On the shortcomings of modularity

Mark Ditsworth

January 26, 2019

When analyzing complex networks of heterogeneous nodes, it is often of interest to know if they are assortatively or disassortatively mixed, and by how much. The metric used for this is modularity (Q), calculated by the fraction of edges connecting nodes of the same class, minus this fraction if the edges were assigned at random.

$$Q = \frac{1}{2m}\sum_{i=1}^{n}\sum_{j=1}^{n} A_{ij}\,\delta(c_i, c_j) - \frac{1}{2m}\sum_{i=1}^{n}\sum_{j=1}^{n}\frac{k_i k_j}{2m}\,\delta(c_i, c_j)$$

The term $\frac{k_i k_j}{2m}$ gives the expected number of edges between node i and node j by assuming every node in the network is equally likely to be one of their neighbors, preserving the degree distribution. If Q is positive, the fraction of edges between like nodes is greater than would be expected at random, and thus the network is considered assortatively mixed. Similarly, if Q is negative, the fraction of edges between like nodes is lesser than would be expected, indicated disassortative mixing.

For some networks, this calculation of modularity is flawed due to the randomization assumptions. Take for example the following scenario:

10 people stand in a circle and each flip a fair coin. Assuming it is possible, every person whose immediate neighbors both flipped heads steps forward. One of these people is selected at random (uniformly). What is the probability that they flipped heads? It is tempting to say 50% since each coin flip is independent. However, as shown in the results of the simulations below, conditioning by both neighbors having flipping heads results in about a 43% probability of having flipped heads.

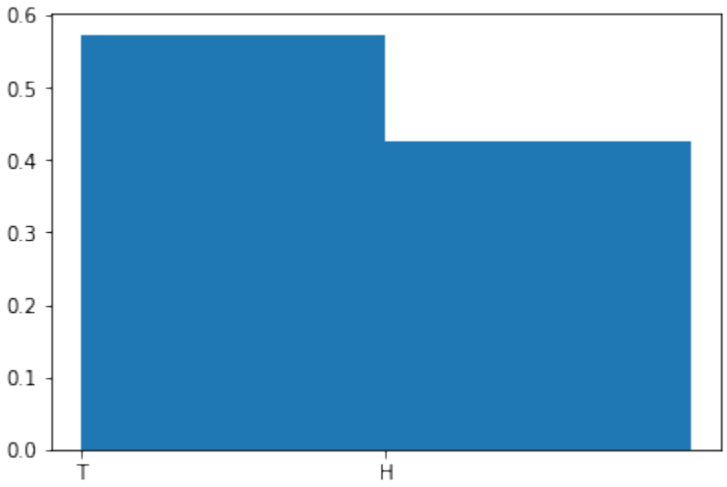
```
In [27]: import numpy as np
         from random import choice
         import matplotlib.pyplot as plt

         results = []

         while len(results)<10000000:
             flips = np.random.randint(0,2,10)
             surrounded_students = []
             for i in [3,4,5,6,7,8,9,10]:
                 group = flips[i-3:i]
                 if group[0]==1 and group[2]==1:
                     surrounded_students.append(group[1])
             # choose one of the students at random
             if len(surrounded_students)>0:
```

```
            results.append(choice(surrounded_students))

    plt.hist(results,bins=[0,1,2],density=True)
    plt.xticks([0,1])
    plt.gca().set_xticklabels(['T','H'])
    plt.show()
```

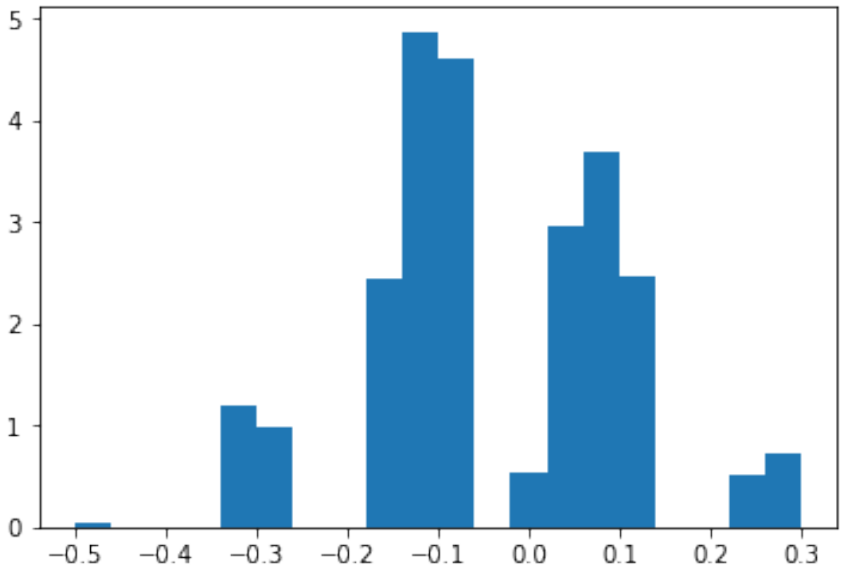


Relating this back to Q, if the ring network described above were to be measured, Q should be expected to be slightly negative on average since HTH patterns are more likely than HHH patterns (equally true for THT versus TTT). This is true, as shown below.

```
In [11]: import zen
         # make ring network
         G = zen.Graph()
         nodes = [0,1,2,3,4,5,6,7,8,9,0]
         for i in range(len(nodes)-1):
             G.add_edge(nodes[i],nodes[i+1])

         modularity = []
         # simulate coin flipping
         for _ in range(100000):
             coin_flips = np.random.randint(0,2,10)
             node_classes = {0:np.where(coin_flips==0)[0],\
                             1:np.where(coin_flips==1)[0]}
             mod = zen.algorithms.modularity(G,node_classes)
             modularity.append(mod)

         plt.hist(modularity,density=True,bins=20)
         plt.show()
         print "Mean:   %.2f"%np.mean(modularity)
         print "Median: %.2f"%np.median(modularity)
```




```
Mean:   -0.05
Median: -0.08
```

However, one may be equally justified in asserting that since these heterogeneous networks are being created at random, **the expected Q should be 0**. The reason the expected Q is not 0 is due to the assumption that node class is fixed and edges are assigned at random. In the scenario given here, edges are fixed and node class is assigned at random. If we were to change the equation for Q to assume stochastic node class, random networks with this process do yield Q of 0.

```
In [14]: from random import choice
         def node_modularity(G,classes,simulation_number=100):
             # fraction of edges belonging to same class
             count = 0.0
             for edge in G.edges():
                 if classes[edge[0]] == classes[edge[1]]:
                     count += 1
             frac = count / G.num_edges

             # fraction of edges belonging to same class
             #  if node classes were random
             frac_rand = []
             class_options = list(set(classes))
             for i in range(simulation_number):
                 classes_random = []
                 for _ in range(G.num_nodes):
                     classes_random.append(choice(class_options))
```

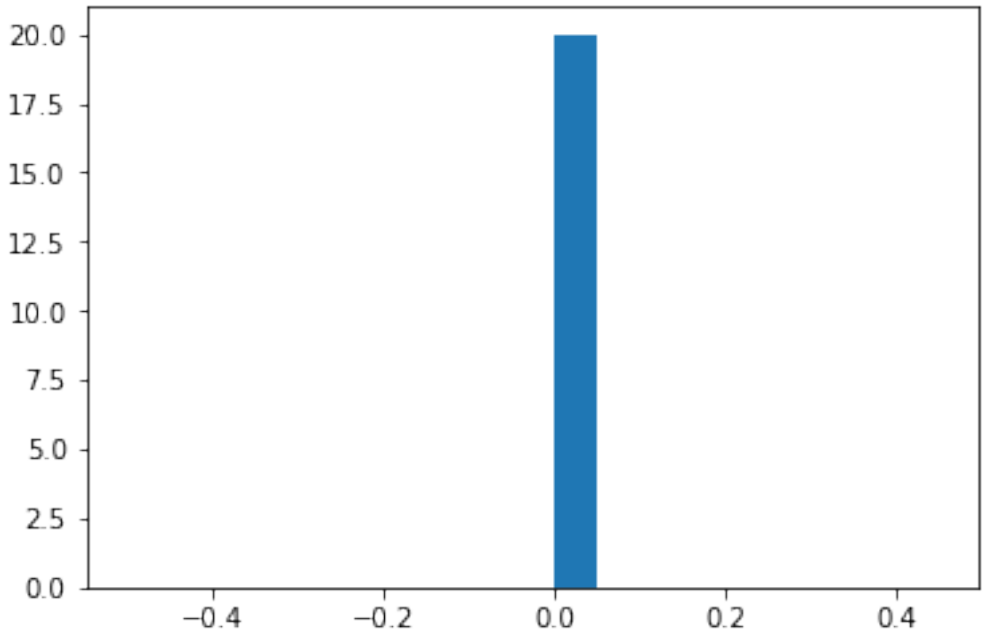
```
            count = 0.0
            for edge in G.edges():
                if classes[edge[0]] == classes[edge[1]]:
                    count += 1
            frac_rand.append(count / G.num_edges)

        modularity = frac - np.mean(frac_rand)
        return modularity
```

In [28]:
```
# make ring network
G = zen.Graph()
nodes = [0,1,2,3,4,5,6,7,8,9,0]
for i in range(len(nodes)-1):
    G.add_edge(nodes[i],nodes[i+1])

modularity = []
# simulate coin flipping
for _ in range(10000):
    coin_flips = np.random.randint(0,2,10)
    #node_classes = {0:np.where(coin_flips==0)[0],\
    #                1:np.where(coin_flips==1)[0]}
    mod = node_modularity(G,coin_flips)
    modularity.append(mod)

plt.hist(modularity,density=True,bins=np.arange(-0.5,0.5,0.05))
plt.show()
print "Mean:   %.2f"%np.mean(modularity)
print "Median: %.2f"%np.median(modularity)
```



```
Mean:   0.00
Median: 0.00
```

When using Q to quantify assortative or disassortative mixing, it should be confirmed that the context of the network modeling is consistent with randomized edge selection. If node class is the random process, then the proposed `node_modularity` calculation should be used instead.