Final Exam

Section 1

Please read `socfb-Northwestern25.edges.gz` and answer the following questions.

- **Q1.** Plot the degree distribution in double logarithmic scale
- **Q2.** Compute the heterogeneity parameter
- **Q3.** Plot the distribution of the clustering coefficients of the nodes
- **Q4.** Plot the knn as a function of k
- **Q5.** Compute the robustness plot by removing nodes at random (failure) and in decreasing order of degree (attack). Put both lines in the same plot.

In [603]:

```
import networkx as nx
G = nx.read_edgelist('socfb-Northwestern25.edgelist')
```

In [604]:

```
len(G.edges())
```

Out[604]:

```
488337
```

In [605]:

```
len(G.nodes())
```

Out[605]:

```
10567
```

In [606]:

```
nx.average_clustering(G)
```

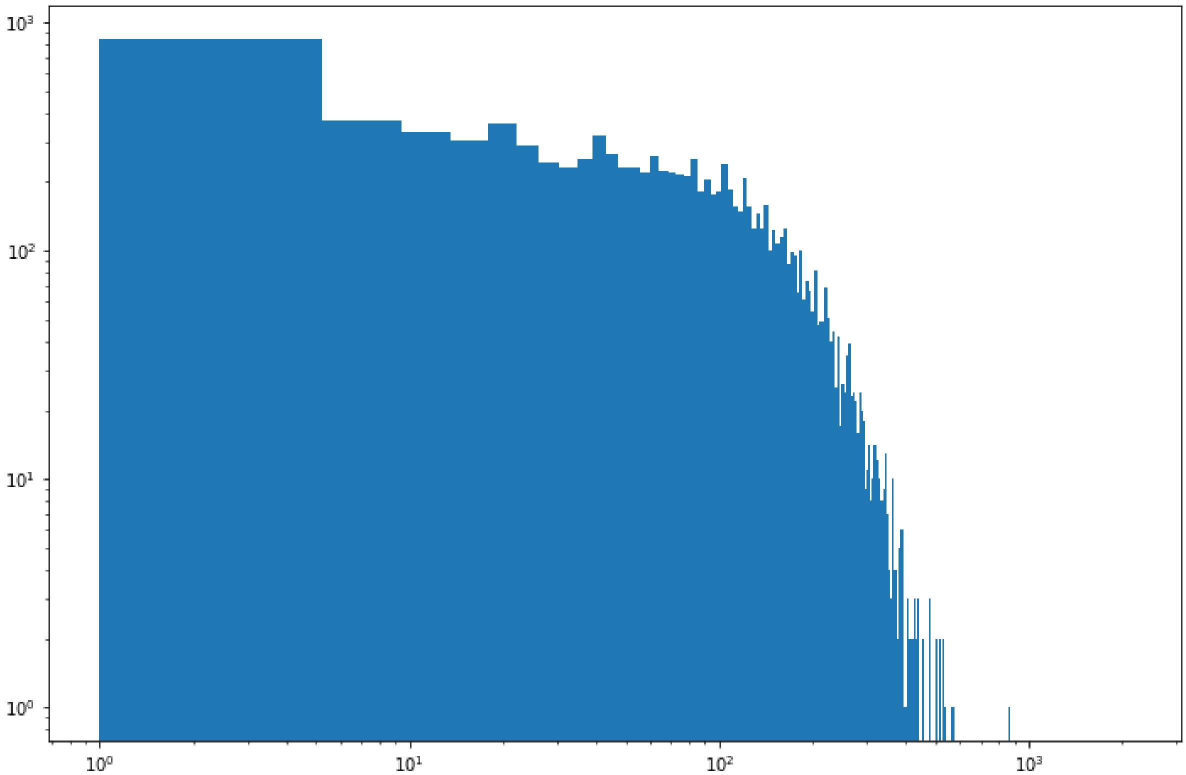
Out[606]:

```
0.2379913948280604
```

Q1

In [607]:

```
import matplotlib.pyplot as plt

degree_sequence = [G.degree(n) for n in G.nodes]
plt.figure(figsize=(12, 8))
plt.hist(degree_sequence,bins=500)
plt.xscale('log')
plt.yscale('log')
```



In [608]:

```
from scipy import stats

stats.describe(degree_sequence)
```

Out[608]:

```
DescribeResult(nobs=10567, minmax=(1, 2105), mean=92.42680041639065, variance
=7140.918335237751, skewness=3.43109618834199, kurtosis=52.60765891928037)
```

Q2

In [609]:

```
import numpy as np

nodes=len(G.nodes())
edges=len(G.edges())

summation=0
for n in G.nodes():
    summation+=G.degree(n)**2
denom=((2*edges)/nodes)**2
numerator=summation/nodes
parameter=numerator/denom
hetero_param = numerator/denom
```

In [610]:

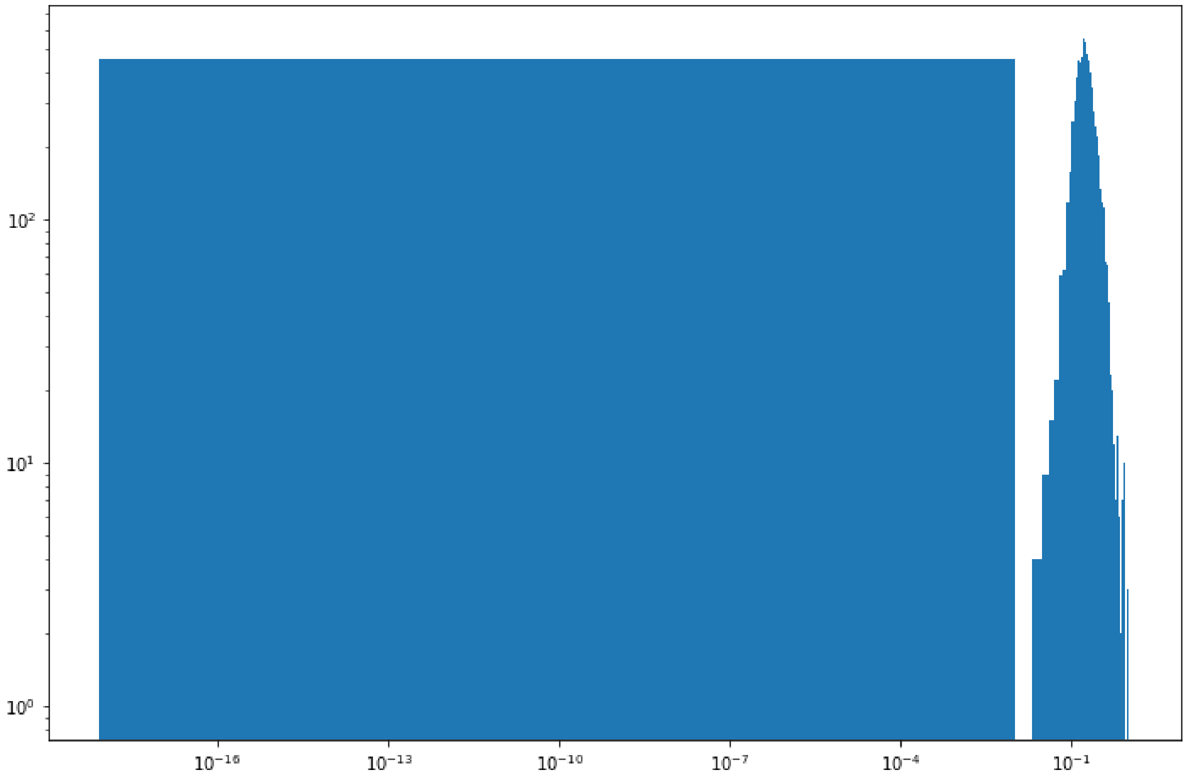
```
print(f"The heterogeneity parameter for this network is {hetero_param}")
```

```
The heterogeneity parameter for this network is 1.8358284067492063
```

Q3

In [611]:

```
cluster_sequence = [nx.clustering(G, n) for n in G.nodes]
plt.figure(figsize=(12, 8))
plt.hist(cluster_sequence,bins=100)
plt.xscale('log')
plt.yscale('log')
```



In [612]:

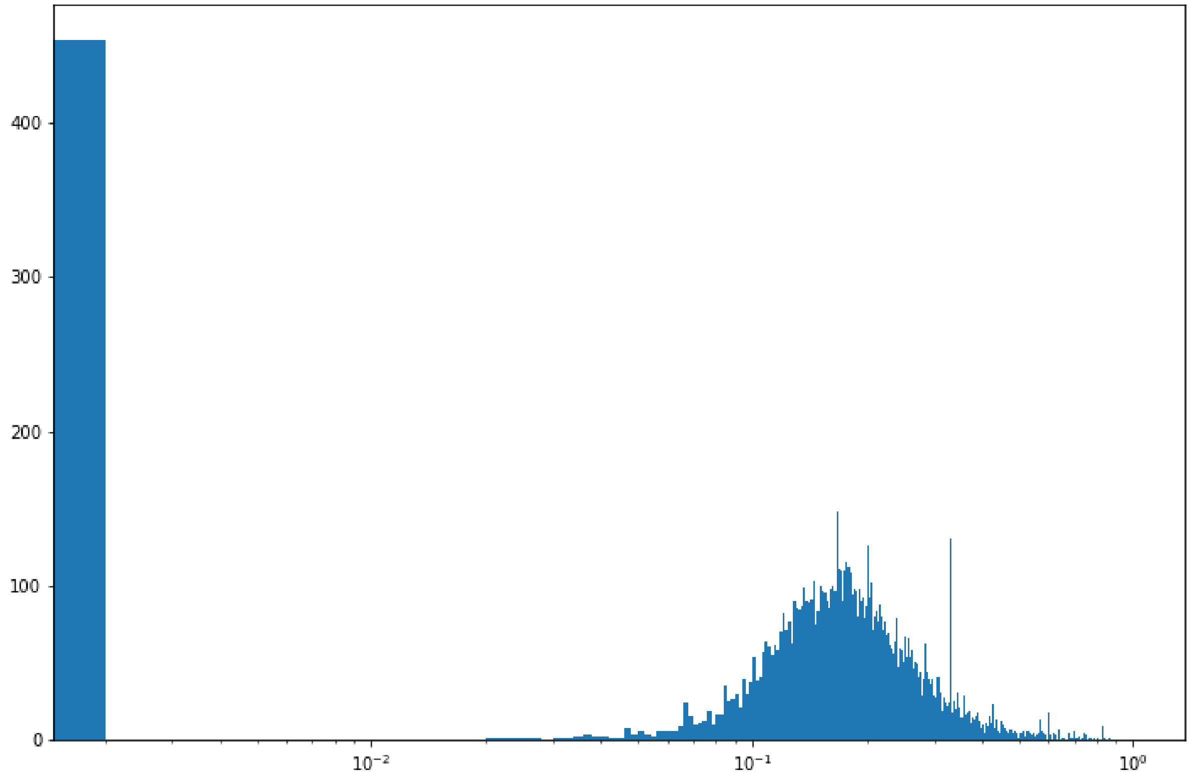
```
len(cluster_sequence)
```

Out[612]:

10567

In [613]:

```
plt.figure(figsize=(12, 8))
plt.hist(cluster_sequence,bins=500)
plt.xscale('log')
#plt.yscale('log')
```



Q4

In [614]:

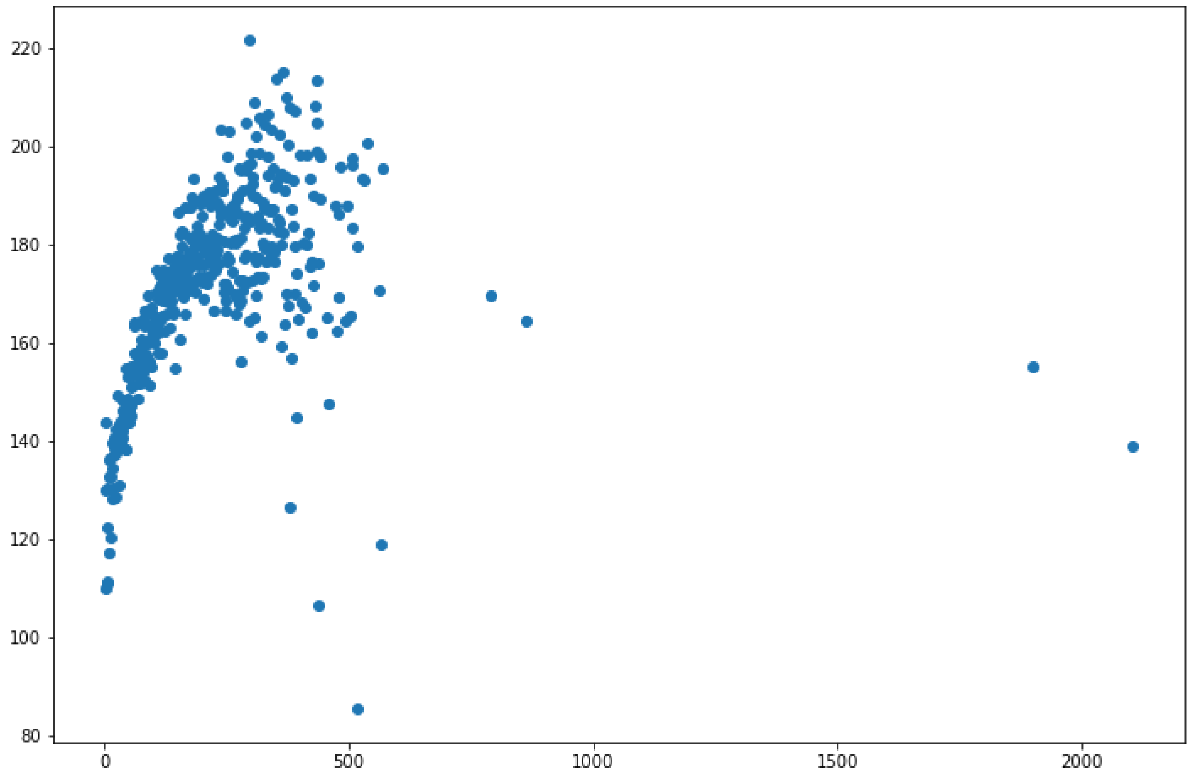
```
import  scipy.stats
knn_dict = nx.k_nearest_neighbors(G)
k, knn = list(knn_dict.keys()), list(knn_dict.values ())
```

In [615]:

```
plt.figure(figsize=(12, 8))
plt.scatter(k, knn)
```

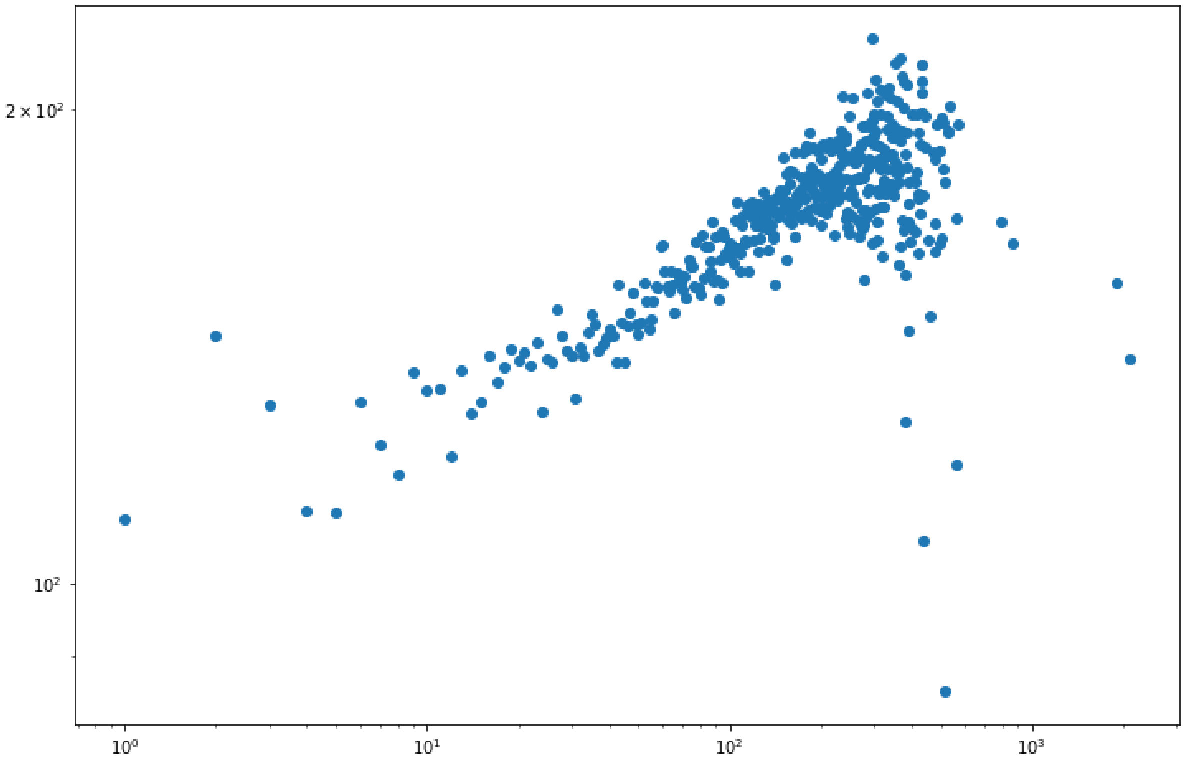
Out[615]:

```
<matplotlib.collections.PathCollection at 0x7f0bff48f518>
```



In [616]:

```
plt.figure(figsize=(12, 8))
plt.scatter(k, knn)
plt.xscale('log')
plt.yscale('log')
```



Q5

In [617]:

```
# Compute the robustness plot by removing nodes at random (failure) and in decreasing orde
r of degree (attack). Put both lines in the same plot.

import random

core = next(nx.connected_components(G))
C = G.copy()

nodes_to_remove = random.sample(list(C.nodes), 2)
C.remove_nodes_from(nodes_to_remove)

number_of_steps = 25
M = G.number_of_nodes() // number_of_steps

num_nodes_removed = range(0, G.number_of_nodes(), M)

N = G.number_of_nodes()
C = G.copy()
random_attack_core_proportions = []

for nodes_removed in num_nodes_removed:
# Measure the relative size of the network core
    core = next(nx.connected_components(C))
    core_proportion = len(core) / N
    random_attack_core_proportions.append(core_proportion)


#If there are more than M nodes, select M nodes at random and remove them
    if C.number_of_nodes() > M:
        nodes_to_remove = random.sample(list(C.nodes), M)
        C.remove_nodes_from(nodes_to_remove)
```

In [618]:

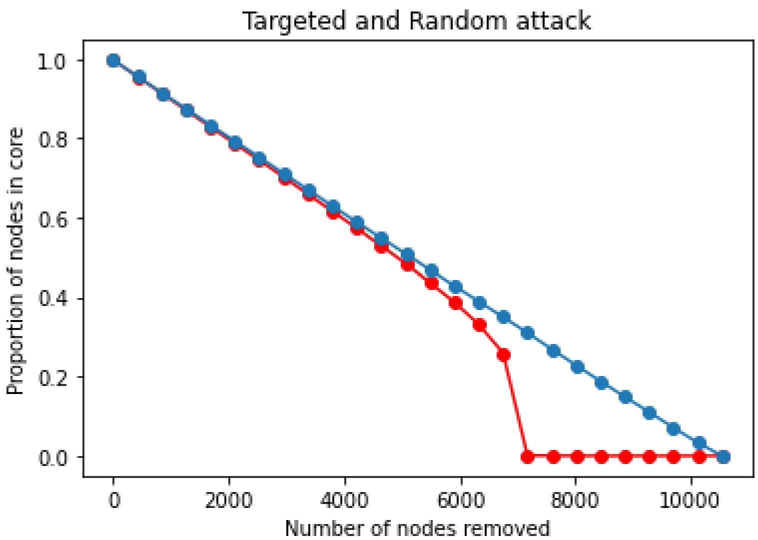
```
num_nodes_removed = range(0, N, M)

C = G.copy()

degree_targeted_attack_core_proportions = []

for nodes_removed in num_nodes_removed:
    # Measure the relative size of the network core
    core = next(nx.connected_components(C))
    core_proportion = len(core) / N
    degree_targeted_attack_core_proportions.append(core_proportion)

 # If there are more than M nodes, select top M nodes and remove them
    if C.number_of_nodes() > M:
        nodes_sorted_by_degree = sorted(C.nodes, key=C.degree, reverse=True)
        nodes_to_remove = nodes_sorted_by_degree[:M]
        C.remove_nodes_from(nodes_to_remove)
```

In [619]:

```
plt.title('Targeted and Random attack')
plt.xlabel('Number of nodes removed')
plt.ylabel('Proportion of nodes in core')
plt.plot(num_nodes_removed, degree_targeted_attack_core_proportions, marker='o',color='re
d')
plt.plot(num_nodes_removed, random_attack_core_proportions, marker='o')
```

Out[619]:

```
[<matplotlib.lines.Line2D at 0x7f0bfe5a8080>]
```



For the following questions please create a randomization of the `socfb-Northwestern25.edges.gz` network that preserves the nodes' degree sequence, by using the **configuration model**.

- **Q6**. Verify that the degree distribution is the same as the one of the original network above
- **Q7**. Plot the knn as a function of k, put it in the same diagram with the one of the original graph. What can you say about the difference?

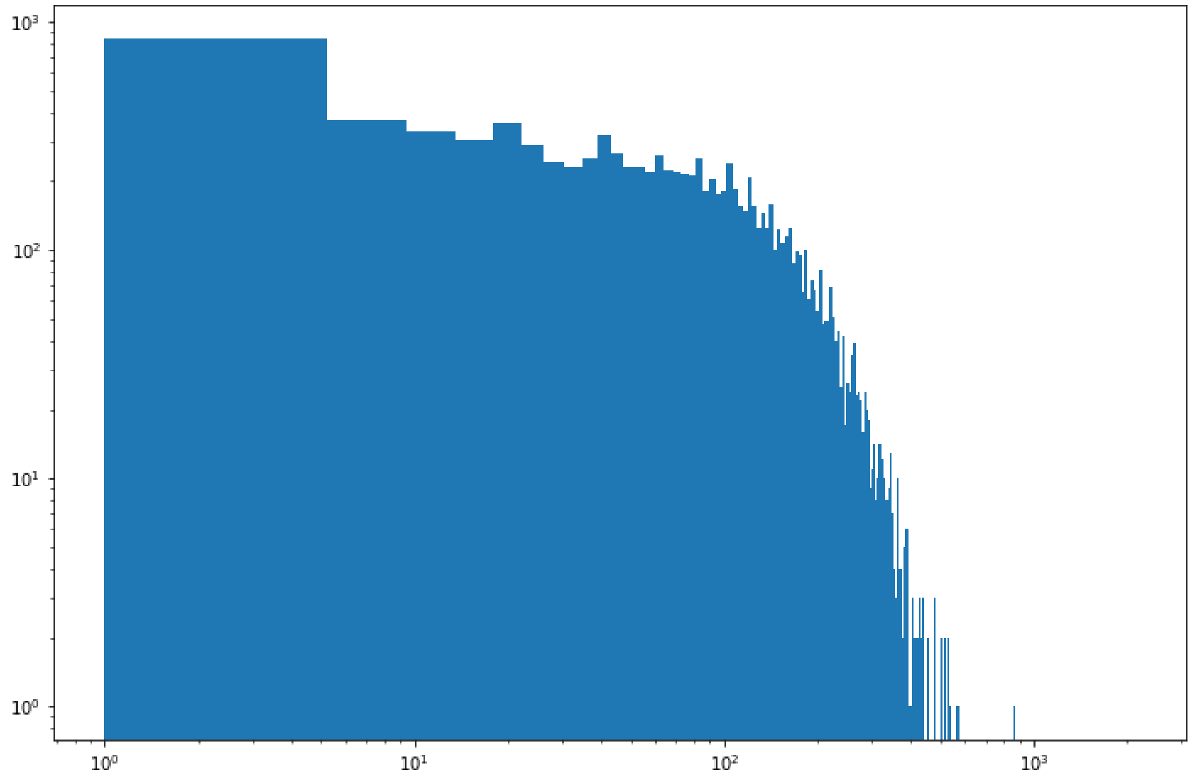
In [620]:

```
R = nx.configuration_model(degree_sequence)
```

Q6

In [621]:

```
R_degree_sequence = [R.degree(n) for n in R.nodes]
plt.figure(figsize=(12, 8))
plt.hist(degree_sequence,bins=500)
plt.xscale('log')
plt.yscale('log')
```



In [622]:

```
stats.describe(R_degree_sequence)
```

Out[622]:

```
DescribeResult(nobs=10567, minmax=(1, 2105), mean=92.42680041639065, variance
=7140.918335237751, skewness=3.43109618834199, kurtosis=52.60765891928037)
```

In [623]:

```
from scipy.stats import ttest_ind, ttest_ind_from_stats

t, p = ttest_ind(degree_sequence, R_degree_sequence, equal_var=False)
```

In [624]:

```
p
```

Out[624]:

```
1.0
```

The above verifies that the distributions are the same

Q7

In [625]:

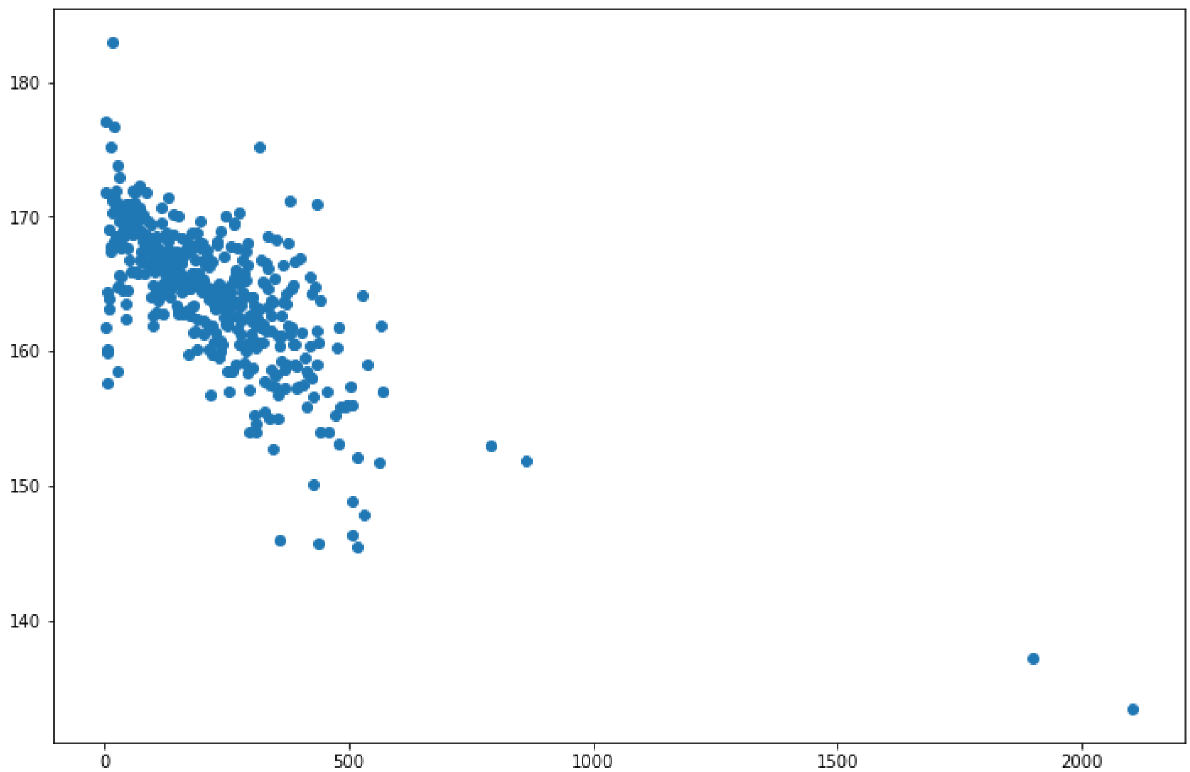
```
knn_dict = nx.k_nearest_neighbors(R)
k, knn = list(knn_dict.keys()), list(knn_dict.values ())
```

In [626]:

```
plt.figure(figsize=(12, 8))
plt.scatter(k, knn)
```

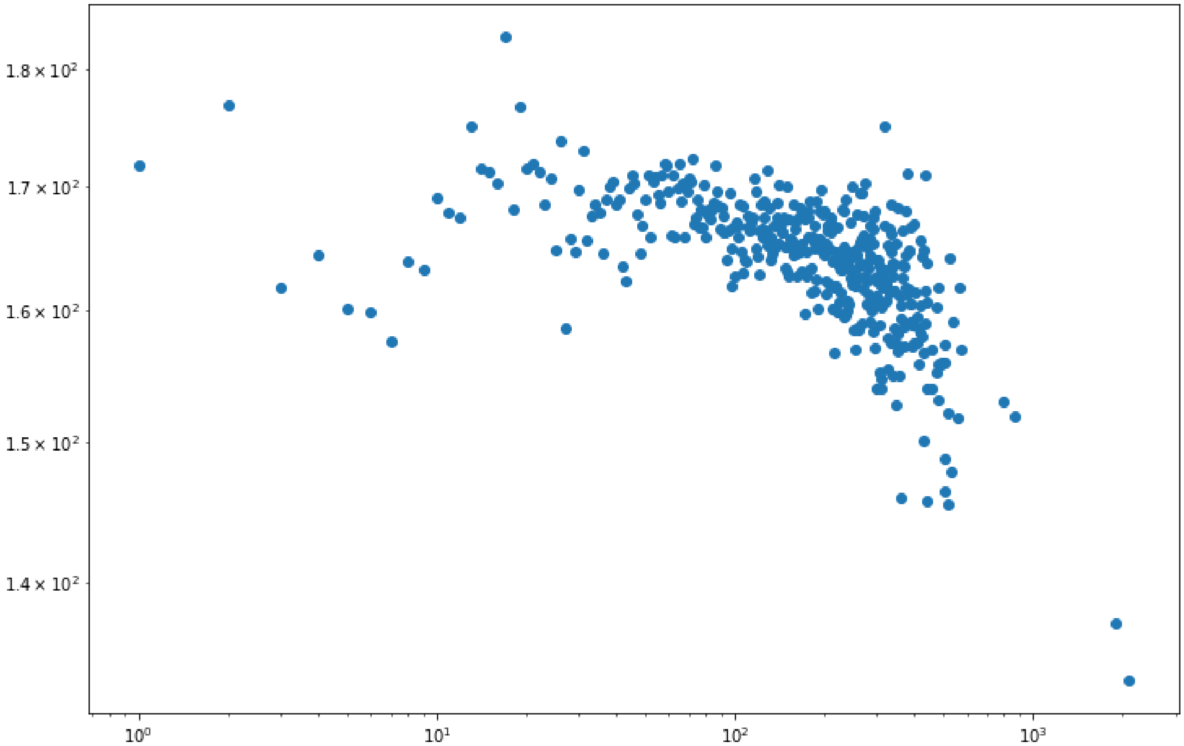
Out[626]:

```
<matplotlib.collections.PathCollection at 0x7f0bee73ecc0>
```



In [627]:

```
plt.figure(figsize=(12, 8))
plt.scatter(k, knn)
plt.xscale('log')
plt.yscale('log')
```



If 〈knn(k)〉 is an increasing function ofk,then high-degree nodes tend to be connected to high-degree nodes, therefore the network is assortative; if 〈knn(k)〉 decreases withk, the network is disassortative. Real world social networks tend to be assortative where high-degree nodes associate with other high-degree nodes. When we build a random network with the same degree sequence, we cannot expect it to retain all of it's real-world social network characteristics

Section 2

Build a network with the random walk model, for p=0.1,0.2, ...,0.9,1.0, the same number of nodes of the `openflights_usa.edges` network and an approximately equal number of links (i.e. the number m of new links per node is given by rounding up the average degree of the network).

- **Q8**. Find the value of p that in your opinion produces the most similar network to the airport network, by comparing the average clustering coefficient and the heterogeneity parameter.
  (Hint: since you have two measures to reproduce, the best p could be the one that leads to the smallest deviation from the empirical values of the average clustering coefficient and heterogeneity parameter. For each variable you compute the difference between the real value and the one of the model network, in absolute value, and divide it by the real value. You then sum this score over the two variables and pick the p yielding the minimal score.)
- **Q9**. Plot the degree distributions of the model network for the optimal p and the real network (in the same diagram).
- **Q10**. Compute the maximum modularity of the optimal model network and of the real network respectively. Use the Louvain algorithm. Compare the modularity values and comment: What's the similarity between the two partitions? Use the normalized mutual information.

Q8

In [786]:

```
G_real = nx.read_edgelist('openflights_usa.edges')

G_real = nx.convert_node_labels_to_integers(G_real, 0)
```

In [787]:

```
len(G_real.edges())
```

Out[787]:

```
2781
```

In [788]:

```
G_real.nodes()
```

Out[788]:

```
NodeView((0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 1
9, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 3
8, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 5
7, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 7
6, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 9
5, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 11
1, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126,
127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 14
2, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157,
158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 17
3, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188,
189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 20
4, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219,
220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 23
5, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250,
251, 252, 253, 254, 255, 256, 257, 258, 259, 260, 261, 262, 263, 264, 265, 26
6, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281,
282, 283, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 295, 296, 29
7, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311, 312,
313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324, 325, 326, 327, 32
8, 329, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343,
344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357, 358, 35
9, 360, 361, 362, 363, 364, 365, 366, 367, 368, 369, 370, 371, 372, 373, 374,
375, 376, 377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388, 389, 39
0, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403, 404, 405,
406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 416, 417, 418, 419, 420, 42
1, 422, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435, 436,
437, 438, 439, 440, 441, 442, 443, 444, 445, 446, 447, 448, 449, 450, 451, 45
2, 453, 454, 455, 456, 457, 458, 459, 460, 461, 462, 463, 464, 465, 466, 467,
468, 469, 470, 471, 472, 473, 474, 475, 476, 477, 478, 479, 480, 481, 482, 48
3, 484, 485, 486, 487, 488, 489, 490, 491, 492, 493, 494, 495, 496, 497, 498,
499, 500, 501, 502, 503, 504, 505, 506, 507, 508, 509, 510, 511, 512, 513, 51
4, 515, 516, 517, 518, 519, 520, 521, 522, 523, 524, 525, 526, 527, 528, 529,
530, 531, 532, 533, 534, 535, 536, 537, 538, 539, 540, 541, 542, 543, 544, 54
5))
```

In [789]:

```
degrees = dict(G_real.degree())
sum_of_edges = sum(degrees.values())
avg_degree = sum_of_edges/546
avg_degree
```

Out[789]:

```
10.186813186813186
```

In [790]:

```
nx.average_clustering(G_real)
```

Out[790]:

```
0.4930453868822472
```

In [ ]:

```

```

In [791]:

```
import networkx as nx
import random

def barabasi_albert_graph_random(N,m,p):
    # 1. Start with a clique of m+1 nodes
    G = nx.complete_graph(m+1)

    for i in range(G.number_of_nodes(), N):
        # 2. Select m different nodes at random
        new_neighbors = []
        possible_neighbors = list(G.nodes)

        j = random.choice(possible_neighbors)
        new_neighbors.append(j)
        possible_neighbors.remove(j)

        for a in range(m-1):
            r=random.random()
            if r < p:
                #choose random neighbor of new neighbor
                j_neighbors=list(G.neighbors(j))
                k=random.choice(j_neighbors)
                new_neighbors.append(k)
                j_neighbors.remove(k)
            else:
                k = random.choice(possible_neighbors)
                new_neighbors.append(k)
                possible_neighbors.remove(k)
        # 3. Add a new node i and link it with the m nodes from the previous step.
        for j in new_neighbors:
            G.add_edge(i, j)

    return G
```

In [792]:

```
def heterogeneity_param(G):
    nodes=len(G.nodes())
    edges=len(G.edges())

    summation=0

    for n in G.nodes():
        summation+=G.degree(n)**2
    denom=((2*edges)/nodes)**2
    numerator=summation/nodes
    parameter=numerator/denom
    hetero_param = numerator/denom

    return hetero_param
```

In [793]:

```
R_1 = barabasi_albert_graph_random(546, 11, .1)
```

In [794]:

```
nx.average_clustering(R_1)
```

Out[794]:

```
0.0688545213885958
```

In [795]:

```
heterogeneity_param(R_1)
```

Out[795]:

```
1.2366038281739329
```

In [796]:

```
m = 11
N = 546
P = [.1, .2, .3, .4, .5, .6, .7, .8, .9, 1]

graphs = []

#results = (clustering, heterogeneity)
results = []

for p in P:
    G = barabasi_albert_graph_random(N, m, p)
    cluster = nx.average_clustering(G)
    hetero = heterogeneity_param(G)
    results.append((cluster, hetero))
    graphs.append(G)
```

In [797]:

```
real_clustering = 0.4930453868822472
real_hetero = 5.347494770144302

scores = []

for result in results:
    clustering = result[0]
    hetero = result[1]
    clustering_score = abs(real_clustering - clustering) /real_clustering
    hetero_score = abs(real_hetero - hetero) / real_hetero
    scores.append(clustering_score + hetero_score)
```

In [798]:

```
import numpy as np
np.argmin(scores)
```

Out[798]:

```
9
```

In [799]:

```
scores[9]
```

Out[799]:

```
0.6741852381115583
```

It's my opinion that p=1 is the best choice

Q9

In [800]:

```
R = graphs[9]
R_degree_sequence = [R.degree(n) for n in R.nodes]
len(R.edges())
```

Out[800]:

```
4331
```

In [801]:

```
G_degree_sequence = [G_real.degree(n) for n in G_real.nodes]
len(G_real.edges)
```

Out[801]:

2781

In [802]:

```
from scipy.stats import ttest_ind, ttest_ind_from_stats

t, p = ttest_ind(G_degree_sequence, R_degree_sequence, equal_var=False)
```

In [803]:

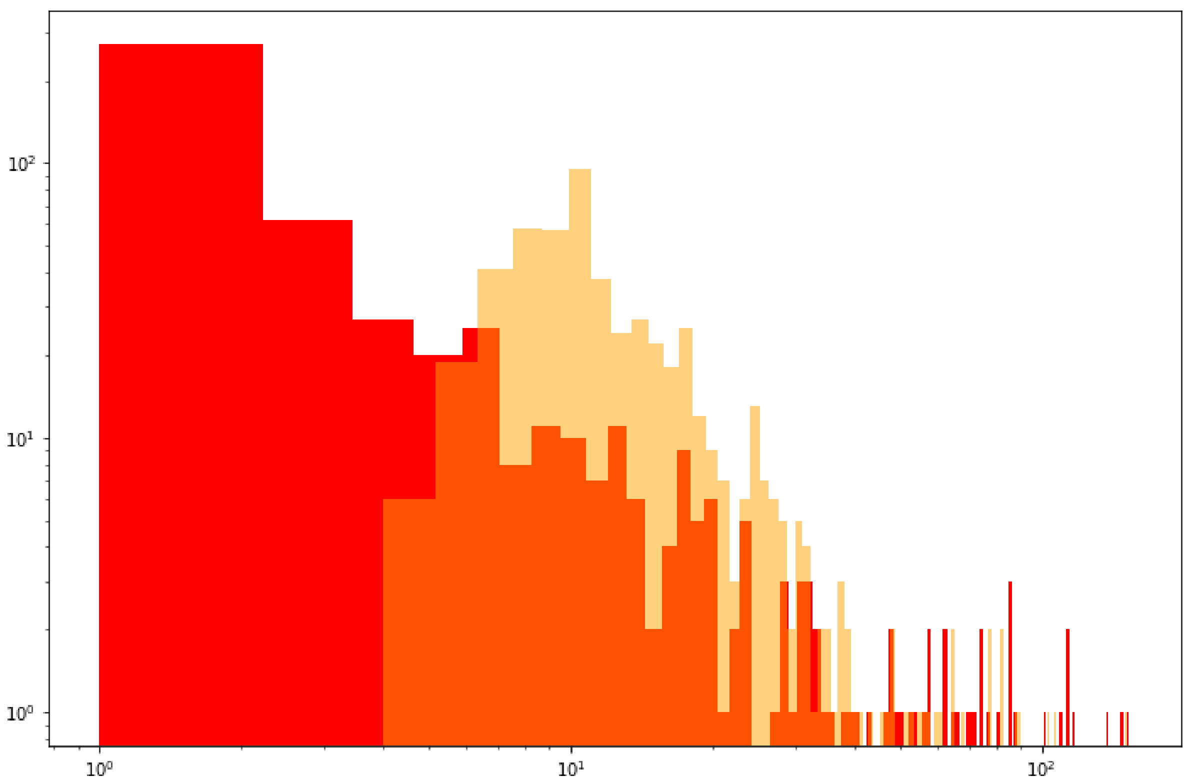
```
p
```

Out[803]:

2.8489784012142944e-07

In [804]:

```
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 8))
plt.hist(G_degree_sequence, color='red', bins = 125)
plt.hist(R_degree_sequence, color='orange',bins = 125, alpha = .5)
plt.xscale('log')
plt.yscale('log')
```



Q10

In [805]:

```
import networkx.algorithms.community as nx_comm
import community as community_louvain
from sklearn.metrics.cluster import normalized_mutual_info_score as mi

communities = []
modularities = []
degrees = []
stds = []

modularity=[]
num_communities=0
avg_degree=0


for _ in range(10):
    partition=community_louvain.best_partition(G_real)
    this_num_communities=sorted(list(partition.values()))[-1]+1
    partition_arr=[set() for i in range(this_num_communities)]
    for n in partition:
        partition_arr[partition[n]].add(n)
    modularity.append(nx_comm.modularity(G_real, partition_arr))
    num_communities += this_num_communities
    these_degrees = G_real.degree()
    sum_of_edges = sum(dict(these_degrees).values())
    avg_degree += sum_of_edges/1000
degrees.append(avg_degree/10)
communities.append(num_communities/10)
modularities.append(np.mean(modularity))
stds.append(np.std(modularity))

print(f"G has avg modularity of {np.mean(modularity)}")
print(f"G has avg # of communities of {num_communities/10}")
print(f"G has avg degree of {avg_degree/10}")
print(f"G has maximum modularity of {sorted(modularity, reverse = True)[0]}")
```

```
G has avg modularity of 0.3518812158478689
G has avg # of communities of 9.5
G has avg degree of 5.561999999999999
G has maximum modularity of 0.35400915520520465
```

In [806]:

```
communities = []
modularities = []
degrees = []
stds = []

modularity=[]
num_communities=0
avg_degree=0
for _ in range(10):
    partition=community_louvain.best_partition(R)
    this_num_communities=sorted(list(partition.values()))[-1]+1
    partition_arr=[set() for i in range(this_num_communities)]
    for n in partition:
        partition_arr[partition[n]].add(n)
    modularity.append(nx_comm.modularity(R, partition_arr))
    num_communities += this_num_communities
    these_degrees = R.degree()
    sum_of_edges = sum(dict(these_degrees).values())
    avg_degree += sum_of_edges/1000
degrees.append(avg_degree/10)
communities.append(num_communities/10)
modularities.append(np.mean(modularity))
stds.append(np.std(modularity))
print(f"R has avg modularity of {np.mean(modularity)}")
print(f"R has avg # of communities of {num_communities/10}")
print(f"R has avg degree of {avg_degree/10}")
print(f"R has maximum modularity of {sorted(modularity, reverse = True)[0]}")
```

```
R has avg modularity of 0.5682318692712768
R has avg # of communities of 10.0
R has avg degree of 8.662000000000003
R has maximum modularity of 0.5718991131096415
```

In [807]:

```
from sklearn.metrics.cluster import normalized_mutual_info_score as mi

R_partition_map = community_louvain.best_partition(R)
G_partition_map = community_louvain.best_partition(G_real)

nmi = mi(list(G_partition_map.values()),list(R_partition_map.values()))
```

In [808]:

```
nmi
```

Out[808]:

```
0.030045365024203339
```

Homegrown implemnation of NMI

I was not able to successfully implement homegrown NMI (as you can see on my first submission), until I found this library which is where this code was adapted from: [https://github.com/altsoph/community_loglike (https://github.com/altsoph/community_loglike)](https://github.com/altsoph/community_loglike)

I've also included my original attempt below this working implementation

In [894]:

```
import math
from math import log, exp, sqrt
from collections import Counter

def eta(data):
    ldata = len(list(data))
    if ldata <= 1: return 0
    _exp = exp(1)
    counts = Counter()
    for d in data:
        counts[d] += 1
    probs = [float(c) / ldata for c in counts.values()]
    probs = [p for p in probs if p > 0.]
    ent = 0
    for p in probs:
        if p > 0.:
            ent -= p * log(p, _exp)
    return ent
```

In [898]:

```
def nmi(x, y):
    sum_mi = 0.0
    x_value_list = list(set(x))
    y_value_list = list(set(y))
    lx = len(list(x))
    ly = len(list(y))
    Px = []
    for xval in x_value_list:
        Px.append( len(list(filter(lambda q:q==xval,x)))/float(lx) )
    Py = []
    for yval in y_value_list:
        Py.append( len(list(filter(lambda q:q==yval,y)))/float(ly) )

    for i in range(len(x_value_list)):
        if Px[i] ==0.:
            continue
        sy = []
        for j,yj in enumerate(y):
            if x[j] == x_value_list[i]:
                sy.append( yj )
        if len(sy)== 0:
            continue
        pxy = []
        for yval in y_value_list:
            pxy.append( len(list(filter(lambda q:q==yval,sy)))/float(ly) )

        t = []
        for j,q in enumerate(Py):
            if q>0:
                t.append( pxy[j]/Py[j] / Px[i])
            else:
                t.append( -1 )
            if t[-1]>0:
                sum_mi += (pxy[j]*log( t[-1]) )
    eta_xy = eta(x)*eta(y)
    if eta_xy == 0.: return 0.,0.
    return 2.*sum_mi/(eta(x)+eta(y))
```

In [899]:

```
nmi = nmi(list(G_partition_map.values()),list(R_partition_map.values()))
```

In [900]:

```
nmi
```

Out[900]:

```
0.030045365024203338
```

In [903]:

```
G_num_communities=sorted(list(G_partition_map.values()))[-1]+1
R_num_communities=sorted(list(R_partition_map.values()))[-1]+1

#break communities apart
R_communities = np.zeros(R_num_communities)
G_communities = np.zeros(G_num_communities)
for i in range(G_num_communities):
    for k in G_partition_map:
        if G_partition_map[k] == i:
            G_communities[i] += 1
for i in range(R_num_communities):
    for k in R_partition_map:
        if R_partition_map[k] == i:
            R_communities[i] += 1

G_communities = np.array(G_communities)
R_communities = np.array(R_communities)

P_g = G_communities/546
P_r = R_communities/546

H_r = -np.sum(Pr*np.log(Pr))
H_g = -np.sum(Pg*np.log(Pg))

for i in range(len(G_communities)):
    for j in range(len(R_communities)):
        if (j,i) in nodes_in_common.keys():
            continue
        nodes_in_common[(i,j)]=0
    for n in G_real.nodes():
        if G_partition_map[n] == i and R_partition_map[list(G_real.nodes()).index(n)] == j
:
            nodes_in_common[(i,j)] += 1

H_gr=[]
for key in nodes_in_common:

    if nodes_in_common[key] > 0:
        P_gr = nodes_in_common[key]/546
        this_Pr = P_r[key[1]]
        H_gr.append(P_gr*np.log(this_Pr/P_gr))

H_gr = np.sum(H_gr)

nmi = (2*H_g + 2*H_gr) / (H_g + H_r)

nmi/100
```

Out[903]:

```
0.019003366375256573
```

The NMI score is very low so I conclude that there is not much similarity between the two partitions.

Section 3

Consider the **coevolution model** on an **Erdoes-Renyi random graph** with 1000 nodes and link probability 0.01. Consider 40 initially distributed opinions and different values for the rewiring probability p= 0.05,0.1, 0.15, 0.20,0.25, ...,0.95.

- **Q11**. Run the simulation for each value of p until the system reaches a stationary state, i.e. until such point where each node has only neighbors of the same opinion as its own.
- **Q12**. Compute the average size of the connected components in the stationary state and plot it as a function of p.

In [412]:

```
G = nx.erdos_renyi_graph(n=1000, p=.001)
```

In [413]:

```
def initial_state(G):
    state = {}
    for node in G.nodes:
        state[node] = random.choice(range(0,40))
    return state
```

In [414]:

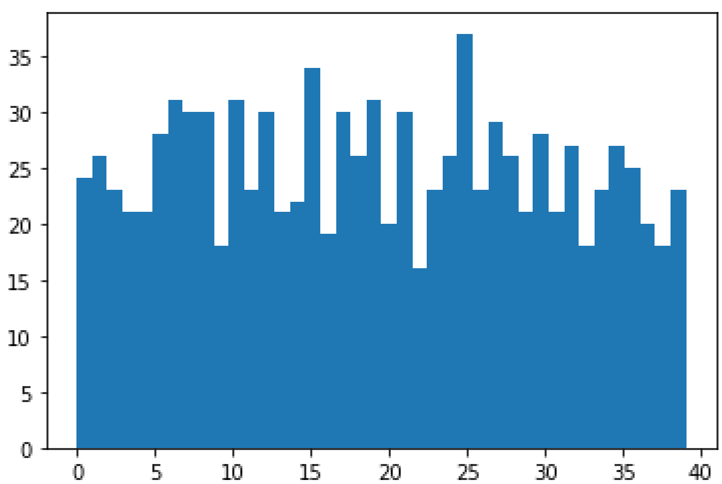
```
def choose_neighbor(G,node):
    return random.choice(list(G.neighbors(node)))

def choose_random(G,node):
    return random.choice(list(G.nodes()))


def state_transition(G, current_state, p):
    next_state = {}
    for node in G.nodes:

        if len(list(G.neighbors(node))) == 0:
            continue

        neighbor = choose_neighbor(G,node)

        r = random.random()

        if r<p:
            new_friend = choose_random(G,node)
            while new_friend in G.neighbors(node):
                new_friend = choose_random(G, node)
            G.remove_edge(node,neighbor)
            G.add_edge(node,new_friend)
        else:
            next_state[node] = current_state[neighbor]

    return next_state, G

#check for convergence
def is_converged(G, current_state):
    for n in G.nodes():
        for neighbor in G.neighbors(n):
            if current_state[n] != current_state[neighbor]:
                return False
    return True
```

In [365]:

```
initial_state = initial_state(G)
plt.hist(list(initial_state.values()),bins=40)
```

Out[365]:

```
(array([24., 26., 23., 21., 21., 28., 31., 30., 30., 18., 31., 23., 30.,
        21., 22., 34., 19., 30., 26., 31., 20., 30., 16., 23., 26., 37.,
        23., 29., 26., 21., 28., 21., 27., 18., 23., 27., 25., 20., 18.,
        23.]),
 array([ 0.   ,  0.975,  1.95 ,  2.925,  3.9  ,  4.875,  5.85 ,  6.825,
         7.8  ,  8.775,  9.75 , 10.725, 11.7  , 12.675, 13.65 , 14.625,
        15.6  , 16.575, 17.55 , 18.525, 19.5  , 20.475, 21.45 , 22.425,
        23.4  , 24.375, 25.35 , 26.325, 27.3  , 28.275, 29.25 , 30.225,
        31.2  , 32.175, 33.15 , 34.125, 35.1  , 36.075, 37.05 , 38.025,
        39.   ]),
 <BarContainer object of 40 artists>)
```



In [426]:

```
len([len(c) for c in sorted(nx.connected_components(E), key=len, reverse=True) if len(c)>1
])
```

Out[426]:

```
136
```

In [429]:

```
#next_state, E = state_transition(E, initial_state, .05)
#current_state = initial_state
#converged = is_converged(E,current_state)


P = np.linspace(0.05, .95, 20)
a = 0

cc = []
max_cc = []
runs = []
clusters = []
for p in P:

    E = G.copy()
    current_state = initial_state(E)
    next_state, E = state_transition(E, current_state, p)
    converged = is_converged(E,current_state)

    num_runs=0

    converged = is_converged(E,current_state)

    while not converged:

        num_runs+=1
        next_state, E = state_transition(E, current_state, p)

        for n in next_state:
            current_state[n] = next_state[n]

        converged = is_converged(E,current_state)

    print(f"For p={p}, it took {num_runs} runs to converge")

    #filtering out components of the graph of size 1
    components = [len(c) for c in sorted(nx.connected_components(E), key=len, reverse=True
) if len(c)>1]

    this_cc = np.mean(components)
    largest_cc = np.max(components)

    cc.append(this_cc)
    max_cc.append(largest_cc)
    clusters.append(len(components))
    runs.append(num_runs)

    print(f"For p={p}, the average size of the connected components is {this_cc}")

    print(f"The largest component for p={p} is {largest_cc}")

    print(f"For p={p}, there are {len(components)} clusters of size greater than 1")

    print("")
```

```
For p=0.05, it took 2064 runs to converge
For p=0.05, the average size of the connected components is 4.89922480620155
The largest component for p=0.05 is 91
For p=0.05, there are 129 clusters of size greater than 1

For p=0.09736842105263158, it took 5486 runs to converge
For p=0.09736842105263158, the average size of the connected components is 4.
968253968253968
The largest component for p=0.09736842105263158 is 185
For p=0.09736842105263158, there are 126 clusters of size greater than 1

For p=0.14473684210526316, it took 3467 runs to converge
For p=0.14473684210526316, the average size of the connected components is 4.
6838235294117645
The largest component for p=0.14473684210526316 is 51
For p=0.14473684210526316, there are 136 clusters of size greater than 1

For p=0.19210526315789472, it took 3073 runs to converge
For p=0.19210526315789472, the average size of the connected components is 4.
90625
The largest component for p=0.19210526315789472 is 176
For p=0.19210526315789472, there are 128 clusters of size greater than 1

For p=0.23947368421052628, it took 1883 runs to converge
For p=0.23947368421052628, the average size of the connected components is 4.
451388888888889
The largest component for p=0.23947368421052628 is 33
For p=0.23947368421052628, there are 144 clusters of size greater than 1

For p=0.28684210526315784, it took 6057 runs to converge
For p=0.28684210526315784, the average size of the connected components is 4.
921259842519685
The largest component for p=0.28684210526315784 is 145
For p=0.28684210526315784, there are 127 clusters of size greater than 1

For p=0.33421052631578946, it took 3240 runs to converge
For p=0.33421052631578946, the average size of the connected components is 4.
787878787878788
The largest component for p=0.33421052631578946 is 94
For p=0.33421052631578946, there are 132 clusters of size greater than 1

For p=0.381578947368421, it took 2070 runs to converge
For p=0.381578947368421, the average size of the connected components is 4.68
38235294117645
The largest component for p=0.381578947368421 is 44
For p=0.381578947368421, there are 136 clusters of size greater than 1

For p=0.4289473684210526, it took 2400 runs to converge
For p=0.4289473684210526, the average size of the connected components is 4.6
69117647058823
The largest component for p=0.4289473684210526 is 147
For p=0.4289473684210526, there are 136 clusters of size greater than 1

For p=0.47631578947368414, it took 7273 runs to converge
For p=0.47631578947368414, the average size of the connected components is 4.
891472868217054
The largest component for p=0.47631578947368414 is 62
```

```
For p=0.47631578947368414, there are 129 clusters of size greater than 1

For p=0.5236842105263158, it took 2757 runs to converge
For p=0.5236842105263158, the average size of the connected components is 4.4
08163265306122
The largest component for p=0.5236842105263158 is 57
For p=0.5236842105263158, there are 147 clusters of size greater than 1

For p=0.5710526315789474, it took 4321 runs to converge
For p=0.5710526315789474, the average size of the connected components is 5.0
894308943089435
The largest component for p=0.5710526315789474 is 138
For p=0.5710526315789474, there are 123 clusters of size greater than 1

For p=0.618421052631579, it took 3871 runs to converge
For p=0.618421052631579, the average size of the connected components is 4.86
9230769230769
The largest component for p=0.618421052631579 is 67
For p=0.618421052631579, there are 130 clusters of size greater than 1

For p=0.6657894736842105, it took 4322 runs to converge
For p=0.6657894736842105, the average size of the connected components is 4.4
14965986394558
The largest component for p=0.6657894736842105 is 73
For p=0.6657894736842105, there are 147 clusters of size greater than 1

For p=0.7131578947368421, it took 3348 runs to converge
For p=0.7131578947368421, the average size of the connected components is 4.5
85714285714285
The largest component for p=0.7131578947368421 is 89
For p=0.7131578947368421, there are 140 clusters of size greater than 1

For p=0.7605263157894736, it took 10901 runs to converge
For p=0.7605263157894736, the average size of the connected components is 4.7
03703703703703
The largest component for p=0.7605263157894736 is 80
For p=0.7605263157894736, there are 135 clusters of size greater than 1

For p=0.8078947368421052, it took 7613 runs to converge
For p=0.8078947368421052, the average size of the connected components is 4.8
68217054263566
The largest component for p=0.8078947368421052 is 65
For p=0.8078947368421052, there are 129 clusters of size greater than 1

For p=0.8552631578947368, it took 8160 runs to converge
For p=0.8552631578947368, the average size of the connected components is 5.3
73913043478261
The largest component for p=0.8552631578947368 is 168
For p=0.8552631578947368, there are 115 clusters of size greater than 1

For p=0.9026315789473683, it took 17816 runs to converge
For p=0.9026315789473683, the average size of the connected components is 4.8
91472868217054
The largest component for p=0.9026315789473683 is 94
For p=0.9026315789473683, there are 129 clusters of size greater than 1

For p=0.95, it took 47953 runs to converge
```

```
For p=0.95, the average size of the connected components is 4.875968992248062
The largest component for p=0.95 is 121
For p=0.95, there are 129 clusters of size greater than 1
```

In [431]:

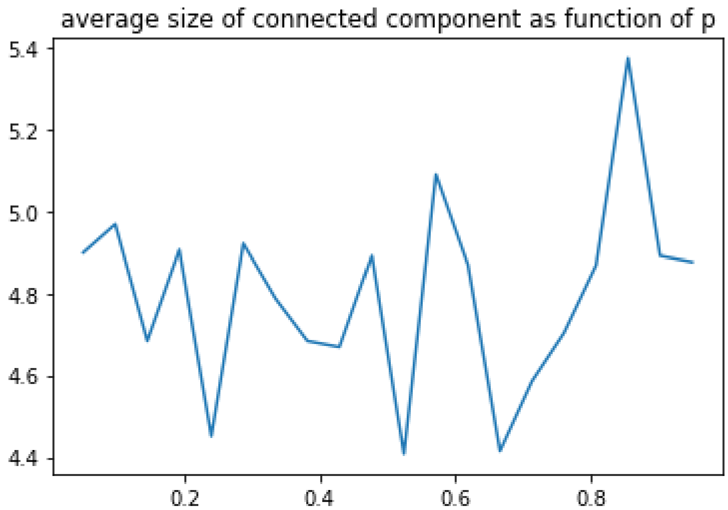
```python
P = np.linspace(0.05, .95, 20)
```

In [432]:

```python
plt.title("average size of connected component as function of p")
plt.plot(P,cc)
```

Out[432]:

```
[<matplotlib.lines.Line2D at 0x7f0c0accbf28>]
```

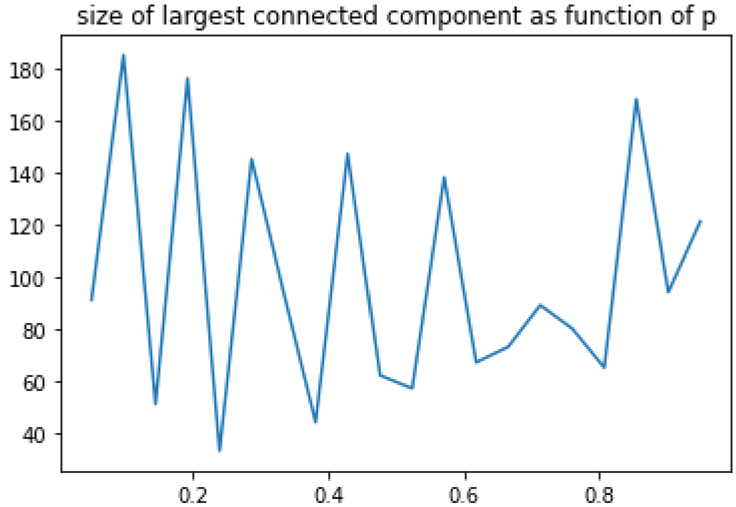


In [433]:

```
plt.title("size of largest connected component as function of p")
plt.plot(P,max_cc)
```

Out[433]:

```
[<matplotlib.lines.Line2D at 0x7f0c0ad607b8>]
```

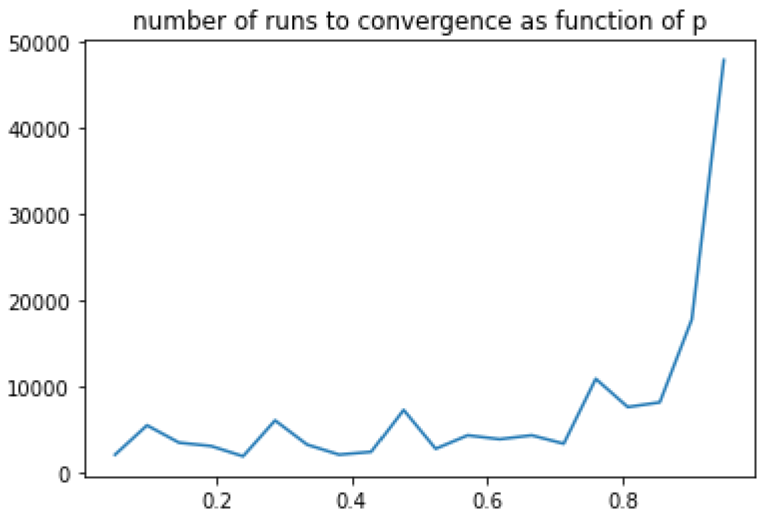


In [434]:

```
plt.title("number of runs to convergence as function of p")
plt.plot(P,runs)
```

Out[434]:

```
[<matplotlib.lines.Line2D at 0x7f0c0ae27e10>]
```

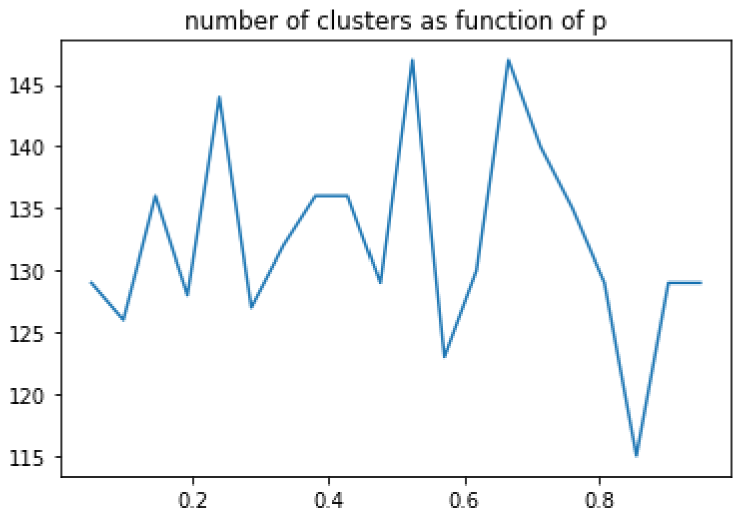


In [435]:

```
plt.title("number of clusters as function of p")
plt.plot(P,clusters)
```

Out[435]:

```
[<matplotlib.lines.Line2D at 0x7f0c0a324a20>]
```



I tried to find trends that I expected to see based on the characteristics of the coevalution model, but I did not find what I was looking for. Possibly this is due to the randomness and if I iterated over each p, 100 times I would see the trends described in the book.

The book says:

"If we start froma random network with average degree larger than one, we know that it has agiant component (Section 5.1), so for selection probability near zero in the longrun there will be a giant community holding the majority opinion, and many smallcommunities with different opinions. For selection probability near one, instead,the link dynamics will break the network into many small components, each mademostly of nodes that were initially assigned one of the distinct opinions. It turnsout that there is an abrupt transition between the scenario with a large majorityopinion and the scenario with many smaller opinion communities of comparablesize. This transition takes place at a threshold value of the selection probability."

I think that the key term for this experiment is, "in the longrun".

In [ ]: