Empir Software Eng (2017) 22:3219–3253
DOI 10.1007/s10664-017-9512-6

Curating GitHub for engineered software projects

Nuthan Munaiah[1] · Steven Kroh[1] · Craig Cabrey[1] · Meiyappan Nagappan[2]

Published online: 18 April 2017

Abstract Software forges like GitHub host millions of repositories. Software engineering researchers have been able to take advantage of such a large corpora of potential study subjects with the help of tools like GHTorrent and Boa. However, the simplicity in querying comes with a caveat: there are limited means of separating the signal (e.g. repositories containing engineered software projects) from the noise (e.g. repositories containing home work assignments). The proportion of noise in a random sample of repositories could skew the study and may lead to researchers reaching unrealistic, potentially inaccurate, conclusions. We argue that it is imperative to have the ability to sieve out the noise in such large repository forges. We propose a framework, and present a reference implementation of the framework as a tool called `reaper`, to enable researchers to select GitHub repositories that contain evidence of an engineered software project. We identify software engineering practices (called dimensions) and propose means for validating their existence in a GitHub repository. We used `reaper` to measure the dimensions of 1,857,423 GitHub repositories. We then used manually classified data sets of repositories to train classifiers capable of

Communicated by: Denys Poshyvanyk

✉ Nuthan Munaiah
nm6061@rit.edu

Steven Kroh
skk8768@rit.edu

Craig Cabrey
cac2573@rit.edu

Meiyappan Nagappan
mei.nagappan@uwaterloo.ca

[1] Department of Software Engineering, Rochester Institute of Technology, 134 Lomb Memorial Drive, Rochester, NY 14623, USA

[2] David R. Cheriton School of Computer Science, University of Waterloo, 200 University Avenue West, Waterloo, Ontario N2L 3G1, Canada

predicting if a given GitHub repository contains an engineered software project. The performance of the classifiers was evaluated using a set of 200 repositories with known ground truth classification. We also compared the performance of the classifiers to other approaches to classification (e.g. number of GitHub Stargazers) and found our classifiers to outperform existing approaches. We found stargazers-based classifier (with 10 as the threshold for number of stargazers) to exhibit high precision (97%) but an inversely proportional recall (32%). On the other hand, our best classifier exhibited a high precision (82%) and a high recall (86%). The stargazer-based criteria offers precision but fails to recall a significant portion of the population.

Keywords Mining software repositories · GitHub · Data curation · Curation tools

1 Introduction

Software repositories contain a wealth of information about the code, people, and processes that go into the development of a software product. Retrospective analysis of these software repositories can yield valuable insights into the evolution and growth of the software products contained within. We can trace such analysis all the way back to the 1970s, when Belady and Lehman (1976) proposed Lehman's Laws of software evolution. Today, the field is significantly invested in retrospective analysis with the Boa project (Dyer et al. 2013) receiving more than $1.4 million to support such analysis.[1]

The insights gained through retrospective analysis can affect the decision-making process in a project, and improve the quality of the software system being developed. An example of this can be seen in the recommendations made by Bird et al. (2011) in their study regarding the effects of code ownership on the quality of software systems. The authors suggest that quality assurance efforts should focus on those components with many minor contributors.

The richness of the data and the potential insights that it represents were the enabling factors in the inception of an entire field of research: Mining Software Repositories (MSR). In the early days of mining software repositories, researchers had limited access to software repositories, which were primarily hosted within organizations. However, with the proliferation of open access software repositories such as GitHub, Bitbucket, SourceForge, and CodePlex for source code and Bugzilla, Mantis, and Trac for bugs, researchers now have an abundance of data from which to mine and draw interesting conclusions.

Every source code commit contains a wealth of information that can be used to gain an understanding of the art of software development. For example, Eick et al. (2001) dived into the rich (15+ year) commit history of a large telephone switching system in order to explore the idea of code decay. Modern day source code repositories provide features that make managing a software project as seamless as possible. While the integration of features provides improved traceability for developers and project managers, it also provides researchers with a single, self-contained, organized, and more importantly, publicly-accessible source of information from which to mine. However, anyone may create a repository for any purpose at no cost. Therefore, the quality of information contained within the forges may be diminishing with the addition of many noisy repositories e.g. repositories containing home work assignments, text files, images, or worse, the backup of a desktop computer.

[1]National Science Foundation (NSF) Grant CNS-1513263.

Kalliamvakou et al. (2014) identified this noise as one of the nine perils to be aware of when mining GitHub data for software engineering research. The situation is compounded by the sheer volume of repositories contained in these forges. As of June, 2016, GitHub alone hosted over 38 million repositories[2] and this number is rapidly increasing.

Researchers have used various criteria to slice the mammoth software forges into data sets manageable for their studies. For example, researchers have leveraged simple filters such as popularity to remove noisy repositories. Filters like popularity (measured as number of watchers or stargazers on GitHub, for example) are merely proxies and may neither be general-purpose nor representative of an engineered software project. The intuition in using popularity as a filter is that popularity may be assumed to be correlated with quality. However, as Sajnani et al. (2014) have shown, popularity is not correlated with quality, questioning the utility of using popularity-based filters. We believe that researchers mining software repositories should not have to reinvent filters to eliminate unwanted repositories. There are a few examples of research that take the approach of developing their own filters in order to procure a data set to analyze:

- In a study of the relationship between programming languages and code quality, Ray et al. (2014) selected 50 most popular (measured by the number of *stars*) repositories in each of the 19 most popular languages.
- Bissyandé et al. (2013b) chose the first 100,000 repositories returned by the GitHub API in their study of the popularity, interoperability, and impact of programming languages.
- Allamanis and Sutton (2013) chose 14,807 Java repositories with at least one fork in their study of applying language modeling to mining source code repositories.

The project web sites for GHTorrent (2016a) and Boa (Iowa State University 2016) list more papers that employ different filtering schemes.

The assumption that one could make is that the repositories sampled in these studies contain engineered software projects. However, source code forges are rife with repositories that do not contain source code, let alone an engineered software project. Kalliamvakou et al. (2014) manually sampled 434 repositories from GitHub and found that only 63.4% (275) of them were for software development; the remaining 159 repositories were used for experimental, storage, or academic purposes, or were empty or no longer accessible. The inclusion of repositories containing such non-software artifacts in studies targeting software projects could lead to conclusions that may not be applicable to software engineering at large. At the same time, selecting a sample by manual investigation is not feasible given the sheer volume of repositories hosted by these source code forges.

The goal of our work is to identify practices that an engineered software project would typically exhibit with the intention of developing a generalizable framework with which to identify such projects in the real-world.

The contributions of our work are:

- A generalizable evaluation framework defined on a set of dimensions that encapsulate typical software engineering practices;
- A reference implementation of the evaluation framework, called `reaper`, available as an open-source project (Munaiah et al. 2016b);
- A publicly-accessible data set of dimensions obtained from 1,857,423 GitHub repositories (Munaiah et al. 2016a).

[2] https://github.com/about/press.

The remainder of this paper is organized as follows: we begin by introducing the notion of an engineered software project in Section 2. We then propose an evaluation framework in Section 2.1 that aims to operationalize the definition of an engineered software project along a set of dimensions. We describe the various sources of data used in our study in Section 3. In Section 4, we introduce the seven dimensions used to represent a repository in our study. In Section 5, we propose two variations to the definition of an engineered software project, collect a set of repositories that conform to the definitions, present approaches to build classifiers capable of identifying other repositories that conform to the definition of an engineered software project. The results from validating the classifiers and using them to identify repositories that conform to a particular definition of an engineered software project from a sample of 1,857,423 GitHub repositories is presented in Section 6. We contrast our study with prior literature in Section 7, discuss prior and potential research scenarios in which the data set and the classifier could be used in Section 8, and discuss nuances of certain repositories in Section 9. We address threats to validity in Section 10 and conclude the paper with Section 11.

2 Engineered Software Project

Laplante (2007) defines software engineering as "a systematic approach to the analysis, design, assessment, implementation, test, maintenance and reengineering of software." A software project may be regarded as "engineered" if there is discernible evidence of the application of software engineering principles such as design, test, maintenance, etc. On similar lines, we define an engineered software project in Definition 1.

Definition 1 An *engineered software project* is a software project that leverages sound software engineering practices in one or more of its dimensions such as documentation, testing, and project management.

Definition 1 is intentionally abstract; the definition may be customized to align with a set of different, yet relevant, concerns. For instance, a study concerned with the extent of testing in software projects could define an engineered software project as a software project that leverages sound software testing practices. In our study, we have customized the definition of an engineered software project in two ways: (a) an engineered software project is similar to the projects contained within repositories owned by popular software engineering organizations such as Amazon, Apache, Microsoft and Mozilla and (b) an engineered software project is similar to the projects that have a general-purpose utility to users other than the developers themselves. We elaborate on these two definitions in the Implementation Section (Section 5). We note that the test for similarity in both these definitions is achieved using a set of quantifiable metrics described in the Dimensions Section (Section 4).

2.1 Evaluation Framework

In order to operationalize Definition 1, we need to (a) identify the essential software engineering practices that are employed in the development and maintenance of a typical software project and (b) propose means of quantifying the evidence of their use in a given software project. The *evaluation framework* is our attempt at achieving this goal.

The evaluation framework, in its simplest form, is a boolean-valued function defined as a piece-wise function shown in (1).

$$f(r) = \begin{cases} true & \text{If repository } r \text{ contains an engineered software project} \\ false & \text{Otherwise} \end{cases} \quad (1)$$

The evaluation framework makes no assumption of the implementation of the boolean-valued function, $f(r)$. In our implementation of the evaluation framework, we have chosen to realize $f(r)$ in two ways: (a) $f(r)$ as a score-based classifier and (b) $f(r)$ as a Random Forest classifier. In both approaches, the implementation of the function, $f(r)$, is achieved by expressing the repository, r, using a set of quantifiable attributes (called dimensions) that we believe are essential in reasoning that a repository contains an engineered software project.

3 Data Sources

In this section, we describe two primary sources of data used in our study. We note that our study is restricted to publicly-accessible repositories available on GitHub and the data sources described in the subsections that follow are in the context of GitHub.

3.1 Metadata

GitHub metadata contains a wealth of information with which we could describe several phenomena surrounding a source code repository. For example, some of the important pieces of metadata are the primary language of implementation in a repository and the commits made by developers to a repository.

GitHub provides a REST API (GitHub Inc 2016a) with which GitHub metadata may be obtained over the Internet. There are several services that capture and publish this metadata in bulk, avoiding the latency of the official API. The GitHub Archive project (GitHub Inc 2016b) was created for this purpose. It stores public events from the GitHub timeline and publishes them via Google BigQuery. Google BigQuery is a hosted querying engine that supports SQL-like constructs for querying large data sets. However, accessing the GitHub Archive data set via BigQuery incurs a cost per terabyte of data processed.

Fortunately, Gousios (2013) has a free solution via their GHTorrent Project. The GHTorrent project provides a scalable and queryable offline mirror of all Git and GitHub metadata available through the GitHub REST API. The GHTorrent project is similar to the GitHub Archive project in that both start with the GitHub's public events timeline. While the GitHub Archive project simply records the details of a GitHub event, the GHTorrent project exhaustively retrieves the contents of the event and stores them in a relational database. Furthermore, the GHTorrent data sets are available for download, either as incremental MongoDB dumps or a single MySQL dump, allowing offline access to the metadata. We have chosen to use the MySQL dump which was downloaded and restored on to a local server. In the remainder of the paper, whenever we use the term database we are referring to the GHTorrent database.

The database dump used in this study was released on April 1, 2015. The database dump contained metadata for 16,331,225 GitHub repositories. In this study, we restrict ourselves to repositories in which the primary language is one of Java, Python, PHP, Ruby, C++, C, or C#. Furthermore, we do not consider repositories that have been marked as deleted and those that are forks of other repositories. Deleted repositories restrict the amount of data available

for the analysis while forked repositories can artificially inflate the results by introducing near duplicates into the sample. With these restrictions applied, the size of our sample is reduced to 2,247,526 repositories.

An inherent limitation of the database is the staleness of data. There may be repositories in the database that no longer exist on GitHub as they may have been deleted, renamed, made private, or blocked by GitHub. As a result, we only consider the repositories that were active at the time our analysis was run.

3.2 Source Code

In addition to the metadata about a repository, the code contained within is an important source of information about the project. Developers typically interact with their repositories using either the `git` client or the GitHub web interface. Developers may also use the GitHub REST API to programmatically interact with GitHub.

We use GitHub to obtain a copy of the source code for each repository. We cannot use GitHub's REST API to retrieve repository snapshots, as the API internally uses the `git archive` command to create those snapshots. As a result, the snapshots may not include files the developers may have marked irrelevant to an end user (such as unit test files). Since we wanted to examine all development files in our analysis, we used the `git clone` command instead to ensure all files are downloaded.

As mentioned earlier, the metadata used in this study is current as of April 1, 2015. However, this metadata may not be consistent with a repository cloned after April 1, 2015, as the repository contributors may have made commits after that date. In order to synchronize the repository with the metadata, we reset the state of the repository to a past date. For each evaluated repository in the database, we retrieved the `date` of the most recent commit to the repository. We then identified the `SHA` of the last commit made to the repository before the end of the day identified by `date` using the command `git log -1 --before="{date} 11:59:59"`. For repositories with no commits recorded in the database, we used the date when the GHTorrent metadata dump was released i.e. 2015-04-01. With the appropriate commit `SHA` identified, the state of the cloned repository was reset using the command `git reset --hard {SHA}`.

4 Dimensions

In this section, we describe the dimensions used to represent a repository in the context of the evaluation framework introduced earlier. In our study, a repository is represented using a set of seven dimensions, they are:

1. *Community*, as evidence of collaboration.
2. *Continuous integration*, as evidence of quality.
3. *Documentation*, as evidence of maintainability.
4. *History*, as evidence of sustained evolution.
5. *Issues*, as evidence of project management.
6. *License*, as evidence of accountability.
7. *Unit testing*, as evidence of quality.

In the selection of dimensions, while relevance to software engineering practices was paramount, we also had to consider aspects such as implementation simplicity and measurement accuracy. We needed the algorithm to measure each of the dimensions to be

generic enough to account for the plethora of programming languages used in the development of software projects, yet be specific enough to produce meaningful results. We acknowledge that this list is subjective, and by no means exhaustive; however, the evaluation framework makes no assumption of either the different dimensions or the way in which the dimensions are used in determining if a repository contains an engineered software project.

In the subsections that follow, we describe each of the seven dimensions in greater detail. In each subsection, we describe the attribute of an software project that the dimension represents, propose a metric to quantify the dimension, and describe an approach to collect the metric from a source code repository. The process of collecting a dimension's metric may require either or both sources of data introduced earlier.

In addition to the seven dimensions enumerated above, we also included repository size, quantified using the Source Lines of Code (SLOC) metric, as a dimension to assess, and control, the influence that repository size may have on the other dimensions. We used a popular Perl utility `cloc` (Danial 2014) to collect the SLOC metric from a repository.

We have developed an open-source tool called `reaper` that is capable of collecting the metric for each of the dimensions from a given source code repository. The source code for `reaper` is available on GitHub at https://github.com/RepoReapers/reaper. In its current version, the capabilities of `reaper` are subject to the following restrictions:

- The source code repository being analyzed must be publicly-accessible on GitHub and
- The primary language of the repository must be one of Java, Python, PHP, Ruby, C++, C, or C#. We chose these languages based on their popularity on GitHub as reported by GitHut (Carlo 2016).

`reaper` was designed with flexibility and extensibility in mind. Extending `reaper`, to add the capability to analyze source code repositories in a new language (say JavaScript) for instance, is fairly trivial and the process is detailed in the project wiki.[3]

4.1 Community

Software engineering is an inherently collaborative discipline. With the advent of the Internet and a plethora of tools that simplify communication, software development is increasingly decentralized. Open source development, in particular, thrives on decentralization, with globally dispersed developers contributing code, knowledge, and ideas to a multitude of open source projects. Collaboration in open source software development manifests itself as a community of developers.

The presence of a developer community indicates that there is some form of collaboration and cooperation involved in the development of the software system, which is partial evidence for the repository containing an engineered software project.

Metric Whitehead et al. (2010) have hypothesized that the development of a software system involving more than one developer can be considered as an instance of collaborative software engineering. We propose a metric, *core contributors*, to quantify the community established around a source code repository.

[3] https://github.com/RepoReapers/reaper/wiki/Extending-reaper.

Definition 2 *Core contributors* is the cardinality of the smallest set of contributors whose total number of commits to a source code repository accounts for 80% or more of the total contributions.

Approach The notion of *core contributors* is prevalent in open source software where a set of contributors take ownership of and drive a project towards a common goal. Mockus et al. (2000) have applied this concept in their study of open source software development practices. The definition of core contributors is the same as that of core developers as defined by Syer et al. (2013).

We computed total contributions by counting the number of commits made to a repository as recorded in the database. We then grouped the commits by author and picked the first n authors for which the cumulative number of commits accounted for 80% of the total contributions. The value of n represents the *core contributors* metric.

There is one issue in the implementation of this metric in `reaper`. We use the GHTorrent data to find unique contributors of a repository. However, GHTorrent has the notion of "fake users" who do not have GitHub accounts (GHTorrent 2016b) but publish their contributions with the help of real GitHub users. For example, a "fake user" makes a commit to a local Git repository, then a "real user" pushes those commits to GitHub using their account. Sometimes, the real and fake users may be the same. This is the case when a developer with a GitHub account makes commits with a secondary email address. This tends to inflate the *core contributors* metric for small repositories with only one real contributor, and could be improved in the future by detecting similar email addresses.

4.2 Continuous Integration

Continuous integration (CI) is a software engineering practice in which developers regularly build, run, and test their code combined with code from other developers. CI is done to ensure that the stability of the system as a whole is not impacted by changes. It typically involves compiling the software system, executing automated unit tests, analyzing system quality and deploying the software system.

With millions of developers contributing to thousands of source code repositories, the practice of continuously integrating changes ensures that the software system contained within these constantly evolving source code repositories is stable for development and/or release. The use of CI is further evidence that the software project might be considered an engineered software project.

Metric The metric for the continuous integration dimension may be defined as a piecewise function as shown below:

$$M_{ci}(r) = \begin{cases} 1 & \text{If repository } r \text{ uses a CI service} \\ 0 & \text{Otherwise} \end{cases}$$

Approach The use of a continuous integration service is determined by looking for a configuration file (required by certain CI services) in the source code repository. An inherent limitation of this approach is that it supports the identification of stateless CI services only. Integration with stateful services such as Jenkins, Atlassian Bamboo, and Cloudship cannot be identified since there may be no trace of the integration in the repository. The continuous integration services currently supported are: Travis CI, Hound, Appveyor, Shippable, MagnumCI, Solano, CircleCI, and Wercker.

4.3 Documentation

Software developers create and maintain various forms of documentation. Some forms are part of the source files, such as code comments, whereas others are external to source files, such as wikis, requirements, and design documents. One purpose of documentation is to aid the comprehension of the software system for maintenance purposes. Among the many forms of documentation, source code comments were found to be most important, second only to the source code itself (de Souza et al. 2005). The presence of documentation, in a sufficient quantity, indicates the author thought of maintainability; this serves as partial evidence towards a determination that the software system is engineered.

Metric In this study, we restrict ourselves to documentation in the form of source code comments. We propose a metric, *comment ratio*, to quantify a repository's extent of source code documentation.

Definition 3 *Comment ratio* is the ratio of the number of comment lines of code (cloc) to the number of non-blank lines of source code (sloc) in a repository r.

$$M_d(r) = \frac{cloc}{sloc + cloc} \tag{2}$$

Approach We use a popular Perl utility `cloc` (Danial 2014) to compute source lines of code and comment lines of code. `cloc` returns blank, comment, and source lines of code grouped by the different programming languages in the repository. We aggregate the values returned by `cloc` when computing the comment ratio.

We note that comment ratio only quantifies the extent of source code documentation exhibited by a repository. We do not consider the quality, staleness, or relevancy of the documentation. Furthermore, we have only considered a single source of documentation—source code comments—in quantifying this dimension. We have not considered other (external) sources of documentation such as wikis, design documents, and any associated README files because identifying and quantifying these external sources may not be as straightforward. We may have to leverage natural language processing techniques to analyze these external documentation artifacts.

4.4 History

Eick et al. (2001) have shown that source code must undergo continual change to thwart feature starvation and remain marketable. A change could be a bug fix, feature addition, preventive maintenance, vulnerability resolution, etc. The presence of sustained change indicates that the software system is being modified to ensure its viability. This is partial evidence towards a determination that the software system is engineered.

Metric In the context of a source code repository, a commit is the unit by which change can be quantified. We propose a metric, *commit frequency*, to be the frequency by which a repository is undergoing change.

Definition 4 *Commit frequency* is the average number of commits per month.

$$M_h(r) = \frac{1}{m} \sum_{i=1}^{m} c_i \tag{3}$$

Where,

- c_i is the number of commits for the month i
- m is the number of months between the first and last commit to the repository r

Approach Each c_i was computed by counting the number of commits recorded in the database for the month i. However, m was computed as the difference, in months, between the date of the first commit and date of the last commit to the repository. If m was computed to be 0, the value of the metric was set to 0.

4.5 Issues

Over the years, there have been a plethora of tools developed to simplify the management of large software projects. These tools support some of the most important activities in software engineering such as management of requirements, schedules, tasks, defects, and releases. We hypothesize that a software project that employs project management tools is representative of an engineered software project. Thus, the evidence of the use of project management tools in a source code repository may indicate that the software system contained within is engineered.

There are several commercial enterprise tools available, however, there is no unified way in which these tools integrate with a source code repository. Source code repositories hosted on GitHub can leverage a deceptively named feature of GitHub—GitHub Issues—to potentially manage the entire lifecycle of a software project. We say deceptively named because an "issue" on GitHub may be associated with a variety of customizable labels which could alter the interpretation of the issue. For example, developers could create user stories as GitHub issues and label them as *User Story*. The richness and flexibility of the GitHub Issues feature has fueled the development of several third party services such as Codetree (Codetree Studios 2016), HuBoard (HuBoard Inc 2016), waffle.io (CA Technologies 2016), and ZenHub (Zenhub 2016). These services use GitHub Issues to support lifecycle management of projects.

Metric In this study, we assume the sustained use of the GitHub Issues feature to be indicative of management in a source code repository. We propose a metric, *issue frequency*, to quantify the sustained use of GitHub Issues in a repository.

Definition 5 *Issue frequency* is the average number of issue events transpired per month.

$$M_i(r) = \frac{1}{m} \sum_{i=1}^{m} s_i \tag{4}$$

Where,

- s_i is the number of issues events for the month i
- m is the number of months between the first and last commit to the repository r

Approach Each s_i was computed by counting the number of issue events recorded in the database for the month i. However, m was computed as the difference in months between the date of the first commit and date of the last commit to the repository. If m was computed to be 0, the value of the metric was set to 0.

An inherent limitation in the approach is that it does not support the discovery of other project management tools. Integration with other project management tools may not be easy to detect because structured links to these tools may not exist in the repository source code.

4.6 License

A user's right to use, modify, and/or redistribute a piece of software is dictated by the license that accompanies the software. Licenses are especially important in the context of open source projects as an article (Software Freedom Law Center 2012) by The Software Freedom Law Center discusses. The article highlights the need for and best practices in licensing open source software.

A software with no accompanying license is typically protected by default copyright laws, which state that the author retains all rights to the source code (GitHub Inc 2016c). Although there is no legal requirement to include a license in a source code repository, it is considered a best practice. Furthermore, the terms of service agreement of source forges such as GitHub may allow publicly-accessible repositories to be forked (copied) by other users. Thus, including a license in the repository explicitly dictates the rights, or lack thereof, of the user making copies of the repository. The presence of a software license is necessary but not sufficient to indicate a repository contains an engineered software project according to our definition of the dimension.

Metric The metric for the license dimension may be defined as a piecewise function as shown below:

$$M_l(r) = \begin{cases} 1 & \text{If repository } r \text{ has a license} \\ 0 & \text{Otherwise} \end{cases}$$

Approach The presence of a license in a source code repository is assessed using the GitHub License API. The license API identifies the presence of popular open source licenses by analyzing files such as `LICENSE` and `COPYING` in the root of the source code repository.

The GitHub License API is limited in its capabilities in that it does not consider license information contained in `README.md` or in source code files. Furthermore, the API is still in "developer preview" and as a result may be unreliable. On the other hand, any improvements in the capabilities of the API is automatically reflected in our approach. In the interim, however, we have overcome some of the limitations by analyzing the files in a source code repository for license information. We identify license information by searching repository files for excerpts from the license text of 12 most popular open source licenses on GitHub. For example, we search for "The MIT License (MIT)" to detect the presence of The MIT License. The 12 chosen licenses were enumerated by the GitHub License API (https://api.github.com/licenses). We note that the interim solution implemented may have its own side-effect in cases where a repository, with no license of its own, includes source code files of an external library instead of defining the library as a dependency. If any of the external library source code files contain excerpts of the license we search for, the license dimension may falsely indicate the repository to contain a license.

4.7 Unit Testing

An engineered product is assumed to function as designed for the duration of its lifetime. This assumption is supported by the subjection of the product to rigorous testing. An

engineered *software* product is no different in that the guarantee of the product functioning as designed is provided by rigorous testing. Evidence of testing in a software project implies that the developers have spent the time and effort to ensure that the product adheres to its intended behavior. However, the mere presence of testing is not a sufficient measure to conclude that the software project is engineered. The adequacy of tests is to be taken into consideration as well. Adequacy of the tests contained within a software project may be measured in several ways (Zhu et al. 1997). Metrics that quantify test adequacy by measuring the *coverage* achieved when the tests are executed are commonly used. Essentially, collecting coverage metrics requires the execution of the unit tests which may in-turn require satisfying all the dependencies that the program under test may have. Fortunately, there are means of approximating adequacy of tests in a software project through static analysis. Nagappan et al. (2005) have used the number of test cases per source line of code and number of assertions per source line of code in assessing the test quantity in Java projects. Additionally, Zaidman et al. (2008) have shown that test coverage is positively correlated with the percentage of test code in the system.

Metric We propose a metric, *test ratio*, to quantify the extent of unit testing effort.

Definition 6 *Test ratio* is the ratio of number of source lines of code in test files to the number of source lines of code in all source files.

$$M_u(r) = \frac{slotc}{sloc} \tag{5}$$

Where,

- $slotc$ is the number of source lines of code in test files in the repository r
- $sloc$ is the number of source lines of code in all source files in the repository r

Approach In order to compute $slotc$, we must first identify the test files. We achieved this by searching for language- and testing framework-specific patterns in the repository. For example, test files in a Python project that use the native unit testing framework may be identified by searching for patterns `import unittest` or `from unittest import TestCase`.

We used `grep` to search for and obtain a list of files that contain specific patterns such as above. We then use the `cloc` tool to compute $sloc$ from all source files in the repository and $slotc$ from the test files identified. Occasionally, a software project may use multiple unit testing frameworks e.g. a Django web application project may use Python's `unittest` framework and Django's extension of `unittest–django.test`. In order to account for this scenario, we accumulate the test files identified using patterns for multiple language-specific unit testing frameworks before computing $slotc$.

The multitude of unit testing frameworks available for each of the programming languages considered makes the approach limited in its capabilities. We currently support 20 unit testing frameworks. The unit testing frameworks currently supported are: Boost, Catch, googletest, and Stout gtest for C++; clar, GLib Testing, and picotest for C; NUnit, Visual Studio Testing, and xUnit for C#; JUnit and TestNG for Java; PHPUnit for PHP; django.test, nose, and unittest for Python; and minitest, RSpec, and Ruby Unit Testing for Ruby.

In scenarios where we are unable to identify a unit testing framework, we resort to considering all files in directories named `test`, `tests`, or `spec` as test files.

5 Implementation

In this section, we describe two of the (potentially) many approaches of implementing the boolean-valued function representing the evaluation framework from (1) in Section 2.1. In both approaches, a repository is represented by the seven (quantifiable) dimensions that were introduced in the previous section.

5.1 Training Data Sets

The boolean-valued function representing the evaluation framework is essentially a classifier capable of classifying a repository as containing an engineered software project or not. The classifier is trained using a set of repositories (called the training data set) that have been manually classified as containing engineered software projects according to some specific definition of an engineered software project.

In the context of our study, we demonstrate training the classifier using two definitions of an engineered software project, they are:

- *Organization* - A repository is said to contain an engineered software project if it is similar to repositories owned by popular software engineering organizations.
- *Utility* - A repository is said to contain an engineered software project if it is similar to repositories that have a fairly general-purpose utility to users other than the developers themselves. For instance, a repository containing a Chrome plug-in is considered to have a general-purpose utility, however, a repository containing a mobile application developed by a student as a course project may not considered to have a general-purpose utility.

In the subsections that follow, we describe the approach used to manually identify repositories that conform to each of the definitions of an engineered software project from above. These repositories compose respective (training) data sets. Additionally, the two (training) data sets were appended with negative instances i.e. repositories that do not conform to either of the definitions of an engineered software project presented above.

5.1.1 Organization Data Set

The process of identifying the repositories that compose the organization data set was fairly trivial. The preliminary step was to manually sift through repositories owned by organizations such as Amazon, Apache, Facebook, Google, and Microsoft and identify a set of 150 repositories. The task was divided such that three of the four authors independently identified 50 repositories each, ensuring that there was no overlap between the individual authors. The manual identification of repositories was supported by a set of guidelines that were established prior to the sifting process. These guidelines dictated the aspects of a repository that were to be considered in deciding whether to include a repository. Some of the guidelines used were (a) repository must be licensed under an open-source license, (b) repository uses comments to document code, (c) repository uses continuous integration, and (d) repository contains unit tests. With 150 repositories identified, the next step was for each author to review the 100 repositories identified by the other two authors to mitigate any biases that may have been induced by subjectivity. The repositories that at least one author marked for review were discussed further. At the end of the discussion, a decision was made to either include the repository or replace it with another repository that was unanimously chosen during the discussion.

`scrapy/scrapy`, `phalcon/incubator`, `JetBrains/FSharper`, and `owncloud/calendar` are some examples of repositories included in the organization data set that are known to contain engineered software projects.

The organization data set is available for download as a CSV file—organization.csv—from GitHub Gist accessible at https://gist.github.com/nuthanmunaiah/23dba27be17bbd0abc40079411dbf066.

5.1.2 Utility Data Set

Unlike the process of identifying the repositories that compose the organization data set, the process of identifying the repositories that compose the utility data set was non-trivial. The repositories that composed the utility data set were identified by manually evaluating a random sample from the 1,857,423 repositories that were analyzed by `reaper`. Similar to the process of composing the organization data set, we used a set of guidelines for deciding if a repository should be included or not. The guidelines dictated the various aspects that were to be considered in deciding whether a repository has a general-purpose utility. The guidelines used here were more subjective than those used in the process of composing the organization data set. Some of the guidelines used were (a) repository contains sufficient documentation to enable the project contained within to be used in a general-purpose setting, (b) repository contains an application or service that is used by or has the potential to be used by people other than the developers, (c) repository does not contain cues indicating that the source code contained within may be an assignment. The potential for bias was mitigated by two authors independently evaluating the same random sample of repositories. The first 150 repositories that both authors agreed to include composed the utility data set.

`veg/hyphy`, `seblin/launchit`, `smcameron/opencscad`, and `apanzerj/zit` are some examples of repositories included in the utility data set that are known to contain engineered software projects.

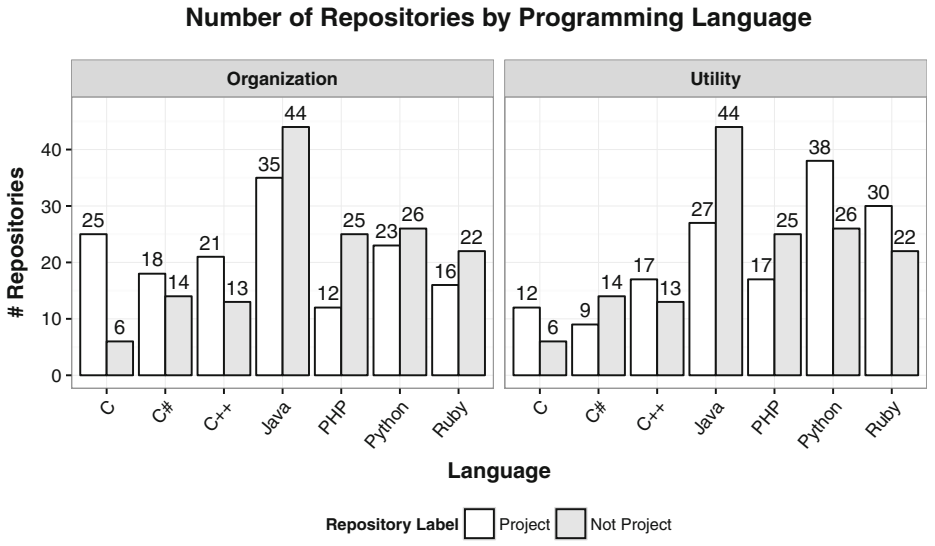


Fig. 1 Number of repositories in the organization and utility data sets grouped by programming language

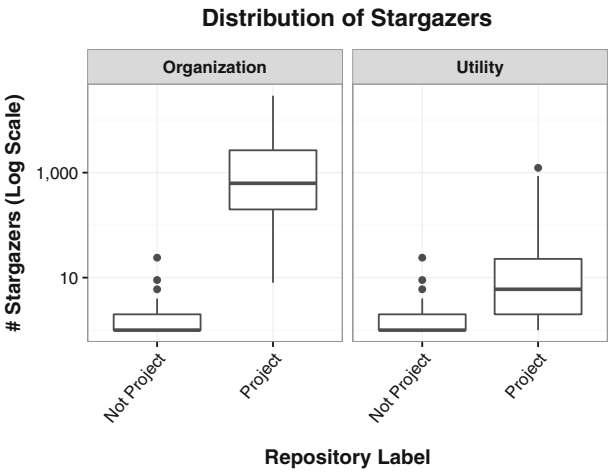


Fig. 2 Distribution of number of stargazers of repositories in the organization and utility data sets

The utility data set is available for download as a CSV file—utility.csv—from GitHub Gist accessible at https://gist.github.com/nuthanmunaiah/23dba27be17bbd0abc40079411dbf066.

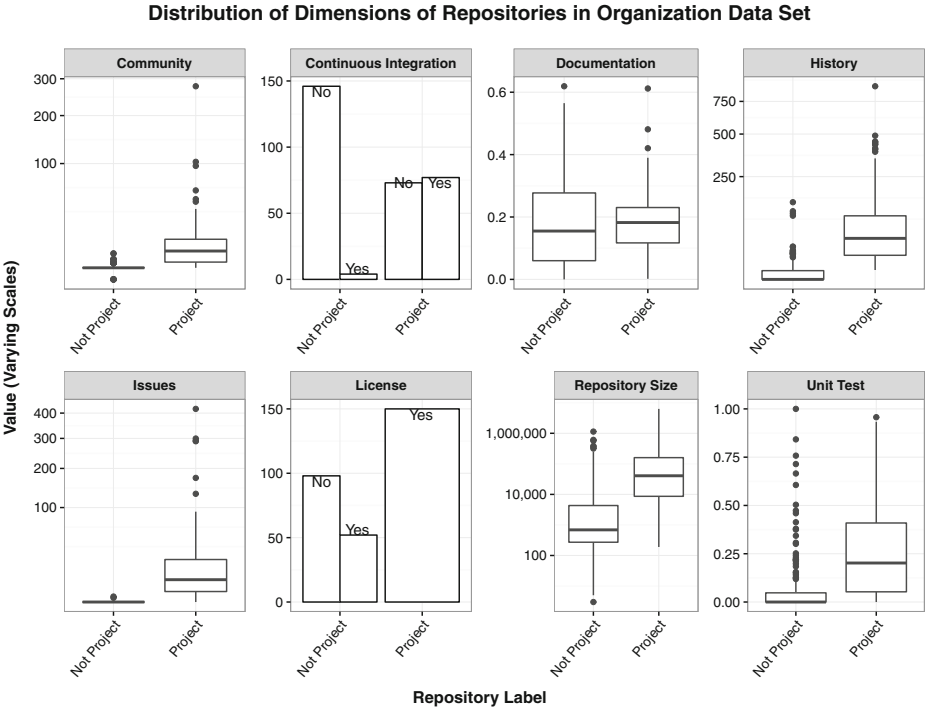


Fig. 3 Distribution of the dimensions of repositories in the organization data set

5.1.3 Negative Instances Data Set

The negative instances data set is essentially a collection of 150 repositories that do not conform to either of the two definitions (i.e. organization and utility) of an engineered software project. The repositories that compose the negative instances data set were identified during the process of composing the utility data set in that, the first 150 repositories (not owned by an organization) that both authors agreed to exclude from the utility data set were considered to be a part of the negative instances data set.

The organization and utility data set files available for download on GitHub Gist also contain repositories from the negative instances data set.

5.1.4 Data Sets Summary

In this section, we summarize the training data sets using visualizations. The repositories that compose the organization and utility data sets are labeled as "project" and the repositories that compose the negative instances data set are labeled as "not project". Shown in Fig. 1 is the number of repositories of each kind ("project" and "not project") grouped by programming language in the organization and utility data sets.

The distribution of number of stargazers of repositories of each kind ("project" and "not project") in organization and utility data sets are shown in Fig. 2.

Additionally, shown in Figs. 3 and 4 are the distributions of all seven dimensions obtained from the repositories in the organization and utility data sets, respectively.

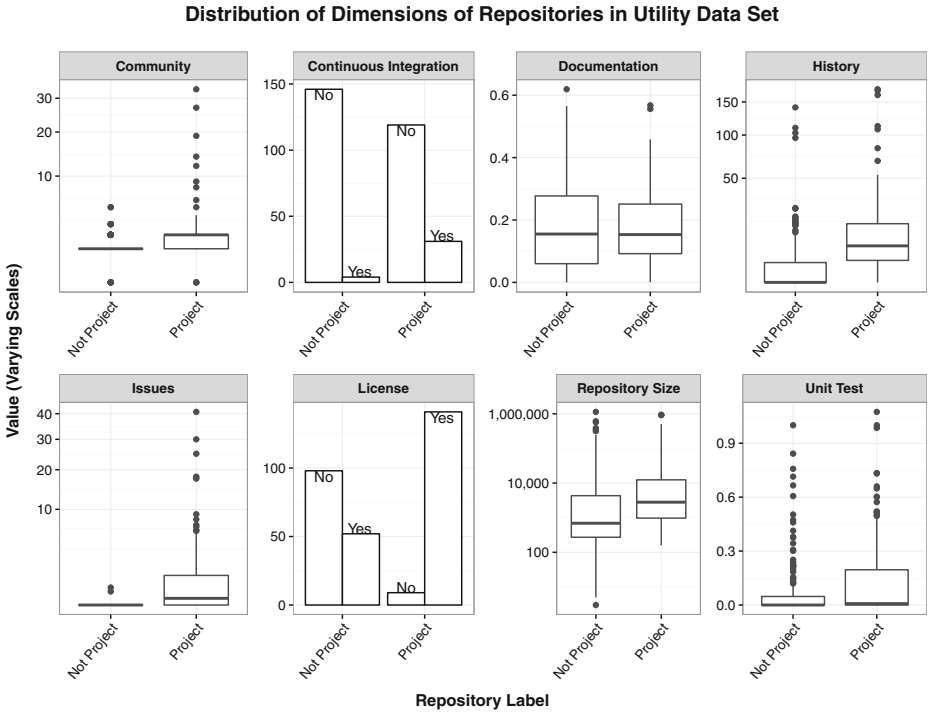


Fig. 4 Distribution of the dimensions of repositories in the utility data set

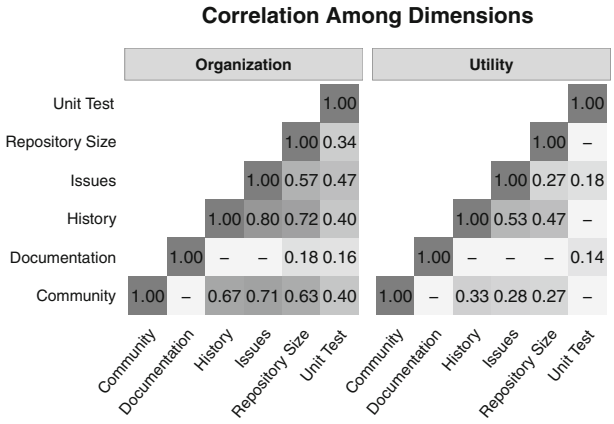


Fig. 5 Spearman's ρ between pairs of dimensions in the organization and utility data sets with - (dash) representing the statistically insignificant correlations

We used the Spearman's Rank Correlation Co-efficient (ρ) to assess the correlation between the various continuous-valued dimensions. Shown in Fig. 5 are the values of Spearman's ρ, when statistically significant at p-value ≤ 0.05, between pairs of dimensions in the organization and utility data sets. - (dash) in the figures represent those correlations that were not statistically significant.

As seen in Fig. 5, there is a moderate-to-strong correlation between the various dimensions in the organization data set. Repository size, in particular, is at least moderately correlated with two of the three continuous-valued dimensions. We further used the non-parametric Mann-Whitney-Wilcoxon (MWW) test and Cliff's δ effect size to evaluate the association between repository size and the binary-valued dimensions (continuous integration and license). Shown in Table 1 are the results from the association analysis in the organization and utility data sets. As seen in the table, except for continuous integration dimension in utility data set, repository size is statistically significantly (p-value ≤ 0.05) associated with the binary-valued dimensions with at least medium effect size. The association results indicate that a larger repository is more likely to have continuous integration and/or license. However, we believe that the association may be an artifact of confounding factors such as a organizations complying with a convention to have a license in all of the repositories that they own. Furthermore, as Rosenberg (1997) points out, the utility of SLOC as a metric is as a covariate of other metrics.

Table 1 p-value from Mann-Whitney-Wilcoxon test for association between binary-valued dimensions (continuous integration and license) and repository size and the corresponding Cliff's δ

Data Set	Dimension (d)	Mean Repository Size		Association	
		$M_d = 0$	$M_d = 1$	p-value	Cliff's δ
Organization	Continuous integration	141,845	159,706	8.559e−08	0.4028
	License	2,696	216,515	<2.2e−16	0.8646
Utility	Continuous integration	35,508	54,360	0.125	0.1597
	License	3,263	56,804	<2.2−16	0.6210

We account for the correlation between repository size and the various dimensions proposed in our study by including repository size as a feature in both the score-based and Random Forest classifiers as suggested in Emam et al. (2001).

5.2 Approaches

In this section, we introduce two approaches of implementing the boolean-valued function representing the evaluation framework.

5.2.1 Score-based Classifier

The score-based classifier is a custom approach to implementing the evaluation framework that allows complete control over the classification. In this approach, the boolean-valued function from (1) in Section 2.1 takes the form shown in (6).

$$f(r) = \begin{cases} true & \text{If } score(r) \geq score_{ref} \\ false & \text{Otherwise} \end{cases} \tag{6}$$

$$score(r) = \sum_{d \in D} h_d(M_d, t_d) \times w_d$$

Where,

- r is the repository to classify
- D is a set of dimensions along which the repository, r, is evaluated. These may be analogous to the software engineering practices e.g., unit testing, documenting source code, etc.
- M_d is the metric that quantifies evidence of the repository, r, employing a certain software engineering practice in the dimension, d. For example, the proportion of comment lines to source lines quantifies documentation.
- t_d is a threshold that must be satisfied by the corresponding metric, M_d, for the repository, r, to be considered engineered along the dimension, d. For example, having a sufficiently high proportion of comment lines to source lines may indicate that the project is engineered along the documentation dimension.
- $h_d(M_d, t_d)$ is a heuristic function that evaluates to 1 if the metric value, M_d, satisfies the corresponding threshold requirement, t_d, 0 otherwise.
- w_d is the weight that specifies the relative importance of each dimension d.
- $score_{ref}$ is the reference score i.e. the minimum score that a repository must evaluate to in order to be considered to contain an engineered software project.

In the case of the score-based classifier, the training data set is used to determine the thresholds, t_d, and compute the reference score, $score_{ref}$. For all repositories in each of the two training data sets (plus the repositories from the negative instances data set), we collected the seven metric values. Outliers were eliminated using the Peirce criterion (Ross 2003). For the boolean-valued metrics, 1 (i.e. True) is the threshold. For all other metrics, the minimum non-zero metric value was chosen to be the corresponding threshold. The threshold values corresponding to each of the seven dimensions, established from repositories in each of the two training data sets, are shown in Table 2. Also shown in Table 2 are the relative weights that we have used in our score-based classifier. We subjectively chose these weights to illustrate that the score-based classifier can be customized. Researchers may alter the weights to exercise finer control over the classification and select repositories that suit their individual studies. Furthermore, we also considered the limitations in collecting the

Table 2 Dimensions and their corresponding weights, metrics, and thresholds (established from the organization and utility data sets)

Dimension (d)	Metric (M_d)	Weight (w_d)	Threshold (t_d)	
			Organization	Utility
Community	Core Contributors	20	2	2
Continuous Integration	Evidence of CI	5	1	1
Documentation	Comment Ratio	10	1.8660e−03	1.1710e−03
History	Commit Frequency	20	2.0895	0.15
Issues	Issue Frequency	5	2.2989e−02	1.1905e−02
License	Evidence of License	20	1	1
Repository Size	Source Lines of Code (SLOC)	10	190	160
Unit Test	Test Ratio	10	1.0160e−03	1.4200e−04

value of the associated metric from a source code repository when deciding the weights. For example, the approach to evaluating a source code repository along the community dimension is more robust and thus its weight is higher than the unit testing dimension, where there are inherent limitations owing to our non-exhaustive set of framework signatures.

The weights and thresholds from Table 2 were used to compute the scores of all repositories in the organization and utility data sets. The distribution of the scores is shown in Fig. 6. The reference score in the organization and utility training data sets was found to be 70 and 30, respectively.

The score-based classifier approach is flexible and enables a finer control over the classification. The weights, in particular, enable the implementer to explicitly define the importance of each dimension. In effect, (6) may be tailored to implement a variety of different classifiers using different set of dimensions, D, and corresponding metrics, thresholds, and weights. For instance, if there is a need to build a classifier that considers gender

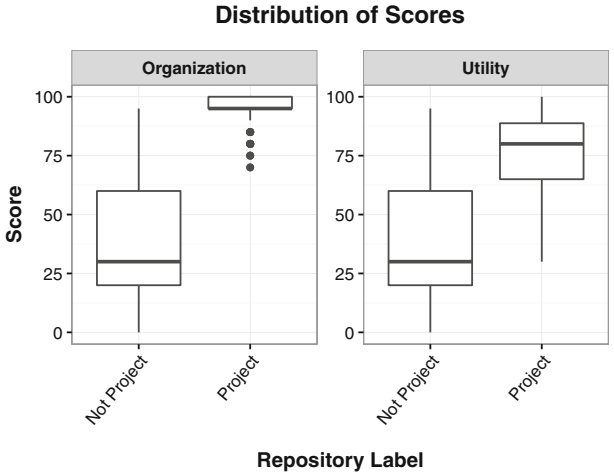


Fig. 6 Distribution of scores for repositories in the organization and utility data sets

bias in the acceptance of contributions in open-source community (like in the work by Kofink 2015), one could introduce a new dimension, say *bias*, define a metric to quantify gender bias in a repository, identify an appropriate threshold, and weight the dimension in relation to other dimensions that may be pertinent to the study.

5.2.2 Random Forest Classifier

Random Forest classifier is a tree-based approach to classification in which multiple trees are trained such that each tree casts a vote which is then aggregated to produce the final classification (Breiman 2001).

The Random Forest classifier is simpler to implement but the simplicity comes at a loss of finer control over the classification. The training data set is the only way to affect the classifier performance. While the implementer has the option to ignore dimensions when training the classifier, there is no way to express relative weighting of dimensions as supported in the score-based classifier.

6 Results

In this section, we present (a) the results from the validation of the classifiers and (b) the results from applying the classifiers to identify (or predict) engineered software projects in a sample of 1,857,423 of the 2,247,526 GitHub repositories that were active at the time the analysis was conducted. Since we have two different classifiers (score-based and Random Forest) trained using two different data sets (organization and utility), the validation and prediction analysis is repeated four times.

6.1 Validation

In this section, we present the approach to and results from the validation of the score-based and Random Forest classifiers trained with organization and utility data sets. The validation is carried out in the context of a set of 200 repositories, called the validation set, for which the ground truth classification was manually established. We considered validation from two perspectives: *internal*, in which the performance of the classifiers itself was validated, and *external*, in which the performance of the classifiers was compared to that of a classification scheme that uses number of stargazers (Ray et al. 2014) as the criteria. We used false positive rate (FPR), false negative rate (FNR), precision, recall, and F-measure to assess the classification performance.

6.1.1 Establishing the Ground Truth

The performance evaluation of any classifier typically involves using the classifier to classify a set of samples for which the ground truth classification is known. On similar lines, to evaluate the performance of the score-based and Random Forest classifiers, we manually composed a set of 200 repositories (100 repositories that have been assessed to contain an engineered software project and 100 repositories that do not). We followed a process similar to that followed in identifying the repositories to compose the utility data set. To ensure an unbiased evaluation, the validation set was independently evaluated by two authors and only those repositories that both authors agreed on were included and appropriately labeled.

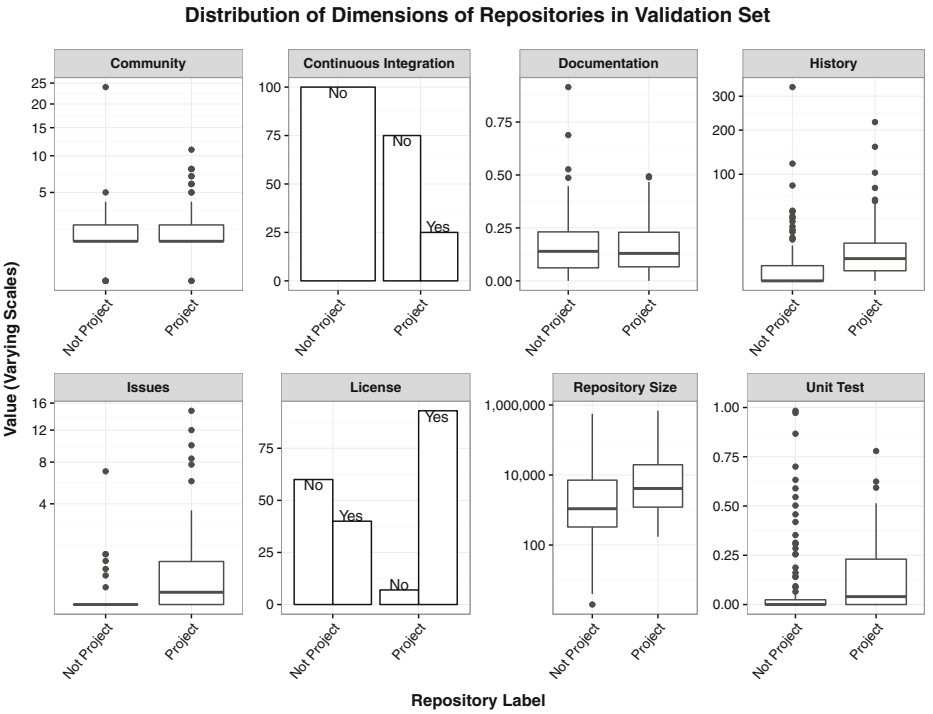


Fig. 7 Distribution of dimensions of repositories in the validation set

Shown in Fig. 7 is the distribution of the seven dimensions collected from repositories in the validation set. As seen in the figure, repositories known to contain engineered software project tend to have higher median values in almost all dimensions.

The validation set is available for download as a CSV file—validation.csv—from GitHub Gist accessible at https://gist.github.com/nuthanmunaiah/23dba27be17bbd0abc40079411dbf066.

6.1.2 Internal Validation

In this validation perspective, the performance of score-based and Random Forest classifiers trained using organization and utility data sets is evaluated in the context of the validation set.

Organization Data Set The performance of score-based and Random Forest classifiers trained with organization data set is shown in Table 3.

Table 3 Performance of score-based and Random Forest classifiers trained with organization data set

Classifier	FPR	FNR	Precision	Recall	F-measure
Score-based	19%	39%	76%	61%	68%
Random Forest	6%	58%	88%	42%	57%

Table 4 Performance of score-based and Random Forest classifiers trained with utility data set

Classifier	FPR	FNR	Precision	Recall	F-measure
Score-based	73%	1%	58%	99%	73%
Random Forest	19%	14%	82%	86%	84%

Clearly, the score-based classifier performs better than the Random Forest classifier in terms of F-measure. If a lower false positive rate is desired, the Random Forest classifier may be better suited as it has a considerably lower false positive rate than the score-based classifier. We now present some examples of repositories from the organization data set that were misclassified by both score-based and Random Forest classifiers.

1. **False Positive** - `software-engineering-amsterdam/sea-of-ql` is the quintessential example of a repository that should not be included in a software engineering study because it is essentially a collaboration space where all students enrolled in a particular course develop their individual software projects. The repository received a score of 95 which is closer to a perfect score of 100. Analyzing the dimensions of the repository, we found, quite unsurprisingly, that almost all dimensions satisfied the threshold requirement. The repository was contributed to by 24 developers with an average of over 330 commits made every month. Although the repository does not have a license, the limitation of our implementation of the license dimension seems to have identified a license in library files that may have been included in the source code repository.
2. **False Negative** - `kzoll/ztlogger` is a repository that contains PHP scripts to log website traffic information. The repository received a score of 40 which is much lower than the reference score of 70. Analyzing the dimensions of the repository, we found that the repository was contributed to by a single developer, did not use continuous integration, issues or unit testing.

Utility Data Set The performance of score-based and Random Forest classifiers trained with utility data set is shown in Table 4.

Clearly, the Random Forest model performs better than the score-based model. A particularly surprising outcome from the validation is the large false positive rate of the score-based classifier. The large false positive rate indicates that the classifier may have classified almost all repositories as containing an engineered software project. We now present some examples of repositories from the utility data set that were misclassified by both score-based and Random Forest classifiers.

1. **False Positive** - `mer-packages/qtgraphicaleffects` is a repository that contains source code for certain visual items that may be used with images or videos. The repository was manually classified as not containing an engineered software project because of the unusual repository organization. Furthermore, looking at other repositories owned by the `mer-packages` organization, it appears that the repository may actually be a copy of the source code of the Qt Graphical Effects module,[4] being ported to the Mer[5] mobile platform. The repository received a score of 95. Analyzing the dimensions of the repository, we found almost all dimensions satisfied the threshold

[4] http://doc.qt.io/qt-5/qtgraphicaleffects-index.html.

[5] http://merproject.org/.

requirement. Although the repository was not manually classified as containing an engineered software project, the inclusion of this particular repository may not be a problem as the project, possibly a clone from a different repository, has a general-purpose utility.

2. **False Negative** - `vzvu3k6k/mcg_source_list` is a repository that contains a simple Ruby on Rails application that may be used to generate Markov Chain using tweets from from Twitter. The `README.md` file contained in the repository indicated that the application was hosted on Heroku, however, it was inaccessible due to an internal server error at time it was accessed during our validation. The application, although simple, does have a general-purpose utility and the `README.md` file had instructions on configuring and deploying the application to Heroku. The repository received a score of 20. Analyzing the source code contained in and the dimensions of the repository, we found that the project was too simple with only 38 commits made by a single developer over the course of 11 days. The repository neither had a license file nor did any of the source code files in the repository have license text in them. The ground truth classification of this repository was that it contained an engineered software project because of the utility of the application.

6.1.3 External Validation

In this validation perspective, the performance of score-based and Random Forest classifiers is compared to that of the stargazers-based classifier used in prior literature (Ray et al. 2014). We previously noted that the popularity of a repository is one potential criterion in identifying a data set for research studies. The intuition is that popular repositories (i.e. repositories with many "stargazers") will contain actual software that people like and use (Jarczyk et al. 2014). The intuition has been the basis for several well-received studies. For example, the papers by Ray et al. (2014) on programming languages and code quality (which has over 34 citations) and Guzman et al. (2014) on commit comment sentiment analysis (which has over 15 citations) use the number of stargazers as a way to select projects for their case studies.[6] These papers use the top starred projects in various languages, which are bound to be extremely popular. The `mongodb/mongo` repository used in the data set by Ray et al. (2014), for instance, has over 8,927 stars.

In Tables 3 and 4 from the previous section, we presented the performance of score-based and Random Forest classifiers trained using organization and utility data sets, respectively. We now use the stargazers-based classifier to classify the repositories from the validation set. In using a stargazers-based classifier, Ray et al. (2014) ordered and picked the top 50 repositories in each of the 19 popular languages. We applied the same filtering scheme to a sample of 1,857,423 GitHub repositories and established the minimum number of stargazers to be 1,123. In other words, a repository is classified as containing an engineered software project (based on popularity) if it has 1,123 or more stargazers. As an exploratory exercise, we also evaluated other thresholds (500, 50 and 10) for number of stargazers. The performance metrics from the stargazers-based classifier for varying thresholds of number of stargazers are shown in Table 5. In cases where the classifier produced no positive classifications (i.e. both true positive and false positive are zeros), precision and F-measure cannot be computed and are shown as NA in the table.

As seen in Table 5, at high thresholds (1,123 and 500) the stargazers-based classifier misclassifies all repositories known to contain engineered software projects. As we lower

[6]Citation counts retrieved from Google Scholar.

Table 5 Performance of stargazers-based classifier against the ground truth for varying thresholds of number of stargazers

Threshold	FPR	FNR	Precision	Recall	F-measure
1,123	0%	100%	NA	0%	NA
500	0%	100%	NA	0%	NA
50	0%	86%	100%	14%	25%
10	1%	68%	97%	32%	48%

the threshold, the performance improves, albeit marginally. The most striking limitation of the stargazers-based classifier is the low percentages of recall. While a repository with a large number of stars is likely to contain an engineered software project, the contrary is not always true.

The validation results indicate that by using the stargazers-based classifier, researchers may be excluding a large set of repositories that contain engineered software projects but may not be popular. In contrast, the score-based and Random Forest classifiers trained on organization and utility data sets perform much better in terms of recall while achieving an acceptable level of precision.

As a next level of validation, we compared the performance of the best classifier from our study to the best stargazer-based classifier from Table 5. In terms of F-measure, the Random Forest classifier trained using the utility data set exhibited the best performance. Similarly, the stargazer-based classifier with 10 as the threshold for number of stargazers performed the best. We compare these two classifiers not in terms of the performance evaluation metrics but in terms of the actual predicted classification. In Table 6, we present the percentage of repositories where (a) both classifiers agreed with the ground truth, (b) both classifiers disagreed with the ground truth, (c) prediction by Random Forest classifier matches the ground truth but that of stargazers-based classifier does not, and (d) prediction by stargazers-based classifier matches the ground truth but that by Random Forest classifier does not.

For repositories that we believe to be useful in mining software repositories data sets, both approaches incorrectly classify the repositories as "not project" 11% of the time. However, Random Forest classifies 57% of the repositories correctly when the stargazers classifies only 3% of the repositories. A prime example of a project that was missed by stargazers-based classifier but correctly classified by the Random Forest classifier is `jruby/jruby-ldap` from the team that maintains the Ruby implementation of the Java Virtual Machine (JVM). The repository contains a Ruby gem for LDAP support in JRuby. Our Random Forest classifies the repository as a project due to its commit history, test suite, and documentation, among other factors. However, the repository has only 7 stars. While the stargazers-based approach misses `jruby/jruby-ldap`, this repository may be a worthy candidate in a software engineering study. We also note that there were only three cases in

Table 6 Comparison of predictions from Random Forest classifier trained with the utility data set and the stargazers-based classifier with a threshold of 10 stargazers for validation set repositories with different ground truth labels

Ground truth	Both agree	Both disagree	Random forest agrees	Stargazers agrees
Project	29%	11%	57%	3%
Not project	81%	1%	0%	18%

which stargazers-based classifier predicted a repository to be a project while Random Forest classifier did not. Hence, any repository classified as a project by the stargazers-based classifier is highly likely to be classified the same by the Random Forest classifier as well.

In the case where the ground truth classification is "not project", 81% of the time, both approaches correctly classified repositories as "not project". In addition, the stargazers-based classifier correctly classified repositories as "not project" 18% of the time where the Random Forest classifier failed to do so even for a single repository. Consider the repository `liorkesos/drupalcamp`, which has sufficient documentation, commit history, and community to be classified as a "project" by the Random Forest classifier, however, the repository is essentially a collection of static PHP files of a Drupal Camp website, not incredibly useful in a general software engineering study. The stargazers-based classifier predicts `liorkesos/drupalcamp` as "not project" only for its lack of stars.

Summary We can make three observations about the suitability of the score-based or Random Forest classifiers to help researchers generate useful data sets. First, the strict stargazers-based classifier ignores many valid projects but enjoys almost 0% false positive rate. Second, the Random Forest classifier trained with the utility data set is able to correctly classify many "unpopular" projects, helping extend the population from which sample data sets may be drawn. Third, the score-based and Random Forest projects have their own imperfections as well. Our classifiers are likely to introduce false positives into research data sets. Perhaps, our classifiers could be used an an initial selection criteria augmented by the stargazers-based classifier. Nevertheless, we have shown that more work can be done to improve the data collection methods in software engineering research.

6.2 Prediction

In this section, we present the results from applying the score-based and Random Forest classifiers to identify engineered software projects in a sample of 1,857,423 GitHub repositories. Shown in Table 7 are the number of repositories classified as containing an engineered software project by the score-based and Random Forest classifiers. With the exception of the score-based classifier trained using the utility data set, the number of repositories classified as containing an engineered software project is, on average, 13.66% of the total number of repositories analyzed. We can also see from Table 7 that there are far fewer repositories similar to the ones owned by software development organizations than there are repositories similar to the ones that have a general-purpose utility. The number of repositories predicted to be similar to the ones that have a general-purpose utility by the score-based classifier is considerably high. A likely explanation for the unusually high number of repositories could be because of the relatively low reference score of 20 established from the utility data set.

Shown in Fig. 8 is a grouping of results by programming language.

Table 7 Number of repositories classified as containing an engineered software project by score-based and Random Forest classifiers trained using organization and utility data sets

Data Set	Classifier	# Repositories (% Total)
Organization	Score-based	201,966 (10.87%)
	Random Forest	112,277 (6.04%)
Utility	Score-based	1,292,902 (69.61%)
	Random Forest	447,145 (24.07%)

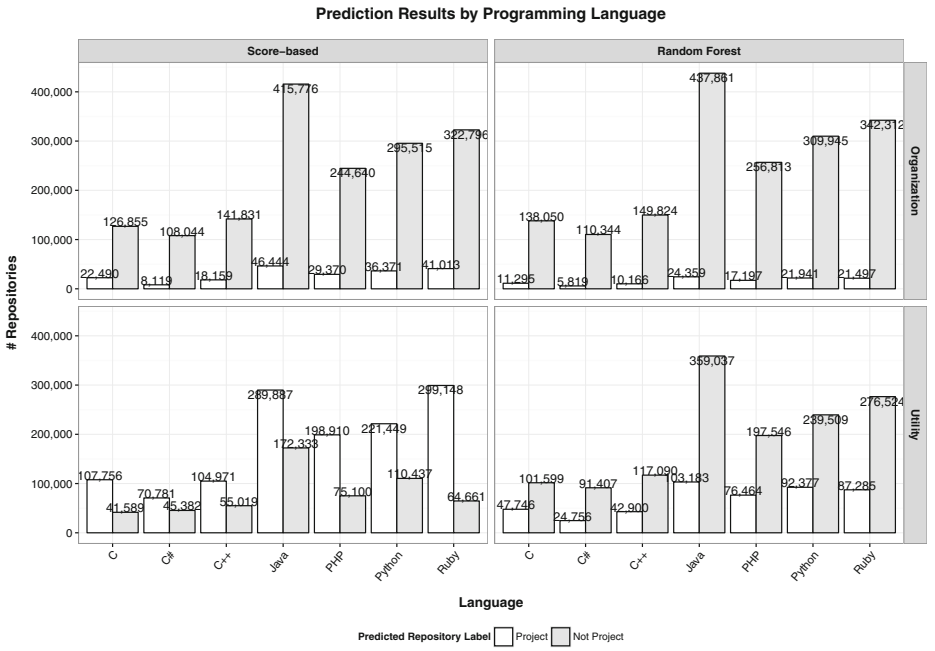


Fig. 8 Number of repositories classified by the score-based and Random Forest classifiers grouped by programming language

The entire data set may be viewed and downloaded as a CSV file from https://reporeapers.github.io. The data set includes the metric values collected from 1,857,423 GitHub repositories.

We hope the data set will help researchers overcome the limitation posed by the arduous task of manually identifying repositories to study, especially in the mining software repositories community.

7 Related Work

An early work by Nagappan (2007) revealed opportunities and challenges in studying open source repositories in an empirical context. Kalliamvakou et al. (2014) described various perils of mining GitHub data; specifically, Peril IV is: "A large portion of repositories are not for software development." In this work, the researchers manually analyzed a sample of 434 GitHub repositories and found that approximately 37% of them were not used for software development. In our study, the best performing classifier predicted that approximately 24.07% of 1,857,423 GitHub repositories contain engineered software projects. Our prediction results highlight the reality that even though 63% of GitHub repositories are used for software development, only a small percentage of those repositories actually contain projects that software engineering researchers may be interested in studying.

Dyer et al. (2013) and Bissyandé et al. (2013a) have created domain specific languages—Boa and Orion, respectively—to help researchers mine data about software repositories. Dyer et al. (2013) have used Boa to curate a sizable number of source code repositories

from GitHub and SourceForge, however, only Java repositories are currently available. In contrast, we have curated over 1,857,423 spanning seven programming languages with the intention of simplifying the process of study selection in large-scale source code mining research.

On the open source community front, Ohloh.net (now Black Duck Open Hub) is a publicly-editable directory of free and open source software projects. The directory is curated by Open Hub users, much like a public wiki, resulting in an accurate and up-to-date directory of open source software. The website provides interesting visualization about software projects curated by Black Duck Open Hub. Tung et al. (2014) have used Open Hub to search for repositories containing engineered software projects as perceived by Open Hub users.

8 Usage Scenarios

In the validation of score-based and Random Forest classifiers, we found the that these classifiers recalled considerably higher number of repositories than a stargazers-based classifier. However, improving the recall of repositories is not as impressive as enabling researchers to exercise finer control over the aspects of repositories that are most pertinent in selecting study subjects for their research. For instance, consider the following studies from prior literature that have all used some ad hoc approach to identify a set of repositories.

- In a study about GitHub issue tracking practices, Bissyandé et al. (2013) started with a random sample of the first 100,000 repositories returned by the GitHub API. The repositories that did not have a the GitHub Issues feature were then removed. The *issues* dimension may have helped in identifying only those repositories that have the GitHub Issues feature turned on and in use.
- In a study of the use of continuous integration practices, Vasilescu et al. (2014) used the GHTorrent (Gousios 2013) database and applied a series of filters to identify a small subset of 223 GitHub repositories to include. The study was restricted to repositories that used Travis CI. However, the use of the *continuous integration* dimension may have simplified the process of selecting all repositories that use continuous integration.
- In a study of license usage in Java projects on GitHub, Vendome (2015) retrieved the metadata for all Java repositories and selected a random sample of 16,221 repositories. The *license* dimension may have been ideal to select all Java repositories that are licensed as open source software.
- In a study of testing practices in open source projects, Kochhar et al. (2013) used the GitHub API to select 50,000 repositories specifically stating that they removed toy projects by manually examining and including only famous projects such as JQuery and Ruby on Rails. The *unit testing* dimension may have helped by removing the need to manually examine the repositories.

Admittedly, the stargazers-based classifier is much simpler to use and Occam's razor would suggest using a simpler solution instead of a (unnecessarily) complex one. However, by using the stargazers-based classifier, a large number of potentially relevant repositories may be ignored.

The aforementioned studies from prior literature focused on specific aspects of software development such as licensing, testing, or issue tracking. While the score-based and Random Forest classifiers may be used to support such studies, the real benefit of the classifiers may only be evident when researchers need access to repositories that simultaneously satisfy

a variety of requirements. For instance, consider the following hypothetical studies in which the classifiers would prove useful in identifying a set of repositories to include.

– A study investigating the relationship between collaboration and testing in open source projects could use the community and unit testing dimensions to identify repositories.
– A study investigating the evolution of documentation in open source projects could use history and documentation dimensions to identify repositories.

We hope that some of these hypothetical studies become a reality and that our data set and classifiers help overcome the barrier to entry.

9 Discussion

In our study, we attempted to identify repositories that contain engineered software projects according to two different definitions of the term. The implementation of one of the definitions involved training two classifiers using repositories in the organization data set. One would assume that the outcome of applying these classifiers can be matched by merely considering all repositories owned by any organization on GitHub as containing an engineered software project. However, not all repositories owned by organizations contain engineered software project. We reuse the validation set from Section 6.1 here to further explore the nuances of repositories owned by organizations.

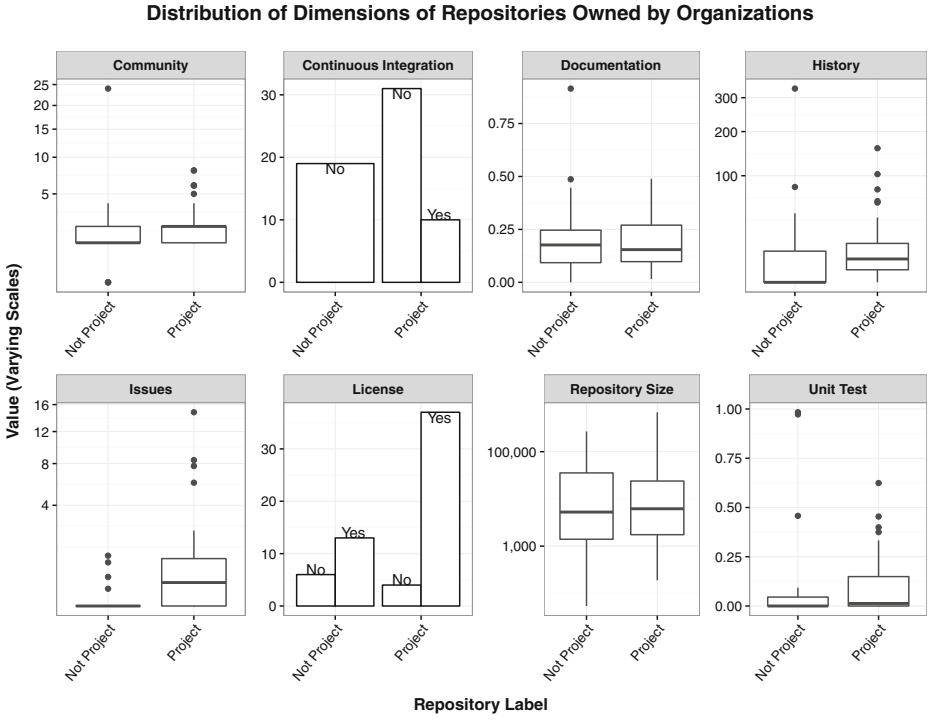


Fig. 9 Comparing the distribution of dimensions of repositories with different manual classification labels but all owned by organizations

The validation set contains 200 repositories, 100 of which are known to contain engineered software project and the remaining 100 are known to not contain engineered software project. 60 of the 200 repositories are owned by organizations.

Shown in Fig. 9 is a comparison between the distribution of the seven dimensions collected from repositories owned by organizations but having different manual classification labels. As seen in the figure, the difference in the distribution of the dimensions provides qualitative evidence to support the notion that not all repositories owned by organizations are similar to one another.

On similar lines, we compared the distribution of the seven dimensions collected from repositories known to contain engineered software project but owned by organizations and users. The comparison is shown in Fig. 10. As seen in the figure, the medians of most dimensions are comparable between the repositories owned by users to that owned by organizations. The similarity in dimensions is exactly the aspect that our approach aims to take advantage of.

Shown in Table 8 is a break down of the prediction results from Section 6.2 into repositories owned by organizations and users. As seen in the table, a considerable number of repositories that were classified as containing engineered software project are owned by individual users. On the other hand, a sizable number of repositories that were classified as not containing an engineered software project were owned by organizations. In effect, filtering repositories based solely on the owner being an organization may lead to the exclusion

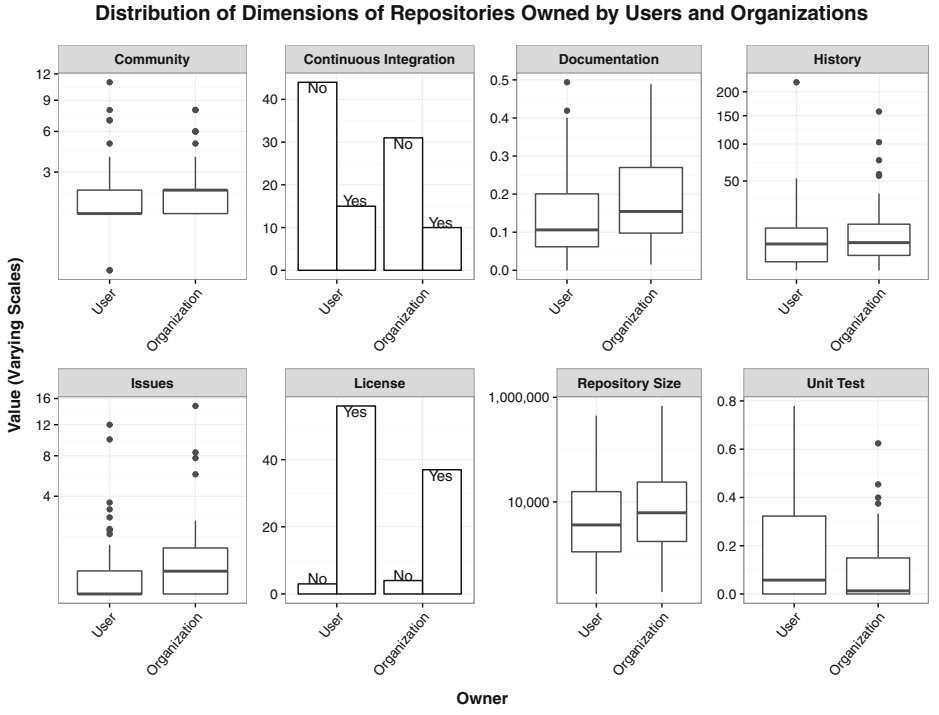


Fig. 10 Comparing the distribution of dimensions of repositories known to contain engineered software project owned by organizations and users

Table 8 Segregation of repositories into those owned by organizations and those owned by individual users classified by the organization data set trained score-based and Random Forest classifiers

Classifier	Predicted Label	# Repositories	
		Organization	User
Score-based	Project	61,072	140,894
	Not Project	150,136	1,505,321
Random Forest	Project	40,614	71,663
	Not Project	170,594	1,574,552

of potentially relevant, user-owned, repositories or the inclusion of repositories that may not contain engineered software project or both.

10 Threats to Validity

10.1 Subjectivity of Dimensions, Thresholds, and Weights

The dimensions used to represent source code repositories in the classification model are subjective, however, we believe that the seven dimensions we have used to be an acceptable default in the context of our study. The open-source tool (`reaper`) developed to measure the dimensions was designed with extensibility in mind. Extending `reaper` to add or modify dimensions is fairly trivial and the process is detailed in the `README.md` file in the `reaper` GitHub repository (Munaiah et al. 2016b).

In addition to the dimensions, the thresholds and weights used in the score-based classifier are subjective as well. Here again, we consider the weights we have used to be an acceptable default in the context of our study, however, alternative weighting schemes may be used to mitigate the subjectivity to a certain extent. Some of these alternative approaches are using (a) a machine learning algorithm to evaluate importance of dimensions using repositories in a training data set, (b) a uniform weighting across dimensions, or (c) a voting-based weighting scheme. Researchers using `reaper` can modify a single file, `manifest.json`, that contains the list of dimensions with their respective thresholds, weights, and settings (e.g. 80% as the cutoff when measuring core contributors) to define their version of a score-based classifier.

As an exploratory exercise, we used the Random Forest classifier trained with organization and utility data sets to assess the relative importance of the variables (dimensions) in the model. Shown in Fig. 11 is the outcome of the exercise. We used the relative importance to adjust the weights in both score-based classifiers. When the classifiers with adjusted weights were used to classify repositories in the sample of 1,857,423 GitHub repositories, the number of repositories classified as containing engineered software projects increased by 36% for score-based classifier trained using the organization data set but decreased by 5% for score-based classifier trained using the utility data set. We, however, chose to retain our weighting scheme as an acceptable default in the context of our study.

10.2 `reaper`-induced Bias

In describing the dimensions measured by `reaper` in Section 4, we outlined the limitations in our approach to collect dimensions' metric from a repository. These limitations may lead to the induction of bias in the repositories selected. For example, if the goal of a study is

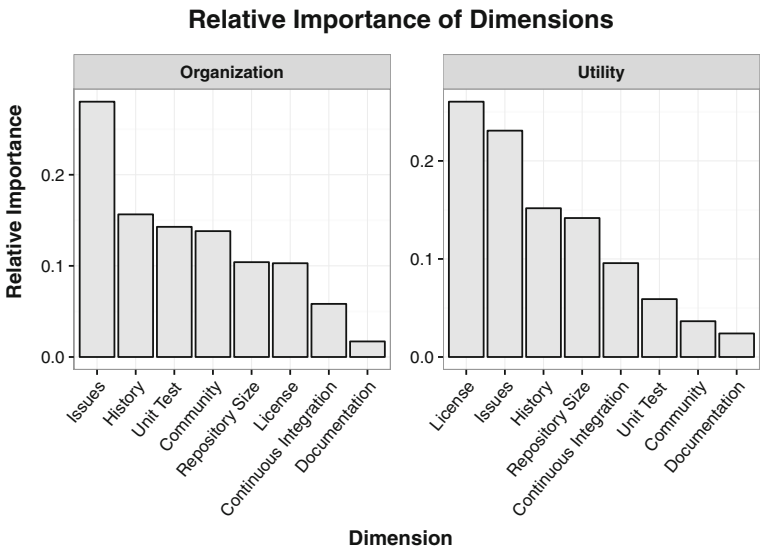


Fig. 11 Relative importance of dimensions in the organization and utility data sets

to analyze the proliferation of unit testing in the real-world, using `reaper` will inherently bias the repositories selected toward unit testing frameworks that are currently recognized by `reaper`. However, the researcher may configure `reaper` such that the unit testing dimension is ignored in the computation of the score thereby mitigating the skew in the repositories selected.

10.3 Extensibility

In addition to `reaper`, the publicly-accessible data set available for download from the project website (https://reporeapers.github.io/) containing the raw values of the seven dimensions for 1,857,423 repositories is an important contribution of our work. A compute cluster with close to 200 nodes took over a month to analyze these repositories. As an alternative to modifying `reaper` and rerunning the analysis, researchers can develop a simple script to directly use the raw values and their own thresholds and weights to compute customized scores for the repositories.

11 Conclusion

The goal of our work was to understand the elements that constitute an engineered software project. We proposed seven such elements, called dimensions. The dimensions are: community, continuous integration, documentation, history, issues, license, and unit testing. We developed an open-source tool called `reaper` that was used to measure the dimensions of 1,857,423 GitHub repositories spanning seven popular programming languages. Two sets of repositories, each corresponding to a different definition of an engineered software project, were composed and a score-based and Random Forest classifiers were trained.

The classifiers were then used to identify all repositories in the sample of 1,857,423 GitHub repositories that were similar to the ones that conform to the definitions of the

engineered software project. Our best performing Random Forest model predicted 24.07% of 1,857,423 GitHub repositories contain engineered software project.

Acknowledgments We thank Nimish Parikh for his contributions during the initial stages of the research. We thank our peers for providing us their GitHub authentication keys which helped us attain the API bandwidth required. We also thank the Research Computing Group at Rochester Institute of Technology for providing us the computing resources necessary to execute a project of this scale.

References

Allamanis M, Sutton C (2013) Mining source code repositories at massive scale using language modeling. In: Proceedings of the 10th working conference on mining software repositories, IEEE Press, Piscataway, NJ, USA, MSR '13, pp 207–216. http://dl.acm.org/citation.cfm?id=2487085.2487127

Belady LA, Lehman MM (1976) A model of large program development. IBM Syst J 15(3):225–252. doi:10.1147/sj.153.0225

Bird C, Nagappan N, Murphy B, Gall H, Devanbu P (2011) Don't touch my code!: examining the effects of ownership on software quality. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th european conference on foundations of software engineering, ACM, New York, NY, USA, ESEC/FSE '11, pp 4–14. doi:10.1145/2025113.2025119

Bissyandé TF, Lo D, Jiang L, Réveillère L, Klein J, Traon YL (2013) Got issues? Who cares about it? A large scale investigation of issue trackers from GitHub. In: 2013 IEEE 24th international symposium on software reliability engineering (ISSRE), pp 188–197. doi:10.1109/ISSRE.2013.6698918

Bissyandé TF, Thung F, Lo D, Jiang L, Réveillère L (2013a) Orion: a software project search engine with integrated diverse software artifacts. In: 2013 18th international conference on engineering of complex computer systems, pp 242–245. doi:10.1109/ICECCS.2013.42

Bissyandé TF, Thung F, Lo D, Jiang L, Réveillère L (2013b) Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In: 2013 IEEE 37th annual computer software and applications conference, pp 303–312. doi:10.1109/COMPSAC.2013.55

Breiman L (2001) Random forests. Mach Learn 45(1):5–32. doi:10.1023/A:1010933404324

CA Technologies (2016) Waffle.io - work better on GitHub issues. https://waffle.io/, accessed: 2016-03-11

Carlo Z (2016) Githut - programming languages and GitHub. http://githut.info, accessed: 2016-03-11

Codetree Studios (2016) Codetree - GitHub issues, managed. https://codetree.com/, accessed: 2016-03-11

Danial A (2014) CLOC – Count lines of code. http://cloc.sourceforge.net/, accessed: 2016-03-11, version: 1.62

de Souza SCB, Anquetil N, de Oliveira KM (2005) A study of the documentation essential to software maintenance. In: Proceedings of the 23rd annual international conference on design of communication: documenting & designing for pervasive information, ACM, New York, NY, USA, SIGDOC '05, pp 68–75. doi:10.1145/1085313.1085331

Dyer R, Nguyen HA, Rajan H, Nguyen TN (2013) Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. In: Proceedings of the 2013 international conference on software engineering, IEEE Press, Piscataway, NJ, USA, ICSE '13, pp 422–431. http://dl.acm.org/citation.cfm?id=2486788.2486844

Eick SG, Graves TL, Karr AF, Marron JS, Mockus A (2001) Does code decay? Assessing the evidence from change management data. IEEE Trans Softw Eng 27(1):1–12. doi:10.1109/32.895984

Emam KE, Benlarbi S, Goel N, Rai SN (2001) The confounding effect of class size on the validity of object-oriented metrics. IEEE Trans Softw Eng 27(7):630–650. doi:10.1109/32.935855

GHTorrent (2016a) Hall of fame. http://ghtorrent.org/halloffame.html, accessed: 2016-03-11

GHTorrent (2016b) The relational DB schema. http://ghtorrent.org/relational.html, accessed: 2016-03-11

GitHub Inc (2016a) Github API v3—github developer guide. https://developer.github.com/v3/, accessed: 2016-03-11

GitHub Inc (2016b) Github archive. https://www.githubarchive.org/, accessed: 2016-06-19

GitHub Inc (2016c) No license - choose a license. http://choosealicense.com/no-license/, accessed: 2016-03-11

Gousios G (2013) The GHTorrent dataset and tool suite. In: Proceedings of the 10th working conference on mining software repositories, IEEE Press, Piscataway, NJ, USA, MSR '13, pp 233–236. http://dl.acm.org/citation.cfm?id=2487085.2487132

Guzman E, Azócar D, Li Y (2014) Sentiment analysis of commit comments in GitHub: an empirical study. In: Proceedings of the 11th working conference on mining software repositories, ACM, New York, NY, USA, MSR 2014, pp 352–355. doi:10.1145/2597073.2597118

HuBoard Inc (2016) Huboard - github issues made awesome. https://huboard.com/, accessed: 2016-03-11

Iowa State University (2016) Publications related to Boa - Boa - Iowa State University. http://boa.cs.iastate.edu/papers/, accessed: 2016-03-11

Jarczyk O, Gruszka B, Jaroszewicz S, Bukowski L, Wierzbicki A (2014) Github projects. Quality analysis of open-source software. Springer International Publishing, Cham, pp 80–94. doi:10.1007/978-3-319-13734-6_6

Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2014) The promises and perils of mining GitHub. In: Proceedings of the 11th working conference on mining software repositories, ACM, New York, NY, USA, MSR 2014, pp 92–101. doi:10.1145/2597073.2597074

Kochhar PS, Bissyandé TF, Lo D, Jiang L (2013) Adoption of software testing in open source projects–a preliminary study on 50,000 projects. In: 2013 17th european conference on software maintenance and reengineering, pp 353–356. doi:10.1109/CSMR.2013.48

Kofink A (2015) Contributions of the under-appreciated: gender bias in an open-source ecology. In: Companion proceedings of the 2015 ACM SIGPLAN international conference on systems, programming, languages and applications: Software for humanity, ACM, New York, NY, USA, SPLASH Companion 2015, pp 83–84. doi:10.1145/2814189.2815369

Laplante P (2007) What every engineer should know about software engineering. What every engineer should know. CRC Press

Mockus A, Fielding RT, Herbsleb J (2000) A case study of open source software development: the apache server. In: Proceedings of the 2000 international conference on software engineering. ICSE 2000 the new millennium, pp 263–272. doi:10.1145/337180.337209

Munaiah N, Kroh S, Cabrey C, Nagappan M (2016a) Home of the reporeapers. https://reporeapers.github.io, accessed: 2016-03-11

Munaiah N, Kroh S, Cabrey C, Parikh N (2016b) Reaper - reference implementation. https://github.com/reporeapers/reaper, accessed: 2016-03-11

Nagappan N (2007) Potential of open source systems as project repositories for empirical studies working group results. Springer, Berlin, pp 103–107. doi:10.1007/978-3-540-71301-2_29

Nagappan N, Williams L, Osborne J, Vouk M, Abrahamsson P (2005) Providing test quality feedback using static source code and automatic test suite metrics. In: 16th IEEE international symposium on software reliability engineering (ISSRE'05), pp 10–94. doi:10.1109/ISSRE.2005.35

Ray B, Posnett D, Filkov V, Devanbu P (2014) A large scale study of programming languages and code quality in github. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, ACM, New York, NY, USA, FSE 2014, pp 155-165. doi:10.1145/2635868.2635922

Rosenberg J (1997) Some misconceptions about lines of code. In: Proceedings fourth international software metrics symposium, pp 137–142. doi:10.1109/METRIC.1997.637174

Ross SM (2003) Peirce's criterion for the elimination of suspect experimental data. J Eng Technol 20(2):38–41

Sajnani H, Saini V, Ossher J, Lopes CV (2014) Is popularity a measure of quality? an analysis of maven components. In: 2014 IEEE international conference on software maintenance and evolution, pp 231–240. doi:10.1109/ICSME.2014.45

Software Freedom Law Center (2012) Managing copyright information within a free software project - software freedom law center. http://softwarefreedom.org/resources/2012/managingcopyrightinformation.html, accessed: 2015-05-15

Syer MD, Nagappan M, Hassan AE, Adams B (2013) Revisiting prior empirical findings for mobile apps: an empirical case study on the 15 most popular open-source android apps. In: Proceedings of the 2013 conference of the center for advanced studies on collaborative research, IBM corp., riverton, NJ, USA, CASCON '13, pp 283–297. http://dl.acm.org/citation.cfm?id=2555523.2555553

Tung YH, Chuang CJ, Shan HL (2014) A framework of code reuse in open source software. In: The 16th asia-pacific network operations and management symposium, pp 1–6. doi:10.1109/APNOMS.2014.6996525

Vasilescu B, van Schuylenburg S, Wulms J, Serebrenik A, van den Brand MGJ (2014) Continuous integration in a social-coding world empirical evidence from GitHub. In: 2014 IEEE international conference on software maintenance and evolution, pp 401–405. doi:10.1109/ICSME.2014.62

Vendome C (2015) A large scale study of license usage on GitHub. In: 2015 IEEE/ACM 37th IEEE international conference on software engineering, vol 2, pp 772–774. doi:10.1109/ICSE.2015.245

Whitehead J, Mistrík I, Grundy J, van der Hoek A (2010) Collaborative software engineering: concepts and techniques. Springer, Berlin, pp 1–30. doi:10.1007/978-3-642-10294-3_1

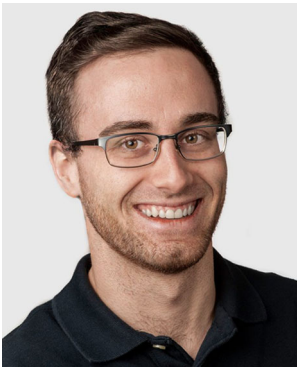
Zaidman A, Rompaey BV, Demeyer S, v Deursen A (2008) Mining software repositories to study co-evolution of production & test code. In: 2008 1st international conference on software testing, verification, and validation, pp 220–229. doi:10.1109/ICST.2008.47

Zenhub (2016) Zenhub - project management for agile teams on GitHub. https://www.zenhub.io/, accessed: 2016-03-11

Zhu H, Hall PAV, May JHR (1997) Software unit test coverage and adequacy. ACM Comput Surv 29(4):366–427. doi:10.1145/267580.267590

Nuthan Munaiah is a Ph.D. student in the B. Thomas Golisano College of Computing and Information Sciences, Rochester Institute of Technology. He has a Bachelors of Engineering degree in Computer Science and Engineering from Visvesvaraya Technological University, India. Prior to starting Graduate school, Nuthan spent five years working as a Software Engineer developing web applications for a popular bank in the New England region of the United States of America. Nuthan's research is focused on analyzing historical vulnerability fixes to identify patterns that may be applied to understanding the engineering failures that led to the vulnerability.

Steven Kroh is a recent alum of the software engineering and honors programs at the Rochester Institute of Technology. He developed an early interest in software through participation in FIRST Robotics, and more recently, the RIT co-op program. Steven now lives in Boston, Massachusetts where he solves engineering challenges in cybersecurity at Carbon Black, Inc.



Craig Cabrey is currently a Production Engineer at Facebook. Previously, he studied at the Rochester Institute of Technology for both his undergraduate Bachelor's and Master's degree in Software Engineering. Under adviser Meiyappan Nagappan, his thesis focused on security vulnerabilities as they relate to modern software engineering projects. Other areas of interest include large scale systems and the applications of Software Engineering principles in industry environments. You can find more information at http://www.craigcabrey.com.

Meiyappan Nagappan is an Assistant Professor in the David R. Cheriton School of Computer Science at the University of Waterloo. His research is centered around the use of large-scale Software Engineering (SE) data to address the concerns of the various stakeholders (e.g., developers, operators, and managers). He received a PhD in computer science from North Carolina State University. Dr. Nagappan has published in various top SE venues such as TSE, FSE, EMSE, and IEEE Software. He has also received best paper awards at the International Working Conference on Mining Software Repositories (MSR '12, '15). He is currently the Editor of the IEEE Software Blog and the Information Director for the IEEE Transactions on Software Engineering. He continues to collaborate with both industrial and academic researchers from the US, Canada, Japan, Germany, Chile, and India. You can find more at mei-nagappan.com.