starting across graphs, and more importantly, test whether or not the observed differences are actually significant, we will use pairwise `Chi-Square tests`. For each pair of networks, compute:

$scipy.stats.chi2contingency([[e_1, 100 - e_1], [e_2, 100 - e_2]])$

where e_1 is the number of trials where more than 50 were infected in $network 1$ and e_2 is the number of trials where more than 50 were infected in $network 2$. We need to report both the $\chi 2 - statistic$ and $p - values$. See the problem statement for details on interpreting the output of the function call.

Finally, we like to answer the following questions about the two synthetic networks:

- Does the $Erdos - Renyi$ graph appear to be more/less susceptible to epidemics than the $Preferential Aachment$ graph?
- In cases where an epidemic does take off, does $Erdos - Renyi$ graph appear to have higher/lower final percentage infected?
- Overall, which of these two networks seems to be more susceptible to the spread of disease?
- Give one good reason why we might expect to see these significant differences (or lack thereof) between $Erdos - Renyi$ and $Preferential Aachment$? (2–3 sentences).

For further analysis on different network, I highly encourage to first try with a smaller number of simulations and only run with 100 simulations once you are confident that this code works fine for your graph. Running 100 simulations is necessary to ensure statistical significance in some of the comparisons.

In [141…

```python
def SIR_simulation(G, beta, delta):
    # declaration of required data-structures
    # susceptible, infected, and recovered nodes
    S = set()
    I = set()
    R = set()

    # list of all the nodes
    nodes = []

    for n in list(G.nodes):
        nodes.append(int(n))
        S.add(n)

    num_nodes = len(nodes)

    # print(num_nodes)
    # print(nodes)

    # choosing the initial single infected node randomly
    initial_infected_idx = random.randint(0, num_nodes)
    initial_infected_node = nodes[initial_infected_idx]

    # initializing infected set
    I.add(initial_infected_node)

    # removing the initial infected node from the susceptible list
    S.remove(initial_infected_node)

    while len(I) > 0:
        # declaration of required data-structures
        S_ = set()
        I_ = set()
        J_ = set()
        R_ = set()

        for u in nodes:
            if u in S:
                for v in G.neighbors(u):
                    if v in I:
                        toss = random.random()

                        if toss <= beta:
                            S_.add(u)
                            I_.add(u)
                            break
            elif u in I:
                toss = random.random()
                if toss <= delta:
                    J_.add(u)
                    R_.add(u)

        S = S - S_
        I = (I | I_) - J_
        R = (R | R_)

    return len(R)/float(num_nodes)
```

In [142…

```python
import random
from random import randrange

random.seed()
beta = 0.05
delta = 0.5
```

In [179…

```python
G_imdb = nx.read_edgelist("data/imdb_actor_edges.tsv", nodetype=int, data=(('weight',

G_imdb_sir = []

for itr in range(100):
    infected_ratio = SIR_simulation(G_imdb, beta, delta)
    G_imdb_sir.append(infected_ratio)
    print("Iteration: {} infected ratio: {}".format(itr, infected_ratio))
```

```
Iteration: 0 infected ratio: 5.6892530010809584e-05
Iteration: 1 infected ratio: 0.6038573135347329
Iteration: 2 infected ratio: 0.6135859361665813
Iteration: 3 infected ratio: 0.00011378506002161917
Iteration: 4 infected ratio: 0.6191614041076406
Iteration: 5 infected ratio: 0.600216191614041
Iteration: 6 infected ratio: 0.5980542754736303
Iteration: 7 infected ratio: 0.6177959833873813
Iteration: 8 infected ratio: 5.6892530010809584e-05
Iteration: 9 infected ratio: 0.5995334812539114
Iteration: 10 infected ratio: 0.6078966831655004
Iteration: 11 infected ratio: 0.6094327814757923
Iteration: 12 infected ratio: 0.6051658417249816
Iteration: 13 infected ratio: 5.6892530010809584e-05
Iteration: 14 infected ratio: 5.6892530010809584e-05
Iteration: 15 infected ratio: 5.6892530010809584e-05
Iteration: 16 infected ratio: 0.6177390908573704
Iteration: 17 infected ratio: 0.6210388575979974
Iteration: 18 infected ratio: 5.6892530010809584e-05
Iteration: 19 infected ratio: 0.6100585993059111
Iteration: 20 infected ratio: 0.6192751891676623
Iteration: 21 infected ratio: 0.6097741366558571
Iteration: 22 infected ratio: 5.6892530010809584e-05
Iteration: 23 infected ratio: 5.6892530010809584e-05
Iteration: 24 infected ratio: 0.6135290436365706
Iteration: 25 infected ratio: 5.6892530010809584e-05
Iteration: 26 infected ratio: 0.6078966831655004
Iteration: 27 infected ratio: 5.6892530010809584e-05
Iteration: 28 infected ratio: 5.6892530010809584e-05
Iteration: 29 infected ratio: 0.613699721226603
Iteration: 30 infected ratio: 0.5991352335438357
Iteration: 31 infected ratio: 0.0005689253001080958
Iteration: 32 infected ratio: 0.6080673607555328
Iteration: 33 infected ratio: 5.6892530010809584e-05
Iteration: 34 infected ratio: 0.6064174773852193
Iteration: 35 infected ratio: 0.6084656084656085
Iteration: 36 infected ratio: 5.6892530010809584e-05
Iteration: 37 infected ratio: 5.6892530010809584e-05
Iteration: 38 infected ratio: 5.6892530010809584e-05
Iteration: 39 infected ratio: 0.00017067759003242875
Iteration: 40 infected ratio: 5.6892530010809584e-05
Iteration: 41 infected ratio: 5.6892530010809584e-05
Iteration: 42 infected ratio: 0.601126472094214
Iteration: 43 infected ratio: 0.6197872219377596
Iteration: 44 infected ratio: 0.6177959833873813
Iteration: 45 infected ratio: 0.00011378506002161917
Iteration: 46 infected ratio: 0.6212095351880298
Iteration: 47 infected ratio: 0.6063605848552085
Iteration: 48 infected ratio: 0.6061899072651761
Iteration: 49 infected ratio: 0.5990214484838141
Iteration: 50 infected ratio: 0.6329293963702566
Iteration: 51 infected ratio: 5.6892530010809584e-05
Iteration: 52 infected ratio: 0.6061330147351652
Iteration: 53 infected ratio: 0.6148375718268191
Iteration: 54 infected ratio: 0.5874722648916197
Iteration: 55 infected ratio: 0.5913409569323548
Iteration: 56 infected ratio: 0.6042555612448085
Iteration: 57 infected ratio: 5.6892530010809584e-05
Iteration: 58 infected ratio: 0.6157478523069921
Iteration: 59 infected ratio: 0.6098310291858678
Iteration: 60 infected ratio: 0.6148944643568299
Iteration: 61 infected ratio: 0.6226318484383001
Iteration: 62 infected ratio: 0.6005575467941059
Iteration: 63 infected ratio: 5.6892530010809584e-05
Iteration: 64 infected ratio: 5.6892530010809584e-05
Iteration: 65 infected ratio: 5.6892530010809584e-05
Iteration: 66 infected ratio: 0.00017067759003242875
Iteration: 67 infected ratio: 5.6892530010809584e-05
Iteration: 68 infected ratio: 0.010638903112021392
Iteration: 69 infected ratio: 5.6892530010809584e-05
Iteration: 70 infected ratio: 5.6892530010809584e-05
Iteration: 71 infected ratio: 5.6892530010809584e-05
Iteration: 72 infected ratio: 0.6028901405245491
Iteration: 73 infected ratio: 0.6014678272742788
Iteration: 74 infected ratio: 0.00011378506002161917
Iteration: 75 infected ratio: 5.6892530010809584e-05
Iteration: 76 infected ratio: 0.6010695795642033
Iteration: 77 infected ratio: 0.6114240200261706
Iteration: 78 infected ratio: 5.6892530010809584e-05
Iteration: 79 infected ratio: 0.6197303294077487
Iteration: 80 infected ratio: 0.6016953973943221
Iteration: 81 infected ratio: 5.6892530010809584e-05
Iteration: 82 infected ratio: 0.00017067759003242875
Iteration: 83 infected ratio: 5.6892530010809584e-05
Iteration: 84 infected ratio: 0.6116515901462138
Iteration: 85 infected ratio: 0.6261591852989703
Iteration: 86 infected ratio: 0.6131307959264949
Iteration: 87 infected ratio: 0.634294817090516
Iteration: 88 infected ratio: 0.6167150253171758
Iteration: 89 infected ratio: 5.6892530010809584e-05
Iteration: 90 infected ratio: 5.6892530010809584e-05
Iteration: 91 infected ratio: 5.6892530010809584e-05
Iteration: 92 infected ratio: 0.6094327814757923
Iteration: 93 infected ratio: 0.00022757012004323833
Iteration: 94 infected ratio: 0.6161461000170677
Iteration: 95 infected ratio: 0.6108550947260625
Iteration: 96 infected ratio: 0.616089207487057
Iteration: 97 infected ratio: 0.00017067759003242875
Iteration: 98 infected ratio: 0.6306536951698242
Iteration: 99 infected ratio: 0.01080958070205382
```

In [177…

```python
G_erdos = nx.read_edgelist("data/SIR_erdos_renyi.txt", nodetype=int, comments='#', cre

G_erdos_sir = []

for itr in range(100):
    infected_ratio = SIR_simulation(G_erdos, beta, delta)
    G_erdos_sir.append(infected_ratio)
    print("Iteration: {} infected ratio: {}".format(itr, infected_ratio))
```

```
Iteration: 0 infected ratio: 5.6892530010809584e-05
Iteration: 1 infected ratio: 5.6892530010809584e-05
Iteration: 2 infected ratio: 0.8082721738635718
Iteration: 3 infected ratio: 5.6892530010809584e-05
Iteration: 4 infected ratio: 0.8024122432724583
Iteration: 5 infected ratio: 0.8061671502531718
Iteration: 6 infected ratio: 5.6892530010809584e-05
Iteration: 7 infected ratio: 0.8063378278432042
Iteration: 8 infected ratio: 0.811173692894123
Iteration: 9 infected ratio: 0.7979746259316152
Iteration: 10 infected ratio: 5.6892530010809584e-05
Iteration: 11 infected ratio: 5.6892530010809584e-05
Iteration: 12 infected ratio: 5.6892530010809584e-05
Iteration: 13 infected ratio: 5.6892530010809584e-05
Iteration: 14 infected ratio: 0.8119701883142744
Iteration: 15 infected ratio: 0.8154406326449337
Iteration: 16 infected ratio: 5.6892530010809584e-05
Iteration: 17 infected ratio: 0.8140183193946635
Iteration: 18 infected ratio: 0.8072481083233771
Iteration: 19 infected ratio: 0.8129373613244582
Iteration: 20 infected ratio: 0.802184673152415
Iteration: 21 infected ratio: 0.8193093246856687
Iteration: 22 infected ratio: 0.8090117767537123
Iteration: 23 infected ratio: 0.809068669283723
Iteration: 24 infected ratio: 0.00011378506002161917
Iteration: 25 infected ratio: 5.6892530010809584e-05
Iteration: 26 infected ratio: 0.8098082721738635
Iteration: 27 infected ratio: 0.8046879444728907
Iteration: 28 infected ratio: 0.8120270808442851
Iteration: 29 infected ratio: 0.8137907492746203
Iteration: 30 infected ratio: 0.8013312852022529
Iteration: 31 infected ratio: 0.8100927348239176
Iteration: 32 infected ratio: 0.7991124765318314
Iteration: 33 infected ratio: 5.6892530010809584e-05
Iteration: 34 infected ratio: 0.8045172668828583
Iteration: 35 infected ratio: 5.6892530010809584e-05
Iteration: 36 infected ratio: 0.00028446265005404792
Iteration: 37 infected ratio: 5.6892530010809584e-05
Iteration: 38 infected ratio: 5.6892530010809584e-05
Iteration: 39 infected ratio: 0.8119132957842635
Iteration: 40 infected ratio: 5.6892530010809584e-05
Iteration: 41 infected ratio: 0.8082152813335609
Iteration: 42 infected ratio: 0.8081583888035501
Iteration: 43 infected ratio: 5.6892530010809584e-05
Iteration: 44 infected ratio: 0.8091255618137339
Iteration: 45 infected ratio: 0.8067929680832907
Iteration: 46 infected ratio: 0.8103771974739716
Iteration: 47 infected ratio: 0.8122546509643284
Iteration: 48 infected ratio: 0.8080446037435285
Iteration: 49 infected ratio: 0.8088979916936906
Iteration: 50 infected ratio: 0.00011378506002161917
Iteration: 51 infected ratio: 0.8068498606133014
Iteration: 52 infected ratio: 0.00011378506002161917
Iteration: 53 infected ratio: 0.8089548842237014
Iteration: 54 infected ratio: 0.8176025487853444
Iteration: 55 infected ratio: 0.8091824543437447
Iteration: 56 infected ratio: 0.8067929680832907
Iteration: 57 infected ratio: 0.8127666837344256
Iteration: 58 infected ratio: 0.8037207714627069
Iteration: 59 infected ratio: 0.812652898674404
Iteration: 60 infected ratio: 0.8084428514536042
Iteration: 61 infected ratio: 5.6892530010809584e-05
Iteration: 62 infected ratio: 0.8092393468737554
Iteration: 63 infected ratio: 0.8084997439836149
Iteration: 64 infected ratio: 0.8166353757751608
Iteration: 65 infected ratio: 0.8110030153040906
Iteration: 66 infected ratio: 0.8157250952949878
Iteration: 67 infected ratio: 0.00011378506002161917
Iteration: 68 infected ratio: 0.8061102577231609
Iteration: 69 infected ratio: 5.6892530010809584e-05
Iteration: 70 infected ratio: 0.8024122432724583
Iteration: 71 infected ratio: 0.8110030153040906
Iteration: 72 infected ratio: 5.6892530010809584e-05
Iteration: 73 infected ratio: 0.00017067759003242875
Iteration: 74 infected ratio: 5.6892530010809584e-05
Iteration: 75 infected ratio: 0.811458155544177
Iteration: 76 infected ratio: 0.8091824543437447
Iteration: 77 infected ratio: 0.8110599078341014
Iteration: 78 infected ratio: 0.8052568697729988
Iteration: 79 infected ratio: 0.8023553507424476
Iteration: 80 infected ratio: 0.800705467372134
Iteration: 81 infected ratio: 0.8058826876031178
Iteration: 82 infected ratio: 0.00011378506002161917
Iteration: 83 infected ratio: 0.8115150480741878
Iteration: 84 infected ratio: 0.8070774307333447
Iteration: 85 infected ratio: 5.6892530010809584e-05
Iteration: 86 infected ratio: 0.8100927348239176
Iteration: 87 infected ratio: 0.8063378278432042
Iteration: 88 infected ratio: 0.8078739261534961
Iteration: 89 infected ratio: 0.8101496273539285
Iteration: 90 infected ratio: 0.8120839733742959
Iteration: 91 infected ratio: 0.8078170336234852
Iteration: 92 infected ratio: 0.00022757012004323833
Iteration: 93 infected ratio: 0.807305000853388
Iteration: 94 infected ratio: 5.6892530010809584e-05
Iteration: 95 infected ratio: 0.8091824543437447
Iteration: 96 infected ratio: 5.6892530010809584e-05
Iteration: 97 infected ratio: 0.8083859589235933
Iteration: 98 infected ratio: 0.00017067759003242875
Iteration: 99 infected ratio: 0.815952665415031
```

In [178…

```python
G_pref = nx.read_edgelist("data/SIR_preferential_attachment.txt", nodetype=int, commen

G_pref_sir = []

for itr in range(100):
    infected_ratio = SIR_simulation(G_pref, beta, delta)
    G_pref_sir.append(infected_ratio)
    print("Iteration: {} infected ratio: {}".format(itr, infected_ratio))
```

```
Iteration: 0 infected ratio: 0.7410820959208057
Iteration: 1 infected ratio: 0.739773567730557
Iteration: 2 infected ratio: 0.7290777720885248
Iteration: 3 infected ratio: 0.7425044091710759
Iteration: 4 infected ratio: 5.6892530010809584e-05
Iteration: 5 infected ratio: 0.7423337315810434
Iteration: 6 infected ratio: 0.7415372361608921
Iteration: 7 infected ratio: 5.6892530010809584e-05
Iteration: 8 infected ratio: 0.744552540251465
Iteration: 9 infected ratio: 0.7429595494111623
Iteration: 10 infected ratio: 0.7311259031689139
Iteration: 11 infected ratio: 0.7400011378506002
Iteration: 12 infected ratio: 0.7425613017010867
Iteration: 13 infected ratio: 5.6892530010809584e-05
Iteration: 14 infected ratio: 0.7396597826705353
Iteration: 15 infected ratio: 0.7384650395403084
Iteration: 16 infected ratio: 0.738863287250384
Iteration: 17 infected ratio: 0.7431871195312055
Iteration: 18 infected ratio: 0.7393184274904705
Iteration: 19 infected ratio: 5.6892530010809584e-05
Iteration: 20 infected ratio: 5.6892530010809584e-05
Iteration: 21 infected ratio: 5.6892530010809584e-05
Iteration: 22 infected ratio: 0.7402287079706434
Iteration: 23 infected ratio: 0.7414803436308812
Iteration: 24 infected ratio: 0.738294361950276
Iteration: 25 infected ratio: 0.7427888718211298
Iteration: 26 infected ratio: 0.7401718154406326
Iteration: 27 infected ratio: 5.6892530010809584e-05
Iteration: 28 infected ratio: 0.736189338339876
Iteration: 29 infected ratio: 0.7415372361608921
Iteration: 30 infected ratio: 5.6892530010809584e-05
Iteration: 31 infected ratio: 0.7507538260226432
Iteration: 32 infected ratio: 0.7340274221994653
Iteration: 33 infected ratio: 5.6892530010809584e-05
Iteration: 34 infected ratio: 0.7426750867611083
Iteration: 35 infected ratio: 0.749047050122319
Iteration: 36 infected ratio: 5.6892530010809584e-05
Iteration: 37 infected ratio: 5.6892530010809584e-05
Iteration: 38 infected ratio: 0.7320930761790977
Iteration: 39 infected ratio: 0.7485350173522216
Iteration: 40 infected ratio: 0.7395459976105138
Iteration: 41 infected ratio: 0.7463162086818
Iteration: 42 infected ratio: 0.7427319792911191
Iteration: 43 infected ratio: 0.7410820959208057
Iteration: 44 infected ratio: 0.7477385219320704
Iteration: 45 infected ratio: 0.7356773055697787
Iteration: 46 infected ratio: 0.7521192467429026
Iteration: 47 infected ratio: 0.7439836149513569
Iteration: 48 infected ratio: 0.7370427262900381
Iteration: 49 infected ratio: 5.6892530010809584e-05
Iteration: 50 infected ratio: 5.6892530010809584e-05
Iteration: 51 infected ratio: 5.6892530010809584e-05
Iteration: 52 infected ratio: 0.7403993855606759
Iteration: 53 infected ratio: 5.6892530010809584e-05
Iteration: 54 infected ratio: 0.7426750867611083
Iteration: 55 infected ratio: 0.742163053991011
Iteration: 56 infected ratio: 0.7475678443420379
Iteration: 57 infected ratio: 0.7416510212209136
Iteration: 58 infected ratio: 0.7356773055697787
Iteration: 59 infected ratio: 5.6892530010809584e-05
Iteration: 60 infected ratio: 0.7471695966319623
Iteration: 61 infected ratio: 0.743073334471184
Iteration: 62 infected ratio: 0.7380098993002219
Iteration: 63 infected ratio: 0.7468282414518974
Iteration: 64 infected ratio: 0.743926722421346
Iteration: 65 infected ratio: 0.7494452978323946
Iteration: 66 infected ratio: 0.7440974000113785
Iteration: 67 infected ratio: 0.7389770723104057
Iteration: 68 infected ratio: 5.6892530010809584e-05
Iteration: 69 infected ratio: 5.6892530010809584e-05
Iteration: 70 infected ratio: 0.7434146896512488
Iteration: 71 infected ratio: 0.7419354838709677
Iteration: 72 infected ratio: 0.7327188940092166
Iteration: 73 infected ratio: 0.7451214655515731
Iteration: 74 infected ratio: 0.7418216988109462
Iteration: 75 infected ratio: 5.6892530010809584e-05
Iteration: 76 infected ratio: 5.6892530010809584e-05
Iteration: 77 infected ratio: 0.7468282414518974
Iteration: 78 infected ratio: 0.7311259031689139
Iteration: 79 infected ratio: 0.7533139898731297
Iteration: 80 infected ratio: 5.6892530010809584e-05
Iteration: 81 infected ratio: 0.7355066279797463
Iteration: 82 infected ratio: 0.7374978665301246
Iteration: 83 infected ratio: 5.6892530010809584e-05
Iteration: 84 infected ratio: 0.7411389884508164
Iteration: 85 infected ratio: 0.7463731012118109
Iteration: 86 infected ratio: 5.6892530010809584e-05
Iteration: 87 infected ratio: 0.7391477499004381
Iteration: 88 infected ratio: 5.6892530010809584e-05
Iteration: 89 infected ratio: 0.7438129373613245
Iteration: 90 infected ratio: 5.6892530010809584e-05
Iteration: 91 infected ratio: 0.7378392217101895
Iteration: 92 infected ratio: 5.6892530010809584e-05
Iteration: 93 infected ratio: 0.7412527735108381
Iteration: 94 infected ratio: 0.7418216988109462
Iteration: 95 infected ratio: 5.6892530010809584e-05
Iteration: 96 infected ratio: 0.7360186607498436
Iteration: 97 infected ratio: 0.7420492689309893
Iteration: 98 infected ratio: 0.7406269556807191
Iteration: 99 infected ratio: 0.7446663253114866
```

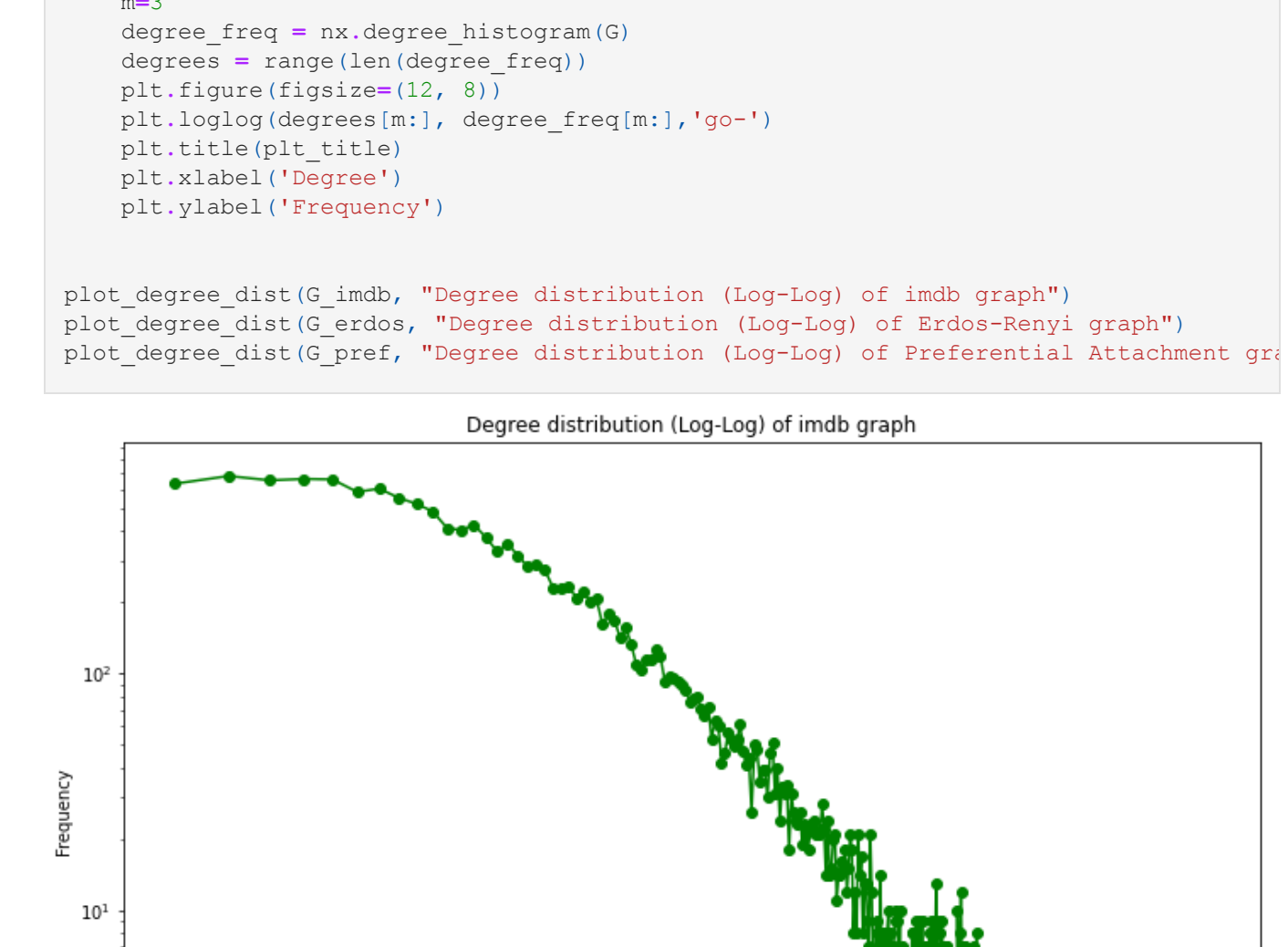
In [189…

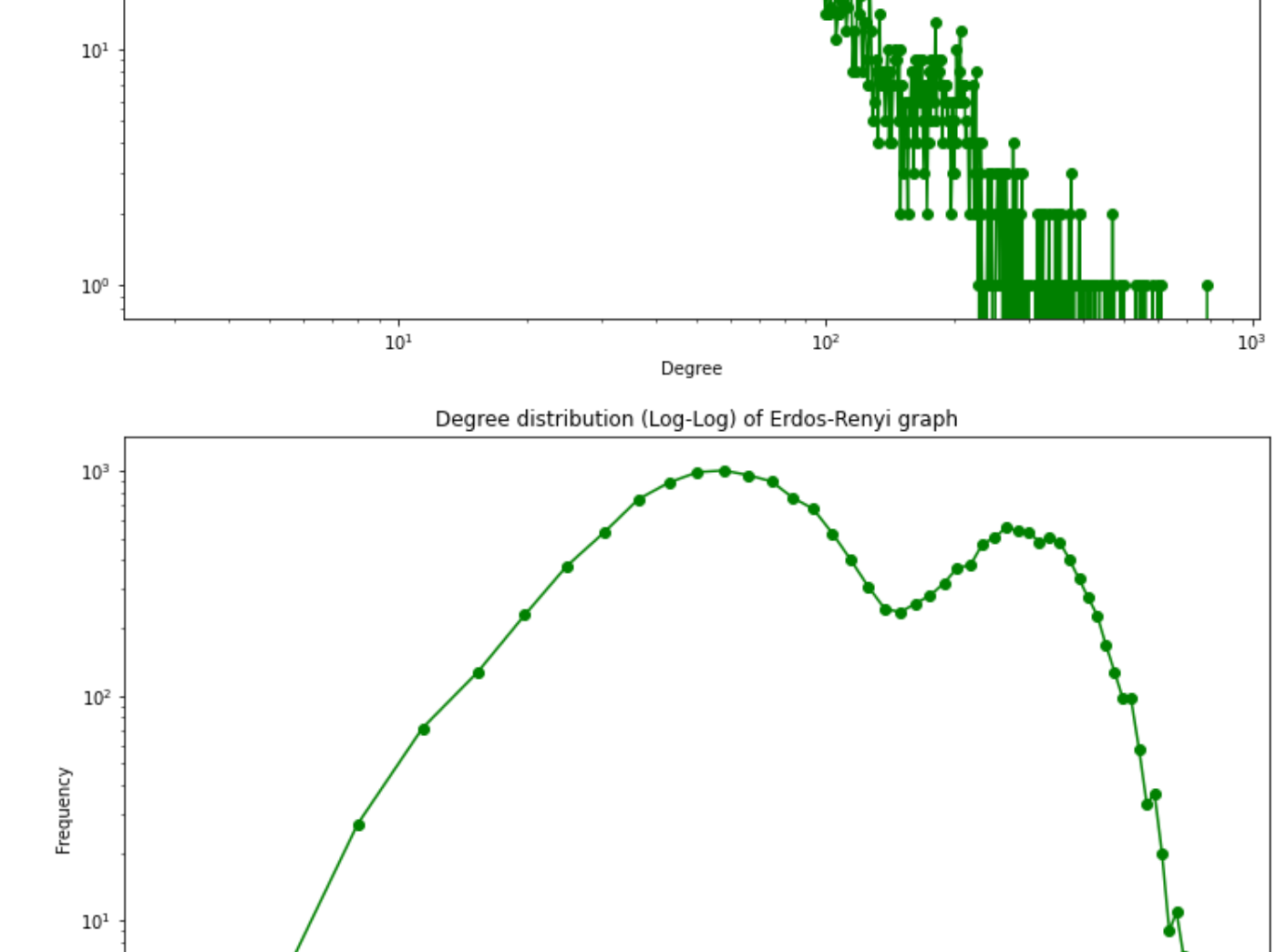
```python
# print(G_erdos_sir)
# print(G_pref_sir)
# print(G_imdb_sir)

import scipy
from scipy.stats import chi2_contingency
# scipy.__version__

e_imdb = len([i for i in G_imdb_sir if i >= 0.5])
e_erdos = len([i for i in G_erdos_sir if i >= 0.5])
e_pref = len([i for i in G_pref_sir if i >= 0.5])

print("e_imdb: {}".format(e_imdb))
print("e_erdos: {}".format(e_erdos))
print("e_pref: {}".format(e_pref))

imdb_vs_erdos_obs = np.array([[e_imdb, (100-e_imdb)], [e_erdos, (100-e_erdos)]])
imdb_vs_erdos_contingent = chi2_contingency(imdb_vs_erdos_obs)

print("imdb Vs. erdos contingent: {}".format(imdb_vs_erdos_contingent))

imdb_vs_pref_obs = np.array([[e_imdb, (100-e_imdb)], [e_pref, (100-e_pref)]])
imdb_vs_pref_contingent = chi2_contingency(imdb_vs_pref_obs)

print("imdb Vs. pref contingent: {}".format(imdb_vs_pref_contingent))

erdos_vs_pref_obs = np.array([[e_erdos, (100-e_erdos)], [e_pref, (100-e_pref)]])
erdos_vs_pref_contingent = chi2_contingency(erdos_vs_pref_obs)

print("erdos Vs. pref contingent: {}".format(erdos_vs_pref_contingent))
```

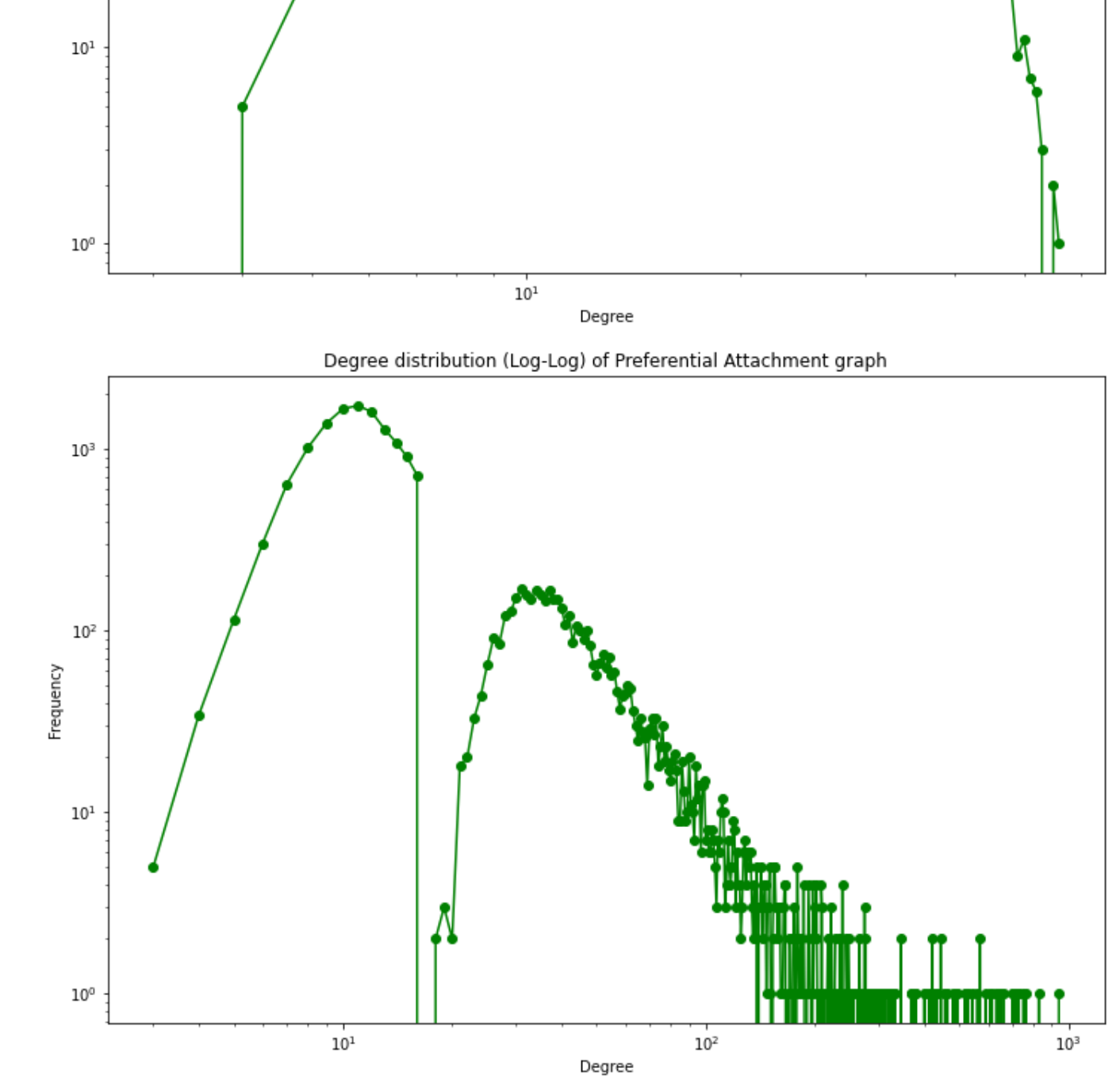
```
e_imdb: 56
e_erdos: 68
e_pref: 73
imdb Vs. erdos contingent: (2.5679117147707977, 0.10905160508448167, 1, array([[62., 3
8.],
       [62., 38.]]))
imdb Vs. pref contingent: (5.59012992684791, 0.01806195638877191, 1, array([[64.5, 35.
5],
       [64.5, 35.5]]))
erdos Vs. pref contingent: (0.38466161798293064, 0.5351189157048366, 1, array([[70.5,
29.5],
       [70.5, 29.5]]))
```

In [199…

```python
def plot_degree_dist(G, plt_title):
    m=3
    degree_freq = nx.degree_histogram(G)
    degrees = range(len(degree_freq))
    plt.figure(figsize=(12, 8))
    plt.loglog(degrees[m:], degree_freq[m:],'go-')
    plt.title(plt_title)
    plt.xlabel('Degree')
    plt.ylabel('Frequency')


plot_degree_dist(G_imdb, "Degree distribution (Log-Log) of imdb graph")
plot_degree_dist(G_erdos, "Degree distribution (Log-Log) of Erdos-Renyi graph")
plot_degree_dist(G_pref, "Degree distribution (Log-Log) of Preferential Attachment gra
```

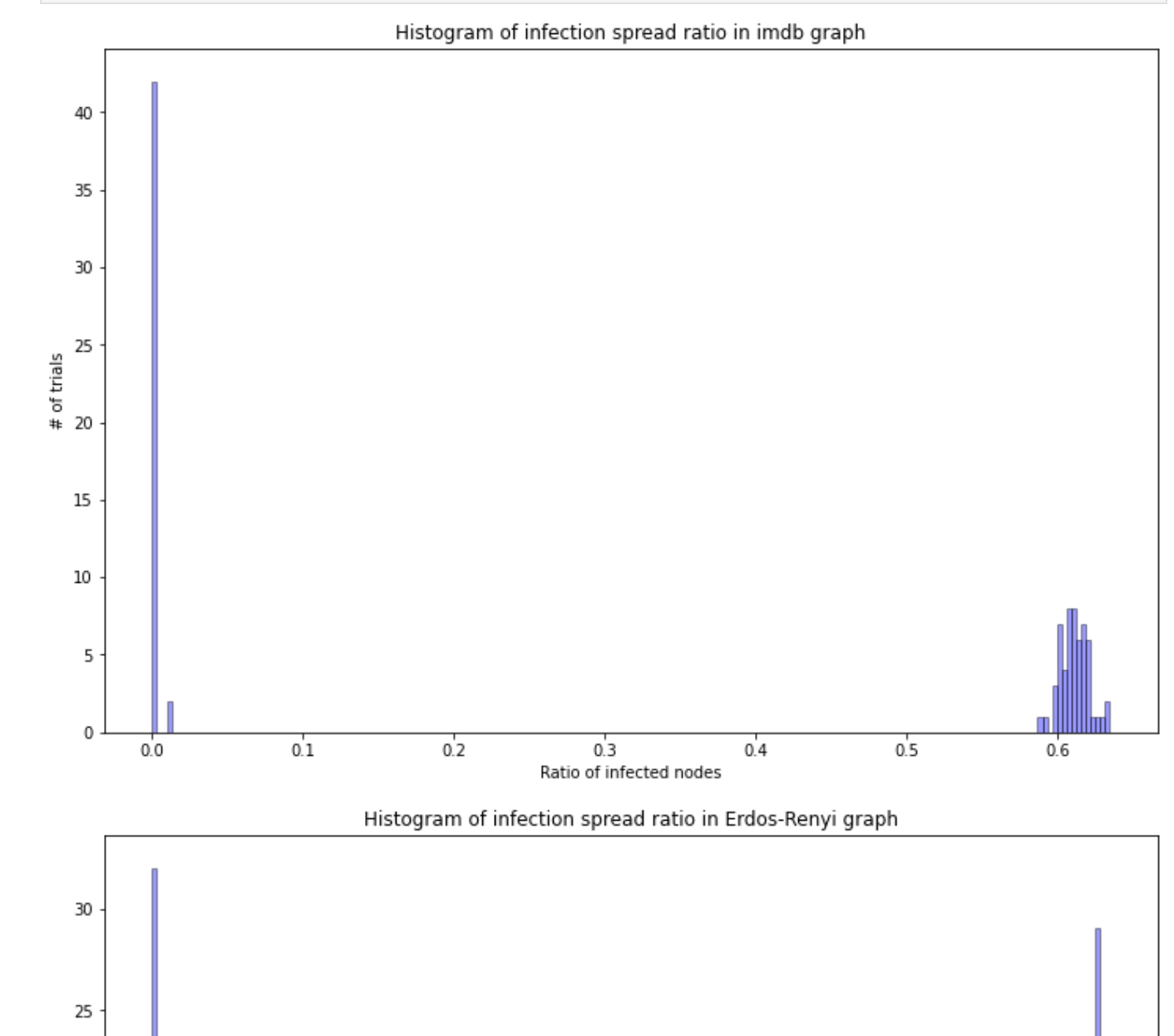


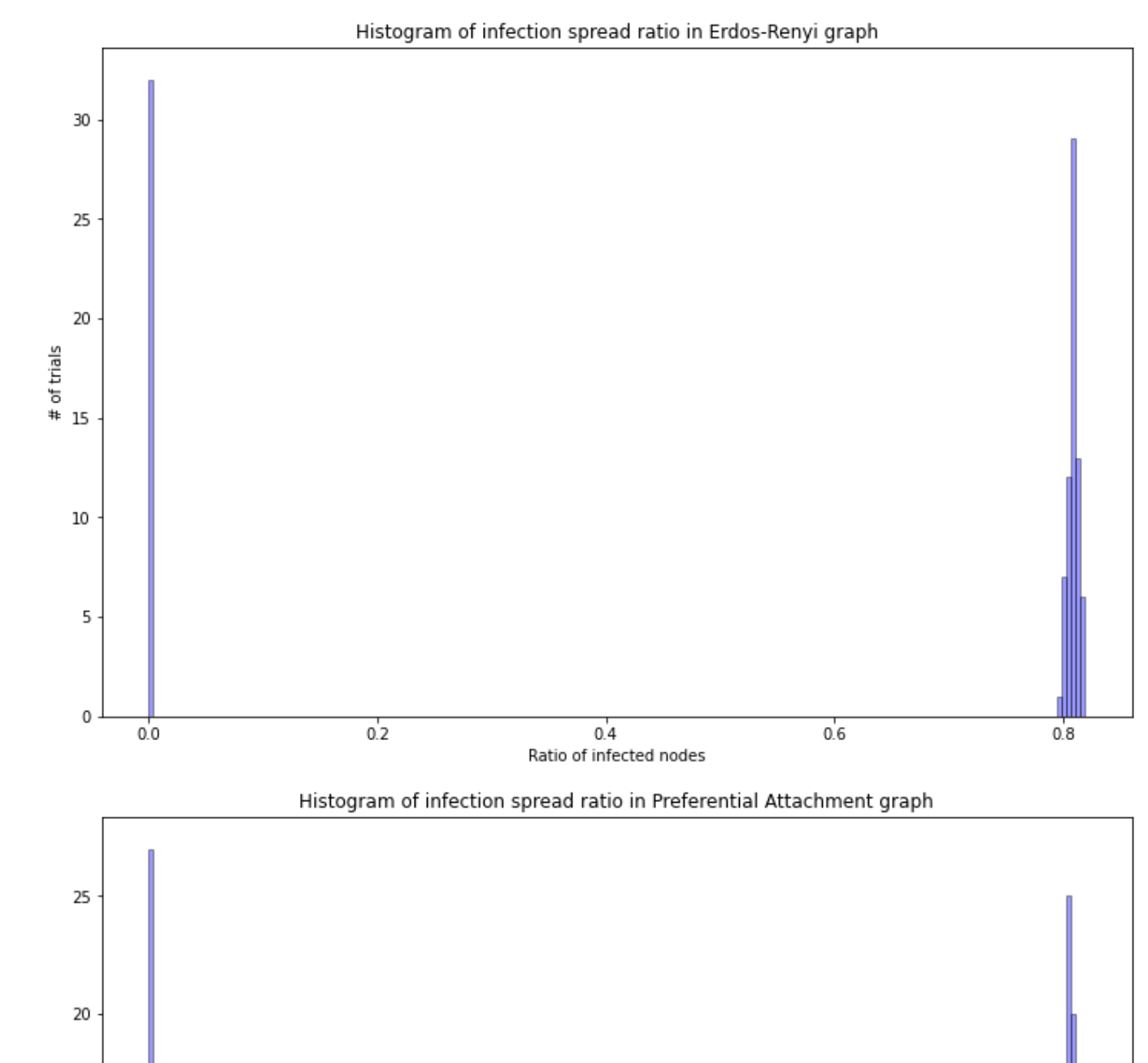




In [210…

```python
import seaborn as sns

def plot_infection_spread(sir_data, plt_title):
    plt.figure(figsize=(12, 8))
    # seaborn histogram
    sns.distplot(sir_data, hist=True, kde=False,
                 bins=int(1000/5), color = 'blue',
                 hist_kws={'edgecolor':'black'})
#     # matplotlib histogram
#     plt.hist(sir_data, color = 'blue', edgecolor = 'black',
#              bins = int(100/5))
    # Add labels
    plt.title(plt_title)
    plt.xlabel('Ratio of infected nodes')
    plt.ylabel('# of trials')

plot_infection_spread(G_imdb_sir, 'Histogram of infection spread ratio in imdb graph')
plot_infection_spread(G_erdos_sir, 'Histogram of infection spread ratio in Erdos-Renyi
plot_infection_spread(G_pref_sir, 'Histogram of infection spread ratio in Preferentia
```





In [212…

```python
cc_imdb = nx.number_connected_components(G_imdb)
cc_erdos = nx.number_connected_components(G_erdos)
cc_pref = nx.number_connected_components(G_pref)

print("Connected components in imdb graph: {}".format(cc_imdb))
print("Connected components in Erdos-Renyi graph: {}".format(cc_erdos))
print("Connected components in Preferential Attachment graph: {}".format(cc_pref))
```

```
Connected components in imdb graph: 19
Connected components in Erdos-Renyi graph: 1
Connected components in Preferential Attachment graph: 1
```

To further analyzing the SIR model results and the susceptibility to epidemics, we further plotted the degree distribution and infection spread ratio of the three network. Let's first plot the occurance of epidemics (more than 50% node infected) in the networks within the 100 simulations:

Graph	# of epidemics (in 100 simulations)
imdb	56
Erdos-Renyi	68
Preferential Attachment	73

Table 4: Number of epidemics (in 100 simulations)

To compare the likelihood of an epidemic starting across graphs, and more importantly, to understand whether or not the observed differences are actually significant, we used pairwise Chi-Square tests. Here is the χ^2 (i.e., test statistic) and p-value result we observed from this test:

e_1	e_2	χ^2 (test statistic)	p-value
imdb	Erdos-Renyi	2.57	0.10
imdb	Preferential Attachment	5.59	0.02
Erdos-Renyi	Preferential Attachment	0.38	0.54

Table 5: chi2 contingency results

Here the χ^2 (i.e., `test statistic`) resembles a normalized sum of squared deviations of between two comparing network property (in our case, we considered the susceptibility to epidemics). The `p-value` is the probability of obtaining test results at least as extreme as the results actually observed. During this, it makes the assumption that the null hypothesis is correct.

- Does the $Erdos - Renyi$ graph appear to be more/less susceptible to epidemics than the $Preferential Aachment$ graph?
  - **Answer:** From `Table-4` we can see, `Erdos-Renyi` graph appear to be less susceptible to epidemics than the `preferential attachment` graph. This is because, the `preferential attachment` graph build in a way so that the nodes with higher degree have higher probability to be attached to more nodes, thus, causing higher spreading of the contagion. We can observe this from the degree distribution we plotted earlier. However, there is no significant difference in epidemic likelihood.
- In cases where an epidemic does take off, does $Erdos - Renyi$ graph appear to have higher/lower final percentage infected?
  - **Answer:** From the histogram of infection spread ratio in Erdos-Renyi graph, we can see it appears to have higher final percentage infected node in cases where an epidemic does take off.
- Overall, which of these two networks seems to be more susceptible to the spread of disease?
  - **Answer:** From the degree distribution plot, we can observe that the `preferential attachment` graph have a heavy tail, meaning it have more higher degree nodes comparing to `Erdos-Renyi` graph. Besides this, both of the graph have a single connected component. This implies, if a node is infected in the `preferential attachment` graph, it have a higher probability that it will infect the neighboring nodes quickly.
- Give one good reason why we might expect to see these significant differences (or lack thereof) between $Erdos - Renyi$ and $Preferential Aachment$? (2–3 sentences).
  - **Answer:** Both of the `Erdos-Renyi` and `Preferential Attachment` have lots of similarities that we can see from the degree distribution and the histogram plotting of infection spread ratio. It is worth observing that, while comparing the with the `imdb` graph through Chi-Square tests, it seems `Erdos-Renyi` graph is more susceptible to the spread of disease (as it gives lower χ^2 with higher `p-value`). To compare the likelihood of an epidemic between `Erdos-Renyi` and `Preferential Attachment` graph, we further used pairwise Chi-Square tests in between them. From that test we got higher likelihood of an epidemic in between them.

Conclusion

In the first part of this assignment, we have explored the `Signed` network. For `signed network`, we did experiment with Slashdot dataset and calculate the frequency of signed triads in both "real" and "shuffled" network to understand whether it hold structural balance. Later we explored `generative` and `receptive` surprise and computed those metric on the same dataset. It helped us to understand the deviation of users behavior from the baseline in context X.

In the second part of this assignment, we have explored the `Disease Spreading` network and implemented the `SIR Model` to understand the likelihood of an epidemic in a given network. For this problem, we did experiment with three datasets, e.g. imdb, erdos-renyi, and preferential-attachment. We have done the pairwise `Chi-Square test` to compare the likelihood of an epidemic starting across graphs.

References

[1] Erdős, Paul, and Alfréd Rényi. 1960. “On the Evolution of Random Graphs.” Bull. Inst. Internat. Statis. 38 (4): 343–47.

[2] NetworkX, “Software for Complex Networks,” https://networkx.github.io/documentation/stable/, 2020, accessed: 2020-10.

[3] NetworkX, “Install NetworkX Documentation,” https://networkx.github.io/documentation/stable/install.html, 2020, accessed: 2020-10.