**Empirical Study with Boa:**

To validate the findings of Pingzer et al. and to achieve the research questions stated in the previous section, we propose a study that aims to construct developer-module networks for several large, open-source projects on GitHub. This study aims to answer both research questions posed in the previous section. Predominatly

***Dataset:***

Due to the many inherent limitations in the GitHub API, the study conducted in this project does not directly access project data from GitHub. As a result, this project will use GitHub data collected by the Boa project as the dataset to pull project data from. The reason Boa was selected for use in this study was due to its rigid schema across all projects, flexible Java API, and domain specific language. Further, the current snapshot of the Boa dataset is from September 2015 allowing us access to more recent GitHub projects. While Boa is fairly advantageous, several limitations are discussed in the Future Work section of this paper.

As such, we developed a Boa DSL query called *project search* that allowed us to search its data repository for projects based on the number of contributors, files, and ratio of regular commits to bug fix commits. Further, we developed Boa DSL query called *contribution data* that is able to pull all data required to build the contribution network. Specifically, this query returns a list of all files in the project, a list of contributors, and a list of files changed by each commit. These two scripts allowed us to easily download data that can be used to construct the developer-module networks used in the study.

Due to the sheer size of the Boa dataset and since each Boa query scans all projects regardless of whether a script targets a specific project, we opted to build a local database that is populated with the results of the Boa queries for each project. To illustrate the length of time required for query completion in Boa, the average run time of our contribution query on 10 different projects is around 5 minutes. As such, by developing a local database, pertinent project details are able to be quickly accessed with or without an internet. Further, interesting projects can be cached and used later.

***Methodology:***

To answer both the research questions stated this study had to transform the data obtained from Boa queries into custom developer-module networks for each project. In our network, a standard graph was constructed with files and contributors as actors and commits as ties. As per the definition of these networks, each commit edge is undirected and connects a contributor actor to a file actor. Importantly, since most users tend to commit to files multiple times, the edges are weighted with the number of commits. From the data downloaded from Boa, we could easily develop these graphs by creating the actors and ties by simply analyzing the file, contributor, and commit information.

Since an important part of the study is predicting bug-proneness, when building the graph, we also had to determine a collection of file actors to be bug-prone. Since Boa did not provide access to GitHub issue data, in our Boa query, we opted to determine whether a commit was addressing a bug fix by scanning the commit log for keywords such as “fix” or “revision”. We denote these fixes to be *bug fix commits.* Luckily, as part of the Boa DSL, these bug fix commits could be detected using a function called *isfixingrevision.* By downloading this additional data from Boa, the file actors in the developer-module networks for this project contain a field that stores the number of bug fix commits. From this, we denote bug proneness by dividing the number of bug fix commits by the number of regular commits for each file actor and taking the median of all these values. Any value that falls above the median is considered *bug-prone* whereas any that falls below is considered *non-bug-prone.* With the developer module networks constructed, centrality metrics are computed. For this paper we compute three centrality metrics; betweenness, closeness, and degree centrality. Each of these values are computed for each of the file actors in the network.

Once the developer-module networks were constructed and relevant network centrality metrics were computed, several different operations had to be performed. First, to determine whether the betweenness, closeness, and degree centralities correlated with the number of commits and bug fix commits for each project, we calculate the Spearman correlations between each of metrics for each project. The metrics include each type of centrality as well as the number of commits and bug fix commits. The input into the Spearman correlation function is a list of each of the metrics for each file in the network and the output is a value between -1 and 1 inclusive indication correlation. Once we calculate the Spearman correlation, we perform a Paired T-Test to determine whether the confidence level falls within a threshold of 0.01.

After computing our Spearman correlations for each project, we then perform logistic regression for each project to determine whether centrality metrics can be accurately used to predict

***Software Tool:***

To efficiently carry out the methodology as described in Section 4.1, a software tool called *NetworkMine* was developed that automates the majority of the process. Written in Java, the *NetworkMine* tool is comprised in two different parts; a Boa miner which pulls all software repository details from the Boa website, and a social network builder which constructs the developer-module network and computes all relevant metrics. This division is important since, in future iterations of the tool, different GitHub mining repositories can be swapped for Boa. The remainder of this section will explain the two sections of the tool in detail.

*Boa Miner:*

The Boa miner component is responsible for pulling project data from Boa’s servers into a local SQLite3 database. The tool allows for users to search for projects based on the number of contributors and files and can run the specified search of any of Boa’s three different GitHub datasets. Once a collection of projects is found that match a user’s desired search parameters, the user can download any number of the returned projects. Figure 1 shows a screenshot of the tool allowing users to select projects to download.

The Boa miner component is written in Java and requires the SQLite JDBC driver and Boa API developed by Iowa State.

*Social Network Builder:*

The social network builder component carries out the brunt of the methodology. Using the project repository data downloaded from the Boa miner component, the social network builder constructs a developer-module network for a specified project and then computes the betweenness, closeness, and degree centrality metrics for that graph. Once complete, the user is then able to see the Spearman correlations for each of those social network metrics and the results of the logistic regression run on the graph. Unlike the Boa miner component, this part of the tool is accessed via the command line. The output of the program is a comma-separated values (CSV) file that contains the results of the Spearman correlation and logistic regression functions.

While the tool was developed during this study, it relies on several open-source libraries including Java Universal Network/Graph Framework (JUNG) and Apache Spark. JUNG is used to develop the graph and compute centrality metrics while Apache Spark, while normally used as a cluster computing framework, is used to perform the logistic regression.

**Results:**

This section describes the results that were obtained after running the *NetworkMine* tool across a collection of projects stored in the Boa dataset. The results described here were obtained through meticulously following the methodology described in the previous section. Here, we described the projects that we selected for use, the results of the Spearman correlation analysis, and the logistic regression analysis.

***Selected Projects:***

While the Boa dataset has a large number of GitHub software projects that could be used in this study, we decided to limit our scope by focusing on a subset of these projects. While this project aims to test whether developer-module networks are able to predict failures over a large number of GitHub projects, too many projects would take far too long to compute and also dilute the results presented in this paper. As such, we decided to select fifteen large projects that would form a good representative sample. Fifteen is a good value since it will allow us to test our hypothesis on a variety of different projects to eliminate the possibility of chance while not being too many. Further, while we wanted larger projects with many files and contributors, we decided not to select the top fifteen largest projects from the Boa dataset since many tended to be divergent forks of the Android project. Including projects of a similar nature would result in duplicate results that could spoil the conclusion.

Based on all these considerations, we selected fifteen projects that each had over 150 contributors and 1,000 files that were from a variety of different software domains. Table 1 shows the several statistics pertaining to the project set that was used. The project dataset can be divided into four different domains; Android tools, programming development software, programming frameworks, and other software. Android tools can be defined as Android operating system components or applications developed by members of the Android open-source community. Programming development software is defined as tools that aid programming such as integrated development environments or task trackers. Programming frameworks are protocols, tools, or libraries that can be used during development. Lastly, other software is simply several software projects that do not fit the other three categories. Of these domains, 4 of the projects used are part of the Android tools domain, 4 are programming development software, 4 are programming frameworks. and 3 are miscellaneous software projects.

***Spearman Correlation:***

Similar to Pingzer et al., for each project that we tested, we computed the Spearman correlation between each of the metrics to determine whether there was a correlation between each of these metrics. In it, values below -0.7 and above 0.7 and above are denoted to be strongly correlated components and values below -0.5 and above 0.5 are denoted to be correlated. Table 1 shows the mean results of the Spearman correlation test computed for each of the five metrics that were used in the study. We present the mean across all projects since it is infeasible to display the results from all of the projects. However, each correlation calculated in each project is significant at the 0.01 level. This was calculated through the use of a Paired T-Test.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **# of Commits** | **# of Bug Fixes** | **Betweenness Centrality** | **Closeness Centrality** | **Degree Centrality** |
| **# of Commits** | 1.00 | 0.811 | 0.507 | 0.173 | 0.852 |
|  | **# of Bug Fixes** | 1 | 0.453 | 0.213 | 0.750 |
|  |  | **Betweenness Centrality** | 1 | 0.627 | 0.723 |
|  |  |  | **Closeness Centrality** | 1 | 0.459 |
|  |  |  |  | **Degree Centrality** | 1 |

Despite the mean, some projects correlated strongly with bug fixes and commit numbers for all centrality metrics while others did not. Table 1 shows the difference in correlation values between the Spearman correlations for the *Caf-Platform* and *Dropwizard* projects. Some of the correlations are close to 0 indicating that the values have no correlation at all. This lack of a positive correlation between bug fixes and the centrality metrics indicates that logistic regression may not work well at predicting bug-prone files for this project. Despite this, on average, the number of bug fixes tends to be positively correlated with all three centralities.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **# of Commits** | **# of Bug Fixes** | **Betweenness Centrality** | **Closeness Centrality** | **Degree Centrality** |
| **# of Commits** | 1.00 / 1.00 | 0.927 / 0.680 | 0.802 / 0.251 | 0.870 / -0.281 | 0.971 / 0.737 |
|  | **# of Bug Fixes** | 1.00 / 1.00 | 0.782 / 0.093 | 0.821 / -0.133 | 0.933 / 0.529 |
|  |  | **Betweenness Centrality** | 1.00 / 1.00 | 0.898 / 0.616 | 0.867 / 0.675 |
|  |  |  | **Closeness Centrality** | 1.00 / 1.00 | 0.926 / 0.332 |
|  |  |  |  | **Degree Centrality** | 1.00 / 1.00 |

Based on both tables presented here, there are some interesting conclusions that can be gained. First, not all the projects had failure or commits correlate highly with some of the three centrality metrics. This is especially evident when looking at the closeness centrality metric in Table 1 as its value indicates that it failed to correlate well with the number of bug fixes and commits metrics for all projects. Despite this, the degree centrality of all projects tended to be highly correlated with the commits and bug fixes. Overall, the results from this portion of the study indicate that these centrality metrics tend to be decently correlated with commits and bug fixes.

***Logistic Regression:***

After computing the Spearman correlations, logistic regression for each developer-module network was conducted. In the regression, we set *bug-prone* files with a label of *1* and *non-bug-prone* files with a label of *0*. Each feature label was accompanied by a dense vector of the standardized value of the three centrality values computed for that file. To ensure each project was thoroughly tested, logistic regression was performed 100 times for each project and 60% of the files in the graph were randomly divided into the training set and 40% into the testing set. In the output from the logistic regression test, was a collection of file labels with a predicted probability score between -1 and 1. We selected 0 as the cutoff point in the output score that divides *non-bug-prone* and *bug-prone.*

Table 1 shows several descriptive metrics for the results of the logistic regression for the top four and bottom four projects. In it, the number of iterations for the logistical regression for each project displayed has been compressed through calculations of the mean and median. Further, the table presents the mean and median precision and recall across all iterations in all projects.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Project** | **Stat** | **Precision** | **Recall** | **Project** | **Stat** | **Precision** | **Recall** |
|  | Mean |  |  |  | Mean |  |  |
| Median |  |  | Median |  |  |
|  | Mean |  |  |  | Mean |  |  |
| Median |  |  | Median |  |  |
|  | Mean |  |  |  | Mean |  |  |
| Median |  |  | Median |  |  |
|  | Mean |  |  |  | Mean |  |  |
| Median |  |  | Median |  |  |
| **All Projects** | Mean | 0.756 | 0.765 |
| Median | 0.730 | 0.781 |

The results obtained from the use of logistic regression are encouraging. While several projects in the bottom four tend to have either very low precision or recall, the projects with the top precision and recall values indicate that centrality metrics can possibly be used as valid bug predictors. For instance, since the <project> project has a precision and recall values of over 80%, there are few false positives and negatives being included in the predictions. Further, many of the projects tested have higher recall values compared to precision. This is desirable in the scope of failure prediction since having false positives included in the bug predictions is less detrimental than having false negatives. For example, if these predictions are used to decide which files to closely inspect files for bugs, false positives will simply mean that more files will need to be inspected. On the other hand, false negatives will mean that certain bug-prone files will miss the closer inspection.

Figure 1 shows a scatterplot of the precision and recall values for the *Caf-Platform* project. Since most of the runs of the logistic regression function present similar precision and recall values, the standard deviation of the data is low. This further validates the results as it demonstrates that, even with different splits of the input graph, the logistic regression function is able to predict the precision and recall fairly well.