Assignment 2: Community Detection Using the Leiden Algorithm

Laxmi Vijayan

June 7, 2023

1 Introduction

Networks are useful for modeling complex systems across several domains, such as biology, sociology, and computer science. Some examples of real-world networks include the neural network and metabolic network, social networks such as Facebook and Reddit, and the internet and World Wide Web.[8] These networks can be understood by the study of attributes such as similarity and dissimilarity between members, grouping, and organization. For example, by identifying a group of densely connected neurons, one might identify a common function. Alternatively, by identifying a group of people that are closely tied, one might identify a community of friends. As such a large body of work exists on how to detect such communities within networks.[8]

Detecting communities is a conceptually and practically challenging task due to several reasons. First, when discussing real-world networks, the definition of community can be ambiguous and multitudinous; it can be defined based on various attributes, and one node can belong to several different communities.[6] Second, once a network has been partitioned into separate communities, the evaluation of the communities needs to be done manually to verify if there is some common attribute shared between members. This process can be subjective depending on the data and leaves room for questions.[6] Lastly when dealing with large datasets, computational complexity is yet another consideration.

Various techniques to partition a graph into communities and evaluate the quality of the partition have been developed. A community is broadly defined here as a group of nodes that are densely connected to each other while sparsely connected to nodes outside the group. Graph partitioning aims to group nodes to form clusters (or communities) such that communities best fit the aforementioned definition and each node is assigned to only one cluster. To verify that clusters meet this aim, a quality function, or quantitative criterion, is used to evaluate the partition.[3]

In this report, a large network from the Stanford Network Analysis Platform's Large Network Dataset Collection was cleaned, partitioned using different quality functions, and evaluated to examine different aspects of community detection.[7]

2 Methods

2.1 Network and Graph Cleaning

Stack Overflow is a stack exchange website where users can post questions and receive answers. A temporal network where an edge represented that user u answered user v's question at time t was created using the entirety of Stack Overlow's history up to March 6, 2016. This network was chosen due to a familiarity with the structure of the website and an interest in understanding how clustering of this data might reflect topic domains on the website.[9]

For the purposes of this exploration, the attribute of time was discarded. Therefore, this might have resulted in parallel edges. Further, it is also possible that the user who posted the question may have answered their own question at a later time resulting in self-loops.

The .TXT file containing an edge list was read into a graph object using the NetworkX package in Python and transformed into an undirected graph.[4] The script used to remove self-loops and parallel edges is included in Appendix A. Some basic metrics about the cleaned graph object which was then exported as an edge list to The cleaned graph object were then exported as a .TSV file.

2.2 Quality Functions and Graph Partitioning

The Leidenalg package for Python was used to conduct the graph partitions.[10] First, the .TSV edge list was loaded into a graph object using the iGraph package for Python.[1] The scripts used to partition can be found in Appendix B.

Two different partition types, Modularity and the Constant Pott's Model (CPM), were used to get partitions with different parameters from the graph. The partition types were determined by which quality function they implemented to ensure the partition meets the quality criterion.

2.2.1 Modularity

One of the most popular quality functions is modularity. This function aims to optimize the compartmentalization of nodes in a graph and can be written in several different ways.[8] This is a formulation of the summation of communities:

$$Q = \frac{1}{2m} \sum_c \left(m_c - \frac{K_c^2}{4m} \right) \tag{1}$$

where m is the total number of edges, m_c is the number edges inside the community c and K_c^2 is the total degree of nodes in the community c.[10] The term $\frac{K_c^2}{4m}$ represents the expected number of edges within the community c by assuming that the edges in the network are randomly distributed. Finally, this $\left(m_c - \frac{K_c^2}{4m}\right)$ represents the difference between the actual number of edges within the community and the expected number of edges. By summing this difference over all communities and normalizing it with the term $\frac{1}{2m}$, the modularity function Q determines the overall difference between the expected and actual number edges and maximizing it helps identify communities that are more densely connected than expected in a random network. A larger Q indicates a stronger community structure.

2.2.2 Constant Pott's Model

The Constant Pott's Model (CPM) is another quality function. This function uses a linear resolution parameter (γ). It can be written in several ways and this is a formulation of the summation over communities:

$$Q = \sum_c \left[m_c - \gamma \binom{n_c}{2} \right] \tag{2}$$

where m_c is the total number of edges inside the community c and $\binom{n_c}{2}$ represents the number of possible edges within the community (n_c represents the number of nodes in community c).[10] By summing the difference between the total number of edges and the total possible number of edges over all communities, the CPM function determines the difference between the actual number of edges and the expected number edges in a random network. Maximizing this value helps identify more densely connected communities. The γ term allows for adjusting the resolution of the communities detected; a small values results in fewer, larger and less dense communities, while larger values yield many smaller, more dense communities. The internal edge density of communities is higher than γ, while external edge density of communities is lower.

Seven partitions were conducted using this partition type with the following resolution parameter values: $0.001, 0.005, 0.01, 0.1, 0.25, 0.5, 0.75$.

2.3 Analysis

The partitions returned a .TSV file containing node IDs and cluster IDs. This file was loaded into a dataframe using the pandas library.[11] Then, descriptive statistics were captured using the NumPy library.[5] Scripts for the analysis can be found in Appendix C.

3 Results

Before processing the temporal network, it had $2,464,606$ nodes and $16,266,395$ edges. After cleaning, all nodes were retained, but $480,579$ edges were discarded. The graph density was calculated using the NetworkX density function, which is given by:

$$d = \frac{2m}{n(n-1)}, \tag{3}$$

where m represents the number of edges and n represents the nodes in the graph. A density of 0 represents an unconnected graph, while a density of 1 represents a complete graph. Finally, there were $45,250$ connected components within this graph (Table 1).

Parallel Edges	False
Self-Loops	None
Number of Nodes	2,464,606
Number of Edges	15,786,816
Graph Density	5.198e-06
Connected Components	45,250

Table 1: Summary statistics of the graph. The table provides an overview of the characteristics of the graph after processing to remove self-loops and parallel edges. It includes information on parallel edges, self-loops, the number of nodes, the number of edges, graph density, and the number of connected components. The graph consists of 2,464,606 nodes and 15,786,816 edges, with a density of 5.198e-06. It exhibits 45,250 connected components.

After partitioning the graph, the distribution of cluster sizes for each partition type and resolution parameter was plotted. The Modularity partition had the fewest number of clusters and the greatest cluster size, followed by CPM 0.001. The partition with the largest number of clusters of approximately similar size was CPM 0.25. This can be seen in Figure 1.

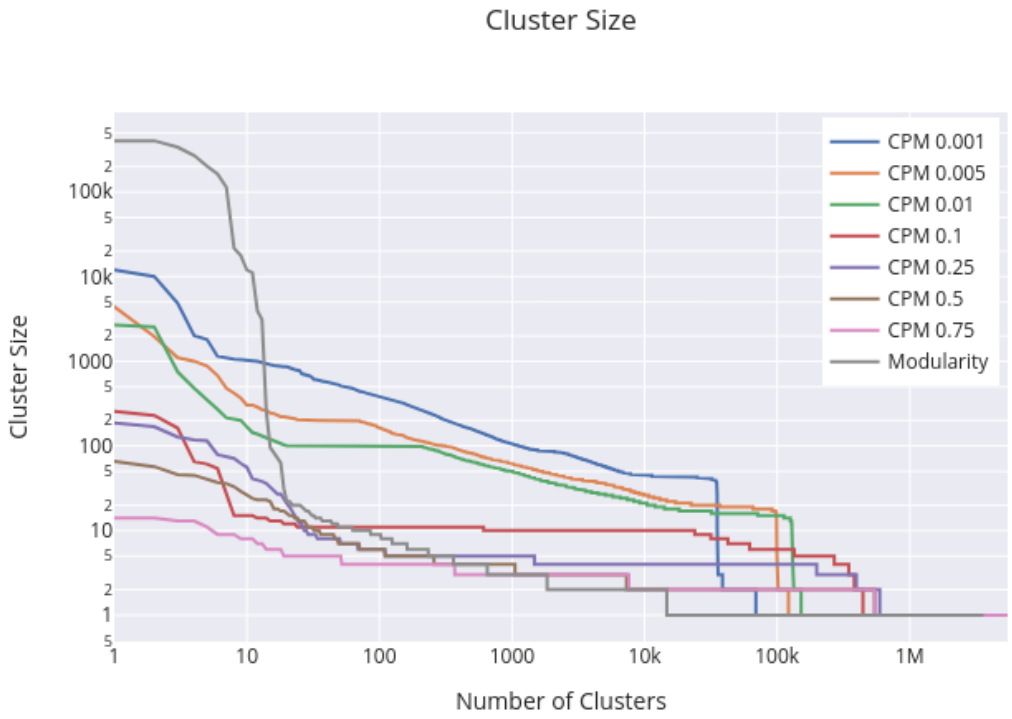

Figure 1: Log-log graph of cluster size. The cluster size for each cluster in each partition was graphed on a log-log scale. The modularity partition shows few, very large clusters. Similarly, CPM 0.001 also shows few, large clusters. Comparatively, CPM 0.25 shows a larger distribution of similarly sized clusters.

Finally, descriptive statistics for each partition were calculated. The clustering efficiency was calculated as the percent of nodes in clusters greater than a size of one. These results are summarized in Table 2.

Partition (Resolution)	Min	25^{th} Q	Median	Mean	75^{th} Q	Max	Efficiency
CPM (0.001)	1	1.0	1.0	1.461	1.0	40,945	1.675
CPM (0.005)	1	1.0	1.0	1.526	1.0	13,023	3.082
CPM (0.01)	1	1.0	1.0	1.537	1.0	6,932	3.865
CPM (0.1)	1	1.0	1.0	1.405	1.0	775	10.339
CPM (0.25)	1	1.0	1.0	1.246	1.0	220	12.288
CPM (0.5)	1	1.0	1.0	1.101	1.0	75	9.924
CPM (0.75)	1	1.0	1.0	1.101	1.0	18	9.924
Modularity	1	1.0	1.0	1.671	1.0	424,557	0.408

Table 2: Cluster Size Distribution The table presents a summary of cluster distribution based on different partition types (CPM and Modularity). For CPM, it also displays the resolution parameter used. The minimum cluster size, 25^{th} quartile, median, mean, 75^{th} quartile, maximum, and clustering efficiency (as a percentage) are provided. The minimum cluster size for all partitions was 1, and the maximum size varied. The cluster sizes were large for small resolution parameters and for modularity. CPM with a resolution of 0.25 had the greatest clustering efficiency with approximately 12% of all clusters containing more than one node.

4 Discussion

The Stack Overflow Answers to Questions network was selected to examine whether clustering could help identify topical domains based on user behavior. The stack exchange website is structured so that one question can have up to five topic tags. These tags and their synonyms are controlled, and as of June 6, 2023, there are approximately 50,000 tags. This information is relevant so far as to understand some possible distribution of user behavior across these topics. Some of these tags, such as 'java' and 'python' may see more use than more obscure or specialized tags. Further, several users likely participated on the website once to either ask a question or answer one, while a few might be super users that participate more regularly. We predicted that we might see a distribution of small clusters in various domains, representing the former user behavior, with a few larger clusters, that represent the latter use case. By this interpretation, CPM 0.1 or CPM 0.25 which showed a more balanced community structure is probably most ideal. However, we neither have tag information as attributes for the edges nor can we manually verify user behavior without additional information.

Since we expect several small communities rather than large ones, and given the large network size, it is reasonable to assert that the modularity partition is not ideal for this cluster. It has been shown to have a resolution limit that may result in it hiding smaller communities.[2]

Several additional metrics might be useful to evaluate before determining which partition type is ideal. Examining the conductance and modularity of each of the clusters might provide more insight into whether these communities are meaningful network structures or just patterns that emerged due to random chance.

5 Conclusion

This was a hands-on approach to examining community detection in real-world networks and understanding the various roadblocks in the process. The ambiguous definition of "community," the lack of ground truth to validate the results, and the subjectivity of the evaluation was made most evident while working with the data. One obvious next step would be to gather more rigorous metrics, such as conductance and modularity for each of the clusters, to better understand the community structures.

References

[1] Gabor Csardi and Tamas Nepusz. The igraph software package for complex network research, 2006.

[2] Santo Fortunato and Marc Barthélemy. Resolution limit in community detection. *Proceedings of the National Academy of Sciences*, 104(1):36–41, 2007.

[3] Santo Fortunato and Claudio Castellano. *Community Structure in Graphs*, pages 490–512. Springer New York, New York, NY, 2012.

[4] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[5] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[6] IEEE International Conference on Data Mining (ICDM). *Defining and Evaluating Network Communities based on Ground-truth*, 2012. Publisher: arXiv Version Number: 3.

[7] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[8] M. E. J. Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103(23):8577–8582, 2006.

[9] Ashwin Paranjape, Austin R. Benson, and Jure Leskovec. Motifs in temporal networks. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, WSDM '17, pages 601–610, New York, NY, USA, 2017. Association for Computing Machinery.

[10] V. A. Traag, L. Waltman, and N. J. van Eck. From Louvain to Leiden: guaranteeing well-connected communities. *Scientific Reports*, 9(1):5233, 2019.

[11] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010.

Appendix A Script for Cleaning Network

```python
import sys
import networkx as nx

if len(sys.argv) < 2:
    sys.exit("Provide input file as the second argument please.")
elif len(sys.argv) < 3:
    output_file = sys.argv[1][:-4] + "Cleaned.tsv"
else:
    output_file = sys.argv[2]

summary_file = sys.argv[1][:-4] + "SummaryStats.txt"

with open(sys.argv[1]) as f:
    g = nx.Graph([line.split()[:2] for line in f])

g.remove_edges_from(nx.selfloop_edges(g))

nx.write_edgelist(g, output_file, delimiter='\t', data=False)

multiedges = g.is_multigraph()
self_loops = nx.number_of_selfloops(g)
num_nodes = g.number_of_nodes()
num_edges = g.number_of_edges()
density = nx.density(g)
connected_comp = nx.number_connected_components(g)

with open(summary_file, 'w') as f:
  f.write(f"Cleaned Graph's Summary Statistics\n\n")

  f.write(f'Parallel Edges: {multiedges}\n')
  f.write(f'Self-Loops: {self_loops}\n')

  f.write(f'Number of Nodes: {num_nodes}\n')
  f.write(f'Number of Edges: {num_edges}\n')
  f.write(f'Graph Density: {density}\n')
  f.write(f'Connected Components: {connected_comp}\n')
```

Script 1: A Python script that accepts an edge list as a .TXT file and removes any self-loops and parallel edges. It returns a .TSV file of an edge list for an undirected graph as well as a .CSV file with some basic metrics about the cleaned graph. It utilizes the NetworkX library.

Appendix B Scripts for Data Clustering

```
import sys
import leidenalg as la
import igraph as ig

if len(sys.argv) < 3:
    sys.exit('Please provide input file as 2nd argument and desired resolution as third argument please.')

output_file = 'CPM' + sys.argv[2] + sys.argv[1][:-11] + 'Clustered.tsv'

G = ig.Graph.Load(sys.argv[1], format='edgelist', directed=False)

x = la.find_partition(G, la.CPMVertexPartition, resolution_parameter = float(sys.argv[2]))

with open(output_file, 'w') as f:
  for n, m in enumerate(x.membership):
    f.write(f'{n}\t{m}\n')
```

Script 2: Python script that clusters an edge list using a given resolution parameter and the CPM partition type from the leidenalg library.

```
import sys
import leidenalg as la
import igraph as ig

if len(sys.argv) < 2:
    sys.exit('Please provide input file as 2nd argument.')

output_file = 'mod' + sys.argv[1][:-11] + 'Clustered.tsv'

G = ig.Graph.Load(sys.argv[1], format='edgelist', directed=False)

x = la.find_partition(G, la.ModularityVertexPartition, seed = 1234)

with open(output_file, 'w') as f:
  for n, m in enumerate(x.membership):
    f.write(f'{n}\t{m}\n')
```

Script 3: A Python script that cluster an edge list using a given resolution parameter and the Modularity partition type from the leidenalg library.

Appendix C Scripts for Simple Analysis

```
import pandas as pd
import numpy as np

statistics = []

file_paths = ['CPM0.001A2Q.tsv', 'CPM0.005A2Q.tsv', 'CPM0.01A2Q.tsv',
              'CPM0.1A2Q.tsv', 'CPM0.25A2Q.tsv', 'CPM0.5A2Q.tsv',
              'CPM0.75A2Q.tsv', 'modA2Q.tsv']

column_names = ['node', 'cluster']

cpm0001 = pd.read_csv(file_paths[0], sep='\t', header=None, names=column_names)
cpm0005 = pd.read_csv(file_paths[1], sep='\t', header=None, names=column_names)
cpm001 = pd.read_csv(file_paths[2], sep='\t', header=None, names=column_names)
cpm01 = pd.read_csv(file_paths[3], sep='\t', header=None, names=column_names)
cpm025 = pd.read_csv(file_paths[4], sep='\t', header=None, names=column_names)
cpm05 = pd.read_csv(file_paths[5], sep='\t', header=None, names=column_names)
cpm075 = pd.read_csv(file_paths[6], sep='\t', header=None, names=column_names)
mod = pd.read_csv(file_paths[7], sep='\t', header=None, names=column_names)

files = [cpm0001, cpm0005, cpm001, cpm01, cpm025, cpm05, cpm075, mod]

file_names = ['CPM 0.001', 'CPM 0.005', 'CPM 0.01', 'CPM 0.1', 'CPM 0.25',
              'CPM 0.5', 'CPM 0.75', 'Modularity']

for file, file_name in zip(files, file_names):
    counts = file.groupby('cluster')['cluster'].count()
    statistics.append({
        'File': file_name,
        'Min': counts.min(),
        '25th Quartile': np.percentile(counts, 25),
        'Median': np.median(counts),
        'Mean': counts.mean(),
        '75th Quartile': np.percentile(counts, 75),
        'Max': counts.max(),
        'Percent > 1': (counts > 1).mean() * 100
    })

statistics_df = pd.DataFrame(statistics)

statistics_df.to_csv('clusterStatistics.csv', index=False)
```

Script 4: A Python script that uses the libraries pandas and numpy to calculate a simple statistical summary of the cluster sizes for each .TSV partition file.