Heidelberg University
Institute of Computer Science
Database Systems Research Group

Lecture: Complex Network Analysis

Prof. Dr. Michael Gertz

Assignment 7

Degree Assortativity and Robustness

`https://github.com/nilskre/CNA_assignments`

Team Member: Patrick Günther, 3660886,
Applied Computer Science
rh269@stud.uni-heidelberg.de

Team Member: Felix Hausberger, 3661293,
Applied Computer Science
eb260@stud.uni-heidelberg.de

Team Member: Nils Krehl, 3664130,
Applied Computer Science
pu268@stud.uni-heidelberg.de

Problem 7-1 Degree Correlations and Assortativity

January 10, 2022

1 Lecture: Complex Network Analysis

Prof. Dr. Michael Gertz

Winter Semester 2021/22

1.1 Assignment 7 - Assortativity and Robustness

Students: Felix Hausberger, Nils Krehl, Patrick Günther

2 1. Build graph

[1]:
```
import pandas as pd
import networkx as nx
import numpy as np
import seaborn as sns
import matplotlib.pylab as plt
import scipy
```

[2]:
```
df_blogs = pd.read_csv('assortativity_networks/blogs.txt', sep="\t", header=None)
df_javax = pd.read_csv('assortativity_networks/javax.txt',␣
 ↪delim_whitespace=True, header=None)
df_network_science = pd.read_csv('assortativity_networks/network-science.txt',␣
 ↪sep="\t", header=None)
```

[3]:
```
df_blogs
```

[3]:

	0	1
0	1	2
1	1	3
2	1	4
3	1	5
4	1	6
...	...	...
33425	975	664
33426	975	67
33427	975	1004

```
33428   975  1224
33429  1028   791

[33430 rows x 2 columns]
```

[4]:
```
# since it is an undirected graph, no parallel edges are added
G_blogs = nx.Graph()
G_blogs.add_edges_from(df_blogs.itertuples(index=False))

G_javax = nx.Graph()
G_javax.add_edges_from(df_javax.itertuples(index=False))

G_network_science = nx.Graph()
G_network_science.add_edges_from(df_network_science.itertuples(index=False))

# remove self-loops
G_blogs.remove_edges_from(nx.selfloop_edges(G_blogs))
G_javax.remove_edges_from(nx.selfloop_edges(G_javax))
G_network_science.remove_edges_from(nx.selfloop_edges(G_network_science))
```

[5]:
```
print(f"Number of nodes in blogs is {G_blogs.number_of_nodes()}.")
print(f"Number of edges in blogs is {G_blogs.number_of_edges()}.")
print()
print(f"Number of nodes in javax is {G_javax.number_of_nodes()}.")
print(f"Number of edges in javax is {G_javax.number_of_edges()}.")
print()
print(f"Number of nodes in network-science is {G_network_science.
 ↪number_of_nodes()}.")
print(f"Number of edges in network-science is {G_network_science.
 ↪number_of_edges()}.")
```

```
Number of nodes in blogs is 1224.
Number of edges in blogs is 16715.

Number of nodes in javax is 6120.
Number of edges in javax is 50290.

Number of nodes in network-science is 1461.
Number of edges in network-science is 2742.
```

3 2. Degree correlation matrix

[6]:
```
def calculate_degree_correlation_matrix(G):
    max_degree = max(deg for n, deg in G.degree)
    # create a dict to save the number of degree combinations
    degrees = []
    for i in range(max_degree+1):
```

```
        for j in range(max_degree+1):
            degrees.append((i,j))

    deg_1 = []
    deg_2 = []
    for i in degrees:
        deg_1.append(i[0])
        deg_2.append(i[1])
    d = {'deg_1': deg_1, 'deg_2': deg_2, 'count': 0}
    degree_correlation_df = pd.DataFrame(data=d)

    for u,v,weight in G.edges(data=True):
        degree_correlation_df.loc[degree_correlation_df.eval(f'deg_1 == {G.
↪degree(u)} & deg_2 == {G.degree(v)}'), 'count'] += 1

    deg_corr_mat = np.zeros((max_degree+1, max_degree+1))
    for index, row in degree_correlation_df.iterrows():
        deg_corr_mat[row['deg_1'], row['deg_2']] = row['count']

    deg_corr_mat = deg_corr_mat + deg_corr_mat.T
    deg_corr_mat_prob = deg_corr_mat / np.sum(deg_corr_mat)

    deg_corr_mat_absolute = deg_corr_mat

    return deg_corr_mat_absolute, deg_corr_mat_prob
```

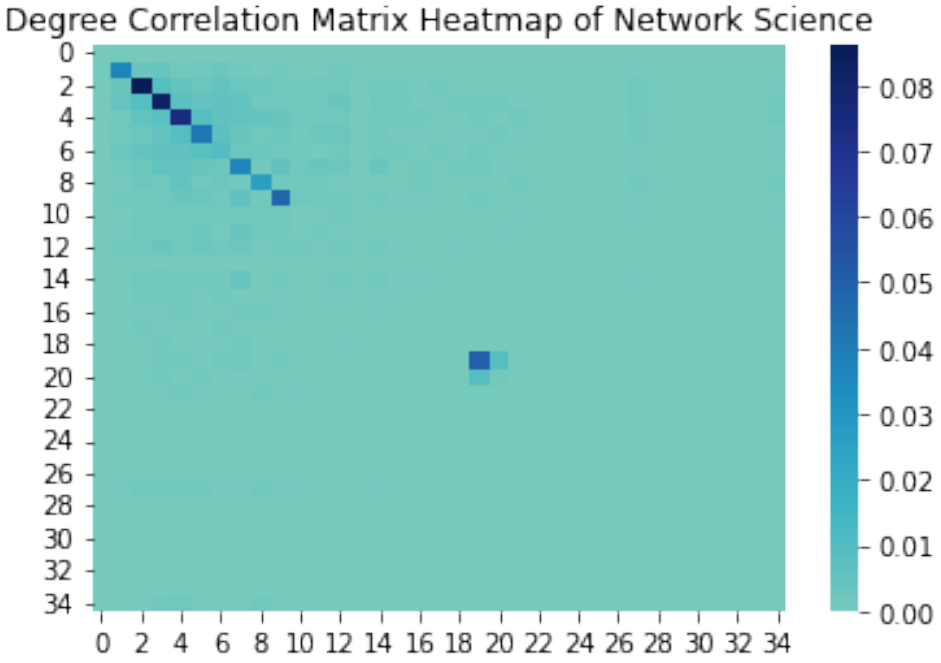
[7]:
```
deg_corr_mat_blogs_absolute, deg_corr_mat_blogs =␣
↪calculate_degree_correlation_matrix(G_blogs)
```

[8]:
```
deg_corr_mat_network_science_absolute, deg_corr_mat_network_science =␣
↪calculate_degree_correlation_matrix(G_network_science)
```

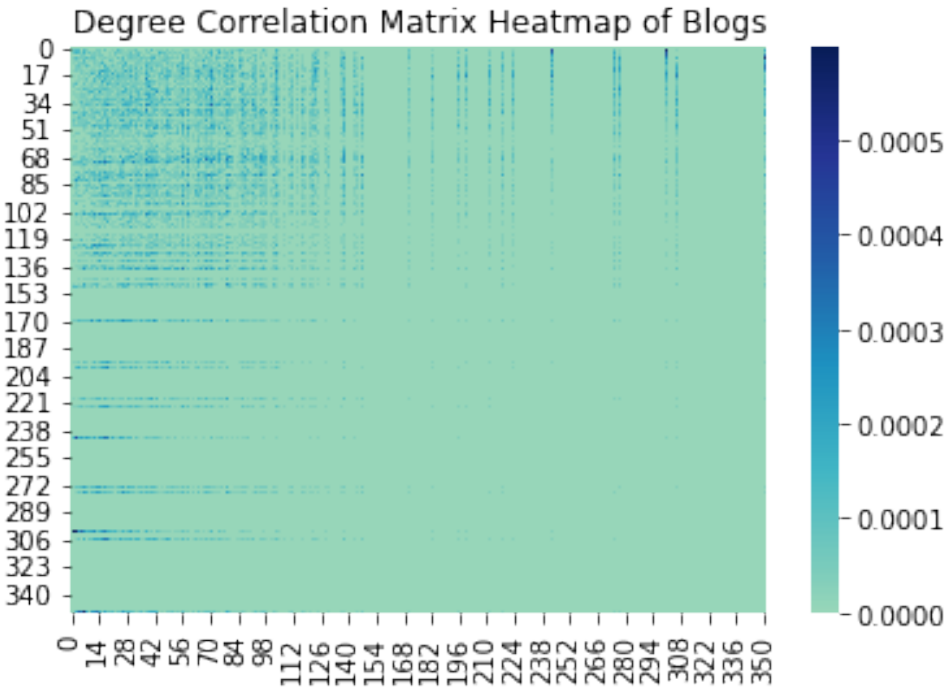
[9]:
```
deg_corr_mat_javax_absolute, deg_corr_mat_javax =␣
↪calculate_degree_correlation_matrix(G_javax)
```

4 3. Heatmap

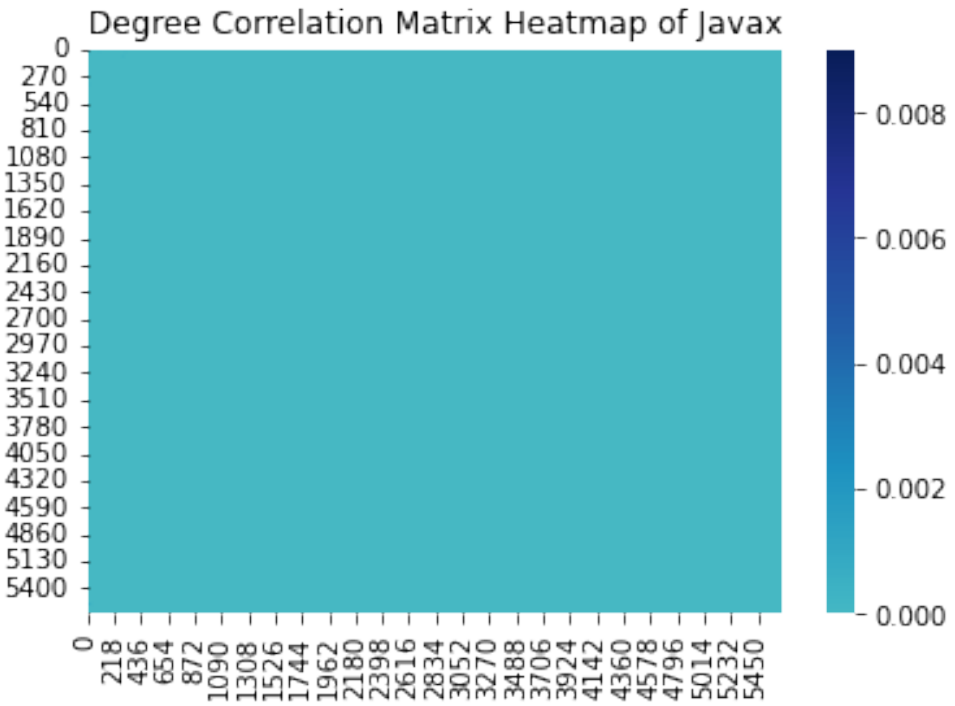
[10]:
```
ax = sns.heatmap(deg_corr_mat_network_science, cmap="YlGnBu", center=0.015)
plt.title("Degree Correlation Matrix Heatmap of Network Science")
plt.show()
```



[11]:
```
ax = sns.heatmap(deg_corr_mat_blogs, cmap="YlGnBu", center=0.00015)
plt.title("Degree Correlation Matrix Heatmap of Blogs")
plt.show()
```



[23]:
```
ax = sns.heatmap(deg_corr_mat_javax, cmap="YlGnBu", center=0.00015)
plt.title("Degree Correlation Matrix Heatmap of Javax")
plt.show()
```



5 4. Nearest neighbor degree

[12]:
```
# calculates nearest neighbor degree for singe nodes
def calculate_k_nn_single_node(G, node):
    neighbors = list(G.neighbors(node))
    return np.sum([G.degree(neighbor) for neighbor in neighbors]) / G.
 ↪degree(node)
```

[13]:
```
# get k_i
degrees_network_science = [G_network_science.degree(node) for node in␣
 ↪G_network_science.nodes]
k_i_network_science = []
for node in list(G_network_science.nodes):
    k_i_network_science.append(calculate_k_nn_single_node(G_network_science,␣
 ↪node))
```

```
degrees_blogs = [G_blogs.degree(node) for node in G_blogs.nodes]
k_i_blogs = []
for node in list(G_blogs.nodes):
    k_i_blogs.append(calculate_k_nn_single_node(G_blogs, node))

k_i_javax = []
degrees_javax = [G_javax.degree(node) for node in G_javax.nodes]
for node in list(G_javax.nodes):
    k_i_javax.append(calculate_k_nn_single_node(G_javax, node))
```

[14]:
```
# calculates nearest neighbor degree for all nodes of degree k
def calculate_k_nn(k, deg_corr_mat_absolute):
    neighbors = deg_corr_mat_absolute[k]
    num_neighbors = np.sum(neighbors)

    return np.sum([k_prime * neighbors[k_prime] / num_neighbors for k_prime in␣
 ↪range(len(neighbors))])
```

[15]:
```
# get k_nn
k_nn_network_science = []
for k in range(len(deg_corr_mat_network_science_absolute[0])):
    k_nn_network_science.append(calculate_k_nn(k,␣
 ↪deg_corr_mat_network_science_absolute))

k_nn_blogs = []
for k in range(len(deg_corr_mat_blogs_absolute[0])):
    k_nn_blogs.append(calculate_k_nn(k, deg_corr_mat_blogs_absolute))

k_nn_javax = []
for k in range(len(deg_corr_mat_javax_absolute[0])):
    k_nn_javax.append(calculate_k_nn(k, deg_corr_mat_javax_absolute))
```

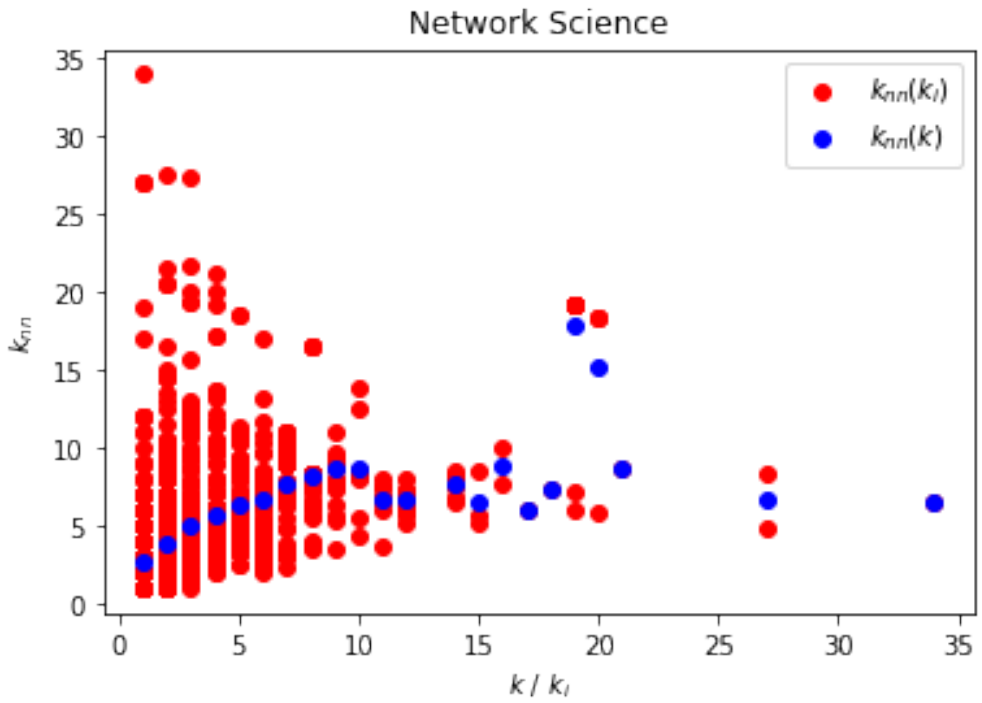
```
/opt/anaconda3/envs/complexnetworkanalysis/lib/python3.7/site-
packages/ipykernel_launcher.py:6: RuntimeWarning: invalid value encountered in
double_scalars
```

[ ]:
```

```

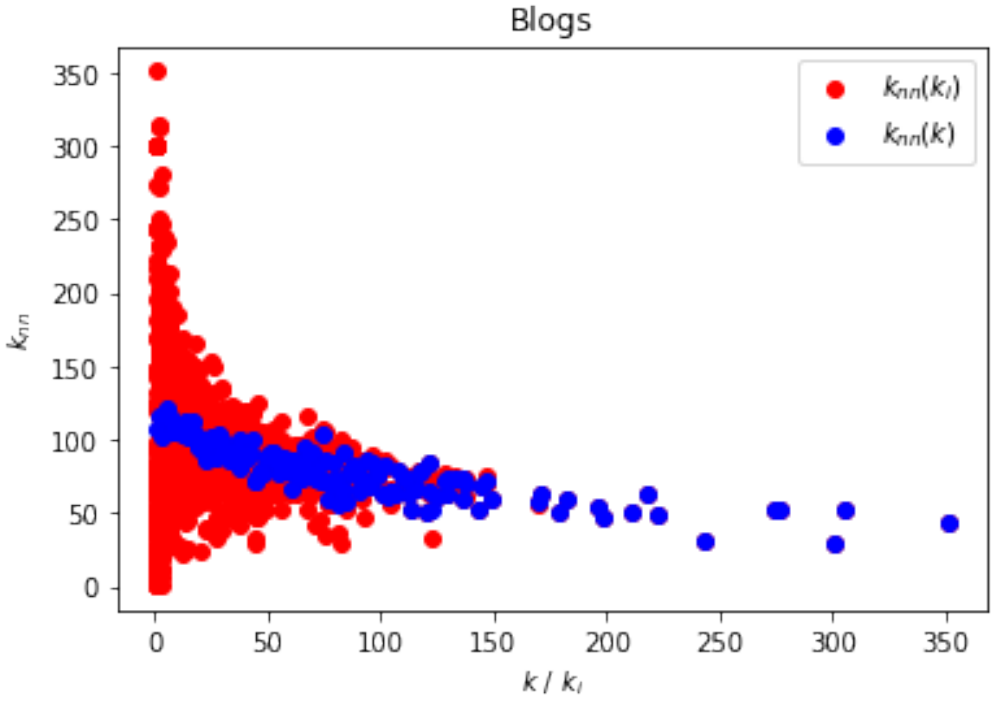
6 Scatter plot

[16]:
```
plt.scatter(degrees_network_science, k_i_network_science, c='red',␣
 ↪label='$k_{nn}(k_i)$')
plt.scatter(range(len(deg_corr_mat_network_science_absolute[0])),␣
 ↪k_nn_network_science, c='blue', label='$k_{nn}(k)$')
```

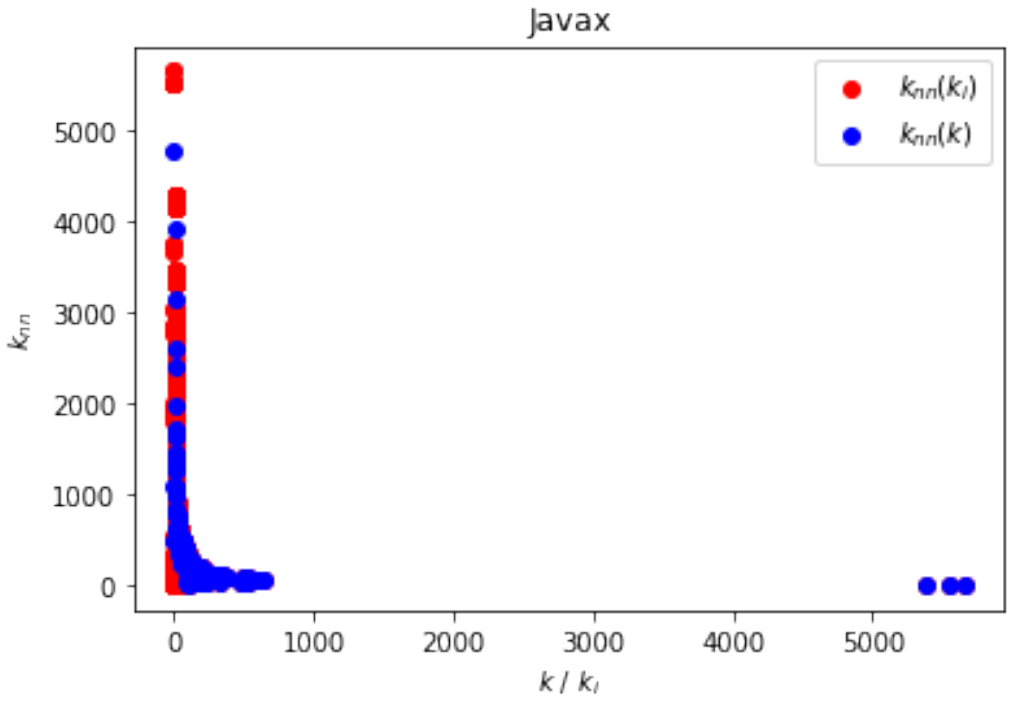
```
plt.title("Network Science")
plt.xlabel('$k$ / $k_i$')
plt.ylabel('$k_{nn}$')
plt.legend()
plt.show()
```



[17]:
```
plt.scatter(degrees_blogs, k_i_blogs, c='red', label='$k_{nn}(k_i)$')
plt.scatter(range(len(deg_corr_mat_blogs_absolute[0])), k_nn_blogs, c='blue',␣
↪label='$k_{nn}(k)$')
plt.title("Blogs")
plt.xlabel('$k$ / $k_i$')
plt.ylabel('$k_{nn}$')
plt.legend()
plt.show()
```



[18]:
```
plt.scatter(degrees_javax, k_i_javax, c='red', label='$k_{nn}(k_i)$')
plt.scatter(range(len(deg_corr_mat_javax_absolute[0])), k_nn_javax, c='blue',␣
↪label='$k_{nn}(k)$')
plt.title("Javax")
plt.xlabel('$k$ / $k_i$')
plt.ylabel('$k_{nn}$')
plt.legend()
plt.show()
```



7 Degree correlation coefficient

[19]:
```
def compute_degree_correlation_coefficient(G, deg_corr_mat):
    max_degree = max(deg for n, deg in G.degree)

    avg_degree = sum(deg for n, deg in G.degree)/len(G.degree)

    q_k = {}
    for deg in range(max_degree + 1):
        p_k = [deg for n, deg in G.degree].count(deg)/len(G.degree)
        q_k[deg] = (deg * p_k)/avg_degree

    sigma_squared = sum([(k**2) * q_k[k] for k in q_k]) - sum([k * q_k[k] for k␣
 ↪in q_k])**2

    r = []

    for j, row in enumerate(deg_corr_mat):
        for k, e_jk in enumerate(row):
            qk = q_k[k]
            qj = q_k[j]
            r.append((j*k*(e_jk-qj*qk))/sigma_squared)
```

```
    r = sum(r)

    return r
```

[20]:
```
print(f"The degree correlation coefficient with our computation for Network␣
 ↪Science is r={compute_degree_correlation_coefficient(G_network_science,␣
 ↪deg_corr_mat_network_science)}")
# to check our computation, we also use the inbuild function of networkx
print(f"The degree correlation coefficient with the inbuild networkx function␣
 ↪for Network Science is r={nx.algorithms.assortativity.
 ↪degree_assortativity_coefficient(G_network_science)}")
```

```
The degree correlation coefficient with our computation for Network Science is
r=0.4616224667525837
The degree correlation coefficient with the inbuild networkx function for
Network Science is r=0.4616224667525835
```

[21]:
```
print(f"The degree correlation coefficient with our computation for Blogs is␣
 ↪r={compute_degree_correlation_coefficient(G_blogs, deg_corr_mat_blogs)}")
# to check our computation, we also use the inbuild function of networkx
print(f"The degree correlation coefficient with the inbuild networkx function␣
 ↪for Blogs is r={nx.algorithms.assortativity.
 ↪degree_assortativity_coefficient(G_blogs)}")
```

```
The degree correlation coefficient with our computation for Blogs is
r=-0.2212328638045546
The degree correlation coefficient with the inbuild networkx function for Blogs
is r=-0.22123286380455423
```

[26]:
```
print(f"The degree correlation coefficient with our computation for Javax is␣
 ↪r={compute_degree_correlation_coefficient(G_javax, deg_corr_mat_javax)}")
# to check our computation, we also use the inbuild function of networkx
print(f"The degree correlation coefficient with the inbuild networkx function␣
 ↪for Javax is r={nx.algorithms.assortativity.
 ↪degree_assortativity_coefficient(G_javax)}")
```

```
The degree correlation coefficient with our computation for Javax is
r=-0.2327051928360141
The degree correlation coefficient with the inbuild networkx function for Javax
is r=-0.23270519283601443
```

[28]:
```
# because it took forever: pickle stuff
import pickle
with open('deg_corr_mat_javax_absolute.pkl','wb') as f:
    pickle.dump(deg_corr_mat_javax_absolute, f)

with open('deg_corr_mat_javax.pkl','wb') as f:
```

```
    pickle.dump(deg_corr_mat_javax, f)
```

[ ]:
```

```

Problem 7-2 Molloy-Reed Criterion

Consider a configuration model network that has nodes of degree 1, 2, and 3 only, with probabilities p_1, p_2, and p_3, respectively. The degree distribution is given by:

$$p_k = \delta_{k,1}p_1 + \delta_{k,2}p_2 + \delta_{k,3}p_3, \begin{cases} \delta_{k,1} = 3 & \text{if } k = 1 \\ \delta_{k,2} = 2 & \text{if } k = 2 \\ \delta_{k,3} = 1 & \text{if } k = 3 \end{cases} \tag{1}$$

1. Compute the first moment $\langle k \rangle$ and the second moment $\langle k^2 \rangle$ of the degree distribution.

   We assume $\delta_{k,k'}$ to be the dirac-delta-function. For the first and second moment is follows:

   $$\langle k \rangle = \sum_{k=1}^{3} kp_k = 1p_1 + 2p_2 + 3p_3 = 3p_1 + 4p_2 + 3p_3$$

   $$\langle k^2 \rangle = \sum_{k=1}^{3} k^2 p_k = 1p_1 + 4p_2 + 9p_3 = 3p_1 + 8p_2 + 9p_3$$

   Note that we substitute p_1 with $3p_1$ and p_2 with $2p_2$ (p_3 remains $2p_3$) as given by equation 1 (slightly confusing by the task description).

2. Using the Molloy-Reed criterion, show that there is a giant component if and only if $p_1 < 3p_3$.

   The Molloy-Reed criteria propagates a giant component exists in case $\kappa = \frac{\langle k^2 \rangle}{\langle k \rangle} > 2$. κ can be calculated as:

   $$\kappa = \frac{\langle k^2 \rangle}{\langle k \rangle} = \frac{1p_1 + 4p_2 + 9p_3}{1p_1 + 2p_2 + 3p_3}$$

   which is only true for

   $$\begin{aligned} \frac{1p_1 + 4p_2 + 9p_3}{1p_1 + 2p_2 + 3p_3} &> 2 \\ 1p_1 + 4p_2 + 9p_3 &> 2p_1 + 4p_2 + 6p_3 \\ 3p_3 &> p_1 \end{aligned}$$

   Note that we use the LHS declaration of p_k from equation 1.

3. In terms of the structure of the network, discuss the meaning of the condition $p_1 < 3p_3$. Why does the result not depend on p_2?

   For the network to have a giant component, the probability of a node having a single degree should be at most three times as high as the probability of a node having a degree of three. This limits the amount of single degree nodes and promotes a faster growth of a giant component since the average degree will most likely not be close to $\langle k \rangle = 1$, but rather higher (assuming we exclude isolated nodes as in equation 1) since single degree nodes cannot prevail the network.

   The probability p_2 fell apart from the equation shown in subtask 2, leading to the assumption that the emergence of a giant component does not need to be dependent on p_2. This makes sense since we know the constraint $p_1 < 3p_3$ holds, which already leads to the corollar that $\langle k \rangle \geq 1$ and therefore leads to the guaranteed emergence of a giant component.

Problem 7-3 Xalvi-Brunet and Sokolov Algorithm

December 29, 2021

Lecture: Complex Network Analysis

Prof. Dr. Michael Gertz

Winter Semester 2021/22

Assignment 7 - Assortativity and Robustness

Problem 7-3: Xalvi-Brunet and Sokolov Algorithm

Students: Felix Hausberger, Nils Krehl, Patrick Günther

[1]:
```
import pandas as pd
import networkx as nx
import numpy as np
import seaborn as sns
import matplotlib.pylab as plt
import scipy
```

1. Xalvi-Brunet and Sokolov algorithm

[2]:
```
def xalvi_brunet_sokolov_algorithm(graph, num_iterations, assortative):
    network = graph.copy()
    for i in range(num_iterations):
        links = np.array(list(network.edges))
        degrees = network.degree()
        # choose two random links
        choosen_indices = np.random.choice(range(len(links)), 2, replace=False)
        choosen_links = links[choosen_indices]

        # get corresponding nodes and their node degrees
        corresponding_nodes = choosen_links.flatten()
        corresponding_node_degrees = np.array([degrees[x] for x in␣
 ↪corresponding_nodes])

        # sort the nodes by their degrees in descending order
        index_array = np.argsort(corresponding_node_degrees)[::-1]
```

```
        ordered_nodes = corresponding_nodes[index_array]
        ordered_nodes_degrees = corresponding_node_degrees[index_array]

        # remove the selected links
        network.remove_edge(choosen_links[0][0], choosen_links[0][1])
        network.remove_edge(choosen_links[1][0], choosen_links[1][1])

        # rewiring
        if assortative == True:
            network.add_edge(ordered_nodes[0], ordered_nodes[1])
            network.add_edge(ordered_nodes[2], ordered_nodes[3])
        else:
            network.add_edge(ordered_nodes[0], ordered_nodes[3])
            network.add_edge(ordered_nodes[1], ordered_nodes[2])

    return network
```

2. Create networks with Xalvi-Brunet and Sokolov algorithm

[3]:
```
df_neutral_network = pd.read_csv('neutral_network.txt', delim_whitespace=True,␣
↪header=None)
```

[4]:
```
G_neutral_network = nx.Graph()
G_neutral_network.add_edges_from(df_neutral_network.itertuples(index=False))
```

[5]:
```
G_assortative = xalvi_brunet_sokolov_algorithm(G_neutral_network, 5000, True)
G_disassortative = xalvi_brunet_sokolov_algorithm(G_neutral_network, 5000, False)
```

[6]:
```
print("Degree Correlation Coefficient")
print("r = 0: neutral network; r < 0: disassortative network; r > 0: assortative␣
↪network \n")
print("Neutral network Degree Correlation Coefficient: {}".format(nx.
↪degree_pearson_correlation_coefficient(G_neutral_network)))
print("Assortative network Degree Correlation Coefficient: {}".format(nx.
↪degree_pearson_correlation_coefficient(G_assortative)))
print("Disassortative network Degree Correlation Coefficient: {}".format(nx.
↪degree_pearson_correlation_coefficient(G_disassortative)))
```

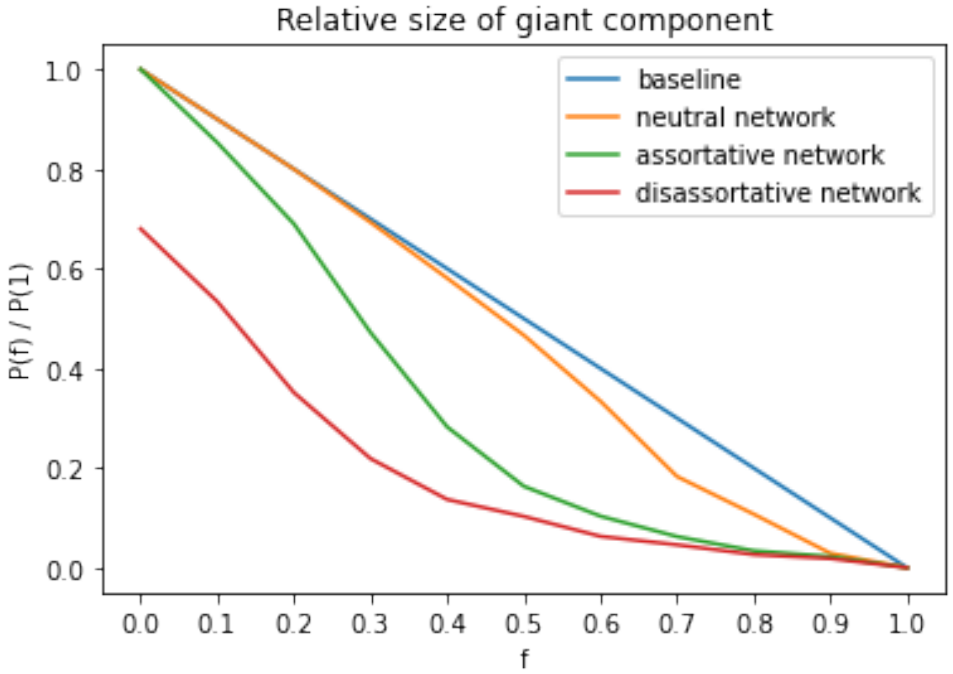
```
Degree Correlation Coefficient
r = 0: neutral network; r < 0: disassortative network; r > 0: assortative
network

Neutral network Degree Correlation Coefficient: -0.009246262701730106
Assortative network Degree Correlation Coefficient: 0.9037799701732246
Disassortative network Degree Correlation Coefficient: -0.6223530885853903
```

3. Plot giant component size

[69]:
```
def get_giant_component_size(network):
    if network.number_of_nodes() > 0:
        giant_component = max(nx.connected_components(network), key=len)
        giant_component_size = len(giant_component)
        return giant_component_size
    else:
        return 0


def get_relative_size_of_giant_component(graph, num_samples=20):
    network = graph.copy()
    number_nodes = network.number_of_nodes()
    f = []

    relative_size_of_giant_component = []
    for f_value in np.arange(0,1.1,0.1):
        giant_component_size = []
        for sample in range(num_samples):
            minimized_network = network.copy()
            number_to_be_removed = int(f_value * number_nodes)

            random_sample = np.random.choice(minimized_network.nodes(),␣
 ↪number_to_be_removed, replace=False)
            minimized_network.remove_nodes_from(random_sample)

            giant_component_size.
 ↪append(get_giant_component_size(minimized_network))
        relative_size_of_giant_component.append(np.mean(np.
 ↪array(giant_component_size)))

    return np.array(relative_size_of_giant_component) / 100

neutral_relative_size_of_giant_component =␣
 ↪get_relative_size_of_giant_component(G_neutral_network)
assortative_relative_size_of_giant_component =␣
 ↪get_relative_size_of_giant_component(G_assortative)
disassortative_relative_size_of_giant_component =␣
 ↪get_relative_size_of_giant_component(G_disassortative)
```

[71]:
```
plt.plot(np.arange(10,-1,-1) / 10, label="baseline")
plt.plot(neutral_relative_size_of_giant_component, label="neutral network")
plt.plot(assortative_relative_size_of_giant_component, label="assortative␣
 ↪network")
```

```
plt.plot(disassortative_relative_size_of_giant_component, label="disassortative␣
↪network")
plt.xticks(ticks=range(11), labels=(np.arange(0,11,1) / 10))
plt.legend()
plt.title("Relative size of giant component")
plt.xlabel("f")
plt.ylabel("P(f) / P(1)")
plt.show()
```



4. Discussion

Discuss the results from the previous task: Which network is the most robust against random failures? Explain why this is the case.

- The plot above shows, that with increasing f (increased number of removed nodes), the size of the giant component decreases slowest in the neutral network. That is why the most robust network against random failues is the neutral network. In a neutral network nodes are linked randomly and consequently the density of links is around the average degree.
- In assortative networks hubs tend to link to each other and small-degree nodes tend to connect to small degree nodes.
- In disassortative networks hubs avoid each other. Small-degree nodes tend to connect to hubs, and hubs tend to connect to small-degree nodes (this is called hub-and-spoke character). When removing hubs the network is quickly divided into parts. This explains the rapid reduction of the giant component size in disassortative networks.

Problem 7-4 Random Failures in Uncorrelated Networks

Compute the critical threshold f_c for each of the following degree distributions, under the assumption that the networks do not exhibit any degree correlation.

1. Poisson distribution, i.e.,

$$p_k = e^{-\mu}\frac{\mu^k}{k!}$$

2. Discrete exponential distribution, i.e.,

$$p_k = (1 - e^{-\lambda})e^{-\lambda k}$$

3. Dirac delta distribution, i.e.,

$$p_k = \delta_{k,k_0} = \begin{cases} 1 & \text{if } k = k_0, \\ 0 & \text{otherwise.} \end{cases}$$

Discuss the consequences of your results for network robustness.
Hint: You may use the first and second moment from the lecture or other literature without a proof.

We know f_c can be calculated by:

$$f_c = 1 - \frac{1}{\frac{\langle k^2 \rangle}{\langle k \rangle} - 1}$$

1. For the poisson distribution, we receive:

$$f_c = 1 - \frac{1}{\frac{\mu^2+\mu}{\mu} - 1} = 1 - \frac{1}{\mu}$$

   This means the higher μ the more robust the network is towards random failures. If $\mu \to \infty$ we receive maximum robustness as all nodes would theoretically need to fail for the network to be considered fragmented (even if this does not make sense since there would not be a network present anymore).

2. For the discrete exponential distribution, we receive (using Wolfram Alpha):

$$\langle k \rangle = \frac{(1 - e^{-\lambda})}{\lambda^2}$$

$$\langle k^2 \rangle = \frac{2(1 - e^{-\lambda})}{\lambda^3}$$

$$f_c = 1 - \frac{1}{\frac{2}{\lambda} - 1}$$

   This means the lower λ the more robust the network is towards random failures. If $\lambda \to 0$ we receive maximum robustness.

3. For the dirac delta distribution, we receive:

$$f_c = 1 - \frac{1}{\frac{k_0^2}{k_0} - 1} = 1 - \frac{1}{k_0 - 1}$$

   Similar to the poisson distribution, the dirac delta distribution becomes more robust towards random failures the higher k_0 is. This becomes maximum for clique like structures.