Detecting Communities using Absorbing Markov Chains

Ankit Kumar

CS224W: Fall 2013

I. INTRODUCTION

Community detection is an important part of network analysis. The goal is simple: to detect how nodes in the graph ought to be grouped into communities. Some algorithms wholly partition the nodes; others allow for some nodes to be considered "community-less". Community detection has several applications: for example, suggesting connections (as in, say, Facebook) and determining network structure. As we'll see in this paper, community detection can also be quite useful to derive features for classification tasks where the data includes interactions that are not easily translatable into features.

For most measures of community structure, it is very computationally complex to directly find the partition which maximizes the measure. For example, finding the partition that maximizes Modularity, a recently introduced and popular measure, is NP-Complete.[1] This suggests the need for algorithms that perhaps estimate good community structure in networks, but run in polynomial time. In this paper, I propose and evaluate one such algorithm.

II. PRIOR WORK

Perhaps the most comprehensive survey of prior work in the field is given by Fortunato[2]. In the paper, Fortunato touches on many different families of algorithms: Divisive, Modularity-based, Spectral, Dynamic, and Staistical Inference-based. Since my own algorithm falls under the Dynamic category, I'll specifically mention a couple of random-walk based algorithms that Fortunato talks about. Zhou[3] proposed three algorithms based on random walks, all of which run in $O(n^3)$. Two are based on a definition of a distance d_{ij} between two nodes i and j, where the distance is a function of the average number of edges that a random walker has to cross to reach j from i. Fortunato also mentions *walktrap* by Pons and Latapy[4], which runs in time $O(mn^2)$, where m is the number of edges. $Walktrap$ defines a distance d_{ij} between nodes i and j as a function of how many times a random walk of length t (where t is a hyperparameter) starting at i ends in j. It is not clear how the hyperparameter t should be chosen, and Pons and Latapy mention that more work needs to be done on the topic.

Newman and Girvan[5] proposed a divisive algorithm. They calculate, for each edge, a score of "betweenness", where betweenness is defined loosely as a function of how responsible that edge is for connecting two bigger sets of vertices. They then remove the edge with the highest betweenness score, recalculate the betweenness scores, and repeat. This produces a dendrogram, where the leaf nodes are the individual vertices and the communities are found by looking at how the graph was split. Because they have to recalculate betweenness scores each iteration, the algorithm has run time problems. For two of measures of betweenness for which they gave complexity, the runtimes are $O(m^2n)$ and $O((n+m)mn^2)$. I will use many of the evaluation methods that Newman and Girvan used.

III. DATA COLLECTION

To test my algorithm, I downloaded two data sets that were mentioned in Newman and Girvan's paper: a network of the 2000 college football season (where teams are nodes and connections are games played; ground-truth communities are conferences) and Zachary's Karate Club (a canonical network where ground-truth communities are known). In addition, I downloaded V. Krebs' political books dataset (where nodes are books and edges represent frequent co-purchasing of books on Amazon.com; ground-truth communities are political slants, hand-labelled by Krebs). All these data sets were found on Mark Newman's personal webpage[6].

In addition, I used a dataset from the 2001 KDD Cup competition[7]. Specifically, Tasks 2 and 3 of the competition involved a dataset that included gene-gene interactions, which can be viewed as a graph.

IV. MATHEMATICAL BACKGROUND

(Most of the notational convention here comes from wikipedia, typically with 'state' replaced with 'node')

An absorbing markov chain is a set of nodes such that there is at least one absorbing state, and it is possible to get from any node to an absorbing node in finite steps.

If an absorbing Markov chain has t transient nodes and r absorbing nodes, then the transition matrix P can be written as:

$$P = \begin{pmatrix} Q & R \\ \mathbf{0} & I_r \end{pmatrix} \tag{1}$$

Where Q is a $t \times t$ matrix that represents the probability of transitioning from a transient node to a transient node, R is a $t \times r$ matrix that represents the probability of transitioning from a transient node into an absorbing node, $\mathbf{0}$ is an $r \times t$ matrix of all 0s that represents the probablity of transitioning from an absorbing node into a transient node (hence all 0s), and I_r is an $r \times r$ identity matrix representing the probability of transitioning from an absorbing node to another absorbing node.

Absorbing markov chains have a "fundamental matrix" denoted N which has the property that N_{ij} is the expected number of visits to node j that a random walk beginning at node i will make before absorption. Note that the probability of transitioning from i to j in exactly k steps is Q^k_{ij}. So:

$$N = \sum_{k=0}^{\infty} Q^k = (I_t - Q)^{-1} \tag{2}$$

Where I_t is a $t \times t$ identity matrix.

An important question is, what is the probability that a random walk starting at node i gets absorbed in absorbing node j (this will be the basis of the algorithm). The answer is B_{ij}, where we can find B as follows:

$$B = NR \tag{3}$$

V. FORMAL DESCRIPTION OF ALGORITHM

Say we want to find r clusters in a graph – note that this graph should have no absorbing nodes. The idea is to add r "cluster nodes" $\{c_i\}$ for $i \in \{1, ..., k\}$ that are absorbing nodes and create edges to those nodes from the original nodes in the graph. We add a hyperparameter α such that at any given node there is an α chance that the walk will transition into a cluster node and a $1 - \alpha$ chance the walk will transition into another one of the original nodes. Then, we classify which cluster a node in the graph belongs to by the cluster node in which it gets absorbed in the most.

The algorithm is an expectation-maximization algorithm. We initialize the r cluster nodes in the way described later in this paper. Then the expectation step assigns each node to one of the clusters, while the maximization step re-connects the r cluster nodes to maximize the following *objective function*:

$$\frac{\sum_{i=0}^{r} P(absorbed\ in\ c_i | start\ in\ a\ node\ in\ cluster\ i)}{r}$$

i.e the average over all clusters of the probability a random walk starting at a randomly chosen node classified in a certain cluster to be absorbed in that cluster.

A. Initialization

Figuring out a good way to initialize the algorithm was a significant part of the work I did between the milestone and now. It should be noted all of the initialization techniques I describe are heuristics, and that if one has domain knowledge about the graph they are clustering, manually picking seed nodes will typically produce better results.

Randomize and Repeat: Noting that we have an objective function with which to compare clusterings (the average overall all the clusters of hte probability that a random walk starting at a node in the cluster is absorbed in that cluster), we can randomly pick K seeds, run the algorithm, and repeat, and after a set number of runs take the run with the highest objective value.

Using N as Distance: Intuitively, the algorithm wants seeds that are "far away" from each other; i.e, random walks starting in one seed shouldn't often end up in the other seed's cluster. N_{ij} represents the number of times that a walk starting at node i visits node j before getting absorbed; this is, then, something of an inverse distance. This heuristic uses a greedy algorithm as follows, the goal being to find K seeds that are "far away" from each other:

```
$tuple \leftarrow argmin(N)$
$seeds \leftarrow [tuple[0], tuple[1]]$
**while** $len(seeds) < K$ **do**
    $seeds.append(argmin_i(\sum_{j \in seeds} N_{i,j})$
**end while**
```

K-Means on N: We can treat each node as lying in a vector space given by the matrix N; i.e, each dimension of the space is another node, and the distance in that dimension is the number of times the node visits that other node before getting absorbed. Now, running K-Means on this vector space gives a rough idea of clustering, but loses all the network structure of the original graph (so it is not very successful on it's own). However, we can run K-Means on N, and then for each cluster take the node in that cluster with the highest degree as a seed node. We then run the Absorbing Markov Chains clustering algorithm on the original graph with these seed nodes.

In practice, using K-Means on N tends to work most consistently. Randomize and Repeat can sometimes outperform K-Means, but the algorithm's dependence on the initial seeds means that in order for those results to be consistent, the algorithm must be randomized and run too many times. Using N as distance works best for some graphs, but tends to fail when the seeds picked have very different degrees, as the cluster corresponding to the highest degree seed usually turns out to be too big; it seems possible to alter the algorithm to take this into account, which could be a direction for future work. Because of it's consistency, all of the following results use the K-Means heuristic.

B. Expectation

We consider each node i to lie in a probability distribution across the cluster, where that probability distribution is exactly B_i.

C. Maximization

Let $V = B^T$. Then for each cluster i V_i is a vector where V_{ij} is the probability that node j is absorbed in cluster i. Now we normalize that vector to sum to 1 so that it's a probability distribution; then the new V_{ij} is the weighted probability that picking a random node in cluster i is node j. Now $VB = K$ gives us an $r \times r$ matrix where (K_{ij} is the probability that starting in a random node assigned to cluster i a random walk will end in cluster j. Since $B = NR$ then $VB = VNR = K$. Once we get our matrix V from the expectation step, we want to change R to maximize the elements on the diagonal of K; i.e we want to maximize $\sum_i K_{ii}$ for $i \in \{1, ..., r\}$.

Given an i, $K_{ii} = (VN)_i \cdot R_i^T$, where R_i^T is the i^{th} column vector of R. This implies that node j contributes to each K_{ii} exactly $((VN)_{ji}^T)(R_{ji})$. We then set R_{ji} to 1 if $i = argmax_i (VN)_{ji}^T$ and 0 otherwise. To see that, note that if it we set both R_{ji} and $R_{ji'}$ to non-negative values, then node j's contribution to the overall sum is $((VN)_{ji}^T)(\beta) + ((VN)_{ji'}^T)(1 - \beta)$ for some β, but surely if $((VN)_{ji'}^T < (VN)_{ji}^T)$ then the contribution would be bigger if all of the probability mass was placed on $((VN)_{ji}^T$.

Because the Maximization step increases the objective function each iteration, the objective function monotonically increases, and the algorithm is guaranteed to converge.

D. Runtime

The algorithm requires one matrix inversion (to find N), which runs in runtime $O(n^3)$; however, this inversion need only happen once. Afterwards, each iteration requires one matrix multiplication (to find B), which also runs in runtime $O(n^3)$. While the overall runtime, then, is $O(n^3)$, the constants of runtime are quite low.

E. Binary versus Fuzzy Classification

Considering nodes as a probability distribution across clusters, rather than simply classifying each node as the cluster that has the most probability in that distribution, allows for more movement in the algorithm, and tends to produce better results than my original method of binary classification. It should be noted that the fuzzy method preserves small clusters more than the binary method; this is because, in the binary method, a small cluster may end up having no nodes at all, while in the fuzzy method, even if every node only rarely is absorbed in a cluster, when computing the maximization step that cluster's vector is normalized to 1 anyways, so nodes that are very rarely absorbed in that cluster (but are more often absorbed in that cluster than other nodes) will be considered to be a big part of that cluster. In theory, this could lead to problems and bad clustering because the algorithm favors small clusters in some sense; in practice, fuzzy clustering typically works better.

VI. RESULTS (USING K-MEANS HEURISTIC)

A. Datasets with Ground Truth

On the football dataset, my algorithm gets 92% accuracy. For comparison, the MCL[8] algorithm has an accuracy of 94%, and WalkTrap has an accuracy of 87%. So while my algorithm beats WalkTrap, it loses to MCL; however, the dataset includes independant football teams (which therefore have no true ground-truth community). My score of 92% is with every independent football team misclassified (the algorithm does not label any team as in the "independent" cluster, as it is clearly not truly a cluster). In addition, every other misclassification in my algorithm (except for one) is a team in either the Sun Belt or Western Athletic conference. The Sun Belt teams played nearly as many games against Western Athletic teams as they did against games in their conference, so it seems expected that the algorithm would fail in this case.

On Zachary's Karate Club, my algorithm gets 100% accuracy. MCL gets 97% and WalkTrap gets 97%. The K-Means heuristic does not pick the administrator and the instructor as seeds, but the nodes it does pick leads to 100% accuracy anyways.

On the political books dataset, my algorithm gets 89% accuracy. I have no comparison for this dataset as I didn't find any other papers analyzing this dataset. However, as the nature of the network is likely quite noisy (for example, people might frequently purchase two books with different political slant to compare the differences), an accuracy of 89% seems good.

B. CS229 Project

My final project for another class I'm currently in (CS229) involved using Machine Learning to classify gene function and localization. The dataset is from the 2001 KDD Cup competition, and has two components: A dataset with genes and a list of given features, and a dataset with gene interaction (specifically, expression correlation). To derive features to use in my classifier, I created a graph where the nodes were the genes, and the weight of an edge is the absolute value of their correlation. I then ran my clustering algorithm on that graph with K=16 (the number of possible localizations). For each gene, I added a feature that corresponded to the cluster it was in. These features were quite informative. To show the power of these features, I re-implemented one of the most simple classification algorithms, softmax regression, and fed it a training set that consisted only of the features that were already given in the data set and the cluster features; no other techniques were done for this classifier. On the competition test set, this softmax regression classifier scored 66.6% accuracy. For reference, a classifier with around 69% accuracy would have been in the top 5 of the competition, so these results are impressive given the simplicity of the classifier.

VII. CONCLUSION

When given a graph in which domain knowledge suggests the existence of K clusters, this algorithm performs remarkably well in finding those K clusters. The algorithm is especially powerful when domain knowledge allows the manual selection of K seed nodes; however, the heuristics outlined in this paper reasonably approximate such manual selection. The algorithm's time complexity is comparable to, if not better than, most algorithms.

For future directions, I'd like to consider using this algorithm as the Expectation step in a larger Expectation-Maximization algorithm that automatically finds K. In particular, since this algorithm has the benefit of allowing us to view nodes as having a probability distribution over clusters, we have a natural notion of "similarity" between two clusters in the form of a dot product (if two clusters are almost the same, i.e nodes tend to be in both clusters equally, their dot product will be high; if they are very different, their dot product will be low because they are "perpendicular"). Perhaps an algorithm could be developed that initially assumes that each node is a seed node, runs the clustering algorithm given that assumption, and then looks at the clusters and merges clusters that are similar enough, and finally re-runs the algorithm with new seed nodes corresponding to the remaining clusters.

Such an algorithm would circumvent the need to know K; however, as many clustering applications involve domain knowledge that suggests the correct value of K, this algorithm is still quite useful.

---

[1] Brandes et al, "On finding Graph Clusterings with Maximum Modularity", http://www-i1.informatik.rwth-aachen.de/en/ mhoefer/Forschung/05-07/Brandes07Modularity.pdf

[2] "Community detection in graphs", http://snap.stanford.edu/class/cs224w-readings/fortunato10community.pdf

[3] Zhou, H., 2003a, Phys. Rev. E 67(6), 061901. Zhou, H., 2003b, Phys. Rev. E 67(4), 041908. Zhou, H., and R. Lipowsky, 2004, Lect. Notes Comp. Sci. 3038, 1062.

[4] "Computing Communities in Large Networks using Random Walks", http://www-rp.lip6.fr/ latapy/Publis/communities.pdf

[5] "Finding and Evaluating Community Structure in Networks", http://snap.stanford.edu/class/cs224w-readings/newman03community.pdf

[6] http://www-personal.umich.edu/ mejn/netdata/

[7] http://pages.cs.wisc.edu/ dpage/kddcup2001/

[8] http://micans.org/mcl/