Identifying Unmaintained Projects in GitHub

Jailton Coelho
Federal University of Minas Gerais, Brazil
jailtoncoelho@dcc.ufmg.br

Marco Tulio Valente
Federal University of Minas Gerais, Brazil
mtov@dcc.ufmg.br

Luciana L. Silva
Federal Institute of Minas Gerais, Brazil
luciana.lourdes.silva@ifmg.edu.br

Emad Shihab
Concordia University,Canada
eshihab@encs.concordia.ca

ABSTRACT

Background: Open source software has an increasing importance in modern software development. However, there is also a growing concern on the sustainability of such projects, which are usually managed by a small number of developers, frequently working as volunteers. *Aims:* In this paper, we propose an approach to identify GitHub projects that are not actively maintained. Our goal is to alert users about the risks of using these projects and possibly motivate other developers to assume the maintenance of the projects. *Method:* We train machine learning models to identify unmaintained or sparsely maintained projects, based on a set of features about project activity (commits, forks, issues, etc). We empirically validate the model with the best performance with the principal developers of 129 GitHub projects. *Results:* The proposed machine learning approach has a precision of 80%, based on the feedback of real open source developers; and a recall of 96%. We also show that our approach can be used to assess the risks of projects becoming unmaintained. *Conclusions:* The model proposed in this paper can be used by open source users and developers to identify GitHub projects that are not actively maintained anymore.

CCS CONCEPTS

• **Software and its engineering** → *Maintaining software*;

KEYWORDS

Unmaintained Projects, Open Source Software, GitHub

ACM Reference Format:
Jailton Coelho, Marco Tulio Valente, Luciana L. Silva, and Emad Shihab. 2018. Identifying Unmaintained Projects in GitHub. In *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (ESEM '18), October 11–12, 2018, Oulu, Finland*. ACM, New York, NY, USA, 10 pages. *https://doi.org/10.1145/3239235.3240501*

1 INTRODUCTION

Open source projects have an increasing relevance in modern software development [11]. For example, many critical software systems are currently available under open source licenses, including operating systems, compilers, databases, and web servers. Similarly, it is common nowadays to depend on open source libraries and frameworks when building and evolving proprietary software. For example, in a recent survey—conducted by Black Duck Software—78% of the over 1,300 companies surveyed acknowledge the use of open source in their daily development.[1] Concretely, Instagram—the popular photo-sharing social network—is currently implemented using more than 20 open source libraries.[2] Furthermore, the emergence of world-wide code sharing platforms—GitHub is the most well-known example—is contributing to transform open source development in a competitive market. Indeed, in a recent survey with open source maintainers we found that the most common reason for the failure of open source projects is the appearance of a stronger competitor in GitHub [8].

However, GitHub does not include clear data about project status, in terms of maintenance activity. Users can access historical data about commits or global project metrics, like number of stars, forks, and watchers. However, based on the values of theses metrics, they should judge themselves whether a project is being actively maintained (and therefore if it is worth to use it or not). Therefore, in this paper we propose and evaluate a machine learning approach to identify unmaintained (or sparsely maintained) projects in GitHub. Our goal is to provide a simple and effective mechanism to alert users about the risks of depending on a GitHub project. This information can also contribute to attract new maintainers to a project. For example, users of libraries facing the risks of discontinuation can be motivated to assume their maintenance.

Previous work in this area relies on the last commit activity to classify projects as unmaintained or in similar status. For example, Khondhu et al. use an one-year inactivity threshold to classify *dormant* projects on SourceForge [15]. The same threshold is used in works by Mens et al. [24], Izquierdo et al. [14], and in our previous work about the motivations for the failure of open source projects [8]. However, in this paper, we do not use this definition when investigating unmaintained projects due to three reasons. First, because defining a threshold to characterize unmaintained projects is not trivial. For example, in the mentioned works, this decision is arbitrary and it is not empirically validated. Second, our intention is to detect unmaintained projects as soon as possible; preferably, without having to wait for one year of inactivity. Third, our definition of unmaintained projects does not assume a complete absence of commits during a given period; instead, a

ESEM '18, October 11–12, 2018, Oulu, Finland

ACM ISBN 978-1-4503-5823-1/18/10…$15.00
https://doi.org/10.1145/3239235.3240501

[1] *https://www.slideshare.net/blackducksoftware/2015-future-of-open-source-survey-results*
[2] *https://www.instagram.com/about/legal/libraries/*

project is considered unmaintained even when sporadic and few commits happen in a given time interval. Stated otherwise, by our definition, unmaintained projects do not necessarily need to be dead, deprecated or archived.

In this paper, we first train ten machine learning models to identify unmaintained projects, using as features standard metrics provided by GitHub about a project's maintenance activity, e.g., number of commits, forks, issues, and pull requests. Then, we select the model with the best performance and validate it by means of a survey with the owners of projects classified as *unmaintained* and also with a set of deprecated GitHub projects. Particularly, we ask three research questions about properties of this model:

RQ1: What is the precision according to GitHub developers? The intention is to check precision in the field, according to the feedback provided by the principal developers of popular GitHub projects.

RQ2: What is the recall when identifying unmaintained projects? Recall is more difficult to compute in the field, because it requires the identification of all unmaintained projects in GitHub. To circumvent this problem, we compute recall considering only projects that declare in their README[3] they are not under maintenance.

RQ3: How early does the model identify unmaintained projects? As mentioned, the proposed model does not depend on an inactivity interval to classify a project as unmaintained. Therefore, in this question, we investigate whether this ability is effective in the field, when identifying the maintenance status of real GitHub projects.

Our contributions are twofold: (1) we propose a machine learning approach to identify unmaintained (or sporadically maintained) projects in GitHub, which achieved a precision of 80% and a recall of 96% when validated with real open source developers and projects; (2) we propose a metric to reveal the maintenance activity level of GitHub projects.

This paper is organized as follows. In Section 2, we present and evaluate a machine learning model to identify unmaintained projects. Section 3 validates this model with GitHub developers and projects that are documented as deprecated. Section 4 defines and discusses the Level of Maintenance Activity (LMA) metric. Section 5 lists threats to validity and Section 6 discusses related work. Section 7 concludes the paper and outlines further work.

2 MACHINE LEARNING MODEL

In this section, we describe our machine learning approach to identify projects that are no longer under maintenance.

2.1 Experimental Design

Dataset. We start with a dataset containing the top-10,000 most starred projects on GitHub (in November, 2017). Stars—GitHub's equivalent for *likes* in other social networks—is a common proxy for the popularity of GitHub projects [4]. Then, we follow three strategies in order, to discard projects from this initial selection. First, we remove 2,810 repositories that have less than two years from the first to the last commit (because we need historical data to compute the features used by the prediction models). Second, we remove 331 projects with null size, measured in lines of code (typically, these projects are implemented in non-programming languages, like CSS, HTML, etc). Finally, we remove 74 non-software projects, which are identified by searching for the following topics: *books* and *awesome-lists*.[4] We end up with a list of 6,785 projects.

Next, we define two subsets of systems: *active* and *unmaintained*. The active (or under maintenance) group is composed by 754 projects that have at least one release in the last month, including well-known projects, like FACEBOOK/REACT, D3/D3, and NODEJS/NODE. Thus, we assume that projects with recent releases are active (under maintenance). By contrast, the unmaintained group is composed by 248 projects, including 104 projects that were explicitly declared by their principal developers as unmaintained in our previous work [8] and 144 *archived* projects. Archiving is a recent feature provided by GitHub that allows developers to explicitly move their projects to a read-only state. In this state, users cannot create issues, pull requests, or comments, but can still fork or star the projects.

Features. Our hypothesis is that a machine learning classifier can identify unmaintained projects by considering features about (1) projects, including number of forks, issues, pull requests, and commits; (2) contributors, including number of new and distinct contributors (the rationale is that maintenance activity might increase by attracting new developers); (3) project owners, including number of projects he/she owns and total number of commits in GitHub (the rationale is that maintenance might be affected when project owners have many projects on GitHub). In total, we consider 13 features, as described in Table 1. The feature values are collected using GitHub's official API. However, they do not refer to the whole history of a project, but only to the last n months, counting from the last commit; moreover, we collect each feature in intervals of m months. The goal is to derive temporal series of feature values, which can be used by a machine learning algorithm to infer trends in the project evolution, e.g., an increasing number of opened issues or a decreasing number of commits. Figure 1 illustrates the feature collection process assuming that n is 24 months and that m is 3 months. In this case, for each feature, we collect 8 data points, i.e., feature values.

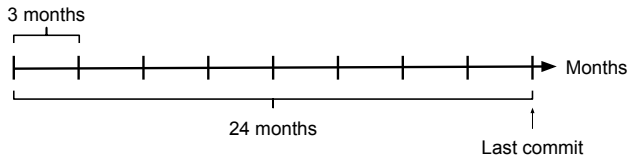


Figure 1: Feature collection during 24 months in 3-month intervals.

We experiment with different combinations of n and m; each one is called a scenario, in this paper. Table 2 describes the total number of data points extracted for each scenario. This number ranges from 13 data points (scenario with features extracted in a single interval of 6 months) to 104 data points (scenario with features extracted in intervals of 3 months during 24 months, as in Figure 1).

[3]READMEs are the first file a visitor is presented to when visiting a GitHub repository.

[4]GitHub topics allow tagging a repository with keywords.

Table 1: Features used to identify unmaintained projects.

Dimension	Feature	Description
Project	Forks	Number of forks created by developers
	Open issues	Number of issues opened by developers
	Closed issues	Number of issues closed by developers
	Open pull requests	Number of pull requests opened by the project developers
	Closed pull requests	Number of pull requests closed by the project developers
	Merged pull requests	Number of pull requests merged by the project developers
	Commits	Number of commits performed by developers
	Max days without commits	Maximum number of consecutive days without commits
	Max contributions by developer	Number of commits of the developer with the highest number of commits
Contributor	New contributors	Number of contributors who made their first commit in the considered period
	Distinct contributors	Number of distinct contributors that committed in the considered period
Owner	Projects created by the owner	Number of projects created by a given owner
	Number of commits of the owner	Number of commits performed by a given owner

Table 2: Scenarios used to collect features and train the machine learning models (length and intervals are in months; data points is the total number of data points collected for each scenario).

Scenario	1	2	3	4	5	6	7	8	9	10
Length	6	6	12	12	12	18	18	24	24	24
Intervals	3	6	3	6	12	3	6	3	6	12
Data points	26	13	52	26	13	78	39	104	52	26

Correlation Analysis. As usual in machine learning experiments, we remove correlated features, following the process described by Bao et al. [2]. To this purpose, we use a clustering analysis—as implemented in a R package named *Hmisc*[5]—to derive a hierarchical clustering of correlations among data points (extracted for the features in each scenario). For sub-hierarchies with correlations larger than 0.7, we select only one data point for inclusion in the respective machine learning model, as common in other works [2, 36]. For example, Figure 2 shows the final hierarchical clustering for the scenario with 24 months, considering a 3-month interval (scenario 8). The analysis in this scenario checks correlations among 104 data points (13 features × 8 data points per feature). As a result, 78 data points are removed due to correlations with other points and therefore do not appear in the dendogram presented in Figure 2. Finally, Table 3 shows the total number and percentage of data points removed in each scenario, after correlation analysis. As we can see, the percentage of removed points is relevant, ranging from 43% (scenario 7) to 75% (scenario 8).

Machine Learning Classifier. We use the data points extracted in each scenario to train and test models for predicting whether a project is unmaintained. In other words, we train and test ten machine learning models, one for each scenario. After that, we select the best model/scenario to continue with the paper. Particularly, we use the Random Forest algorithm [5] to train the models because it has several advantages, such as robustness to noise and outliers [36]. In addition, it is adopted in many other software engineering works [13, 25, 28, 30]. We compare the result of Random Forest with two baselines: baseline #1 (all projects are predicted as unmaintained) and baseline #2 (random predictions). We use the Random Forest implementation provided by *randomForest*'s R package[6] and 5-fold stratified cross validation to evaluate the models effectiveness. In 5-fold cross validation, we randomly divide the dataset into five folds, where four folds are used to train

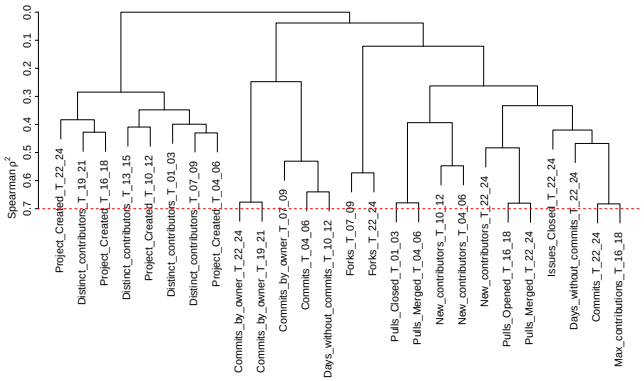


Figure 2: Correlation analysis for the 104 data points collected for the features in scenario 8 (24 months, 3-month interval). 78 data points (75%) are removed in this case, due to correlations with other data points, and therefore they do no appear in this final clustering.

Table 3: Total number and percentage of data points removed in each scenario, after correlation analysis.

Scenario	1	2	3	4	5	6	7	8	9	10
#	17	6	38	18	7	56	17	78	34	19
%	65	46	73	69	54	72	43	75	65	73

[5]http://cran.r-project.org/web/packages/Hmisc/index.html

[6]https://cran.r-project.org/web/packages/randomForest/

Table 4: Prediction results (mean of 100 iterations, using 5-cross validation); best results are in bold.

Metrics	0.5 Year		1 Year			1.5 Years		2 Years		
	3 months	6 months	3 months	6 months	12 months	3 months	6 months	3 months	6 months	12 months
Accuracy	0.90	0.91	0.91	0.90	0.89	0.91	0.90	**0.92**	0.91	0.90
Precision	0.83	0.87	0.87	0.84	0.82	0.86	0.83	**0.86**	0.85	0.83
Recall	0.78	0.74	0.77	0.75	0.72	0.78	0.76	**0.81**	0.79	0.73
F-measure	0.80	0.79	0.81	0.79	0.77	0.82	0.79	**0.83**	0.82	0.78
Kappa	0.74	0.74	0.76	0.73	0.70	0.76	0.73	**0.78**	0.76	0.71
AUC	0.86	0.85	0.86	0.85	0.83	0.87	0.85	**0.88**	0.87	0.84

a classifier and the remaining fold is used to test its performance. Specifically, stratified cross validation is a variant, where each fold has approximately the same proportion of each class [5]. We perform 100 rounds of experiments and report average results.

Evaluation Metrics. When evaluating the projects in the test fold, each project has four possible outcomes: (1) it is truly classified as unmaintained (True Positive); (2) it is classified as unmaintained but it is actually an active project (False Positive); (3) it is classified as an active project but it is actually an unmaintained one (False Negative); and (4) it is truly classified as an active project (True Negative). Considering these possible outcomes, we use six metrics to evaluate the performance of a classifier: precision, recall, F-measure, accuracy, AUC (Area Under Curve), and Kappa, which are commonly adopted in machine learning studies [10, 17, 21, 35, 36]. Precision and recall measure the correctness and completeness of the classifier, respectively. F-measure is the harmonic mean of precision and recall. Accuracy measures how many projects are classified correctly over the total number of projects. AUC refers to the area under the Receiver Operating Characteristic (ROC) curve. Finally, kappa evaluates the relationship between the observed accuracy and the expected one [31], which is particularly relevant in imbalanced datasets, as the dataset used to train the machine learning models (754 active projects *vs* 248 unmaintained ones).

2.2 Experimental Results

Table 4 shows the results for each scenario. As we can see, Random Forest has the best results (in bold) when the features are collected during 2 years, in intervals of 3 months. In this scenario, precision is 86% and recall is 81%, leading to a F-measure of 83%. Kappa is 0.78—usually, kappa values greater than 0.60 are considered quite representative [18]. Finally, AUC is 0.88, which is an excellent result in the Software Engineering domain [21, 34, 36]. Table 5 compares the results of the best scenario/model with baseline #1 (all projects are predicted as unmaintained) and baseline #2 (random predictions). Despite the baseline under comparison, there are major differences in all evaluation metrics. For example, F-measure is 0.37 (baseline #1) and 0.30 (baseline #2), against 0.83 (proposed model).

Random Forest produces a measure of the importance of the predictor features. Table 6 shows the top-5 most important features by Mean Decrease Accuracy (MDA), for the best model. Essentially, MDA measures the increase in prediction error (or reduction in prediction accuracy) after randomly shifting the feature values [6, 22]. As we can see, the most important feature is the number of commits in the last time interval (i.e., the interval delimited by months 22-24, $T_{22,24}$), followed by the maximal number of days without commits in the same interval and in the interval $T_{10,2}$. As also presented in Table 6, the first four features are related to commits; the first feature non-related with commits is the number of issues closed in the first time interval ($T_{1,3}$).

Table 5: Comparison of the proposed machine learning model with baseline #1 (all projects are predicted as unmaintained) and baseline #2 (random predictions).

Metrics	Model	Baseline #1	Baseline #2
Accuracy	0.92	0.22	0.49
Precision	0.86	0.22	0.22
Recall	0.81	1.00	0.48
F-measure	0.83	0.37	0.30
Kappa	0.78	0.00	0.01
AUC	0.88	0.50	0.49

Table 6: Top-5 most relevant features, by Mean Decrease Accuracy (MDA).

Feature	Period	MDA
Commits	$T_{22,24}$	38.5
Max days without commits	$T_{22,24}$	28.6
Max days without commits	$T_{10,12}$	21.9
Max contributions by developer	$T_{16,18}$	21.1
Closed issues	$T_{1,3}$	18.0

3 EMPIRICAL VALIDATION

In this section, we *validate* the proposed machine learning model by means of a survey with the owners of projects classified as *unmaintained* and also with a set of deprecated GitHub projects. Overall, our goal is to strengthen the confidence on the practical value of the model proposed in this work. Particularly, we provide answers to three research questions about this model:

RQ1: What is the precision according to GitHub developers?

RQ2: What is the recall when identifying deprecated projects?

RQ3: How early does the model identify unmaintained projects?

3.1 Methodology

RQ1: To answer RQ1, we conduct a survey with GitHub developers. To select the participants, we first apply the proposed machine learning model in all projects from our dataset that were not used in the model's construction, totaling 5,783 projects (6,785 − 1,002 projects). Then, we select 2,856 projects classified as unmaintained by the proposed model. From this sample, we remove 264 projects whose developers were recently contacted in our previous surveys [8, 9]. We make this decision to not bother again these developers, with new e-mails and questions. Finally, we remove 2,270 projects whose owners do not have a public e-mail address on GitHub. As a result, we obtain a list of 323 survey participants (2,856 − 2,270 − 264). However, before e-mailing these participants, the first author inspected the main page of each project on GitHub, to check whether it includes mentions to the project status, in terms of maintenance. We found 21 projects whose documentation states they are no longer maintained, by means of messages like this one:

This project is deprecated. It will not receive any future updates or bug fixes. If you are using it, please migrate to another solution.

Therefore, we do not send mails to the project owners, in such cases; and automatically consider these 21 projects as *unmaintained*.

Survey Period: The survey was performed in the first two weeks of May, 2018. It is important to highlight that the machine learning model was constructed using data collected on November, 2017. Therefore, the *unmaintained* predictions evaluated in the survey refer to this date. We wait five months to ask the developers about the status of their projects because it usually takes some time until developers actually accept the unmaintained condition of their projects. In other words, this section is based on predictions performed and available on November, 2017. However, these predictions are validated five months later, on May, 2018.

Survey Pilot and Questions: Initially, we perform a pilot survey with 75 projects (≈ 25%), randomly selected among the remaining 302 projects (323 − 21 projects). We e-mail the principal developers of these projects with a single open question, asking them to confirm (or not) if their projects are unmaintained. We received 23 answers, which corresponds to a response ratio of 30.6%. Then, two authors of this paper analyzed together the received answers to derive a list of recurrent themes. As a result, the following three common maintainability status were identified:[7]

(1) **My project is under maintenance and new features are under implementation (6 answers):** As an example, we can mention the following answer:

[Project-Name] is still maintained. I maintain the infrastructure side of the project myself (e.g., make sure it's compatible with the latest Ruby version, coordinate PRs and issues, mailing list, etc.) while community provides features that are still missing. One such big feature is being developed as we speak and will be the highlight of the next release. (P57)

(2) **My project is finished and I only plan to fix important bugs (13 answers):** As an example, we mention the following answers:

It's just complete, at least for now. I still fix bugs on the rare occasion they are reported. (P10)

I view it as basically "done". I don't think it needs any new features for the foreseeable future, and I don't see any changes as urgent unless someone discovers a major security vulnerability or something. I will probably continue to make changes to it a couple times per year, but mostly bugfixes. (P68)

(3) **My project is deprecated and I do not plan to implement new features or fix bugs (4 answers):** As an example, we can mention the following answer:

The project is unmaintained and I'll archive it. (P74)

After the pilot study, we proceed with the survey, by e-mailing the remaining 227 projects. However, instead of asking an open question—as in the pilot—we provide an objective question to the survey participants, about the maintainability status of their projects. In this objective question, we ask the participants to select one (out of three) status identified in the pilot study, plus an *other* option. This fourth option also includes a text form for the participants detailing their answers, if desired. Essentially, we change to an objective question format to make answering the survey easier, but without limiting the respondents' freedom to provide a different answer from the listed ones. In this final survey, we received 89 answers, representing a response ratio of 39.2%. When considering both phases (pilot and final survey), we sent 302 e-mails, received 112 answers, representing an overall response ratio of 37.1%. After adding the 21 projects that document their maintainability status, this empirical validation is based on 133 projects.

RQ2: To answer this second question, we construct a ground truth with projects that are no longer being maintained. First, we build a script to download the README (the main page of GitHub's projects) and search for a list of sentences that are commonly used to declare the unmaintained state of GitHub projects. This list is reused from our previous work [8], where we study the motivations for the failure of open source projects. It includes 32 sentences; in Table 7, we show a partial list, with 15 sentences.

Table 7: Sentences documenting unmaintained projects.

no longer under development, no longer supported or updated,
deprecation notice, dead project, deprecated, unmaintained,
no longer being actively maintained, not maintained anymore,
not under active development, no longer supported,
is not supported, is not more supported, no longer supported,
no new features should be expected, isn't maintained anymore

We searched (in May, 2018) for these sentences in the README of 5,783 projects, which represent all 6,785 projects selected for this work minus 1,002 projects used in Section 2. In 451 READMEs (7.8%) we found the mentioned sentences. Then, the first author of this paper carefully inspected each README, to confirm the sentences indeed refer to the project's status, in terms of maintenance. In the case of 112 projects (≈ 25%), he confirmed this fact. Therefore, these projects are considered as a ground truth for this investigation. Usually, the unconfirmed cases refer to the deprecation of specific elements, e.g., methods or classes.

[7] Project names are omitted, to preserve the respondent's anonymity; survey participants are identified by means of labels, in the format Pxx, where *xx* is an integer.

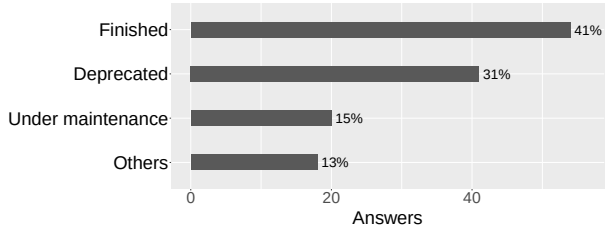


Figure 3: Survey answers about projects' status.

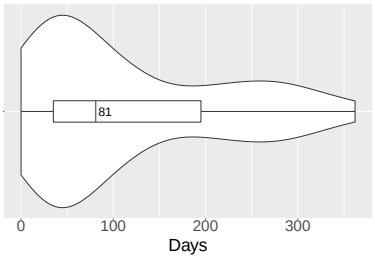


Figure 4: Days since last commit for projects classified as unmaintained (considering the date of November, 2017, when the proposed model was computed).

RQ3: To answer this third research question, we rely on projects whose unmaintained status, as predicted by the proposed model, is confirmed by the participants of the survey conducted in RQ1. Then, we compute the number of days between November 30, 2018 (when the machine learning model proposed in this paper was built) and the last commit of the mentioned projects. For projects where this interval is less than one year, the proposed model is better than the strategy adopted in previous work [8, 14, 15, 24], which requires one year of commit inactivity to identify unmaintained projects.

3.2 Results

RQ1: Precision according to GitHub developers

Before presenting the precision results, Figure 3 shows the survey results, including answers retrieved from the project's documentation, answers received in the pilot and answers received in the final survey. As we can see, the most common status, with 54 answers (41%) refers to *finished* projects, i.e., cases where maintainers see their projects as feature-completed and only plan to resume the maintenance work if relevant bugs are reported.[8] We also received 41 answers (31%) mentioning the projects are deprecated and no further maintenance is planned, including the implementation of new features and bug fixes. Finally, we received 18 answers in the *other* option. In this case, four participants did not describe their answer or provide unclear answers; furthermore, one participant mentioned his project is in a *limbo* state:

The status of [Project-Name] fits into a special category. Some of the tools it's based on are either deprecated or not powerful enough for the goal of the project. This is part of the reason what's keeping the project from being "done". I would call this status ***limbo****. (P24)*

Seven participants answered their projects are *stalled*, as in this answer:

It is under going a rewrite... but has been ***stalled*** *based on my own priorities (P33)*

To compute precision, we consider as *true positive* answers related to the following status: finished (54 answers), deprecated (41 answers), stalled (7 answers), and limbo (1 answer). The remaining answers are interpreted as *false positives*, including answers mentioning that new features are being implemented (20 answers) and the answers associated to the fourth option (*other* option), but without including a description or with an unclear description (4 answers). By following this criteria, we received 103 true positive answers and 26 false positive ones, resulting in a precision of 80%.[9]

> By validating the proposed model with 129 GitHub developers, we achieve a **precision** of 80%, which is a result very close to the one obtained when building the model (86%).

RQ2: Recall considering deprecated projects

The proposed machine learning model classifies 108 (out of 112) projects of the constructed ground truth as unmaintained, which represents a recall of 96%. This value is significantly greater than the one computed when testing the model in Section 2. Probably, this difference is explained by the fact that only projects that are completely unmaintained expose this situation in their READMEs. Therefore, it is easier to detect and identify this condition.

> By validating the proposed model with projects that declare themselves as unmaintained, we achieve a **recall** of 96%.

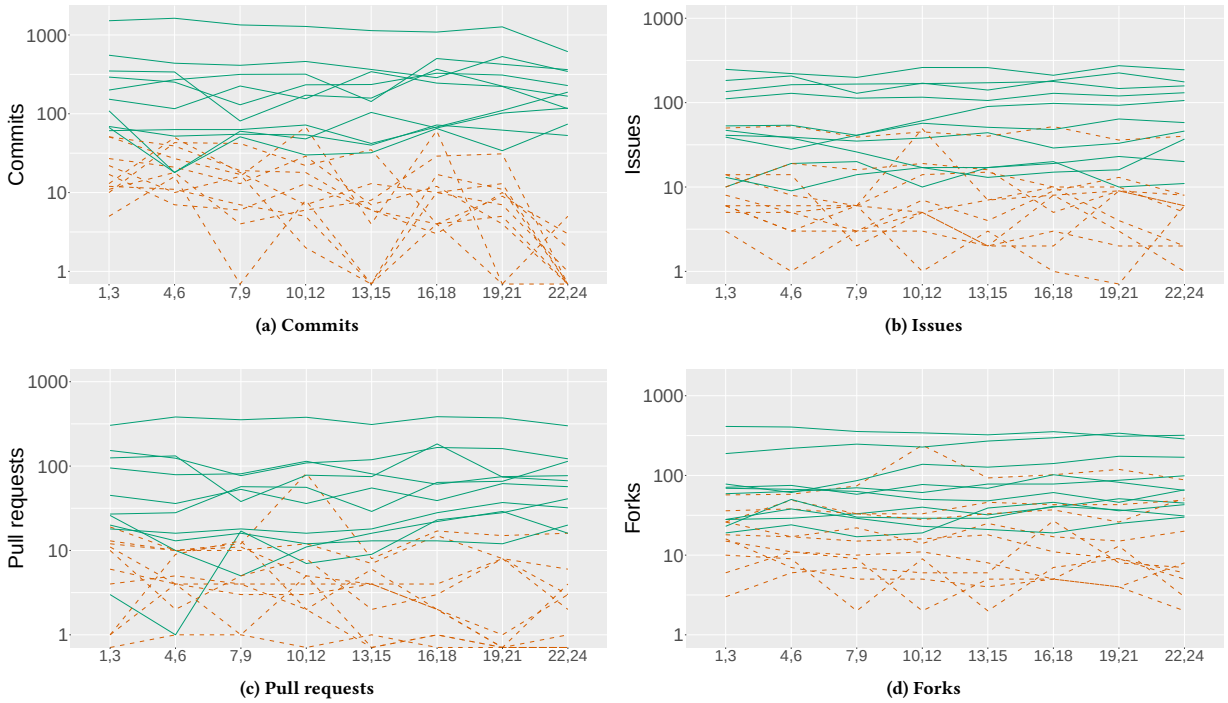
RQ3: How early can we detect unmaintained projects?

77 (out of 103) projects classified as true positives by the surveyed developers have commits after November, 2016. Therefore, these projects would not be classified as unmaintained using the strategy followed in the literature, which requires one year of commit inactivity. In other words, in November, 2017, the proposed model classified 77 projects as unmaintained, despite the existence of recent commits, with less than one year. Figure 4 shows a violin plot with the age of such commits, considering the date of November, 2017. The first, second, and third quartiles are 35, 81, and 195. Interestingly, for two projects the last commit occurred exactly on November, 30, 2018. Despite this fact, the proposed model classified these projects as unmaintained in the same date. If we relied on the standard threshold of one year without commits, these projects would have had to wait one year to receive this classification.

> 75% of the studied projects are classified as unmaintained despite having recent commits, performed in the last year.

[8] In our previous work [8], we also identified finished or completed open source projects. However, we argued these projects do not contradict Lehman's Laws about software evolution [20], because they usually deal with a very stable or controlled environment (whereas Lehman's Laws focus on E-type systems, where *E* stands for evolutionary).

[9] This computation of precision assumes that finished projects are unmaintained. However, we recognize that the risks of using finished projects might be lower than the ones faced when using deprecated or stalled projects.



(a) Commits (b) Issues (c) Pull requests (d) Forks

Figure 6: Number of commits, issues, pull request, and forks over time of ten projects with maximal LMA (green lines) and ten projects with the lowest LMA in our dataset (red, dashed lines). Metrics are collected in intervals of 3 months (x-axis).

4 LEVEL OF MAINTENANCE ACTIVITY

In this section, we define a metric to express the *level of maintenance activity* of GitHub projects, i.e., a metric that reveals how often a project is being maintained. The goal is to alert users about projects that although classified as under maintenance by the proposed model are indeed close to an unmaintained status.

4.1 Definition

The proposed machine learning model—generated by Random Forest —consists of multiple decision trees built randomly. Each tree in the ensemble determines a prediction to a target instance and the most voted class is considered as the final output. One possible prediction type of the Random Forest is the matrix of class probabilities. This matrix represents the proportion of the trees' votes. For example, projects predicted as *under maintenance* have probability p ranging from 0.5 to 1.0. If $p = 0.5$, the project is very similar to an unmaintained project; by contrast, $p = 1.0$ means the project is actively maintained. Using these probabilities, we define the *level of maintenance activity (LMA)* of a GitHub project as follows:

$$LMA = 2 * (p - 0.5) * 100$$

This equation simply converts the probabilities p computed by Random Forest to a range from 0 to 100; LMA equals to 0 means the project is very close to an unmaintained classification (since $p = 0.5$); and LMA equals to 100 denotes a project that is actively maintained (since $p = 1.0$).

4.2 Results

Figure 5 shows the LMA values for each project predicted as *under maintenance* (2,927 projects, after excluding the projects used to train and test the proposed model, in Section 2). The first, second, and third quartiles are 48, 82, and 97, respectively. In other words, most studied projects are under constant maintenance (median 82). Indeed, 171 projects (5.8%) have a maximal LMA, equal to 100. This list includes well-known and popular projects such as TWBS/BOOTSTRAP, METEOR/METEOR, RAILS/RAILS, WEBPACK/WEBPACK, and ELASTIC/ELASTICSEARCH.

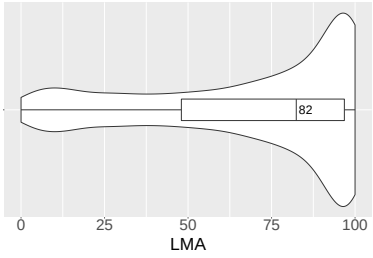
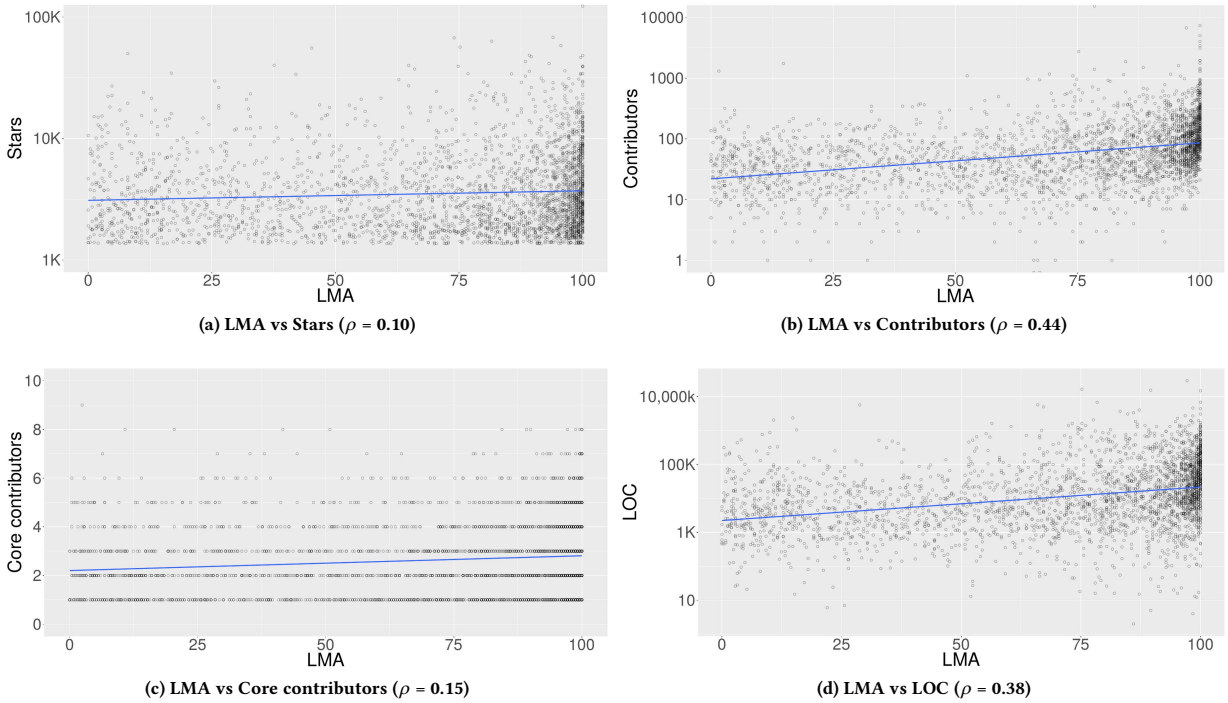


Figure 5: Level of maintenance activity (LMA).

Figure 6 compares a random sample of 10 projects with LMA equals to 100 (actively maintained, therefore) with ten projects with the lowest LMA ($0 \leq LMA \leq 0.4$). These projects are compared using number of commits (Figure 6a), number of issues (Figure 6b), number of pull requests (Figure 6c), and number of forks (Figure 6d), in



(a) LMA vs Stars (ρ = 0.10)

(b) LMA vs Contributors (ρ = 0.44)

(c) LMA vs Core contributors (ρ = 0.15)

(d) LMA vs LOC (ρ = 0.38)

Figure 7: Correlating LMA with (a) stars, (b) contributors, (c) core contributors, and (d) size. Spearman's ρ is also presented.

the last 24 months. Each line represents the project's metric values. The figures reveal major differences among the projects, regarding these metrics. Usually, the projects with high LMA present high values for the four considered metrics (commits, issues, pull requests, and forks), when compared with projects with low LMA. In other words, the figures suggest that LMA plays an aggregator role of maintenance activity over time.

Figure 7 shows scatterplots correlating LMA and number of stars, contributors, core contributors, and size (in LOC) of projects classified as under maintenance. To identify core contributors, we use the most common heuristic described in the literature: core contributors are the ones responsible together for at least 80% of the commits in a project [16, 26, 32]. To measure the size of the projects, in lines of code, we used the tool AlDanial/cloc[10], considering only the programming languages in the TIOBE list.[11] We also compute Spearman's rank correlation test for each figure. The correlation with stars and with core contributors is very weak (ρ = 0.10 and ρ = 0.15, respectively); with size, the correlation is weak (ρ = 0.38); and with contributors, it is moderate (ρ = 0.44); all p-values are less than 0.01. Therefore, it is common to have highly popular projects, by number of stars, presenting both low and high LMA values. For example, one project has 50,034 stars, but LMA = 8. A similar effect happens with size. For example, one project has ≈2 MLOC, but LMA = 10.8. The highest correlation is observed with contributors, i.e., projects with more contributors tend to have higher levels of maintenance activity.

[10]https://github.com/AlDanial/cloc
[11]https://www.tiobe.com/tiobe-index

4.3 Validation with False Negative Projects

In Section 3, we found four projects that although declared by their developers as *unmaintained* are predicted by the proposed machine learning model as *under maintenance*. Therefore, these projects are considered false negatives, when computing recall. Two of such projects has a very low LMA: nicklockwood/iRate (LMA = 2) and gorangajic/react-icons (LMA = 12). Therefore, although predicted as *under maintenance*, this project is similar to projects classified as *unmaintained*, as suggested by its low LMA. A second project has an intermediate LMA value: spotify/HubFramework (LMA = 39.2). Finally, one project Homebrew/homebrew-php has a high LMA value (LMA = 99.2). However, this project was migrated to another repository, when facing continuous maintenance. In other words, in this case, the GitHub repository was deprecated, but not the project; therefore, Homebrew/homebrew-php is a false, false negative (or a true negative).

5 THREATS TO VALIDITY

The threats to validity of this work are described as follows:

External Validity: Our work examines open source projects on GitHub. We recognize that there are popular projects in other platforms (e.g., Bitbucket, SourceForge, and GitLab) or projects that have their own version control installations. Thus, our findings may not generalize to other open source or commercial systems. A second threat relates to the features we have considered. By adding other features, we may improve the prediction of unmaintained projects; however, given our high prediction performance, we feel confident that our features are effective. Also, some of the features

we use may not be available in other projects, however, most of our features are available in most code control repositories/ecosystems. In the future, we intend to investigate additional projects and consider more features.

Internal Validity: The first threat relates to the selection of the survey participants. We surveyed the project owner, in the case of repositories owned by individuals, or the developer with the highest number of commits, in the case of repositories owned by organizations. We believe the developers who replied to our survey are the most relevant given their level of activity in the project. It is also possible that most missing answers are from developers of unmaintained projects. As a second threat, the themes of the survey were defined and organized by the authors of the paper. As with any human activity, the derived themes may be subject to bias and different researchers might reach different observations. However, to mitigate this threat, a first choice of themes was conducted in parallel by the first two authors of this paper. Also, they attended several meetings during a whole week to improve the initial selected themes. A third threat relates to the parameters used to perform our experiment. We set the number of trees to 100 to train our classifier. To attenuate the bias of our results, we run 5-fold cross-validation and use the average performance for 100 rounds. A forth threat is related to how the accuracy of our machine learning approach was evaluated. We relied on developer replies about their projects to evaluate the performance of our machine learning classifier. In some cases, the developer replies (or developers who did not reply) may impact our results. That said, our survey had a response rate of 37.1%, which is very high for a software engineering study, giving us confidence in the reported performance results.

Construct Validity: A first threat relates to the definition of active projects. We consider as active projects those with at least one release in the last month (Section 2). We acknowledge a threat in the definition of the time frame. To mitigate this threat, the first paper's author inspected each selected project to look for deprecated projects (21 projects declare they are no longer being maintained) and we conduct a survey with 112 developers to confirm our findings. A second threat is related to the projects we studied. Our dataset is composed of the most starred projects (and additional filtering). Although the starred projects may not be representative of all open source projects, we did carefully select such projects to ensure that our study is conducted on real (and not toy) projects.

6 RELATED WORK

Machine Learning. Recently, the application of machine learning in software engineering contexts has gained much attention. Several researchers have used machine learning to accurately predict defects (e.g. [28]), improve issue integration (e.g., [1]), enhance software maintenance (e.g., [12]), and examine developer turnover (e.g., [2]). For example, Gousios et al. [12] investigate the use of machine learning to predict whether a pull request will be merged. They extract 12 features organized into three dimensions: pull request, project, and developer. They conduct their study using six algorithms (Logistic Regression, Naive Bayes, Decision Trees, AdaBoost with Decision Trees, and Random Forest). Bao et al. [2] build a model to predict developer turnover, i.e., whether a developer will leave the company after a period of time. They collect several features based on developers monthly report from two companies. The authors evaluate the performance of five classifiers (KNN, Naive Bayes, SVM, Decision Trees, and Random Forest). In both studies, Random Forest outperforms the results of other algorithms. In another study, Martin et al. [23] train a Bayesian model to support app developers on causal impact analysis of releases. They mine time-series data about Google Play app over a period of 12 months and survey developers of significant releases to check their results. Tian et al. [36] use Random Forest to predict whether an app will be high-rated. They extract 28 factors from eight dimensions, such as app size and library quality. Their findings show that external factors (e.g., number of promotional images) are the most influential factors. Our study also uses machine learning techniques, however, our main goal is to detect projects that are not going to be actively maintained. Moreover, our study extracts project, contributor and owner features that we input to the machine learning models.

Open source projects maintainability. In previous work [8], we survey maintainers of 104 failed open source projects to understand the rationale for such failures. Their findings revealed that projects fail due to reasons associated with project properties (e.g., low maintainability), project team (e.g., lack of time of the main contributor), and to environment reasons (e.g., project was usurped by a competitor). Later, we report results of a survey with 52 developers who recently became core contributors on popular GitHub projects [9]. Our results show the developer's motivations to assume an important role in FLOSS projects (e.g., to improve the projects because they use them), the project characteristics (e.g., a friendly community), and the obstacles they faced (e.g., lack of time of the project leaders).

Also related is the work by Yamashita et al. [37], which adapts two population migration metrics in the context of open source projects. Their analysis enables the detection of projects that may become obsolete. Khondhu et al. [15] report that more than 10,000 projects are inactive on SourceForge. They use the maintainability index (MI) [27] to compare the maintainability between inactive projects and projects with different statuses (active and dormant). Their results reveal that the majority of inactive systems are abandoned with a similar or increased maintainability, when compared to their initial status. Nonetheless, there are critical concerns on using MI as a predictor of maintainability [3]. Eghbal [11] reports risks and challenges to maintain modern open source projects. She argues that open source plays a key role in the digital infrastructure of our society today. Opposed to physical infrastructure (e.g., bridges and roads), open source projects still lack a reliable and sustainable source of funding. Other recent research on open source has focused on the organization of successful open source projects [26] and on how to attract and retain contributors [7, 19, 29, 33, 38]. Our work enhances the aforementioned work by contributing factors and proposing the use of machine learning to accurately identify projects that are not going to be actively maintained.

7 CONCLUSION

In this paper, we proposed a machine learning model to identify unmaintained GitHub projects. By our definition, this status includes three types of projects: finished projects, deprecated projects, and

stalled projects. We validated the proposed model with the principal developers of 129 projects, achieving a precision of 80% (RQ1). Then, we used the model with 112 deprecated projects—as explicitly mentioned in their GitHub page. In this case, we achieved a recall of 96% (RQ3). We also showed that the proposed model can identify unmaintained projects early, without having to wait for one year of inactivity, as commonly proposed in the literature (RQ3). Finally, we defined a metric, called Level of Maintenance Activity (LMA), to assess the risks of projects become unmaintained. We provided evidence on the applicability of this metric, by investigating its usage in 2,927 projects classified as under maintenance, in our dataset.

Due to its high accuracy (precision= 80% and recall= 96%), the model proposed in this paper can be used by developers to check the maintenance status of an open source project, before deciding to use it. This information has a key value, since there is a growing concern on the sustainability of modern open source projects [11].

As future work, we intend to implement a tool to provide information about the maintenance status and the level of maintenance activity of open source projects. The dataset used in this paper is available at: *https://zenodo.org/record/1313637*. Due to privacy concerns, we are omitting the name of the unmaintained projects.

ACKNOWLEDGMENTS

Our research is supported by CAPES, FAPEMIG, and CNPq. We would also like to thank the 112 GitHub developers who kindly answered our survey.

REFERENCES

[1] Daniel Alencar da Costa, Surafel Lemma Abebe, Shane McIntosh, Uirá Kulesza, and Ahmed E Hassan. 2014. An Empirical Study of Delays in the Integration of Addressed Issues. In *International Conference on Software Maintenance and Evolution (ICSME)*. 281–290.

[2] Lingfeng Bao, Zhenchang Xing, Xin Xia, David Lo, and Shanping Li. 2017. Who will leave the company?: a large-scale industry study of developer turnover by mining monthly work report. In *14th International Conference on Mining Software Repositories (MSR)*. 170–181.

[3] Dennis Bijlsma, Miguel Alexandre Ferreira, Bart Luijten, and Joost Visser. 2012. Faster issue resolution with higher technical quality of software. *Software Quality Journal* 20, 2 (2012), 265–285.

[4] Hudson Borges, Andre Hora, and Marco Tulio Valente. 2016. Understanding the Factors that Impact the Popularity of GitHub Repositories. In *32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 334–344.

[5] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[6] M Luz Calle and Víctor Urrea. 2010. Letter to the editor: stability of random forest importance measures. *Briefings in bioinformatics* 12, 1 (2010), 86–89.

[7] Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2012. Who is going to mentor newcomers in open source projects?. In *20th International Symposium on the Foundations of Software Engineering (FSE)*. 44–54.

[8] Jailton Coelho and Marco Tulio Valente. 2017. Why Modern Open Source Projects Fail. In *11th Symposium on The Foundations of Software Engineering (FSE)*. 186–196.

[9] Jailton Coelho, Marco Tulio Valente, Luciana L. Silva, and Andre Hora. 2018. Why We Engage in FLOSS: Answers from Core Developers. In *11th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. 114–121.

[10] Daniel Alencar da Costa, Surafel Lemma Abebe, Shane McIntosh, Uirá Kulesza, and Ahmed E. Hassan. 2014. An empirical study of delays in the integration of addressed issues. In *30th International Conference on Software Maintenance and Evolution (ICSME)*. 281–290.

[11] Nadia Eghbal. 2016. *Roads and Bridges: The Unseen Labor Behind Our Digital Infrastructure.* Technical Report. Ford Foundation.

[12] Georgios Gousios, Martin Pinzger, and Arie van Deursen. 2014. An exploratory study of the pull-based software development model. In *36th International Conference on Software Engineering (ICSE)*. 345–355.

[13] Andre Hora, Marco Tulio Valente, Romain Robbes, and Nicolas Anquetil. 2016. When Should Internal Interfaces be Promoted to Public?. In *24th International Symposium on the Foundations of Software Engineering (FSE)*. 280–291.

[14] Javier Luis Cánovas Izquierdo, Valerio Cosentino, and Jordi Cabot. 2017. An Empirical Study on the Maturity of the Eclipse Modeling Ecosystem. In *20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 292–302.

[15] Jymit Khondhu, Andrea Capiluppi, and Klaas-Jan Stol. 2013. Is it all lost? A study of inactive open source projects. In *9th International Conference on Open Source Systems (OSS)*. 61–79.

[16] Stefan Koch and Georg Schneider. 2002. Effort, co-operation and co-ordination in an open source software project: GNOME. *Information Systems Journal* 12, 1 (2002), 27–42.

[17] Ahmed Lamkanfi, Serge Demeyer, Emanuel Giger, and Bart Goethals. 2010. Predicting the severity of a reported bug. In *7th IEEE Working Conference on Mining Software Repositories (MSR)*. 1–10.

[18] J. Richard Landis and Gary G. Koch. 1977. The Measurement of Observer Agreement for Categorical Data. *Biometrics* 33, 1 (1977), 159–174.

[19] Amanda Lee, Jeffrey C. Carver, and Amiangshu Bosu. 2017. Understanding the Impressions, Motivations, and Barriers of One Time Code Contributors to FLOSS Projects: A Survey. In *39th International Conference on Software Engineering (ICSE)*. 1–11.

[20] Meir M. Lehman. 1980. Programs, life cycles, and laws of software evolution. *IEEE* 68, 9 (1980), 1060–1076.

[21] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. 2008. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering* 34, 4 (2008), 485–496.

[22] Gilles Louppe, Louis Wehenkel, Antonio Sutera, and Pierre Geurts. 2013. Understanding variable importances in forests of randomized trees. In *26th Advances in Neural Information Processing Systems (NIPS)*. 431–439.

[23] William Martin, Federica Sarro, and Mark Harman. 2016. Causal impact analysis for app releases in google play. In *24th International Symposium on Foundations of Software Engineering (FSE)*. 435–446.

[24] Tom Mens, Mathieu Goeminne, Uzma Raja, and Alexander Serebrenik. 2014. Survivability of Software Projects in Gnome–A Replication Study. In *7th International Seminar Series on Advanced Techniques & Tools for Software Evolution (SATToSE)*. 79 – 82.

[25] Tim Menzies, Andrew Butcher, David Cok, Andrian Marcus, Lucas Layman, Forrest Shull, Burak Turhan, and Thomas Zimmermann. 2013. Local versus global lessons for defect prediction and effort estimation. *IEEE Transactions on software engineering* 39, 6 (2013), 822–834.

[26] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. 2002. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11, 3 (2002), 309–346.

[27] Paul Oman and Jack Hagemeister. 1992. Metrics for assessing a software system's maintainability. In *8th International Conference on Software Maintenance (ICSM)*. 337–344.

[28] Fayola Peters, Tim Menzies, and Andrian Marcus. 2013. Better cross company defect prediction. In *10th Working Conference on Mining Software Repositories (MSR)*. 409–418.

[29] Gustavo Pinto, Igor Steinmacher, and Marco A. Gerosa. 2016. More common than you think: An in-depth study of casual contributors. In *23th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 112–123.

[30] Foster Provost and Tom Fawcett. 2001. Robust classification for imprecise environments. *Machine learning* 42, 3 (2001), 203–231.

[31] Karthik Ramasubramanian and Abhishek Singh. 2017. *Machine Learning Model Evaluation.* Apress.

[32] Gregorio Robles, Jesus M. Gonzalez-Barahona, and Israel Herraiz. 2009. Evolution of the core team of developers in libre software projects. In *6th International Working Conference on Mining Software Repositories (MSR)*. 167–170.

[33] Igor Steinmacher, Tayana U. Conte, Christoph Treude, and Marco A. Gerosa. 2016. Overcoming open source project entry barriers with a portal for newcomers. In *38th International Conference on Software Engineering (ICSE)*. 273–284.

[34] Ferdian Thung, David Lo, and Lingxiao Jiang. 2012. Automatic defect categorization. In *19th Working Conference on Reverse Engineering (WCRE)*. 205–214.

[35] Yuan Tian, David Lo, Xin Xia, and Chengnian Sun. 2015. Automated prediction of bug report priority using multi-factor analysis. *Empirical Software Engineering* 20, 5 (2015), 1354–1383.

[36] Yuan Tian, Meiyappan Nagappan, David Lo, and Ahmed E. Hassan. 2015. What are the characteristics of high-rated apps? a case study on free Android applications. In *30th International Conference on Software Maintenance and Evolution (ICSME)*. 301–310.

[37] Kazuhiro Yamashita, Shane McIntosh, Yasutaka Kamei, and Naoyasu Ubayashi. 2014. Magnet or sticky? an oss project-by-project typology. In *11th working conference on mining software repositories (MSR)*. 344–347.

[38] Minghui Zhou and Audris Mockus. 2015. Who will stay in the floss community? modeling participant's initial behavior. *Transactions on Software Engineering* 41, 1 (2015), 82–99.