

# 1 Curating GitHub for Engineered Software 2 Projects

3 Nuthan Munaiah<sup>1</sup>, Steven Kroh<sup>1</sup>, Craig Cabrey<sup>1</sup>, and Meiyappan  
4 Nagappan<sup>2</sup>

5 <sup>1</sup>Department of Software Engineering, Rochester Institute of Technology, 134 Lomb  
6 Memorial Drive, Rochester, NY, USA 14623

7 <sup>2</sup>David R. Cheriton School of Computer Science, University of Waterloo, 200 University  
8 Avenue West, Waterloo, Ontario, Canada N2L 3G1

9 Corresponding author:

10 Nuthan Munaiah and Meiyappan Nagappan

11 Email address: nm6061@rit.edu, mei.nagappan@uwaterloo.ca

## 12 ABSTRACT

13 Software forges like GitHub host millions of repositories. Software engineering researchers have been  
14 able to take advantage of such a large corpora of potential study subjects with the help of tools like  
15 GHTorrent and Boa. However, the simplicity in querying comes with a caveat: there are limited means  
16 of separating the signal (e.g. repositories containing engineered software projects) from the noise  
17 (e.g. repositories containing home work assignments). The proportion of noise in a random sample of  
18 repositories could skew the study and may lead to researchers reaching unrealistic, potentially inaccurate,  
19 conclusions. We argue that it is imperative to have the ability to sieve out the noise in such large repository  
20 forges.

21 We propose a framework, and present a reference implementation of the framework as a tool called  
22 `reaper`, to enable researchers to select GitHub repositories that contain evidence of an engineered  
23 software project. We identify software engineering practices (called dimensions) and propose means  
24 for validating their existence in a GitHub repository. We used `reaper` to measure the dimensions of  
25 1,994,977 GitHub repositories. We then used the data set train classifiers capable of predicting if a  
26 given GitHub repository contains an engineered software project. The performance of the classifiers was  
27 evaluated using a set of 200 repositories with known ground truth classification. We also compared the  
28 performance of the classifiers to other approaches to classification (e.g. number of GitHub Stargazers)  
29 and found our classifiers to outperform existing approaches. We found stargazers-based classifier to  
30 exhibit high precision (96%) but an inversely proportional recall (27%). On the other hand, our best  
31 classifier exhibited a high precision (82%) and a high recall (83%). The stargazer-based criteria offers  
32 precision but fails to recall a significant portion of the population.

## 33 1 INTRODUCTION

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

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

<sup>1</sup>National Science Foundation (NSF) Grant CNS-1513263

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

51 Every source code commit contains a wealth of information that can be used to gain an understanding  
52 of the art of software development. For example, Eick et al. (2001) dived into the rich (fifteen-plus  
53 year) commit history of a large telephone switching system in order to explore the idea of code decay.  
54 Modern day source code repositories provide features that make managing a software project as seamless  
55 as possible. While the integration of features provides improved traceability for developers and project  
56 managers, it also provides MSR researchers with a single, self-contained, organized, and more importantly,  
57 publicly-accessible source of information from which to mine. However, anyone may create a repository  
58 for any purpose at no cost. Therefore, the quality of information contained within the forges may  
59 be diminishing with the addition of many noisy repositories e.g. repositories containing home work  
60 assignments, text files, images, or worse, the backup of a desktop computer. Kalliamvakou et al. (2014)  
61 identified this noise as one of the nine perils to be aware of when mining GitHub data for software  
62 engineering research. The situation is compounded by the sheer volume of repositories contained in these  
63 forges. As of June, 2016, GitHub alone hosts over 38 million repositories<sup>2</sup> and this number is rapidly  
64 increasing.

65 Researchers have used various criteria to slice the mammoth software forges into data sets manageable  
66 for their studies. For example, MSR researchers have leveraged simple filters such as popularity to remove  
67 noisy repositories. Filters like popularity (measured as number of watchers or stargazers on GitHub, for  
68 example) are merely proxies and may neither be general-purpose nor representative of an engineered  
69 software project. Furthermore, MSR researchers should not have to reinvent filters to eliminate unwanted  
70 repositories. There are a few examples of research that take the approach of developing their own filters  
71 in order to procure a data set to analyze:

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

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

81 The assumption that one could make is that the repositories sampled in these studies contain engineered  
82 software projects. However, source code forges are rife with repositories that do not contain source code,  
83 let alone an engineered software project. Kalliamvakou et al. (2014) manually sampled 434 repositories  
84 from GitHub and found that only 63.4% (275) of them were for software development; the remaining  
85 159 repositories were used for experimental, storage, or academic purposes, or were empty or no longer  
86 accessible. The inclusion of repositories containing such non-software artifacts in studies targeting  
87 software projects could lead to conclusions that may not be applicable to software engineering at large.  
88 At the same time, selecting a sample by manual investigation is not feasible given the sheer volume of  
89 repositories hosted by these source code forges.

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

93 The contributions of our work are:

- 94 • A generalizable evaluation framework defined on a set of dimensions that encapsulate typical  
95 software engineering practices;

---

<sup>2</sup><https://github.com/about/press>

- 96 • A reference implementation of the evaluation framework, called *reaper*, available as an open-
- 97 source project (Munaiah et al., 2016c);
- 98 • A publicly-accessible data set of dimensions obtained from 1,994,977 GitHub repositories (Munaiah
- 99 et al., 2016b).

100 The remainder of this paper is organized as follows: we begin by introducing the notion of an  
 101 engineered software project in Section 2. We then propose an evaluation framework in Section 2.1 that  
 102 aims to operationalize the definition of an engineered software project along a set of dimensions. We  
 103 describe the various sources of data used in our study in Section 3. In Section 4, we introduce the eight  
 104 dimensions used to represent a repository in our study. In Section 5, we define propose two variations  
 105 to the definition of an engineered software project, collect a set of repositories that conform to the  
 106 definitions, present approaches to build classifiers capable of identifying other repositories that conform  
 107 to the definition of an engineered software project. The results from validating the classifiers and using  
 108 them to identify repositories that conform to a particular definition of an engineered software project  
 109 from a sample of 1,994,977 GitHub repositories is presented in Section 6. We contrast our study with  
 110 prior literature in Section 7, discuss prior and potential research scenarios in which the data set and the  
 111 classifier could be used in Section 8, and discuss nuances of certain repositories in Section 9. We address  
 112 threats to validity in Section 10 and conclude the paper with Section 11.

## 113 2 ENGINEERED SOFTWARE PROJECT

114 Laplante (2007) defines software engineering as “a systematic approach to the analysis, design, assessment,  
 115 implementation, test, maintenance and reengineering of software”. A software project may be regarded as  
 116 “engineered” if there is discernible evidence of the application of software engineering principles such as  
 117 design, test, maintenance, etc. On similar lines, we define an engineered software project in Definition  
 118 2.1.

119 **Definition 2.1.** An *engineered software project* is a software project that leverages sound software  
 120 engineering practices in each of its dimensions such as documentation, testing, and project management.

121 Definition 2.1 is intentionally abstract; the definition may be customized to align with a set of different,  
 122 yet relevant, concerns. For instance, a study concerned with the extent of testing in software projects  
 123 could define an engineered software project as a software project that leverages sound software testing  
 124 practices. In our study, we have customized the definition of an engineered software project in two  
 125 ways: (a) an engineered software project is similar to the projects contained within repositories owned by  
 126 popular software engineering organizations such as Amazon, Apache, Microsoft and Mozilla and (b) an  
 127 engineered software project is similar to the projects that have a general-purpose utility to users other  
 128 than the developers themselves. We elaborate on these two definitions in the Implementation Section (§5).

### 129 2.1 Evaluation Framework

130 In order to operationalize Definition 2.1, we need to (a) identify the essential software engineering  
 131 practices that are employed in the development and maintenance of a typical software project and (b)  
 132 propose means of quantifying the evidence of their use in a given software project. The *evaluation*  
 133 *framework* is our attempt at achieving this goal.

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

$$136 f(r) = \begin{cases} true & \text{If repository } r \text{ contains an engineered software project} \\ false & \text{Otherwise} \end{cases} \quad (1)$$

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

### 142 3 DATA SOURCES

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

#### 146 3.1 Metadata

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

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

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

166 The database dump used in this study was released on April 1, 2015. The database dump contained  
167 metadata for 16,331,225 GitHub repositories. In this study, we restrict ourselves to repositories in which  
168 the primary language is one of Java, Python, PHP, Ruby, C++, C, or C#. Furthermore, we do not consider  
169 repositories that have been marked as deleted and those that are forks of other repositories. Deleted  
170 repositories restrict the amount of data available for the analysis while forked repositories can artificially  
171 inflate the results by introducing near duplicates into the sample. With these restrictions applied, the size  
172 of our sample is reduced to 2,247,526 repositories.

173 An inherent limitation of the database is the staleness of data. There may be repositories in the  
174 database that no longer exist on GitHub as they may have been deleted, renamed, made private, or blocked  
175 by GitHub.

#### 176 3.2 Source Code

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

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

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

194 commit SHA identified, the state of the cloned repository was reset using the command `git reset`  
195 `--hard {SHA}`.

## 196 4 DIMENSIONS

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

- 200 1. *Architecture*, as evidence of code organization.
- 201 2. *Community*, as evidence of collaboration.
- 202 3. *Continuous integration*, as evidence of quality.
- 203 4. *Documentation*, as evidence of maintainability.
- 204 5. *History*, as evidence of sustained evolution.
- 205 6. *Issues*, as evidence of project management.
- 206 7. *License*, as evidence of accountability.
- 207 8. *Unit testing*, as evidence of quality.

208 In the selection of dimensions, while relevance to software engineering practices was paramount, we  
209 also had to consider aspects such as implementation simplicity and measurement accuracy. We needed  
210 the algorithm to measure each of the dimensions to be generic enough to account for the plethora of  
211 programming languages used in the development of software projects, yet be specific enough to produce  
212 meaningful results. We acknowledge that this list is subjective, and by no means exhaustive; however, the  
213 evaluation framework makes no assumption of either the different dimensions or the way in which the  
214 dimensions are used in determining if a repository contains an engineered software project.

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

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

- 224 • The source code repository being analyzed must be publicly-accessible on GitHub and
- 225 • The primary language of the repository must be one of Java, Python, PHP, Ruby, C++, C, or C#. We  
226 choose these languages based on their popularity on GitHub as reported by GitHub (Carlo Zapponi,  
227 2016).

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

### 231 4.1 Architecture

232 IEEE 1471 defines software architecture as “the fundamental organization of a system embodied in its  
233 components, their relationships to each other and to the environment, and the principles guiding its design  
234 and evolution” (Maier et al., 2001). In effect, software architecture is the high-level structure of a software  
235 system that depicts the relationships between various components that compose the system.

236 Software architecture comes in all shapes, sizes, and complexities. There is no one fixed architecture  
237 to which all software systems adhere; however, software systems that employ some form of architecture  
238 have discernible relationships between components that compose the system. These components can be  
239 as granular as functions and as coarse as entire binaries. For our purposes, any software system that has  
240 well-defined relationships between its components can be said to have an architecture. The presence of an  
241 architecture indicates that some form of design was employed in the development of the software project.  
242 Consequently, the software project may contain further evidence of being an engineered software project.

<sup>3</sup><https://github.com/RepoReapers/reaper/wiki/Extending-reaper>



## 243 Metric

244 We approximate repository architecture as an undirected graph in which nodes represent files in the  
 245 repository and edges represent relationships between the files. We hypothesize that a well-architected  
 246 software system, at the very least, will have a majority of its files related to one another. We propose  
 247 a metric, *monolithicity*, to quantify the extent to which source files in the repository are related to one  
 248 another.

249 **Definition 4.1.** *Monolithicity* is the ratio of the number of nodes in the largest connected sub-graph to the  
 250 number of nodes in the entire architecture graph.

251 There is a specific case that must be handled when calculating *monolithicity*. For example, certain  
 252 projects may have only one source file present in the repository. Zazworka et al. (2011) investigated the  
 253 impact of using god classes in the architecture of a software system and found that their inclusion has a  
 254 detrimental effect on that system's quality. Thus, we consider a repository with a single source file to  
 255 have no architecture.

## 256 Approach

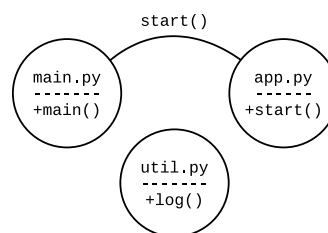
257 Bowman et al. (1999) designed a methodology to extract the high-level architecture of the Linux  
 258 kernel. The researchers began by grouping source files into modules based on naming and directory  
 259 structure. Individual source file relationships were promoted to relationships between the modules to  
 260 produce an architectural overview of the Linux kernel. We use a similar approach to reverse engineer  
 261 software architecture from the source code of a software system.

262 We use a popular syntax highlighter package called Pygments (Georg Brandl and Pygments Contrib-  
 263 utors, 2016) to identify symbol definition and usage in source files. Although Pygments' primary use  
 264 case is as a syntax highlighter, the lexer that powers it internally is suited for our purposes. Furthermore,  
 265 Pygments supports lexical analysis of over 300 programming languages ensuring our implementation can  
 266 scale beyond the languages chosen in this study.

267 A repository may contain files with source code in multiple programming languages. We use  
 268 `ack` (Lester, 2014) to obtain a list of files that contain code in the primary language (as noted in the  
 269 database) of the repository. We then follow a two pass approach to construct the architecture graph.

- 270 1. Pygments lexically analyzes each source file identified by `ack` to collect the symbols defined and  
 271 referenced by the file. The file, along with a list of its symbol definitions and references, is added  
 272 as a node to the graph.
- 273 2. Iterating through all the nodes in the graph, an edge is added between a node A, and a node B, if at  
 274 least one symbol referenced by A is defined by B.

275 Figure 1 shows the architecture graph constructed from an example repository containing three source  
 276 files: `main.py`, `app.py`, and `util.py`. Each node displays a file name above a list of externally  
 277 visible symbols. The edges represent cross-file symbol references. The monolithicity of this graph is  
 278  $2/3 = 0.66$ , as the largest connected component has two files (`main.py` and `app.py`).



**Figure 1.** Example view of an extracted architecture

279 While simple, this approach does have limitations, notably:

- 280 • **Dynamic Loading:** Lack of support for programming languages that support dynamic loading  
 281 e.g. JavaScript. JavaScript is a dynamic language which makes it difficult to determine symbol  
 282 references among source files purely from analyzing the contents of the source files.

- 283 • Over-approximation: Since the approach relies purely on the name of the symbols, the resulting  
284 architecture may depict relationships that may not occur in the source code e.g., when multiple files  
285 define symbols with the same name.
- 286 • Language Bias: Since the approach only considers files containing source code in the primary  
287 language of the repository, there is a possibility whereby a significant portion of source code  
288 in the repository may be excluded. For example, a repository meant for the development of a  
289 Django application may be flagged with JavaScript being the primary language if the percentage  
290 of JavaScript code is higher than that of Python, perhaps by the inclusion of vendor files such as  
291 jquery.js or bootstrap.js in the repository.

292 An alternative to Pygments may have been to use static analysis tools to generate a call graph from  
293 which the monolithicity metric may be estimated. Such an approach would involve (a) identifying  
294 an appropriate static analysis tool from among the plethora of tools available for each of the seven  
295 programming languages considered in our study and (b) developing a parser to transform the output from  
296 each of the identified static analysis tools to a graph.

297 As an exploratory exercise, we computed the difference between the value of the monolithicity metric  
298 estimated using the Pygments approach and that estimated using the static analysis approach. In this  
299 exercise, we restricted ourselves to the C programming language because a Python script capable of  
300 parsing a call graph generated by GNU cflow—a static call graph generation utility for programs written  
301 in C<sup>4</sup>—was available as a component in an open-source project called the Attack Surface Meter (Munaiah  
302 et al., 2016a). The empirical data set in this exercise was composed of 50 randomly selected repositories in  
303 which the primary programming language was C. The median difference in the value of the monolithicity  
304 metric was 0.1384 (with a standard deviation of 0.2256) with the Pygments-based approach producing  
305 lower values for the monolithicity metric. While the approach to estimating monolithicity using the  
306 static analysis approach is more accurate, the implementation overhead is substantial, deterring us from  
307 pursuing this approach any further.

308 The Pygments-based approach does have one implementation concern: computational complexity.  
309 The algorithm to generate the architecture graph has a computational complexity of  $O(n^3)$ . In the case of  
310 some very large projects, the approach was not feasible to get a result in a reasonable amount of time.  
311 Therefore, the reference implementation includes a timeout period. If the tool is unable to calculate the  
312 monolithicity of a project within the designated time period, the calculation is halted and the repository  
313 receives a no score for the architecture dimension alone.

## 314 4.2 Community

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

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

### 323 Metric

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

327 **Definition 4.2.** *Core contributors* is the cardinality of the smallest set of contributors whose total number  
328 of commits to a source code repository accounts for 80% or more of the total contributions.

### 329 Approach

330 The notion of *core contributors* is prevalent in open source software where a set of contributors take  
331 ownership of and drive a project towards a common goal. Mockus et al. (2000) have applied this concept

<sup>4</sup><http://www.gnu.org/software/cflow/>

332 in their study of open source software development practices. The definition of core contributors is the  
 333 same as that of core developers as defined by Syer et al. (2013).

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

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

### 346 4.3 Continuous Integration

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

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

#### 355 Metric

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

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

#### 358 Approach

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

### 365 4.4 Documentation

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

#### 373 Metric

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

**Definition 4.3.** *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  $r$ .

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



### 376 Approach

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

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

### 388 4.5 History

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

### 394 Metric

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

**Definition 4.4.** *Commit frequency* is the average number of commits per month.

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

397 Where,

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

### 400 Approach

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

### 404 4.6 Issues

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

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

### 421 Metric

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

**Definition 4.5.** *Issue frequency* is the average number of issue events transpired per month.

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

425 Where,

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

#### 428 Approach

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

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

#### 435 4.7 License

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

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

#### 448 Metric

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

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

#### 450 Approach

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

454 The GitHub License API is limited in its capabilities in that it does not consider license information  
 455 contained in README.md or in source code files. Furthermore, the API is still in "developer preview"  
 456 and as a result may be unreliable. On the other hand, any improvements in the capabilities of the  
 457 API is automatically reflected in our approach. In the interim, however, we have overcome some of  
 458 the limitations by analyzing the files in a source code repository for license information. We identify  
 459 license information by searching repository files for excerpts from the license text of 12 most popular  
 460 open source licenses on GitHub. For example, we search for "The MIT License (MIT)" to detect the  
 461 presence of The MIT License. The 12 chosen licenses were enumerated by the GitHub License API  
 462 (<https://api.github.com/licenses>). We note that the interim solution implemented may

463 have its own side-effect in cases where a repository, with no license of its own, includes source code files  
464 of an external library instead of defining the library as a dependency. If any of the external library source  
465 code files contain excerpts of the license we search for, the license dimension may falsely indicate the  
466 repository to contain a license.

#### 467 4.8 Unit Testing

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

#### 483 Metric

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

**Definition 4.6.** *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.

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

485 Where,

- 486 • *slotc* is the number of source lines of code in test files in the repository *r*
- 487 • *sloc* is the number of source lines of code in all source files in the repository *r*

#### 488 Approach

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

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

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

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

## 507 5 IMPLEMENTATION

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

### 512 5.1 Training Data Sets

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

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

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

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

#### 531 5.1.1 Organization Data Set

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

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

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

#### 551 5.1.2 Utility Data Set

552 Unlike the process of identifying the repositories that compose the organization data set, the process  
553 of identifying the repositories that compose the utility data set was non-trivial. The repositories that  
554 composed the utility data set were identified by manually evaluating a random sample from the 1,994,977  
555 repositories that were analyzed by `reaper`. Similar to the process of composing the organization data set,  
556 we used a set of guidelines for deciding if a repository should be included or not. The guidelines dictated  
557 the various aspects that were to be considered in deciding whether a repository has a general-purpose

558 utility. The guidelines used here were more subjective than those used in the process of composing the  
 559 organization data set. Some of the guidelines used were (a) repository contains sufficient documentation  
 560 to enable the project contained within to be used in a general-purpose setting, (b) repository contains an  
 561 application or service that is used by or has the potential to be used by people other than the developers, (c)  
 562 repository does not contain cues indicating that the source code contained within may be an assignment.  
 563 The potential for bias was mitigated by two authors independently evaluating the same random sample of  
 564 repositories. The first 150 repositories that both authors agreed to include composed the utility data set.

565 `brandur/casseo`, `stephentu/forwarder`, `smcameron/opencscad`, and `apanzerj/zit`  
 566 are some examples of repositories included in the utility data set that are known to contain engineered  
 567 software projects.

568 The organization data set is available for download as a CSV file—`utility.csv`—from GitHub Gist acces-  
 569 sible at <https://gist.github.com/nuthanmunaiiah/23dba27be17bbd0abc40079411dbf066>.

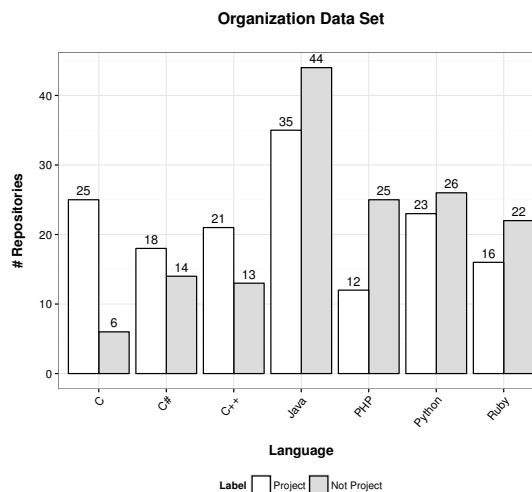
### 570 5.1.3 Negative Instances Data Set

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

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

### 578 5.1.4 Data Sets Summary

579 In this section, we summarize the training data sets using visualizations. The repositories that compose the  
 580 organization and utility data sets are labeled as “project” and the repositories that compose the negative  
 581 instances data set are labeled as “not project”. Shown in Figures 2 and 3 is the number of repositories of  
 582 each kind (“project” and “not project”) grouped by programming language in the organization and utility  
 583 data sets, respectively.



**Figure 2.** Number of repositories in the organization data set grouped by programming language

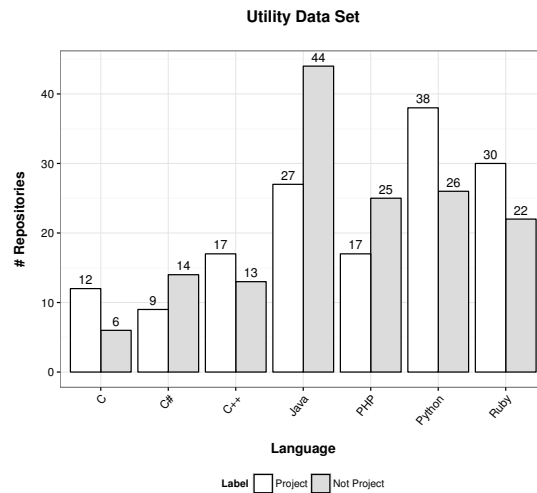
584 The distribution of number of source lines of code (SLOC) in and number of stargazers of repositories  
 585 of each kind (“project” and “not project”) in organization and utility data sets are shown in Figures 4 and  
 586 5, respectively.

587 Additionally, shown in Figure 6 and 7 are the distributions of all eight dimensions obtained from the  
 588 repositories in the organization and utility data sets, respectively.

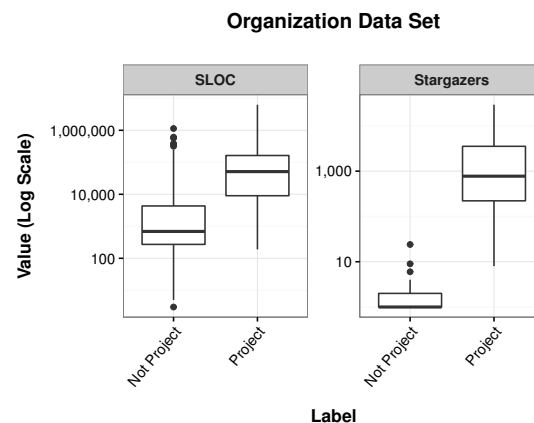
## 589 5.2 Approaches

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





**Figure 3.** Number of repositories in the utility data set grouped by programming language



**Figure 4.** Distribution of SLOC in and number of stargazers of repositories in the organization data set

### 592 5.2.1 Score-based Classifier

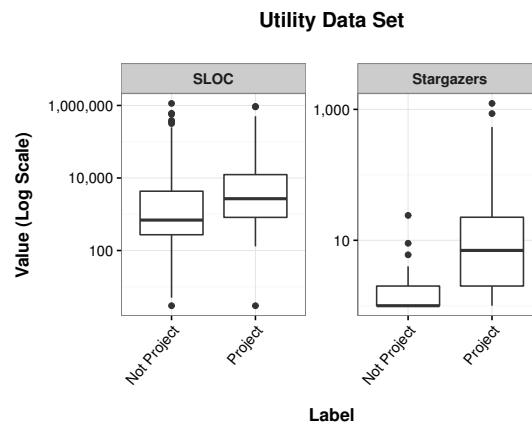
593 The score-based classifier is a custom approach to implementing the evaluation framework that allows complete control over the classification. In this approach, the boolean-valued function from (1) (Section 2.1, 594 takes the form shown in Equation (6). 595

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

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

596 Where,

- 597 •  $r$  is the repository to classify
- 598 •  $D$  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.
- 599 •  $M_d$  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 600 quantifies documentation. 601 602



**Figure 5.** Distribution of SLOC in and number of stargazers of repositories in the utility data set

- 603 •  $t_d$  is a threshold that must be satisfied by the corresponding metric,  $M_d$ , for the repository,  $r$ , to be
- 604 considered engineered along the dimension,  $d$ . For example, having a sufficiently high proportion of
- 605 comment lines to source lines may indicate that the project is engineered along the documentation
- 606 dimension.
- 607 •  $h_d(M_d, t_d)$  is a heuristic function that evaluates to 1 if the metric value,  $M_d$ , satisfies the correspond-
- 608 ing threshold requirement,  $t_d$ , 0 otherwise.
- 609 •  $w_d$  is the weight that specifies the relative importance of each dimension  $d$ .
- 610 •  $score_{ref}$  is the reference score i.e. the minimum score that a repository must evaluate to in order to
- 611 be considered to contain an engineered software project.

612 In the case of the score-based classifier, the training data set is used to determine the thresholds,

613  $t_d$ , and compute the reference score,  $score_{ref}$ . For all repositories in each of the two training data

614 sets (plus the repositories from the negative instances data set), we collected the eight metric values.

615 Outliers were eliminated using the Peirce criterion (Ross, 2003). For the boolean-valued metrics, 1 (i.e.

616 True) is the threshold. For all other metrics, the minimum non-zero metric value was chosen to be the

617 corresponding threshold. The threshold values corresponding to each of the eight dimensions, established

618 from repositories in each of the two training data sets, are shown in Table 1. Also shown in Table 1 are

619 the relative weights that we have used in our score-based classifier. We note that these weights, while

620 subjective, are an acceptable default. Furthermore, we also considered the limitations in collecting the

621 value of the associated metric from a source code repository when deciding the weights. For example,

622 the approach to evaluating a source code repository along the architecture dimension is more robust and

623 thus its weight is higher than the unit testing dimension, where there are inherent limitations owing to our

624 non-exhaustive set of framework signatures.

625 The weights and thresholds from Table 1 were used to compute the scores of all repositories in the

626 organization and utility data sets. The distribution of the scores is shown in Figure 8. The reference score

627 in the organization and utility training data sets was found to be 65 and 30, respectively.

628 The score-based classifier approach is flexible and enables a finer control over the classification. The

629 weights, in particular, enable the implementer to explicitly define the importance of each dimension. In

630 effect, Equation (6) may be tailored to implement a variety of different classifiers using different set of

631 dimensions,  $D$ , and corresponding metrics, thresholds, and weights. For instance, if there is a need to

632 build a classifier that considers gender bias in the acceptance of contributions in open-source community

633 (like in the work by Kofink (2015)), one could introduce a new dimension, say *bias*, define a metric

634 to quantify gender bias in a repository, identify an appropriate threshold, and weight the dimension in

635 relation to other dimensions that may be pertinent to the study.

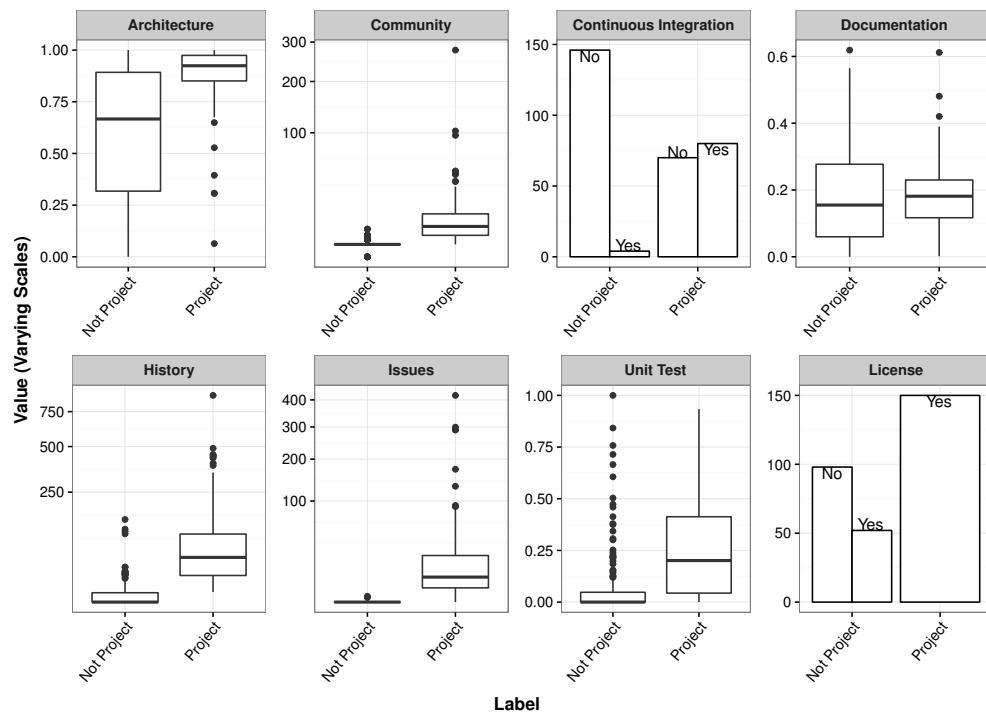
### 636 5.2.2 Random Forest Classifier

637 Random forest classifier is a tree-based approach to classification in which multiple trees are trained such

638 that each tree casts a vote which is then aggregated to produce the final classification (Breiman, 2001).

639 The random forest classifier is simpler to implement but the simplicity comes at a loss of finer control

## Distribution of Dimensions of Repositories in Organization Data Set



**Figure 6.** Distribution of the dimensions of repositories in the organization data set

640 over the classification. The training data set is the only way to affect the classifier performance. While the  
 641 implementer has the option to ignore dimensions when training the classifier, there is no way to express  
 642 relative weighting of dimensions as supported in the score-based classifier.

## 643 6 RESULTS

644 In this section, we present (a) the results from the validation of the classifiers and (b) the results from  
 645 applying the classifiers to identify (or predict) engineered software projects in a sample of 1,994,977  
 646 GitHub repositories. Since we have two different classifiers (score-based and random forest) trained using  
 647 two different data sets (organization and utility), the validation and prediction analysis is repeated four  
 648 times.

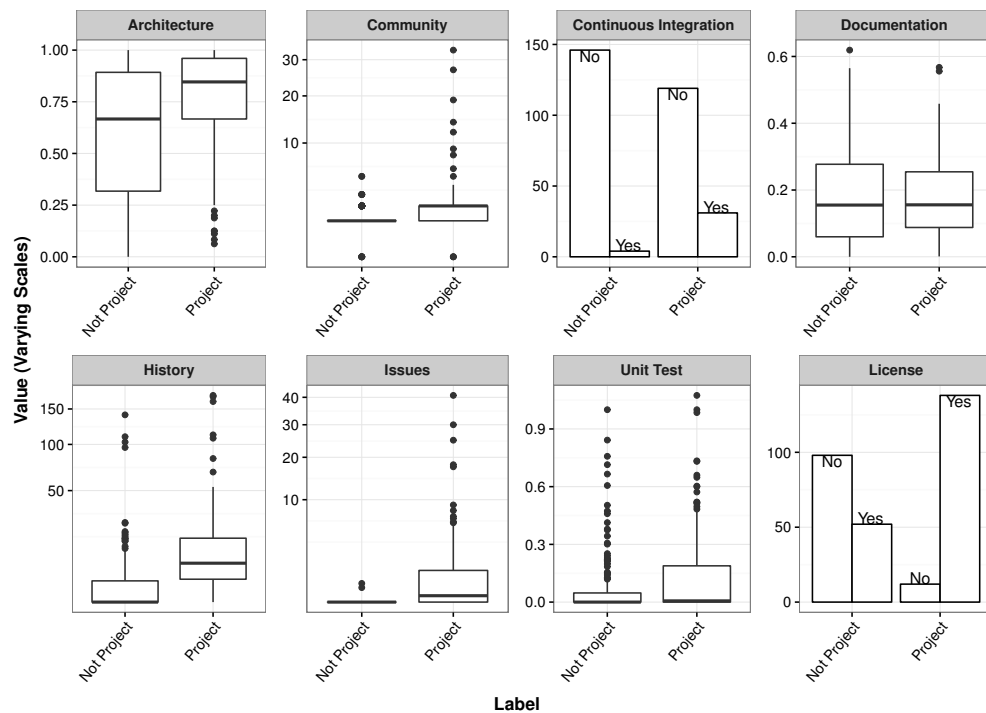
### 649 6.1 Validation

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

#### 658 6.1.1 Establishing the Ground Truth

659 The performance evaluation of any classifier typically involves using the classifier to classify a set of  
 660 samples for which the ground truth classification is known. On similar lines, to evaluate the performance  
 661 of the score-based and random forest classifiers, we manually composed a set of 200 repositories (100  
 662 repositories that have been assessed to contain an engineered software project and 100 repositories that do  
 663 not). We followed a process similar to that followed in identifying the repositories to compose the utility

Distribution of Dimensions of Repositories in Utility Data Set



**Figure 7.** Distribution of the dimensions of repositories in the utility data set

664 data set. To ensure an unbiased evaluation, the validation set was independently evaluated by two authors  
 665 and only those repositories that both authors agreed on were included and appropriately labeled.

666 Shown in Figure 9 is the distribution of the eight dimensions collected from repositories in the  
 667 validation set. As seen in the figure, repositories known to contain engineered software project tend to  
 668 have higher median values in almost all dimensions.

669 The validation set is available for download as a CSV file—validation.csv—from GitHub Gist accessi-  
 670 ble at <https://gist.github.com/nuthanmunaiyah/23dba27be17bbd0abc40079411dbf066>.

### 671 6.1.2 Internal Validation

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

### 674 Organization Data Set

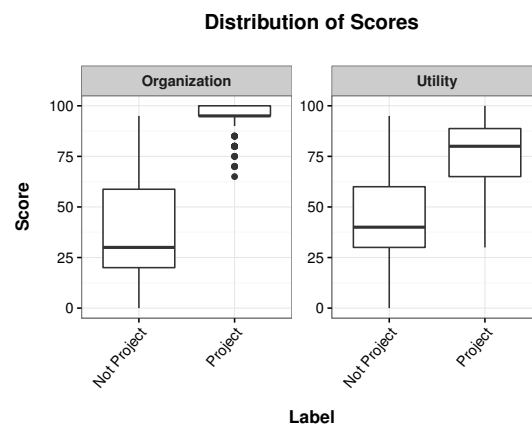
675 The performance of score-based and random forest classifiers trained with organization data set is  
 676 shown in Table 2.

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

- 682 1. **False Positive** - `software-engineering-amsterdam/sea-of-ql` is the quintessential  
 683 example of a repository that should not be included in a software engineering study because it is  
 684 essentially a collaboration space where all students enrolled in a particular course develop their  
 685 individual software projects. The repository received a score of 95 which is closer to a perfect score  
 686 of 100. Analyzing the dimensions of the repository, we found, quite unsurprisingly, that almost all  
 687 dimensions satisfied the threshold requirement. The repository was contributed to by 24 developers  
 688 with an average of over 330 commits made every month. Although the repository does not have a

**Table 1.** Dimensions and their corresponding weights, metrics, and thresholds (established from the organization and utility training data sets)

Dimension ( $d$ )	Metric ( $M_d$ )	Weight ( $w_d$ )	Threshold ( $t_d$ )	
			Organization	Utility
Architecture	Monolithicity	20	6.4912E-01	6.2500E-02
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	1.5000E-01
Issues	Issue Frequency	5	2.2989E-02	1.1905E-02
License	Evidence of License	10	1	1
Unit Test	Test Ratio	10	1.0160E-03	1.4200E-04



**Figure 8.** Distribution of scores for repositories in the organization and utility data sets

689 license, the limitation of our implementation of the license dimension seems to have identified a  
 690 license in library files that may have been included in the source code repository.  
 691 2. **False Negative** - `kzoll/ztlogger` is a repository that contains PHP scripts to log website  
 692 traffic information. The repository received a score of 20 which is considerably lower than the  
 693 reference score of 65. Analyzing the dimensions of the repository, we found that the repository was  
 694 contributed to by a single developer, had limited architecture, did not use continuous integration,  
 695 issues or unit testing.

#### 696 Utility Data Set

697 The performance of score-based and random forest classifiers trained with utility data set is shown in  
 698 Table 3.

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

704 1. **False Positive** - `mer-packages/qtgraphicaleffects` is a repository that contains source  
 705 code for certain visual items that may be used with images or videos. The repository was manually



Distribution of Dimensions of Repositories in Validation Set

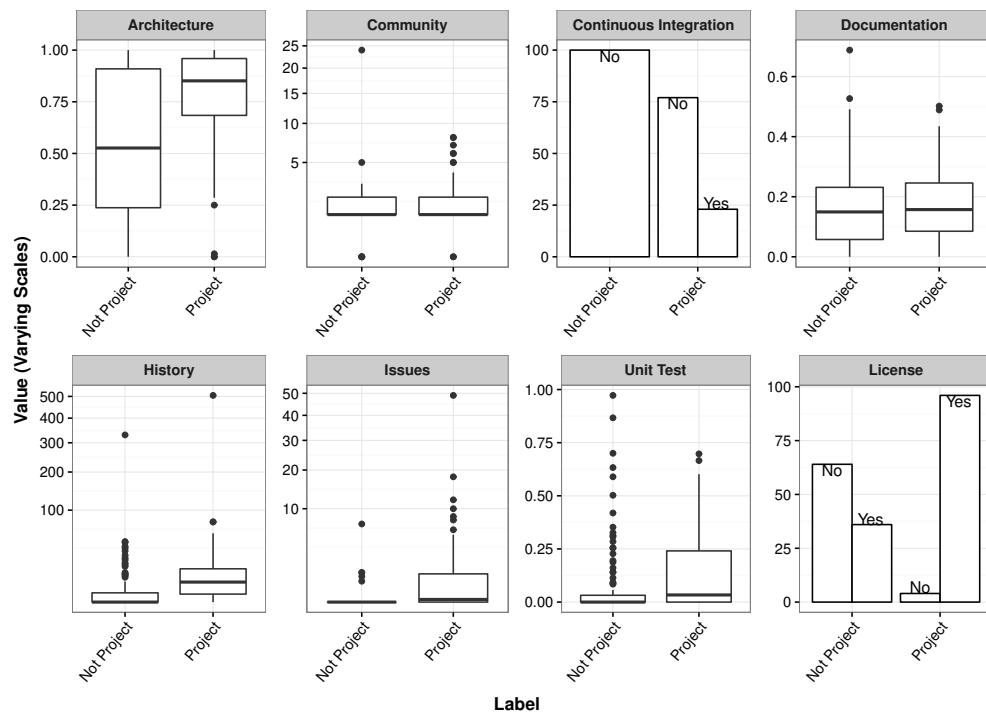


Figure 9. Distribution of dimensions of repositories in the validation set

Table 2. Performance of score-based and random forest classifiers trained with organization data set

Classifier	FPR	FNR	Precision	Recall	F-measure
Score-based	14%	32%	83%	68%	75%
Random forest	4%	59%	91%	41%	57%

706 classified as not containing an engineered software project because of the unusual repository orga-  
 707 nization. Furthermore, looking at other repositories owned by the `mer-packages` organization,  
 708 it appears that the repository may actually be a copy of the source code of the Qt Graphical Effects  
 709 module,<sup>5</sup> being ported to the Mer<sup>6</sup> mobile platform. The repository received a score of 95. Ana-  
 710 lyzing the dimensions of the repository, we found almost all dimensions satisfied the threshold  
 711 requirement. Although the repository was not manually classified as containing an engineered  
 712 software project, the inclusion of this particular repository may not be a problem as the project,  
 713 possibly a clone from a different repository, has a general-purpose utility.

714 2. **False Negative** - `EsotericSoftware/dnsmadeeasy` is a repository that contains a simple  
 715 Java tool that periodically updates the IP addresses in the DNS servers maintained by DNS Made  
 716 Easy.<sup>7</sup> Clearly, the tool has a general-purpose utility for any customer using the DNS Made Easy  
 717 service. However, the repository received a score of 10. Analyzing the source code contained in  
 718 and the dimensions of the repository, we found that the project is too simple. The architecture  
 719 dimension could not be computed because the repository contains only a single source file which  
 720 has no source code comments. The repository does have a license but does not use continuous  
 721 integration, unit testing, or issues. However, the ground truth classification of this repository was  
 722 that it contained an engineered software project because of the utility of the Java tool.

<sup>5</sup><http://doc.qt.io/qt-5/qtgraphicaleffects-index.html>

<sup>6</sup><http://merproject.org/>

<sup>7</sup><http://www.dnsmadeeasy.com/>

**Table 3.** Performance of score-based and random forest classifiers trained with utility data set

Classifier	FPR	FNR	Precision	Recall	F-measure
Score-based	88%	1%	53%	99%	69%
Random forest	18%	17%	82%	83%	83%

### 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.<sup>8</sup> 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 2 and 3 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 1,994,977 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 stars. 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 4. 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.

**Table 4.** 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%	89%	100%	11%	20%
10	1%	73%	96%	27%	42%

As seen in Table 4, at high thresholds (1,123 and 500) the stargazers-based classifier misclassifies all repositories known to contain engineered software projects. As we lower 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 4. In terms of F-measure, the random forest classifier trained

<sup>8</sup>Citation counts retrieved from Google Scholar

756 using the utility data set exhibited the best performance. Similarly, the stargazer-based classifier with 10  
 757 as the threshold for number of stargazers performed the best. We compare these two classifiers not in  
 758 terms of the performance evaluation metrics but in terms of the actual predicted classification. In Table 5,  
 759 we present the percentage of repositories where (a) both classifiers agreed with the ground truth, (b) both  
 760 classifiers disagreed with the ground truth, (c) prediction by random forest classifier matches the ground  
 761 truth but that of stargazers-based classifier does not, and (d) prediction by stargazers-based classifier  
 762 matches the ground truth but that by random forest classifier does not.

**Table 5.** 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	26%	17%	83%	27%
Not project	82%	1%	82%	99%

763 For repositories that we believe to be useful in MSR data sets, both approaches incorrectly classify the  
 764 repositories as “not project” 17% of the time. However, random forest classifies 83% of the repositories  
 765 correctly when the stargazers classifies only 27% of the repositories. A prime example of a project  
 766 that was missed by stargazers-based classifier but correctly classified by the random forest classifier  
 767 is `jrubby/jruby-ldap` from the team that maintains the Ruby implementation of the Java Virtual  
 768 Machine (JVM). The repository contains a Ruby gem for LDAP support in JRuby. Our random forest  
 769 classifies the repository as a project due to its architecture, commit history, test suite, and documentation,  
 770 among other factors. However, the repository has only 7 stars. While the stargazers-based approach  
 771 misses `jrubby/jruby-ldap`, this repository may be a worthy candidate in a software engineering  
 772 study. We also note that there was only one case in which stargazers-based classifier predicted a repository  
 773 to be a project while random forest classifier did not. Hence, any repository classified as a project by the  
 774 stargazers-based classifier is highly likely to be classified the same by the random forest classifier as well.

775 In the case where the ground truth classification is “not project”, 82% of the time, both approaches  
 776 correctly classified repositories as “not project”. In addition, the stargazers-based classifier correctly  
 777 classified repositories as “not project” 99% of the time where the random forest classifier did so 82%  
 778 of the time. Consider the repository `liorkesos/drupalcamp`, which has sufficient documentation,  
 779 commit history, and community to be classified as a “project” by the random forest classifier, however,  
 780 the repository is essentially a collection of static PHP files of a Drupal Camp website, not incredibly useful in a  
 781 general software engineering study. The stargazers-based classifier predicts `liorkesos/drupalcamp`  
 782 as “not project” only for its lack of stars.

### 783 Summary

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

### 793 6.2 Prediction

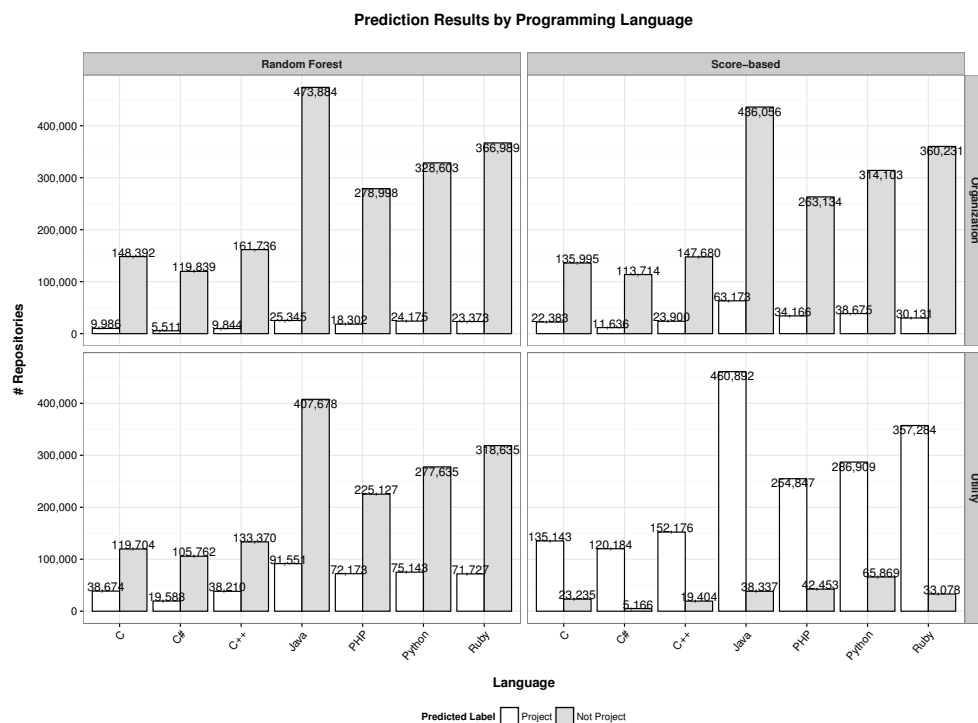
794 In this section, we present the results from applying the score-based and random forest classifiers to  
 795 identify engineered software projects in a sample of 1,994,977 GitHub repositories. Shown in Table 6  
 796 are the number of repositories classified as containing an engineered software project by the score-based  
 797 and random forest classifiers. With the exception of the score-based classifier trained using the utility  
 798 data set, the number of repositories classified as containing an engineered software project is, on average,  
 799 12.45% of the total number of repositories analyzed. We can also see from Table 6 that there are far fewer

800 repositories similar to the ones owned by software development organizations than there are repositories  
 801 similar to the ones that have a general-purpose utility. The number of repositories predicted to be similar  
 802 to the ones that have a general-purpose utility by the score-based classifier is considerably high. A likely  
 803 explanation for the unusually high number of repositories could be because of the relatively low reference  
 804 score of 30 established from the utility data set.

**Table 6.** 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	224,064 (11.23%)
	Random forest	118,073 (5.92%)
Utility	Score-based	1,767,435 (88.59%)
	Random forest	402,815 (20.19%)

805 Shown in Figure 10 is a grouping of results by programming language.



**Figure 10.** Number of repositories classified by the score-based and random forest classifiers grouped by programming language

806 As mentioned in Section 4.1 (Architecture), the computational complexity may prevent the collection  
 807 of the monolithicity metric for certain large repositories. There were 4,451 such repositories in our data set  
 808 (a mere 0.22% of the total number of repositories). On average, 1,770 of the 4,451 repositories (39.77%)  
 809 were classified as containing an engineered software project with the architecture dimension defaulted to  
 810 zero.

811 The entire data set may be viewed and downloaded as a CSV file from <https://reparepers.github.io>.  
 812 The data set includes the metric values collected from each repository. The data set  
 813 available online contains information pertaining to 2,247,526 GitHub repositories but 252,105 of those  
 814 repositories were inactive at the time reaper was run and as a result the metric values will all be NULL.

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

## 817 7 RELATED WORK

818 An early work by Nagappan (2007) revealed opportunities and challenges in studying open source  
819 repositories in an empirical context. Kalliamvakou et al. (2014) described various perils of mining  
820 GitHub data; specifically, Peril IV is: “A large portion of repositories are not for software development”.  
821 In this work, the researchers manually analyzed a sample of 434 GitHub repositories and found that  
822 approximately 37% of them were not used for software development. In our study, the best performing  
823 classifier predicted that approximately 20.19% of 1,994,977 GitHub repositories contain engineered  
824 software projects. Our prediction results highlight the reality that even though 63% of GitHub repositories  
825 are used for software development, only a small percentage of those repositories actually contain projects  
826 that software engineering researchers may be interested in studying.

827 Dyer et al. (2013) and Bissyandé et al. (2013) have created domain specific languages—Boa and  
828 Orion, respectively—to help researchers mine data about software repositories. Dyer et al. (2013) have  
829 used Boa to curate a sizable number of source code repositories from GitHub and SourceForge, however,  
830 only Java repositories are currently available. In contrast, we have curated over 1,994,977 spanning seven  
831 programming languages with the intention of simplifying the process of study selection in large-scale  
832 source code mining research.

833 On the open source community front, Ohloh.net (now Black Duck Open Hub) is a publicly-editable  
834 directory of free and open source software projects. The directory is curated by Open Hub users, much  
835 like a public wiki, resulting in an accurate and up-to-date directory of open source software. The website  
836 provides interesting visualization about software projects curated by Black Duck Open Hub. Tung et al.  
837 (2014) have used Open Hub to search for repositories containing engineered software projects as perceived  
838 by Open Hub users.

## 839 8 USAGE SCENARIOS

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

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

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



865 The aforementioned studies from prior literature focused on specific aspects of software development  
 866 such as licensing, testing, or issue tracking. While the score-based and random forest classifiers may  
 867 be used to support such studies, the real benefit of the classifiers may only be evident when researchers  
 868 need access to repositories that simultaneously satisfy a variety of requirements. For instance, consider  
 869 the following hypothetical studies in which the classifiers would prove useful in identifying a set of  
 870 repositories to include.

- 871 • A study investigating the relationship between collaboration and testing in open source projects  
 872 could use the community and unit testing dimensions to identify repositories.
- 873 • A study investigating the evolution of documentation in open source projects could use history and  
 874 documentation dimensions to identify repositories.

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

## 877 9 DISCUSSION

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

885 The validation set contains 200 repositories, 100 of which are known to contain engineered software  
 886 project and the remaining 100 are known to not contain engineered software project. 45 of the 200  
 887 repositories are owned by organizations.

888 Shown in Figure 11 is a comparison between the distribution of the eight dimensions collected from  
 889 repositories owned by organizations but having different manual classification labels. As seen in the  
 890 figure, the difference in the distribution of the dimensions provides qualitative evidence to support the  
 891 notion that not all repositories owned by organizations are similar to one another.

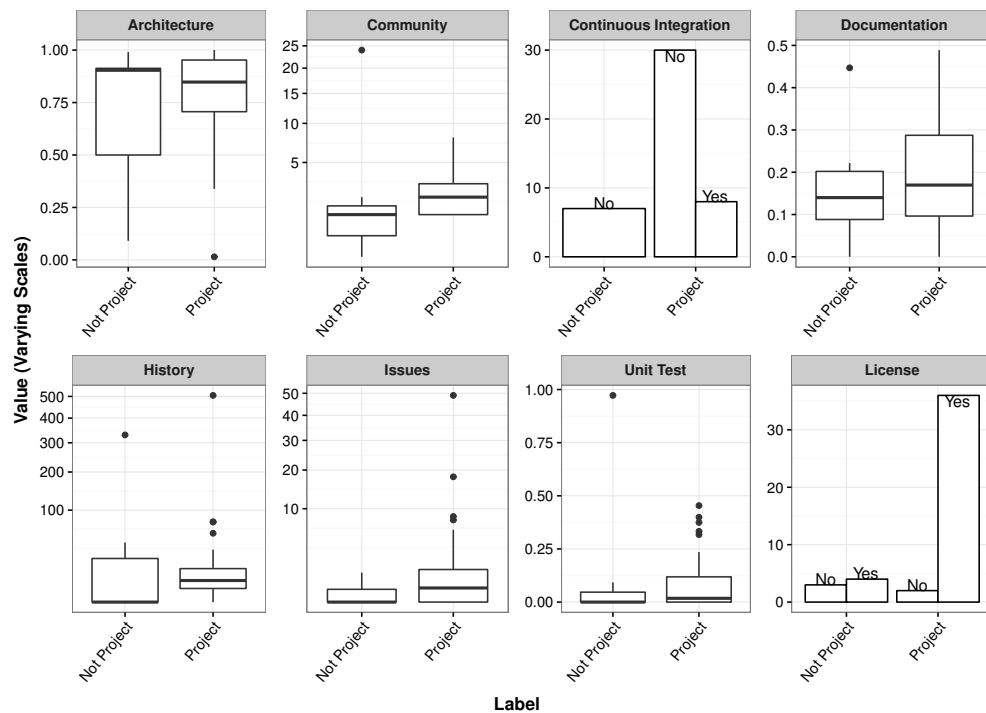
892 On similar lines, we compared the distribution of the eight dimensions collected from repositories  
 893 known to contain engineered software project but owned by organizations and users. The comparison is  
 894 shown in Figure 12. As seen in the figure, the medians of most dimensions are comparable between the  
 895 repositories owned by users to that owned by organizations. The similarity in dimensions is exactly the  
 896 aspect that our approach aims to take advantage of.

897 Shown in Table 7 is a break down of the prediction results from Section 6.2 into repositories owned by  
 898 organizations and users. As seen in the table, a considerable number of repositories that were classified  
 899 as containing engineered software project are owned by individual users. On the other hand, a sizable  
 900 number of repositories that were classified as not containing an engineered software project were owned  
 901 by organizations. In effect, filtering repositories based solely on the owner being an organization may  
 902 lead to the exclusion of potentially relevant, user-owned, repositories or the inclusion of repositories that  
 903 may not contain engineered software project or both.

**Table 7.** 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	64,649	159,415
	not project	157,034	1,613,879
Random forest	project	41,747	71,639
	not project	179,936	1,701,655

## Distribution of Dimensions of Repositories Owned by Organizations



**Figure 11.** Comparing the distribution of dimensions of repositories with different manual classification labels but all owned by organizations

## 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 eight dimensions we have used to be an acceptable default. 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., 2016c).

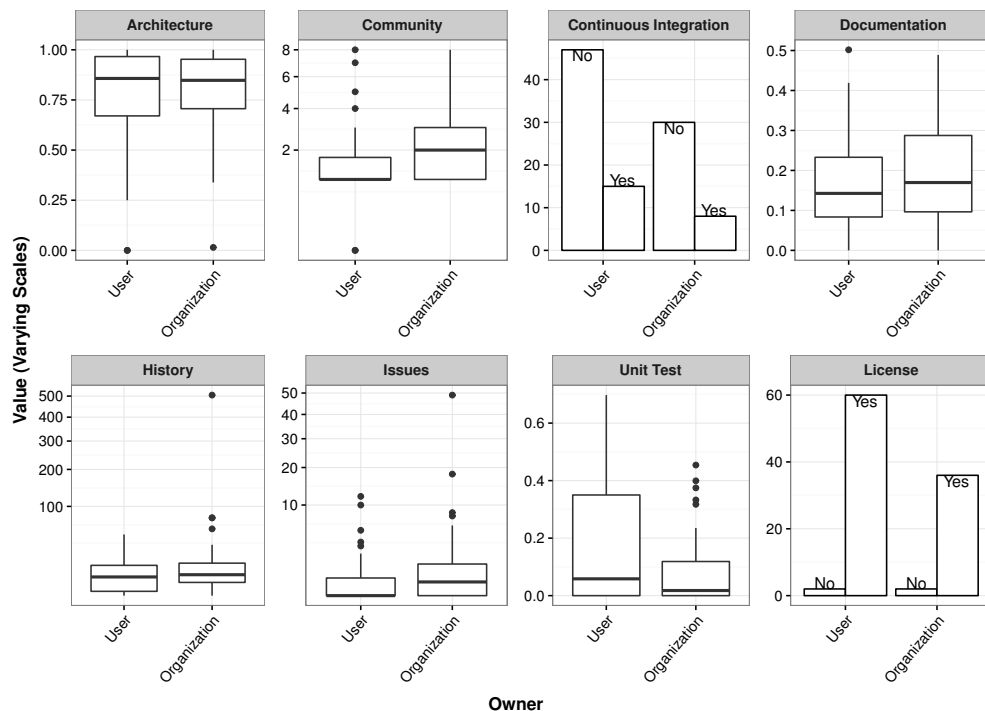
In addition to the dimensions, the thresholds and weights used in the score-based classifier are subjective as well. Here again, we consider the weights we have used to be an acceptable default, however, alternative weighting schemes may be used to mitigate the subjectivity to a certain extent. Some 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 Figure 13 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 1,994,977 GitHub repositories, the number of repositories classified as containing engineered software projects increased by 22% and 2% for score-based classifiers trained using the organization and utility data sets, respectively. We, however, chose to retain our weighting scheme as an acceptable default.

### 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

Distribution of Dimensions of Repositories Owned by Users and Organizations



**Figure 12.** Comparing the distribution of dimensions of repositories known to contain engineered software project owned by organizations and users

929 bias in the repositories selected. For example, if the goal of a study is to analyze the proliferation of unit  
 930 testing in the real-world, using *reaper* will inherently bias the repositories selected toward unit  
 931 testing frameworks that are currently recognized by *reaper*. However, the researcher may configure *reaper*  
 932 such that the unit testing dimension is ignored in the computation of the score thereby mitigating the skew  
 933 in the repositories selected.

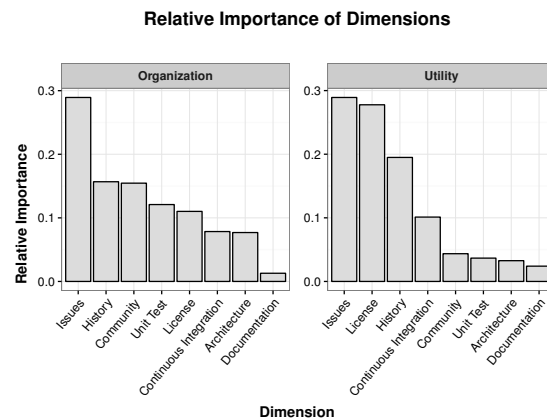
### 934 10.3 Extensibility

935 In addition to *reaper*, the publicly-accessible data set available for download from the project website  
 936 (<https://reporeapers.github.io/>) containing the raw values of the eight dimensions for  
 937 2,247,526 repositories is an important contribution of our work. A compute cluster with close to 200  
 938 nodes took over a month to analyze these repositories. As an alternative to modifying *reaper* and  
 939 rerunning the analysis, researchers can develop a simple script to directly use the raw values and their  
 940 own thresholds and weights to compute customized scores for the repositories.

## 941 11 CONCLUSION

942 The goal of our work was to understand the elements that constitute an engineered software project.  
 943 We proposed eight such elements, called dimensions. The dimensions are: architecture, community,  
 944 continuous integration, documentation, history, issues, license, and unit testing. We developed an open-  
 945 source tool called *reaper* that was used to measure the dimensions of 2,247,526 GitHub repositories  
 946 spanning seven popular programming languages. Two sets of repositories, each corresponding to a  
 947 different definition of an engineered software project, were composed and a score-based and random  
 948 forest classifiers were trained.

949 The classifiers were then used to identify all repositories in the sample of 1,994,977 GitHub repositories  
 950 that were similar to the ones that conform to the definitions of the engineered software project. Our best  
 951 performing random forest model predicted 20.19% of 1,994,977 GitHub repositories contain engineered  
 952 software project.



**Figure 13.** Relative importance of dimensions in the organization and utility data sets

## ACKNOWLEDGMENTS

We thank Nimish Parikh for his contributions during the initial stages of the research. We thank our peers for providing us with 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. and Sutton, C. (2013). Mining Source Code Repositories at Massive Scale using Language Modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 207–216. IEEE.
- Belady, L. A. and Lehman, M. M. (1976). A model of large program development. *IBM Systems journal*, 15(3):225–252.
- Bird, C., Nagappan, N., Murphy, B., Gall, H., and 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*, pages 4–14. ACM.
- Bissyandé, T., Thung, F., Lo, D., Jiang, L., and Reveillere, L. (2013). Orion: A Software Project Search Engine with Integrated Diverse Software Artifacts. In *Engineering of Complex Computer Systems (ICECCS), 2013 18th International Conference on*, pages 242–245.
- Bissyandé, T. F., Lo, D., Jiang, L., Réveillère, L., Klein, J., and Traon, Y. L. (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)*, pages 188–197.
- Bissyandé, T. F., Thung, F., Lo, D., Jiang, L., and Réveillere, L. (2013). Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects. In *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*, pages 303–312. IEEE.
- Bowman, I. T., Holt, R. C., and Brewster, N. V. (1999). Linux as a case study: Its extracted software architecture. In *Proceedings of the 21st International Conference on Software Engineering, ICSE ’99*, pages 555–563, New York, NY, USA. ACM.
- Breiman, L. (2001). Random Forests. *Machine Learning*, 45(1):5–32.
- CA Technologies (2016). Waffle.io - Work Better on GitHub Issues. <https://waffle.io/>. Accessed: 2016-03-11.
- Carlo Zapponi (2016). GitHub - Programming Languages and GitHub. <http://github.info>. Accessed: 2016-03-11.
- Center, S. F. L. (2012). Managing copyright information within a free software project.
- 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, S. C. B., Anquetil, N., and de Oliveira, K. M. (2005). A study of the documentation essential

- 990 to software maintenance. In *Proceedings of the 23rd Annual International Conference on Design of*  
991 *Communication: Documenting & Designing for Pervasive Information*, SIGDOC '05, pages 68–75,  
992 New York, NY, USA. ACM.
- 993 Dyer, R., Nguyen, H. A., Rajan, H., and Nguyen, T. N. (2013). Boa: A language and infrastructure  
994 for analyzing ultra-large-scale software repositories. In *35th International Conference on Software*  
995 *Engineering*, ICSE'13, pages 422–431.
- 996 Eick, S. G., Graves, T. L., Karr, A. F., Marron, J. S., and Mockus, A. (2001). Does code decay? assessing  
997 the evidence from change management data. *Software Engineering, IEEE Transactions on*, 27(1):1–12.
- 998 Georg Brandl and Pygments Contributors (2016). Pygments: Python Syntax Highlighter. <http://pygments.org/>. Accessed: 2016-03-11.
- 1000 GHTorrent (2016). GHTorrent Hall of Fame. <http://ghtorrent.org/halloffame.html>.  
1001 Accessed: 2016-03-11.
- 1002 GHTorrent (2016). GHTorrent: The relational DB schema. <http://ghtorrent.org/relational.html>. Accessed: 2016-03-11.
- 1003 GitHub, Inc. (2016a). GitHub API v3 — GitHub Developer Guide. <https://developer.github.com/v3/>. Accessed: 2016-03-11.
- 1004 GitHub, Inc. (2016b). GitHub Archive. <https://www.githubarchive.org/>. Accessed: 2016-  
1005 06-19.
- 1006 GitHub Inc. (2016). No License - Choose a License. [http://choosealicense.com/no-](http://choosealicense.com/no-license/)  
1007 [license/](http://choosealicense.com/no-license/). Accessed: 2016-03-11.
- 1008 Gousios, G. (2013). The GHTorrent Dataset and Tool Suite. In *Proceedings of the 10th Working*  
1009 *Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA. IEEE  
1010 Press.
- 1011 Guzman, E., Azócar, D., and Li, Y. (2014). Sentiment analysis of commit comments in github: an  
1012 empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*,  
1013 pages 352–355. ACM.
- 1014 HuBoard Inc. (2016). HuBoard - GitHub issues made awesome. <https://huboard.com/>. Accessed:  
1015 2016-03-11.
- 1016 Iowa State University (2016). Publications Related to Boa - Boa - Iowa State University. [http://](http://boa.cs.iastate.edu/papers/)  
1017 [boa.cs.iastate.edu/papers/](http://boa.cs.iastate.edu/papers/). Accessed: 2016-03-11.
- 1018 Jarczyk, O., Gruszka, B., Jaroszewicz, S., Bukowski, L., and Wierzbicki, A. (2014). Github projects.  
1019 quality analysis of open-source software. In *Social Informatics*, pages 80–94. Springer.
- 1020 Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., and Damian, D. (2014). The  
1021 Promises and Perils of Mining GitHub. In *Proceedings of the 11th Working Conference on Mining*  
1022 *Software Repositories*, MSR 2014, pages 92–101, New York, NY, USA. ACM.
- 1023 Kochhar, P. S., Bissyandé, T. F., Lo, D., and Jiang, L. (2013). Adoption of Software Testing in Open  
1024 Source Projects—A Preliminary Study on 50,000 Projects. In *2013 17th European Conference on*  
1025 *Software Maintenance and Reengineering*, pages 353–356.
- 1026 Kofink, A. (2015). Contributions of the Under-appreciated: Gender Bias in an Open-source Ecology. In  
1027 *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Program-*  
1028 *ming, Languages and Applications: Software for Humanity*, SPLASH Companion 2015, pages 83–84,  
1029 New York, NY, USA. ACM.
- 1030 Laplante, P. (2007). *What Every Engineer Should Know about Software Engineering*. CRC Press.
- 1031 Lester, A. (2014). Beyond grep: ack 2.14, a source code search tool for programmers. [http://](http://beyondgrep.com/)  
1032 [beyondgrep.com/](http://beyondgrep.com/). Accessed: 2016-03-11, Version: 2.14.
- 1033 Maier, M., Emery, D., and Hilliard, R. (2001). Software architecture: introducing ieee standard 1471.  
1034 *Computer*, 34(4):107–109.
- 1035 Mockus, A., Fielding, R. T., and Herbsleb, J. (2000). A case study of open source software development:  
1036 the Apache server. In *Proceedings of the 22nd international conference on Software engineering*, pages  
1037 263–272. Acm.
- 1038 Munaiah, N., Gonzalez, K. C., and Meneely, A. (2016a). GitHub - andymeneely/attack-surface-metrics:  
1039 Scripts for collecting metrics of the attack surface. [https://github.com/andymeneely/](https://github.com/andymeneely/attack-surface-metrics)  
1040 [attack-surface-metrics](https://github.com/andymeneely/attack-surface-metrics). Accessed: 2016-10-26.
- 1041 Munaiah, N., Kroh, S., Cabrey, C., and Nagappan, M. (2016b). Home of the reporeapers. [https://](https://reporeapers.github.io)  
1042 [reporeapers.github.io](https://reporeapers.github.io). Accessed: 2016-03-1.



- 1045 Munaiah, N., Kroh, S., Cabrey, C., and Parikh, N. (2016c). reaper - reference implementation.  
1046 <https://github.com/RepoReapers/reaper>. Accessed: 2016-03-11.
- 1047 Nagappan, N. (2007). Potential of open source systems as project repositories for empirical studies  
1048 working group results. In Basili, V., Rombach, D., Schneider, K., Kitchenham, B., Pfahl, D., and Selby,  
1049 R., editors, *Empirical Software Engineering Issues. Critical Assessment and Future Directions*, volume  
1050 4336 of *Lecture Notes in Computer Science*, pages 103–107. Springer Berlin Heidelberg.
- 1051 Nagappan, N., Williams, L., Osborne, J., Vouk, M., and Abrahamsson, P. (2005). Providing test quality  
1052 feedback using static source code and automatic test suite metrics. In *Software Reliability Engineering,*  
1053 *2005. ISSRE 2005. 16th IEEE International Symposium on*, pages 10–pp. IEEE.
- 1054 Ray, B., Posnett, D., Filkov, V., and Devanbu, P. (2014). A Large Scale Study of Programming Languages  
1055 and Code Quality in Github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on*  
1056 *Foundations of Software Engineering*, pages 155–165. ACM.
- 1057 Ross, S. M. (2003). Peirce’s criterion for the elimination of suspect experimental data. *Journal of*  
1058 *Engineering Technology*, 20(2):38–41.
- 1059 Syer, M. D., Nagappan, M., Hassan, A. E., and Adams, B. (2013). Revisiting prior empirical findings for  
1060 mobile apps: an empirical case study on the 15 most popular open-source Android apps. In *CASCON*,  
1061 pages 283–297.
- 1062 Tung, Y.-H., Chuang, C.-J., and Shan, H.-L. (2014). A framework of code reuse in open source software.  
1063 In *Network Operations and Management Symposium (APNOMS), 2014 16th Asia-Pacific*, pages 1–6.  
1064 IEEE.
- 1065 Vasilescu, B., van Schuylenburg, S., Wulms, J., Serebrenik, A., and van den Brand, M. G. J. (2014).  
1066 Continuous Integration in a Social-Coding World: Empirical Evidence from GitHub. In *2014 IEEE*  
1067 *International Conference on Software Maintenance and Evolution*, pages 401–405.
- 1068 Vendome, C. (2015). A Large Scale Study of License Usage on GitHub. In *2015 IEEE/ACM 37th IEEE*  
1069 *International Conference on Software Engineering*, volume 2, pages 772–774.
- 1070 Whitehead, J., Mistrík, I., Grundy, J., and van der Hoek, A. (2010). Collaborative software engineering:  
1071 concepts and techniques. In *Collaborative Software Engineering*, pages 1–30. Springer.
- 1072 Zaidman, A., Van Rompaey, B., Demeyer, S., and Van Deursen, A. (2008). Mining software repositories  
1073 to study co-evolution of production & test code. In *Software Testing, Verification, and Validation, 2008*  
1074 *1st International Conference on*, pages 220–229. IEEE.
- 1075 Zazworka, N., Shaw, M. A., Shull, F., and Seaman, C. (2011). Investigating the Impact of Design Debt on  
1076 Software Quality. In *Proceedings of the 2nd Workshop on Managing Technical Debt, MTD ’11*, pages  
1077 17–23, New York, NY, USA. ACM.
- 1078 Zenhub (2016). ZenHub - Project Management for Agile Teams on GitHub. [https://www.zenhub.](https://www.zenhub.io/)  
1079 [io/](https://www.zenhub.io/). Accessed: 2016-03-11.
- 1080 Zhu, H., Hall, P. A., and May, J. H. (1997). Software unit test coverage and adequacy. *Acm computing*  
1081 *surveys (csur)*, 29(4):366–427.