Exercise 9: Networks 1

Kristian Urup Olesen Larsen[1]

[1]kuol@econ.ku.dk

April 16, 2020

Problem 9.1.1. - making a nx graph

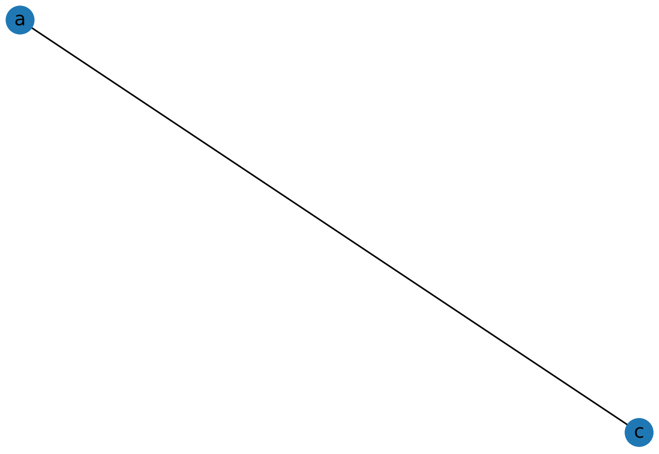

Fig 1. A very simple two-node network.

The first step of creating our first graph will be to create an instance of nx.Graph. Because we use nx.Graph and not nx.DiGraph our graph will be *undirected*.

```
g = nx.Graph()
```

I will add three nodes to the graph, 'a', 'a', and 'c'. You can do this one at a time with g.add_node, or all at once as i do here

```
g.add_nodes_from(['a', 'b', 'c'])
```

Notice that there is no reassignment here, i.e. we don't write g = g.add_nodes_from(...). To understand why, consider a dummy-class that mimicks the functionality of nx.Graph, it might look something like

```
class DummyGraph:
    def __init__(self):
        self.nodes = []
        self.edges = []

    def add_nodes_from(self, nodes):
        for node in nodes:
            self.nodes.append(node)
```

Notice that DummyGraph.add_nodes_from does not return anything (and thus defaults to returning None). If we tried to write g = g.add_nodes_from(...), g would be None. However when add_nodes_from is called it directly modifies the instance self. Many modules implement their classes differently. For instance in Pandas, the reassignment pattern is fine and we use it all the time. Which one to use depends on the package you are using so you will have to get used to both.

With this detour out of the way, let us add some edges and remove the 'b' node.

```
g.add_edges_from([('a', 'b'), ('a', 'c')])
g.remove_node('b')
```

Finally we can draw the network; all of the networkx plots are made with matplotlib so you can use all of the matplotlib code you know to modify the figure.

```
nx.draw(g, with_labels = True)
plt.savefig('writeup/first_network.pdf')
```

Problem 9.1.2 - The Erdős-Rényi graph

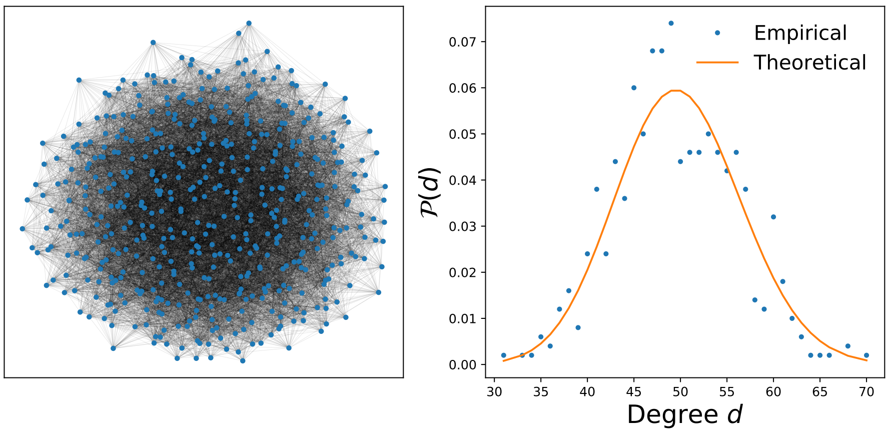

Fig 2. Erdős-Rényi network with $n = 500$ and $p = 0.1$.

The ER graph $G_{n,p}$ is a random graph constructed by fixing n and then independently for each pair of nodes forming a connection with probability p. Notice that even for p close to 1 there is still a non-zero probability that the network is very loosely connected. Let us begin by creating an ER-network. I will give it 500 nodes instead of the 100 mentioned in the exercise to reduce the role of randomness a little.

```
ER = nx.gnp_random_graph(n=500, p=0.1)
```

Now i want both the empirical degree distribution, which is simply the percentage of nodes with a given degree k, calculated for all relevant values of k, and the theoretical degree distribution given by

$$p(k) = \binom{n-1}{k} p^k (1-p)^{n-1-k}. \tag{1}$$

To compute the empirical distribution I will need to count the number of occurences of each degree k in the network, and for the theoretical distribution I need a function that computes n choose k, so i will import a counter and a function that computes $\binom{n}{k}$,

```
from collections import Counter
from scipy.special import comb
```

with these two tools i am ready to write a function that computes all of the things I need for my figure. It will need a network as input, and the value of p, since this enters in the equation above, and is not stored in the network object.

```
def compute_degree_distribution(network, p):
    N = network.number_of_nodes()

    # Compute the empirical degree distribution
    degrees = sorted([d for n, d in network.degree()])  # Degree of each node
    counts = Counter(degrees)  # Count the number of times each value of d appears
    degrees, counts = zip(*counts.items())   # extract into two lists
    empirical_probs = [c/N for c in counts]    # convert from counts to probabilities.

    # Compute the theoretical degree distribution
    theoretical_probs = [comb(N - 1, k) * (p**k) * (1-p)**(N - 1 - k) for k in degrees]

    return degrees, empirical_probs, theoretical_probs
```

With this function we can compute the degree distribution of the ER network we created above.

```
degr, probs, ther = compute_degree_distribution(ER, 0.1)
```

Now we have all the ingredients we need to create the figure. I first set up a two-panel figure and plot the network on the first panel. I use some of the more detailed drawing methods to control the size and position of the nodes and edges.

```
fig, ax = plt.subplots(1,2, figsize = (10,5))
pos = nx.spring_layout(ER)
nx.draw_networkx_nodes(ER, pos, node_size = 10, with_labels=False, ax = ax[0])
nx.draw_networkx_edges(ER, pos, alpha = 0.1, width=0.1, ax = ax[0])
```

Then on the second panel we can draw the empirical and theoretical degree distribution. I plot the empirical distribution as dots to reduce clutter in the figure.

```
ax[1].plot(degr, probs, '.', label = 'Empirical')
ax[1].plot(degr, ther, label = 'Theoretical')
ax[1].set_xlabel('Degree $d$', fontsize = 20)
ax[1].set_ylabel('$\mathcal{P}(d)$', fontsize = 20)
ax[1].legend(frameon=False, fontsize = 16)
```

Problem 9.1.3 - The Barabási-Albert network

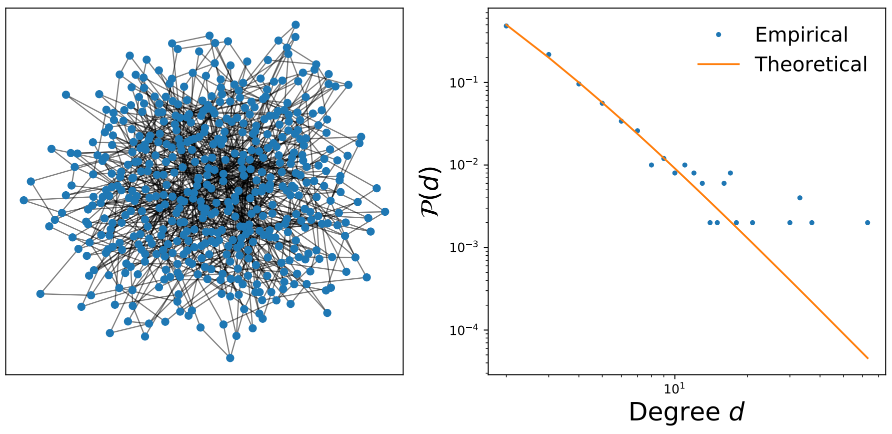

Fig 3. Barabási-Albert network with $n = 500$ and $m = 2$.

The Barabási-Albert network is a model intended to explain how many real-world networks take shape, and in particular why the topology of real networks is very different from the one produced by e.g. the Erdős-Rényi model. The BA model is generative, in that it add nodes to a small starting network. As new nodes are added to the network, the probability that they form a connection to each of the existing nodes is proportional to the degree of each existing node.

The code for this problem is almost identical to the one used for the ER network. In this problem I simulate the network using nx.barabasi_albert_graph and use a modified version of compute_degree_distribution which implements the following equation for computing the theoretical degree distribution

$$p(k) = \frac{2m(m+1)}{k(k+1)(k+2)} \tag{2}$$

I also draw the degree distribution with log axes because $p(k) \approx 2m^2k^{-3}$.

Problem 9.2.1 - Time slice function

In this question we are asked to convert the sample code into a function that takes in a dataset of connections, subsets to the links happening between t_0 and t_1 and returns a nx.DiGraph of the sliced network. Notice that the timestamps count seconds since some reference time (Jan 1st 1970). To get a human friendly representation we can use the datetime module. This can be a bit difficult to work with (dates are difficult). Let us begin by importing the module and take a look at the dates in t0 and t1

```
from datetime import datetime as dt
print(dt.fromtimestamp(t0), dt.fromtimestamp(t1))
```

By wrapping datetime.fromisoformat, which convert strings formatted like 'yyyy-mm-dd-hh' into datetime objects, and datetime.timestamp we can pass t0 and t1 as easily readable strings. After doing this conversion the function is more or less copy paste of the code given in the exercise

```
def create_slice(data, t0, t1):
    if isinstance(t0, str):
        t0 = dt.timestamp(dt.fromisoformat(t0))
    if isinstance(t1, str):
        t1 = dt.timestamp(dt.fromisoformat(t1))

    slice = data.loc[data.timestamp.between(t0, t1)]
    slice = slice.groupby(['user1', 'user2']).size().reset_index(name='weight')
    return nx.from_pandas_edgelist(slice, 'user1', 'user2', 'weight', create_using=nx.DiGraph)
```

Now we can either use the function with the already calculated t0 and t1, or we can give dates as strings:

```
DG = create_slice(data, '2008-01-23', '2009-01-23')
```

Problem 9.2.2 - Degree distributions

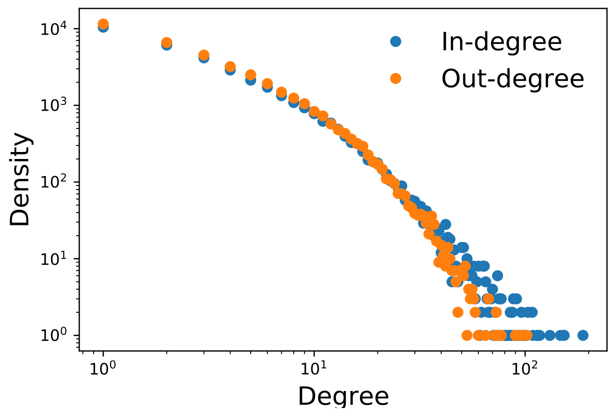

Fig 4. In- and out-degree of the facebook wall network over the last year of data.

Now that we have a directed network there are two degree distributions to plot. I will first write a function that i probabily should have written earlier. I need a function that accepts input like the one produced by nx.Graph.degree and returns the degree distributions. This code is very similar to the one I have already written for problem 9.1.2.

```
def degree_distribution(degrees):
    degrees = sorted([d for n, d in degrees])  # degree sequence
    counts = Counter(degrees)
    degrees, counts = zip(*counts.items())
    return degrees, counts
```

With this function computing and plotting the degree distributions is as easy as

```
degin, cntin = degree_distribution(DG.in_degree())
degout, cntout = degree_distribution(DG.out_degree())

plt.plot(degin, cntin, 'o', label = 'In-degree')
plt.plot(degout, cntout, 'o', label = 'Out-degree')
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Degree', fontsize=16)
plt.ylabel('Density', fontsize=16)
plt.legend(frameon=False, fontsize=16)
```

Problem 9.2.3 - Clustering coefficients

Clustering measures the degree of local connectivity in a network. Practically we can compute this clustering coefficient for each node in the network with nx.clustering. We must remember that our network is weighted with weights being the total number of ineractions between user *i* and user *j*,

```
clustering = nx.clustering(DG, weight='weight')
```

Like the degree distribution the clustering coefficient are highly non-normal. To get a sensible visualization we can plot a log-binned histogram. To do this we first need logspace bins. To construct these we need the min and max bounds of the coefficients. Because the log of 0 is divergent we will not take the min directly. Instead we will pick the *second smallest* value,

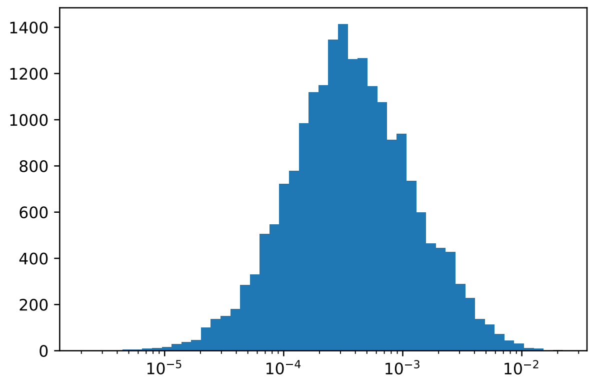

Fig 5. Node clustering coefficients.

```
emin = sorted(set(clustering.values()))[1]
emax = sorted(set(clustering.values()))[-1]
```

Now we can construct the bins with `np.logspace` and plot the resulting histogram.

```
  ebins = np.logspace(np.log10(emin), np.log10(emax), 50)
  plt.hist(clustering.values(), bins=ebins)
  plt.xscale("log")
```

Problem 9.2.4 - Visualizing the network

This should be as simple as calling `netwulf.visualize` on your network.