CommunityFinder: Software to Analyze Communities that Develop in MOOC Discussion Forums

Max Kanter Kalyan Veeramachaneni
Any Scale Learning for All
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Contents

List of Figures

List of Tables

Appendices

Abstract

CommunityFinder is software to analyze the communities that develop in massively online open courses (MOOCs). The goal of the tool is to give researchers and teachers further insight into their classes. The software organizes a corpus of forum posts into an Interaction Graph that represents how students interact with each other. From this graph, the software can perform community detection using two different algorithms. Along with the source code, we also release the results of running CommunityFinder on two different edX courses.

1 Introduction

Massively online open courses, or MOOCs, are offered by universities across the country, including MIT. Over 100 classes are offered on platforms like edX and Coursera and this number is growing. Due to the nature of being online and open, a lot of data is being collected about how students interact with the course material.

The discussion forums in these classes are the primary way that students and teachers interact in these online courses. For example, the first offering of 6.002x (Circuits and Electronics) in Spring 2012 had over 90,000 posts made.

In this paper, we present CommunityFinder, which is software that extracts structured representation of how student interact with MOOCs and then performs community analysis on this structured representation. The software outputs information on several aspects of the community structure in an online course.

2 Motivation

Understanding the community structure of a graph can bring improved understanding of complex systems. MOOCs provide a new complexity to education in not only the size of their enrollment, but also in the different way that they teach material. If we understand qualitatively how students interact with each other and instructors in a course, we will be able to better understand how successes and failures of MOOCs. This understanding will allow educators to improve these online course moving forward.

3 Previous Work

The problem of identifying communities is very hard and not yet satisfactorily solved. The dependency on domain makes it difficult to generate general purpose algorithms. Further, some domains such as social network data require not only accurate, but efficient algorithms to process large amounts of data. However, there is a large interdisciplinary effort to solve it because of the number of applications.

The community structure in a network of interactions represented as a graph where entities are vertices and interactions are edges is a clustering of a graph such that vertices in a cluster have more edges to vertices in the same cluster and fewer edges to vertices in other clusters [3]. How we interpret the clusters that form is often dependent on the domain of the problem, but they often are interpreted as vertices playing a similar role. For example, given an network of how scientific researchers have collaborated with each other, we would expect to find that communities correspond to the different scientific disciplines.

In the domain of online education, we might want to identify the students that take various roles in a class. Huang et al. [5] focused on the most focal subset of contributors on MOOC forum that they label as *superposters*. They suggest the *superposters* can be models for the ideal student because they often make high quality posts. Their study seeks to examines contribution patterns, demographics, and course performance and enrollment of these *superposters*. The researchers conclude that these posters make high-value contributions and also encourage further student engagement.

Other researchers analyzed post content from business strategy class on Coursera [4]. This meant considering over 15,600 posts. They looked at 5 aspects a student's postings and then used Bayesian Non-negative Matrix Factorization to extract communities. They looked specifically at two sub-forums and identified communities such as committed crowd engagers, discussion initiators, strategists, and individuals.

4 Student Relationships in Discussion Forums

Students interact in the discussion forums. Because of this, we can infer inter-student relationships based on a corpus of forum posts.

The discussion forms we consider in this paper are organized as a collections of threads. Within a thread, students have the opportunity to converse with each other on a common topic. Figure 1 is an example thread from a class currently being offered on edX.

We rely on the thread structure of the conversations to determine student relationships. A thread is a hierarchy of posts where every post besides the first has a parent post that it is associated with. The parent/child relationship is interpreted as the child post being a reply to the parent. We use this interpretation to say author of a child post has interacted with the author the parent post. We only consider the immediate parent.

We consider two students to be related if they interact in a thread. The strength of this relationship is determined by how often one student replies to another. The idea is that the more often we find one student replying to another, the more likely are in the same communitiy.

Using this understanding of how stu-

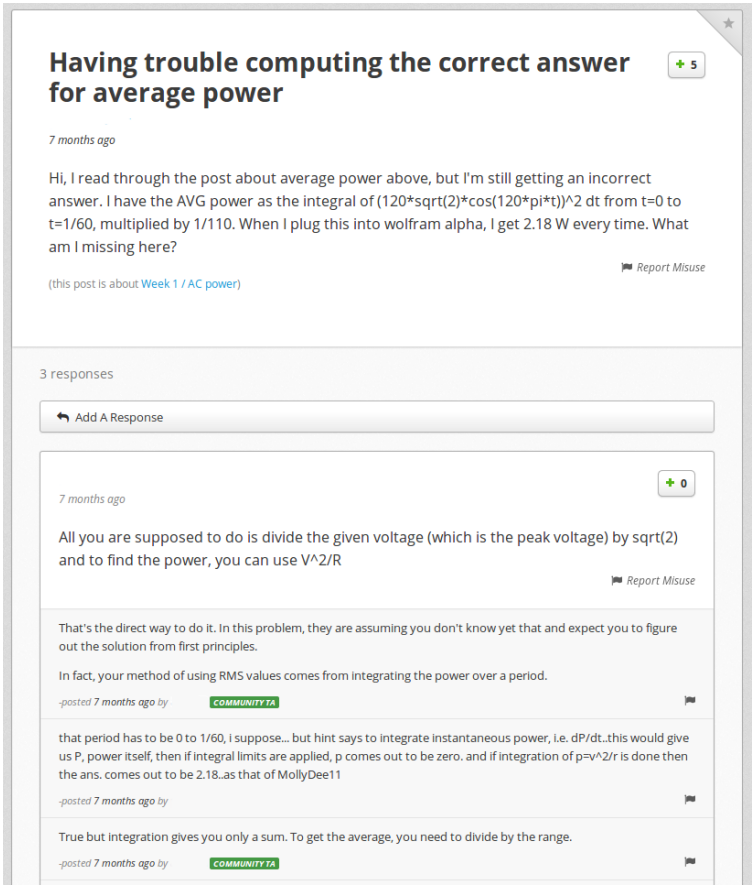

Figure 1: An example thread from 6.002x Spring 2014, which is currently being offered.

dents who post on the forum are related, we present the idea of an Interaction Graph.

4.1 Interaction Graph

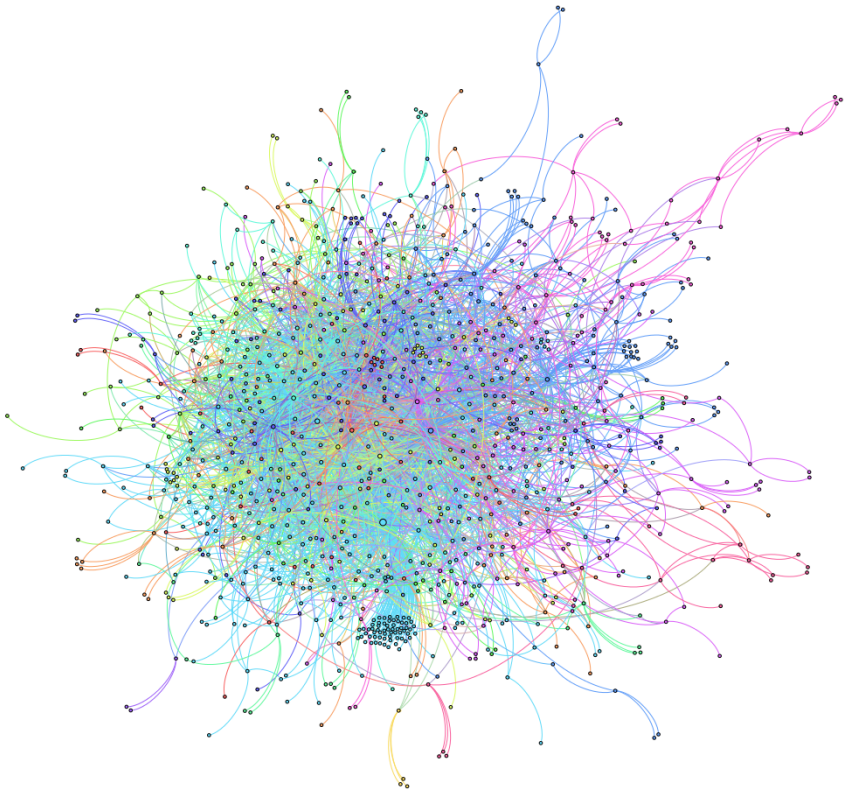

Figure 2: This is a visualization of the Interaction Graph for 6.002x Fall 2012. This graph has 2163 students who made 6902 posts.

In an Interaction Graph, each vertex represents a student in the class. For two students i and j, the graph has a directed edge from i to j if i has made a comment in direct reply to j. The edge between i and j is weighted by the number of times that i directly replied to j across all threads in the forum. We use this directed and weighted model to preserve as much information as possible about the the original forum interactions. Only students who are involved in a thread with at least one reply are considered. Figure 2 shows what the graph of student interactions looks like for 6.002x offered in Fall 2012.

5 Community Detection

We considered two different approaches to identifying communities: spectral clustering and Louvain Method.

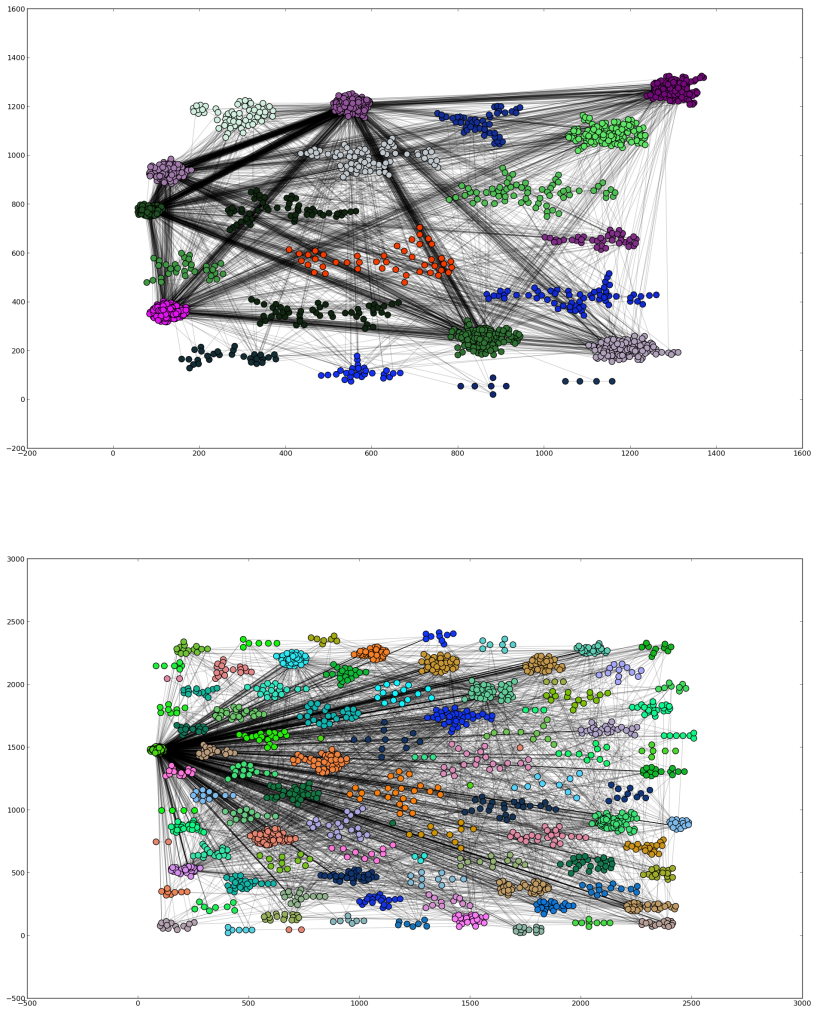

Figure 3: Visualization of the results of the highest modularity partitioning of 6.002x Fall 2012. On the top is the the best Louvain Method partition. On the bottom is the best spectral clustering partition.

5.1 Spectral Clustering

Spectral clustering is a group of techniques that transforms the initial set of verticies in space by using eigenvectors. After this transformation, the points are clustered using a standard algorithm [3]. Spectral clustering can be broken down into 3 parts

Affinity Matrix The algorithm uses an affinity matrix that describes the similarity between two vertices. In our case, the affinity matrix is exactly an instance of the Interaction Graph. This works because the ij entry of an Interaction Graph encodes how the strength of the relationship between i and j.

Dimension Reduction The algorithm uses the spectrum of the graph Laplacian to reduce the dimension of the matrix. The Laplacian, L, of the graph is defined as

$$L = D - A \tag{1}$$

where D is a diagonal matrix with $d_{ii} = deg[i]$ and A is the adjacency matrix. In our implementation[1], we use the normalized graph Laplacian which is defined as

$$L' = D^{-\frac{1}{2}} L D^{-\frac{1}{2}} \tag{2}$$

This is preferred to the unnormalized Laplacian because it performs better under general conditions [8]. Using the eigenvectors of the normalized Laplacian, each vertex is transformed into a lower dimension space where the coordinates are elements of the eigenvectors. This reduction is important because it makes the important features of the initial matrix more prominent before applying clustering. Thus, this method allows clusters to be identified that would not have been identified by standard clustering techniques. For example, spectral clustering tends to transform the initial into points into a convex sets of points, which are easier to cluster [3].

Clustering The algorithm uses a discrete method proposed by Yu and Shi to do clustering in the lower dimension space. This method was chosen over KMeans for clustering because it is often faster and more robust to random initialization than KMeans [9]. The clustering is found by iteratively searching for a discrete partition closest to the eigenvector embedding. This done by first normalizing the eigenvector embedding to the space of the partition matrix. Next, a we fix an optimal discrete partition matrix and then calculate the optimal rotation matrix. These to steps are performed until convergence. The discrete partition matrix is returned as the solution to the clustering problem.

5.2 Louvain Method

The Louvain method is a greedy optimization method for determining communities in large networks [2]. It seeks to optimize the modularity of a partition of the network. Given a partition of the graph, the modularity measures the relative frequency of links inside communities as compared to links between communities. It is scalar value between -1 and 1 and is calculated as follows

$$Q = \frac{1}{2m} \sum_{i,k} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j) \tag{3}$$

[1] In this paper, we use an implementation of spectral clustering and discrete clustering provided by scikit-learn [6].

where A_{ij} is the weight of the edge between i and j, $k_i = \sum_j A_{ij}$, c_i is the community that i is assigned to, the $\delta(u, v)$ is 1 if $u = v$ and 0 otherwise, and $m = \frac{1}{2} \sum_{ij} A_{ij}$.

Exact modularity optimization is NP-hard [2]. The algorithm starts by assigning a unique community to each node and then iteratively performing two steps until maximum modularity is achieved. First, the method looks at every node i and its each of its neighbors j. Node i is removed from its community and added to the community of j where the gain in modularity is highest. If the modularity cannot be improved by moving i, then the community assignment will not change. This is repeated until no more changes in community can be performed to increase modularity. The second step is to build a new network where the nodes are the communities discovered in the first step. Edges between nodes in the same community are self loops in the is new graph. These steps are repeated iteratively until no more changes occur during an iteration, which means a maximum of modularity has been reached. This produces a hierarchy of communities. This method has been observed to be very fast and appears to run in $O(nlogn)$ [2], where n is the number of nodes in the network[2].

6 CommunityFinder Software

6.1 Overview

CommunityFinder identifies the community structure in a set of forum posts. It does this by infering the strength of a pair of students' relationship based on how often they make posts to each other. After this structure is determined for all students, CommunityFinder offers two methods of inferring the community structure implied by student interactions.

6.2 Data Sources

In order to extract an Interaction Graph for analysis, CommunityFinder can query from multiple data sources. Currently, it supports a MySQL database in the MOOCdb [7] format, a Mongo database dump as provied from edX, and an edge list formatted as a file of comma separated values. We recommend using MOOCdb as it is a standardized schema for representing MOOC data.

6.3 Design Decisions

CommunityFinder is built with flexibility in mind. It is not meant to be tied to an specific MOOC platform. To this end, it is has support for querying directly from any course data that is in the standardized MOOCDB schema. This is the recommended way to use the software.

However, it is easy to extend the software to support other data sources. Included in the initial release is support for extracting graph structure from the Mongo database dumps that the edX platform creates.

Even more, the software supports users who provide their own dataset. An edge list represented as a CSV can be used for analysis by this software. This means that it is possible to use CommunityFinder to find community structure in any graph.

The software organizes the Interaction Graph into a hierarchy of higher level features as shown in Figure 5. Students are the lowest level. Students are defined by an individual who post on the forum. Student are

[2]In this paper, we use an implementation for directed graphs for the NetworkX library [1].

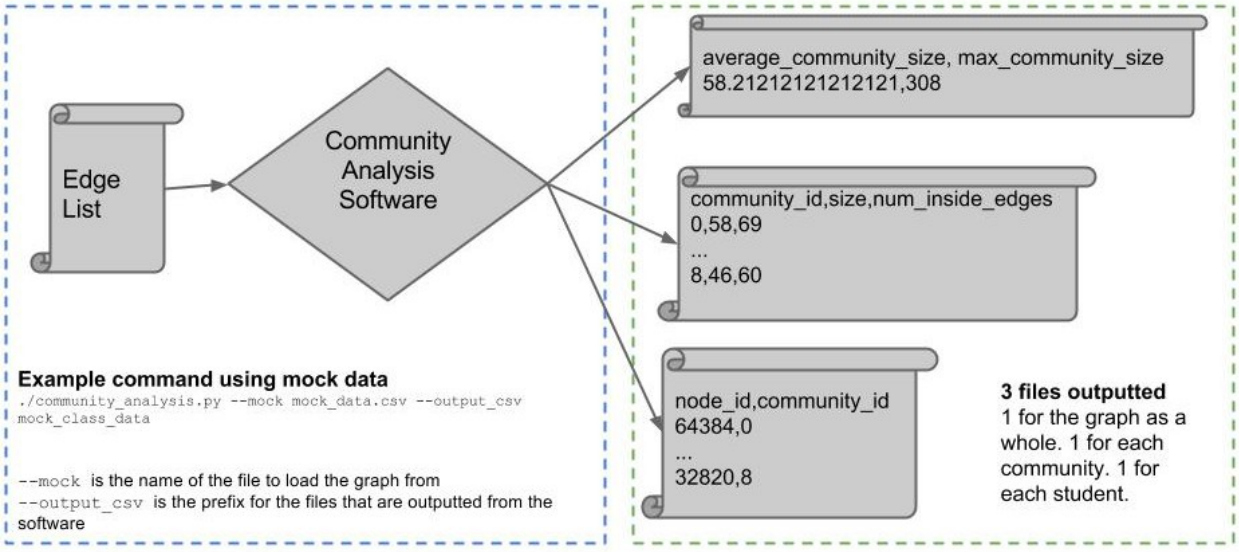

Figure 4: CommunityFinder performs analysis on an edge list and output three files that describe the community structure discovered by the graph

grouped together to forum communities. A community is a collection of students and has connections to other communities. This hierarchy has a few important implications

- It is how the data is structured to be exported from the software. This tool is not meant to do analysis beyond community identification. Thus, it is important that the software exports all relevent features about the community structure of the graph that later stages of analysis might use. On the practical side, this structure enabling the programming of additional features for each of the levels.
- It gives a good building block for extending functionality. As I discuss the Future Work section, we might want add higher level abstractions such as how the communities develop over time in a class. In order to do this, we simply can create a higher level abstraction that relates several community graph objects.

6.4 Adding Functionality

Each file that the program outputs is the enumeration of instances of a particular part of the hierarchy in Figure 5. Right now, these are CommunityGraph, Community, and Student. This means that CommunityFinder outputs a file of comma separated values for each instance at each level. The values that are output depend on the list of properties at the top of each class definition. To add another property, simply add a string to this list. The proprieties can involve additional computation by using python decorators to label certain methods as properties.

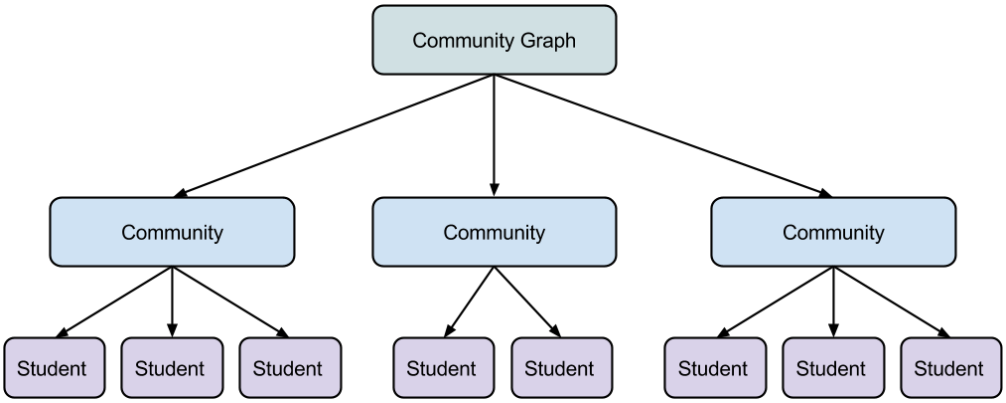

Figure 5: This represents how the CommnityFinder software understands the structure of the interaction graph. The hierarchical structure was chosen to enable flexibility as the software is developed.

7 Results

The Louvain Method has the advantage of not requiring that that number of communities be explicitly set when running the algorithm. For spectral clustering, we simply try all possible values for the number of communities between 1 and the half the total students in the class and present the partition of the highest modularity.

Using CommunityFinder, we extracted the Interaction Graph from two past edX offerings of the course 6.002x. We note that CommunityFinder only counts students that had an interaction with at least 1 other student. The results of running CommunityFinder are presented in Table 1.

Course Offering	6.002x Spring 2012	6.002x Fall 2012	6.002x Fall 2012	6.002x Spring 2012	6.002x Spring 2013
Method	Louvain	Louvain	Spectral	Louvain	Spectral
Number of Students	8816	2163	2163	710	710
Number of Edges	52362	5641	5641	1564	1564
Number of Posts	71440	6902	6902	1914	1914
Number of Communities	37	77	114	32	72
Average Community Size	238.3	19.0	22.1	20.3	9.9
Largest Community	1438	204	413	87	76
Modularity	.360	.510	.442	.539	.502

Table 1: Summary of results of running CommunityFinder 3 edX offerings of 6.002x. We do not present the results of 6.002x Spring 2012 using the spectral clustering because CommunityFinder ran out of memory during the testing for that course.

The running time of these community detection algorithms is also an important consideration if we want to be able to analyze a large number of classes. Table 2 shows the running time for 6.002x Fall 2012.

Unlike Louvain Method, spectral clustering requires the number of communities in order to produce a result. The result we present in Table 1 show the number of clusters that achieves the highest modularity.

Method	Louvain	Spectral 10 communities	Spectral 50 communities	Spectral 100 communities
Running Time	.792 sec	3.300 sec	5.068 sec	6.630 sec

Table 2: The total running times for CommunityFinder under different parameters.

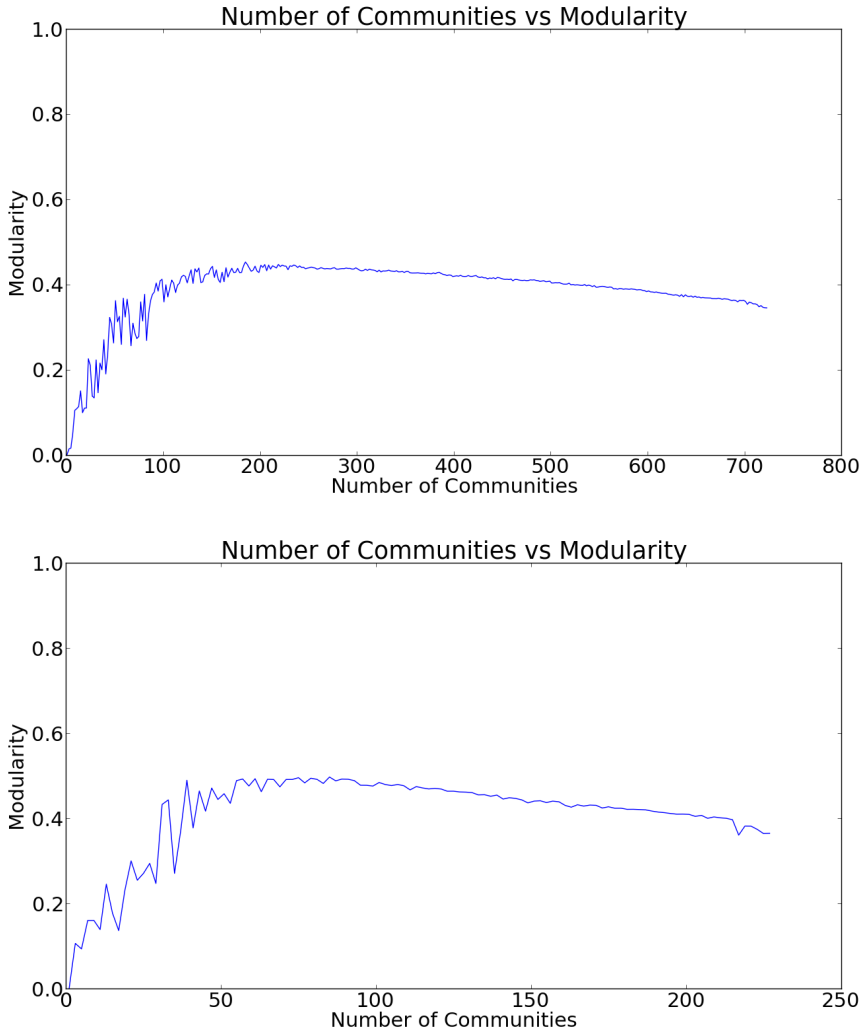

Figure 6: Number of Communities vs Modularity. 6.002x Fall 2012 is on top and 6.002x Spring 2013 is on bottom

Figure 6 shows how the number of communities specified affects the modularity of interaction graph for 6.002x Fall 2012.

8 Discussion

The initial results show the Louvain Method obtains a higher modularity than spectral clustering in two offerings of 6.002x. In the third offering, only the Louvain Method produced results. The Louvain Method also achieves this maximum modularity with a lower number of of communities when compared to spectral clustering.

Spectral clustering requires the number of communities to be specified. In order to optimize modularity, CommunityFinder does a brute force search over the number of communities. Figure 6 shows how the modularity varies as we vary the number of communities. We can see that increasing the number of communities increase modularity up to a certain point. After that, further increases in number of communities decreases the modularity.

The other important consideration is performance. The Louvain Method vary clearly outperforms spectral clustering on our dataset. The running time of Louvain method for 6.002x Fall 2012 is nearly 10x faster than using spectral clustering to determine 100 communities on equivalent Interaction Graph. The advantage becomes clearer when we determine the optimal number of communities in spectral clustering because we must actually run the the algorithm multiple times. For example, in our testing, we tested nearly 700 different values for the number of communities in 6.002x Fall 2012. This further amplifies the performance advantage of the Louvain Method.

Overall, these two results suggest that the Louvian Method is better for community detection under the goal of modularity optimization. If we consider another metric the results may change. Additionally, we only ran this analysis on three datasets, all from offerings the same course. We may see different results for other courses, especially ones in disciplines other than computer science.

9 Contributions

- Described how to transform a corpus of forum posts into an Interaction Graph
- Examined 2 methods for community detection on an Interaction Graph and presented intital results
- Implemented software to extract Interaction Graph from various representation of the raw data and run community analysis.

10 Future Work

CommunityFinder is first and foremost a tool to do community analysis. As such, it is meant to be used to do further research on MOOCs. The functionality should grow to accommodate the needs of the people using the tool. Thus, it will be important to see how people use CommunityFinder.

The are a few good ideas already for future features. For instance, CommunityFinder could implement other community detection algorithms. There has been a lot of work on other algorithm that have not been tested. Some of the most interesting are the ones that do not limit a student to being in only one community. Developing support overlapping communities is a one possible future feature. A second feature is one that tracks how community structure changes over time. Researchers might want to know how a community morphs week to week in order to assess how effectively course material is being taught.

Finally, an interesting project would be to use CommunityFinder to relate superposters (as identified by Huang et al.) to their community assignment.

References

[1] Thomas Aynaud. Community detection for networkxs documentation, 2009.

[2] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 10:8, October 2008.

[3] S. Fortunato. Community detection in graphs. , 486:75–174, February 2010.

[4] Nabeel Gillani, Rebecca Eynon, Michael Osborne, Isis Hjorth, and Stephen Roberts. Communication communities in moocs. *CoRR*, abs/1403.4640, 2014.

[5] Jonathan Huang, Anirban Dasgupta, Arpita Ghosh, Jane Manning, and Marc Sanders. Superposter behavior in mooc forums. In *Proceedings of the First ACM Conference on Learning @ Scale Conference*, L@S '14, pages 117–126, New York, NY, USA, 2014. ACM.

[6] Scikit-Learn. Spectral clustering, 2014.

[7] Kalyan Veeramachaneni, Franck Dernoncourt, Colin Taylor, Zachary Pardos, and Una-May OReilly. Moocdb: Developing data standards for mooc data science. In *AIED 2013 Workshops Proceedings Volume*, page 17, 2013.

[8] Ulrike Von Luxburg, Mikhail Belkin, and Olivier Bousquet. Consistency of spectral clustering. *The Annals of Statistics*, pages 555–586, 2008.

[9] Stella X Yu and Jianbo Shi. Multiclass spectral clustering. In *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, pages 313–319. IEEE, 2003.

Base Class for Data Source

```
class MoocData():
    def get_edges():
        raise NotImplementedError

    def anon_rows(self, rows):
        """
        rows ---> rows of edges list
        """

        mapping = {} #real id to "fake" id
        new_rows = []
        for r in rows:
            if r[0] not in mapping:
                mapping[r[0]] = random.randint(0,1000000)

            if r[1] not in mapping:
                mapping[r[1]] = random.randint(0,1000000)

            new_row = ( mapping[r[0]], mapping[r[1]], r[2] )
            new_rows.append(new_row)

        return new_rows, mapping

    def export_edges(self,f):
        rows = self.get_edges()
        rows, mapping = self.anon_rows(rows)

        with open(f+"_mapping", 'w') as out:
            out.write(json.dumps(mapping))

        with open(f+"_edges", 'w') as out:
            for r in rows:
                out.write( "%d,%d,%d\n" % (r[0], r[1], r[2]) )
```

Class for Mock Data Source

```
class MockMOOC(MoocData):
    """
    Mock datasource. Takes a file name. look at mock_data.csv for example
    """
    def __init__(self, f):
        self.f = f

    def get_edges(self):
        edges = []
        with open(self.f, "r") as edges:
            edges = [e.split(",") for e in edges]
            return [(int(e[0]), int(e[1]), int(e[2])) for e in edges]
```

Class for MOOCdb Data Source

```
class MySQLMOOC(MoocData):
    def __init__(self, db=None, mock=True):
        mysql = db
        self.cur = mysql.cursor()

    def get_edges(self, start_date='2012-03-04 16:57:49', end_date='2012-03-06 16:57:49'):
        query = """
        SELECT child.user_id, parent.user_id,  count(*)
        FROM moocdb.collaborations  child
        JOIN moocdb.collaborations parent
        ON parent.collaboration_id=child.collaboration_parent_id
        WHERE child.collaboration_timestamp BETWEEN %s and %s
        GROUP BY child.user_id, parent.user_id;
        """

        self.cur.execute(query, (start_date, end_date))
        rows = self.cur.fetchall()

        return rows
```

Class for MongoDB Data Source

```
class MongoMOOC(MoocData):
    """
    command to load dump
    '/mongodb/bin/mongoimport.exe --db %s --collection %s  --jsonArray < "%s"' % (db, colle
    """
    def __init__(self, db):
        self.db = db
    def get_edges(self, start_date='2012-03-04 16:57:49', end_date='2012-03-06 16:57:49'):
        graph = self.db.collaborations.aggregate([
        {
            '$match' : {
                'parent_ids' : {
                    '$not' : { '$size' : 0 }
                }
            }
        },
        {
            '$unwind' : "$parent_ids"
        },
        {
            '$group' : {
                '_id':"$author_id",
                'neighbors' : {
                    '$push': "$parent_ids"
                }
            }
        }])

        result = []
        for node in graph['result']:
            neighbors = db.collaborations.aggregate([
                {
                    '$match' : {
                        "_id" : {
                            '$in' : node['neighbors']
                        }
                    }
                },
                {
                    '$group' : {
                        "_id" : "$author_id",
                        "weight" : {
                            '$sum' : 1
```

```
                    }
                }
            }

        ])

        for neighbor in neighbors["result"]:
            edge = (int(node["_id"]), int(neighbor['_id']), neighbor['weight'])
            result.append(edge)

    return result
```