SAGH: A Social Analysis tool for GitHub

CS224w — Project Final Report

Group 15: Akash Das Sarma, Ashish Gupta, Jaeho Shin
Computer Science Department, Stanford University

December 10, 2012

Abstract

In this paper, we model the open source software development community as a heterogeneous social network of projects and developers. In particular, we work with the novel dataset of the GitHub developer community and provide a toolkit for extracting and analyzing a social network from the raw GitHub archives. We propose a developer recommendation system as one interesting application of our comprehensive toolkit. We also propose, implement, and compare different graph algorithms for computing similarity as possible solutions to the recommendation problem. Finally, we provide a generalized experimental framework in which the different algorithms can be compared against different desired input metrics thereby creating an adaptable recommendation tool.

1 Introduction

Problem
The interactions in the open source development community implicitly define a social network over developers and projects. In this work we tackle the problem of providing a framework for, as well as performing, the analysis and characterization of this inherent network for a new and popular development community, GitHub. We build a toolkit to cleanly extract this graph structure and analyze various properties on the resulting social network. This new dataset and the built system allow for the possibility of some interesting applications on development social networks. We implement and test one such application, a recommendation system, and analyze the results.

Motivation
Extracting the underlying social network can provide us with interesting insight into the structure of the development community. We can now use social network metrics and analysis techniques to characterize this domain. This also allows us to build novel graph based applications over the set of developers and projects that were not possible on the previous, relatively unstructured dataset. For instance, our proposed recommendation system application could benefit several employers or recruitment agencies to short-list and rank potential employees from the pool of developers present in the large GitHub development community.

Novelty and Challenges
Our work involves a novel and large dataset that has seen very little past work. As a result, extracting a meaningful, clean and annotated social network is a major challenge. The resulting social network is also heterogeneous and multi-typed with basic entities (nodes) and their relationships (edges) being drawn from a set of different classes. This adds to the difficulty of analyzing and characterizing the network, as well as designing and implementing useful application-oriented algorithms on this rich graph structure.

Contributions and Results
We build a toolkit to automate the graph extraction process. This toolkit uses the git logs to extract information about developers working on individual repositories. It aggregates this information across all repositories, in addition to using other developer-developer relations and project software dependencies to construct our overall graph. We measure various metrics, and general statistics on this graph and use them to analyze and comment on the graph structure.

We motivate a developer recommendation system as one application of our system and this novel dataset. We propose different algorithms to solve this problem and analyze an extension to one of the (SimRank [3]) algorithms that gives the user an intuitive handle to tune the properties of the desired output.

We implement the proposed algorithms and compare their output with respect to two different metrics. We analyze the results and evaluate the different algorithms for these metrics, but our robust system framework allows for the easy implementation and testing of any desired different metric.

2 Related Work

Open Source Development Domain While there is some past work in studying the open source development community, very few people have looked at the GitHub dataset, which is now emerging to be one of the most popular social coding forums. In particular, to our best knowledge, only Yuri et al.[8] work on the GitHub dataset. Their data collection method, however, relies on the GitHub API whose restriction on number of requests severely affects their data collection time and possibility for automation. They also do not look at the underlying social network structure of the dataset, restricting their scope of analysis.

Other works [2, 5, 4, 6, 9] work with datasets other than GitHub, but treat the open source development domain as a social network. They, however, primarily consider the homogeneous networks by considering independent networks with either only developers or only repositories as basic entities, thereby losing a lot of the inherent structure and information present in the original, general heterogeneous graph. In [2], they categorize developers into different categories, such as the core group and peripheral group. In [5], the authors show open source networks to be self organizing collaboration network satisfying preferential attachment and power-law relationships. In [6], the authors show the open source social network to satisfy the small world network model. In book chapter [9], the authors look at the at

four social network on SourceForge and cluster developers into different groups and relate the inclusion of these groups to small world phenomena.

Graph Algorithms Jeh and Widom, [3], explore the problem of computing similarities between nodes of a graph based on purely the structural context, without using any internal information about nodes. They also propose an extension to SimRank base on node popularity, but do not explore this idea in detail. Ioannis et. al., [1], identify certain drawbacks in the original algorithm and propose modifications and extensions to the algorithm.

Work [10] model heterogeneous graphs and use coupled information between different types of nodes to arrive at better rankings than by working on single homogeneous graphs. Zhou et al [10] considers the collaboration and citation network on authors and publications, and propose a modified pageRank algorithm to incorporate information from various multi-typed relationships. Sun et al. [7] consider the problem of clustering on heterogeneous multi-typed graphs and incorporate this with ranking to improve both clustering and ranking accuracies.

3 GitHub Data

GitHub is a repository of Git repositories with additional features for collaboration. Millions of open-source software developers are using it to host their projects' source code and interact with each other through the social features it provides.

3.1 How GitHub differs from others

GitHub is a much richer source of network data compared to SourceForge, or other equivalent open-source hosting service that has existed before. This is mainly because of the decentralized nature of the underlying version control system, Git, making it possible to record all the interactions between developers in a much finer granularity. Rather than restricting all developers to work through a central repository and keeping a single line of history as most of the traditional hosting services had done, GitHub made it so easy for anyone to simply fork the project and evolve on its own. On GitHub, new developers contribute to existing projects by first forking, then coding and sending pull-requests back to the original project. This single change results in a huge difference since it not only makes the latest activities of participating developers more visible, but also records all the traces of interactions among developers both successful ones and failures in a much more fine-grained level.

3.2 GitHub as a Heterogeneous Network

GitHub adds a rich social dimension to the set of Git repositories, each of which itself is already a complex network describing the history of the source tree. GitHub tracks all the forks of a single project, while each project can include other projects as submodules. Developers own, watch, and star multiple projects and forks, and they can create commits, pull-requests, issues, and comments on them. While GitHub allows developers to explicitly follow other developers of their interest, we can derive indirect relationships among them from the traces they left on GitHub and Git repositories. For instance, we could identify collaborators from commit histories, pull-requests, comments on issues, and other records of communication. There are various types of nodes and edges that exist in the GitHub network making it a heterogeneous network. Figure 1 shows the different types of nodes and the edges among them GitHub explicitly exposes via their API. Two most important node types among them are user and repo for developers, and projects respectively. Many other entity types exist to record the traces of interactions among the developers and history of changes in the source tree, e.g., pull-requests, commits. There are many different edge types that (a) directly relate the user and repo nodes which allow us to view it as a bipartite graph, and those that (b) are reflexive, such as follows and forkOf, which can be retained to form a homogeneous subgraph. Other edge types (c) imply indirect relationships between the two important node types, enriching the previous two different views. To apply well-known analysis techniques we can always collapse this heterogeneous network of GitHub into either a bipartite or homogeneous graph.

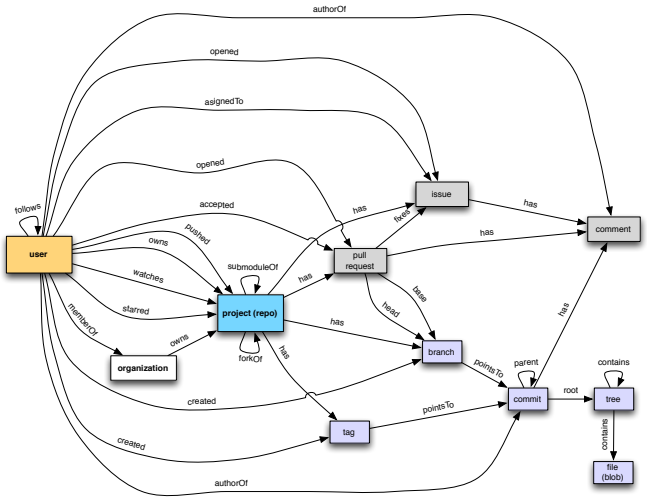

Figure 1: Schema of GitHub's Heterogeneous Network

3.3 Graph Model

We model the heterogeneous graph over the GitHub open-source social network, to infer similar developers from the graph structure. There are three types of nodes and three types of weighted edges we are interested in.

Developer Node (D) People related to a particular project. GitHub allows a developer to fork a parent project, without making actual contributions to the parent project. We only consider developers with commits to the parent project as developers on that project.

Project Node (P) Projects, contains files and has associated developers who contributes to them.

Developer-Project Edges ($D-P$) We form a bipartite graph over developer and project nodes, where there is an edge between a developer D and project P, if D has made contribution to P. Moreover, the weight of $D-P$ edges is the number of commits of D to P (indicating the strength of relation of P with D).

File Node (F) Files, contained in a project, can be changed by multiple commits and hence by many developers. In order to find more fine-grained relationships between developers across different projects, we decided to use this level of information, than relying on

the project level. For a greater data-set, we believe treating project as a basic entity may be more practical and also capture the relationship between developers.

Developer-File Edges ($D-F$) We construct a similar bipartite graph between developers and files (where files are modules and libraries belonging to a single project). File nodes across all projects are considered and there is an edge between developer D and file F if D has worked on F. This graph is used for calculating similarities between developers based on the following assumption. We claim that a single project may consist of different themes and developers may like to work on the files sharing the same themes across different projects, which might help us to identify similar developers better.

Developer-Developer Edges ($D-D$) Basically, this is the type of edge for which we want to infer a weight using various methods. We can think two developers are more similar because they worked on greater number of common files in a project, or just the number of projects in a broader context.

Project Programming Languages GitHub provides a summary of programming languages used in each project. We use this attribute as a coarse evaluation metric for measuring the performance of several different algorithms that find similar developers. Our intuition is that similar developers will tend to have a similar distribution of programming language uses, and vice versa. For each project, a vector of languages with how many number of bytes it's been used is available from GitHub. For developer's language vector, we use the sum of each language's fraction in the project weighted by the his/her commit fraction to it.

3.4 GitHub Data Statistics

We have collected 6,059 repositories from GitHub starting the crawl from `torvalds/linux`, `django/django`, and `textmate/textmate`. From those repositories, we picked a sample of 106, 322 repos having 21914, 32766 developers, and 576994, 2786237 files respectively to test our ideas. From those sample, we have computed several simple statistics, such as the distribution of programming language and the degree distribution in the $D-F$ graph, which are shown in Figure 2 and 3. According to these figures, we can see that the degree distribution follows power-law, and while most projects use only one or two programming languages, most developers contribute to multiple projects that uses a mix of ten different languages. However, one possible flaw of it is that the language information is derived at a project-granularity, so having contributed to a project using multiple languages doesn't necessarily mean the participating developers are fluent in all of them.

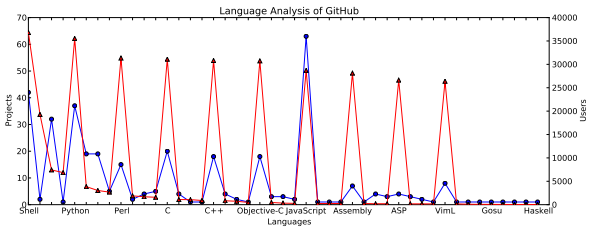

Figure 2: Histogram of Programming Languages

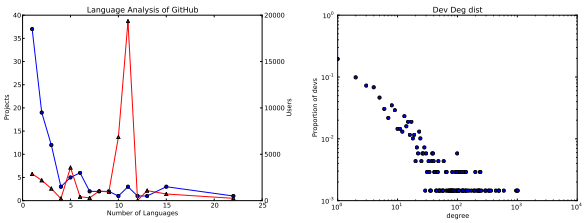

Figure 3: Distribution of Number of Languages and Degrees

4 Developer and Project Recommender System

Expressing the GitHub dataset as an annotated, heterogeneous social network allows us to efficiently infer relationships between entities, that were not apparent from the raw data. It also opens the dataset out to a number of interesting graph algorithms, allowing us to manipulate the data and extract information in new and sophisticated ways. This can be exploited to create a number of interesting applications based on the open source data.

One example of a useful class of applications in this domain is that of a recommendation system. Consider the following search problems:

- Finding a set of experienced and proficient developers to work on a new project, from the point of view of a hiring agency or company.
- Given a developer, finding a set of similar, but novel developers, as a team building problem
- Finding a set of interesting and popular projects to work on from a given domain, from the point of view of a developer
- Finding a set of projects similar to a new/current one, as a training set

In this section, we discuss one of these recommendation problems. Similar algorithms and analyses will hold for the other applications. In particular, we develop an application that takes as input a set of developers and outputs the top-k developers that are similar to the input set.

4.1 Similarity Score

We denote the "similarity" of two developers d_1 and d_2 by $S(d_1,d_2)$. Note that while the SimRank algorithm gives symmetric similarity scores, we also implement an asymmetric extension to SimRank, where we use $S(d_1,d_2)$ to denote the similarity of d_2 *with respect to* d_1. Given sets $O=\{o_1,o_2,\ldots,o_m\}$, $I=\{i_1,i_2,\ldots,i_n\}$, we define $S(O,I)=\sum_{i=1}^{|O|}\sum_{j=1}^{|I|}S(o_i,i_j)$. Now, we need to formalize the notion of similarity of a pair of nodes, $S(a_1,a_2)$.

Intuitively, we want developers who have worked on a lot of similar projects to be similar. Similarly, we expect projects with a large number of common or similar developers, to be similar. Therefore, starting from initial similarity values on developer pairs, we can identify similar projects. Using these similar projects, we can update and improve our developer similarity scores, which, in turn, again feedback into the developer scores. We can repeat this process until the set of pairwise similarity converges to a stable value.

Jeh and Widom identify this mutually recursive nature of similarity scores in [3], and use it to develop algorithms to calculate the similarity ranking and test their algorithms on the citation network. These measures can be combined with other similarity measures, such as text mining, to obtain an overall score. For the scope of this paper, however, we restrict ourselves to this structural similarity measure and treat nodes as black-boxes, thereby using the connectivity of the graph as the only input to the similarity measurement.

4.1.1 SimRank

We define our similarity score between two nodes similar to that in [3]. We use D_i to denote a general developer node, F_i a general project (or file) node, and $N(x)$ to denote the neighbours of node x. Note that $N(x)$ contains the set of developers pointing to (or who have worked on) x, when x is a project node, and $N(x)$ contains the set of projects worked on by x, when x is a developer. Then, we define $S(D_1,D_2)$ and $S(F_1,F_2)$ as:

$$S(D_1,D_2) = \frac{C}{|N(D_1)||N(D_2)|} \sum_{i=1}^{|N(D_1)|} \sum_{j=1}^{|N(D_2)|} S(N_i(D_1),N_j(D_2))$$

where

$$N(D) = \{N_1(D),N_2(D),\ldots,N_{|N(D)|}(D)\}$$

and similarly for $S(F_1,F_2)$.

The similarity scores for all pairs of nodes (within a node set in our developer-file bipartite graph) are calculated by using a fixed-point iteration method. We start off with some initial values for the similarities of all node pairs. These values are then used to recompute similarities for all node pairs in the second iteration, and so on until the process converges. Formally, let $\forall D_1,D_2 : S_i(D_1,D_2)$ be the current computed values. Then, we define S_{i+1} as follows:

$$S_{i+1}(D_1,D_2) = \frac{C}{|N(D_1)||N(D_2)|} \sum_{i=1}^{|N(D_1)|} \sum_{j=1}^{|N(D_2)|} S_i(N_i(D_1),N_j(D_2))$$

(similarly for (F_1,F_2) similarities.

In [3] the authors show that this process converges, with $\forall D_1,D_2 : S_i(D_1,D_2) \rightarrow S(D_1,D_2)$ as $i \rightarrow \infty$. In most cases, however, just a small finite number of iterations are sufficient for the algorithm to converge.

4.2 Popularity Biased SimRank

The bipartite SimRank algorithm in [3] is less effective for nodes with small degrees, as they have little contextual information. To counter this, they propose an extension to the standard algorithm. They introduce a popularity bias, P to remove the negative bias against objects with small degree (developers with a small number of projects). The modified SimRank^P score is given by:

$$S^P(a,b) = S(a,b)|I(b)|^P$$

This P-bias gives a tunable handle to the similarity scores and can be used to adjust the output scores. The choice of P, however, is not intuitive and can not be made by easily. We study the impact of the popularity bias, P in our developer-file domain and try to capture the semantic notion of similarity desired by the user. We propose a

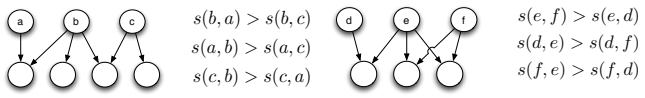

Figure 4: Example Graph1 (left) and Graph 2 (right) with constraints

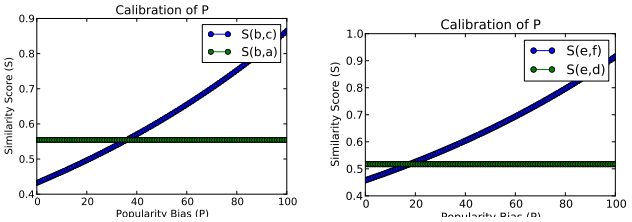

Figure 5: Similarity Scores of a and c with b in Graph1, and d and f with e in Graph2

method by which users can enter constraints that describe their intuitive ideas of structural or semantic similarity, which we then translate into ranges of desired P values.

SimRank scores after the first iteration gives the number of files common to both developers divided by the product of the number of files both developers are involved individually. Further iterations can be treated as improvements on this score. If $P = 1$, then developers working on huge number of files will have higher similarity score compared with any other developer. Hence, we study a range of P values.

In Figure 1, we can observe that developer a has all his files common with b, as compared to c, who has half of his files common with b. Intuitively, we may want/expect a to be more similar than c with respect to b. SimRank with P in the range $[0,0.35]$ gives the desired similarity score. However, P in the range $[0.35,1]$ reverses the order of the computed similarity score.

In Figure 2, we observe that d and f both have all their file common with e. However, since f has more files common with e, we may expect or want higher similarity scores for f with e. This, as can be observed from Figure 2 at $P = 0$, however, is not what the original SimRank algorithm gives. SimRank with P in $[0,0.18]$ gives the opposite order of similarity scores. However, P = $[0.18,1]$ values gives us the desired intuitive similarity order. This example motivates us to provide the user with a tunable popularity knob to reflect their specific notions of similarity in the final results.

Let the popularity range satisfying constraint C with context to graph G be $rangeP(C^G)$. It is easy to see that $rangeP(C_1^{G_1} \wedge C_2^{G_2})$ = $rangeP(C_1^{G_1}) \cap rangeP(C_2^{G_2})$. This allows the range of P-value to be calculated by treating each constraint independently for each graph and taking the intersection of the P ranges over the different constraint-graph pairs at the end.

The above examples show that the P values can capture different notion of similarities desired by the user. We use our above examples to motivate our procedure to calculate desired P values. We could give examples in the form of small fundamental graphs capturing different key, intuitive structural cases (like our two examples) and ask users to input the desired order of similarity between objects in those graphs. We could then use the above procedure to restrict the range of P to satisfy all the constraints across all the examples. In our example, for instance, (from Figure 3 and Figure 4), we conclude that

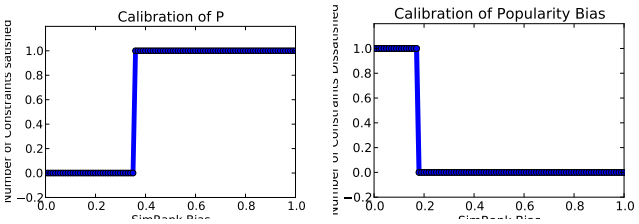

Figure 6: Popularity range satisfying constraints in Graph 1 and 2

$P \in [0.18, 0.35]$ will satisfy the constraints over both the graphs.

This phase can be treated as a training phase to set the P value which can be used later over the entire developer-file graph to compute similarities. In our implementation and analyses, we experiment with different P values, and study their effect on the similarity scores of the developers.

5 System Implementation

5.1 Overall Structure

We designed both our toolkit and repository to be flexible and easy to extend. As shown in Figure 7, It consists of many small scripts that serve different purposes, such as crawling the GitHub network, cloning the git repositories, deriving graphs, computing various statistics, computing SimRank and evaluating the quality of pairs with a different metric. To maximize the value of our effort, we package them into a single distributable executable that can be used by different people to run analyses on different sets of GitHub projects. We tried to modularize the computations by projects, and later combine the partial results for global analyses whenever possible, so that adding, updating or removing a subset of projects would not impact the whole. The current toolkit is up as a GitHub project at `https://github.com/netj/sagh`.

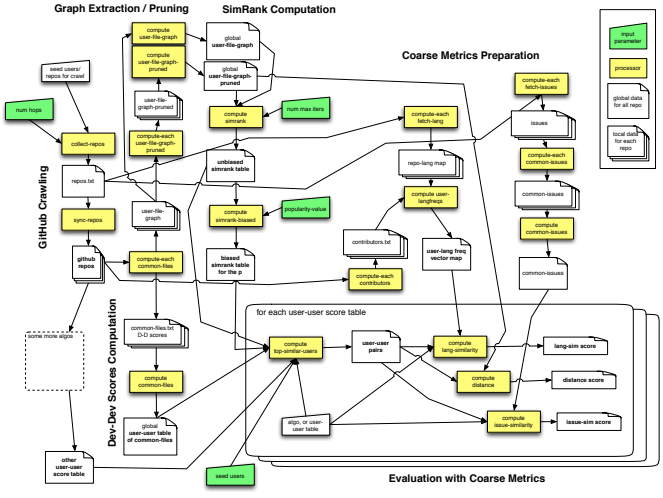

Figure 7: Data-flow in our toolkit for GitHub analysis

5.2 Data Collection

First, we collect the list of repositories in the GitHub network by crawling highly forked repositories of the developers, then fetch more developers from the forks of those repositories, and repeat the previous steps. There is in fact no simple way to download the entire network of GitHub from the outside. Although GitHubArchive.org records all the events happening at GitHub and makes it available to the public, with only a fixed window of time, event information alone is not enough to capture the underlying structure of the social network in GitHub. Furthermore, we found that the official GitHub API has restrictions in many ways and provides insufficient information compared to what is shown to humans. For instance, finding the number of forks of the root repo from the list of repos a particular developer owns is impossible without sending many subsequent API requests. So, we parsed the same web pages we see to collect the list of repositories and then cloned them for further processing.

We derive graphs from the git logs of those local copies, which can be extracted very efficiently once we have cloned them. Instead of collecting the network of the forks for each project via GitHub API, we are approximating the collaboration network of each git to the successful contributions, and trading accuracy for the cost of communication. However, having clones of the actual git repositories enabled us to analyze at a finer granularity (file-level). By spending more GitHub API calls, we could even capture the negative interactions between developers, and the ongoing forks as well.

5.3 Implementation

Parallel SimRank In order to speed up the computation of SimRank, we parallelized the iterations using Python's `multiprocess` library. Computation for each of SimRank's iteration is independent of each other, so they can be parallelized to arbitrary number of processors. However, at the end of each iteration, all partial results must be merged back to a single table for the next iteration. To minimize the transfer of data between the processes, we carefully used the `multiprocess` library to exploit the operating system's copy-on-write behavior for memory management between the forked processes. For instance, while computing the SimRank table for 106-repository sample, although memory used by each of the processes increased up to 18GB, we could manage to run 20 workers on a 192GB main memory machine utilizing most of the cores. Further distributing the computation using MapReduce or Pregel-like framework to reduce the time required to get the SimRank results remains as a future work.

Parallelized Data Processing We have also parallelized most of the independent data collection and processing operations for each repository, e.g., graph extraction and pruning, fetching data via GitHub API accesses. This was an obvious engineering choice we had to make, given the fact that all machines in the InfoLab cluster we used have at least 16 cores and sometimes up to 64. Furthermore, we have intensively used operating system's pipe facilities to streamline the data extraction and transformations between parallel processes where we could.

6 Experiment Design

In this section, we describe the structure of our experiments. The goal of our experiments is to implement and compare different algorithms

for the developer recommendation algorithm against different metrics. Our high-level experiment scheme (see Figure 8) is as follows:

- Pick input developer d_i
- Give d_i as input to algorithms $A_1, A_2, A_3, \ldots, A_n$
- Let the output top-k sets (highest score developers with respect to d_i) from the corresponding algorithms be $O_1, O_2, O_3, \ldots, O_n$
- We compute the score of algorithm A_j with respect to metric M_k as $score_{d_i}^{M_k}(A_j) = \sum_{x \in O_j} M_k(d_i, x)$
- Compare the algorithms with respect to this metric M_k
- Do this for all metrics M_k

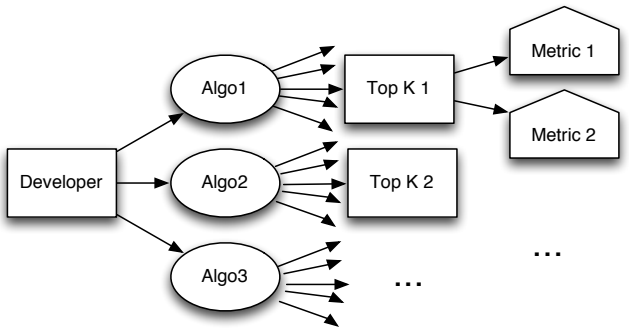

Figure 8: Experiment workflow

6.1 Algorithms

We implement and test the following algorithms for developer recommendations:

1) *DD* algorithm:
Compute $score(d_1, d_2) = |\{f | (d_1, f) \in E \wedge (d_2, f) \in E\}|$. That is, the similarity score of two developers is simply the number of files they have in common. This algorithm returns the top-k developers having highest number of mutual files with the input developer d_i.

2) SimRank
We implement the SimRank algorithm and for developer d_i return the developers with the k highest SimRank scores. We observe that the SimRank scores for two developers at end of the first iteration in the fixed point iteration algorithm described in Section4.1.1 is exactly the scores computed by the *DD* algorithm.

3) Biased SimRank^P
We implement the P-biased version of the SimRank algorithm for different values of P and treat each as a different algorithm.

6.2 Evaluation Metrics

We calculate two metrics on each of the output sets for the different algorithms.

Language Cosine Distance
For this metric, we characterize users in terms of the languages they have worked with. Each user is represented in terms of a vector where each component corresponds to his involvement or experience in that language. Let $\mathbf{L} = (L_1, L_2, \ldots, L_m)$ be the set of all languages. Let d_i be any developer and $\mathbf{P}(d_i)$ be the set of projects that d_i has worked on. Conversely, let $\mathbf{D}(p)$ denote the set of developers that have worked on project p. For project $p \in \mathbf{P}(d_i)$, let $c^p(d_i)$ be the number of commits made by developer d_i. Also, let $V(p) = (l_1^p, l_2^p, \ldots, l_m^p)$ be the fraction of bytes coded in languages $(L_1, L_2, \ldots, L_m)$ respectively. We extract this information using the GitHub API.
Now, we define the language vector $v(\cdot)$ of a developer as:

$$v(d_i) = (v_1, v_2, \ldots, v_m), \text{ where } v'_j = \sum_{p \in \mathbf{P}(d_i)} \frac{c^p(d_i)}{\sum_{d \in \mathbf{D}(p)} c^p(d)} l_j^p$$

and $v_j = \frac{v'_j}{\sum_{k=1}^{m} v_k}$. Intuitively, v_i is a measure of the "experience" a developer has in language L_i. It calculates the fraction of number of commits made by the developer to any particular project weighted by the fraction of bytes in that project coded in L_i, and then sums over all projects that the developer has participated in. We use this vector to characterize any developer and compare different developer-language vectors to compare developers.

We use the cosine distance between two vectors as our metric for language *dissimilarity*, that is, for $v(a) = (a_1, \ldots, a_m)$ and $v(b) = (b_1, \ldots, b_m)$, we have

$$\text{cosine distance}(a, b) = 1 - \frac{\sum_{i=1}^{m} a_i \cdot b_i}{\sqrt{\sum_{i=1}^{m} (a_i)^2} \sqrt{\sum_{i=1}^{m} (b_i)^2}}$$

This standard distance measure captures the dissimilarity between two language vectors, and is the first metric we compare our algorithms on.

Average Distance
In addition to the language-based cosine distance, we also compute the average distance between the input developer and its output top-k set. Here, by average distance, we mean the actual path length from the input developer to a developer in the output top-k in the graph, averaged over all top-k developers. While this distance measure is not directly an intuitive measure of similarity, it serves two purposes: (a) It captures some properties that affect the desirability of an output recommendation. For instance, developers that are too close to the input developer are ones who have already worked on a project with him. From the perspective of new team-building, or novel recommendations, this immediate-neighbour set of developers may not be interesting. Conversely, developers who are too far away from the input developer in the developer-file probably work on projects that are completely unrelated to those of the input developer, albeit having other similarities, such as a similar language vector.

(b) Looking at the average distances of output sets for distance algorithms gives us an idea of the biases present in our different algorithms, in addition to suggesting pruning techniques. For instance, as we shall see in Section 7, the average distances of the output developers given by the SimRank algorithm suggests that it is biased toward users within a constant radius, and gives us support for a pruning technique that can make the algorithm much more efficient.

We now use this experimental framework, to compare the recommendation algorithms over these evaluation metrics.

6.3 Pruning

Due to the large size of the graph, we had to perform extensive pruning to enable efficient computation of the similarity scores by various algorithms.

We do two levels of pruning: File pruning is done first on the subgraph for each repository based on the number of commits per file (at least 50). This removes a number of less prominent files. Then, we perform a second level pruning where we remove developers with exactly one commit per file. We believe that this narrows down the set of developers to those who are actively involved in specific domains, and eliminates developers who make small random contributions.

7 Evaluation

We first test our experimental setup on the graph constructed on a small number of repositories. Here, we work with 13 repositories over 3,825 (569 after pruning) developers, 35,855 (889 after pruning) files. We present our results in the form of two plots, one for each metric: cosine distance and average distance. We use the *DD* algorithm's output as a baseline for the language cosine metric. We seed the graph with 200 randomly chosen developers as our input and plot the average metric score for the top-50 output set corresponding to each input developer as one point in our graph.

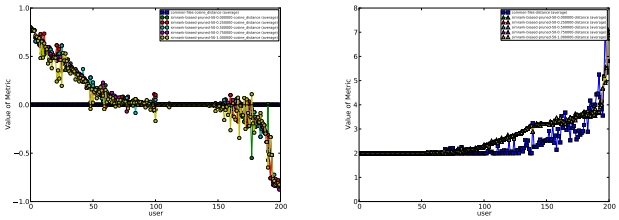

Figure 9: Cosine distance (left) and average distance (right) of the top 50 similar developers for a randomly chosen 200 from the 13 repos sample

We observe that there is no significant difference between the metric scores of the SimRank algorithms over different P values. We also note that the SimRank values give better cosine similarity scores in general (for a larger fraction of input developers) as compared to the *DD* algorithm.

Next, we evaluate the score of the algorithms on our 106 repos sample. After pruning, our 106-repos sample reduces to a graph with 6048 developers and 11605 files. For SimRank computation corresponding to this graph, we used 22 parallel worker processors, and most of the instances converged in about 8 iterations with approximate total time of 6-7 hours.

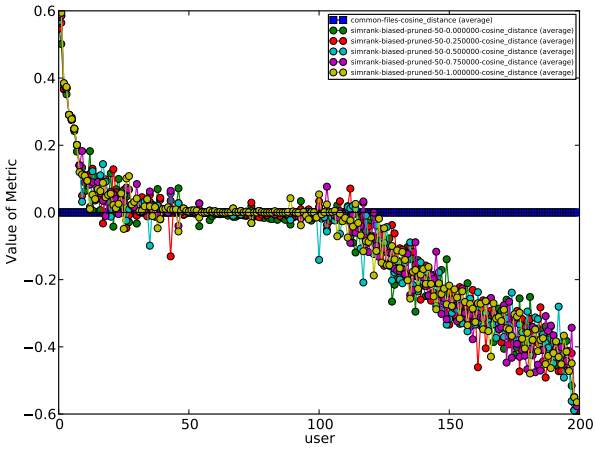

Figure 10: Cosine distance for a randomly chosen set of 200 users (106 repos)

We sort these plots in increasing order of the average cosine metric value of the different SimRank algorithms. Observe that changing the value of P does not significantly affect the values for the metric. We believe that this may be due to the fact that a large fraction of the top-k developers are nodes with small, or single degree, that is, developers who have worked just a small number of files.

In Fig 10, we observe that most of the SimRank and SimRank^P algorithms give similar results for varying values of P.

We also evaluated these metrics for different choices of seed users. 1) We use the highest degree nodes as input to test the following hypotheses:
a) We suspect that the language cosine metric score is inherently biased towards the *DD* algorithm in the GitHub dataset. A large fraction of developers in this domain work in a very small number of projects, and as a result have a very similar language vector to other developers who have also primarily only worked on the same small set of projects. These developers also find their way to the top-k of the *DD* algorithm as they work on a number of common files with the input developer. SimRank on the other hand tries to find hidden transitive similarity between developers, which is not captured by out method of computing the language vector.
b) High degree developers correspond to developers that have worked on a high number of files, which increases their interaction with other developers. This increases the probability of developers in the top-k to have files belonging to the same projects, which, in turn may increase the similarity of the language vector between the top-k developers of the given developer.
2) We also test our algorithms on input sets of low degree developers to test if the converse of the above statement holds true. In other words, we wish to test if low degree nodes inherently have more similar language vectors with developers who have worked on the same (small number of) files, thereby biasing the results in favour of the *DD* algorithm.

When we plot the cosine distance scores for the algorithms, using

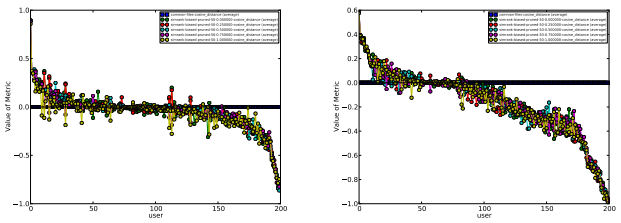

Figure 11: Cosine distance for high (left) and low (right) degree developers from the 106 repos sample

high-degree (well-connected) developers as seed input, we observe that the scores of the various SimRank algorithm marginally improve when compared to the *DD* graph (Fig 11). We also observe that the SimRank algorithms give slightly lower scores with low-degree developers as input. This observations gives support to our above hypotheses.

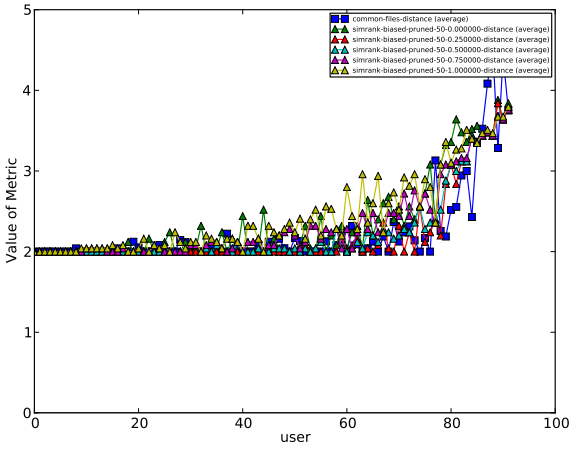

Figure 12: Average distance for a randomly chosen set of 200 users (106 repos)

We observe that the average distance graphs are similar for all input cases, and hence, we only show one plot here (Fig 12). We observe that the average distance metrics for the SimRank algorithms are consistently higher than the *DD* algorithm. This is as expected, since the *DD* algorithm only looks for top-k among the developers who have worked on common files with the input developers (distance 2 in the bipartite graph), only picking further away developers when there are insufficient immediate neighbours to fill the top-k. We observe that the output top-k set for SimRank algorithms generally within distance 5-6 from the input developers. This suggests that we can implement an approximate, efficient version of the SimRank where we assume that all developers far away (say distance > 10) from an input seed can be assumed to have small or zero similarity. This will reduce the space of our computations considerably and allow for a much more efficient implementation without affecting the final output significantly.

8 Future Work

We currently experiment with a bunch of algorithms and evaluate them for developer recommendation system on certain metrics. The above experience leaves us with multiple insights to extend the current work.

- The current SimRank implementation assumes an unweighted graph - we need to extend it to work in our weighted domain.

- Our generalized graph contains developer-developer edges and project-project edges as well. We need to extend the algorithm to work on more general heterogeneous graphs.
  Developer-Developer ($D-D$) Edges: Developers working on the same files indicates indirect collaboration. We can capture direct collaboration, where changes made by developer are accepted by the owner of the project branch. The owner D_2 might reject or accept the changes hence leading to positive or negative collaboration. Developer-Developer edges can also be weighted by performing sentiment analysis on views expressed by a developers on forums.
  Project-Project ($P-P$) Edges: We can weight two projects if they belong to a similar domains. We can consider the number of common shared libraries, across different projects. We need to compare libraries across different languages, and avoid biasing towards more popular languages like Java and C.

- We observe that SimRank essentially returns Top-K nodes within a certain r distance. SimRank needs to be modified to compute the similarity scores within the r distance. This might help us significantly in reducing the computational complexity.

9 Conclusion

We build a toolkit to extract the social network from GitHub data to analyze the open-source software community, and provide support for interesting applications in this domain. We propose recommendation systems as one such application, and build a prototype to suggest similar developers to a given seed input. We implement different algorithms to the solution to this problem, and compare and evaluate the resulting output with respect to two metrics. Our system is adaptable and allows for easy implementation and testing of different combinations of algorithms and metrics. Based on our experiments, we made observations about the used algorithms and metrics and their correlation, as well as used the results to propose efficient approximations to the implemented algorithms.

References

[1] Ioannis Antonellis, Hector Garcia-Molina, and Chi chao Chang. Simrank++: Query rewriting through link analysis of the click graph. Technical Report 2007-32, Stanford InfoLab, 2007.

[2] Shih-Kun Huang and Kang-min Liu. Mining version histories to verify the learning process of legitimate peripheral participants. In *Proceedings of the 2005 international workshop on Mining software repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM.

[3] G. Jeh and J. Widom. Simrank: a measure of structural-context similarity. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 538–543. ACM, 2002.

[4] Luis Lopez-Fernandez, Gregorio Robles, and Jesus M. Gonzalez-Barahona. Applying social network analysis to the information in cvs repositories.

[5] G. Madey, V. Freeh, and R. Tynan. The open source software development phenomenon: An analysis based on social network theory. In *Americas conf. on Information Systems (AMCIS2002)*, pages 1806–1813, 2002.

[6] J. Martinez-Romo, G. Robles, J. Gonzalez-Barahona, and M. Ortuño-Perez. Using social network analysis techniques to study collaboration between a floss community and a company. *Open Source Development, Communities and Quality*, pages 171–186, 2008.

[7] Yizhou Sun, Jiawei Han, Peixiang Zhao, Zhijun Yin, Hong Cheng, and Tianyi Wu. Rankclus: integrating clustering with ranking for heterogeneous information network analysis. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 565–576, New York, NY, USA, 2009. ACM.

[8] Y. Takhteyev and A. Hilts. Investigating the geography of open source software through github.

[9] J. Xu, S. Christley, and G. Madey. *Application of social network analysis to the study of open source software*. Elsevier Press, 2006.

[10] Ding Zhou, Sergey A. Orshanskiy, Hongyuan Zha, and C. Lee Giles. Co-ranking authors and documents in a heterogeneous network. In *Proceedings of the 2007 Seventh IEEE International Conference on Data Mining*, ICDM '07, pages 739–744, Washington, DC, USA, 2007. IEEE Computer Society.