CSCI - 5352 (Network Analysis and Modeling) - Problem Set 2

Name of Student : Rajarshi Basak

September 26, 2018

[1] (a) The expected degree $\langle k \rangle$ of a vertex in group m is given by

$$\langle k \rangle = \sum_{m'=0}^{\binom{n}{2}} \frac{2m'}{n} Pr(m')$$

$$=> \langle k \rangle = \frac{2}{n} \sum_{m'=0}^{\binom{n}{2}} m' Pr(m')$$

$$=> \langle k \rangle = \frac{2}{n} \binom{n}{2} p_m$$

since $\langle m \rangle = \sum_{m'=0}^{\binom{n}{2}} m' Pr(m') = \binom{n}{2} p_m$

$$=> \langle k \rangle = (n_m - 1) p_m$$

Hence substituting the value of p_m, given by

$$p_m = A(n_m - 1)^{-\beta}$$

we have,

$$=> \langle k \rangle = (n_m - 1) A (n_m - 1)^{-\beta}$$

which reduces to

$$\boxed{\langle k \rangle = A(n_m - 1)^{1-\beta}}$$

[1] (b) The expected value $\langle C_m \rangle$ of the local clustering coefficient for vertices in group m is the ratio of the number of triangles to the number of connected triples in the network. Since every edge is independent and identically distributed in a random graph, the relation above reduces to

$$C \propto \frac{\binom{n}{3} p^3}{\binom{n}{2} p^2} = p = \frac{c}{n-1}$$

Thus, for our case

$$\langle C_m \rangle = p_m = \frac{A}{(n_m - 1)^{\beta}}$$

or

$$\boxed{\langle C_m \rangle = A(n_m - 1)^{-\beta}}$$

[1] (c) We have, from **[1] (a)** above,

$$\langle k \rangle = A(n_m - 1)^{1-\beta}$$

Thus,

$$\langle k \rangle^{\frac{-\beta}{1-\beta}} = A^{\frac{-\beta}{1-\beta}} \left((n_m - 1)^{1-\beta} \right)^{\frac{-\beta}{1-\beta}}$$

$$=> \langle k \rangle^{\frac{-\beta}{1-\beta}} = A^{\frac{-\beta}{1-\beta}} \Big(n_m - 1 \Big)^{-\beta}$$

$$=> \langle k \rangle^{\frac{-\beta}{1-\beta}} = \frac{A^{\frac{-\beta}{1-\beta}}}{A} \Big[A(n_m - 1)^{-\beta} \Big]$$

after multiplying and dividing by by A on the numerator and the denominator. Since $\langle C_m \rangle = A(n_m - 1)^{-\beta}$ The equation above reduces to

$$=> \langle k \rangle^{\frac{-\beta}{1-\beta}} = A^{\frac{-(1+\beta)}{1-\beta}} \langle C_m \rangle$$

which can be rewritten as

$$\langle C_m \rangle = A^{\frac{1+\beta}{1-\beta}} \langle k \rangle^{-\beta/1-\beta}$$

Now since A and β are constants in this problem, we have

$$\boxed{\langle C_m \rangle \propto \langle k \rangle^{-\beta/1-\beta}}$$

[2] (a) The expected number of triangles in the network is given by

$$\binom{n}{3} p^3$$

since there are $\binom{n}{3}$ number of ways of selecting any 3 vertices from n vertices, and the probability that all 3 of them have an edge is given by $p * p * p = p^3$.

Now since the average degree $\langle k \rangle = c = (n-1)p$, the expected number of triangles now becomes

$$\binom{n}{3} \Big[\frac{c}{n-1} \Big]^3 = \frac{n(n-1)(n-2)}{6} * \frac{c^3}{(n-1)(n-1)(n-1)}$$

For large n, $n \simeq n-1 \simeq n-2$. Hence in the limit of large n, the expected number of triangles in the network is

$$\boxed{\frac{1}{6} c^3}$$

In other words, the number of triangles is constant, neither growing nor vanishing in the limit of large n.

[2] (b) The expected number of connected triples in the network is given by

$$\binom{n}{3} \binom{3}{2} p^2$$

since there are $\binom{n}{3}$ number of ways of selecting any 3 vertices from n vertices, there are 3 $\big($ $_{2 ways of making a connected triple from th}$ = p^2

Using the relation for the average degree given by $\langle k \rangle = c = (n-1)p$, the expected number of triples is given by

$$\binom{n}{3} \Big[\frac{c}{n-1} \Big]^2 = \frac{n(n-1)(n-2)}{6} * 3 * \frac{c^2}{(n-1)(n-1)}$$

which in the limit of large n (i.e. $n \simeq n-1 \simeq n-2$) becomes

$$\frac{3n}{6} * c^2$$

$$\boxed{\frac{1}{2} n c^2}$$

[2] (c) The clustering coefficient C is given by $C =$ (3* number of triangles)/ (number of connected triples) (from Equation (7.41) in *Networks*)

From **[2] (a)**, the expected number of triangles in the network is $\frac{1}{6}c^3$, and from **[2] (b)**, the expected number of connected triples in the network is $\frac{1}{2}nc^2$. Using these results in the equation described above, the clustering coefficient is defined as

$$C = \frac{3 * \frac{c^3}{6}}{\frac{1}{2}nc^2}$$

or,

$$C = \frac{c}{n} \simeq \frac{c}{n-1}$$

for large n, which agrees with the value for large n given in Equation (12.11) in *Networks*.

[3] Since a single edge divides the network of n vertices into two disjoint regions (i.e. subnetworks of size n_A and n_B), the shortest distance from node A to any node in subnetwork of size n_B is one larger than the shortest distance from node B to all nodes in the same subnetwork of size n_B. Due to this, the sum of distances of all shortest paths from node A given by $\sum_j d_{Aj}$ has the following relation with $\sum_j d_{Bj}$:

$$\sum_j d_{Aj} = \sum_j d_{Bj} + n_B - n_A$$

In other words, here $\sum_j d_{Aj}$ is actually $\sum_j d_{Bj}$ corrected for node A being one closer to all nodes in subnetwork of size n_A and one further from all nodes in subnetwork of size n_B.

Since $n \neq 0$, dividing both sides of the above equation by n, we have,

$$=> \frac{1}{n}\sum_j d_{Aj} = \frac{\sum_j (d_{Bj} + n_B - n_A)}{n}$$

or,

$$=> \frac{\sum_j d_{Aj}}{n} = \frac{\sum_j d_{Bj}}{n} + \frac{n_B}{n} - \frac{n_A}{n}$$

Now, from the definition of closeness centrality as given by Equation (7.29) in *Networks*, $C_A = \frac{n}{\sum_j d_{Aj}}$ and $C_B = \frac{n}{\sum_j d_{Bj}}$ are the closeness centralities of nodes A and B respectively.

Thus the above equation reduces to

$$=> \frac{1}{C_A} = \frac{1}{C_B} + \frac{n_B}{n} - \frac{n_A}{n}$$

or,

$$\boxed{\frac{1}{C_A} + \frac{n_A}{n} = \frac{1}{C_B} + \frac{n_B}{n}}$$

[4] Let us name the vertex in question to be y.

We consider two vertices i and j in the same disjoint region, let us say the uth region. We observe that none of the shortest paths between these two vertices pass through the vertex y. Now, if i and j are two vertices in two different disjoint regions, then the shortest path from vertex i to j necessarily passes through vertex y since this is the only vertex connecting the two disjoint regions. When we remove the vertex y, these two regions get separated.

Since the number of shortest paths/geodesics between l vertices is given by l^2, we conclude that the number of geodesics passing through the vertex y is the sum of all possible geodesics in the graph (given by n^2) minus the number of geodesics for each of the disjoint regions (given by $\sum_{i=1}^{k} n_m^2$).

Hence the betweenness centrality b_y of the vertex y given by

$$b_i = \sum_{jk} \sigma_{jk}(i)$$

where $\sigma_{jk}(i)$ is the number of paths from $j \to k$ that pass through i becomes

$$\boxed{b_y = n^2 - \sum_{i=1}^{k} n_m^2}$$

[5] (a) The table showing the values of the four centralities for each family with each (family, score) pair arranged in decreasing order of centrality is as shown below:

Degree Centrality	**Harmonic Centrality**	**Eigenvector Centrality**	**Betweenness Centrality**
('Medici', 6)	('Medici', 1.667)	('Medici', 0.43)	('Medici', 0.0023)
('Guadagni', 4)	('Ridolfi', 1.867)	('Strozzi', 0.356)	('Guadagni', 0.0016)
('Strozzi', 4)	('Albizzi', 1.933)	('Ridolfi', 0.342)	('Albizzi', 0.0012)
('Albizzi', 3)	('Tornabuoni', 1.933)	('Tornabuoni', 0.326)	('Bischeri', 0.001)
('Castellani', 3)	('Guadagni', 2.0)	('Guadagni', 0.289)	('Salviati', 0.001)
('Bischeri', 3)	('Barbadori', 2.133)	('Bischeri', 0.283)	('Tornabuoni', 0.0009)
('Peruzzi', 3)	('Strozzi', 2.133)	('Peruzzi', 0.276)	('Strozzi', 0.0009)
('Tornabuoni', 3)	('Bischeri', 2.333)	('Castellani', 0.259)	('Ridolfi', 0.0009)
('Ridolfi', 3)	('Castellani', 2.4)	('Albizzi', 0.244)	('Barbadori', 0.0008)
('Barbadori', 2)	('Salviati', 2.4)	('Barbadori', 0.212)	('Castellani', 0.0007)
('Salviati', 2)	('Acciaiuoli', 2.533)	('Salviati', 0.146)	('Peruzzi', 0.0006)
('Acciaiuoli', 1)	('Peruzzi', 2.533)	('Acciaiuoli', 0.132)	('Lamberteschi', 0.0006)
('Ginori', 1)	('Ginori', 2.8)	('Lamberteschi', 0.089)	('Acciaiuoli', 0.0005)
('Lamberteschi', 1)	('Lamberteschi', 2.867)	('Ginori', 0.075)	('Pazzi', 0.0005)
('Pazzi', 1)	('Pazzi', 3.267)	('Pazzi', 0.045)	('Ginori', 0.0004)
('Pucci', 0)	('Pucci', 0.0)	('Pucci', 0.0)	('Pucci', 0.0)

(i) From the table, it is evident that the 'Medici' family is the most important family in the network, as all the centrality values rank the 'Medici' family at the top of the list (i.e. within each centrality, the 'Medici' family occupies the highest rank). Hence the scores agree with the Padgett and Ansell's claim that the 'Medici' occupied a structurally important position in the network.

(ii) According to the degree and betweenness centrality, the second most important family in the network is 'Guadagni', whereas according to the harmonic and eigenvector centrality, the second most important families are 'Ridolfi' and 'Strozzi' respectively.

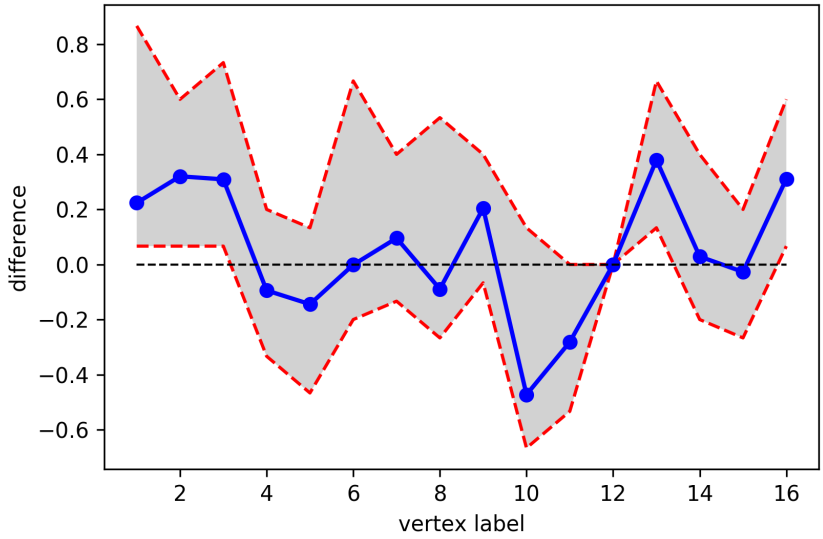
[5] (b) According to the configuration model, the average value of the harmonic centrality for each of the families over an ensemble of 1000 graphs are as follows:

Family	**Average Harmonic Centrality**
Acciauoli	2.757
Albizzi	2.254
Barbadori	2.443
Bischeri	2.239
Castellani	2.256
Ginori	2.799
Guadagni	2.096
Lamberteschi	2.778
Medici	1.871
Pazzi	2.794
Peruzzi	2.252
Pucci	0.000
Ridolfi	2.247
Salviati	2.429
Strozzi	2.106
Tornabuoni	2.245

(i) It is evident from the table above that, again, the most important family in the network in the family is the 'Medici' family, the second most important is the 'Guadagni' family, and the third most important family is the 'Strozzi' family.

(ii) The results from part (5a) have changed slightly as there is slight reshuffling of the spots from 2 to 15, but the winner remains the same.

Figure 1: Figure showing the difference between a node's harmonic centrality on G and its average harmonic centrality in the ensemble



The code for Problem 5 (a) is shown below.

```
import matplotlib.pyplot as plt
import numpy as np
import networkx as nx
import glob
import operator
from math import sqrt

graph = nx.read_adjlist('medici_adj_list.txt')
graph_2 = nx.read_adjlist('medici_edge_list.txt')
total_edges = len(graph.edges())
total_vertices = len(graph.nodes())
mean_degree = 2*(total_edges)/total_vertices
print("The mean degree is",mean_degree)

nodes = graph.nodes()

family_ids = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
families = ["Acciaiuoli", "Albizzi", "Barbadori", "Bischeri", "Castellani",
            "Ginori", "Guadagni", "Lamberteschi", "Medici", "Pazzi",
            "Peruzzi", "Pucci", "Ridolfi", "Salviati", "Strozzi",
            "Tornabuoni"]

family = dict(zip(family_ids, families))

# Computing the degree centrality for each family

def deg_cent(graph):
    degree_centralities = {} # Dictionary for storing the degree centralities
    for j in graph.nodes(): # Iterate through each node in the graph
        degree_centralities[j] = len(graph.edges(j))
       # Storing the number of edges for each node

    sorted_degrees = sorted(degree_centralities.items(),
                            key=operator.itemgetter(1), reverse=True)
    # Sorting the degree centralities in decreasing order of importance
    return sorted_degrees

#Computing the harmonic centrality for each family
```

```
def harm_cent(graph):
    shortest_paths = {} # Dictionary for storing the shortest paths from each vertex
    harmonic_centralities = {} # Dictionary for storing the harmonic centralities
    sum_of_geodesics = 0
    for k in graph.nodes(): # Iterating through each node

        shortest_paths = nx.single_source_shortest_path_length(graph,k)
        # Fidning the SSSP length from each node to all other nodes
        sum_of_geodesics = sum(shortest_paths.values())/(total_vertices - 1)
        # Summing over all the geodesics for each source vertex
        harmonic_centralities[k] = sum_of_geodesics
        # Storing the harmonic centrality for each node
        shortest_paths = {}


    sorted_harmonics = sorted(harmonic_centralities.items(),
                              key=operator.itemgetter(1))
    # Sorting the harmonic centralities in decreasing order of importance
    return sorted_harmonics

#Computing the eigenvector centralities

def eig_cent(G, max_iter=100, tol=1.0e-6, nstart=None,
                            weight='weight'):

    if type(G) == nx.MultiGraph or type(G) == nx.MultiDiGraph:
        raise nx.NetworkXException("Not defined for multigraphs.")

    if len(G) == 0:
        raise nx.NetworkXException("Empty graph.")

    if nstart is None:

        # choose starting vector with entries of 1/len(G)
        x = dict([(n,1.0/len(G)) for n in G])
    else:
        x = nstart

    # normalize starting vector
    s = 1.0/sum(x.values())
    for k in x:
        x[k] *= s
    nnodes = G.number_of_nodes()

    # make up to max_iter iterations
    for i in range(max_iter):
        xlast = x
        x = dict.fromkeys(xlast, 0)

        # do the multiplication y^T = x^T A
        for n in x:
            for nbr in G[n]:
                x[nbr] += xlast[n] * G[n][nbr].get(weight, 1)

        # normalize vector
        try:
            s = 1.0/sqrt(sum(v**2 for v in x.values()))

        # this should never be zero?
        except ZeroDivisionError:
```

```
            s = 1.0
        for n in x:
            x[n] *= s

        # check convergence
        err = sum([abs(x[n]-xlast[n]) for n in x])
        if err < nnodes*tol:
            #return x
            sorted_eigenvector = sorted(x.items(),
                                key=operator.itemgetter(1), reverse=True)
            return sorted_eigenvector

#Computing the betweenness centrality for each family

def bet_cent(graph_2):
    betweenness_centralities = {} # Dictionary for storing the betweenness centralitie
    total_paths = 0
    list_of_paths = []
    #vertex_squared = total_vertices**2
    passes_through_i = 0


    for j in graph_2.nodes():
        for k in graph_2.nodes():
            #path = nx.shortest_path(graph_2, j, k)
            paths = nx.all_shortest_paths(graph_2, j, k)
            # Calculating the shortest path between each pair of nodes
            # and storing it in a list
            #print(paths)
            #if (len(path) != 1):
            for path in paths:
                #print(path)
                list_of_paths.append(path)
                total_paths += 1

    for i in graph.nodes(): # Iterating through each node
        for j in list_of_paths: # Iterating through all the shortest paths
            if (i in j): # If a node is included in a path
                passes_through_i += 1
                # Increment the counter for nunber of paths passing through
        betweenness_centralities[i] = passes_through_i/((total_paths)
        *(total_vertices**2))
        # Update the betweenness centrality dictionary for each node
        passes_through_i = 0 # Reset the counter for the number of paths
        # passing through

    sorted_betweenness = sorted(betweenness_centralities.items(),
                                key=operator.itemgetter(1), reverse=True)
    # Sorting the betweenness centralities in decreasing order of importance

    return sorted_betweenness

# Generating the final lists for all the sorted centralities

cd = deg_cent(graph)
ch = harm_cent(graph)
ce = eig_cent(graph)
cb = bet_cent(graph_2)

list_of_deg_cent = []
list_of_harm_cent = []
```

```
list_of_eig_cent = []
list_of_bet_cent = []

sz = np.shape(cd)
for i in range(sz[0]):
    temp = cd[i]
    ids = int(temp[0])
    print(families[ids])
    list_of_deg_cent.append((families[ids], np.around(temp[1], decimals = 3)))

sz = np.shape(ch)
for i in range(sz[0]):
    temp = ch[i]
    ids = int(temp[0])
    print(families[ids])
    list_of_harm_cent.append((families[ids], np.around(temp[1], decimals = 3)))

sz = np.shape(ce)
for i in range(sz[0]):
    temp = ce[i]
    ids = int(temp[0])
    print(families[ids])
    list_of_eig_cent.append((families[ids], np.around(temp[1], decimals = 3)))

sz = np.shape(cb)
for i in range(sz[0]):
    temp = cb[i]
    ids = int(temp[0])
    print(families[ids])
    list_of_bet_cent.append((families[ids], np.around(temp[1], decimals = 4)))
```

It is to be noted that the code for finding the eigenvector centrality was borrowed from https://www.geeksforgeeks.org/eigenvector-centrality-centrality-measure/

The code for Problem 5 (b) is shown below.

```
import networkx as nx
import matplotlib.pyplot as plt
import operator
import numpy as np


graph_A = nx.read_adjlist('medici_adj_list.txt')

def harm_cent(graph):
    shortest_paths = {}
    harmonic_centralities = {}
    sum_of_geodesics = 0
    total_vertices = len(graph.nodes())
    for k in graph.nodes():
        shortest_paths = nx.single_source_shortest_path_length(graph,k)
        sum_of_geodesics = sum(shortest_paths.values())/(total_vertices - 1)
        harmonic_centralities[k] = sum_of_geodesics
        shortest_paths = {}

    sorted_harmonics = sorted(harmonic_centralities.items(),
                              key=operator.itemgetter(0))
    return sorted_harmonics

hc_A = harm_cent(graph_A)
hc_B = np.zeros((len(graph_A),2))
```

```
data = np.zeros((len(graph_A), 1000))

deg_seq = [1,3,2,3,3,1,4,1,6,1,3,0,3,2,4,3]

graph=nx.configuration_model(deg_seq)

dict_nodes_harm = {}
sum_arr = np.zeros((len(graph),5))
sum_arr[:,0] = range(len(graph))
count = 1

# Generating the random graphs from the degree sequence
while (count <= 1000):
    graph=nx.configuration_model(deg_seq)
    hc = harm_cent(graph)
    print(hc)
    for i in range(len(graph)):
        sum_arr[i,1] += hc[i][1]
        data[i, count-1] = hc[i][1]
    count += 1
    #dict_nodes_harm[]

sum_arr[:,1] = sum_arr[:,1]/1000

for i in range(len(graph_A)):
    hc_B[i,0]=i;
    ind = int(hc_A[i][0])
    hc_B[ind,1] = hc_A[i][1]

for i in range(len(data)):
    #print(i)
    sum_arr[i,2] = np.percentile(data[i,:], 25) - hc_B[i,1] # 25th percentile
    sum_arr[i,3] = np.percentile(data[i,:], 75) - hc_B[i,1] # 75th percentile
    sum_arr[i,4] = sum_arr[i,1] - hc_B[i,1] # Mean value

# Plotting the graph
fig, ax = plt.subplots()
ax.plot(sum_arr[:,0]+1.,sum_arr[:,2],'--',color='red')
ax.plot(sum_arr[:,0]+1.,sum_arr[:,3],'--',color='red')
ax.fill_between(sum_arr[:,0]+1., sum_arr[:,2], sum_arr[:,3],facecolor='gray', alpha=0.
ax.plot(sum_arr[:,0]+1.,sum_arr[:,4],'-o',lw=2,color='blue')
ax.plot(np.linspace(1, 16, 100, endpoint=True),np.zeros((100,1)),'--',lw=1,color='blacl
plt.xlabel('vertex label')
plt.ylabel('difference')
plt.savefig('diff_vs_vertexlabel-plot-2.png', dpi = 300)
plt.show
```

It is to be noted that nx.configuration.model function for generating random graphs given a degree sequence generates graphs that include self-loops and multi-edges, which does not tally with the real-world network of families being connected.

Note: I would like to thank Subhayan De for helping me with the plot.