COMPUTER SCIENCE

A Decision Tree Based Approach to Filter Candidates for Software Engineering Jobs Using GitHub Data

Songtao Hei

Thesis under the direction of Professor Christopher Jules White

A challenge for companies hiring software engineers is the large number of candidate profiles on LinkedIn, Monster.com, and other job websites and the inability to easily filter top candidates from these lists. In this paper, we propose a novel approach for utilizing the social network structure in GitHub and a decision tree algorithm to solve this problem efficiently and filter candidate software engineers. The approach is based off the idea that the centrality value of a node (i.e., candidate engineer) in the graph of GitHub users is an approximate indicator of the value of the programmer. To reduce the number of candidates that are considered in a job selection process, a threshold centrality value can be used to filter job candidates based on their importance in the GitHub user graph. A challenge with this approach is that, since GitHub has millions of users, calculating the centrality for every node in the GitHub user graph is an expensive operation. To overcome this challenge, we train a decision tree to predict a user's centrality based on a limited subset of their attributes. To generate training data for the decision tree from the unlabeled GitHub user graph, a threshold centrality value is chosen and a part of the user graph is labeled with Accepted or Rejected based on whether or not the corresponding user meets the threshold centrality. We also collect the total number of the each kind of public GitHub event each user has generated and we use the number of these GitHub events as training attributes for each user in the training dataset. Once decision trees are built with this training dataset, recruiters can use these decision trees to process large quantities of software engineering job candidates and to improve the judgment of HR departments. Based on empirical results from experiments that we conducted with GitHub user data, our approach can reach a precision of 96%. Moreover, this method saves future expensive network centrality computation as the GitHub social graph changes over time.

Approved: Christopher Jules White Date: 3/20/15

A Decision Tree Based Approach to Filter Candidates for Software Engineering Jobs Using GitHub Data

By

Songtao Hei

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Computer Science

May, 2015

Nashville, Tennessee

Approved:

Professor Christopher Jules White
Professor Douglas C. Schmidt

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

It has become increasingly challenging to evaluate large numbers of software engineering candidates for jobs and there are few objective indicators to help human resource (HR) departments screen candidates. According to a 2012 Jobvite survey [1], most recruiters take social network information into account when hiring: 54% of recruiters use Twitter, 66% Facebook, and 97% LinkedIn as recruitment tools, and a total 92% of the recruiters use or plan to use social networks for recruiting. Furthermore, 49% of the recruiters saw an increase in the quantity of candidates, 43% reported an increase in candidate quality, and 20% reported it took less time to hire if they used social network data. Therefore, it is a reasonable approach to take the GitHub social network into account in the software engineering recruitment process. Furthermore, because the decision making process for recruitment is based on the biases of each recruiter, it would be valuable for recruiters to have some kind of objective and intuitive indicator to reference when filtering candidates. The objective of my thesis is to provide an efficient approach for recruiters to estimate a job candidate's skill based on their impact on the GitHub user network.

GitHub, as a popular social coding community and has attracted a large number of programmers to contribute to the open source community. Millions of registered GitHub users have helped to establish a very complex social network through the 'Follow' mechanism provided by GitHub. In this paper, we view every user as a node in a directed graph; and if one user has followed another user, then there is a directed edge from this node to the node that user represents. Although not all programmers are on GitHub, we can still use the graph built by this method to approximately evaluate candidates in the recruiting process. The graph built by this method generally represents the community structure and influence relationships among the programmers. According to this social structure, we can explore the approximate impact of a programmer on this graph and the open source community.

In this thesis, we use the centrality value of a node in the graph we build as an indicator of the value of the programmer. Although this is not a very objective standard to assess

the skill level of the programmer, it can be at least used as an approximate indicator to judge how active and devoted a programmer is and how impactful their projects are on the open source community – both of which are indirect indications of programmer skill. The intuition behind this idea is that, although not all good programmers are socially active and influential on GitHub, most of the socially active and influential programmers are valuable to the coding community, which is an objective standard to take into consideration in the recruiting process. Although not perfect, this estimate can be a good metric that helps to facilitate HR's work of narrowing down a list of candidates.

What we want to achieve in this project is to build a model that can quickly tell a recruiter if a candidate is likely to have a high impact on the GitHub community or not. A challenge with this vision is that in order to build the training dataset for this goal, we need to compute the centrality value for every user and select a threshold to label users who are deemed acceptable candidates – the computational complexity is high and the appropriate threshold is not obvious. Further, there are many different kinds of centrality calculations that can be used for a graph. In this paper, we compare three different centrality computation methods and conduct experiments to examine their impact on the outcome.

In order to train a decision tree to predict the centrality value of a user, we need training attributes to base the prediction on. There are 25 types of events that a user can generate through the GitHub API. We used the GitHub Archive [2] to collect all events generated by each user and used the number of each type of event that a user produced as the features that we based our decision tree centrality threshold predictions on.

For the decision tree, we used the random forest algorithm and trained it with our collected dataset. We conducted experiments to test the trained random forest model and how well it predicted if a candidate's centrality value was above the threshold used to filter out job candidates. We used four different evaluation methods to examine the trained model. The final result shows that our model can correctly filter out greater than 96% of candidates.

An additional benefit that we will show in this paper is that this approach eliminates the need for expensive network centrality computations in the future as the GitHub user graph evolves. Each day, there is a huge volume of changes in both the nodes and edges of the GitHub graph every day. If we were only to use centrality value to judge if a candidate should be accepted or not, we would need to recalculate the centrality value for the entire user graph of GitHub, which would be very expensive considering that there are over 5 million nodes in the graph and continues to grow.

The remainder of this paper is structured as following: part II, Challenges, describes the challenges that we faced with this work, part III, Related work, covers prior research in this area, part IV, Methodology and implementation, provides a detailed description of our methodology, part V, Empirical results, presents results from our experiments with the GitHub user graph, and part VI, Concluding remarks and future work.

II. CHALLENGES

The main existing challenge in the technology recruiting process is the lack of an objective indicator to filter large quantities of applicants. Recruiters have limited time to review each candidate's resume line by line and objectively determine if an applicant deserves a follow-up phone interview. Recruiters rely on their experience and recruitment networks, but the whole process is still subjective and subject to significant bias. However, the most significant challenge is still the fact that the number of applicants is unwieldy, and there are insufficient tools to preprocess the candidates' information efficiently and give meaningful indications of potential skill.

Also the method of using centrality value brings the challenge of computing centrality value, which is very expensive in practical setting. For this research, we used ***Betweenness centrality***, for which the best algorithm so far has a computing complexity of O(VE) for unweighted graphs and O(VE+V(V+E)logV) for weighted graph with the space complexity of O(VE), where V stands for the amount of the vertex in the graph, and E stands for the amount of edges in the graph [3]. In a 16-core *high-cpu* Google Compute Engine Virtual Machine [4], we profiled the algorithm using graph-tool [5], which has an average 398.3

seconds per call for a random graph with 39,796 vertices and 301,498 edges. With a graph of millions of nodes and edges, as is the case with the GitHub social graph, the running time for the algorithm increases dramatically. Due to the large size of the GitHub graph, it is impossible or highly expensive to recalculate the centrality value on an ongoing basis. This large computational cost is the primary motivation for our proposed prediction that can predict centrality values with little computational expense directly from the user's GitHub usage history.

III. RELATED WORK

In this section we present prior research related to the proposed approach. There is a long history of research with the machine learning algorithms used in our research: decision trees [6] and ensemble methods of random forest [7]. The centrality value of a graph and the three graph centrality computing methods we employed in our experiment are Betweenness centrality [8], PageRank centrality [9] , and Eigenvector centrality [10], which were developed in prior work. For validation our results, we use four well-known metrics: logarithmic loss [11], precision, recall, and F-1[12]. The remainder of this section covers each of these key related works that we rely on.

Decision tree

The decision tree algorithm is a supervised machine learning method that can be used for classification and regression into a finite set of labeled classes (e.g., car, person, flower, etc.). In this work, the decision tree is used for binary classification of whether or not a candidate's graph centrality is expected to be above a threshold value. To build a binary classification tree, a recursive partitioning process is applied to the training data set. Every time the training dataset is split, a test is performed to find out which attribute of the training dataset has the highest information gain and then the dataset is split along that attribute. This process repeats recursively until there is no data to partition or a subset is found that has the same value for the target attribute in all data items. There are a number of variations on the splitting method that optimize for speed, accuracy, or other properties.

ID3 [13] (Iterative Dichotomiser 3) was firstly introduced by Ross Quinlan in 1986. The algorithm builds a multi-branch tree by using a greedy algorithm to split the dataset using the categorical feature that will yield the lowest entropy. This algorithm is the original decision tree learning algorithm but doesn't consider attributes with continuous numeric values. The drawback of this algorithm is that modeling continuous attributes as discrete values and applying the splitting algorithm can run into problems of overfitting, which lead to poor prediction results.

C4.5 [14] is a successor to the ID3 algorithm, which has much better performance when operating on numeric continuous attributes in a training dataset. To overcome prior limitations with continuous attributes, the algorithm dynamically defines a discrete attribute that splits continuous numeric values into two branches that represent the attribute, partitioning the continuous attribute value into a discrete set of intervals. The C4.5 algorithm transforms the tree model from ID3 into sets of if-then guidelines for classification. The accuracy of each if-then rule is assessed to decide the order in which they should be applied to produce the overall highest accuracy classification. Pruning is carried out by removing rule preconditions for any rule that performs better without the precondition. Quinlan et al.'s latest C5.0 [15] algorithm has a number of improvements to make it more memory-efficient compared to C4.5 while also producing more accurate predictions.

CART [16] (Classification and Regression Trees) is the method we employed in the thesis, which is the standard implementation in the scikit-learn [17] python library we chose to use in our experiments. This algorithm is very similar to C4.5 in the model building process and dataset partitioning methods, but it differs in that it supports numerical target variables, thus it can function as a regression learning algorithm instead of just purely as a classifier. The algorithm does not compute rule sets. CART constructs binary trees through a process of thresholding attributes with continuous values and chooses the values that yield the largest information gain when splitting the dataset into two parts. For this research, we needed to process and experiment on large datasets and the parallel computing capabilities of CART in scikit-learn implementation produced good performance.

There are several advantages of choosing decision trees as our binary classifier. Firstly, the decision tree model is relatively simple to understand and to interpret in the program, and the model itself can be easily visualized, which will facilitate further analysis in the future. Decision trees also don't require substantial data preprocessing before a data set can be processed. This is especially important for this project because the GitHub dataset itself is large and transformation and processing can be time consuming and error-prone. Compared to other techniques that require data normalization, creating dummy variables, and removing blank values, etc., decision trees are much more convenient to use with large-scale datasets. Further, with the application of an ensemble method, such as random forest (which will be introduced in the next subsection), the model can perform as accurately as other techniques.

The cost of using the trained tree model to predict classes is fast since the prediction time complexity is logarithmic with respect to the number of data points used to split the dataset in the training process of the algorithm. The decision tree model is also one of the few machine learning techniques that is able to handle both numerical and categorical data. Although in the setting of the experiment in this thesis there are only numerical data attributes, this approach stills has the clear advantage of leaving room for further development in this project by adding non-numeric attributes from GitHub, such as location, employer, etc. Most other techniques for classification are usually specialized to datasets with one type of variable, limiting their extensibility. Decision trees are also able to handle multi-output classification [18], which is also can be of great importance for the genericity of the model and the further development of the model in the future.

A white box model is used in the decision tree model, which makes it easier to generate an analysis of the model itself and it easier to tweak the attributes used to build the decision tree to improve its performance. Each attributes importance can be easily observed in the model and the explanation for classification decisions can be easily inferred from the tree. Conversely, in a black box model environment like in an artificial neural network, the results may be much more difficult to interpret, and improving feature

selection and or transforming the model by combining it with other models to boost the performance are difficult.

With decision trees, it is also easy to validate a model built from a training dataset by applying statistical tests, which is a powerful way to account for the reliability of the model you have built from empirical experiment results. In this research, this property makes it easy to check the correctness of our assumptions. Decision trees also perform well when the training data is incomplete. The model can still have a high chance that it will generate the right output for the data. This is also a reflection of the stability of the prediction performance of the decision tree model. This ability to handle incomplete data is also important for the experiments in this thesis, since we are using GitHub Archive data, which is prone to errors and omissions.

There are also several disadvantages of the decision tree model we used for our experiment, which deserve consideration in the experimentation process and in future work. Firstly, decision trees tend to build over-complicated trees that do not generalize the data well, which is a problem previously mentioned called overfitting. Solutions to this problem range from the pruning methods employed by the C4.5 algorithm to controlling the minimum number of dataset samples needed at each leaf node.

Although decision trees exhibit stable prediction performance, as previously mentioned, the process of building decision trees from the training dataset can be unstable because small changes in the training dataset may cause the process to choose the wrong attribute to branch on or select incorrect values for continuous value attribute partitioning. This is especially important for this project because of the adoption of the GitHub Archive, which is not the official data store of GitHub.

It is also worth noted that the decision tree model is likely to create biased trees if certain classes in the target attributes dominate the training dataset. This potential for certain attributes dominating the dataset and skewing the results is the reason why we need to compute the centrality for every user to avoid the generation of an unbalanced training dataset. It is also important for validation when we used K-Fold and stratified K-

Fold validation will make sure the sample datasets preserve the original distribution of each class in the target attribute.

Random forest

Random forest is a widely used ensemble method [19] for machine learning and is the method we employ in our final data model to avoid the problem of the overfitting. The goal of ensemble methods is to combine multiple models from the same machine learning algorithm in order to improve prediction performance as a whole. Generally, an ensemble model built from multiple models has better robustness than a single model. There are two families of ensemble methods for machine learning: averaging methods and boosting methods.

With averaging methods, multiple independent models are queried for predictions and the final prediction is produced by averaging the multiple independent predictions. It is usually the case that this type of generalized average data model has a much higher accuracy than any of the single data models that is the part of the ensemble because the prediction variance is reduced. With boosting methods, the single data models are built sequentially and each newly built single data model tries to reduce the bias of the all the previously built single data models. This process continues until the last data model is reached. The goal of the methodology is to gradually improve from a single weak data model to a powerful ensemble data model.

Random forest is an averaging method. In random forest, each tree in the ensemble is built from a sample of the training dataset. When partitioning takes place, the split point is computed from a random subset of the features that are selected rather than all of the features. The chosen split point may not be the best split point in the features, but will instead be the best split point in the randomly selected set. Because of the random selection of features, each individual model may be more biased than a decision tree built with consideration of every feature. However, as a result of the randomness that is introduced into the models and the averaging of the predictions, the process decreases the variance of the predictions and the biases generated from each single data model can

be compensated for. In most cases, this compensation overcomes the individual biases to improve classification accuracy.

Centrality value

We used three approaches to compute graph centrality values in this thesis: *betweenness centrality*, *page rank centrality*, and *eigenvector centrality*. Centrality values are a measure of the importance of a node or edge within a graph. The applications of centrality are very diverse. One of the most popular applications of centrality values is to use centrality to find the most influential persons in a social network. This research also centrality to social network analysis but on the GitHub user network. In addition to social network analysis, there are also many other important applications of centrality values, such as finding the key infrastructure nodes in the Internet or locating the main spreaders of disease in an viral epidemic.

Because centrality values were first introduced in the context of social network analysis, most of the terms in centrality originate from sociological studies, such as the betweenness centrality used in this research. The way that centrality is used to find the most important nodes in a network is by defining functions over the nodes of a graph, which compute a real numeric value for each of the node in the graph. Depending on the value computed from the function, we can rank the importance of the nodes according to the computed centrality value. Because the analytical framework is flexible, different aspects of importance can be taken into account when evaluating a node. There are many different definitions of graph centrality that reflects different features or concepts of importance in the graph. Usually, centrality approaches are grouped into two distinguishing families. In the first family, influence is calculated based on the role a node has played in a specific type of network flow or transfer through the network. Centralities are categorized by the type of network flows that are considered important. In the second family, the impact of a node is described by its participation in the cohesiveness of the graph. In both of families, there are many different measures of importance, which is why there are so many variations in how centrality values are calculated.

In the family of centralities built on measures of cohesiveness, the majority of centrality calculations belong to one group, which considers each node's importance in a "walk" through the graph. Different walking structure definitions give different values for the walking distance from one node to another node. For example, degree centrality is based on walks of length 1[20] and eigenvector centrality is based on infinite length walks. For centralities characterized by walking structure, there exist two different categories. One category of centrality focuses on the length of the walks, often labeled as Radial, like degree centrality and eigenvector centrality. Other centrality measures focus on counts of how many walks pass through a given node, such as betweenness centrality.

In this research, the three centrality computation methods that we compared in our experiment are all from the cohesiveness family and consider walking structure in the graph. For representation of centralities utilizing shortest paths, we chose betweenness centrality, which is from the medial walk category. For walking structure length-based centrality, we used eigenvector centrality and page rank centrality. We did not experiment with any centrality values based on walking length one since they don't reflect enough information in the graph.

Betweenness centrality

Betweenness centrality calculates a centrality value for every node and edge in the graph. We only consider node centrality in our experiments, since every node in the GitHub graph represents a user. The centrality computation is straightforward – a shortest path algorithm is run against the graph. For every node, the algorithm keeps track of the number of times the node is part of the shortest path between two other nodes. This algorithm was first developed as a measure to evaluate the impact of specific individuals in the communications of a social network. Nodes that have a high probability of appearing in a randomly chosen shortest path from any two nodes in the graph have a higher betweenness centrality value. The equation for computing betweenness centrality is defined for a node v as follows: for a graph G = (V, E) where V represent the vertices, and E represents the edge [21]:

$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \quad (1)$$

where σ_{st} is the total number of shortest paths from node s to node t and $\sigma_{st}(v)$ defines the number of times v appears on those paths.

Eigenvector centrality

Eigenvector centrality is also a commonly used centrality value to indicate the social impact of a user. It is essentially the eigenvector of the weighted adjacency matrix of the graph A with the largest eigenvalue λ [22]. To be specific, the eigenvector centrality is the solution of:

$$A\,\mathbf{x} = \lambda\,\mathbf{x} \quad (2)$$

where A is the adjacency matrix and λ is the largest eigenvalue.

PageRank centrality

For a graph G, the value of PageRank for each vertex can be given by following iterative relation where $\Gamma^-(v)$ represents the in-neighbors of vertex v, $d^+(u)$ represents the out-degree of the vertex w, with d symbols damping factor, and N symbols the number of the nodes in the graph [23].

$$PR(v) = \frac{1-d}{N} + d \sum_{u \in \Gamma^-(v)} \frac{PR(u)}{d^+(u)} \quad (3)$$

The whole PageRank algorithm is based on the intuition that information on a web page can be ranked based on link popularity. A page should be ranked higher if there are more links to the page. This concept was first introduced in the context of web pages but it is also an applicable algorithm for social graphs. The final result of the algorithm is a probability distribution, in which for each node x in the graph, the PageRank value represents the probability that a randomly selected node would have a link to node x. PageRank centrality values can be computed for a graph of the any size. In the beginning of the computation, the distribution of the PageRank value is evenly divided among all the nodes. Theoretically, the computation process is iteratively applying Page Rank equation

shown above until The PageRank value doesn't change anymore for a vertex in graph. Practically, the function a threshold value is used to stop the iteration. Whenever an iteration produces a PageRank value change below the threshold, the iteration stops and the algorithm return the values computed for the nodes.

Metrics for model evaluation

In this subsection, we will introduce the metrics that we used in our experiments to evaluate the performance of our random forest decision tree data model, and thus validate our assumption that we can use the existing social graph information to find the approximate mapping relationship between the accumulated historical usage data of the user and their social impact in the GitHub social network. Once this model is built, we can use the same model to infer the approximate social impact of future candidates in the GitHub social graph and provide an objective indicator to help recruiters more effectively screen candidates.

Precision, recall and F-measure

We used total four methods of evaluation in our experiments. Three of them are precision, recall, and f-measure. The precision of a test in the experiment is an evaluation of the ability of the data model not to classify an original positive sample as negative. The recall of a test evaluates the ability of the data model to find all the existing positive samples in the test dataset. The computation of the F-measure (F_β measure) is based on the weighted harmonic mean of the precision and recall, which can be seen as a standard for the generalized ability for the data model to find all the positive samples in the test dataset without including significant negative examples. All three of the evaluation methods, including F_β measure with β changing flexibly, range from 0 to 1, where 1 is the best score, and 0 is the worst. The F-measure transforms into F-1 measure when $\beta = 1$, meaning the precision and the recall are deemed equally important in the evaluation.

When using these measures in a binary classification, we use the terms "positive" and "negative" as an indication of the data model's acceptance of a user as a potential

candidate. The terms “true” and “false” are used to label whether the prediction is equal to the expectation (e.g., correct candidates are accepted and incorrect candidates are rejected). With these terms, we define the following relationships:

	Actual result in the test	
Prediction	true positive : Correct result	false positive : Unexpected result
	false negative: Missing result	true negative : Correct absence of result

Table 1: Relationship between the prediction and the actual test result

From this table, we define the following:

$$precision = \frac{true\ positive}{true\ positive + false\ positve} \quad (4)$$

$$recall = \frac{true\ positive}{true\ positive + false\ negative} \quad (5)$$

$$F_\beta = (1 + \beta^2)\frac{precision \times recall}{\beta^2 precision + recall} \quad (6)$$

These equations are of great importance in evaluating a data model’s performance. The precision is a measure of how likely our approach is to correctly accept a candidate, which is directly linked to the trustworthiness of our approach in correctly filtering the candidate pool (e.g., it does not accept candidates that are not sufficiently skilled). The recall is linked directly to the ability of our approach to accept as many fit candidates as possible without introducing unfit candidates (e.g., a measure of not over filtering the list). Finally, the F-1 measure is a generalized measure of our approach’s ability to balance these two tasks.

Logarithmic loss

The fourth evaluation that we apply is logarithmic loss, which is also known as logistic regression loss or cross-entropy loss. Log loss is commonly used to test the performance of (multinomial) logistic regressions and neural networks. It is can also be a good fit to test binary classification with random forest ensemble methods, since the final decision tree can produce multiple results for each decision tree in each test, which will makes it convenient to calculate the probability distributions. Logarithmic loss, itself, is a more objective standard to evaluate the overall performance of the model compared to pure accuracy-based metrics.

The equation for log loss per sample data test in this project can be computed through the equation below, where the target attribute's value is 0 or 1, and probability estimate p is equal to the probability that the target attribute has value 1:

$$L_{log}(y, p) = -\log \Pr(y \mid p) = -y \log p + (1 - y) \log(1 - p) \quad (7)$$

IV. METHODOLOGY AND IMPLEMENTATION

The methodology for implementing the system was as follows. First, we collected every 'Follow' event from GitHub, then we built a graph based on these user to user relationships (e.g. if user A follows user B, then there is an out edge from A to B in the graph). Next, we computed every node's centrality value in the in the graph. Third, we collected the number of each type of GitHub event a user has generated (e.g., measures of different types of activity on GitHub), and transform the data into a training dataset. Finally, we used the random forest decision tree algorithm on our dataset to build a model to aid in predicting a user's graph centrality and whether or not they should be accepted as a job candidate.

The challenges in this project can be divided into two parts. The first challenge was how to collect the right data from GitHub and then convert it into a right format for the algorithm to process and reason on. The second challenge was how to design and conduct experiments from the data to accurately evaluate the random forest decision tree model.

The only way to collect public data from GitHub is to use GitHub's official API to access the GitHub public timeline. However, GitHub's strict limit of 60 requests per second

through the GitHub API made it nearly impossible to collect the data we needed to apply our machine learning algorithms. To stay within their restrictions, it would take months to just collect all the basic profile information of GitHub users need to generate the initial graph. To overcome this limitation, I used the open source GitHub Archive to collect the data. The project streams every event from GitHub's public timeline [24] into a Google BigQuery [25] public dataset [26] on Google Cloud Platform [27], which made it possible to query the table for all needed data.

Because githubachive.org has a record of all the public GitHub timeline events since 2011, one can easily access and query any public GitHub event since 2011. The test data for this thesis spans all public GitHub data from February, 2011 to January, 2015. In order to build the graph of 'Follow' relationships among users, we used the following BigQuery query:

```
//code to collect the graph
select *
from [githubarchive:github.timeline]
where type = 'FollowEvent';
```

To form the final dataset for training, we also need to compute the centrality value for the every user in the dataset. The challenge here is that the dataset is very large and contains 4+ million users and creates a graph with even more edges (e.g., follow relationships between users). To run a network centrality algorithm on this big graph is very time consuming and is one of the problems we are trying to solve with our approach. By training a machine learning model, we only need to run the network centrality algorithm once to train the model and then we can use our model to predict centrality for future users without performing the graph centrality computation again. To speed up the computation for the GitHub social graph, we used the graph analysis library graph-tool [28]. In our experiments, which employed OpenMP [29], we ran the algorithm in parallel on the

multiple cores virtual machine provided by Google Compute Engine to speed up the computation and reduce the running time from two or three weeks to a single day.

The training attributes we selected are the total number of each type of GitHub event a user has generated. There are a total of 25 different event types [30] in GitHub and most of the event counts can be accessed through the GitHub Archive with the following BigQuery query:

```
//code to collect training attributes data
select actor, type, count(type) as events
from [githubarchive:github.timeline]
group each by actor, type;
```

A subset of the users don't have any follow events and don't have any followers, causing them to be disconnected from the graph. For these users, we labeled their centrality value as -1, since they do not impact the overall graph centrality calculations. After collecting of the all the mappings between users and the total number of each event type generated by each user, we still need label users with their centrality values.

When we are done building the graph and the computing the centrality values, we choose the mean centrality value, and the median centrality value of the overall graph respectively in our experiment as a threshold to indicate if a user/node in the graph has an above average social influence in the GitHub community or not. All the users with a centrality value larger than the threshold are labeled as "Accepted" and all others are labeled as "Rejected." The choice of the value for the threshold is flexible and can be adapted to each company's preference on the aggressiveness with which candidates are filtered out. In this research, we chose the mean of all the centrality values as the threshold to examine the feasibility of the method and identify "above average" candidates.

We use random forest decision trees as basis for predicting if a user had a centrality value above the threshold based solely on the counts of the different types of events they generate in GitHub. The implementation we used to build the data model was written in

Python and part of the open source scikit-learn library. After building the random forest decision tree, we can use the model to process future job candidates without recomputing centrality values again – even though the GitHub social graph is changing every day. Theoretically, we should only have to recalculate the centrality values if the social structures and relationships in the GitHub community change substantially, which means the mapping relationship we have generalized from the current network no longer reflect the real-world relationships among the users in the GitHub social graph. However, it is unlikely that this will be the case in the near future since the GitHub community is very mature and unlikely to change radically.

V. Empirical Results

In this section we describe the experiments that we conducted to evaluate our approach. As discussed in the previous sections, there are four metrics that we used to examine the performance of our model, log loss, precision, recall, and f-measure. The main difficulty in setting up the experiment for this project is that there isn't a real-world test dataset of job candidates and sorted hiring preferences to compare against. Our analyses focused on how accurately the model predicted whether or not a particular user was above the chosen centrality threshold and not whether or not a user was an appropriate candidate for a specific job.

To validate the accuracy of the centrality threshold predictions, we use cross-validation on the training dataset. Cross-validation is a technique that uses part of the training dataset to do the training and tests the model with the remainder of the dataset. It is a common model validation technique that indicates how well the data model should function on an independent new data set.

There are many methods for performing cross validation. We chose to evaluate our model with K-fold cross validation. K-fold cross-validation divides the training dataset into K equal parts and uses up to the Kth part as a test dataset while the remaining K-1 parts are used for training. To achieve fairer test results for the whole dataset, we used stratified K-fold to enhance the test process. Stratified K-fold evaluation is done just like the original

K-fold validation, except that the partitioning is done in a manner that preserves the distribution of the class values across each part. We used stratified 5-fold validation for our model. For threshold using the mean of the centrality values, the results for betweenness centrality are shown in Table 2, for PageRank centrality in Table 3, and eigenvector centrality in Table 4. For threshold using the median of the centrality values, the results are respectively shown in Table 5, Table 6 and Table 7. The results were obtained from a Python scikit-learn implementation. The final random forest decision tree data model we built has a 100 single decision tree estimator in the ensemble.

For betweenness centrality with mean value as threshold:

	Result					Mean
Log loss	0.037	0.035	0.034	0.034	0.033	0.035
Precision	0.80	0.73	0.71	0.73	0.74	0.74
Recall	0.46	0.53	0.53	0.53	0.54	0.52
F-1 score	0.58	0.62	0.61	0.62	0.63	0.61

Table 2: Betweenness centrality based model with mean value as threshold

For PageRank centrality with mean value as threshold:

	Result					Mean
Log loss	0.125	0.122	0.122	0.123	0.123	0.123
Precision	0.73	0.65	0.65	0.66	0.67	0.67
Recall	0.28	0.30	0.31	0.30	0.30	0.30
F-1 score	0.40	0.41	0.42	0.42	0.42	0.41

Table 3: PageRank centrality based model with mean value as threshold

For eigenvector centrality with mean value as threshold:

	Result					Mean
Log loss	0.086	0.078	0.078	0.079	0.077	0.080
Precision	0.75	0.64	0.64	0.65	0.64	0.67
Recall	0.19	0.23	0.24	0.23	0.22	0.22
F-1 score	0.30	0.34	0.35	0.34	0.33	0.33

Table 4: Eigenvector centrality based model with mean value as threshold

We can also directly compare them in the chart, which will be more intuitive to do some cross sectional analysis to see the difference:

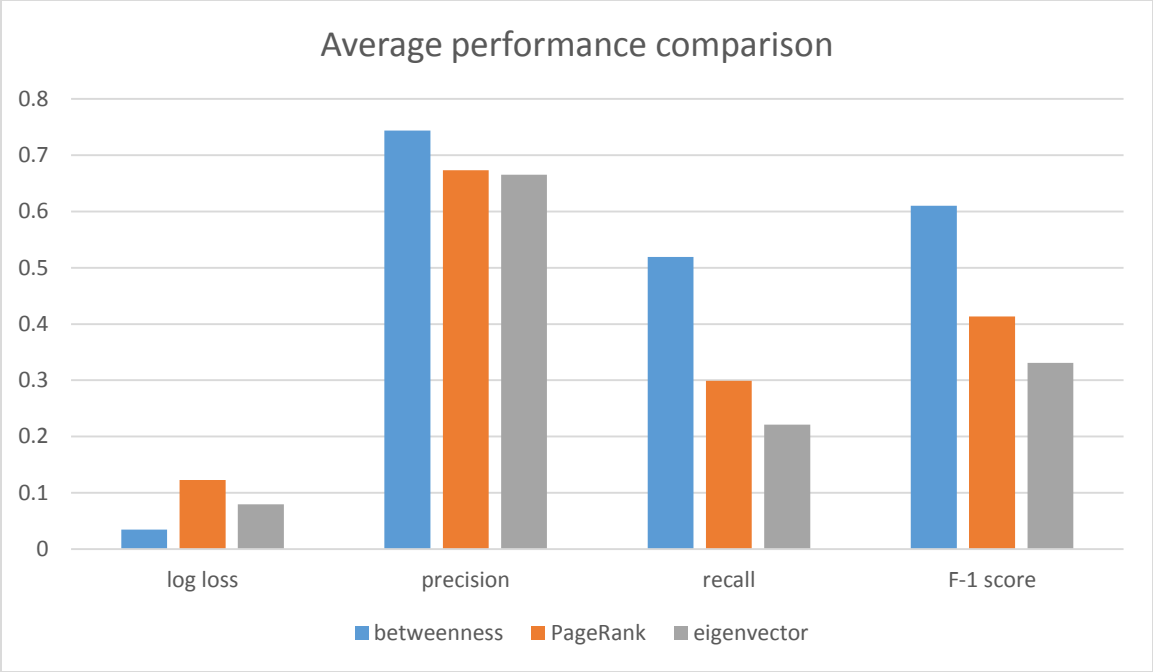

Figure 1: Performance comparison with mean value as threshold

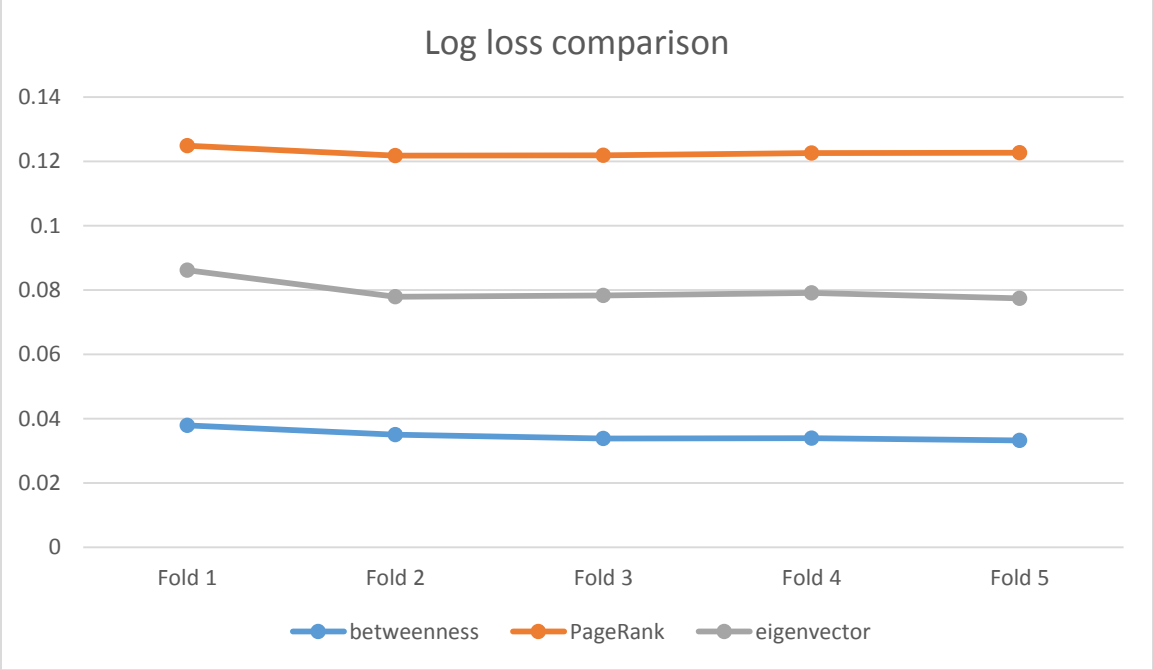

Figure 2: log loss comparison with mean value as threshold

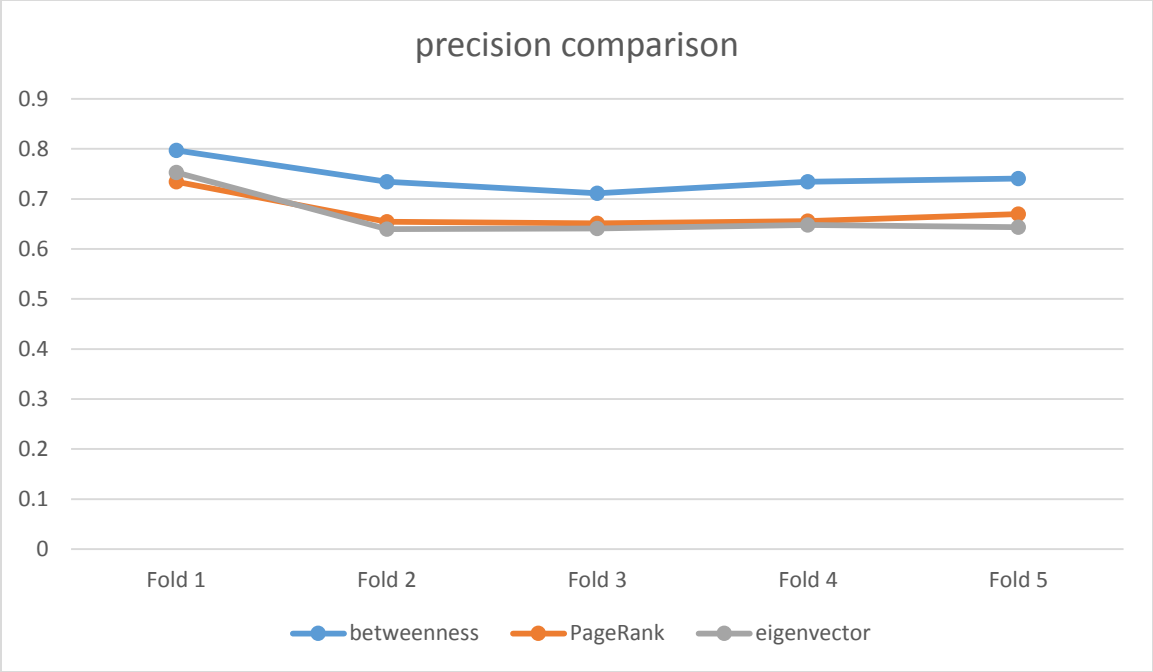

Figure 3: precision comparison with mean value as threshold

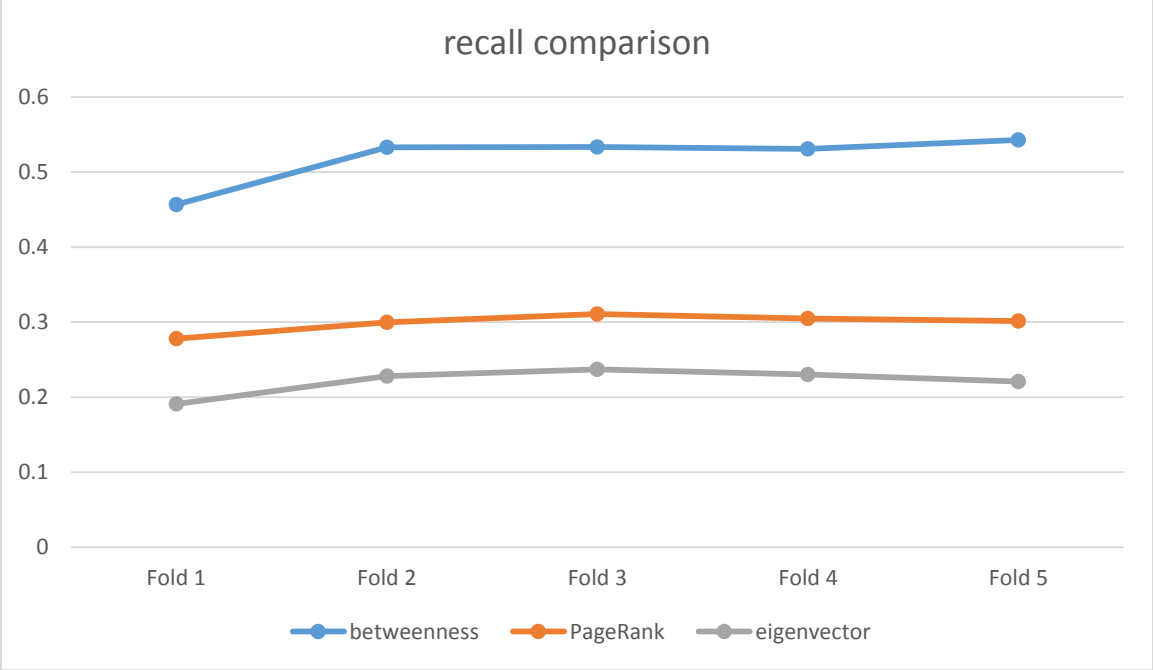

Figure 4: recall comparison with mean value as threshold

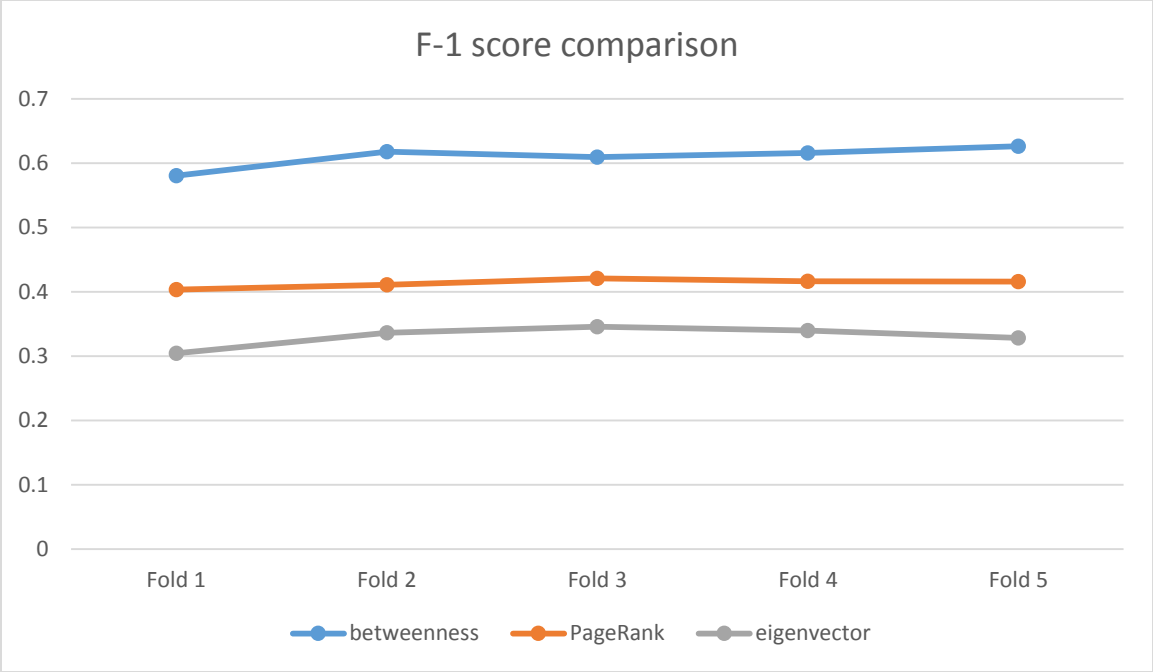

Figure 5: F-1 score comparison with mean value as threshold

For betweenness centrality with median value as threshold:

	Result					Mean
Log loss	0.22	0.23	0.22	0.22	0.22	0.22
Precision	0.97	0.96	0.96	0.96	0.96	0.96
Recall	0.75	0.73	0.74	0.74	0.74	0.74
F-1 score	0.84	0.83	0.84	0.83	0.84	0.84

Table 5: Betweenness centrality based model with median value as threshold

For PageRank centrality with median value as threshold:

	Result					Mean
Log loss	0.25	0.24	0.24	0.24	0.24	0.24
Precision	0.72	0.68	0.67	0.67	0.67	0.68
Recall	0.39	0.44	0.44	0.43	0.43	0.43
F-1 score	0.50	0.53	0.53	0.53	0.52	0.52

Table 6: PageRank centrality based model with median value as threshold

For eigenvector centrality with median value as threshold:

	Result					Mean
Log loss	0.24	0.24	0.23	0.24	0.22	0.24

Precision	0.75	0.70	0.69	0.70	0.71	0.71
Recall	0.41	0.45	0.45	0.44	0.44	0.44
F-1 score	0.53	0.55	0.54	0.54	0.54	0.54

Table 7: Eigenvector centrality based model with median value as threshold

The same kind of charts can also be drew for comparison:

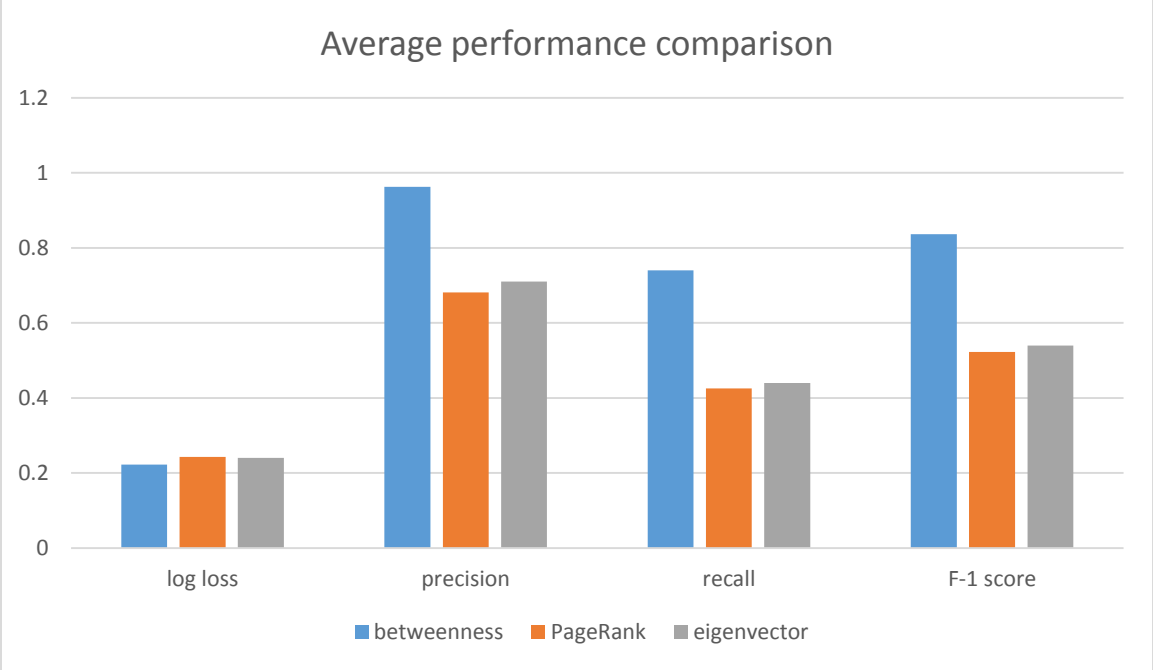

Figure 6: Performance comparison with median value as threshold

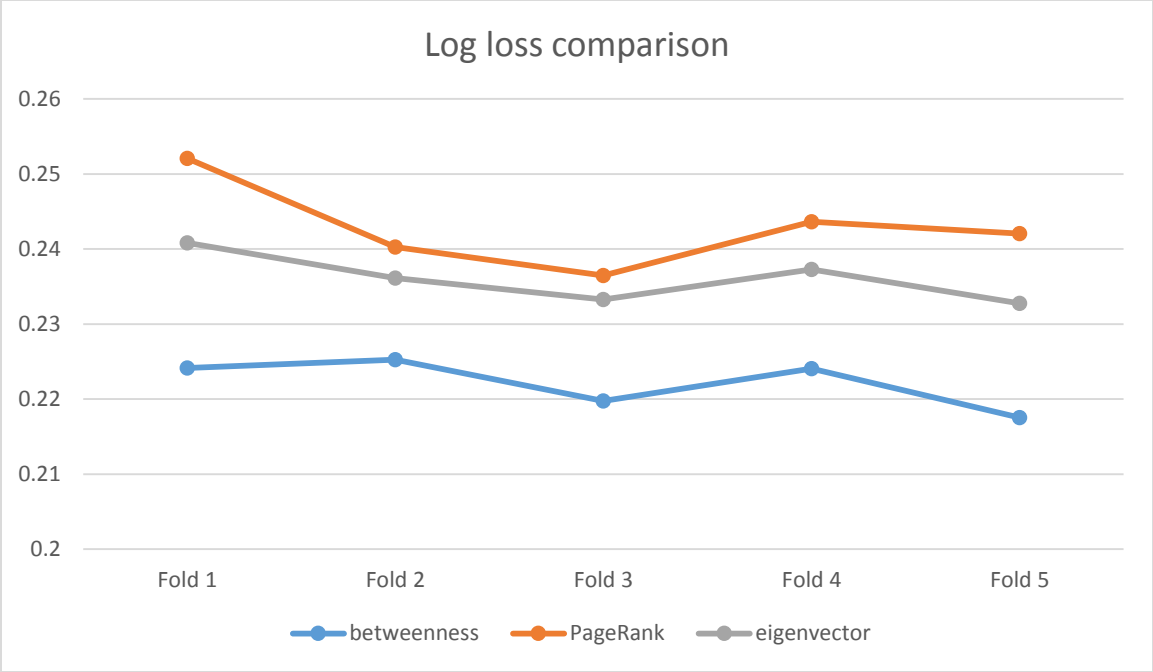

Figure 7: log loss comparison with median value as threshold

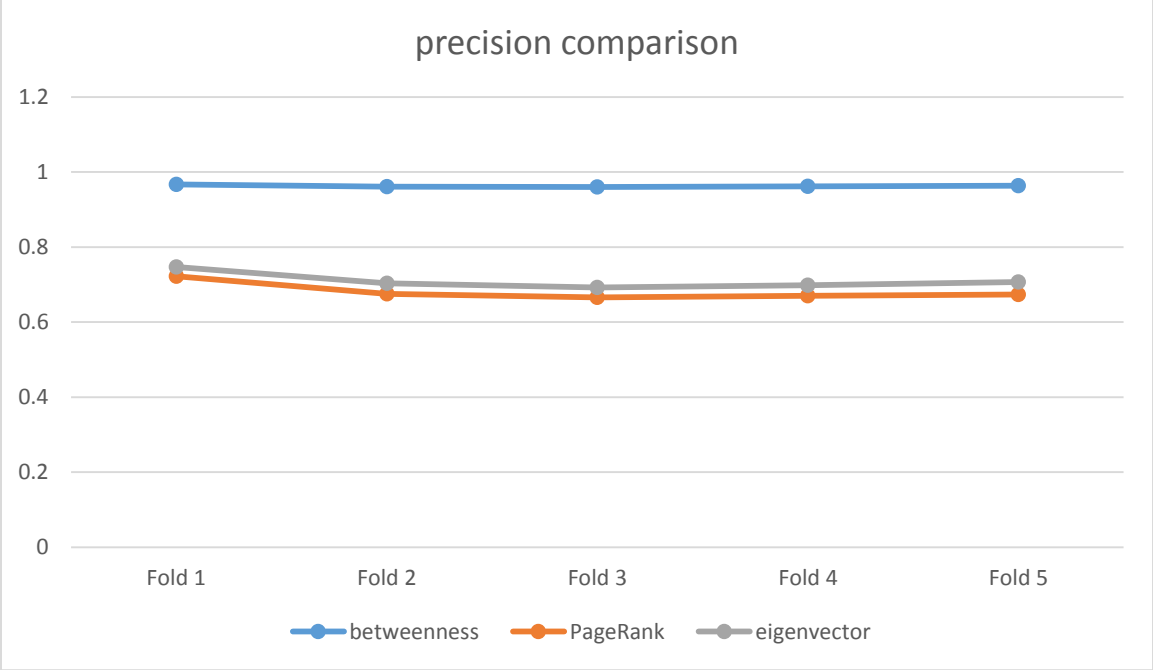

Figure 8: precision comparison with median value as threshold

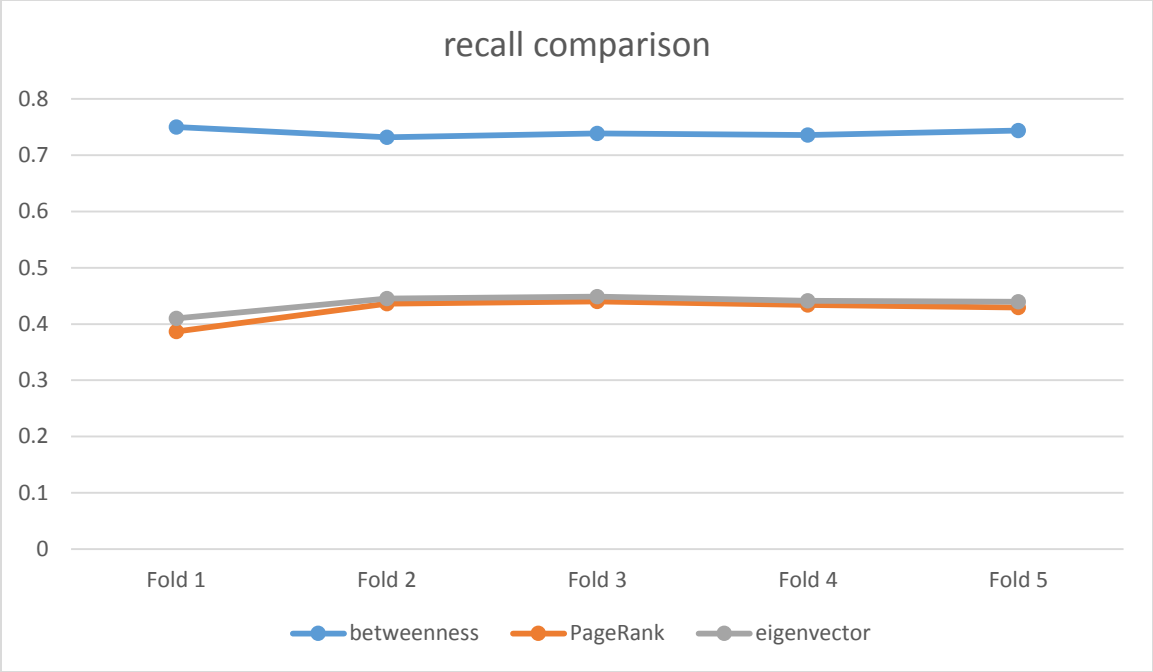

Figure 9: recall comparison with median value as threshold

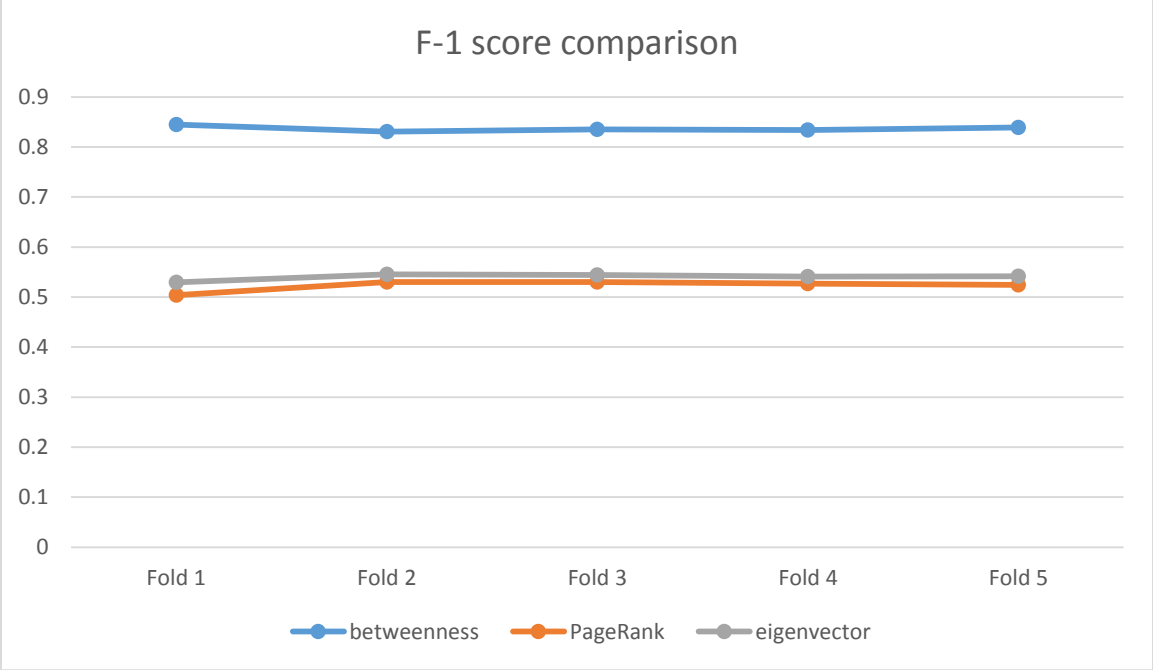

Figure 10: F-1 score comparison with median value as threshold

We can also compare the average performance of the model using mean value as the threshold and median value as threshold vertically for the three centrality computing methods respectively:

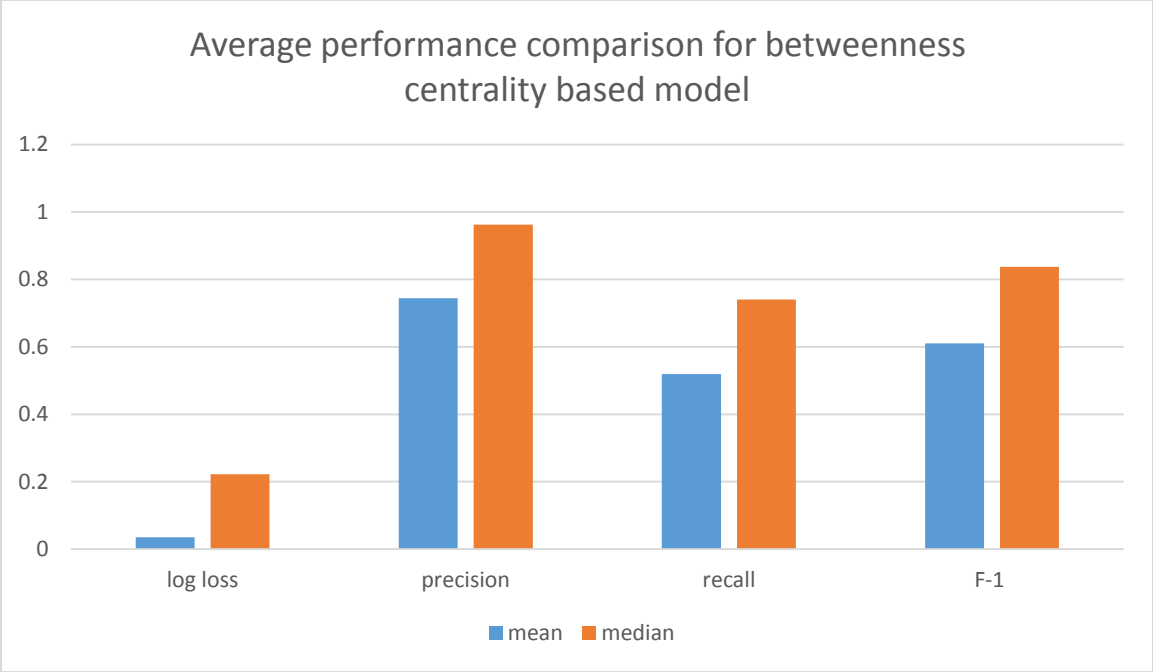

Figure 11: performance comparison for betweenness centrality based model with different threshold

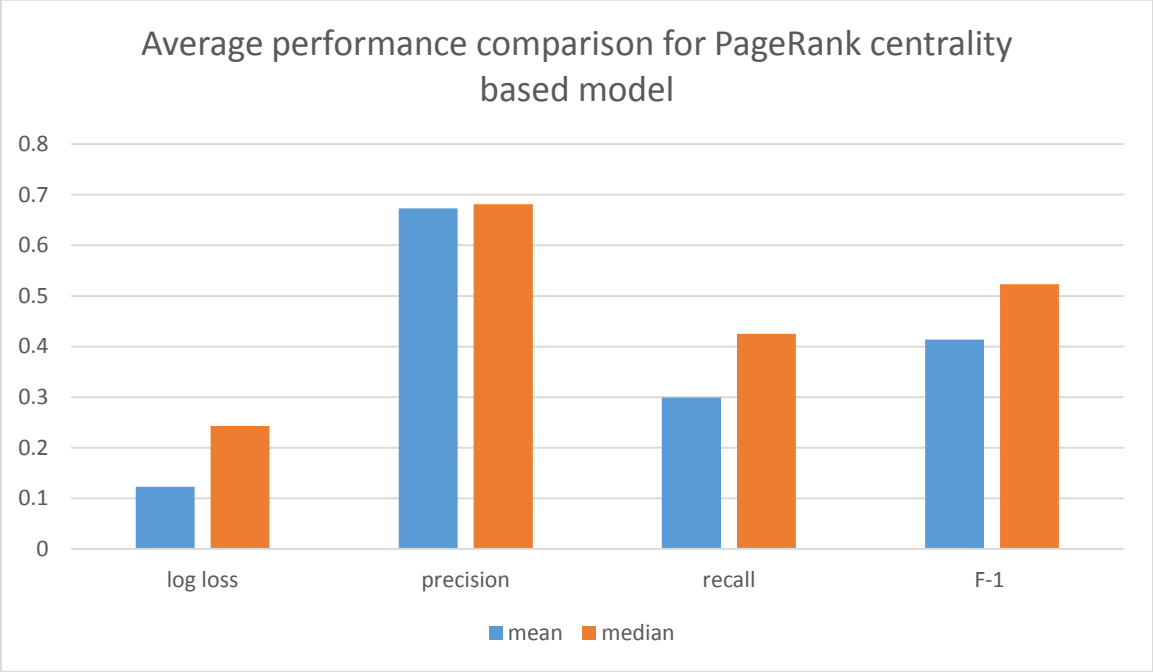

Figure 12: performance comparison for PageRank centrality based model with different threshold

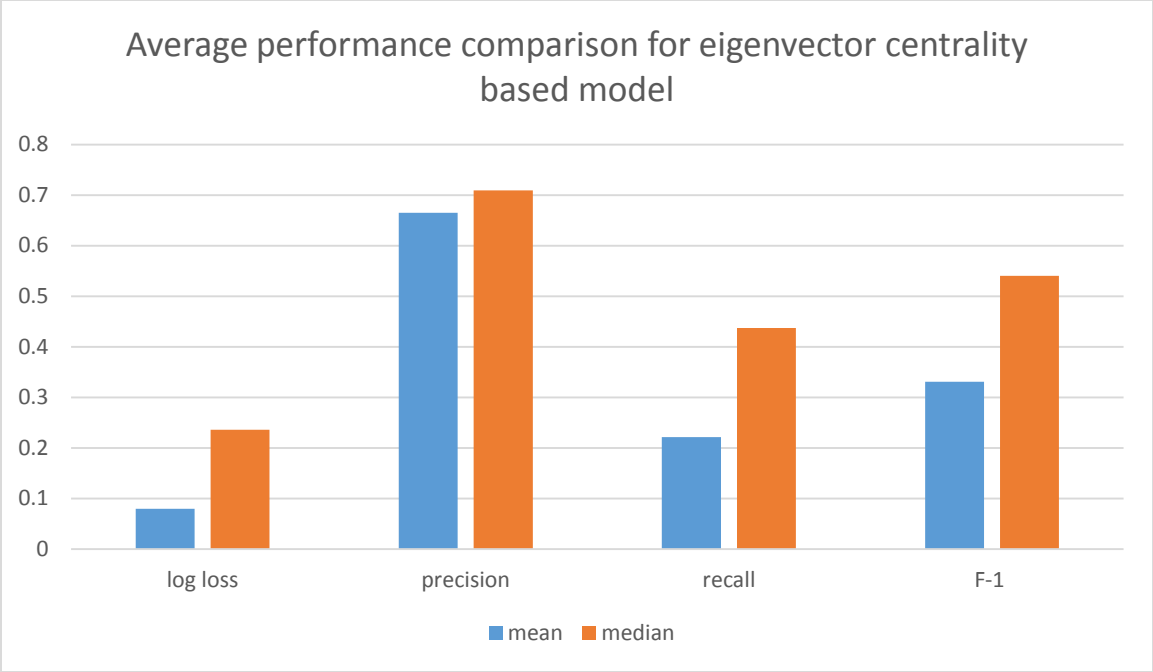

Figure 13: performance comparison for eigenvector centrality based model with different threshold

It is clear that all six models generated by utilizing different centrality computing methods and different thresholding methods don't have a very high log loss, which indicates that they are all accurate models. However, as discussed previously, the precision, recall, and F-1 scores are better evaluators of the model. The precision shows the ability of the model to find highly influential candidates in the GitHub user graph without mixing in the candidates with lower impact. The recall models the ability to discover larger numbers of high quality candidates without introducing poor candidates. The F-1 score is a generalized standard to evaluate how well the data model balances the two metrics.

From these three metrics, we can see some interesting results. First, the betweenness centrality model seems to have much better performance than the two radial based centrality computing methods and has better scores in every one of the three metrics for two thresholding methods. The PageRank centrality and eigenvector centrality seem to not be good approaches for finding influential candidates and not overlooking influential

candidates. However, because we only built 100 trees in our random forest, it is possible that performance might improve with larger numbers of trees.

For betweenness centrality, however, the model shows some real promising value since the precision of the prediction of a candidate's centrality score can reach as high as 80% and has an average 74% accuracy in the mean value threshold method, 97% and 96% in the median threshold method, for predicting correctly whether candidates are above the threshold influence on GitHub. If we use this model in the recruiting process, this model tells you whether the candidate is a programmer with a higher impact on GitHub and is right as much as 96% of the time.

Also, it can be seen clearly in the chart we have built above that the median is a better thresholding methods to choose in the training process. Except for the log loss metrics, the models with median value as threshold all outperform the models with mean value as threshold for three centrality computing methods respectively. A possible explanation is that the distribution of the centrality values for GitHub social graph is very unbalanced, and using mean value as threshold may cause the samples labeled with "Accepted"/"Rejected" too dominant in the training dataset, thus yielding biased tree in the model.

VI. Concluding Remarks and Future Work

To filter and more objectively assess the large number of the candidates in the software engineering recruitment process, we proposed a novel approach in this thesis to preprocess the candidates efficiently using their network centrality value in the GitHub user graph. By experimenting with three different centrality value computation methods: betweenness centrality, PageRank centrality, and eigenvector centrality, and a random forest prediction model, we showed that we could predict if a user's betweenness centrality value was above a median threshold with 96% accuracy. This result supports our assumption that we can use GitHub user graph information to find an approximate relationship between the accumulated historical usage data of a user and their impact on

the GitHub network. Moreover, we can use the same model to infer the approximate impact of future candidates on the GitHub social graph.

There is a lot space for future work for this project. First, we plan to evaluate other centrality value computations to compare their performance. Second, we plan to try more different methods to compute threshold values and compare their impact on overall performance. Finally, there are also plenty of other machine learning algorithms can be tried with this approach that will also be interesting to compare.

REFERENCES

[1] http://www.jobvite.com/press-releases/2012/jobvite-social-recruiting-survey-finds-90-employers-will-use-social-recruiting-2012/

[2] https://www.githubarchive.org/

[3] U. Brandes, “A faster algorithm for betweenness centrality”, Journal of Mathematical Sociology, 2001

[4] https://cloud.google.com/compute/

[5] http://graph-tool.skewed.de/

[6] L. Breiman, J. Friedman, R. Olshen, and C. Stone. Classification and Regression Trees. Wadsworth, Belmont, CA, 1984.

[7] L. Breiman, “Random Forests”, Machine Learning, 45(1), 5-32, 2001

[8] Freeman, Linton (1977). "A set of measures of centrality based on betweenness". Sociometry 40: 35–41.

[9] P. Lawrence, B. Sergey, M. Rajeev, W. Terry, “The pagerank citation ranking: Bringing order to the web”, Technical report, Stanford University, 1998

[10] A. N. Langville, C. D. Meyer, “A Survey of Eigenvector Methods for Web Information Retrieval”, SIAM Review, vol. 47, no. 1, pp. 135-161, 2005

[11] C.M. Bishop (2006). Pattern Recognition and Machine Learning. Springer, p. 209.

[12] Powers, David M W (2007). "Evaluation: From Precision, Recall and F-Factor to ROC, Informedness, Markedness & Correlation". Journal of Machine Learning Technologies 2 (1): 37–63.

[13] Quinlan, J. R. 1986. Induction of Decision Trees. Mach. Learn. 1, 1 (Mar. 1986), 81 - 106

[14] Quinlan, J. R. C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers, 1993.

[15] J. R. Quinlan. Improved use of continuous attributes in C4.5. Journal of Artificial Intelligence Research, 4:77-90, 1996.

[16] T. Hastie, R. Tibshirani and J. Friedman. Elements of Statistical Learning, Springer, 2009

[17] http://scikit-learn.org/stable/

[18] Tsoumakas, Grigorios; Katakis, Ioannis (2007). "Multi-label classification: an overview". International Journal of Data Warehousing & Mining 3 (3): 1–13.

[19] Opitz, D.; Maclin, R. (1999). "Popular ensemble methods: An empirical study". Journal of Artificial Intelligence Research 11: 169–198.

[20] L. C. Freeman, "Centrality in Social Networks: I. Conceptual Clarification." Social Networks, 1, 1979, 215-239.

[21] Brandes, Ulrik (2008). "On variants of shortest-path betweenness centrality and their generic computation". Social Networks 30: 136–145

[22] A. N. Langville, C. D. Meyer, “A Survey of Eigenvector Methods for Web Information Retrieval”, SIAM Review, vol. 47, no. 1, pp. 135-161, 2005

[23] Arasu, A. and Novak, J. and Tomkins, A. and Tomlin, J. (2002). "PageRank computation and the structure of the web: Experiments and algorithms". Proceedings of the Eleventh International World Wide Web Conference, Poster Track. Brisbane, Australia. pp. 107–117.

[24] https://developer.github.com/v3/activity/feeds/

[25] https://cloud.google.com/bigquery/

[26] https://bigquery.cloud.google.com/table/githubarchive:day.events_20150101

[27] https://cloud.google.com/

[28] http://graph-tool.skewed.de/

[29] http://openmp.org/wp/

[30] https://developer.github.com/v3/activity/events/types/