PHYS 7332 — Network Science Data II
Assignment 1

Giuliano Porciúncula Guedes

September 29, [Fall] 2025

Question 1 (15 points) More databases.
There are lots of databases out there that are very different from SQL. Although SQL is probably what you'll see most often out in the real world, it isn't optimal for all use cases. Let's read up on a few other kinds of databases to better understand what's out there. For each of the databases listed below, please explain how it structures its data (e.g. key/value, tabular, etc.) and tell us what kinds of data/applications it's most useful for. Additionally, give us an example of a bad use case for that specific database and explain why it is poorly suited for that particular use case.

(a) Redis

(b) MongoDB

(c) Cassandra

(d) Neo4j

Solution

(a) Redis is an in-memory key-value data structure store that supports a wide range of data types, including Strings, Hashes, Lists, Sets, Sorted Sets, Vector sets, Streams, Bitmaps, Bitfields, Geospatials, JSONs, Probabilistic data structures, and Time Series. Because Redis stores its data entirely in memory, it is extremely fast and optimized for high-performance tasks where low latency is critical. This makes it ideal for use cases such as caching, real-time analytics, and leaderboards. However, Redis is not well-suited for very large data sets or applications requiring complex queries or persistence of massive amounts of information. Since all data must fit within available RAM, scaling beyond memory limits can become costly and impractical.

Source: https://redis.io/docs/latest/develop/

(b) MongoDB is a document-oriented database that stores data in BSON (Binary JSON) format. Each document organizes information into field–value pairs and supports both nested structures and references between documents. This flexible schema design allows MongoDB to naturally represent hierarchical or heterogeneous data, making it useful when network entities have complex attributes or multi-level relationships. However, MongoDB is not well-suited for intensive network traversal or path-tracing operations. Since it lacks native graph traversal capabilities, multi-hop queries across highly connected data require multiple lookups or application-level logic, which can be inefficient for large or dense networks.

Source: https://www.mongodb.com/docs/

(c) Apache Cassandra is a distributed, wide-column database designed for handling massive amounts of data. It stores data in tables with rows and dynamic columns grouped into column families. Each row can have a different set of columns, which allows for flexible and sparse data modeling. Cassandra's greatest strength lies in its scalability and fault tolerance. It can efficiently handle high write and read throughputs. This makes it ideal for applications requiring high availability and continuous uptime, such as monitoring large communication networks, tracking dynamic graph metrics, or storing time-series data from distributed systems. However, Cassandra is not well-suited for complex joins, ad-hoc queries, or multi-hop graph traversals. Its query model is designed for predictable access patterns, so irregular queries can be inefficient or difficult to express.

Source: https://cassandra.apache.org/

(d) Neo4j is a native graph database designed to store and manage data as nodes, relationships, and properties. Instead of tables or documents, Neo4j structures data as a graph, where nodes represent entities and relationships represent direct connections between them. Each node or relationship can hold properties, allowing rich, attribute-based modeling. Its key strength is efficient graph traversal. Neo4j uses index-free adjacency, meaning each node directly references its connected nodes, enabling constant-time relationship lookups regardless of dataset size. This makes it ideal for applications that rely on exploring complex connections, such as social network analysis, pathfinding, recommendation systems, and link prediction in network science. However, Neo4j is less suitable for workloads that involve massive, unconnected data sets or require high write throughput across distributed clusters. Its focus on relationship-centric queries means it is not optimized for bulk analytical operations, time-series storage, or fast simple key-value lookups.

Source: https://neo4j.com/docs/

Question 2 (40 points). Exploring community detection.

(a) Download a network dataset from https://networkrepository.com/ with between $1000 < N < 5000$ nodes (select a different dataset from the one you used in Assignment 1). Select a modularity maximization algorithm to detect communities in this graph.

i Visualize your network, coloring the nodes by community membership.

ii Report the modularity of the partition your algorithm found.

iii Report the number of communities detected.

iv Store your partition as a dictionary, in the form of node id: community id.

(b) Using the same network as in (a), use graph-tool to create a partition of your network (i.e., run community detection). Visualize your network again, coloring the nodes by community membership; use the same network visualization layout as the in (a). Store your partition as a dictionary, in the form of node id: community id.

(c) Create a degree-preserving randomization of your network, and run the two community detection algorithms from above on your randomized graph. Store each partition as a dictionary, in the form of node id: community id.

(d) You should now have four different dictionaries, corresponding to the partitions created in (a-c). Create a figure with four subplots: In each, plot the community size distribution of each partition. Your figure should be publication-ready quality, with well-labeled, consistent axes, log-scaled where relevant, etc.

(e) For the different partitions computed above, discuss how you would approach comparing these partitions quantitatively. For example, imagine a function $f(b_1, b_2)$ that takes two partitions (b_1 and b_2) as input and returns a number corresponding to how similar/dissimilar the partitions are. What properties should this function f have? For this question, base your answer on a technique used in the literature, after a brief search about methods for quantifying differences between graph partitions.

Solution

Chosen network: https://networkrepository.com/ia-email-univ.php
It has 1133 nodes and 5451 edges.

(a) We will used the modularity maximization as described in Clauset, A., Newman, M. E., & Moore, C. "Finding community structure in very large networks." Physical Review E 70(6), 2004.

Figure 1 portraits a visualization of our network. In the figure itself we see the modularity of 0.5034 with a total of 11 communities. We can see that each community has a distinct color. The biggest community is the dark blue with 319 nodes, followed by the orange with 238 nodes, all the way down to the light blue with only 3 nodes.

(b) For this item, we utilize the algorithm following Tiago P. Peixoto, "Efficient Monte Carlo and greedy heuristic for the inference of stochastic block models", Phys. Rev. E 89, 012804 (2014), DOI: 10.1103/PhysRevE.89.012804 [sci-hub, @tor], arXiv: 1310.4378.

Figure 2 shows our communities with the same node positions as Figure 1. Since they did not agree at all, we can group the communities together into the circumference of the circle. Figure 3 shows exactly that. We can see that as we go counterclock-wise, our communities diminish in size, from the biggest with 156 nodes represented by black all the way down to the smallest with 8 nodes represented by gray.

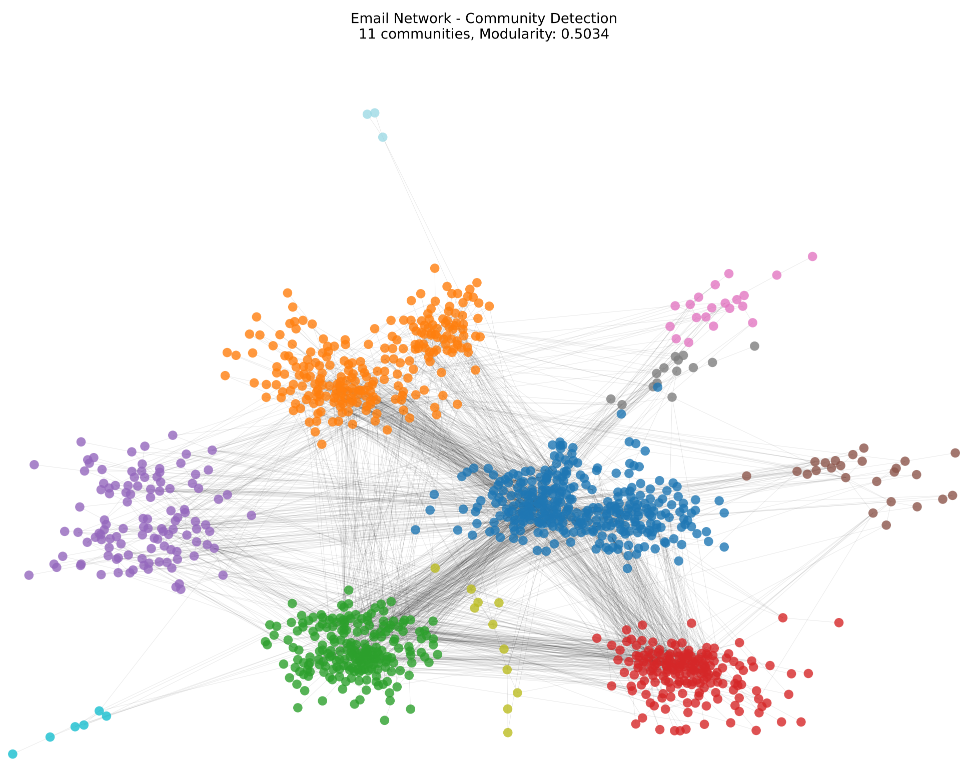

Figure 1

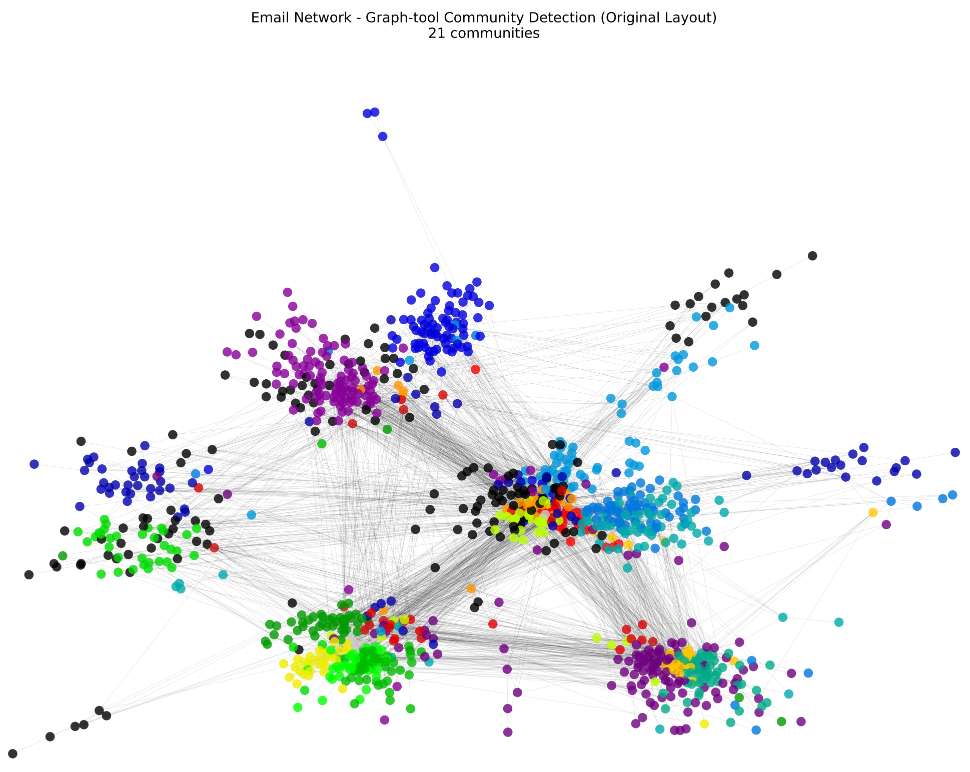

Figure 2

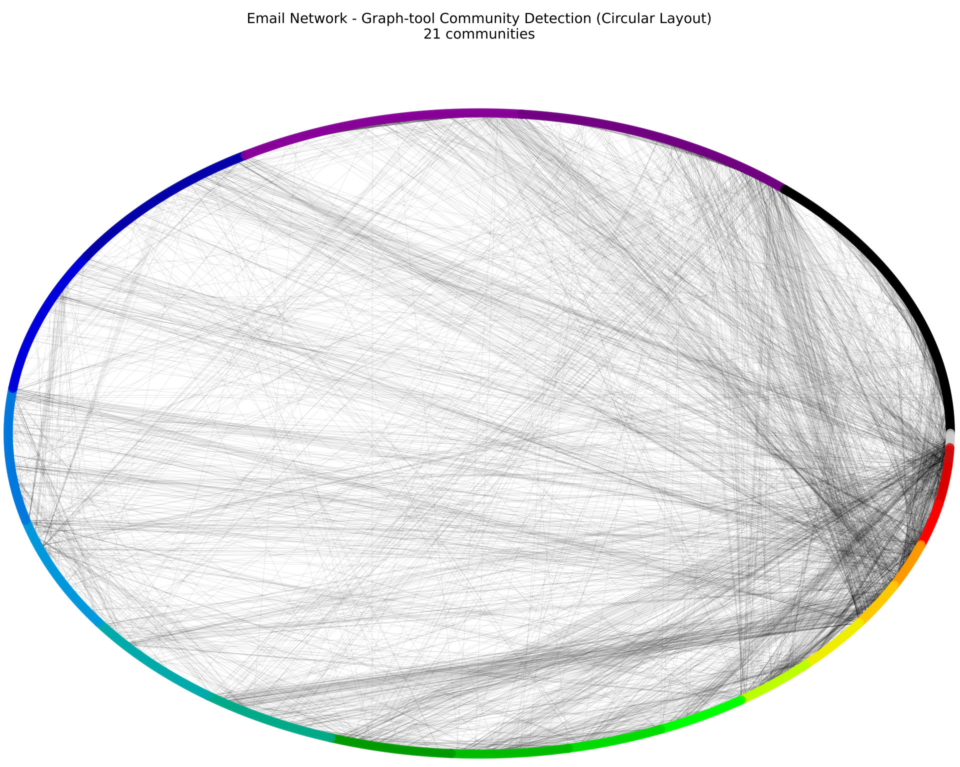

Figure 3

(c) Method (a) for the randomized network yielded 16 communities with a modularity of 0.2812. Method (b) for the randomized network yielded only one community. Method (b) utilizes statistichal inference by the stochastic blockmodel, while method (a) uses the maximization of the modularity. Since method (b) only detected one community, this means that the other communities found by method (a) are not statistichal significant. We can back this up also by looking at the modularity of the network, which is quite low.

(d) Figure 4 shows 4 plots of Community Size (nodes) versus community rank for both original network and randomized network and both modularity maximization and blockmodel. For both modularity maximization methods we observe a jagged curve, where the blockmodel for the original network is more flush. The randomized blockmodel shows us the presence of only one community, indicating that the randomization destroys the community. The modularity maximization fails to capture that.

(e) The function f should be label invariant, able to handle different number of communities for both $\boldsymbol{U}$ and $\boldsymbol{V}$, bounded $[0, 1]$ and interpretable (extremes are identical / totally different), corrected for chance, symmetric $f(\boldsymbol{U}, \boldsymbol{V}) = f(\boldsymbol{V}, \boldsymbol{U})$.

Let Ω be our set of nodes. A set of communities $\boldsymbol{U}$ partitions Ω into $\{U_i\}_{N_{C,U}}$ where $\cup_{i=1}^{N_{C,U}} U_i = \Omega$ and $U_i \cap U_j = \emptyset$ for $i \neq j$. Let $\boldsymbol{V}$ also be a set of cummunities. The entropy of a set of communities is given by

$$S(\boldsymbol{U}) = -\sum_{u \in \boldsymbol{U}} p(u) \log[p(u)] \tag{1}$$

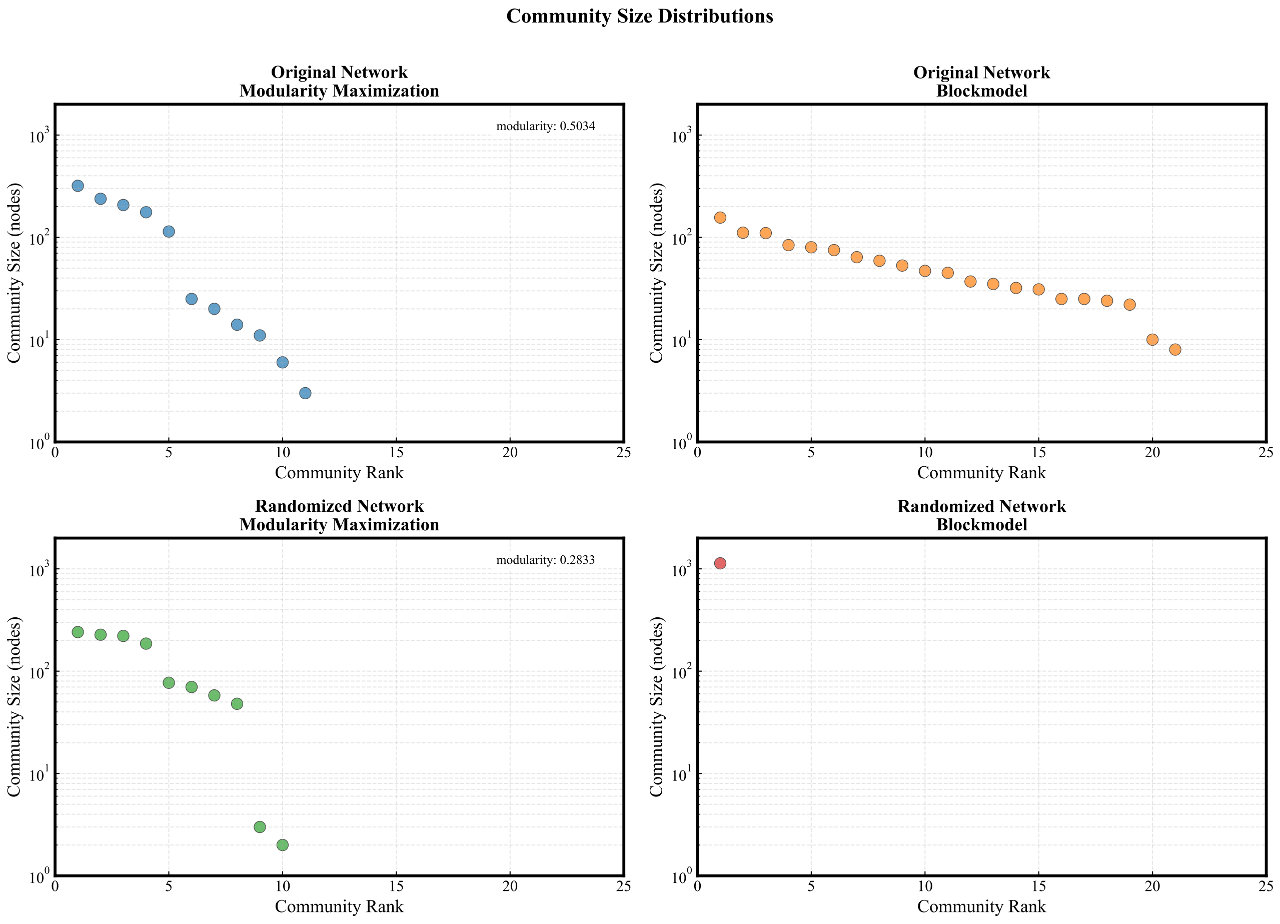

Figure 4

The joint entropy of two sets of communities is given by

$$S(\boldsymbol{U},\boldsymbol{V}) = -\sum_{u\in\boldsymbol{U}}\sum_{v\in\boldsymbol{V}} p(u,v)\log[p(u,v)] \tag{2}$$

The conditional entropy between set $\boldsymbol{U}$ know $\boldsymbol{V}$ is given by The joint entropy of two sets of communities is given by

$$S(\boldsymbol{U}|\boldsymbol{V}) = -\sum_{u\in\boldsymbol{U}}\sum_{v\in\boldsymbol{V}} p(u,v)\log\left[\frac{p(u,v)}{p(v)}\right] \tag{3}$$

And the mutual information $I(\boldsymbol{U},\boldsymbol{V}) = I(\boldsymbol{V},\boldsymbol{U}) = S(\boldsymbol{U}) - S(\boldsymbol{U}|\boldsymbol{V}) = S(\boldsymbol{V}) - S(\boldsymbol{V}|\boldsymbol{U})$ between $\boldsymbol{U}$ and $\boldsymbol{V}$ is given by

$$I(\boldsymbol{U},\boldsymbol{V}) = \sum_{u\in\boldsymbol{U}}\sum_{v\in\boldsymbol{V}} p(u,v)\log\left[\frac{p(u,v)}{p(u)p(v)}\right] \tag{4}$$

It ranges from 0 to $\min[S(\boldsymbol{U}),S(\boldsymbol{V})]$, thus the normalized mutual information can be written by

$$I_N(\boldsymbol{U},\boldsymbol{V}) = \frac{I(\boldsymbol{U},\boldsymbol{V})}{\min[S(\boldsymbol{U}),S(\boldsymbol{V})]} \tag{5}$$

The closer together $\boldsymbol{U}$ and $\boldsymbol{V}$ are, the closer $I_N(\boldsymbol{U},\boldsymbol{V})$ is to 0.
$I_N(\boldsymbol{U},\boldsymbol{V})$ has a problem that it is not corrected by chance. To take that into account, we can

calculate the adjusted mutual information $I_{A,N}(\boldsymbol{U},\boldsymbol{V})$

$$I_{A,N}(\boldsymbol{U},\boldsymbol{V}) = \frac{I(\boldsymbol{U},\boldsymbol{V}) - E[I(\boldsymbol{U},\boldsymbol{V})]}{\min[S(\boldsymbol{U}),S(\boldsymbol{V})] - E[I(\boldsymbol{U},\boldsymbol{V})]} \tag{6}$$

Where $E[I(\boldsymbol{U},\boldsymbol{V})]$ is the expected value for $I(\boldsymbol{U},\boldsymbol{V}]$ assuming it was by chance.

Source for (d): https://jmlr.csail.mit.edu/papers/volume11/vinh10a/vinh10a.pdf

Question 3 (30 points). Benchmarks for community detection.
There are many ways to "benchmark" community detection algorithms; one famous approach is known as the Lancichinetti-Fortunato-Radicchi (LFR) benchmark1.

(a) Read the LFR Benchmark paper (Lancichinetti, A., Fortunato, S., & Radicchi, F. (2008). Benchmark graphs for testing community detection algorithms. Physical Review E, 78(4), 046110. doi: 10.1103/PhysRevE.78.046110.). Describe the core motivation behind the paper, its main contributions, and the basics of the benchmark (i.e., what are the inputs to the benchmark, what is the output, etc.).

(b) On page 2 of the article—in Section II: The Benchmark —the authors describe the five steps involved in sampling benchmark graphs.

i Step 1 is reminiscent of the procedure for creating powerlaw configuration model networks. Write a function that outputs a powerlaw degree sequence, $\{k_1, k_2, \cdots, k_n\}$, which can be used as input in a configuration model. Your function should take N (network size), γ (powerlaw degree exponent), and $\langle k \rangle$(average degree) as inputs.

ii Step 2 introduces a mixing parameter, μ, that is used to assign a fraction of a node's total degree, k_i, to be internal community edges (i.e., connecting to nodes with the same community label as node i), such that $k_i^{\text{in}} = k_i(1-\mu)$; with probability μ, therefore, the remaining edges of i are connected to nodes external to the community, $k_i^{\text{out}} = k_i\mu = k_i - k_i^{\text{in}}$. Write a function that implements this step.

iii Step 3 resembles Step 1, but instead of defining powerlaw degree values, it defines a powerlaw community size distribution. What additional considerations does the paper say are needed in order to implement this step? No code is required to answer this.

iv (Optional): Finish writing the rest of the function, putting Steps 1-5 together. Note: This is a challenging function to implement and, as such, is optional. If you don't have time to implement this, feel free to explore existing (proper) implementations of this algorithm, either online or with AI assistance.

(c) networkx has a built-in function, nx.LFR_benchmark_graph(), that can generate these benchmark graphs. Many users of this algorithm report its inconsistency in convergence. Explore this function's usage for yourself: Are there certain parameter combinations that make the algorithm converge more/less successfully? Based on your reading of the original LFR paper, and your experiments above, briefly describe why you think this function has difficulty converging.

Solution

(a) The paper "Benchmark graphs for testing community detection algorithms" by Lancichinetti, Fortunato, and Radicchi (2008) introduces the LFR benchmark, a more realistic framework for evaluating community detection methods in complex networks. The authors were motivated by the limitations of earlier benchmarks, such as the Girvan–Newman model, which assumed uniform community sizes and homogeneous degree distributions, features rarely found in real networks. To address this, they designed a benchmark that reproduces key properties of real-world systems, including heterogeneous (power-law) degree distributions and power-law community size distributions, along with a mixing parameter μ that controls how strongly communities are defined (i.e., the fraction of a node's links connecting outside its community).

The LFR benchmark takes as inputs parameters like the number of nodes, average degree, degree and community-size exponents, and the mixing parameter. It outputs a synthetic network with a known "ground-truth" community structure, which allows researchers to test whether algorithms can correctly recover it. By varying μ and other parameters, the benchmark can systematically increase the difficulty of detection tasks. This approach provides a more rigorous and scalable way to compare community detection algorithms, revealing weaknesses that simpler benchmarks fail to expose.

(b) We know that the power law degree distribution follows $p(k) \sim k^{-\gamma}$. Let's say we have $k_{\text{min}} \leq k \leq k_{\text{max}}$. Thus by assuming $p(k) = Ck^{-\gamma}$, we can find C by normalization:

$$\int_{k_{\text{min}}}^{k_{\text{max}}} p(k) = 1 \qquad \Longrightarrow \qquad C^{-1} = \int_{k_{\text{min}}}^{k_{\text{max}}} k^{-\gamma} dk \tag{7}$$

Solving the integral we have

$$C = \frac{\gamma - 1}{k_{\text{min}}^{-(\gamma-1)} - k_{\text{max}}^{-(\gamma-1)}} \tag{8}$$

But we do not necessarily have k_{min} and k_{min}. So now, we can calculate $\langle k \rangle$:

$$\langle k \rangle = \int_{k_{\text{min}}}^{k_{\text{max}}} kp(k) = C \int_{k_{\text{min}}}^{k_{\text{max}}} k^{-(\gamma-1)} \tag{9}$$

$$\langle k \rangle = \frac{\gamma - 1}{\gamma - 2} \frac{k_{\text{min}}^{-(\gamma-2)} - k_{\text{max}}^{-(\gamma-2)}}{k_{\text{min}}^{-(\gamma-1)} - k_{\text{max}}^{-(\gamma-1)}} \tag{10}$$

Setting no upper bound $k_{\text{max}} \to \infty$ we have $\langle k \rangle$ and C to be

$$\langle k \rangle = \frac{\gamma - 1}{\gamma - 2} k_{\text{min}} \qquad \Longrightarrow \qquad k_{\text{min}} = \frac{\gamma - 2}{\gamma - 1} \langle k \rangle \tag{11}$$

$$C = (\gamma - 1) k_{\text{min}}^{(\gamma-1)} \tag{12}$$

The cummulative degree distribution up to k is given by

$$F(k) = \int_{k_{\min}}^{k} p(k')dk' = C\int_{k_{\min}}^{k} (k')^{-\gamma}dk' \tag{13}$$

$$F(k) = C\frac{k_{\min}^{-(\gamma-1)} - k^{-(\gamma-1)}}{\gamma-1} = C\frac{k_{\min}^{-(\gamma-1)}}{\gamma-1} - C\frac{1}{\gamma-1}k^{-(\gamma-1)} \tag{14}$$

And we can substitute C and $k_{\min}$ to write

$$F(k) = 1 - \left(\frac{\gamma-1}{\gamma-2}\frac{k}{\langle k\rangle}\right)^{-(\gamma-1)} \tag{15}$$

So with the cumulative degree distribution, we can generate random numbers respecting $p(k)$ from an uniform $[0,1]$ distributions. To do that, we simply set $F(k) = u$ and invert $k = F^{-1}(u)$. This method works because we are finding the value of k that respects $p_{k-1} < u < p_k$. Our result is

$$k = \frac{\gamma-2}{\gamma-1}\langle k\rangle (1-u)^{-1/(\gamma-1)} \tag{16}$$

Since k is discrete, our final expression we need to take the floor for our final expression

$$\boxed{k = \left\lfloor \frac{\gamma-2}{\gamma-1}\langle k\rangle (1-u)^{-1/(\gamma-1)} \right\rfloor} \tag{17}$$

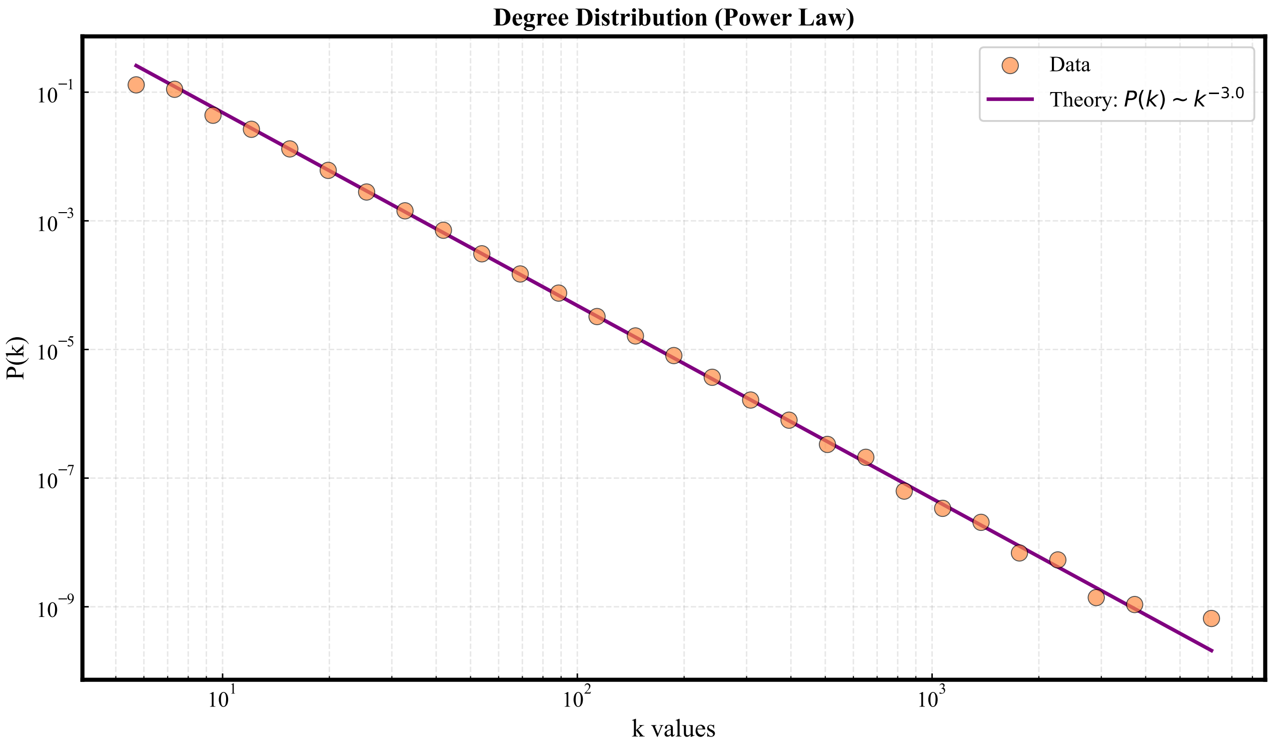

Figure 5

Figure 5 shows the distribution generated using our method. We generated 1000000 values of k with $\gamma = 3$ and $\langle k \rangle = 10$. Note the lack of hubs, just the straight power law.

Note that for community distribution we also have a power-law, but with exponent β instead of γ. Also instead of looking at $\langle k \rangle$ we would've been looking at k_{min}, making these changes we have the community size S generation equation as

$$\boxed{S = \left\lfloor S_{\text{min}} \left(1 - u\right)^{-1/(\beta-1)} \right\rfloor} \tag{18}$$

To generate the community sizes, we need that $S_{\text{max}} > k_{\text{max}}$ where k_{max} here denotes the highest value of k that we generated. We also need that $S_{\text{min}} > k_{\text{min}}$, thus we can set $S_{\text{min}} = k_{\text{min}} + 1$ where k_{min} here denotes the lowest value of k that we generated. Finally, we need that $\sum S_i = N$. Thus we will add communities if $\sum S_i < N$, remove communities if $\sum S_i > N$, and if $\sum S_i = N$ we check if $S_{\text{max}} > k_{\text{max}}$, if it's true we are finished.

Now we try to add nodes randomly to a community. If the degree of the node is smaller than the community size then it joints, it doesn't otherwise. If the community is already full, one other node is kicked out.

After that is done, we check each stub of each node against μ and add it to the counter of either the links inside or outside the community.

Unfortunately this is as far as I could go due to time constraints. The remaining part was to do the linking. Overall you'll have to connect the nodes inside their communities following their inside degrees, then connect the nodes outside, respecting their communities ID.

(c) The nx.LFR_benchmark_graph() function in NetworkX often fails to converge when its parameters make the underlying community structure mathematically difficult or impossible to realize. The algorithm tries to assign each node a number of internal and external links according to the mixing parameter, but this only works if communities are large enough to accommodate the internal degree requirements of high-degree nodes. When parameters such as a low μ (strong community structure), small community sizes, heavy-tailed degree distributions, or high average degrees are combined, some nodes end up needing more internal connections than a community can provide. This leads to the algorithm repeatedly attempting—and failing—to rewire edges to satisfy these constraints, ultimately hitting the maximum iteration limit. Conversely, when communities are larger, μ is moderate or high, and degree distributions are less skewed, the algorithm converges quickly. These convergence problems are not so much bugs as they are reflections of the inherent structural incompatibility between certain parameter choices and the theoretical requirements of the LFR model, which the original paper also notes as a limitation in generating highly heterogeneous or tightly constrained networks.

Question 4 (15 points). Python packages and software management.

Sometimes, you want to put a chunk (or several related chunks) of Python code into a package that you or others can install. This helps you and others use your code across several different projects and organize it in ways that make sense for you. You've developed several useful functions in this class so far; now it's time to turn them into a package for your future use.

You may find this tutorial (1) and its sequel (2), on Python packaging and making a package installable, to be useful.

(1)https://www.pyopensci.org/python-package-guide/tutorials/intro.html#python-packaging-101
(2)https://www.pyopensci.org/python-package-guide/tutorials/installable-code.html

(a) Distill some of the code we've written in this class so far into a basic toolkit that you'll reuse. Pick 6-10 functions (or objects) that you think would make up a useful network science toolkit based on code we've worked on in class. Note: You should not be planning on writing much new code for this question!

(b) Create a new GitHub repository with a README.md file that thoroughly explains the functions and/or objects you put in this package. Link it in your answer.

(c) Following the tutorials linked above, copy your code into a file(s) in your new repository, making sure it's organized in a way that works for you.

(d) Again, following the tutorials linked above, make your repository/package pip installable. If everything works correctly, we should be able to clone your repository, pip install it into an environment, and import your functions/objects.

Solution

Done. To install it, use:
pip install git+https://github.com/giuliano-porciuncula/netscidata2fall2025pipassignment.git

Jupyer notebook for all the questions available here.

(Optional) Question 5 (5 bonus points).
For every problem set this semester, there will be an opportunity to earn extra credit. Since this is a new course, we are eager to receive genuine feedback on the material, pace, instruction style, etc. of the course so far. Please provide any feedback about the lectures in recent weeks and/or this assignment.

Answer

I think the pacing of the classes are good, no complaints there. My problem really is with how long it takes to do some of the homeworks, specially in light of all the load. It's not that things are hard, it's just that it takes so much time that it's a bit overwhelming. For example, next homework is only available in a week, but this one is due today. For me, at least, would be nice to have this extra week to take a bit more time and appreciation on what the homework presents to us.