Lab3: Centrality Measures (Newman 7.1-7.7)

I have provided some stub code in `lab3_stub.ipynb` located online. At the top of the notebook change the name and NetID listed there. Please make sure you label your print statements, so we know what you are printing. In the end you are submitting one `ipynb` file and one `pdf` file, both named with your NetID: `lab3_abc123456`.

In this lab you will use a co-appearance network of characters from The *Lord of the Rings*. Although it was made into a series of movies, the story was first a book by J. R. R. Tolkien. The nodes of the network are the characters in the book and and edges between nodes carry the weight equivalent to the number of chapters in which both characters appear together. Initially, the network is loaded in the stub code as an unweighted network, but your analysis should be done to be compatible with the weighted version as well. In the end you will present the results of both the weighted and unweighted analysis.

The Stub Code

Throughout this lab you will have a vector (in Python it is a NumPy array, let's call it `v`) that represents the centrality of all the nodes, so it will be of length `n`, where `n` is the number of nodes in the network. I've prewritten some code that will help to easily print out the top 5 characters in terms of the centrality vector v. You'll see this already at the top of the stub code. This defines a function called `print_top_5(G,v,num)` which takes in the networkx Graph object `G` and the centrality vector `v`. It creates a new list of tuples, with each tuple having the index of the node and the centrality value of the node. It then sorts this based on the centrality score. Then it prints the top `num` (default of 5) node names according to their centralities.

Here, I've used the term "node index", which doesn't actually exist explicitly in the NetworkX Graph object. So what I mean by node index is the index that node appears in the list of nodes provided by `G.nodes()`. The function `G.nodes()` provides a list of nodes in a given order. As long as we don't add or remove nodes, the ordering of this list stays constant (it may stay consistent after node/edge addition, but all bets are off). This ordering is also what determines the ordering of the adjacency matrix returned by `nx.adjacency_matrix(G)`. I have included two helper functions along these lines. The stub code function `node_index(G,n)` returns what we mean here as node index of a node `n` in the graph `G`.

The next helper function `index_of_max(v)` returns the index of the maximum value in the list/array `v`. Technically it returns the first index that has this value (this will only occur if the maximum value is repeated). The `numpy.where` function returns an array where the inner condition is true, so the `[0]` at the end extracts the first element of the array. Here we are testing for elements of `v` that are equal to the maximum value in `v`. It is possible that two (or more) nodes have exactly the same centrality, so the `where(...)` function might return a

vector with two (or more) entries. However, if two nodes have the same centrality, then we don't care which one is returned, so taking the first is still a fine solution.

Section 7.1: Degree Centrality

Compute the degree centrality of the nodes in *The Lord of the Rings* network. Although the network is currently unweighted, do this assuming that the network is weighted, so degree centrality doesn't just count the number of neighbors, but returns the sum of the weights of the edges to all the neighbors. You could do this by looping through the network itself, or by looping over the adjacency matrix. Use the function we defined above to print out the top 5 characters with highest centrality in terms of degree. These characters are simply the characters with the most (and in the weighted case, strongest) connections to other characters.

Section 7.2: Eigenvector Centrality

Calculate the eigenvector centrality of the nodes in *The Lord of the Rings* network. Do this using the built-in NetworkX `eigenvector_centrality(...)` function as well as computing it directly from the adjacency matrix (using linear algebra). Print the top characters using the `print_top_5(...)` function, labeling the output of both methods.

You will find the `numpy.linalg.eig` function useful, which calculates both the eigenvalues and eigenvectors of a matrix (in our import statement we have renamed `numpy.linalg` as `la`, so calling this would just be written as `la.eig(...)`). Note that these values will come back as complex numbers, since eigenvalues and eigenvectors are in general complex. However, recall, that for an undirected network, we have a symmetric adjacency matrix, which has real eigenvalues and eigenvectors (recall your linear algebra!). Even though the imaginary part is zero, the function still returns a complex value, e.g., `4.52 + 0.j`. So before you use them, take their absolute value: `numpy.abs(x)` or real value `numpy.real(x)`, to get only the real part. I've gotten you started by calling `la.eig` and identifying the index of the largest eigenvalue.

Although eigenvector centrality (and some of the other centrality measures) uses eigenvalues and eigenvectors, typically for large networks -- large matrices -- it is not computationally efficient to calculate and then use the eigenvalues and eigenvectors. Instead, most centrality methods will use an iterative scheme to calculate an approximation of the eigenvalues and eigenvectors. The same is true for the matrix inversions we will see in the next section and the book discusses the iterative scheme a bit more in depth. If you have extra time, you can try to implement this iterative scheme for any of these methods!

Let's spend some time to build our intuition about eigenvector centrality. Here, we are going to confirm that the values of eigenvector centralities are selected so that the process of passing centralities around the graph is in "steady-state". If it was a perfect steady-state, then the centralities of a node's neighbors would add up to the centrality of that node. We observed in class that we instead get that the a node's centrality is equal to the sum of its neighbors' centralities, normalized by the largest eigenvalue. Confirm that this is the case for a node index

of your choosing by printing these two quantities: the centrality of that character and the (weighted) sum of the centralities of that character's neighbors (normalized by the largest eigenvalue).

Next we will observe the convergence of the terms of eigenvector centrality using the iterative scheme. We will see that 10 steps is enough to observe this behavior for this network. The code is fully functional and does not require you to modify anything to work (provided you haven't changed the names of `k`, `v`, and `k1_idx`). Notice I am taking `num_steps` steps starting from an initial guess of centrality that is rather arbitrary (`x` is all zeros except I make one entry nonzero). At each step `i`, I am normalizing the centrality vector `x`, then determining the coefficients of the expansion: $x = c_1v_1 + c_2v_2 + \ldots + c_nv_n$ (which I call `cs` in the code), then propagating forward along the edges by one step. Then I'm plotting each of the c_i over the steps of this iteration. What I want you to notice is that x is converging towards the direction of v_1, the eigenvector corresponding to the largest eigenvalue. Turn in this plot. In your own words, make a short statement that describes what is going.

Section 7.3: Katz Centrality

We discussed several issues with eigenvector centrality, especially in directed networks (namely that nodes without any incoming edges are worth nothing). Although this may be acceptable in some cases, we now look at a model that gives every node a base level of centrality (given by β). Calculate Katz centrality for the nodes in *The Lord of the Rings* network using linear algebra, or if you are curious, you can use the same type of iterative scheme used for eigenvector centrality. Try a few values for α and print out the top 5 central characters in each case.

Along the way, you'll need to do a few matrix manipulations. These NumPy functions may be useful:

- `numpy.ones((n,1))` creates an n-dimensional vector of all ones
- `numpy.eye(n)` creates an `n` x `n` identity matrix
- `la.inv(A)` calculates the inverse of a matrix
- `numpy.dot(A,B)` use this to do matrix multiplication -- the `*` notation does not do matrix multiplication

Section 7.4: PageRank Centrality

PageRank introduces another perspective on centrality in which the centrality of a node is not simply passed on to all its neighbors. Instead its centrality is divided by its degree before it is passed on. Calculate PageRank centrality (by linear algebra) for *The Lord of the Rings* network and print the top 5 characters with highest centrality.

In class we defined PageRank centrality as $x = (I - \alpha AD^{-1})^{-1}.\mathbf{1}$ and indicated that $0 \leq \alpha < 1$. For a connected undirected graph, show that the vector $v_1 = (k_1, k_2, \ldots, k_n)$, where k_i is the degree of node i, is an eigenvector of AD^{-1} and using this show why α is upper bounded by 1.

Section 7.5: Hubs & Authorities

Since hubs and authorities only occur for directed networks, we will not calculate them for *The Lord of the Rings* network. In our class discussion we identified that the ranking of authorities of a directed graph G coincided with eigenvector centrality of the cocitation network of G and that the ranking of hubs coincided with eigenvector centrality of the bibliographic coupling network of G. In doing this we observed that they shared an eigenvalue of $(\alpha\beta)^{-1}$. Now derive the relationship between the corresponding eigenvectors for hubs and authorities – i.e., express the hub eigenvector in terms of the authorities eigenvector.

Section 7.6: Closeness Centrality

Closeness centrality is yet another centrality measure for networks, but we will skip this in our analysis of *The Lord of the Rings* network.

Section 7.7: Betweenness Centrality

Betweenness is our final centrality measure, however, there are many, many other measures that exist and are used in the field of network science. In the original definition, betweenness centrality reports how many of the shortest paths between any two nodes in the network go through a particular node. In the context of *The Lord of the Rings* network, high betweenness of a particular person indicates that the shortest way to connect from most characters to most other characters includes this person. Betweenness is an existing algorithm in NetworkX: `nx.betweenness_centrality(...)`. Using this function print the top 5 characters with highest betweenness centrality.

Weighted Analysis

The original Lord of the Rings network is actually a weighted network. Now load the network as a weighted network by changing the boolean flag `unweighted=True` to `unweighted=False`.

Changing the definition of `G` at the top of your file is (almost) all you need to do – although do think through and make sure you have implemented all your centrality calculations compatible with weighted edges. In particular, chances are that your computation of eigenvector centrality being a steady-state no longer works. If this is the case, update this calculate to incorporate weights. You will notice that it is backwards compatible (works still for the unweighted version). Re-run your analysis and comment on the changes that you see due to including the weights in the analysis. In particular, offer explanations for why some of the rankings change and some do not (recall, we're most interested in the order of the characters, not about the particular numerical values). Here it would make sense to rely on your knowledge of the movies/books. Feel free to talk with friends about this if you're not familiar with the story (shame on you!).

A Sequel

Actually, it is a prequel. Tolkien also wrote *The Hobbit*, which takes place before the events in *The Lord of the Rings*. There are a number of characters in common between the books, so it is interesting to see what happens to our centrality measures when we include *The Hobbit* information into the network. In this case, I have done the exact same process as before to create the network, however, now with both The Hobbit and *The Lord of the Rings* books included. You can do this analysis for just the weighted version of this network. Observe again the changes from including the new characters. Do some of the changes make sense based on what you know about The Hobbit (the movie - or Wikipedia - will give you a good enough guess for this). The line to import this network is already in your stub code, just commented out at the top.

Submission

In this lab you will run the same code several different ways, so in this case, please submit a PDF of the results for all these runs (LotR unweighted, LotR weighted, LotR+Hobbit weighted). It would make sense to present this as a combined table rather than three separate long print outs. This would help highlight for each centrality how it changes from unweighted to weighted to adding *The Hobbit*. When you turn in your `.ipynb` file (and the printout, please leave it in the right form to run this last analysis (the sequel with Hobbit and Lord of the Rings, weighted analysis).