Spring 2017

# Community Detection for Counter-Terrorism

Patrick Michael Kelly
*Bard College*

# Community Detection for Counter-Terrorism

A Senior Project submitted to The Division of Science, Mathematics, and Computing of Bard College

By Patrick Kelly

Annandale-on-Hudson, New York

May 2017

# Abstract

Community detection in large networks is a process that has been heavily researched in the past decade due to the the emergence of online social networks. For Twitter, Inc., analyzing terrorists communities is vital in the fight against ISIS recruiters who use the twitter platform to radicalize people around the world. The goal of this project is to develop an algorithm which can accurately detect communities in large networks and to provide textual analysis on the discovered communities. Our algorithm combines the results of two unsupervised clustering algorithms to find communities in a given network. One algorithm uses the structure of the network, and the other algorithm uses the text associated with the nodes. Our algorithm is tested on a hand labeled ground truth twitter network and applied to an ISIS twitter recruiting network.

# Acknowledgements

I would like to thank...

- My parents for supporting me through everything.
- Professor Sven Anderson and Professor Khondaker Salehin for pushing me each week and advising my research with unending positive energy.
- My friends and teammates for the great times we've had together.
- Coach Adam Turner for helping me become better on and off the court.

# Dedication

I dedicate this project to Alexei Phillips '06.

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Background

This community detection algorithm utilizes concepts from the areas of network science and text analysis. The algorithm was inspired by the ISIS Twitter Recruiting data set (kaggle.com). The data set features 17,000 tweets by 112 ISIS recruiters from 2014-2016. There has been significant research in both network science and text analysis on the topic of clustering within each field independently, however, effectively combining methods from both fields has not yet been explored to a significant degree. The CESNA algorithm [7] uses network attributes to cluster sets of nodes into communities where nodes can be associated with multiple communities. First, it is important to acknowledge that nodes in communities share properties and have many relationships amongst themselves which are independent of their network structures. This is key to understanding that there are multiple sources of data which can be used to perform the clustering task. In our case, we have a network with text data(tweets) associated with the nodes. This means that our input data is more complex than the input data necessary to run CESNA.

In this paper, we consider **true communities** to be groups of people that share interests or live in the same area. The algorithm that we build is a community detection algorithm, but it is important distinguish the types of communities we strive to discover are different than

what typical community detection algorithms search for. While many researchers have defined "communities" as groups of densely connected nodes, this is only a part of detecting "true" communities as we define them.

The network is constructed by assigning each user to a node and each tweet @ another recruiter as an edge. Take for example the following tweet written by the recruiter named @ISBAQIYA "@spamci16: More than 70000 childrens have been killed by Assad regime. Why cry only for the drowned child? Hypocricy of the world." This tweet would lead to the construction of an edge between @ISBAQIYA and @spamci16. When analyzing the text, all tweets from one user are appended together and treated as one document. The following sections feature brief explanations of the theory behind each step of our algorithm which utilizes both the network structure and the text data to group the users into communities.

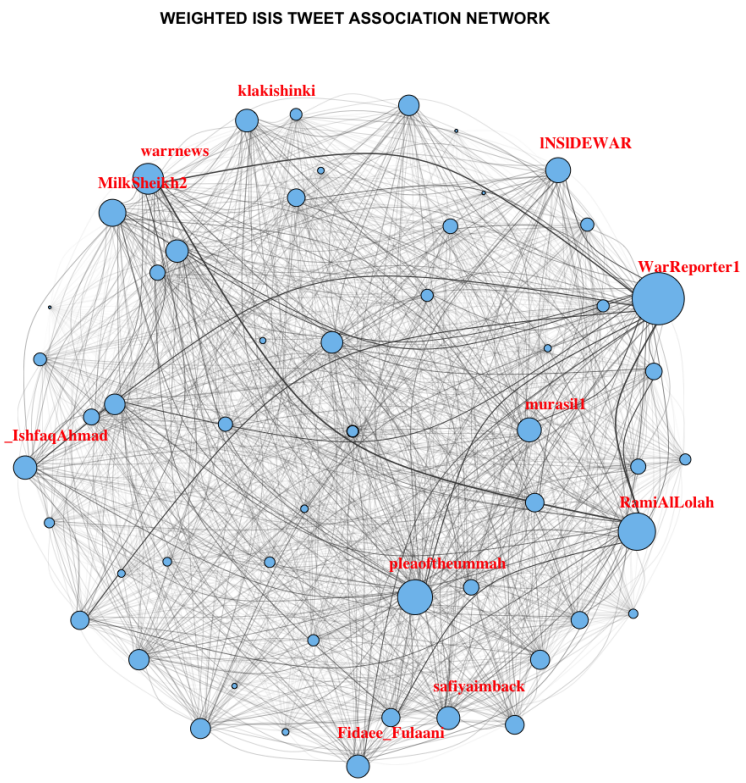


Figure 1.1: The ISIS Twitter Recruiting Network.

## 1.2 Network Science

Network Science is the study of systems using their network structure or topology. A network's structure can be described by a bunch of nodes and edges between pairs of nodes that represent a connection of some sort. In the case of the ISIS twitter recruiting data set, the recruiters themselves are represented by nodes and the social interactions between the users are represented as edges.

A directed network is a network that contains only edges that can only run in one direction. We build an undirected network, which is a network that contains edges that point in both directions which represents a simple connection between nodes[8]. An undirected network is utilized because much more research has been produced in community detection algorithms in undirected networks.

A network can be represented as a square matrix. A matrix that corresponds to a given network is called it's adjacency matrix. The rows and columns are represented by the nodes in the network, with a 1 or 0 in position $(v_i, v_j)$. If $v_i$ is adjacent to $v_j$ than position $(v_i, v_j)$ is represented by a 1. If $v_i$ is not adjacent to $v_j$ than position $(v_i, v_j)$ is represented by a 0. We take an undirected network, and the resulting adjacency matrix is symmetric. A weighted network can be represented in an adjacency matrix by assigning values that correspond to the weights of the connections [10].

There are a few key terms to understand that are used to describe characteristics of networks. The **degree** of a node in a network is the number of edges connected to it. The **path** is any sequence of nodes such that every consecutive pair of nodes in the sequence is connected by an edge in the network. A path's **length** is described by the number of edges traversed along the path. The **geodesic path** is the shortest path between any two nodes. The **centrality** of a node is a measurement of the node's importance in the network. The most simple way of measuring centrality is the degree. The higher the degree, the more central the node is to the network. But as networks grow larger and have higher complexity, it's easy to see that degree is not always the best way to define centrality. A more complex measurement of centrality is

**closeness centrality**. Closeness is a measure of the degree to which an individual is near all other individuals in a network. It is the inverse of the sum of the shortest distances between each node and every other node in the network. Another method, called **betweenness centrality**, is a measure of the extent to which a node is connected to other nodes that are not connected to each other. Its essentially a measure of the degree to which a node serves as a bridge [10].

Numerous methods have been developed over the years in order to analyze networks based off of their structure. The development of online social networks like Facebook and Twitter led to a heightened interest in grouping network nodes into communities. Communities are found in networks that have tightly knit groups of nodes with many edges between them with looser connections outside of the communities. Community structure stemmed from the concepts of network clustering. Clustering is based off of a concept known as network transitivity, which is a postulate which states that two nodes that are both neighbors of the same third node have a heightened probability of also being neighbors of one another.

As research has progressed, three major categories of community detection algorithms have emerged: hierarchical, optimization, and others. We implement Clique Percolation due to it's inherent ability to detect overlapping communities. Clique Percolation is considered a member of the "other" category in the greater score of community detection algorithms. Clique Percolation utilizes network structure and probability concepts to group nodes in a graph into communities.

## 1.3 Clique Percolation

Clique Percolation is an effective algorithm for detecting overlapping communities in large graphs. Before the creation of the Clique Percolation clustering algorithm, most techniques used to find communities in large networks required the division of networks into smaller connected clusters by the removal of key edges which connect dense sub-graphs. This process is not effective for discovering overlapping networks because the removed edges could be key in detecting a different community later in the process.

The following definitions are important for understanding the clique percolation algorithm. **Cliques** are fully connected sub-graphs of k vertices. **K-clique adjacency** means two k-cliques are adjacent if they share k1 vertices. A **k-clique chain** is a sub-graph which is the union of a sequence of adjacent k-cliques. Two k-cliques are **k-clique-connected** if they are parts of a k-clique chain. A **k-clique percolation cluster** (or component): is a maximal k-clique-connected sub-graph, meaning it is the union of all k-cliques that are k-clique-connected to a particular k-clique.

The algorithm finds k-cliques, which correspond to fully connected sub-graphs of k nodes. It understands a community as the maximal union of k-cliques that can be reached from each other through a series of adjacent k-cliques. First, all of the existent maximal k-clique percolation clusters for the given k are discovered. The k-clique percolation cluster is a maximal k-clique-connected sub-graph. This is understood as the union of all k-cliques that are k-clique-connected to a particular k-clique. The percolation transition takes place when the probability of two vertices being connected by an edge reaches the threshold $p_c(k) = [(k-1)N]^{-1/(k-1)}$ It is proven in [3] that the success in overlapping community detection with clique percolation on randomized networks translates to success on real networks. This is because only small clusters would be expected for any k at which the networks is below the transition point, but large clusters do appear, they must correspond to locally dense structures.

## 1.4 TF and TF-IDF

While analyzing the network structure is an effective approach to discover communities in the ISIS twitter recruiting data, text clustering is another logical approach. Treating all tweets by a given user as a personal document, TF and TF-IDF are useful methods to vectorize the tweets associated with each recruiter.

Term frequency (TF) is a simple way of vectorizing text where all words in the corpus are featured in a vector for each document, and the frequency of each term is reflected in the number representing the corresponding word. The problem with TF is that there are many

words that may be used many times but are not helpful in clustering. TF-IDF is usually favored over TF for this reason.

TF-IDF is the term frequency-inverse document frequency. The result of calculating the TF-IDF for each word in a document is a vector of weights, with the importance of a term in its contextual document corpus represented by the weight. First, the term frequency is calculated as normalized frequency, which is the ratio of the number of occurrences of a word in its document to the total number of words in its document. To calculate the inverse document frequency, the logarithm of the ratio of the number of documents in the corpus to the number of documents containing the given term is calculated. This inversion assigns a higher weight to terms that are rare. Multiplying the TF and IDF gives the TF-IDF which values terms that are frequent to a particular user, but rare in the corpus. The TF-IDF weighting scheme assigns to term $t$ a weight in document $d$ given by $tf - idf_{t,d} = tf_{t,d} \times idf_t$ With the standard vector space model, a set of documents $S$ can be expressed as a $m \times n$ matrix $V$, where m is the number of terms in the dictionary and n is the number of documents in $S$. Each column $V_j$ of $V$ is an encoding of a document in S and each entry $V_{ij}$ of vector $V_j$ is the significance of term i with respect to the semantics of $V_j$, where i ranges across the terms in the dictionary [4]. With the important words for each document highlighted by TF-IDF we test multiple clustering techniques on the vectorized text data.

## 1.5 Non-negative Matrix Factorization

In Non-negative Matrix Factorization (NMF), a matrix $V$ is factorized into two $W$ and $H$ where all of the elements are non-negative. In terms of text clustering, the size of $V$ would be (number of words) by (number of documents). NMF requires an expected number to be given and this number is the number of features that it should find. The features matrix $W$ would be of size (number of words) by (number of features). The coefficients matrix would become (number of features) by (number of documents). The product of $W$ and $H$ is a matrix of the same size as the input matrix $V$ and is a fairly reasonable approximation of the input matrix $V$.

Each column in the resulting matrix from $WH$ is a linear combination of the column vectors in the features matrix $W$ with coefficients supplied by the coefficients matrix $H$. NMF for text clustering could be summed up as an extraction of latent features from high-dimensional data.

NMF's power is in it's ability to look at complex data and find hidden patterns. It finds a decomposition of samples X into two matrices $W$ and $H$ of non-negative elements, by optimizing for the squared Frobenius norm (we choose Frobenius norm because it is available in open source): $\arg\min_{W,H} \frac{1}{2}||X - WH|| = \frac{1}{2}\sum_{i,j}(X_{ij} - WH_{ij})^2$ [12].

It has been observed by many that NMF can produce a parts-based representation of the data set, resulting in interpretable models.

## 1.6 Set Theory

In order to utilize both the network structure and text associated with the nodes, we take concepts from set theory to combine the results of two clustered data types. These definitions are useful for understanding the steps taken in the process of algorithm combination. The **union** of A and B is the set of all objects that are a member of A, or B, or both. The **intersection** of the sets A and B is the set of all objects that are members of both A and B. The **set difference** of U and A is the set of all members of U that are not members of A. The **symmetric difference** of sets A and B is the set of all objects that are a member of exactly one of A and B (elements which are in one of the sets, but not in both). These definitions will help in our set matching for both combining the results of the two algorithms and for comparing to ground truth communities.

## 1.7 Choosing K

Choosing the amount of clusters to search for is a challenging problem faced by anyone dealing with clustering. K is the number representative of whatever is being searched for. In clique

percolation, k represents the size of the cliques that the algorithm discovers for. In NMF, k is the the number of features to be found.

For Cliaue Percolation, we set k = 3 on the test data and the ground truth data. In our ground truth data set, this produces a communities of sizes that make sense for the size of the data set. With k<3, there are too many small communities. With k>3, there are too few communities. Because the ISIS data set is of similar size and structure, we assume that 3 also produces communities with sizes that make logical sense.

For NMF, we utilize the NMF silhouette method to determine the optimal number of clusters. The silhouette score for a user gives us a measure of how closely matched it is to data within it's cluster and how loosely it is matched to data of the neighbouring cluster. We choose the number of features in NMF which maximize the average silhouette score of the entire network. The following equations represent how the silhouette score for a single user s(i) is calculated.

$$s(i) = \begin{cases} 1 - a(i)/b(i), & \text{if a(i)} < \text{b(i)} \\ 0, & \text{if a(i)} = \text{b(i)} \\ b(i)/a(i) - 1, & \text{if a(i)} > \text{b(i)} \end{cases}$$

$$s(i) = \frac{b(i) - a(i)}{max(a(i),\ b(i))}$$

In general, a very large average silhouette width can be taken to mean that the clustering algorithm has discovered a very strong clustering structure. [11] However, if the data set contains multiple outliers, non-outliers may appear to be tighter than they actually are. Visualizing the data and removing clear outliers would be an effective step to aid the silhouette scoring method.

## 1.8 Latent Dirichlet Allocation

To provide information on the discovered communities, we utilize Latent Dirchlet Allocation (LDA). LDA can automatically discover topics in a corpus of documents by learning the topic representation of each document and the words associated to each topic. We utilize LDA in an unconventional way by using it only for it's ability to find words associated with 1 topic. We know that our algorithm outperforms LDA in clustering the users, and we use LDA only to find words associated with the clusters that have already been discovered. By setting K equal to 1 and only searching through the users for one community, we find associated words for each community.

The idea is that documents are represented as random mixtures over latent topics, where each topic is characterized by a distribution over words. Figure 1.2 represents LDA as a probabilistic generative model and emphasizes the three levels of LDA. $\alpha$ and $\beta$ are sampled once while generating the corpus. They are considered corpus-level parameters which feed the document-level variables $\theta$. For each word, $z_{dn}$ and $w_{dn}$ sample once in the end of the process [1]. This bag of words approach returns a list of words that are representative of each topic. We utilize the list of representative words to attempt to understand each detected community.

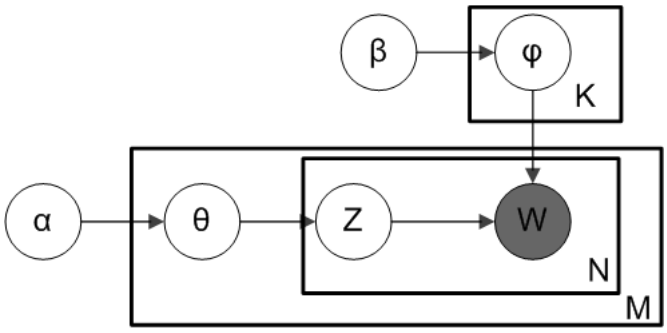


Figure 1.2: Probabilistic Model of LDA [1].

## 1.9 The Big-Clam

The BigClam algorithm utilizes the underlying adjacency matrix of the given graph to maximize the likelihood of a new matrix $F$ which represents how strongly associated each node is with each community. BigClam formulates community detection as a variant of non-negative matrix factorization. Similar to NMF, BigClam aims to learn factors that can recover the adjacency matrix of a given network. Bloch coordinate gradient descent is used to solve the convex optimization problem of updating each $F_u$ in the matrix $F$ [6]. The original Big-Clam algorithm utilizes shortcuts to lessen it's computational cost, we created a python implementation without shortcuts because our data sets have edge lists with lengths in the thousands rather than the millions. The BigClam algorithm is unable to accurately detect communities in our ground truth data set. It's impossible to point out exactly why the algorithm could not perform on our ground truth network. We do know that the authors construct their model under the following assumption: "On average 95 percent of all communities overlap with at least one other community and only 15 percent of communitys members belong to only that community. We thus examine the structure of community overlaps by measuring the probability of a pair of nodes being connected given that they belong to k common communities. [6] " In our ground truth community, far less than 95 percent of communities overlap with at least one community. This may have been a reason for why the model misclassified most of the nodes.

## 1.10 K-Means Textual Clustering with PCA

Conducting K-Means clustering on high dimensional TF-IDF data requires some kind of dimensionality reduction. We use principal component analysis (PCA) to reduce the text data's dimensionality to 2 dimensions. PCA performs a linear mapping of the data by maximizing the variance of the data in the low-dimensional representation. The eigen-vectors on the matrix are computed and those that correspond to the largest eigenvalues can now be used to reconstruct a large fraction of the variance of the original data. The goal is to retain the important variance in the data while shrinking its dimensionality. A quick way to think about how PCA works is

by organizing data as an m by n matrix, where m is the number of measurement types and n is the number of samples. Then subtracting off the mean for each measurement type. Finally calculate the eigen-vectors of the covariance [5].

K-means works like this: Let X = ($x_1$, ,$x_n$) be n data points. We partition them into K mutually disjoint clusters. The K-means clustering objective can be written as:

$J_{kmeans} = \sum_{i=1}^{i} \min_{1<k<_k} ||x_i - f_k||^2 = \sum_{k=1}^{k} \sum_{i \in C_k} ||x_i = f_k||^2$

K-means has been shown to be successful at using it's Euclidean distance measure to efficiently cluster text documents [12]. However, NMF significantly outperforms K-means clustering in our ground truth data set. NMF is better at detecting patterns hidden patterns. The vectorized textual data requires methods like NMF which are more complex.

## 1.11 Past Research

The Stanford Network Analysis Project (SNAP) is a leader in community detection research. SNAP developed the CESNA algorithm, which is the first community detection algorithm to utilize node attributes to aid the detection process. This project is partially inspired by the CESNA algorithm because CESNA demonstrates that different kinds of data can be combined to improve community detection efficiency. CESNA capitalizes on the fact that an algorithm may fail to account for import structure in data when looking at the data independently. The authors of paper about CESNA explain that "it is important to consider both sources of information together and consider network communities as sets of nodes that are densely connected, but which also share some common attributes. Node attributes can complement the network structure, leading to more precise detection of communities; additionally, if one source of information is missing or noisy, the other can make up for it. However, considering both node attributes and network topology for community detection is also challenging, as one has to combine two very different modalities of information [7]." We take these sentiments and replace node attributes with textual data because our data sets give us tweets.

# Chapter 2

# Methods

## 2.1 Overview

Given a network with textual node attributes, our algorithm can detect sub-communities within the network. In network science "communities" are not strictly defined. In our case, we define communities as users who have similar interests or are associated with certain places. We perform clique percolation on the network, NMF clustering on the text (tweets), and use set comparison and union to combine the results of two algorithms. The following sections are sequentially ordered and contain brief explanations of the process at each step.

## 2.2 Clique Percolation

The first step in our algorithm is to run clique percolation on the given network to discover k communities. We first utilize the clique percolation algorithm on the network to discover k-cliques. We choose k=3 .

## 2.3 NMF on the Text

The text must be cleaned before clustering. We find that removing typos and strange characters before clustering produces tighter clusters. We use the following code to remove user-names from the tweets.

```
def remove_users(tweet):
    text = tweet.lower()
    removeAt = re.sub(r"@\w+", "", text)
    return(removeAt)
```

In order to discover typos and messy tweets written by bots, we use the list returned by "newCount" to find and remove them from the data set.

```
split = " ".join(data["tweet"]).split()
ns_tweets = [w for w in split if not w in stopwords.words("english")]
newCount = Counter(ns_tweets).most_common(100)
print(newCount)
```

We print "newCount" and remove all messy words. We continue this process until all unrecognizable words are removed from the text.

After cleaning the tweets, we find that the NMF silhouette R package is best for estimating the optimal number of features for NMF to find. We adjust how we deal with the results of NMF to account for overlapping communities by not just assigning nodes to the community that has the max score. We edit the NMF clustering process by allowing any nodes in the top 75 percentile of all scores in that feature.

```
tfidf_docs = Tfidf_vectorizer.fit_transform(docs)
nmf = model.fit_transform(tfidf_docs)
```

This code builds an NMF model for the vectorized TF-IDF text data and returns the transformed data. It's the transformed matrix which represents the users in the rows and the different features in the columns. For each user, if they are in the 75th percentile or above for all scores

in each column, they are considered to be a part of the cluster that the corresponding column represents.

## 2.4 Set Comparisons

With K cliques (clique percolation) and F features (NMF) the sets must be matched to attempt to improve the detection. We create an intersection score which provides a score for best matching sets. We define intersection score as:

```
def intersectscore(net,text):
    a = set(net)
    b = set(text)
    inter = set(a).intersection(b)
    maxlen = max(len(a),len(b))
    return len(inter)/maxlen
```

Any set in K with at least a .25 intersection score will be matched with a set in F. If there are multiple sets in F that create an intersection score higher than .25, the set that produces the highest score will be matched. The two sets will union to produce a new set which is appended to the list of matched sets M. After all sets from K and F are compared, we consider the final group of sets to be all sets in M (union sets) and all the sets not included in M will be considered independent sets. By performing this set matching, we are combining similar sets, under the assumption that if there are common users in sets, they are representative of the same community. Those sets that are not matched will be left on there own, as we assume that one data type presents a community that might not be visible in the other data type.

## 2.5 Topic Extraction

With LDA, we extract words which are representative of the text for each discovered community. We set the number of topics to 1, and find the words representative of the 1 topic for each

community discovered. These lists of words help gain an understanding of who the clusters of users are and what they talk about.

## 2.6 Evaluation on Ground Truth

In order to evaluate the performance of the combined algorithms, we build a set comparison method to find the closest matching sets to the ground truth communities.

```
bad0 = []
bad1 = []
scorelist = []
for com in allGt:
    comMax = []
    for group in finalcommunities:
        score = intersectscore(com,group)
        comMax.append((allGt.index(com)+1,finalcommunities.index(group)+1,score))
    another = []
    for it in comMax:
        if it[0] not in bad0 and it[1] not in bad1:
            another.append(it)
        else:
            another.append((100,100,-200))
    themax = max(another,key=lambda x:x[2])
    scorelist.append(max(another,key=lambda x:x[2]))
    bad0.append(themax[0])
    bad1.append(themax[1])
```

"scorelist" returns a list of all of the discovered communities matched to the closest ground truth communities. With this list, we can evaluate the F1 scores on each matched set.

# Chapter 3

# Results

## 3.1 The Ground Truth Data Set

In order to obtain a ground truth community we scrape the network of @patmikekelly4 via tweepy. We scrape the list of followers for each user that follows @patmikekelly4. We then create an edge list of all of the users who follow each other while taking @patmikekelly4 out of the edge list because having one central user in the network would not occur in a terrorist recruitment network. Next, we scrape the last 300 tweets that each user published. Finally, we hand label each user into a list of ground truth communities. 8 communities are formed and all have at least 3 users. The users are members of one or more of the following communities: Bard College, the state of Massachusetts, passion for music, passion for politics, plays a sport (Athletics), plays basketball, a family member of @patmikekelly4, or computer science student/professional. For example, someone who plays music at Bard would be considered part of both the Bard and the Music ground truth communities. The data set includes 143 users. Figure 3.1 gives a visual of all of the users in the ground truth network and who they are connected with.

For the network diagrams in the following sections: A **red dot** means that the user is **correctly classified** in the community. A **blue dot** means that the algorithm **incorrectly classified** the user into it's respective community. An **orange dot** means **missed classification**. Missed classification means that the user is a part of the ground truth community, but

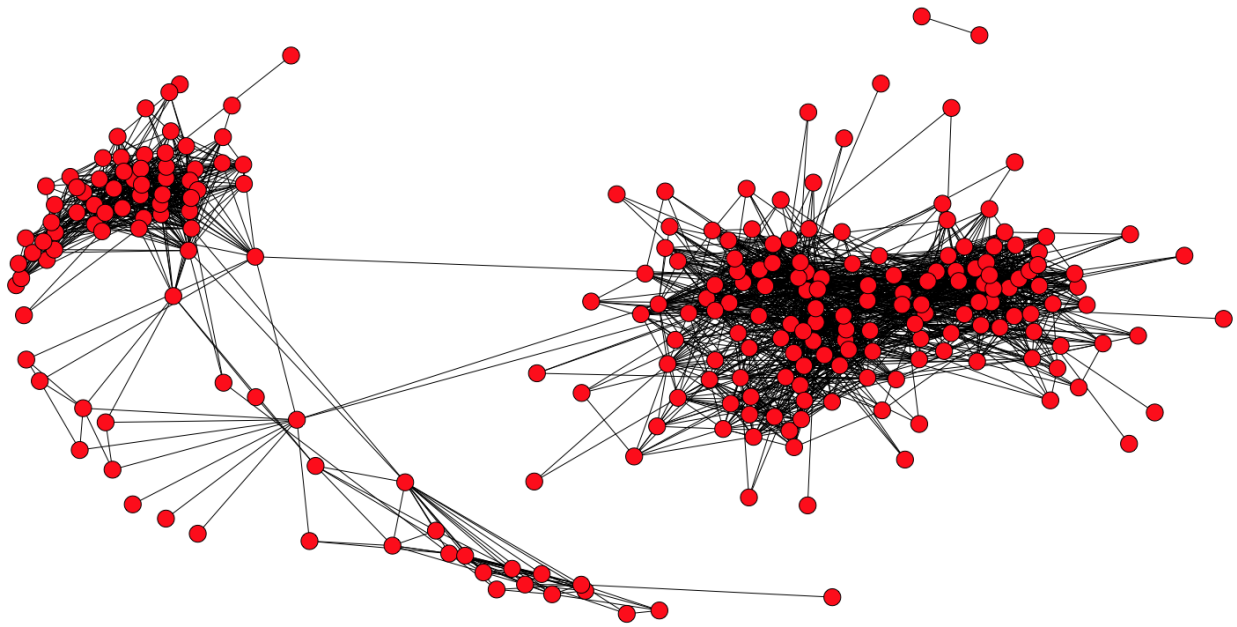

Figure 3.1: The Ground Truth Network.

not classified as a part of that community by the algorithm.

We choose F1 scores to rank the ability of the algorithm because F1 considers both precision and the recall of the test. [2] We look at each user as a binary number telling us if that user is in the community we are examining. Precision is the number of correct positive results divided by the number of all positive results. Recall is the number of of correct positive results divided by the number of positive results that should have been returned. The F1 scores creates a weighted average of precision and recall that returns between 0 and 1. In order to rank the success of each algorithm as a whole, we calculate the average F1 score on each community and multiply it by the number of detected communities divided by the total number of ground truth communities.

## 3.2 The Bard Community

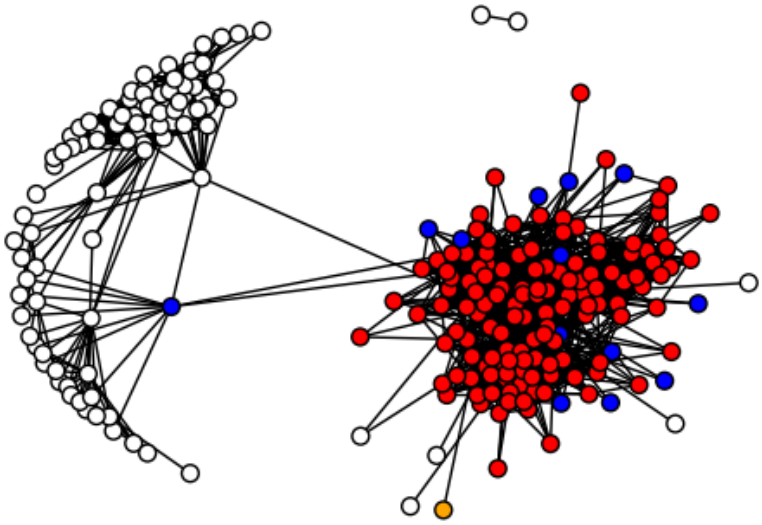

Figure 3.2: Bard Clique Percolation Results with F1 Score: 0.948905109489.

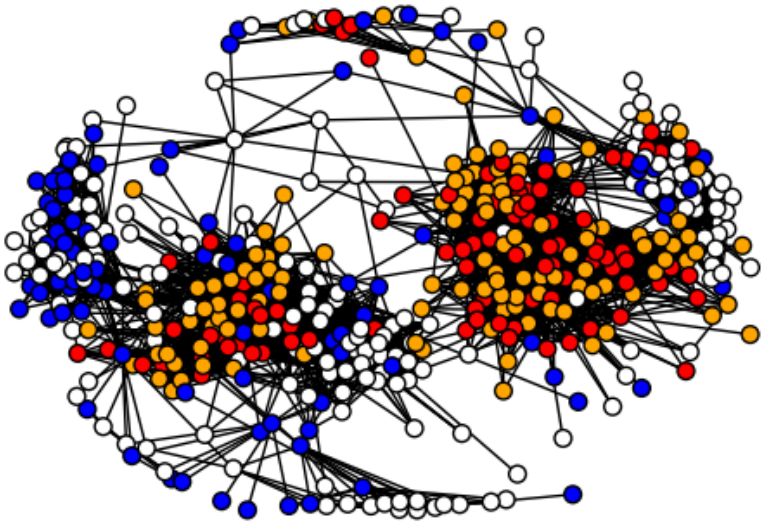

Figure 3.3: Bard NMF Results with F1 Score: 0.476987447699.

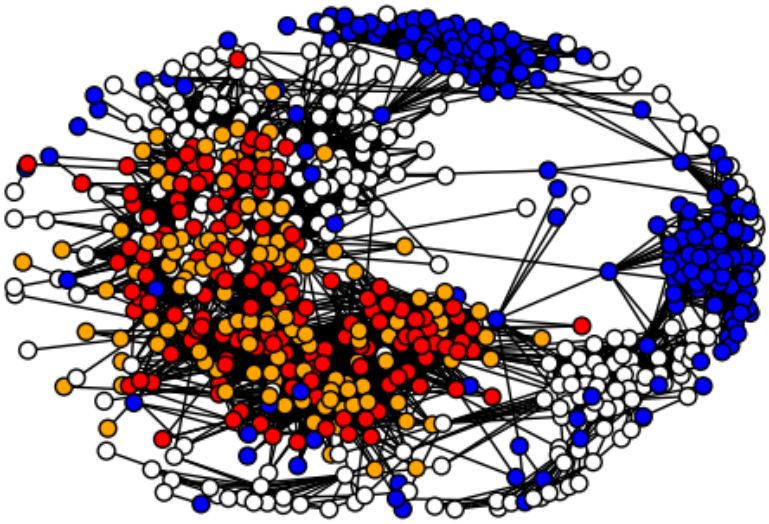

Figure 3.4: Bard Combination Results with F1 Score: 0.501766784452.

The Bard community is by far the largest community in the network. Clique percolation performs extremely well for this community because of the density of it's connections. Unfortunately, because of the diversity of interests amongst Bard students, it's NMF pairing brought it's overall F1 score down significantly. To try to improve this, future algorithms could attempt to understand that it's dealing with a larger community which overlaps other communities, and could assign more weight to the clique percolation than to the textual clustering.

The LDA extracted topics are:

[im, gonna, literally, dying, sorry, glad, way, like, girl, eating, tired, mad, crying, really, season, date, sad, going, facebook, f*cking]

## 3.3 The Massachusetts Community

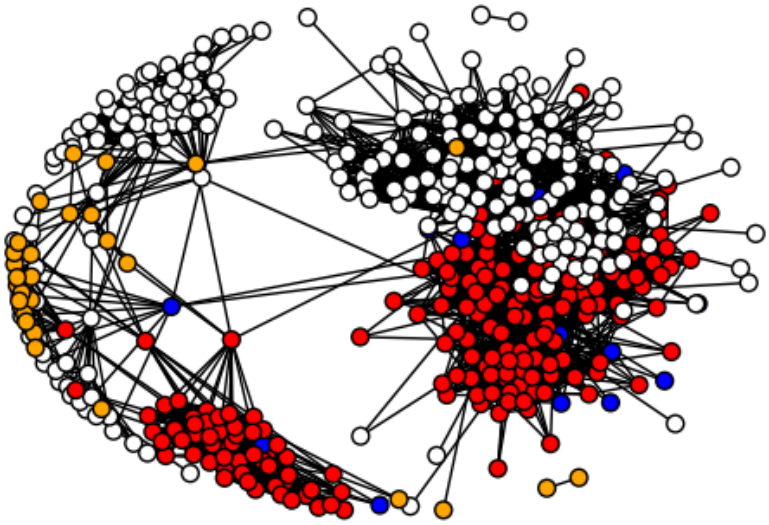

Figure 3.5: Massachusetts Clique Percolation Results with F1 Score: 0.786206896552.

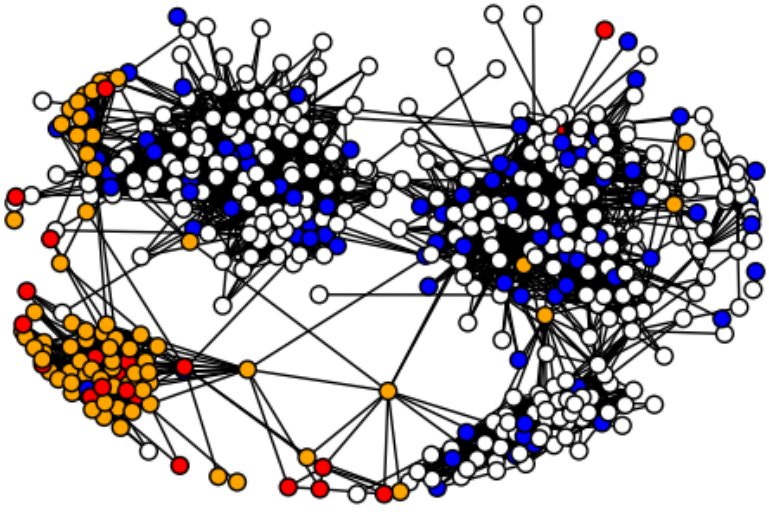

Figure 3.6: Massachusetts NMF Results with F1 Score: 0.271428571429.

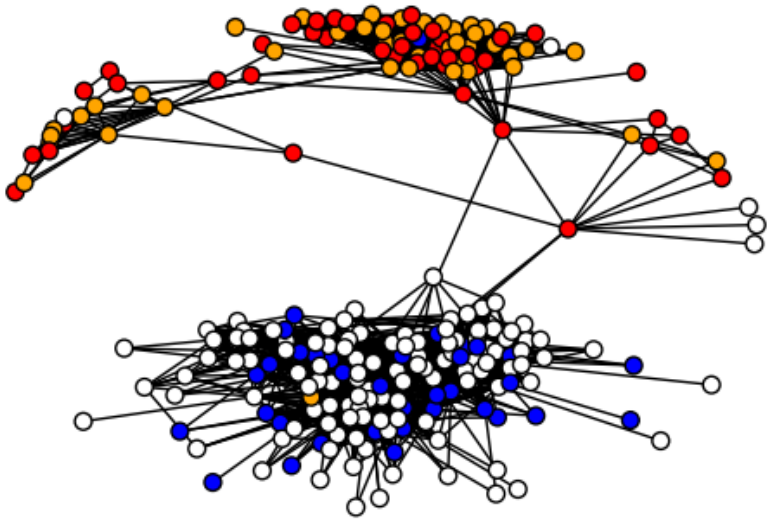

Figure 3.7: Massachusetts Combination Results with F1 Score: 0.524390243902.

The Massachusetts community is the second largest community in the ground truth data set. Massachusetts makes up almost all of the nodes that are not a part of the Bard community. Together these two communities make up a majority of the network and both contain most of the sub communities within themselves. Once again, the NMF clustering hurts the performance that clique percolation had initially achieved, but this is expected as the topics vary greatly amongst the Massachusetts network.

The LDA extracted topics are:

[thanks, connect, follow, looking, following, miss, got, voice, lmk, time, hope, need, told, fan, congrats, best, dont, libertarians, nice, florida]

## 3.4 The Athletics Community

**Athletics Clique Percolation Results**

No athletics community detected.

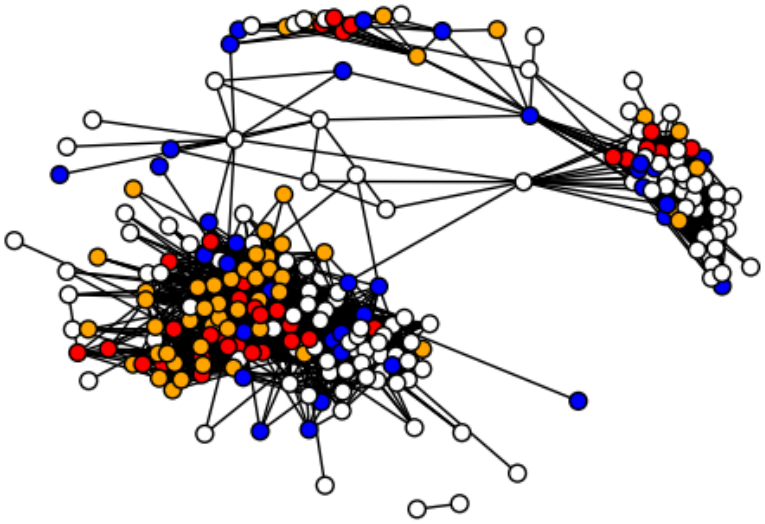

Figure 3.8: Athletics NMF Results with F1 Score: .371428571429.

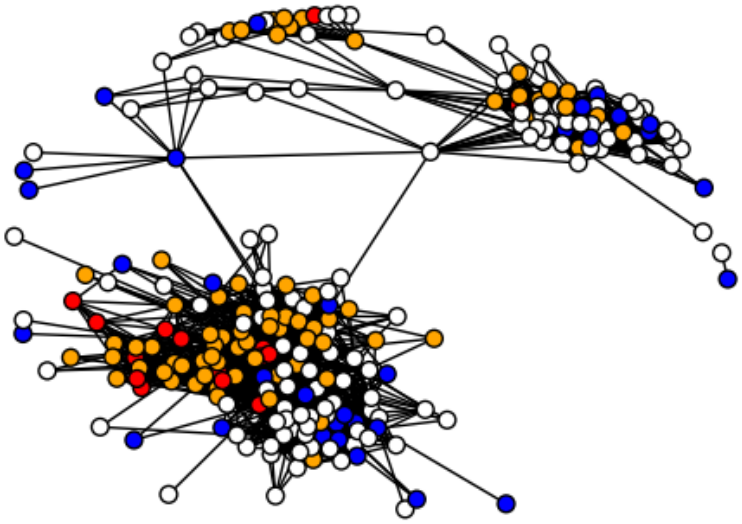

Figure 3.9: Athletics Combination Results with F1 Score: 0.225563909774.

After Bard and Massachusetts, clique percolation isn't expected to perform well because these communities have a lesser likelihood of being connected, and a greater likelihood of talking about similar subjects. Many of the sub-communities aren't directly connected. As many of the athletics community members are both a part of Bard and Massachusetts, clique percolation couldn't possibly detect their membership as they are not directly connected. NMF performs well in finding athletes referring to similar topics.

The LDA extracted topics are:

[love, people, thank, ty, life, today, free, advisor, probably, rarely, mom, hope, bro, youre, pic, bard, thing, obamas, ur, culture]

## 3.5 The Music Community

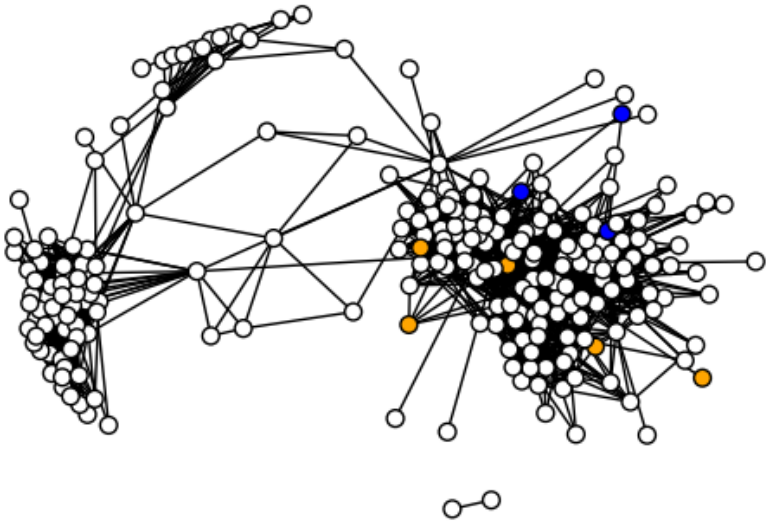

Figure 3.10: Music Clique Percolation Results with F1 Score: 0.

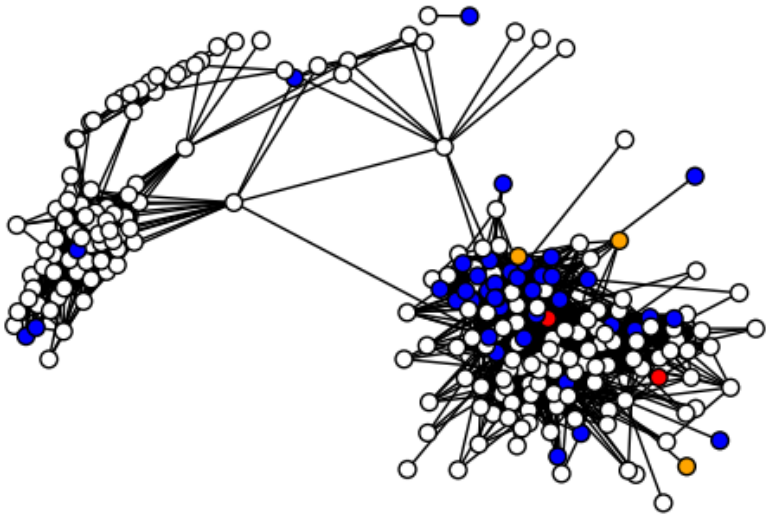

Figure 3.11: Music NMF Results with F1 Score: .29219325239.

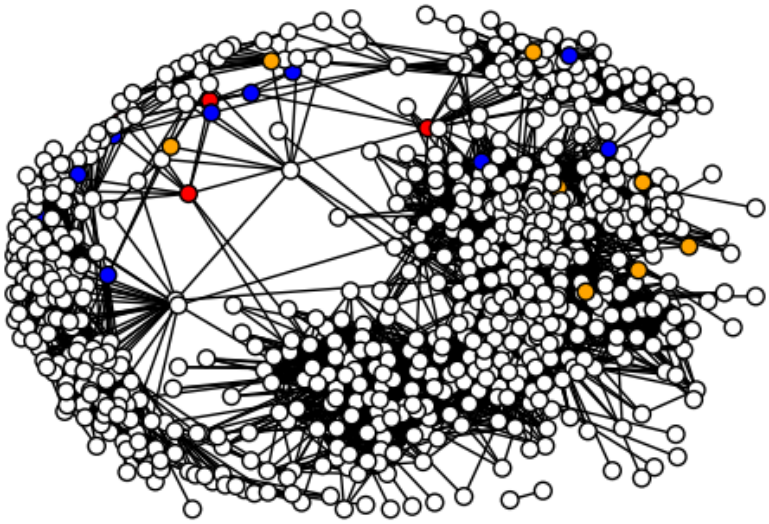

Figure 3.12: Music Combination Results with F1 Score: .201293491293.

Clique percolation is unable to detect the actual music community. NMF performs well on this community with an F1 score of .29.

LDA extracted topics are:

[home, hvac, ecolife, life, eco, protection, help, heating, comfort, water, air, energy, new, furnace, costs, save, comfortable, make, systems, sure]

## 3.6 The Family Community

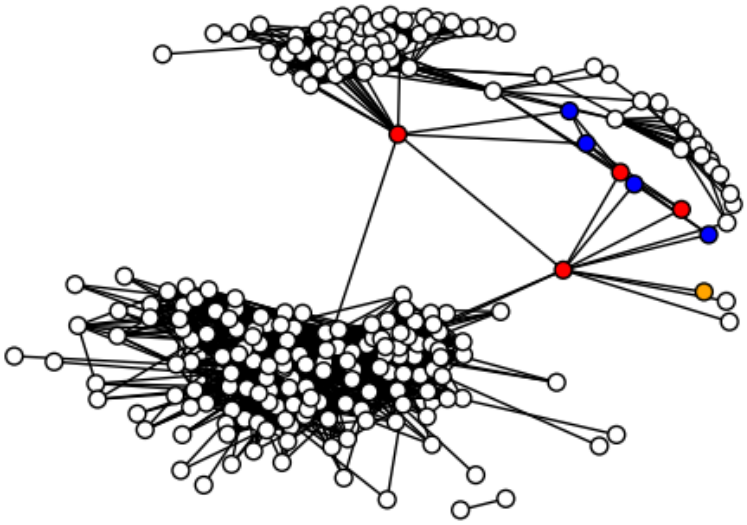

Figure 3.13: Family Cliaue Percolation Results with F1 Score: 0.615384615385.

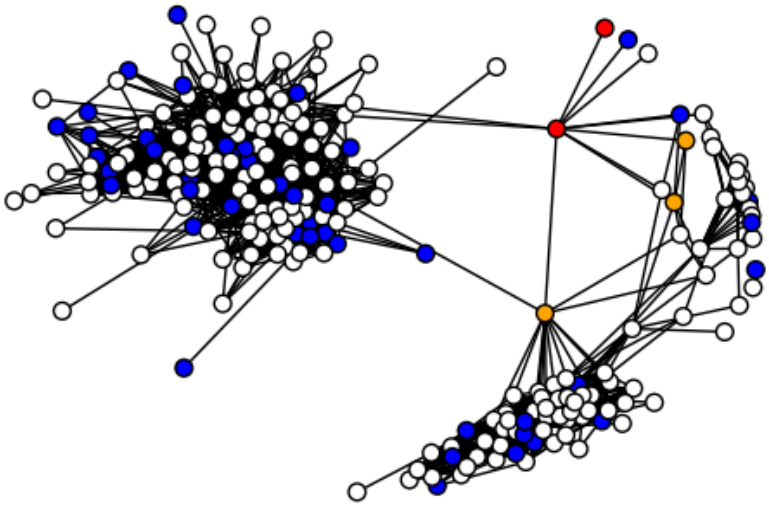

Figure 3.14: Family NMF Results with F1 Score: 0.0769230769231.

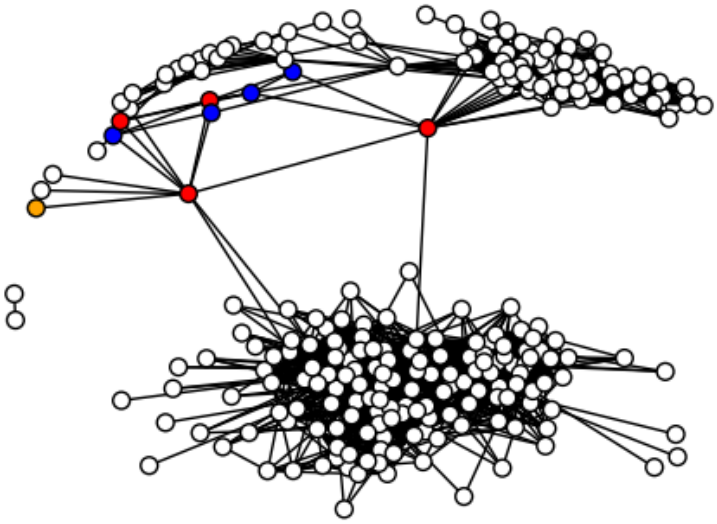

Figure 3.15: Family Combination Results with F1 Score: 0.585384615385.

Members of @patmikekelly14's family are detected at a surprisingly high rate. This tiny community is detected very well by clique percolation, probably due to it's nature of being a small community which are densely connected and not connected to the large communities that make up a majority of the network. NMF achieved minor success.

LDA extracted topics are:

[just, like, think, man, dude, did, going, got, really, lol, idea, boss, fight, roommate, years, mother, past, team, dont, stop]

## 3.7 The Politics Community

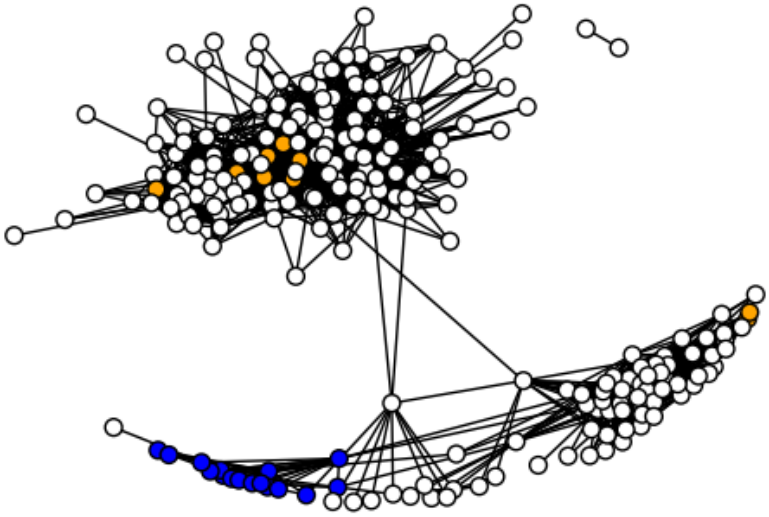

Figure 3.16: Politics Clique Percolation Results with F1 Score: 0.

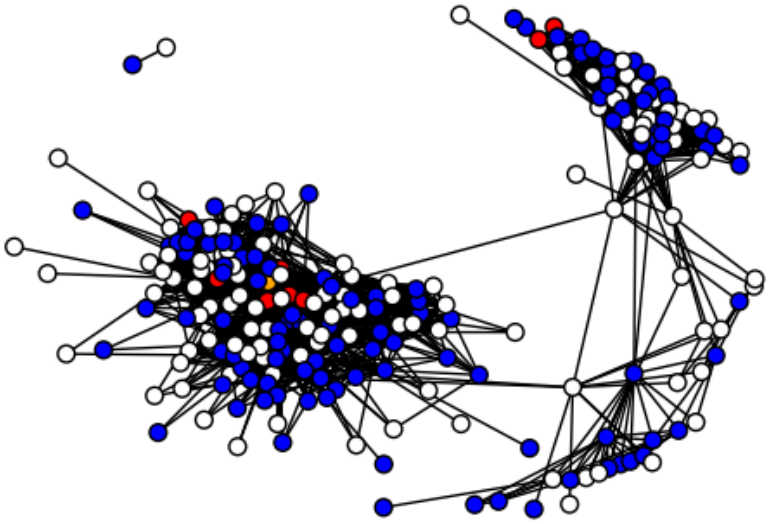

Figure 3.17: Politics NMF Results with F1 Score: 0.124031007752.

**Combination Results**

No politics community detected.

With all of the political dialogue on Twitter in the past year cluttering the text data, the true passionate political users are unable to be accurately detected by NMF.

## 3.8 The Basketball Community

**Clique Percolation Results**

No basketball community detected.

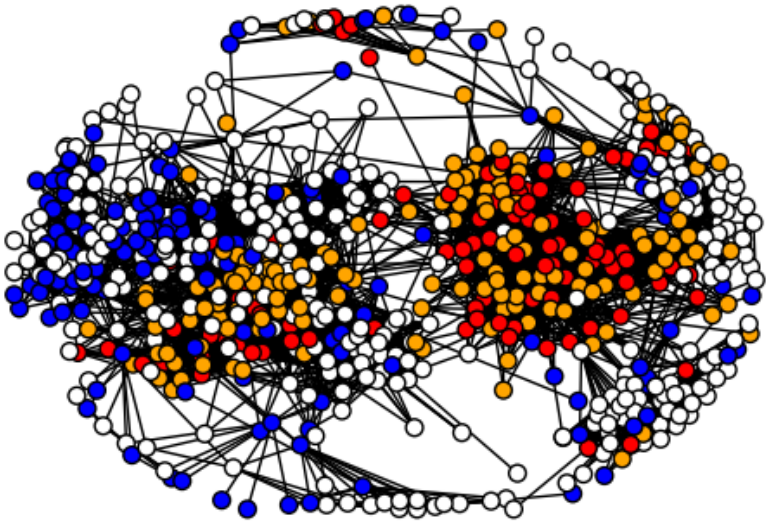

Figure 3.18: Basketball NMF Results with F1 Score: 0.410109209349.

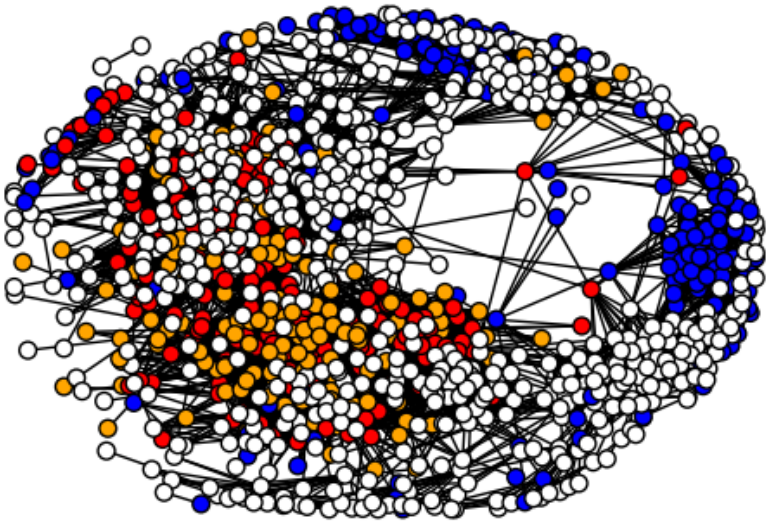

Figure 3.19: Basketball Combination Results with F1 Score: 0.310344827586.

The basketball community is detected well by the NMF textual clustering, but is not detected by clique percolation. With an overall F1 score of .31, this sub-community is detected fairly well.

LDA extracted topics are:

[day, great, happy, best, photo, big, hours, good, february, finsup, makes, perfect, senior, valentines, going, weekend, new, presidentsday, tomorrow, youre]

## 3.9 The Computer Science Community

### Clique Percolation Results

No computer science community detected.

### NMF Results

No computer science community detected.

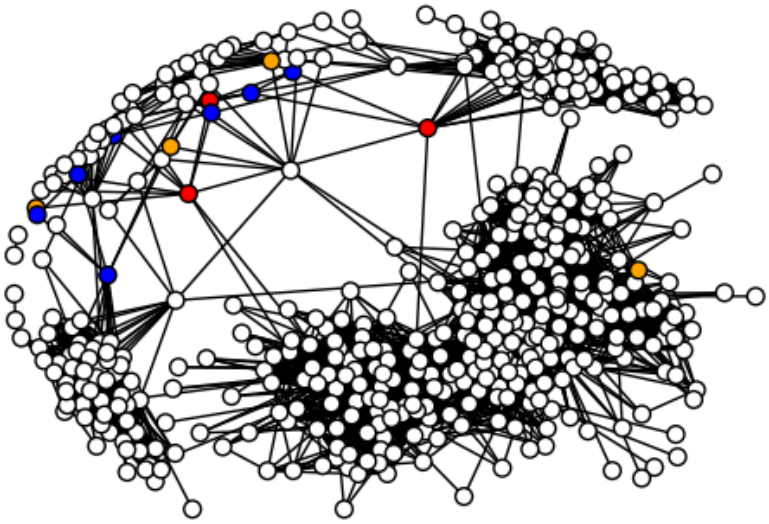

Figure 3.20: Computer Science Combination Results with F1 Score: 0.310344827586.

The computer science community results are the most intriguing, because while the algorithm does not recognize the community at first, the union of it's closest matching results create a fairly accurate final community. This is possible due to the nature of the matching process in this algorithm. When deciding which sets to union, these two sets could have been the mismatched in clique percolation or in NMF, but when combined they match closest to the computer science community.

LDA extracted topics are:

[data, machinelearning, science, big, machine, learning, python, mining, miningtechniques, bruce, analytics, bigdata, hadoop, datascience, ai, know, power, datamining, scientist, better]

## 3.10 Failure of BigClam and K-means

To illustrate the difficulty that our community detection problem presents by our definition of community, we share the results of the BigClam (Network Structure) and K-Means (Textual Clustering). The BigClam algorithm failed to detect any communities with an F1 score of higher than .19. Visually, the easiest community to detect would be the Bard community, as it

is the biggest and most densely connected community. Since the bigclam attempts to maximize the likelihood of the matrix F using sets of neighbors and non-neighbors to understand the network, we expect the Bard community to be an easy find for the BigClam. The algorithm produces almost identical sets of users in the Snap Stanford C Library version and our python implementation. Figure 3.21 displays the results of the python implementation for the Bard community.

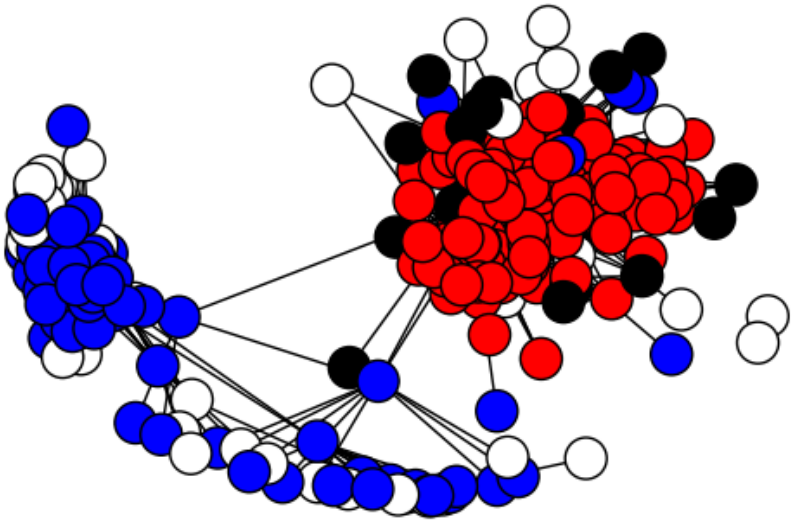

Figure 3.21: BigClam Bard Community Results with F1 Score: .461929432.

We hoped to build a mathematical model on top of the BigClam because not only does it offer overlapping community detection at scale, but it also returns a matrix $F$ which gives a metric representing confidence in the grouping of each user. $F_u$ in the resulting matrix offers a non-negative scoring of how well the user fits into each community where each community is a different column in the matrix $F$.

Dimensionality reduction and k-means clustering also failed to utilize the tweets to group users into something close to the ground truth communities. There are two possibilities for why k-means failed to recognize clusters of users given the tweets. The data could have been non-spherical. The nature of K-means utilizing euclidean distance fails to find non-spherical clusters, whereas other algorithms can detect patterns in the data and can find non-spherical clusters.

K-means also struggles with clusters of differing sizes. As K-means attempts to minimize the within-cluster sum of squares, larger clusters are split because they are given more weight and it ends up splitting the cluster. K-means and PCA produce an extremely low average F1 score on our ground truth data set.

## 3.11 Average F1 Scores by Algorithm

An average F1 score is calculated by multiplying the average F1 score of the detected communities and multiplying that by the number of communities detected by 8, the total number of communities.

| **Algorithm** | Clique Percolation | NMF Text Clustering | Combined Algorithms |
|---|---|---|---|
| **Equation** | 0.39915684431167 * 6/8 | 0.24766934442273 * 6/8 | 0.37043279809681 * 7/8 |
| **Score** | 0.2993676332337525 | 0.1857520083170475 | 0.32412869833470875 |

## 3.12 Results on ISIS Data set

For the ISIS recruiting data set there ended up being 8 Communities formed by 4 clique percolation communities and 7 NMF communities. After set comparison, we end up with 8 communities detected. Because we have no knowledge of the true communities, presenting the list of users grouped by the algorithm is unnecessary. Instead we display the terms extracted from these communities by LDA:

**Topics in Community 1:** [http, english, new, sheikh, al, video, statement, wilayat, mte2, jn, ha, coming, soon, regarding, time, media, el, town, wilayataljazirah, people]
**Topics in Community 2:** [isis, killed, iraq, army, ramadi, soldiers, iraqi, attack, today, west, deirezzor, near, claims, breakingnews, forces, fighters, militants, islamicstate, breaking, huge]
**Topics in Community 3:** [allah, protect, ya, azzawajal, muslims, brothers, swt, akbar, make, help, people, said, family, accept, freemuslimprisoners, messenger, love, dua, reward, abu]
**Topics in Community 4:** [syria, russia, assad, geneva, usa, talks, regime, war, airstrikes,

russian, deirezzor, opposition, people, kids, idlib, madaya, homs, children, assads, kerry]

**Topics in Community 5:** [know, did, dont, al, just, fact, shia, caliphate, ottomon, islam, muslims, want, used, didnt, world, created, america, ummah, sheikh, religion]

**Topics in Community 6:** [follow, brother, sisters, account, khair, support, dear, spread, true, path, jazakallah, baqiyah, shout, family, nation, great, jazakallahu, retweet, surprise, dm]

**Topics in Community 7:** [akhi, jazakallahu, ya, shout, im, nasheed, dm, sheikh, happened, know, link, inschallah, check, source, khair, better, plz, akh, punishment, surprise]

**Topics in Community 8:** [state, islamic, fighters, israel, military, says, territory, group, attacks, time, turkish, soldiers, terrorist, enter, explosive, sinai, wilayatsalahuddin, iraq, general, spy]

With closeness centrality[10], we discover the central figures for each ISIS recruiting community. The central figures for each ISIS community: Community 1: @WarReporter1

Community 2: @KhalidMaghrebi

Community 3: @UncleSamCoco

Community 4: @GunsandCoffee70

Community 5: @RamiAlLolah

Community 6: @KhalidMaghrebi

Community 7: @Nidalgazaui

Community 8: @ismailmahsud

# Chapter 4

# Conclusion

## 4.1 Summary of Thesis Achievements

Our combination of algorithms was able to detect sub-communities in a network with a score of 0.32412869833470875 which is strong considering the difficulty of the problem. The average score is slightly stronger than both the clique percolation and NMF textual clustering, which shows that our algorithm is able to effectively combine both forms of the given data to aid in the community detection problem.

We also demonstrate the difficulty of our problem by displaying the failure of BigClam and K-Means textual on the ground truth data set. Both K-Means and BigClam have proven to be successful when applied to simpler data on large networks.

## 4.2 Applications

This algorithm can be applied to detect communities in a network in which the users have textual attributes such as tweets. When applied to the ISIS twitter network, we are able to group them into sub-communities, summarize what each community is discussing, and learn about who the users are. Marketing agencies may have an interest in this algorithm to improve

their ability to target customers in particular communities given limited data.

## 4.3 Future Work

With a larger ground truth data set in a larger network, a more thorough evaluation can occur on how to best deal with the problem of community detection in networks with nodes with textual attributes. Given that there are an immense amount of algorithms that could deal with a problem of this nature, more combinations of network clustering and text clustering algorithms could be applied using our set matching process.

Further, the model could be enhanced by extending the algorithms to handle the dynamics in the networks and to detect the evolutions of communities along with the time. Understanding the shaping of a network can allow an algorithm to better understand the network's communities. Several studies have been devoted to such extensions without much success, but with the added textual data, better results may appear.

Finally and most importantly, building a model that allows the network analysis and text analysis to occur simultaneously would most likely produce stronger results than what we see with our set matching. We could not build a dynamic model due to the failure of the Bigclam on our data set. Our thesis illustrates that putting time into building a single model that would combine network science and text analysis to detect communities in networks would likely be worthwhile and produce strong results.

# Bibliography


[1] Ng A., Blei D., and Jordan M. "Latent Dirichlet Allocation". In: *Journal of Machine Learning Research* (2003).

[2] Orman G. et al. "On Accuracy of Community Structure Discovery Algorithms". In: *Journal of Convergence Information Technology* (2011).

[3] Nyi I.D., Palla G., and Vicseki T. "Clique Percolation in Random Networks". In: *Physical Review Letters* (2005).

[4] Ramos J. "Using TF-IDF to Determine Word Relevance in Document Queries". In: *Proceedings of the first instructional conference on machine learning.* (2003).

[5] Shlens J. "A Tutorial on Principal Component Analysis". In: *Google Research* (2014).

[6] Yang J. and Leskovec J. "Overlapping Community Detection at Scale: A Nonnegative Matrix Factorization Approach". In: *Web Search and Data Mining* (2013), pp. 1–9.

[7] Yang J., Mcauley J., and Leskovec J. "Community Detection in Networks with Node Attributes". In: *Stanford University* (2015), pp. 4–5.

[8] Girvan M. and Newman M.E.J. "Community Structure in Social and Biological Networks". In: *Physics Abstract Service* (2002), pp. 1–5.

[9] Kay M. *Generating a network graph of Twitter followers using python and networkx.* 2014. URL: `http://mark-kay.net/2014/08/15/network-graph-of-twitter-followers/` (visited on 12/30/2016).

[10] Newman M. *Networks: An Introduction.* Oxford University Press, Inc., 2010.

[11] Rousseeuw P. “Silhouettes: a graphical aid to the interpretation and validation of cluster analysis”. In: *Journal of Computational and Applied Mathematics* (1986).

[12] Li T. and Ding C. “Non-negative Matrix Factorizations for Clustering: A Survey”. In: *Semantics Scholar* (2013).

# Appendix A

# The BigClam Python Implementation

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import community
import math
import itertools as IT
import scipy
#A cut, vertex cut, or separating set of a connected graph G
#is a set of vertices whose removal renders G disconnected.
def cut_size(G, S, T=None, weight=None):
    edges = nx.edge_boundary(G, S, T)
    return sum(weight for u, weight in edges)
#the degree of each node in the graph
def volume(G, S, weight=None):
    degree = G.degree
    dictVol = degree(S,weight=weight)
    summ = []
    for key, value in dictVol.items():
        summ.append(value)
    return sum(summ)
#number of cut edges / min(volume of the set, volume of other set)
def conductance(G, S, T=None, weight=None):
    if T is None:
        T = set(G) - set(S)
    num_cut_edges = cut_size(G, S, T, weight=weight)
    volume_S = volume(G, S, weight=weight)
    volume_T = volume(G, T, weight=weight)
    return num_cut_edges / min(volume_S, volume_T)
#k = INIT method = bigclaminit.py
k = 7
F = np.zeros((len(ls),k))
c= {}
#initialize the matrix F
def graphConductance(G):
    for w in set(G):
        newSet = set(G.neighbors(w))
        c[w] = (conductance(G,newSet))
        nx.set_node_attributes(G, 'conductance', c)
    localGraph = G
```

```
    for i in range(1,k):
        com = nx.get_node_attributes(G,'conductance')
        comNode = min(com, key=com.get)
        community = G.neighbors(comNode)+[comNode]
        G = set(G)-set(community)
        for x in range(len(localGraph.nodes())):
            if x in community:
                F[x-1][k-2] = 1
            if x in G:
                F[x-1][k-1] = 1
    return(F)
#iterating through the edges of Graph.
#Assign 1 node as u and 1 as v.
F = graphConductance(Graph)
nodes = Graph.nodes()
#orthogonal dot product try and finally
def catchInf(exponential):
    e = np.log(1-exponential)
    if e == float('-inf'):
        return -10
    else:
        return e
#loops through the edges of the graph
#the Likelihood of the entire matrxi F
#the scalar given here is our overall
#likelihood for each loop iteration through the entire matrix.
#this loop will run until the likelihood stays the same
lF = []
def likelihoodOfF(Graph):
    for node in Graph.edges():
        # u  = a node from the edge
        u = node[0]
        # v = another node from the edge
        v = node[1]
        #grabs the Fu Row from connection strength matrix F
        Fu = F[u-1,:]
        #same thing for Fv
        Fv = F[v-1,:]
        FvT = np.transpose(Fv)
        # dot product of Fu,FvT
        mFuFvT = np.dot(Fu,FvT)
        #exponential of Fu.Fv
        expVal = math.exp(-mFuFvT)
        fVal = catchInf(expVal)
        #fVal = (np.log(1-expVal))
        lF.append(fVal)
    sLF = sum(lF)
#now the compliment set
    nLF = []
    nonEdges = list(nx.non_edges(Graph))
    for node in nonEdges:
        nU = node[0]
        nV = node[1]
        nFu = F[nU-1,:]
        nFv = F[nV-1,:]
        nFvT = np.transpose(nFv)
        nmFuFvT = np.dot(nFu,nFvT)
        nLF.append(nmFuFvT)
    snLF = sum(nLF)
    like = sLF - snLF
```

```
    return like
#loops through each node,
#and then the neighbors for each node
def likelihoodOfFu(Graph,node):
    neighLike = []
    nonNeighLike = []
    for neighbor in Graph.neighbors(node):
        u = node
        v = neighbor
        Fu = F[u-1,:]
        Fv = F[v-1,:]
        FvT = np.transpose(Fv)
        mFuFvT = np.dot(Fu,FvT)
        expVal = math.exp(-mFuFvT)
        fVal = catchInf(expVal)
        neighLike.append(fVal)
    #loop through non-neighbors of node u
    lSum = sum(neighLike)
    for non in nx.non_neighbors(Graph,node):
        u = node
        v = non
        Fu = F[u-1,:]
        Fv = F[v-1,:]
        FvT = np.transpose(Fv)
        mFuFvT = np.dot(Fu,FvT)
        nonNeighLike.append(mFuFvT)
    nLSum = sum(nonNeighLike)
    fLSum = lSum - nLSum
    return fLSum
def divExp(divNum):
    func = 1 - math.exp(divNum)
    return func
#computing the gradient
def computeGradient(Graph,node):
    neighLike = []
    nonNeighLike = []
    for neighbor in Graph.neighbors(node):
        u = node
        v = neighbor
        Fu = F[u-1,:]
        Fv = F[v-1,:]
        FvT = np.transpose(Fv)
        mFuFvT = np.dot(Fu,FvT)
        expVal = math.exp(-mFuFvT)
        divVal = math.exp(-mFuFvT)
        if divVal == 1:
            oneMinusDivVal = 1
        else:
            oneMinusDivVal = 1 - divVal
        fExpVal = expVal/oneMinusDivVal
        fvProd = np.multiply(Fv,fExpVal)
        neighLike.append(fvProd)
    sumFvProd = sum(neighLike)
    for non in nx.non_neighbors(Graph,node):
        v = non
        Fv = F[v-1,:]
        nonNeighLike.append(Fv)
    sumOfFv = sum(nonNeighLike)
    gradient = sumFvProd - sumOfFv
    return gradient
```

```
alpha = .001
maxIter = 1000
thres = .0001
curL = 0.0
i = 0

def getCommunity(rowNum):
    rowData = V[rowNum]
    com = rowData.argmax()
    return com
def backTrackingLineSearch(theGraph,alpha):
    stepSize = 1
    for i in range(maxIter):
        for node in Graph.nodes():
            deltaV = computeGradient(Graph,node)
            com = getCommunity(node-1)
            newVal = stepSize * deltaV[com]
            if newVal < minVal:
                newVal = minVal
            if newVal > maxVal:
                newVal = maxVal
            sVal = np.dot(computeGradient(Graph,node),likelihoodOfFu(Graph,node))
            if likelihoodOfFu(Graph,node) < initLikelihood + alpha * stepSize * sVal[com]:
                stepSize = stepSize * Beta
            else:
                break
        if i == maxIter-1:
            stepSize = 0.0
    return stepSize
def bigClam(Graph):
    while i < maxIter:
        i = i + 1
        #get likelihood before update
        prevL = likelihoodOfF(Graph)
        for node in Graph.nodes():
            #c = community
            #update matrix with gradient
            for c in range(0,k):
                grad = computeGradient(Graph,node)
                val = alpha * grad[c]
                update = F[node-1,c] + val
        #keep matrix between 0 and 1
                if update <= 0 :
                    F[node-1,c] = 0
                elif update >= 1:
                    F[node-1,c] = 1
                else:
                    F[node-1,c] = update
            #current likelihood
            curL = likelihoodOfF(Graph)
            if curL - prevL <= thres*abs(prevL):
                break
            else:
                prevL = curL
            alpha = backTrackingLineSearch(Graph,alpha)
        print('iteration:', i, curL)
    print(F)
```

# Appendix B

# Generating a Ground Truth Data Set From Twitter

## B.1 getFollowers.py

```
#run this file in terminal and name it getfollowers.py
#Get the key/token from OAuth and enter your own username into the
#screen name field
#Will print a list of the user IDs of your followers on twitter

import time
import tweepy
c_key = ''
c_secret = ''
a_token = ''
a_token_secret = ''
auth = tweepy.OAuthHandler(c_key, c_secret)
auth.set_access_token(a_token, a_token_secret)
api = tweepy.API(auth)
ids = []
for page in tweepy.Cursor(api.followers_ids, screen_name="patmikekelly4").pages():
    ids.extend(page)
    time.sleep(20)
print(ids)
```

## B.2 getTheTweets.py

```
#name this file: getTheTweets.py
#enter the list of user_ids that you created with
#generatenetwork.py into the list named "ids"
#this code will create a CSV containing all of the
#users tweets associated with their user_ID
```

```
import tweepy
import csv
import pandas as pd
#Put your list of user_ids in here:
ids = []
#Twitter API credentials
consumer_key = ""
consumer_secret = ""
access_key = ""
access_secret = ""
#authorize twitter, initialize tweepy
auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_key, access_secret)
api = tweepy.API(auth)
outtweets = []
def get_all_tweets(screen_name):
        #Twitter only allows access to a users most recent
        #3240 tweets with this method
        #initialize a list to hold all the tweepy Tweets
    alltweets = []

    new_tweets = api.user_timeline(screen_name = screen_name,count=200)
    for tweet in new_tweets:
        outtweets.append((tweet.id_str,tweet.text.encode("utf-8")))
    return outtweets
all_of_them = []
for user in ids:
    try:
        u = api.get_user(user)
        tweets = get_all_tweets(u.screen_name)
        all_of_them.append(tweets)
    except tweepy.TweepError as e:
        pass
with open('all_tweets.csv', 'w') as file:
    output = csv.writer(file, delimiter=',')
    output.writerows([
        ['id', 'tweet']
    ])
#after running this file you will have a list of user_ids, and a
#list of the last x number of tweets that each user composed.
```

## B.3 buildNetwork.py

This code is based off of the tutorial written by [9] and modified to fit our needs.

```
#This code will genenate JSONs for each user containing
#a list of who each user is following with up to 400 users for each.

import tweepy
import time
import os
import sys
import json
import argparse
FOLLOWING_DIR = 'following'
```

```
MAX_FRIENDS = 400
FRIENDS_OF_FRIENDS_LIMIT = 400
if not os.path.exists(FOLLOWING_DIR):
    os.makedir(FOLLOWING_DIR)
enc = lambda x: x.encode('ascii', errors='ignore')
CONSUMER_KEY = ''
CONSUMER_SECRET = ''
ACCESS_TOKEN = ''
ACCESS_TOKEN_SECRET = ''
# == OAuth Authentication ==
# This mode of authentication is the new preferred way
# of authenticating with Twitter.
auth = tweepy.OAuthHandler(CONSUMER_KEY, CONSUMER_SECRET)
auth.set_access_token(ACCESS_TOKEN, ACCESS_TOKEN_SECRET)
api = tweepy.API(auth)
def get_follower_ids(centre, max_depth=1, current_depth=0, taboo_list=[]):
    # print 'current depth: %d, max depth: %d' % (current_depth, max_depth)
    # print 'taboo list: ', ','.join([ str(i) for i in taboo_list ])
    if current_depth == max_depth:
        print ('out of depth')
        return taboo_list
    if centre in taboo_list:
        # we've been here before
        print ('Already been here.')
        return taboo_list
    else:
        taboo_list.append(centre)
    try:
        userfname = os.path.join('twitter-users', str(centre) + '.json')
        if not os.path.exists(userfname):
            print ('Retrieving user details for twitter id %s' % str(centre))
            while True:
                try:
                    user = api.get_user(centre)
                    d = {'name': user.name,
                         'screen_name': user.screen_name,
                         'id': user.id,
                         'friends_count': user.friends_count,
                         'followers_count': user.followers_count,
                         'followers_ids': user.followers_ids()}
                    with open(userfname, 'w') as outf:
                        outf.write(json.dumps(d, indent=1))
                    user = d
                    break
                except tweepy.TweepError as error:
                    print (type(error))
                    if str(error) == 'Not authorized.':
                        print ('Can''t access user data - not authorized.')
                        return taboo_list
                    if str(error) == 'User has been suspended.':
                        print ('User suspended.')
                        return taboo_list
                    errorObj = error[0][0]
                    print (errorObj)
                    if errorObj['message'] == 'Rate limit exceeded':
                        print ('Rate limited. Sleeping for 15 minutes.')
                        time.sleep(15 * 60 + 15)
                        continue
                    return taboo_list
        else:
```

```
            user = json.loads(file(userfname).read())
        screen_name = enc(user['screen_name'])
        fname = os.path.join(FOLLOWING_DIR, screen_name + '.csv')
        friendids = []
        # only retrieve friends of Pat... screen names
        if screen_name ==('patmikekelly4'):
            if not os.path.exists(fname):
                print ('No cached data for screen name "%s"' % screen_name)
                with open(fname, 'w') as outf:
                    params = (enc(user['name']), screen_name)
                    print ('Retrieving friends for user "%s" (%s)' % params)
                    # page over friends
                    c = tweepy.Cursor(api.friends, id=user['id']).items()
                    friend_count = 0
                    while True:
                        try:
                            friend = c.next()
                            friendids.append(friend.id)
                            params = (friend.id, enc(friend.screen_name), enc(friend.name))
                            outf.write('%s\t%s\t%s\n' % params)
                            friend_count += 1
                            if friend_count >= MAX_FRIENDS:
                                print ('Reached max no. of friends for "%s".' % friend.screen_name)
                                break
                        except tweepy.TweepError:
                            # hit rate limit, sleep for 15 minutes
                            print ('Rate limited. Sleeping for 15 minutes.')
                            time.sleep(15 * 60 + 15)
                            continue
                        except StopIteration:
                            break
            else:
                friendids = [int(line.strip().split('\t')[0]) for line in file(fname)]

            print ('Found %d friends for %s' % (len(friendids), screen_name))
            # get friends of friends
            cd = current_depth
            if cd+1 < max_depth:
                for fid in friendids[:FRIENDS_OF_FRIENDS_LIMIT]:
                    taboo_list = get_follower_ids(fid, max_depth=max_depth,
                        current_depth=cd+1, taboo_list=taboo_list)
            if cd+1 < max_depth and len(friendids) > FRIENDS_OF_FRIENDS_LIMIT:
                print ('Not all friends retrieved for %s.' % screen_name)
    except Exception as error:
        print ('Error retrieving followers for user id: ', centre)
        print (error)
        if os.path.exists(fname):
            os.remove(fname)
            print ('Removed file "%s".' % fname)
        sys.exit(1)
    return taboo_list
if __name__ == '__main__':
    ap = argparse.ArgumentParser()
    ap.add_argument("-s", "--screen-name", required=True, help="Screen name of twitter user")
    ap.add_argument("-d", "--depth", required=True, type=int, help="How far to follow user network")
    args = vars(ap.parse_args())
    twitter_screenname = args['screen_name']
    depth = int(args['depth'])
    if depth < 1 or depth > 3:
        print ('Depth value %d is not valid. Valid range is 1-3.' % depth)
```

```
        sys.exit('Invalid depth argument.')
    print ('Max Depth: %d' % depth)
    matches = api.lookup_users(screen_names=[twitter_screenname])
    if len(matches) == 1:
        print (get_follower_ids(matches[0].id, max_depth=depth))
    else:
        print ('Sorry, could not find twitter user with screen name: %s' % twitter_screenname)
```

## B.4 twitterNetwork.py

This is the last step in the process. This code takes the central user out of the network, and creates an edge list in a CSV of who is following who from the user list.

```
import glob
import os
import json
import sys
from collections import defaultdict
users = defaultdict(lambda: { 'followers': 0 })
for f in glob.glob('twitter-users/*.json'):
    data = json.load(file(f))
    screen_name = data['screen_name']
    users[screen_name] = { 'followers': data['followers_count'] }
SEED = 'patmikekelly4'
def process_follower_list(screen_name, edges=[], depth=0, max_depth=2):
    f = os.path.join('following', screen_name + '.csv')
    if not os.path.exists(f):
        return edges
    followers = [line.strip().split('\t') for line in file(f)]
    for follower_data in followers:
        if len(follower_data) < 2:
            continue
        screen_name_2 = follower_data[1]
        # use the number of followers for screen_name as the weight
        weight = users[screen_name]['followers']
        edges.append([screen_name, screen_name_2, weight])
        if depth+1 < max_depth:
            process_follower_list(screen_name_2, edges, depth+1, max_depth)
    return edges
edges = process_follower_list(SEED, max_depth=3)
with open('twitter_network.csv', 'w') as outf:
    edge_exists = {}
    for edge in edges:
        key = ','.join([str(x) for x in edge])
        if not(key in edge_exists):
            outf.write('%s\t%s\t%d\n' % (edge[0], edge[1], edge[2]))
            edge_exists[key] = True
```