CSC 521: Monte Carlo Algorithms
Members: Kyle Kassen, Tate Teague
Group 11: Random Networks
Final Project Report | March 2022

Introduction + Domain Knowledge

Graph Networks are collections of nodes and edges interconnected to one another based on underlying relationships between each node. Often these edges contain attributes that give us additional information about the strength/direction/type of relationship. These networks can model many real-life scenarios such as communities of friends and colleagues to abstract ideas and ontological frameworks. It may not seem obvious what randomness has to do with what seems like a well-defined structure however as these networks grow and become more abstract it becomes necessary to run simulations on deviations of the graph to see where critical connections in the base network may lie. This was the use case we followed as we did not have the associated labels nor edge attributes from our edge list network data provided by Amazon. What we did believe we would be able to study were different inherent centrality and density metrics that would provide insight into the inherent structure.

Resources, Tooling, and Dataset

Resources: https://bit.ly/36VTNRP, https://bit.ly/3HAWZiw

The supplemental textbook and code tutorials provided us with concise theoretical explanations and simplified implementation examples for some of the operations that can be performed on networks.

Tooling: https://networkx.org/

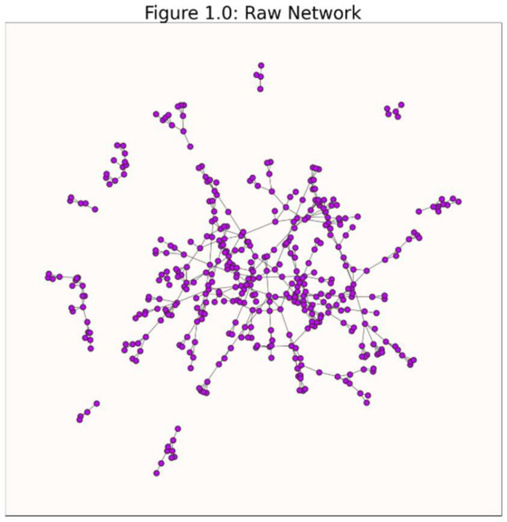
The NetworkX Python library provided many convenient methods which considerably reduced the complexity of working with network graph data structures and operations

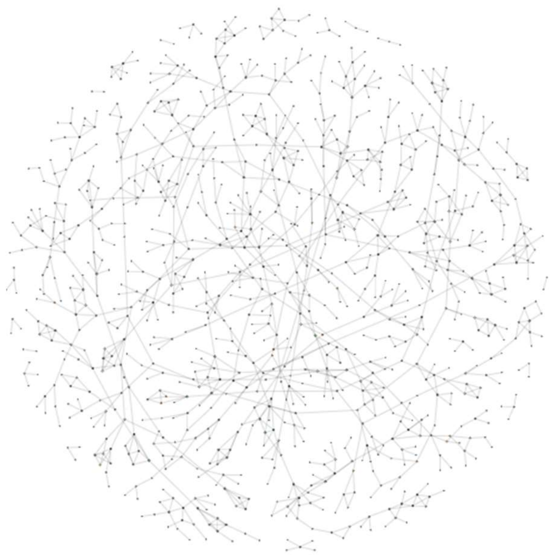
Dataset: https://bit.ly/3HKRZb0

Network was collected by crawling Amazon website. It is based on Customers Who Bought This Item Also Bought feature of the Amazon website. If a product A is frequently co-purchased with product B and vice-versa (i.e., product B is co-purchased with product A), then graph contains an edge between A and B.

- *On the left, you can see a visual of the first 500 entries of the Amazon dataset*
- *On the right, you can see a visual of the entire Amazon dataset*

Figure 1.0: Raw Network

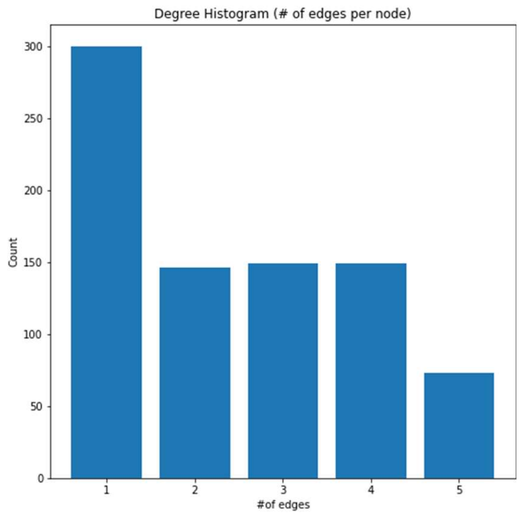




Base Network: Exploratory Data Analysis

To understand the structure of a network there are three common Centrality metrics used to understand edge and node distribution across the network. Those three metrics are as follows:

- **Betweenness Centrality:** Number of times a node lies on the shortest path between other nodes
- **Degree Centrality:** The number of edges held by each node
- **Closeness Centrality:** The score each node gets in relation to closeness (or how many connections) of it with respect to all other nodes in network



Aside from the base metrics there were other interesting metrics and distributions that gave additional insight. Not all of them were selected or used moving forward as they were somewhat meaningless in random simulations without knowing more about the label's attributes associated with the nodes and edges.

To the right we can see a histogram distribution of edges for all nodes across the network. It appears a majority of the nodes have only one edge and the others are 2, 3 or 4 edges to a node. Other measures such as graph density start to give better insight into the full picture. Network graph density is a ratio of the edges in the graph to the maximum possible number of edges it could have. For our raw Amazon network graph, we had a density of roughly 0.00547 or relatively low ratio, this makes since when looking at the distributions of degree across the nodes with a mean degree of ~2.

Some other base stats surrounding our graph are shown below to the right in figure 1.

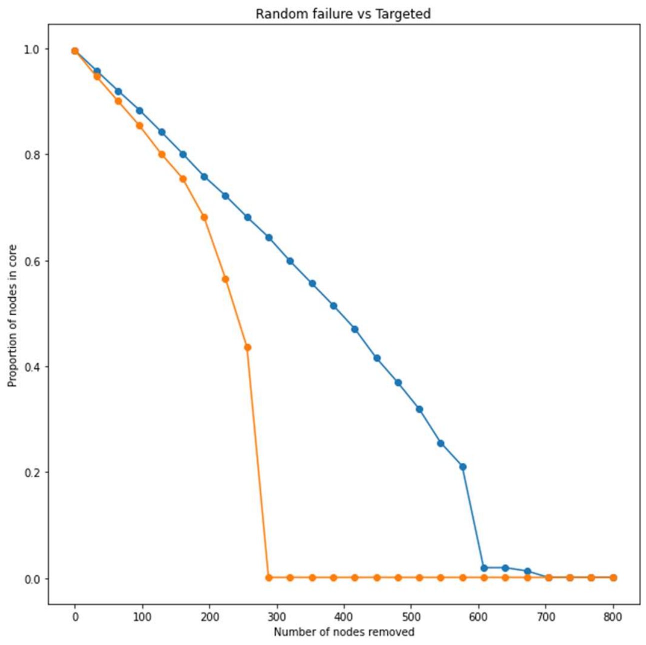
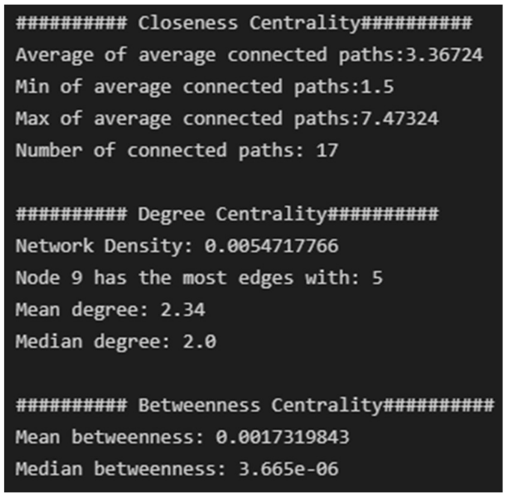


Figure 2



```
########## Closeness Centrality##########
Average of average connected paths:3.36724
Min of average connected paths:1.5
Max of average connected paths:7.47324
Number of connected paths: 17

########## Degree Centrality##########
Network Density: 0.0054717766
Node 9 has the most edges with: 5
Mean degree: 2.34
Median degree: 2.0

########## Betweenness Centrality##########
Mean betweenness: 0.0017319843
Median betweenness: 3.665e-06
```

Figure 1

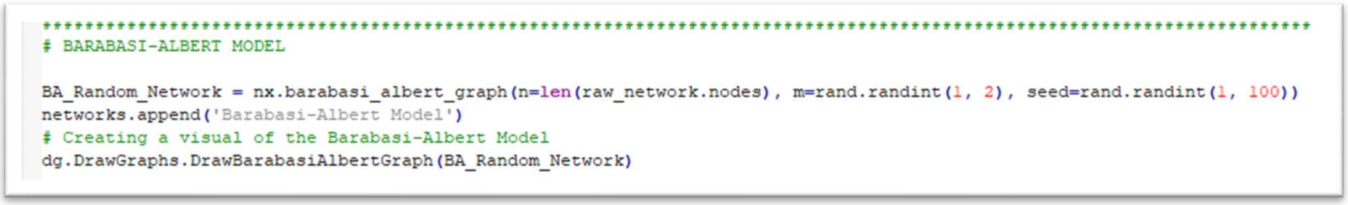
Shown above on the left we have what is referred to as a robustness test. The test aims to identify how interconnected and evenly distributed the network is. This is measured by comparing random failure to targeted failure. Targeted failure is the systematic removal of nodes that have the most connections (highest degree) then a measure of the percent of nodes remaining in core (largest connected bit). This is compared to random failure where nodes are randomly selected and removed measuring the same proportion of nodes remaining in core.

Random Network Models

Barabasi-Albert Model…

Formally, the Barabasi-Albert Model is an emergent graph of *n* nodes where each node has *m* edges that demonstrate preferential attachment with existing nodes of high degree. The idea was to simulate the emergence of a variation on the original network including a variation on categorical or thematical relationships that exist between nodes.

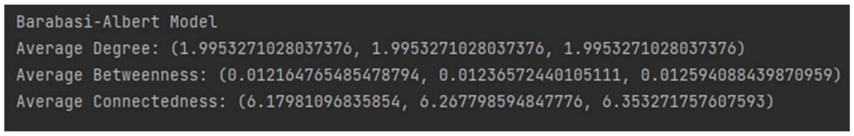
The implementation of the Barabasi-Albert Model is documented in the below code snippet. The method takes three arguments: 1) n=number of nodes (based on the Amazon dataset), 2) m=number of edges to attach from a new node to existing nodes (based on the degree of the Amazon dataset), and 3) seed=random number generation for adding the randomness facility.



```
###############################################################################################
# BARABASI-ALBERT MODEL

BA_Random_Network = nx.barabasi_albert_graph(n=len(raw_network.nodes), m=rand.randint(1, 2), seed=rand.randint(1, 100))
networks.append('Barabasi-Albert Model')
# Creating a visual of the Barabasi-Albert Model
dg.DrawGraphs.DrawBarabasiAlbertGraph(BA_Random_Network)
```

Output…

When compared to our Raw Network metrics, we notice that the Barabasi-Albert Model consistently records similar measures of degree. (Barabasi-Albert=~1.99 Vs. Raw Network=~2.33). We observe a doubling in measurements for Connectedness (Barabasi-Albert=~6.26 Vs. Raw Network=~3.36) as well as a large difference in Betweenness (Barabasi-Albert=~0.012 Vs. Raw Network=~0.0017).



```
Barabasi-Albert Model
Average Degree: (1.9953271028037376, 1.9953271028037376, 1.9953271028037376)
Average Betweenness: (0.012164765485478794, 0.01236572440105111, 0.012594088439870959)
Average Connectedness: (6.17981096835854, 6.267798594847776, 6.353271757607593)
```

Upon visual inspection of this version of the Barabasi-Albert Model, we observe a large cluster toward the center of the graph. We also note that several nodes display a one-to-many relationship with other nodes. While the visual is changing with each run of the simulation, these basic observations remain consistently present.

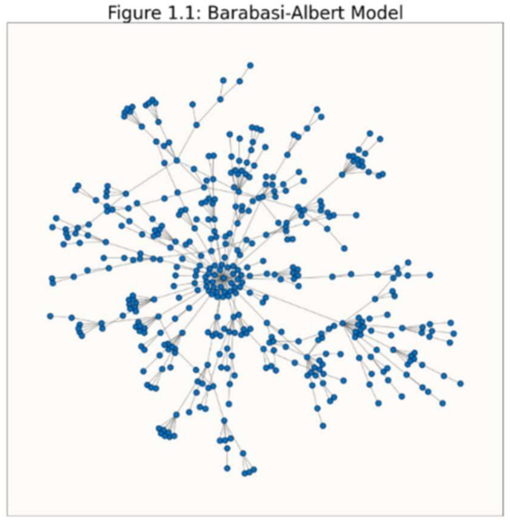
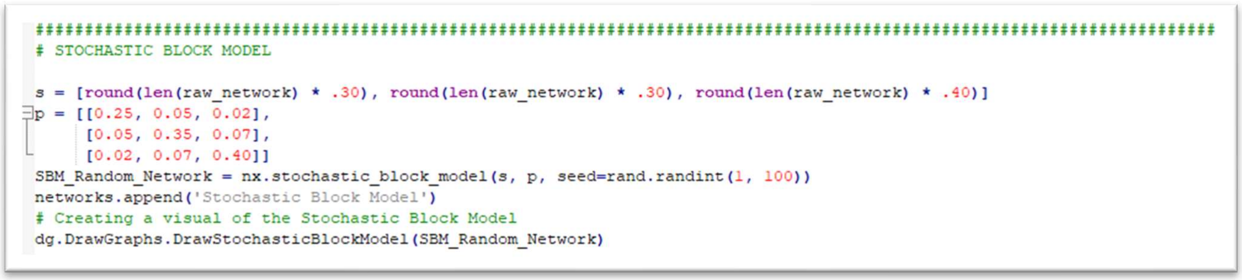


Figure 1.1: Barabasi-Albert Model

Stochastic Block Model…

The Stochastic Block Model is unique in that it provided dual functionality in the form of a generative random network model with clustering. Interestingly, the Stochastic Block Model is commonly used for detection of clustering/community structure in existing networks. This model partitions the nodes in blocks of predetermined sizes, and places edges between pairs of nodes independently with a predetermined probability that depends on the blocks.

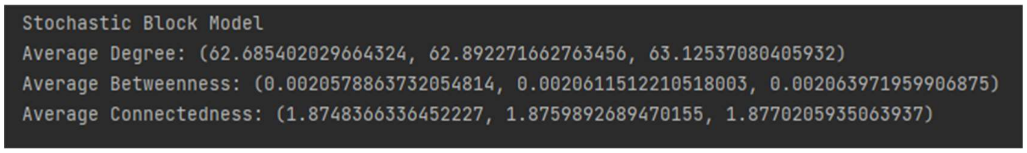
The implementation of the Stochastic Block Model is documented in the below code snippet. The method takes three arguments: 1) s=predefined blocks (partitions) equal to the number of nodes in the underlying Amazon dataset, 2) p=predefined probability matrix that must be square and symmetric, and 3) seed=random number generation for adding the randomness facility.



```
#########################################################################################################################
# STOCHASTIC BLOCK MODEL

s = [round(len(raw_network) * .30), round(len(raw_network) * .30), round(len(raw_network) * .40)]
p = [[0.25, 0.05, 0.02],
     [0.05, 0.35, 0.07],
     [0.02, 0.07, 0.40]]
SBM_Random_Network = nx.stochastic_block_model(s, p, seed=rand.randint(1, 100))
networks.append('Stochastic Block Model')
# Creating a visual of the Stochastic Block Model
dg.DrawGraphs.DrawStochasticBlockModel(SBM_Random_Network)
```

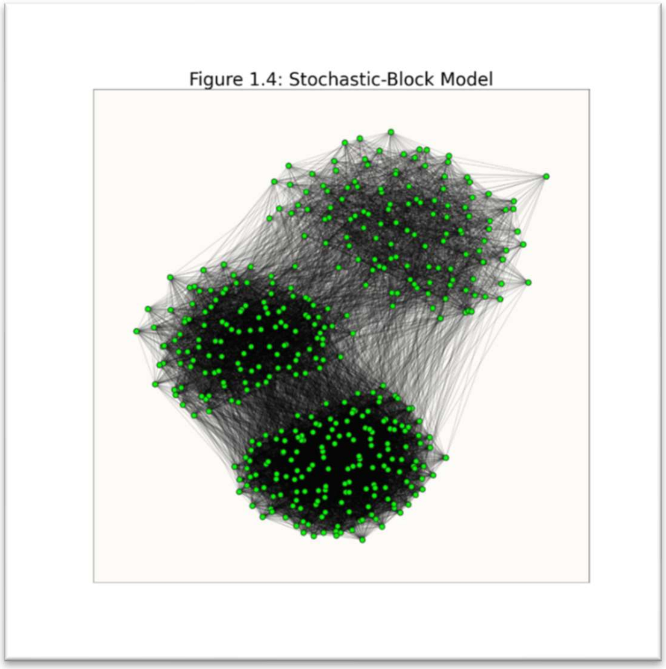
Output…

When compared to our Raw Network metrics, we notice that the Stochastic Block Model is the only model that records a Betweenness measurement similar to the Raw Network. (Stochastic Block=~0.0020 Vs. Raw Network=~0.0017). We observe an extremely high measure of degree (Stochastic Block=~62.89 Vs. Raw Network=~2.33). Connectedness is lower than that of our Raw Network (Stochastic Block=~1.87 Vs. Raw Network=~3.36).

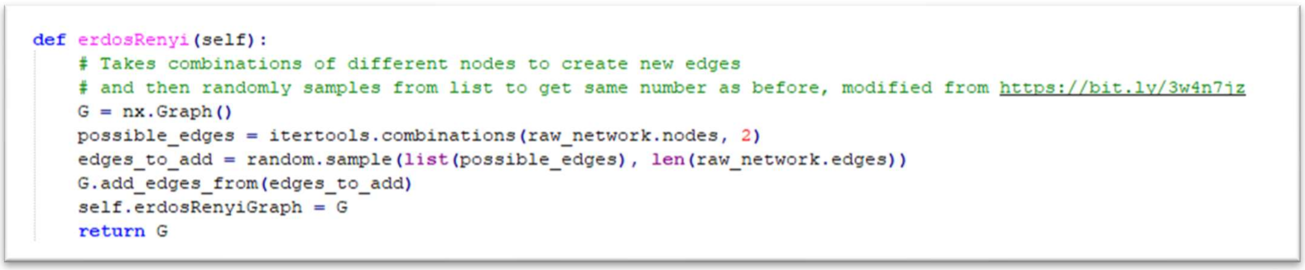


```
Stochastic Block Model
Average Degree: (62.685402029664324, 62.892271662763456, 63.12537080405932)
Average Betweenness: (0.0020578863732054814, 0.0020611512210518003, 0.002063971959906875)
Average Connectedness: (1.8748366336452227, 1.8759892689470155, 1.8770205935063937)
```

The three clusters are evident when considering our visual representation of this Random Network Model. We also observe that the number of edges produced by the model does not reflect the underlying dataset. We note that lower two of the three clusters are more dense than upper right cluster with an outlier node in the top right corner of the visual.

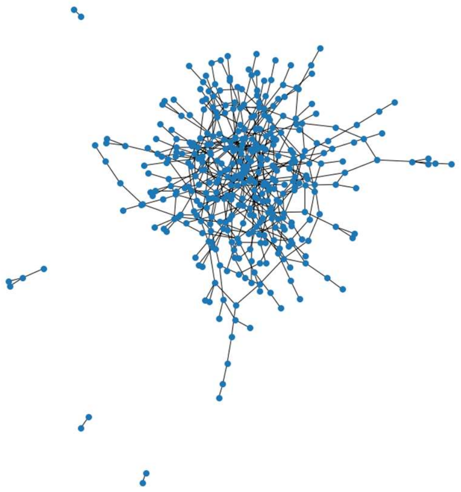


Erdos-Renyi Model…

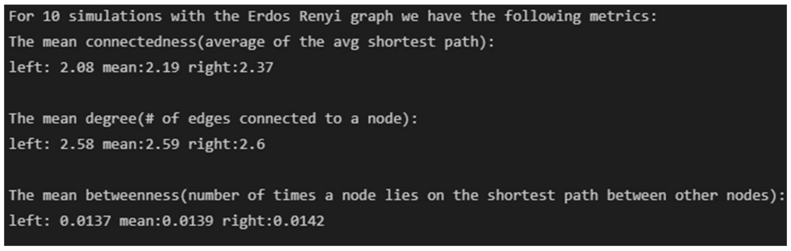


```
def erdosRenyi(self):
    # Takes combinations of different nodes to create new edges
    # and then randomly samples from list to get same number as before, modified from https://bit.ly/3w4n7jz
    G = nx.Graph()
    possible_edges = itertools.combinations(raw_network.nodes, 2)
    edges_to_add = random.sample(list(possible_edges), len(raw_network.edges))
    G.add_edges_from(edges_to_add)
    self.erdosRenyiGraph = G
    return G
```

The Erdos-Renyi Model had two slightly different derivations that we explored. The first being a probabilistic based algorithm that given a number of nodes and edge list chose whether or not to connect them based on an underlying probability. The other, method I ended up implementing was, was based of random combinations of edges and nodes from an existing edgelist. The algorithm could take the number of nodes and edges in the existing base network and generate new random combinations. This proved useful in validating any underlying groupings and structure qualities. Simulating these combinations repeatedly helped us arrive at averages we felt accurately defined inherent qualities of the graph.



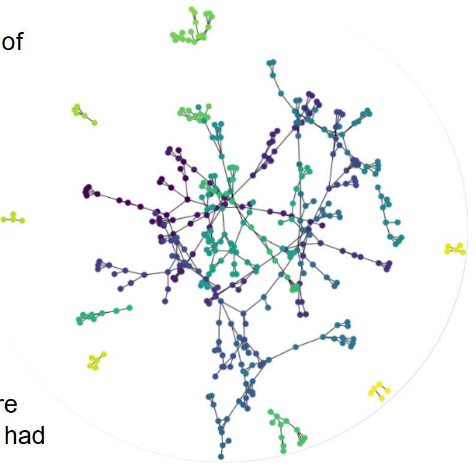
We can see that while the number of edges remained the same the interconnectedness of the core seems much messier. This overlapping however did little to change the core centrality metrics. We can see similar values for mean degree(to be expected) and average closeness. We do however see increased betweenness up to .0139 from .0017 in our base graph.



```
For 10 simulations with the Erdos Renyi graph we have the following metrics:
The mean connectedness(average of the avg shortest path):
left: 2.08 mean:2.19 right:2.37

The mean degree(# of edges connected to a node):
left: 2.58 mean:2.59 right:2.6

The mean betweenness(number of times a node lies on the shortest path between other nodes):
left: 0.0137 mean:0.0139 right:0.0142
```

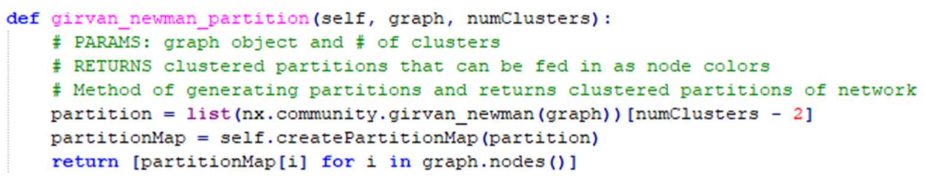
Clustering Algorithms

Girvan-Newman Clustering Algorithm…

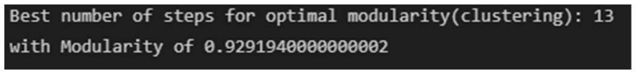


To get an idea of what our base clusters would look like we implemented both of clustering methods on the defined amazon network. To the right we can see the different partition sets that define a cluster or community as it is referred to with respect to network graphs. The way Girvan-Newman clustering algorithm approached clustering by finding modularity among different clusters.

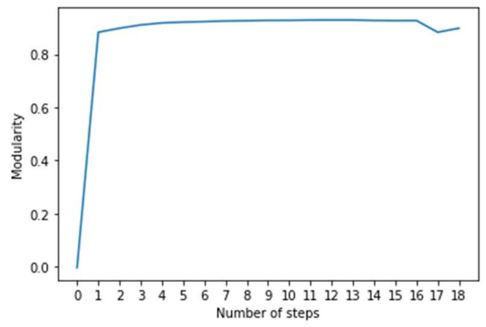
Modularity refers to the strength of connectedness within communities. High modularity in turns represents strong connections with different communities or clusters. To find these highest modularity partitions the approach that is used by the Girvan-Newman method is an iterative removal of the highest betweenness centrality for all links. This creates a partition sequence with varying numbers of partitions. In theory the set with the highest modularity should be chosen. As we can see this is not always clear cut as it would seem that there seemed to be very small marginal improvement after 2 partitions yet the highest had 13 different segments.



```
def girvan_newman_partition(self, graph, numClusters):
    # PARAMS: graph object and # of clusters
    # RETURNS clustered partitions that can be fed in as node colors
    # Method of generating partitions and returns clustered partitions of network
    partition = list(nx.community.girvan_newman(graph))[numClusters - 2]
    partitionMap = self.createPartitionMap(partition)
    return [partitionMap[i] for i in graph.nodes()]
```

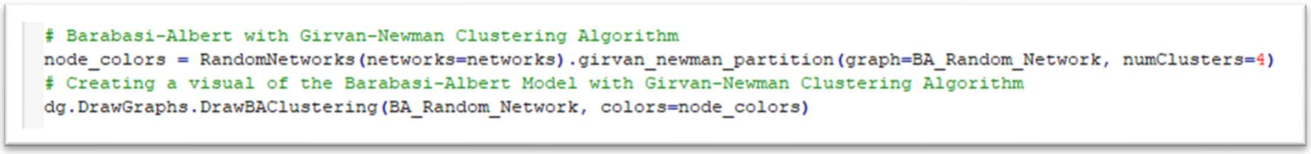


```
Best number of steps for optimal modularity(clustering): 13
with Modularity of 0.9291940000000002
```



Barabasi-Albert Model: Girvan Newman Clustering Algorithm…

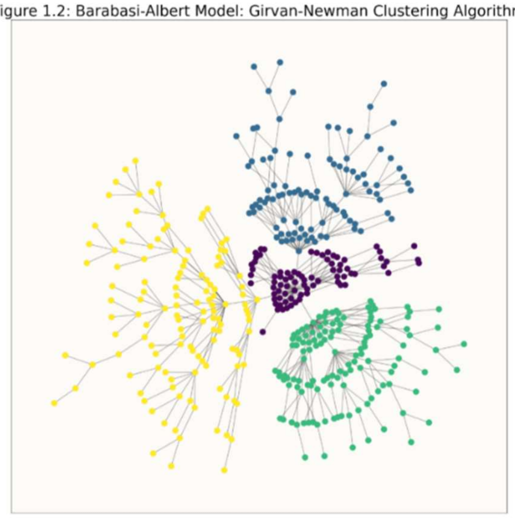
The implementation of the Girvan-Newman Clustering Algorithm on the Barabasi-Albert Model is documented in the below code snippet. The code utilizes the `girvan_newman_partion()` function defined in the previous section. A predetermined number of four partitions is specified.



```
# Barabasi-Albert with Girvan-Newman Clustering Algorithm
node_colors = RandomNetworks(networks=networks).girvan_newman_partition(graph=BA_Random_Network, numClusters=4)
# Creating a visual of the Barabasi-Albert Model with Girvan-Newman Clustering Algorithm
dg.DrawGraphs.DrawBAClustering(BA_Random_Network, colors=node_colors)
```

The Kamada-Kawai layout from the NetworkX Library separates the four partitions of the Girvan Newman Clustering Algorithm cleanly as demonstrated in the visual.

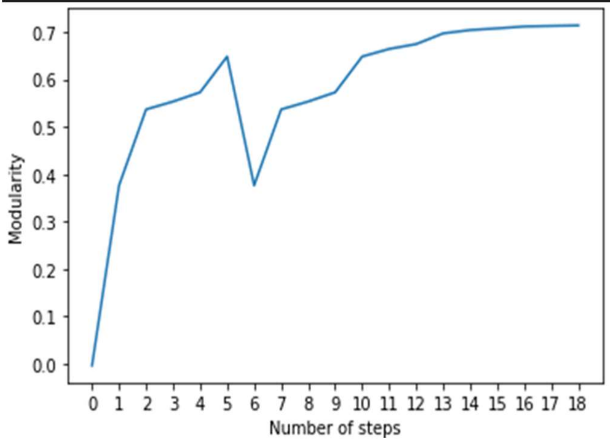
Figure 1.2: Barabasi-Albert Model: Girvan-Newman Clustering Algorithm

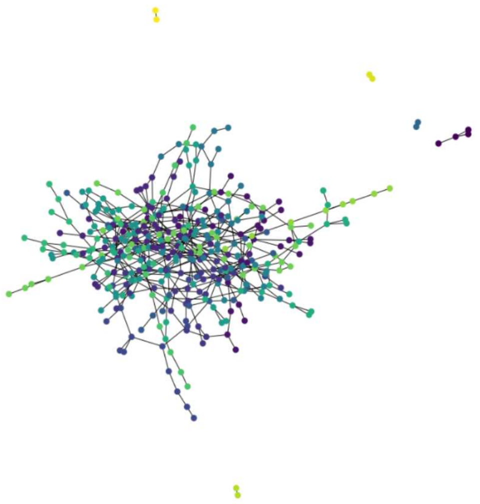


Erdos-Renyi Model: Girvan Newman Clustering Algorithm…

We can see that the modularity does trail off as clearly as with the raw network. This is hard to interpret without the underlying labels but what appears to be occurring is the edge nodes that may not have been connected are now being randomly connected across the center of the graph instead of being left hanging. This could be suggesting groupings of edge products or users as they span across the entire network. We still see a composition of different groups making up the center of our graph. We also see the lower modularity associated with these partitions.

```
Best number of steps for optimal modularity(clustering): 19
with Modularity of 0.7135319999999999
```

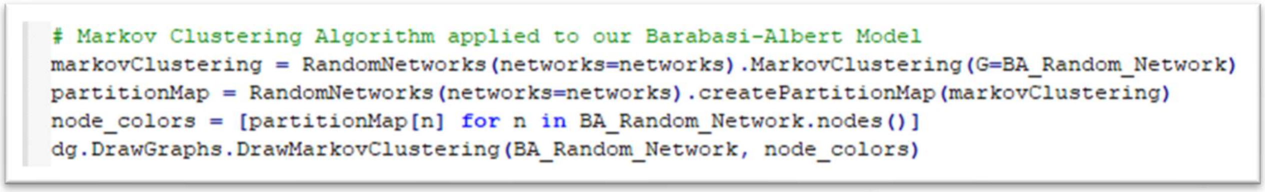




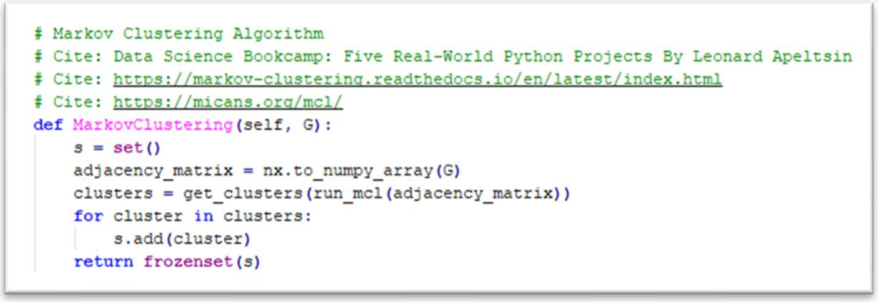
Markov Clustering Algorithm…

The Markov Clustering Algorithm models stochastic flow in Networks. Interestingly, it was the only clustering algorithm in our project that didn't require a pre-defined number of a partitions. In other words, it's classified as an unsupervised learner. This made it unique and potentially interesting because it was 'creative' or 'generative' in partitioning the network.

The implementation of the Markov Clustering Algorithm utilizes the `markov_clustering` Library. We went to extra lengths to make sure the implementation aligned with the existing structure of our project since it was added in the later stages of development. We pip installed and imported the library, reviewed the documentation, created the function `def MarkovClustering(self, G)`, and produced a visual of the Markov Clustering Algorithm on the Barabasi-Albert Random Network Model. The MarkovClustering function returns a 'frozenset' which we programmed because the NetworkX library built-in methods for clustering all return frozensets (thus, this made the function mesh well with the existing project code).



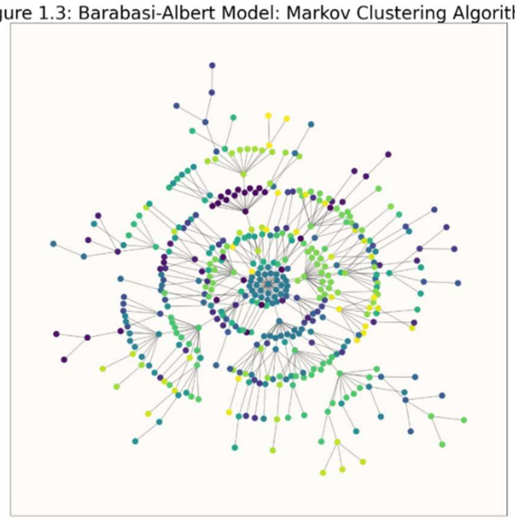
```python
# Markov Clustering Algorithm applied to our Barabasi-Albert Model
markovClustering = RandomNetworks(networks=networks).MarkovClustering(G=BA_Random_Network)
partitionMap = RandomNetworks(networks=networks).createPartitionMap(markovClustering)
node_colors = [partitionMap[n] for n in BA_Random_Network.nodes()]
dg.DrawGraphs.DrawMarkovClustering(BA_Random_Network, node_colors)
```



```python
# Markov Clustering Algorithm
# Cite: Data Science Bookcamp: Five Real-World Python Projects By Leonard Apeltsin
# Cite: https://markov-clustering.readthedocs.io/en/latest/index.html
# Cite: https://micans.org/mcl/
def MarkovClustering(self, G):
    s = set()
    adjacency_matrix = nx.to_numpy_array(G)
    clusters = get_clusters(run_mcl(adjacency_matrix))
    for cluster in clusters:
        s.add(cluster)
    return frozenset(s)
```

When reviewing the visual output, we can observe that the number of partitions is high by counting the number of different colored nodes.

Figure 1.3: Barabasi-Albert Model: Markov Clustering Algorithm



Code Design

Our project was developed using Git + Azure DevOps which brought an aspect of professionalism to the process considering the importance of version control and DevOps tooling used in industry. It was also helpful given the fact that each of the team members were in separate time zones.

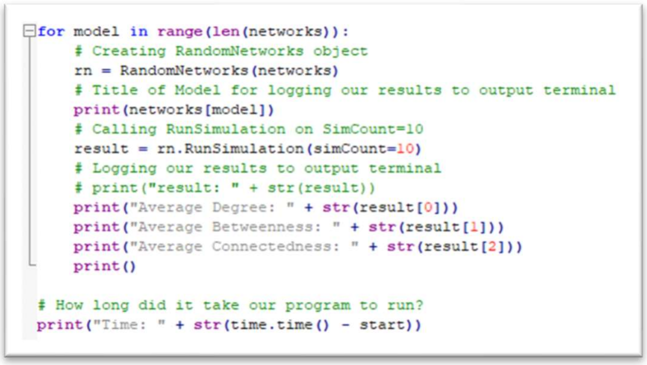
We began development with Professor McDonalds invaluable advice: 1) ‘start small and build’, 2) ‘first get it working, then get it working fast’. More specifically, we started by only using the `SimulateOnce()` method from the base MonteCarlo.py class; gradually expanding capabilities from this simple starting point.

Our project contains three files and largely follows a code structure similar to the examples used in class, including: 1) a DrawGraphs.py class used for generating visuals and keeping our main file clean, 2) the MonteCarlo.py base class, and 3) a RandomNetworks.py file which holds the specifics of our simulation implementation.

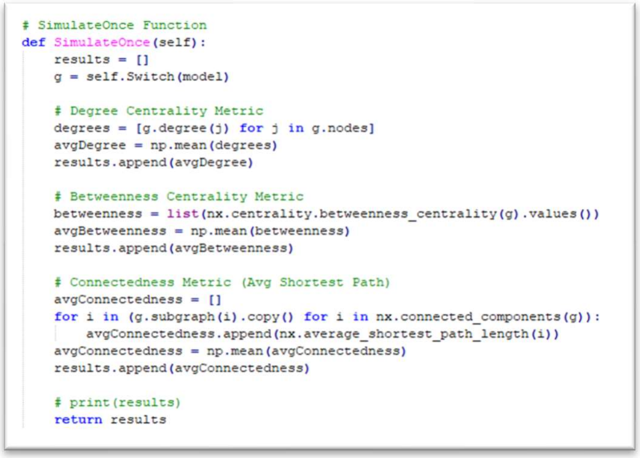
We did most of our developing/testing at `simCount=10` [~90 seconds]. We also ran the program at `simCount=100` [~400 seconds] and `simCount=1000` [~4,000 seconds] (*NOTE: you must comment out the convergence clause ‘if statements’ when using greater than `simCount=100` for the `RunSimulation()` method of the base MonetCarlo.py file).

Main components of the RandomNetworks.py file…

The `RandomNetworks` class takes a `MonteCarlo` object, defines a series of functions, and loops through a list of random network models while calling the `RunSimulation()` method for each specific random network model. This allowed us to run the program once; avoiding having to manually run each model separately [or] creating multiple objects with repeatable code. The `SimulateOnce()` function captures and stores our measurements in a results list for each `simCount` which is passed to our MonteCarlo.py base class.



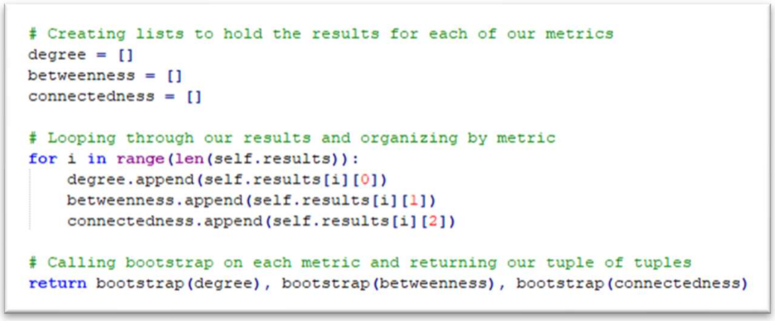
```
for model in range(len(networks)):
    # Creating RandomNetworks object
    rn = RandomNetworks(networks)
    # Title of Model for logging our results to output terminal
    print(networks[model])
    # Calling RunSimulation on SimCount=10
    result = rn.RunSimulation(simCount=10)
    # Logging our results to output terminal
    # print("result: " + str(result))
    print("Average Degree: " + str(result[0]))
    print("Average Betweenness: " + str(result[1]))
    print("Average Connectedness: " + str(result[2]))
    print()

# How long did it take our program to run?
print("Time: " + str(time.time() - start))
```



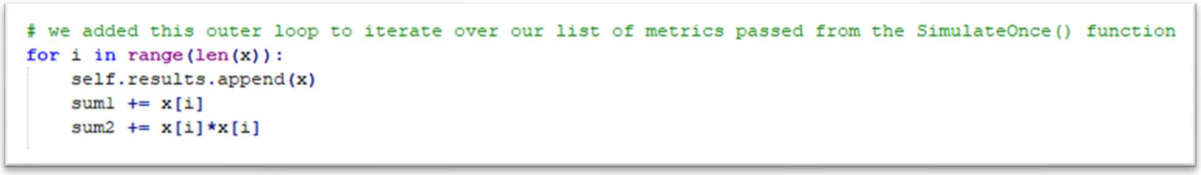
```
# SimulateOnce Function
def SimulateOnce(self):
    results = []
    g = self.Switch(model)

    # Degree Centrality Metric
    degrees = [g.degree(j) for j in g.nodes]
    avgDegree = np.mean(degrees)
    results.append(avgDegree)

    # Betweenness Centrality Metric
    betweenness = list(nx.centrality.betweenness_centrality(g).values())
    avgBetweenness = np.mean(betweenness)
    results.append(avgBetweenness)

    # Connectedness Metric (Avg Shortest Path)
    avgConnectedness = []
    for i in (g.subgraph(i).copy() for i in nx.connected_components(g)):
        avgConnectedness.append(nx.average_shortest_path_length(i))
    avgConnectedness = np.mean(avgConnectedness)
    results.append(avgConnectedness)

    # print(results)
    return results
```

Modifications to the MonteCarlo.py Base Class…

We made several modifications to the MonteCarlo.py base class as documented in the below code snippets. We added an outer loop to iterate over our list of metrics passed from the `SimulateOnce()` function. We created three separate lists to hold the results for each of our metrics. We loop through our results and organize the information by metrics before calling the `bootstrap()` function on each metric and returning a tuple of tuples as documented in the below diagram and output snippet.

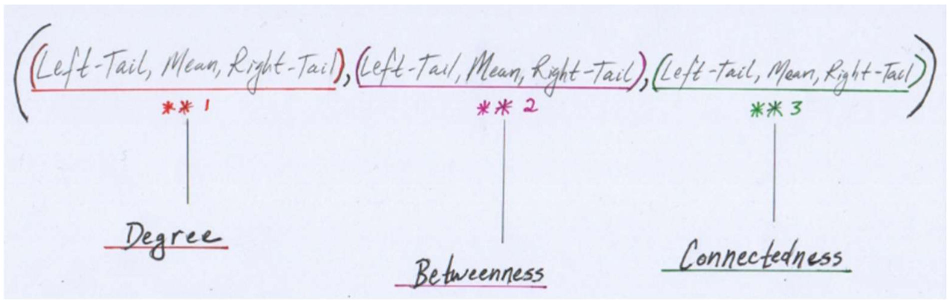


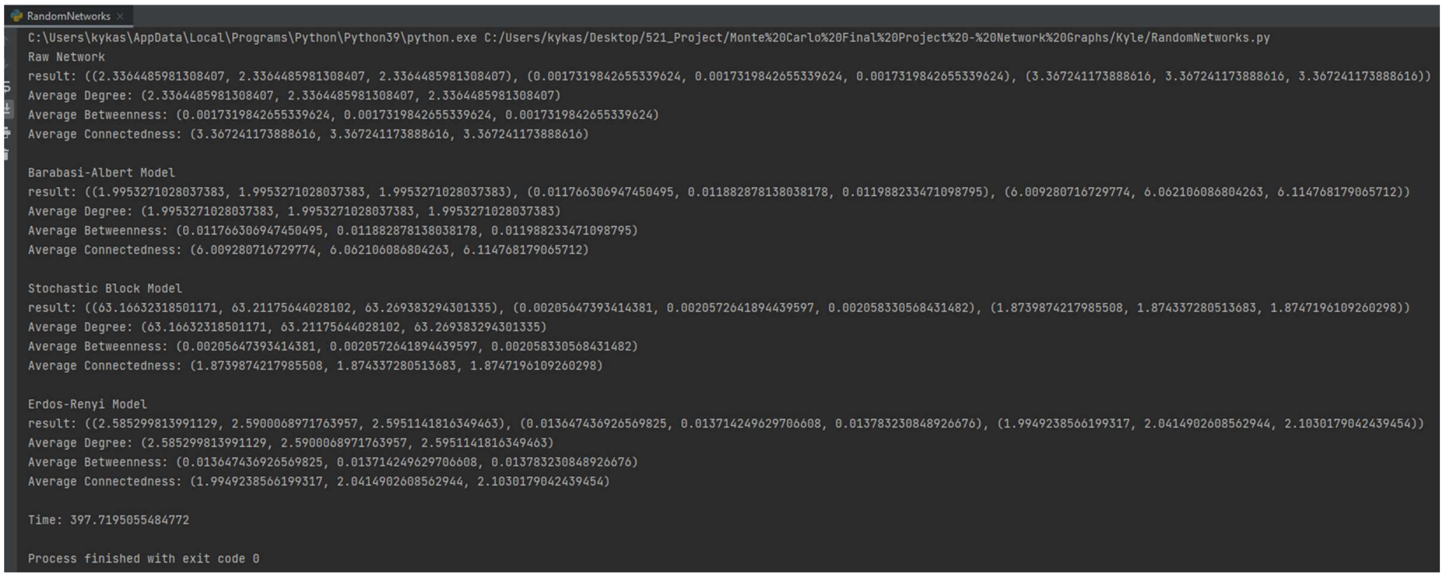
```
# Creating lists to hold the results for each of our metrics
degree = []
betweenness = []
connectedness = []

# Looping through our results and organizing by metric
for i in range(len(self.results)):
    degree.append(self.results[i][0])
    betweenness.append(self.results[i][1])
    connectedness.append(self.results[i][2])

# Calling bootstrap on each metric and returning our tuple of tuples
return bootstrap(degree), bootstrap(betweenness), bootstrap(connectedness)
```



```
# we added this outer loop to iterate over our list of metrics passed from the SimulateOnce() function
for i in range(len(x)):
    self.results.append(x)
    sum1 += x[i]
    sum2 += x[i]*x[i]
```

Output Formatting…


((Left-Tail, Mean, Right-Tail), (Left-Tail, Mean, Right-Tail), (Left-tail, Mean, Right-Tail))
**1
**2
**3
Degree
Betweenness
Connectedness



```
C:\Users\kykas\AppData\Local\Programs\Python\Python39\python.exe C:/Users/kykas/Desktop/521_Project/Monte%20Carlo%20Final%20Project%20-%20Network%20Graphs/Kyle/RandomNetworks.py
Raw Network
result: ((2.3364485981308407, 2.3364485981308407, 2.3364485981308407), (0.0017319842655339624, 0.0017319842655339624, 0.0017319842655339624), (3.367241173888616, 3.367241173888616, 3.367241173888616))
Average Degree: (2.3364485981308407, 2.3364485981308407, 2.3364485981308407)
Average Betweenness: (0.0017319842655339624, 0.0017319842655339624, 0.0017319842655339624)
Average Connectedness: (3.367241173888616, 3.367241173888616, 3.367241173888616)

Barabasi-Albert Model
result: ((1.9953271028037383, 1.9953271028037383, 1.9953271028037383), (0.011766306947450495, 0.011882878138038178, 0.011988233471098795), (6.009280716729774, 6.062106086804263, 6.114768179065712))
Average Degree: (1.9953271028037383, 1.9953271028037383, 1.9953271028037383)
Average Betweenness: (0.011766306947450495, 0.011882878138038178, 0.011988233471098795)
Average Connectedness: (6.009280716729774, 6.062106086804263, 6.114768179065712)

Stochastic Block Model
result: ((63.16632318501171, 63.21175644028102, 63.269383294301335), (0.0020564739341438l, 0.002057264189443959 7, 0.002058330568431482), (1.8739874217985508, 1.874337280513683, 1.8747196109260298))
Average Degree: (63.16632318501171, 63.21175644028102, 63.269383294301335)
Average Betweenness: (0.00205647393414381, 0.0020572641894439597, 0.002058330568431482)
Average Connectedness: (1.8739874217985508, 1.874337280513683, 1.8747196109260298)

Erdos-Renyi Model
result: ((2.585299813991129, 2.5900068971763957, 2.5951141816349463), (0.013647436926569825, 0.013714249629706608, 0.013783230848926676), (1.9949238566199317, 2.0414902608562944, 2.1030179042439454))
Average Degree: (2.585299813991129, 2.5900068971763957, 2.5951141816349463)
Average Betweenness: (0.013647436926569825, 0.013714249629706608, 0.013783230848926676)
Average Connectedness: (1.9949238566199317, 2.0414902608562944, 2.1030179042439454)

Time: 397.7195055484772

Process finished with exit code 0
```

Obstacles

Throughout the development of the project, we took note of four obstacles, including: 1) the size of the dataset, 2) handling graph data structures 3) performing operations on graphs, and 4) partitioning. The unifying theme – time, space, and complexity concerns. We considered optimizations such as using NumPy structures, but we were limited by the incompatibility of types across the major components of our program. Also, we considered using Google's Colab product for increased computing power in testing large datasets. We have included a few notes to document our strategy in addressing these four concerns throughout project development

The raw dataset is enormous (Nodes=91.8K, Edges=125.7K). We quickly realized that even something as simple as loading the dataset into our program was costly. Therefore, we experimented with reducing the size in a series of decreasing intervals (5,000 entries, 2,500 entries, 1,000 entries, etc.) until the dataset became manageable. We are currently using the first 500 entries of the dataset which seemed to provide a good balance between network robustness (and) time, space, and complexity.

The NetworkX Python library provided many convenient methods which have considerably reduced the difficulties associated with network graph data structures and operations. That said, we placed extra effort in understanding, reviewing, and reading the documentation for the methods used within our project. In fact, it was necessary in order to make all the separate components of the program cohesive.

Partition operations are the most expensive operations within our program. This is immediately observable when running the program. Nonetheless, the functionality was vital to our clustering efforts and had two layers – 1) creating the partitions from our data structures, 2) creating the visual appearance of partitions from our data structures. These two layers had to be handled separately but consistently.

Summary

Key Results and Insights…

For each random network model, metrics seems to algin with the raw network metrics in 1-2 out of 3 cases. For the Barabasi-Albert Model, we see similarities in degree while the betweenness and connectedness metrics are not consistent. For the Stochastic Block Model, we see similarities for the betweenness and connectedness metrics while the degree metric is much higher. For the Erdos-Renyi Model we see similarities for the degree while the betweenness and connectedness metrics are not consistent.

It was difficult to derive meaning from the results of the clustering algorithms without the underlying product labels. That said, categorical and thematic structures are definitely present in our network models. This is intuitive if we think about the process for which the dataset represents. If you've ever purchased anything on Amazon, you will see a section labelled "customers who purchased item 'a' also purchased items 'b,c,d, etc.'; these items are grouped in some pre-existing way. We note that many variations of categorical and thematic structures can emerge from simulating random network models and applying clustering algorithms (including subsets of those categories and in the case of the Markov Clustering Algorithm – potentially categories that hadn't existed previously).

Requirements for further analysis…

In order to extend the analysis, we would require the underlying product labels for the nodes. It would also be advantageous to obtain the pre-existing (sets, categories, themes, subsets, groupings, etc.) of those items within the Amazon marketplace.

Appendix + Notes

• Output Snippets, Code Snippets, and Visuals are included throughout the report as we felt that including them with context and explanation provided for clear and concise reporting.

• The project source code is attached under *source.zip*.

> • NOTE: The Final Version is located within the *Merge* Folder. (Don't forget to change the file-path for the dataset in the *RandomNetworks.py* file).
> • The Jupyter Notebook and Tate's underlying source code (before the Merge) can be found within the *Tate* Folder.

• Both Individual Reports are attached under *TateTeage_IndividualReport.pdf* and *KyleKassen_IndividualReport.pdf*.