RESEARCH NOTE

IDENTIFYING NEW GATEKEEPERS IN SOCIAL MEDIA NETWORKS

JONATHAN LEWIS*

Abstract

This research note implements the steps to identify sets of key nodes in networks, following a method proposed by Ortiz-Arroyo (2010) and employed by Jürgens et al. (2011) to identify “new gatekeepers” in Twitter networks of political communication. It provides Python code to reproduce Ortiz-Arroyo’s results, and discusses ways of reducing processing time when using the code to identify key nodes in larger networks.

I. Introduction

Although “fake news” has replaced “echo chambers” as the online moral panic du jour, much attention continues to be paid to the factors affecting the political information to which Internet users are exposed. While ideas such as echo chambers and the “filter bubble” (Pariser, 2011) are intuitively persuasive, researchers have investigated whether and to what extent such phenomena can be empirically observed, and furthermore what effect such selective exposure might have on political attitudes and behavior.

One effort in this area took the idea of gatekeeping that had long been used in traditional mass media studies and applied it to social media (Jürgens, Jungherr, & Schoen, 2011). In traditional media studies, gatekeepers are the media professionals, especially journalists and editors, who take decisions about what issues and stories to cover and which viewpoints to report in the stories they choose to run. These gatekeepers thus affect the information about politics and other important issues that reach citizens, and by extension influence the content and outcomes of political debates and elections. As digitalization and the Internet have revolutionized media production and consumption, media scholars have noted that a wider group of actors are starting to function as gatekeepers (Friedrich, Keyling, & Brosius, 2015).

Jürgens et al. focus on political communication on Twitter, where exchanges between users have a “small world” structure in which dense groups of friends are connected to other groups by a relatively small number of connections. Jürgens et al. posit that the users linking these smaller groups play the role of “new gatekeepers”, and that these new gatekeepers’ decisions about which content to retweet will affect the content of information reaching the wider online public.

* Graduate School of Social Sciences, Hitotsubashi University. jonathan_lewis@mac.com.
Jürgens et al. use a method of identifying these new gatekeepers suggested by Ortiz-Arroyo (Ortiz-Arroyo, 2010). This paper shows how to implement the equations outlined by Ortiz-Arroyo in Python code, as a first step to analyzing a dataset of connections between users tweeting in English and Japanese about the controversial issue of whaling. I hope that the code provided will be useful to others performing similar analysis. In addition, it provides an example of how to turn equations found in academic papers into working code, which might be useful for students who are more comfortable with code than equations.

II. Identifying New Gatekeepers

Ortiz-Arroyo discusses the various ways that have been developed to measure the importance of nodes, including degree centrality, betweenness, closeness, and eigenvector centrality, to name only the most widely used. As Borgatti had previously shown, these measures all make assumptions about the kind of exchanges taking place between network nodes, and therefore may be more or less appropriate to particular cases (Borgatti, 2005). Furthermore, Ortiz-Arroyo notes that most of these popular measures aim to identify individually important nodes rather than sets of important nodes. Turning to methods of identifying sets of important nodes, Ortiz-Arroyo notes previous work on the communication efficiency of networks, which Latora and Marchiori define in Equation 1 (Latora & Marchiori, 2004):

Equation 1

\[ E(G) = \frac{\sum_{i \neq j \in G} e_{ij}}{N(N-1)} = \frac{1}{N(N-1)} \sum_{i \neq j \in G} \frac{1}{d_{ij}} \]

where \( G \) is a network containing \( N \) nodes, and \( e_{ij} \) is “the communication efficiency, which is proportional to the inverse of \( d_{ij} \) (the shortest path length between two nodes \( i, j \))” (Ortiz-Arroyo, 2010).

We can implement this equation in Python code as shown in Code block 1. I am using the NetworkX library for network analysis which comes installed with Anaconda, the most widely used data science package for Python.

As Ortiz-Arroyo goes on to explain, Latora and Marchiori remove each node in the network one at a time, and calculate how much the efficiency of the network falls each time. Nodes that reduce network efficiency most are the key players. Both Ortiz-Arroyo and Jürgens et al. test their algorithms on a network used by Borgatti, so we do the same here. The network is shown in Figure 1; for a network of this size, it is easier to recreate it manually rather than search for the original online. We have saved the network in GraphML format.

In Code block 2 we open the network file, then calculate the efficiency first of the whole network, and then of the network minus one node in turn, each time using the function introduced in Code block 1. The results are shown in Code results 2. Visualizing this data makes it easy to compare our results to those of Ortiz-Arroyo to check if our algorithm is working properly. The code to do this is in Code block 3, and the resulting graph is shown in Figure 2. The actual scores differ slightly from those in the original, but the shape of the graph is identical.
Ortiz-Arroyo then offers two methods for calculating the importance of sets of nodes. The first uses the concept of connectivity, as shown in Equation 2:

**Equation 2**

\[ x(v) = \frac{\text{deg}(v)}{2N}, \quad N > 0 \]

Here \( \text{deg}(v) \) is the degree of a node \( v \), i.e. the number of edges to/from it, and \( N \) is the total number of edges in the network. Code block 4 implements this as a Python function.

The second method is called *centrality*, which “can be defined in terms of the number of..."
shortest or geodesic paths that have $v_i$ as source and the rest of nodes in the graph as targets” (Ortiz-Arroyo, 2010). The concept is explained in Equation 3; “$spaths(v)$ is the number of shortest paths from a given node to all the other nodes in the graph, and $spaths(v_1, v_2, \cdots, v_M)$ is the total number of shortest paths $M$ that exists across all the nodes in the graph.”

**Equation 3**

$$\gamma(v) = \frac{spaths(v)}{spaths(v_1, v_2, \cdots, v_M)}$$

We implement this equation in Code block 5. The important point here is that we call NetworkX’s all_shortest_paths function to obtain all the shortest paths between two nodes. Many network analysis tools will return a single shortest path between two nodes; however, in many networks there is more than one shortest path between any given pair of nodes. For example, in the network shown in Figure 1, the nodes a and e are connected by two shortest paths, a-b-e and a-d-e.

Connectivity and centrality as introduced above apply to individual nodes, but we can use them to calculate two different measures of the entropy of the network as a whole. The connectivity entropy and centrality entropy of a network are described by Ortiz-Arroyo in Equation 4 and Equation 5 respectively:

**Equation 4 Connectivity entropy**

$$H_{co}(G) = \sum_{i=1}^{n} x(v_i) \times \log_2 x(v_i)$$

**Equation 5 Centrality entropy**

$$H_{ce}(G) = \sum_{i=1}^{n} \gamma(v_i) \times \log_2 \gamma(v_i)$$

These equations are implemented as Python code in Code block 6 and Code block 7 respectively.

The next step is to remove one node at a time and see how far the two scores (for network connectivity and network centrality) differ from the score for the complete network. In each loop, we create a temporary copy of the network from which we will remove the node: see Code block 8. The results are output as follows (Code results 1):

**Code results 1**

**whole network connectivity entropy:** 4.112997562960809

**whole network centrality entropy:** 4.207125287453831

We then output the result as a graph showing the fall in entropy caused by the removal of each node; the code to do this is in Code block 9 and the resulting graph is shown in Figure 3. The key point shown by this graph, as Ortiz-Arroyo explains, is that centrality entropy variation reveals the three most important nodes in the network (see Figure 1) to be h, m and q. Node h is more important than node i despite the two having relatively similar positions in the network because if node h is removed then node r is completely isolated and the network is divided into three parts, whereas removing node i only divides the network into two parts. The N most important nodes in a network are therefore the N nodes that, when removed, have the most
impact on centrality entropy. Figure 3 suggests that centrality entropy variation is a better method than connectivity entropy variation for identifying key sets of players.

We now have working code for identifying key nodes, or new gatekeepers in networks of social media communication.

III. Final Word: Getting Results Quicker

The main problem using the code described here to identify key players in large networks is that it takes a long time to run. As a preliminary effort, we identified users who tweeted about whaling in Japanese and who sent or received at least one mention to each other; the largest component in this network had 5,872 nodes (users) and 9,808 edges (mentions). Running the code in Code block 9, it took approximately two hours to calculate the centrality entropy of the network after removing a node; this suggests that nearly 12,000 hours of computer time will be required to process all the nodes. Clearly, we need to find ways to bring this time down to a more realistic number. The lowest-hanging fruit is to run the code in parallel on multiple machines. It might also be possible to make use of multi-core processors to run the code in parallel on the same machine. We also need to compare network analysis libraries\(^1\) and consider ways of optimizing the code.

\(^1\) iGraph for one is said to be quicker than NetworkX, but on investigation iGraph for Python seems not to return all shortest paths between two nodes.
Code

Code block 1  Graph efficiency

```python
import networkx as nx

def graph_efficiency(network):
    total_shortest_path_length = 0
    # get the inversed lengths of the shortest paths between all nodes
    for source_node in network:
        for target_node in network:
            if source_node != target_node:
                if nx.has_path(network, source_node, target_node):
                    sh_path_ln = 1 /
                nx.shortest_path_length(network, source_node, target_node)
                # add them up
                total_shortest_path_length += sh_path_ln
    # finally, multiply by the normalizer
    nnodes = network.number_of_nodes()
    normalizer = 1 / (nnodes * (nnodes - 1))
    return(total_shortest_path_length * normalizer)
```

Code block 2  Impact on network efficiency

```python
import networkx as nx

whole_graph_efficiency = graph_efficiency(G)
print("Efficiency of the whole network:", str(whole_graph_efficiency))
data_list = []
node_list = []
for node in G.nodes():
    data_list.append([node])
node_list.append(node)
for i, node in enumerate(node_list):
    if node == 0:
        pdb.set_trace()
    H = nx.read_graphml('OrtizArroyoBorgatti.graphml')
    H = remove_isolates(H)
    H.remove_node(node)
    rem_efficiency = graph_efficiency(H)
data_list[i].append(rem_efficiency)
print(data_list)
```
Code results 2  Network efficiency

Efficiency of the whole network: 0.45012531328320776
{'a', 0.4538904450669154}, {'b', 0.45062247121070625}, {'c', 0.4516651104886398}, {'d', 0.4488185496420787}, {'e', 0.4538904450669154}, {'f', 0.4408263305322127}, {'g', 0.4332555244319494}, {'h', 0.28507625272331144}, {'i', 0.30631808278867084}, {'j', 0.41028114949683575}, {'k', 0.4526455026455024}, {'l', 0.4521008403361341}, {'m', 0.40130710954248346}, {'n', 0.45112044817927155}, {'o', 0.45591347650171155}, {'p', 0.4437130407718639}, {'q', 0.4370370370370371}, {'r', 0.45929038281979423}, {'s', 0.47167755991285387]

Code block 3  Visualizing efficiency results

```python
import pandas as pd
import matplotlib.pyplot as plt

df = pd.DataFrame.from_records(data_list, columns=['Node','Graph efficiency'])
df.set_index('Node', inplace = True)
df.plot()
plt.title('Efficiency Variation')
xlabels = tuple(df.index.values.tolist())
plt.xticks(np.arange(len(xlabels)), xlabels)
plt.xlim(0, 0.5)
plt.grid(True, axis='y')
plt.savefig('EfficiencyVariation.pdf', dpi=300)
plt.show()
```

Code block 4  Node connectivity

```python
def node_connectivity(network,node):
    total_edges = network.number_of_edges()
    degree = network.degree(node)
    return(degree / (2 * total_edges))
```

Code block 5  Node centrality

```python
def node_centrality(network,node):
    total_shortest_paths = 0
    this_source_node_count = 0
    for source_node in network:
        for target_node in network:
            if source_node != target_node:
                if nx.has_path(network, source_node, target_node):
                    this_node_all_shortest = nx.all_shortest_paths(network, source_node, target_node)
                    for shortest_path in this_node_all_shortest:
                        if source_node == node:
                            this_source_node_count += 1
                            total_shortest_paths += 1
    return(this_source_node_count / total_shortest_paths)
```
Code block 6  Connectivity entropy

def connectivity_entropy(network):
    # first calculate the connectivity of every node in the network
    # store results in a dictionary
    connectivity_dict = {}
    for node in network:
        connectivity_dict[node] = node_connectivity(network, node)
    connectivity_entropy = 0
    for node in connectivity_dict.keys():
        if connectivity_dict[node] > 0:
            this_node_connectivity_score = connectivity_dict[node] * np.log2(connectivity_dict[node])
        connectivity_entropy += this_node_connectivity_score
    return(-connectivity_entropy)
connectivity_entropy(G)

Code block 7  Centrality entropy

def centrality_entropy(network):
    centrality_dict = {}
    total_shortest_paths = 0
    node_count = network.number_of_nodes()
    for a, source_node in enumerate(network):
        this_source_node_path_count = 0
        for target_node in network:
            if source_node != target_node:
                if nx.has_path(network, source_node, target_node):
                    this_node_all_shortest = nx.all_shortest_paths(network, source_node, target_node)
                    for shortest_path in this_node_all_shortest:
                        this_source_node_path_count += 1
                        total_shortest_paths += 1
                    centrality_dict[source_node] = this_source_node_path_count
    for node in centrality_dict.keys():
        centrality_dict[node] = centrality_dict[node] / this_source_node_path_count
    centrality_entropy = 0
    for node in centrality_dict.keys():
        if centrality_dict[node] > 0:
            this_node_centrality_score = centrality_dict[node] * np.log2(centrality_dict[node])
        centrality_entropy += this_node_centrality_score
    return(centrality_entropy)
Network entropy with one node removed

```python
whole_conn_ent = connectivity_entropy(G)
whole_cent_ent = centrality_entropy(G)
print("whole network connectivity entropy: " + str(whole_conn_ent))
print("whole network centrality entropy: " + str(whole_cent_ent))
```

```python
data_list = []
node_list = []
for node in G.nodes():
    data_list.append([node])
    node_list.append(node)
for i, node in enumerate(node_list):
    H = nx.read_graphml('OrtizArroyoBorgatti.graphml')
    H = remove_isolates(H)
    H.remove_node(node)
    data_list[i].append(connectivity_entropy(H))
    data_list[i].append(centrality_entropy(H))
```

Visualizing entropy variation

```python
df = pd.DataFrame.from_records(data_list,columns =['Node','Connectivity entropy','Centrality entropy'])
df.set_index('Node',inplace = True)
df.plot()
plt.title('Entropy Variation')
xlabels = tuple(df.index.values.tolist())
plt.xticks(np.arange(len(xlabels)),xlabels)
plt.ylim(3.85, 4.2)
plt.grid(True,axis='y')
plt.savefig('EntropyVariation.pdf', dpi=300)
plt.show()
```

**References**

Ortiz-Arroyo, D. (2010). Discovering sets of key players in social networks. In A. Abraham,
A.-E. Hassanien, & V. Snasel (Eds.), *Computational Social Network Analysis* (pp. 27-47). Springer.