Exercise 10: Networks 2

Kristian Urup Olesen Larsen[1]

[1]kuol@econ.ku.dk

April 22, 2020

Problem 10.1.1. - Null models

To describe networks in a statistically rigorous way we need a comparison network - this is exactly like in the regular econometrics where we often compare parameter estimates $\hat{\beta}$ to the exact value 0 and ask if we can confidently say that our data has $\beta \neq 0$. The null of $\beta = 0$ is equivalent to assuming that y and x are unrelated, i.e. y and x are relatively random. In networks the relevant comparison is more complicated but essentially the same thing. A null model is a way to permute an original network to form random comparison networks. Comparing to a null model can then teach us if specific features of our network are significantly non-random. Unlike in linear regression there are many ways in which a network can be random and the exact choice of null model must match the kinds of questions we want to answer.

To measure if the diameter of a network is statisticially different from random under a null with random degree-preserving edge swaps you would compute the difference in diameter between your real network and the diameter in k permuted null models. If this difference is significant our network has a significantly non-random diameter.

Problem 10.1.2. - Degree preserving edge swaps

The double edge swap preserves node degrees by matching pairs of nodes when edges are swapped, so that all nodes that loose an edge also gains a new one. This will render a network in which each node has the same degree as the original, but where edges are randomly shuffled. Note that sometimes a swap is not possible, e.g. if $u - v$, $x - y$ and $u - x$. In this case swapping $u - v$ and $x - y$ will remove an edge. To avoid getting stuck in these cases the algorithm will try a maximum of `max_tries` times before terminating.

Problem 10.1.3. - Facebook data

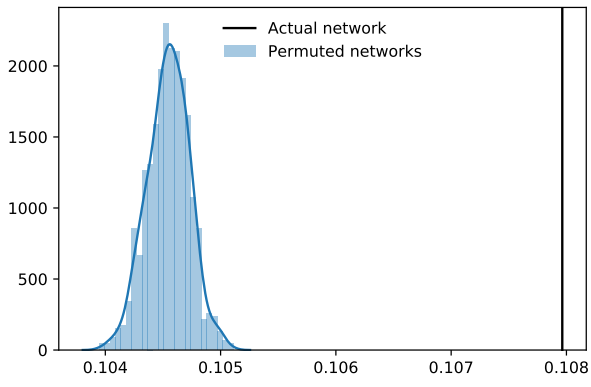

Fig 1. Average clustering coefficients in permuted networks.

The first code for this question can be taken directly from last weeks exercises. This time create an undirected graph instead of the directed one we made last time. After loading the network data I create a networkx graph G. The ALCC of the real facebook network can then be computed via

```
truth = nx.average_clustering(G)
```

which comes out to be ≈ 0.108. Remember that the edge swaps are made in place so G will change as you permute the edges and you will have to reload the data to get back this result unless you copy G. To keep things structured let us begin by writing a simple worker function that does all of the heavy computations for us

```
def worker(G):
    G_ = G.copy()
    G_ = nx.double_edge_swap(G_, nswap=1000, max_tries = 5000)
    return nx.average_clustering(G_)
```

We can then easily write a loop that calls this worker 1000 times. To keep our impatience at bay we can use tqdm to add a progress bar to the loop.

```
from tqdm import tqdm
alcc = []
for _ in tqdm(range(1000)):
    alcc.append(worker(G))
```

Notice we could have written this code as a list-comprehension, but in this case running worker takes far longer than managing the loop. For that reason a list-comprehension will take almost as long time to run as the loop. Once the loop completes we can compute the p-value by computing the share of permuted networks that has an ALCC at least as large as the original network. This should be straight forward to do, and looking at figure 1 clearly suggests that clustering in the facebook network is higher than expected under random links.

Problem 10.2.1. - Equation explanation

The modularity

$$Q = \frac{1}{2m} \sum_{ij} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j) \tag{1}$$

- The summation runs over all node pairs ij in the graph.
- The dirac delta can be written in python as

  ```
  def dirac(a,b):
      return int(a == b)
  ```

  It ensures only nodes in the same cluster contribute to the total value of Q. This makes sense since Q is a kind of "cluster quality measure".
- $k_i k_j / 2m$ is the expected weight of the link between nodes i, j under random degree-preserving edge placement. In an unweighted graph without self-loops this is the same as the probability that i and j are connected. To see that this is indeed the expected number of connections note that an edge from i (a stub) can connect to a total of $2m - 1$ other stubs. The number of "chances" there are that i, j connects is $k_i k_j$. The expected number/weight of connections is the number of ways i and j can connect divided by the total number of possible connections and $2m - 1 \approx 2m$ for large m.

- Inside the brackets is "excess connectivity" i.e. how much more nodes are connected compared to a network with randomly placed edges. To get a measure of clustering we want to sum excess connectivity within each assigned cluster.
- The normalization $1/2m$ scales the modularity by the total number of stubs available to form connections.

Problem 10.2.1. - Modularity implementation

This problem is all about converting the math in (1) into code. As Ulf states the way to approach this is to think about the math "from the inside out". Assuming you have a function dirac(a,b) let us first note that c is stored as a list so that $\delta(c_i, c_j) =$dirac(c[i], c[j]). Next, A_{ij} can simply be found by taking the ith row and the jth column of A, A[i,j] and $k_i =$A[i,:].sum(), $k_j =$A[:,j].sum(). The number of edges m can be found by summing the whole matrix and dividing by two to avoid double counting A.sum()/2.

To implement the sum $\sum_{ij}$ we need all combination of i and j - something we can easily get with two nested for-loops. With this knowledge ready we can easily implement the modularity equation as a python function.

```
def modularity(A, c):
    n = A.shape[0]    # Matrix is symmetric
    Q = 0
    m = A.sum() / 2
    for i in range(n):
        ki = A[i,:].sum()
        for j in range(n):
            kj = A[:,j].sum()
            Q += (A[i,j] - ki*kj/(2*m)) * dirac(c[i], c[j])
    return Q/(2*m)
```

Problem 10.2.1. - Optimal labeling

To find the optimal labeling of the clusters I will implement the louvain method. The algorithm is described in the exercises so let us not spend time on that here, but rather jump straight into the code. Since we will be swapping out elements of c a lot while keeping track of the original element in c it will be useful to define the following helper function

```
def put(array, index, value):
    _arr = array.copy()
    _arr[index] = value
    return _arr
```

which simply places a new value at a given index position in an array without overwriting the original data.

Now on to implementing the actual louvain algorithm. The first step will be to initialize a clustering that assigns each node to it's own cluster and compute the baseline modularity of this clustering

```
def louvain(A, max_iter = 50):
    c = np.arange(A.shape[0])
    Q = modularity(A,c)
    old_Q = None
```

Next we enter a loop that governs the maximum number of iterations the algorithm can take. I also have an early-stopping criteria here, that ensures the algorithm stops when Q is no longer improving.

```
for _ in range(max_iter):
    if old_Q == Q:
        break
```

Then let us pick out a random node j in the network, find all it's neighbors and compute the change in modularity when moving i into the cluster of each of its neighbors

```
i = np.random.randint(A.shape[0])
js = np.where(A[i, :] == 1)[0]
delta_Qs = [modularity(A, put(c, i, c[j])) - Q for j in js]
```

If none of the possible changes to c improved the modularity we go directly to the next iteration in the loop

```
if max(delta_Qs) <= 0:
    continue
```

Otherwise we have found an improved cluster assignment which we will use as the new baseline for the following iterations. In this case we set c to the best alternative clustering we have tried and update the modularity score to reflect this new clustering.

```
else:
    j_star = js[np.argmax(delta_Qs)]
    c = put(c, i, c[j_star])
    old_Q = Q
    Q = modularity(A, c)
```

Finally the function should `return` c. Running `louvain(A)` should then give a result that clusters the first three nodes and the last three nodes which is very intuitive if you take a closer look at the adjacency matrix.