PSG COLLEGE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

19OH01 - Social and Economic Network Analysis

Topic : Analyzing trust with Social Networks

Team Members:

18Z307-Aravinth M
18Z317-Guru Krish S
18Z331-Kiran G
18Z333-Manish P
18z347-Roan Mathew

Problem Statement:

We have analyzed the who-trust-whom online social network of a general consumer review site Epinions. It was a website where people could review products. Users were able to register for free and earn money according to how much their reviews were found useful.We have performed the structural analysis, interpreted and described structural balance properties using the Epinions dataset.

Dataset:

- Description:This is a who-trust-whom online social network of a general consumer review site Epinions.com. Members of the site can decide whether to "trust" each other. All the trust relationships interact and form the Web of Trust which is then combined with review ratings to determine which reviews are shown to the user.
- Dataset Statistics:

Dataset statistics	
Nodes	75879
Edges	508837
Nodes in largest WCC	75877 (1.000)
Edges in largest WCC	508836 (1.000)
Nodes in largest SCC	32223 (0.425)
Edges in largest SCC	443506 (0.872)
Average clustering coefficient	0.1378
Number of triangles	1624481
Fraction of closed triangles	0.0229
Diameter (longest shortest path)	14
90-percentile effective diameter	5

- Dataset Link: https://snap.stanford.edu/data/soc-Epinions1.html

Tools used:

- Python: We have used the Python Language for the coding part because of its User-friendly Data Structures.
- NetworkX:NetworkX is the most popular Python package for manipulating and analyzing graphs.NetworkX is suitable for real-world graph problems and is good at handling big data as well.
- Python Louvain:This module implements community detection.It uses the louvain method described in Fast unfolding of communities in large networks and it depends on Networkx to handle graph operations.

Challenges Faced:

- Our graph didn't come with any previous information about the structure of the underlying communities in the graph. So, we used the Louvain algorithm to detect them.

- ❖ In the Rich Club effect, due to the size and the power-law nature of our network, we observed only the top hundred nodes and decided not to use the percentages.

Contribution of Team Members:

Roll No:	Name	Contribution
18Z307	Aravinth M	Examining the graph using Bow-Tie structure and detecting communities in the graph.
18Z317	Guru Krish S	Finding Degree correlation in the graph.
18Z331	Kiran G	Studying about Rich club effect and Negative rich club and implementing it.
18Z333	Manish P	Studying about Power law and implementing it.
18z347	Roan Mathew	Finding relationship between different types of degrees in the graph.

Annexure I: Code:

Analysis of Epinions social network

Import the necessary packages

```
import networkx as nx
import pandas as pd
import pylab
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from community import community_louvain
import numpy as np
import community
import random
```

Creating the graph

```
graph = nx.DiGraph()
epi_csv = pd.read_csv('soc-sign-epinions.csv')
epi_csv.columns = ['From','To','Weight']
# We check if the file has loaded
epi_csv.head()
  # Adding the edges.
  for index,row in epi_csv.iterrows():
```

```
   graph.add_edges_from([(row[0],row[1])],weight = row[2])
```

Creating the DataFrame containing degrees

```
# Number of nodes
print(graph.number_of_nodes())
  # Number of edges
print(graph.number_of_edges())
  # Positive and negative edges
  Stats =
pd.DataFrame(graph.out_degree(),columns=['From','Outdegree']).sort_values('Fro
m')
Stats_2 =
pd.DataFrame(graph.in_degree(),columns=['To','Indegree']).sort_values('To')
b= epi_csv.groupby('From',as_index=False)[['Weight']].sum().sort_values('From')
c= epi_csv.groupby('To',as_index=False)[['Weight']].sum().sort_values('To')
Stats = Stats.merge(b,on='From',how='left')
Stats_2 = Stats_2.merge(c,on='To',how='left')
Stats['Pos_out'] = (  Stats['Outdegree'] + Stats['Weight']  )/2
Stats['Neg_out'] = (  Stats['Outdegree'] - Stats['Weight']  )/2
Stats_2['Pos_in']= (Stats_2['Indegree']  + Stats_2['Weight'])/2
Stats_2['Neg_in']= (Stats_2['Indegree']  - Stats_2['Weight'])/2
Stats = pd.merge(Stats,Stats_2,left_on='From', right_on='To').drop('To',
axis=1)
Stats = Stats.drop(['Weight_x','Weight_y'],axis=1)
Stats.fillna(0,inplace=True)
Stats.head()
Stats.describe()
print(Stats.max(axis=0))
```

Basic stats

```
print(graph.number_of_edges())
print(graph.number_of_nodes())
print(nx.average_clustering(graph))
print(nx.transitivity(graph))
print(nx.density(graph))
```

Log-Log plots for Outdegree and indegree relationship:

```
plt.figure(figsize=(4,4))
plt.plot(Stats['Pos_out'],Stats['Neg_out'],'ro',markersize=0.25)
plt.xscale('log')
plt.yscale('log')
```

Log-Log plots for positive and negative weighted outgoing edges in the network:

```
plt.figure(figsize=(4,4))
plt.plot(Stats['Pos_in'],Stats['Neg_in'],'ro',markersize=0.25)
plt.xscale('log')
plt.yscale('log')
```

Log-Log plots for positive and negative ingoing edges:

```
plt.figure(figsize=(5,5))
plt.plot(Stats['Outdegree'],Stats['Indegree'],'ro',markersize=0.4)
```

```
plt.xscale('log')
plt.yscale('log')
```

Plotting the assortativity coefficients of the nodes.

```
Deg_cor = nx.average_neighbor_degree(graph,target='out')
dict_list = []
for key, value in Deg_cor.items():
   temp = [key,value]
   temp[0] = graph.degree(key,'out')
   dict_list.append(temp)
dfa1 = pd.DataFrame(dict_list,columns =['Outdegree','Average neighbors
outdegree'])
dfa2 = dfa1.groupby('Outdegree',as_index=False)['Average neighbors
outdegree'].mean()
plt.plot(dfa1['Outdegree'],dfa1['Average neighbors
outdegree'],'ro',markersize=0.3)
plt.plot(dfa2['Outdegree'],dfa2['Average neighbors
outdegree'],'bo',markersize=1)
plt.xscale('log')
plt.yscale('log')
```

```
Deg_cor = nx.average_neighbor_degree(graph,target='in')
dict_list = []
for key, value in Deg_cor.items():
   temp = [key,value]
   temp[0] = graph.degree(key,'in')
   dict_list.append(temp)
dfa1 = pd.DataFrame(dict_list,columns =['Indegree','Average neighbors
indegree'])
dfa2 = dfa1.groupby('Indegree',as_index=False)['Average neighbors
indegree'].mean()
plt.plot(dfa1['Indegree'],dfa1['Average neighbors
indegree'],'ro',markersize=0.3)
plt.plot(dfa2['Indegree'],dfa2['Average neighbors indegree'],'bo',markersize=1)

plt.xscale('log')
plt.yscale('log')
```

Checking assortativity

```
print(nx.degree_assortativity_coefficient(graph,'in','in'))
print(nx.degree_assortativity_coefficient(graph,'out','out'))
```

Checking the power law

```
import math
pos = Stats['Indegree'][Stats['Indegree'] != 0]
pos = pos.transform(lambda x: math.floor(math.log(x)) )

prob = pos.value_counts(normalize=True)
threshold = 0.0001
mask = prob > threshold
```

```
tail_prob = prob.loc[~mask].sum()
prob = prob.loc[mask]
prob['other'] = tail_prob
prob.plot(kind='bar',log=True,color='r')
```

```
import math
pos = Stats['Outdegree'][Stats['Outdegree'] != 0]
pos = pos.transform(lambda x: math.floor(math.log(x)) )

prob = pos.value_counts(normalize=True)
threshold = 0.0001
mask = prob > threshold
tail_prob = prob.loc[~mask].sum()
prob = prob.loc[mask]
prob['other'] = tail_prob
prob.plot(kind='bar',log=True,color='r')
```

Assortativity coefficients

```
nx.degree_assortativity_coefficient(graph,'in')
nx.degree_assortativity_coefficient(graph,'out')
nx.degree_assortativity_coefficient(graph)
```

Size of components for bowtie structure

```
SCC = max(nx.strongly_connected_components(graph), key=len)
print('Size of maximal strongly connected component is ' + str(len(SCC)))
WCC = max(nx.weakly_connected_components(graph), key=len)
print('Size of maximal weakly connected component is ' + str(len(WCC)))
```

Community detection in graph

```
SCC = max(nx.strongly_connected_components(graph), key=len)
scc_com = graph.subgraph(SCC).copy()
scc2 = nx.Graph(scc_com)
  # Removing the weights.
  for u,v,d in scc2.edges(data=True):
    d['weight']=1
  communities =community_louvain.best_partition(scc2,random_state=5)
print(set(communities.values()))
```

Getting the sizes of the communities

```
ind_graph = community.induced_graph(communities, scc2)
  inv_map = dict()
for key, value in communities.items():
    inv_map.setdefault(value, list()).append(key)
# Removing the small communities
  inv_map2 = {key:val for key, val in inv_map.items() if len(val) >= 100}
sizes = np.array([len(inv_map2[k]) for k in list(inv_map2.keys())])
  # Removing edges from the graph:
  inv_keys = [key for key in inv_map2]
ind_nodes = list(ind_graph.nodes())
```

```
for node in ind_nodes:
   if node not in inv_keys:
       ind_graph.remove_node(node)
print(sizes)
```

Checking the rich club effect

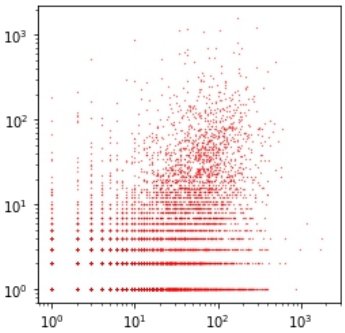
```
# We want to see the rich club effect in the 100 edges with the highest
indegree.
picked = 100
rce1 = Stats[['From','Indegree']].sort_values(by=['Indegree'],
ascending=False)[0:picked]
rich = list(rce1.From.values)
rece_graph = graph.subgraph(rich).copy()
# Dropping first value

densities = np.zeros((3,picked))
pos_edges = 0
neg_edges = 0
for rank, node1 in enumerate(rich[1::]):

   for rank2, node2 in enumerate(rich[:rank+1]):
       if graph.has_edge(node1, node2):
           if graph[node1][node2]['weight'] == 1:
               pos_edges += 1
           else:
               neg_edges += 1
       if graph.has_edge(node2, node1):
           if graph[node2][node1]['weight'] == 1:
               pos_edges += 1
           else:
               neg_edges += 1

   densities[0][rank] = pos_edges/(rank+2)/(rank+1)
   densities[1][rank] = neg_edges/(rank+2)/(rank+1)
   densities[2][rank] = (pos_edges + neg_edges)/(rank+2)/(rank+1)


plt.plot( [i for i in range(0,picked)],densities[0,],'r-',label='Positive edge
density')
plt.plot( [i for i in range(0,picked)],densities[1,],'b-',label='Negative edge
density')
plt.plot( [i for i in range(0,picked)],densities[2,],'g-',label='Total
density')
plt.legend(loc="upper right")
plt.ylim((0,1))
plt.xlim((-2,picked-2))
plt.show()
  # Checking the one sidedness
nx.reciprocity(rece_graph)
```

Negative rich club

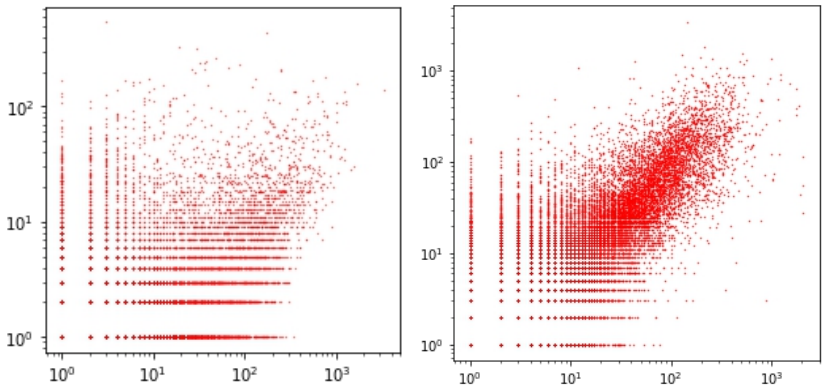
```
# We want to see the rich club effect in the 100 edges with the highest
negative_degree.
picked = 16
rce2 = Stats[['From','Neg_in']].sort_values(by=['Neg_in'],
ascending=False)[4:picked+4]
rich = list(rce2.From.values)
# Dropping first value
  densities = np.zeros((3,picked))
pos_edges = 0
neg_edges = 0
  for rank, node1 in enumerate(rich[1::]):
      for rank2, node2 in enumerate(rich[:rank+1]):
        if graph.has_edge(node1, node2):
            if graph[node1][node2]['weight'] == 1:
                pos_edges += 1
            else:
                neg_edges += 1
        if graph.has_edge(node2, node1):
            if graph[node2][node1]['weight'] == 1:
                pos_edges += 1
            else:
                neg_edges += 1

    densities[0][rank] = pos_edges/(rank+2)/(rank+1)
    densities[1][rank] = neg_edges/(rank+2)/(rank+1)
    densities[2][rank] = (pos_edges + neg_edges)/(rank+2)/(rank+1)
plt.plot( [i for i in range(0,picked)],densities[0,],'r-',label='Positive edge
density')
plt.plot( [i for i in range(0,picked)],densities[1,],'b-',label='Negative edge
density')
plt.plot( [i for i in range(0,picked)],densities[2,],'g-',label='Total
density')
plt.legend(loc="upper right")
plt.ylim((0,1))
plt.xlim((-1,picked-2))
plt.show()
```

Annexure II: Snapshots of the Output:

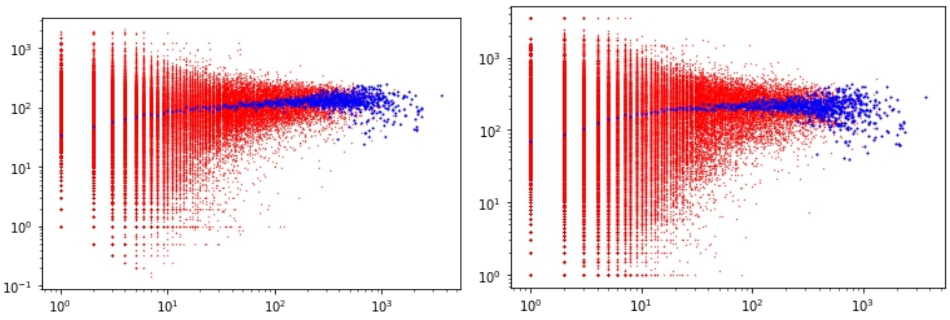
- Log-Log plots for Outdegree and indegree relationship:



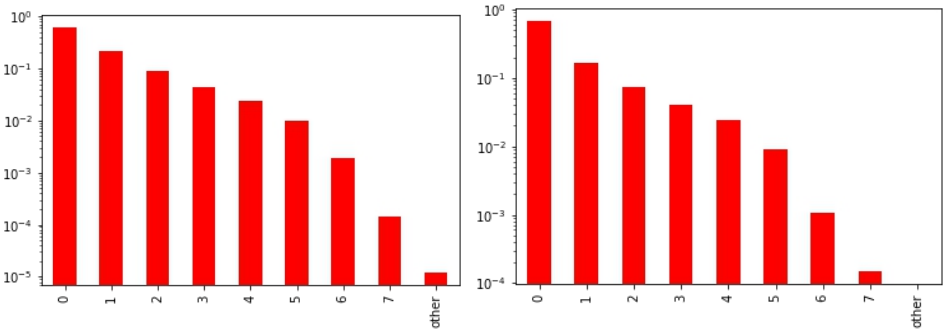
- Log-Log plots for positive and negative weighted outgoing and ingoing edges in the network:



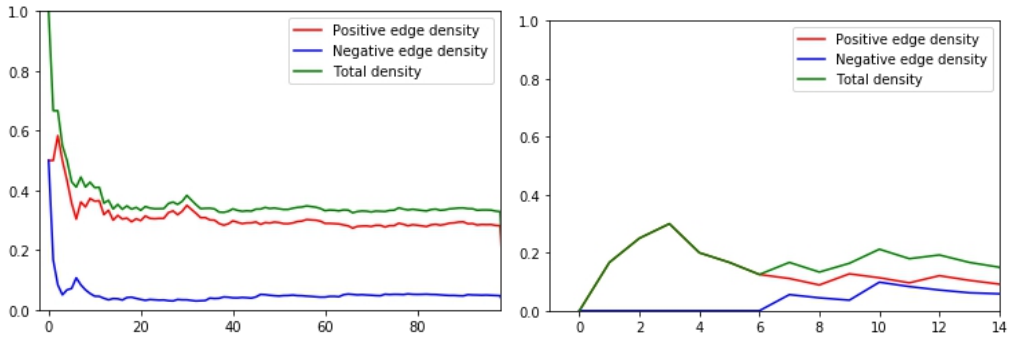
- Plotting the outdegree and indegree assortativity coefficients of the nodes:



- Checking the power law for indegree and outdegree:



- Checking the rich club effect and Negative rich club:



Reference:

- J. Kleinberg J. Leskovec D. Huttenlocher. Signed Networks in Social Medias. 2010.
- https://towardsdatascience.com/social-network-analysis-of-trust-the-epinions-dataset-22769505672d#:~:text=Conclusion%20We%20have%20seen%20that,correlations%20and%20gained%20valuable%20insights.
- https://dl.acm.org/doi/pdf/10.1109/ASONAM.2012.59
- https://chengjunwang.com/web_data_analysis/demo2_simulate_networks/
- https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5567565/
- https://toreopsahl.com/tnet/weighted-networks/weighted-rich-club-effect/
- https://people.cs.vt.edu/~irchen/5984/pdf/Sherchan-acm-CSUR13.pdf
- HANCHARD, S. 2008. Measuring trust and social networks - Where do we put our trust online? Hitwise. http://weblogs.hitwise.com/sandra-hanchard/2008/09/measuring trust and social net.html.

Plagiarism Detector .net

Date: April, 02 2021

PLAGIARISM SCAN REPORT

Excluded Url : None

Content Checked For Plagiarism

Analysis of Epinions social network Import the necessary packages import networkx as nx import pandas as pd import pylab import matplotlib.pyplot as plt import matplotlib.cm as cm from community import community_louvain import numpy as np import community import random Creating the graph graph = nx.DiGraph() epi_csv = pd.read_csv('soc-sign-epinions.csv') epi_csv.columns = ['From','To','Weight'] # We check if the file has loaded epi_csv.head() ? # Adding the edges. ? for index,row in epi_csv.iterrows(): graph.add_edges_from([(row[0],row[1])],weight = row[2]) Creating the DataFrame containing degrees # Number of nodes print(graph.number_of_nodes()) ? # Number of edges print(graph.number_of_edges()) ? # Positive and negative edges ? Stats = pd.DataFrame(graph.out_degree(),columns=['From','Outdegree']).sort_values('From') Stats_2 = pd.DataFrame(graph.in_degree(),columns=['To','Indegree']).sort_values('To') b= epi_csv.groupby('From',as_index=False)[['Weight']].sum().sort_values('From') c= epi_csv.groupby('To',as_index=False)[['Weight']].sum().sort_values('To') Stats = Stats.merge(b,on='From',how='left') Stats_2 = Stats_2.merge(c,on='To',how='left') Stats['Pos_out'] = (Stats['Outdegree'] + Stats['Weight'])/2 Stats['Neg_out'] = (Stats['Outdegree'] - Stats['Weight'])/2 Stats_2['Pos_in']= (Stats_2['Indegree'] + Stats_2['Weight'])/2 Stats_2['Neg_in']= (Stats_2['Indegree'] - Stats_2['Weight'])/2 Stats = pd.merge(Stats,Stats_2,left_on='From', right_on='To').drop('To', axis=1) Stats = Stats.drop(['Weight_x','Weight_y'],axis=1) Stats.fillna(0,inplace=True) Stats.head() Stats.describe() print(Stats.max(axis=0)) Basic stats print(graph.number_of_edges()) print(graph.number_of_nodes()) print(nx.average_clustering(graph)) print(nx.transitivity(graph)) print(nx.density(graph)) Log-Log plots of degrees: Out degree and indegree relationship, relationship between positive and negative weighted outgoing edges in the network and relationship between positive and negative ingoing edges. plt.figure(figsize=(4,4)) plt.plot(Stats['Pos_out'],Stats['Neg_out'],'ro',markersize=0.25) plt.xscale('log') plt.yscale('log') plt.figure(figsize=(4,4)) plt.plot(Stats['Pos_in'],Stats['Neg_in'],'ro',markersize=0.25) plt.xscale('log') plt.yscale('log') plt.figure(figsize=(5,5)) plt.plot(Stats['Outdegree'],Stats['Indegree'],'ro',markersize=0.4) plt.xscale('log') plt.yscale('log') Plotting the assortativity coefficients of the nodes. Deg_cor = nx.average_neighbor_degree(graph,target='out') dict_list = [] for key, value in Deg_cor.items(): temp = [key,value] temp[0] = graph.degree(key,'out') dict_list.append(temp) dfa1 = pd.DataFrame(dict_list,columns =['Outdegree','Average neighbors outdegree']) dfa2 = dfa1.groupby('Outdegree',as_index=False)['Average neighbors outdegree'].mean() plt.plot(dfa1['Outdegree'],dfa1['Average neighbors outdegree'],'ro',markersize=0.3) plt.plot(dfa2['Outdegree'],dfa2['Average neighbors outdegree'],'bo',markersize=1) plt.xscale('log') plt.yscale('log') Deg_cor = nx.average_neighbor_degree(graph,target='in') dict_list = [] for key, value in Deg_cor.items(): temp = [key,value] temp[0] = graph.degree(key,'in') dict_list.append(temp) dfa1 = pd.DataFrame(dict_list,columns =['Indegree','Average neighbors indegree']) dfa2 = dfa1.groupby('Indegree',as_index=False)['Average neighbors indegree'].mean() plt.plot(dfa1['Indegree'],dfa1['Average neighbors indegree'],'ro',markersize=0.3) plt.plot(dfa2['Indegree'],dfa2['Average neighbors indegree'],'bo',markersize=1) ? plt.xscale('log') plt.yscale('log') Checking assortativity print(nx.degree_assortativity_coefficient(graph,'in','in')) print(nx.degree_assortativity_coefficient(graph,'out','out')) Checking the power law import math pos = Stats['Indegree'][Stats['Indegree'] != 0] pos = pos.transform(lambda x: math.floor(math.log(x))) ? prob = pos.value_counts(normalize=True) threshold = 0.0001 mask = prob > threshold tail_prob = prob.loc[~mask].sum() prob = prob.loc[mask] prob['other'] = tail_prob prob.plot(kind='bar',log=True,color='r') import math pos = Stats['Outdegree'][Stats['Outdegree'] != 0] pos = pos.transform(lambda x: math.floor(math.log(x))) ? prob = pos.value_counts(normalize=True) threshold = 0.0001 mask = prob > threshold tail_prob = prob.loc[~mask].sum() prob = prob.loc[mask] prob['other'] = tail_prob prob.plot(kind='bar',log=True,color='r') Assortativity coefficients nx.degree_assortativity_coefficient(graph,'in') nx.degree_assortativity_coefficient(graph,'out') nx.degree_assortativity_coefficient(graph) Size of components for bowtie structure SCC = max(nx.strongly_connected_components(graph), key=len) print('Size of maximal strongly connected component is ' + str(len(SCC))) WCC = max(nx.weakly_connected_components(graph), key=len) print('Size of maximal weakly connected component is ' + str(len(WCC))) Community detection in graph SCC = max(nx.strongly_connected_components(graph), key=len) scc_com = graph.subgraph(SCC).copy() scc2 = nx.Graph(scc_com) ? #