Lab 4: Large Scale Structure, Random Networks

Download the stub code and rename it as your own for the following exercises. At the top of the file create a commented out section (using #) and place your names and NetIDs there. Please make sure you label your print statements, so we know what you are printing.

Small World

Many, if not most, networks evidence the small world property made popular by pop-culture references to "six degrees of separation" - or possibly the game "six degrees of Kevin Bacon", in which you attempt to link any actor/actress to Kevin Bacon with no more than six connections (through movies). What we would like to show in this section of the lab is that this small world effect is a fundamental network effect. It comes about because of the types of connections that exist in most types of networks. To see this more clearly we are going to introduce our first random graph (we'll see more of these later in the lab). Researchers developed this type of undirected graph for this very purpose and called the networks it generates *small world networks*.

A small world network is created based on a number of nodes, `n`, the number of local connections, `q` (which must be even), and the probability to randomly connect nodes, `p` (which is between 0 and 1). First, `n` nodes are added to the network and formed into a big circle according to their index. Then each node is connected to the `q` nodes closest to it along the circle - so each node is connected to the `q/2` nodes next to it on its "left" and the `q/2` nodes next to it on its "right". The second step is to randomly connect any two nodes according to a certain probability. This means we pick every pair of nodes and essentially flip a coin (a coin with unequal probabilities for heads and tails: `p` for heads and `1-p` for tails). If the coin is heads, then we create an edge between the two nodes. If it is tails, we do not add an edge.

In the stub code, I already have functions that create the small world network and also have a few lines below that call the visualizer. Run the code without modification and it will (in the visualizer) show you the network created in the first step with only local connections, and then after a few seconds it will add the random connections.

You don't have to calculate the diameter exactly, but write down your (visual) observations about the diameter of the network with only the local connections and then with the random connections added in. In the first case, without random connections, why is the diameter so high?

Let's go a little bit further to investigate this. Uncomment the next two sections of code. The parameter `p` controls the amount of random connectivity that is added to the network. Let's see what happens as we change `p` from a small value (near 0) to a large value (near 1). I already have the `for` loop set up for you with a new small world network being generated with increasingly larger `p`. Your job is to calculate the diameter (recall the `zen.diameter()` function) and store it in a list. Once you have that stored, use the next second chunk of code to plot the diameter versus the `p` values. The small world effect claims that the diameter of the graph should scale proportional to log(n). In the second plot statement, I've already plotted this value.

Make a brief comment on the how and why these random connections (qualitatively) lead to the small world effect.

Power Law Networks

As we have mentioned in class, the degree distributions of many networks - or at least part of the degree distributions - can often be captured well by a power law model: $p_k = Ck^{-\alpha}$. Here, we will take a closer look at fitting data from three real networks to a power law.

1. **`japanese.edgelist`**
   ```
   Word-adjacency (directed) networks, in which each node represents a word and a directed connection occurs
   when one word directly follows the other in the text.

   Citation: R Milo, S Itzkovitz, N Kashtan, R Levitt, S Shen-Orr, I Ayzenshtat, M Sheffer & U Alon,
   Superfamilies of designed and evolved networks. Science, 303:1538-42 (2004)
   ```
2. **`ca-HepTh.edgelist`**
   ```
   Arxiv HEP-TH (High Energy Physics - Theory) collaboration network is from the e-print arXiv and covers
   scientific collaborations between authors papers submitted to High Energy Physics - Theory category. If
   ```

```
an author i co-authored a paper with author j, the graph contains a undirected edge from i to j. If the
paper is co-authored by k authors this generates a completely connected (sub)graph on k nodes.
```

```
The data covers papers in the period from January 1993 to April 2003 (124 months). It begins within a few
months of the inception of the arXiv, and thus represents essentially the complete history of its HEP-TH
section.
```

```
Citation: J. Leskovec, J. Kleinberg and C. Faloutsos. Graph Evolution: Densification and Shrinking
Diameters. ACM Transactions on Knowledge Discovery from Data (ACM TKDD), 1(1), 2007.
```

3. **`soc-Epinions1.edgelist`**

```
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.
```

```
Citation: M. Richardson and R. Agrawal and P. Domingos. Trust Management for the Semantic Web. ISWC, 2003.
```

Ultimately, we are interested in computing the exponent α. Recall the formula discussed in class (equation 8.6 in Newman):

$$\alpha = 1 + N\left[\sum_i \ln\left(\frac{k_i}{k_{\min} - \frac{1}{2}}\right)\right]^{-1}$$

It depends on the cutoff degree k_{min}, N is the number of nodes with degree greater than or equal to k_{min}, and the sum is performed over only the nodes with $k_i \geq k_{min}$. Here, we will simply use visual inspection to select k_{min}. There is a corresponding quantification of the uncertainty in equation (8.7) in Newman.

Since the power law model is meant to represent the degree distribution, this is a good place to start. As we've discussed, we can overcome some challenges of working directly with the degree distribution, by instead looking at the cumulative degree distribution. The exponent of the degree distribution is α, whereas the exponent of the cumulative degree distribution is α-1. `zen` has functions for both of these quantities: `zen.degree.ddist(G,normalize=False)` and `zen.degree.cddist(G,inverse=True)`. `ddist` returns a NumPy array such that the k^th element is the number of nodes with degree k (i.e., `v[k]`). If you set the `normalize=True`, then the kth element is the fraction of nodes with degree k. This is the same as dividing by `G.num_nodes`. `cddist`, with the `inverse` flag set, returns the cumulative degree distribution, so the kth element is the fraction of nodes with degree k or more (notice that `cddist` is automatically normalized). Both of these functions have the ability to separately report the in or out degrees (you can do this by setting the parameter `direction`), but we won't go into this level of detail for this lab.

Use these functions to create two plots - one of the degree distribution, which will be a bar plot, and one of the cumulative degree distribution, which should be plotted on a log-log plot (where both axes are logarithmically spaced). The code to plot this in Python is already in the stub code, but you have to insert what the "x" and "y" values should be. Use these plots to determine a good k_{min}. This will specify the bottom of the range of degrees for which the cumulative degree distribution plot appears linear. I've selected these graphs on purpose so that they show you varying levels of power-law behavior. One of them doesn't really follow a power-law, but find a fit for it anyway. Recall that the power-law is a tail phenomena, describing the relative number of hubs, hence we look to fit the tail of the distribution with a power-law degree distribution.

Now that you have your k_{min} value in hand, you'll be able to calculate α and the corresponding uncertainty, σ. There is a subtle point in the equation for αthat is easy to miss. In particular, the k_i is really the *degree sequence* - not the degree distribution - so if 7 nodes have degree of 4, then k_i=4 is repeated 7 times in the sum of this formula.

Include your plots in the PDF document you submit.

Giant Component in the Erdos-Renyi Random Graph

One of the important phenomena related to the random graph model `G(n,p)` is the sudden emergence of the giant component when the average degree exceeds a value of one. Your goal is to recreate the figure that plots the fraction of nodes in the giant component (y-axis) against the average degree (x-axis). `zen` has several random network models built in, available in the `zen.generating` module. For this section you will use the function `zen.generating.erdos_renyi(n,p)`. You are also likely going to want to use the `zen` function `zen.algorithms.components(...)` to identify the size of the largest component in the network. This function returns a list of sets, where each set is a component of the graph and contains the nodes (by node object) within the component. Another useful function will be `numpy.arange(a,b,step)`, where this generates points from `a` up to, but not including `b` by steps of `step`, e.g., `numpy.arange(0,2,0.5) = [0.0,0.5,1.0,1.5]`. While you can compute the data for this plot several different ways, let's specifically plot fraction of nodes in the giant component against the *expected* average degree – this will let you pick a value for `c` and then determine the value of `p` that should (on average) generate a graph with this expected average degree.

You have now seen several examples of plotting, so use that same general structure to make this plot. It should look something like: `plt.plot(c,S,'b',linewidth=2)` to plot the list `c` as the x-values and `S` as the y-values and make the line blue with thickness 2.

Once you have your plot, you'll notice that the line is pretty rough and in some cases it may go up and down, even though the theoretical results predicts a curve for the expected giant component size that is monotonic with increasing average degree. Why are you seeing this non-monotonic behavior?

The solution to overcoming this (which you should now implement), is to average the values of component size over many instances of the "same" random graph, i.e., generate many random graphs with the same parameters (`n`, `p`) and take the average of their giant component sizes. This is a common approach when conducting empirical work with random networks. Because these models represent a family – a distribution in fact – of graphs, studying just one instance is rarely enough to make a general assessment related to the model. An easy way to compute the average of many instances is to put them all into a NumPy array, let's call it `a`, (it has to be an array, not a list, to do this next step) and then use `a.mean()` (this function is actually a method of the object type NumPy array, but it is the same as this general NumPy function of the same name `numpy.mean(...)`). To be clear about the variation we are seeing between different graph instances (essentially to tell people how representative the mean statistic is), we should also calculate the standard deviation of the values, which we can do with `a.std()` (see similar function `numpy.std(...)`). To plot the standard deviation, we need error bars, which can be done easily with `plt.errorbar(c,S_mean,yerr=S_std)`, where `yerr` is the size of the error bars.

Let's use this plot to observe the "convergence" of the Erdos-Renyi model to its expected behavior as the graph gets larger. Make three plots, one for `n=10`, one for `n=100`, and one for `n=1000`. Now that you are essentially going to use the same code more than once, except for this parameter change, good programming practice says that you should put this into a function that you can then call as many times as you need. So create a function that creates this plot and takes as input arguments the number of nodes and the number of graphs to average over (this will involve moving most of your current code into a new function). Now just call that function three times, for `n=10`, `n=100`, and `n=1000`. The call for `n=1000` will likely take a couple of minutes to calculate, so stretch and practice some yoga while you wait. Observe the differences in the plots and explain how this supports the notion that the statistics of the model converge to their expected values as `n` gets larger. When you turn in your code, comment out the command for the `n=1000` version (this will save us a lot of time grading, and anyway we've already practiced our yoga).

For extra insight (this paragraph is just for your own understanding and not to be turned in), set the number of graphs that you are averaging over to be 1 again. Now the curve is generated from just one Erdos-Renyi graph again. You can see the effect of convergence very clearly!

Visualizing Methods of Formation

Because this section doesn't have a deliverable (nothing to turn in), you should create this in a separate python file. We will use the visualizer to watch how networks are created under various methods of formation. `zen` has implementations of Erdos-Renyi and Barabasi-Albert models already, however, they are not ready to use with the visualizer, so I have reimplemented them in the file `randomnet.py`, which should already be a part of the `zend3js` folder. Since you will already import that folder for the visualizer, just add `import randomnet` as well in your import statements. The documentation of the built-in versions still apply to the `randomnet` versions, only the new ones also take a graph parameter as well. The built-in generating methods for Local Attachment and Duplication Divergence are, however, ready to use. In each case, you will want to create an empty graph: `G = zen.Graph()` (or `zen.DiGraph()`) and give this to the visualizer. Then call the formation method, i.e., one of the following, to watch the graph be built:

```
G = randomnet.erdos_renyi(30,0.1,graph=G)
G = zen.generating.local_attachment(30,3,2,graph=G)
```

```
G = randomnet.barabasi_albert(30,4,graph=G)
G = zen.generating.duplication_divergence_iky(30,0.1,graph=G)
```

Note that Local Attachment is the only type one that you have to use a `DiGraph` (the others have directed versions, however, it is simpler just to use the undirected version for now). Also, you have already seen the small world network be built in a previous lab. Feel free to adjust the parameters of the model and observe the changes in the output.

Degree Distributions of Random Network Models

We discussed several models of network formation in lecture, namely, Erdos-Renyi (ER), small-world (SM), Barabasi-Albert (BA, preferential attachment), Local Attachment (LA), Duplication Divergence (DD). We noted that in the limit, ER graphs have a Poisson degree distribution; that BA, LA, and DD have scale free distributions; and that SM has a vaguely right-skewed distribution.

Earlier in this lab, we plotted the degree distributions of several real world graphs. Use this same code to look at the degree distributions and cumulative degree distributions of these networks. Do this for small, intermediate, and large values of the various parameters that determine these networks, however, fix the number of nodes to be some large number (e.g., 10,000). Save these figures and create a summary as part of your PDF submission. In this document, make sure the label the plots with their corresponding network model and parameters. Also note the ones that exhibit scale-free degree distributions. What are their power-law exponents?

Fitting Random Models

While random network models are used to understand how a formation mechanism leads to network properties, they are also used as coarse approximations of real world networks. To get a sense for how to tune network models, and observe some deficiencies, try selecting parameters for the ER, LA, and DD models to match the basic network properties of a real world network including number of nodes and edges (and hence the average degree), global clustering, and degree distribution (use the same code you developed earlier to plot the degree distribution). In some cases, we have discussed analytic relationships between a parameter and these network statistics – for example, there is an expression that relates the ER probability `p` to the expected number of edges in the graph. For other parameters, you will have to make educated iterative guesses (or some sort of regression). In addition to these random network models, also investigate the properties of the configuration model. In the stub code there are functions to return the degree sequence of a graph and also the return an instance of the configuration model, given a degree sequence. In your PDF, present the actual statistics for the `ca-Hep` network and the statistics for the fitted ER, LA, DD, and configuration model networks.

Configuration Model

One of the outcomes from analyzing the configuration model was the friendship paradox, in which a node's neighbors appear to have a more-then-average number of connections. Compute this for the two networks:

1. **`texas_road_sample.edgelist`**
   This is a snowball sample of a road network of Texas attained from SNAP. Intersections and endpoints are represented by nodes, and the roads connecting these intersections or endpoints are represented by undirected edges. The original network has over 1.3 million nodes; this network has been sampled (in snowball fashion) to have around the same number as the airport network below.
   Citation: J. Leskovec, et al. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. Internet Mathematics, 2009.

2. **`international_airports.edgelist`**
   This dataset provides a graph of international airports. Vertices represent airports, and edges exist wherever there were flight(s) between the airports. Edges are weighted by the number of flights between airports.
   Citation: Opsahl, T. Why Anchorage is not (that) important: Binary ties and Sample selection, 2011.

In particular, compute the average degree of the network and compare it to the average degree of neighbor nodes. The two networks have very different scales of this behavior. Explain why we see this effect much more strongly in one network than the other.

The airport network is actually a directed network, but for the purposes of this activity, treat it as an undirected graph. To avoid repeated edges when you read it in use the following `ignore_duplicate_edges` flag:

```
        G = zen.edgelist.read(‘international_airports.edgelist’, ignore_duplicate_edges=True)
```