PROMISE '17

The 13th International Conference on Predictive Models and Data Analytics in Software Engineering

Nov 8, 2017
Toronto, Canada

General Chair:
Burak Turhan

Program Chairs:
David Bowes & Emad Shihab

The Association for Computing Machinery
2 Penn Plaza, Suite 701
New York New York 10121-0701

ACM COPYRIGHT NOTICE. Copyright © 2017 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1-(212)-869-0481, or permissions@acm.org.

For other copying of articles that carry a code at the bottom of the first or last page, copying is permitted provided that the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, +1-978-750-8400, +1-978-750-4470 (fax).

ACM ISBN: 978-1-4503-5305-2

Welcome to the 13th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE 2017).

It is our pleasure to welcome you to the 13th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE 2017), to be held in Toronto, Canada, on November 8, 2017, co-located with the 11th International Symposium on Empirical Software Engineering and Measurement (ESEM 2017).

PROMISE is an annual forum for researchers and practitioners to present, discuss and exchange ideas, results, expertise and experiences in construction and/or application of predictive models and data analytics in software engineering. Such models and analyses could be targeted at: planning, design, implementation, testing, maintenance, quality assurance, evaluation, process improvement, management, decision making, and risk assessment in software and systems development.

This year PROMISE received a total of 25 paper submissions. After a reviewing process, in which each paper has been reviewed by at least three members of the program committee, and in which an online discussion was held, we accepted a total of 12 papers (1 (out of 4) short papers and 11 (out of 21) full papers), which will be presented in 4 technical sessions. The acceptance criteria were entirely based on the quality of the papers, without imposing any constraint on the number of papers to be accepted.

We are delighted to announce an outstanding keynote that will be delivered by Prof. Ahmed E. Hassan, Queen's University. His keynote will be about "Are we drinking too much Big Data and Machine Learning Kool-Aid?!".

We would like to thank all authors for submitting high quality papers, and program committee members as well as external reviewers for their timely and accurate reviewing activity. Last, but not least, we would like to thank the ESEM 2017 organizers for hosting PROMISE 2017 as a co-located event and for their logistic support in the organization of the conference.

We hope you will enjoy PROMISE 2017. We certainly will!

Many thanks from Burak Turhan, David Bowes and Emad Shihab.

The PROMISE 2017 Organization and Steering Committee:

Leandro Minku	University of Leicester
Andriy Miranskyy	Ryerson University
Massimiliano Di Penta	University of Sannio
Burak Turhan	Brunel University
Hongyu Zhang	University of Newcastle

General Chair:

Burak Turhan	Brunel University

Program Chairs:

David Bowes	University of Hertfordshire
Emad Shahib	Concordia University

Publicity and Social Media Chair:

Federica Sarro	University College London

Publication Chair:

David Bowes	University of Hertfordshire

Local Organization Chair:

Andriy Miranskyy	Ryerson University

And special thanks to our PC and reviewers:

Lefteris Angelis	Aristotle University of Thessaloniki
Gabriele Bavota	Università della Svizzera italiana (USI)
Hoa Khanh Dam	University of Wollongong
Massimiliano Di Penta	University of Sannio
Tracy Hall	Brunel University
Rachel Harrison	Oxford
Jacky Keung	City University of Hong Kong
Foutse Khomh	DGIGL, École Polytechnique de Montréal
Ekrem Kocaguneli	West Virginia University
Gernot Liebchen	Bournemouth University
Chris Lokan	University of New South Wales @ ADFA
Lech Madeyski	Wroclaw University of Science and Technology
Shane Mcintosh	McGill University
Emilia Mendes	The University of Auckland
Tim Menzies	ncstate, CS
Leandro Minku	University of Leicester
Andrea Mocci	Faculty of Informatics, University of Lugano, Switzerland
Jaechang Nam	University of Waterloo
Jasmin Ramadani	University of Stuttgart
Rudolf Ramler	Software Competence Center Hagenberg
Daniel Rodriguez	The University of Alcalá
Federica Sarro	University College London
Martin Shepperd	Brunel University
Ayse Tosun	Istanbul Technical University
Stefan Wagner	University of Stuttgart
Hironori Washizaki	Waseda University
Dietmar Winkler	Vienna University of Technology
Xin Xia	University of British Columbia
Yang Ye	Stevens Institute of Technology
Yuming Zhou	Nanjing University, China

Contents

delegate copy

Studying Test Case Failure Prediction for Test Case Prioritization

Tanzeem Bin Noor
Highjump Software Inc.
125 Commerce Valley Drive West
Markham, Ontario L3T 7W4
tanzeem.noor@gmail.com

Hadi Hemmati
Department of Electrical and Computer Engineering
2500 University Drive NW
Calgary, Alberta T2N 1N4
hadi.hemmati@ucalgary.ca

ABSTRACT

Background: Test case prioritization refers to the process of ranking test cases within a test suite for execution. The goal is ranking fault revealing test cases higher so that in case of limited budget one only executes the top ranked tests and still detects as many bugs as possible. Since the actual fault detection ability of test cases is unknown before execution, heuristics such as “code coverage" of the test cases are used for ranking test cases. Other test quality metrics such as “coverage of the changed parts of the code" and “number of fails in the past"’ have also been studied in the literature. **Aims:** In this paper, we propose using a logistic regression model to predict the failing test cases in the current release based on a set of test quality metrics. **Method:** We have studied the effect of including our newly proposed quality metric (“similarity-based" metric) into this model for tests prioritization. **Results:** The results of our experiments on five open source systems show that none of the individual quality metrics of our study outperforms the others in all the projects. **Conclusions:** However, the ranks given by the regression model are more consistent in prioritizing fault revealing test cases in the current release.

KEYWORDS

Test Case Prioritization; Test Case Quality Metrics; Regression Model

ACM Reference format:
Tanzeem Bin Noor and Hadi Hemmati. 2017. Studying Test Case Failure Prediction for Test Case Prioritization. In *Proceedings of PROMISE , Toronto, Canada, November 8, 2017,* 10 pages.
https://doi.org/10.1145/3127005.3127006

1 INTRODUCTION

In software testing, the ultimate effectiveness of test cases is measured based on their ability to detect real faults, i.e., how many faults the test can reveal when it executes. Unfortunately, this measure is not always practical because one needs to know about the effectiveness of test cases before execution (e.g., in the context of test case prioritization).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PROMISE , November 8, 2017, Toronto, Canada
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5305-2/17/11…$15.00
https://doi.org/10.1145/3127005.3127006

Test case prioritization is an important element in software quality assurance in practice. It is even more crucial in modern software development processes, where continuous integration is widely adapted. Source codes are frequently changed due to either adding new functionality or fixing known bugs. A study shows that in Google, more than 20 code changes occur in every minute [38]. Each change typically requires (re)testing the software. This will typically lead to huge test suites. For instance in the previous example from Google, the frequent changes has led to 1.5 million test executions per day at Google, in 2014.

Most of the time, software companies have limited resources (e.g., personnel, budget, and time) for testing and they can not retest the entire test suite after every change. To solve this problem, test case prioritization is used, which assigns ranks to the test cases and the tests are executed according to the order of their ranks, until the testing budget is ran out. Test cases are prioritized in a way that they detect the potential faults earlier. Therefore, it is very important to rank the effective test cases higher so that the faults can be detected by executing only the higher ranked tests within specific budget. As the ultimate effectiveness metric (i.e., actual fault detection rate) can be measured only after test execution, it can not be used in the context of test case prioritization, which assigns ranks to the test cases before execution. Therefore, we need other test quality metrics to estimate the effectiveness of the tests cases and prioritize them accordingly.

Several metrics such as code coverage, code change-related metrics, previous fault detection capability etc. are used to measure the test case quality. Code coverage measures how much of the source code (e.g., number of lines, blocks, conditions etc.) from the program is covered during the test execution. In contrast, change-related test quality metrics emphasize more on covering the changed part of the program source, which is also very crucial, specially in the context of regression testing. Previous fault detection metrics take a different approach to measure test quality. They consider a test as effective if it detects any fault from previous releases.

In this paper, we have studied the effectiveness of several existing metrics (i.e., coverage, change-related metrics, previous fault detection capability etc.) in prioritizing fault revealing tests. We specifically focus on combining existing metrics using a regression model and predict the probability of a test case detecting a fault in the current release. Therefore, test cases are prioritized based on their probability of failure (the higher the probability, the higher the priority). We have conducted a series of empirical study using five open source java projects (total 247 versions) to evaluate ranking of the failing test cases provided by the regression model in the context of test case prioritization. The results of our study show that in general, combining several existing metrics in a regression

model provides better ranking compared to the ranking provided by individual metrics.

We have also experimented with including a newly introduced test quality metric "Similarity-based metric" [26] into our regression models. The results show that the model including the new metric outperforms the model with only traditional metrics.

The rest of this paper is organized as follows: We mention background and some basic quality metrics in section 2. Our proposed regression model using similarity-based metrics and existing metrics is explained in section 3. We discuss our experiments and results in section 4. Section 5 mentions some of the related works. Finally, section 6 concludes the paper and mentions our future work.

2 BACKGROUND

Test cases are prioritized using some quality metrics. Test case quality metrics are used to evaluate the tests and find the scope of improvement required in the test cases. A well-known set of such quality metrics are test adequacy criteria [27, 29]. A test adequacy criterion is a predicate that defines which properties of a program must be exercised, if the test is to be considered adequate with respect to the specific criterion [14]. Two main types of test adequacy criteria are coverage-based and fault-based test adequacy criteria [27, 29].

Besides these test adequacy criteria, there are some other quality metrics that may have high correlation to fault detection ability of the test cases, such as historical fault detection, coverage, size of the test, complexity, code churn, etc. [7]. In the rest of this section, we explain two categories of test adequacy criteria and historical fault detection metric.

2.1 Coverage-based Test Adequacy Criteria

Coverage is a commonly used test adequacy criteria that indicates how much of the program is executed when the test case runs. A test case may reveal a fault when it executes a segment from the program that causes the failure and therefore, high coverage (executing more segments from the source code) of a test case may indicate higher probability of failure. Coverage based test adequacy criteria is further grouped into control-flow coverage and data-flow coverage. Adequacy criteria based on control-flow could be categorized into number of coverage levels, such as Procedure/Method Coverage, Branch Coverage, Statement Coverage, Path Coverage and condition coverage.

Data-flow coverage criteria depend on the flow of data through the program. These criteria focus on the values associated with variables (def) and how these associations can affect the execution of the program (use) [39]. The most well-known data-flow coverage criterion is DU-pair coverage, where the goal is covering all pairs of definitions and uses per variable [29]. According to data-flow coverage criterion, a test case is considered to be effective when it's DU-pair coverage is high.

2.2 Fault-based Test Adequacy Criteria

Fault-based adequacy criteria measures the quality of the test cases by their ability to detect known faults, as an estimate for their ability for detecting unknown faults [18]. The known faults are either real faults or artificial faults (mutants – small syntactic changes in the program [29]). In mutation testing, the effectiveness of the test case is measured by the number of killed mutants by the test case. More killed mutants by a test case infers that the test case is more effective in finding real faults in the program, as well and hence the tests are ranked higher than the others.

2.3 Historical Fault Detection

Historical fault detection quality metric considers a test case to be effective in the current release if the same test was also able to detect faults in previous releases (i.e., the test has failed previously) [2, 7, 21, 40]. The rationale behind this is that studies have shown that the tests with higher historical fault detection rate are more likely to fail in the current version as well [40]. Therefore, the previously failing tests are also prioritized in the current release, specially, in the context of regression testing.

However, the test case that fails in the current release might not be always same as the previously failed test cases, but they might be quite similar. For example, when a test case is slightly modified from previous release, the test case is not considered as effective according to the traditional historical fault detection quality metric, since it did not exist in the previous releases, to fail and therefore, it is ranked lower in the prioritized list. However, if the original test case from the previous release was a failing test, there is still high chance that the modified version fails in the new release. In our previous work [26], we have shown such test cases that are similar to the previously failing test cases, in terms of their method sequence calls.

We proposed a set of similarity-based metrics that uses execution traces of the previously failed test cases from history and measures quality of a test case in the current release using it's similarity to the past failing traces. We found that our proposed similarity-based metrics which also utilize historical faults (i.e., failing traces), outperforms the traditional historical fault detection metric, in terms of identifying fault revealing test cases in the current release.

3 METHODOLOGY

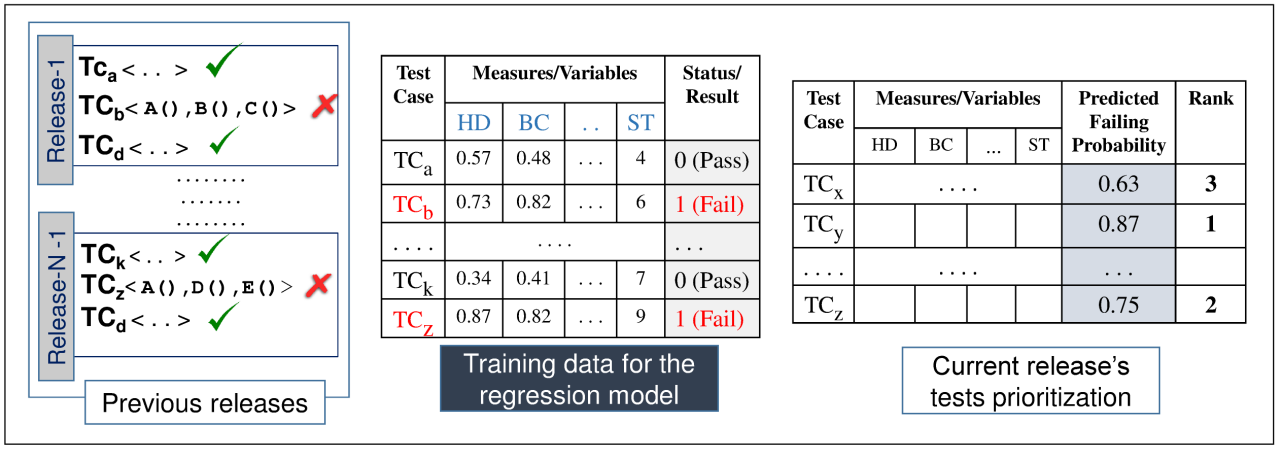
In this section, we investigate the effectiveness of individual test quality metrics as well as a combined version of them in a regression model. In addition, we have proposed to build a regression model that uses the existing quality metrics and our recently proposed similarity-based metrics. In the rest of this section, we explain the quality metrics that we use to build the model in this study.

3.1 Traditional Quality Metrics

In this paper, we consider the following traditional quality metrics for test case prioritization:

3.1.1 Traditional Historical Fault detection metric (TM). The traditional historical fault-detection metric (TM) considers a test as effective if it detects any fault from previous versions (i.e., failed previously) [1, 2, 7, 21, 40]. We measure TM of a a given test case, in one release, by counting the number of it's failing occurrences. The higher the TM, the higher the test case rank.

3.1.2 Method Coverage (MC). Method Coverage (MC) is a simple control-flow coverage criterion that measures procedure/method coverage of a test case. For each test case, the method coverage



Test Case	Measures/Variables				Status/ Result
	HD	BC	. .	ST	
TC_a	0.57	0.48	. . .	4	0 (Pass)
TC_b	0.73	0.82	. . .	6	1 (Fail)
. . . .	. . . .				. . .
TC_k	0.34	0.41	. . .	7	0 (Pass)
TC_z	0.87	0.82	. . .	9	1 (Fail)

Test Case	Measures/Variables				Predicted Failing Probability	Rank
	HD	BC	...	ST		
TC_x	. . . .				0.63	**3**
TC_y					0.87	**1**
. . . .	. . . .				. . .	
TC_z					0.75	**2**

Figure 1: Test case prioritization using regression model

refers to the number of source code methods called from the test case (directly or indirectly), divided by the total number of methods in the source code [27]. According to MC metrics, the test case that covers more methods/procedures, is considered to be more effective and ranked higher.

3.1.3 Changed Method Coverage (CMC). Often times in test prioritization, specially in the context of regression testing, one would like to make sure that modified parts of the source code are well-tested [7]. The simple MC coverage is not the best metric for measuring such coverage because for instance one can cover most of the modified part of the source code with a low-to-moderate total coverage (MC) [26]. In this paper, we define Changed Method Coverage (CMC) metric of a test case as the number of changed methods from the previous version that are directly or indirectly executed when the test case is executed, and again the higher CMC value of a test case suggests a higher rank.

3.1.4 Size of Tests (ST). Typically, size of a test case refers to either the LOC (Line of Codes) or the number of "assertions" in a test case. Note that size of a test case is not necessarily correlated with its coverage [29]. For example, test case t_1 can cover a method with one assertion and test case t_2 do the same but try it with 5 more assertions. In this case, t_2 may have higher chances to detect faults than t_1 [26]. In our study, we consider LOC (Line of Codes) as a measure of test size (ST), where larger sizes indicate that the tests have higher probability to detect faults.

3.2 Similarity-based Quality Metrics

In our previous work, we defined a family of similarity-based metrics that prioritize the test cases when they are similar to any of the failed test cases from previous versions [26]. We defined similarity between test cases based on their execution traces containing sequences of method calls from program source code. Execution traces of method calls for all previously failed test cases are collected from history (once per release) and compared against the method call sequences of the current release test cases.

A similarity function was used to determine the current release test's similarity to the previously failed test cases. The inputs of a similarity function are: a) traces from previously failed test cases and b) a trace from the current release test case. The similarity function, at the end, returns a similarity value as output. Finally, the quality of the test cases are measured based on the returned similarity values and those with higher similarity values are prioritized. [26].

In our previous work, we defined four different similarity functions to implement the similarity-based metrics [26]. In this study, we propose to build a regression model using the similarity-based metrics along with the existing metrics mentioned in the previous subsection. In the rest of this subsection we briefly explain the four similarity-based metrics:

3.2.1 Basic Counting (BC). The Basic Counting (BC) similarity function simply counts the unique method calls from the current release's tests trace that also appear in the past failing sequences. In other words, BC is equal to the number of common and unique method calls between the given test case trace and the previously failed traces. We have normalized the similarity values between 0 and 1 by diving the total number of common and unique method calls by the total number of unique method calls from the history [26].

3.2.2 Edit Distance (ED). The general edit distance function is a sequence-aware function that considers the order of method calls in the traces. We have used one of the most well-known implementation of edit-distance, called Levenshtein distance [9], where the similarity between two sequences of method calls are calculated using the different rewarding points that are assigned to all the matches, mismatches and gaps in the sequences [26]. So, the lower edit distance indicates higher similarity and the test cases having higher similarity are ranked higher.

3.2.3 Hamming Distance (HD). Hamming Distance between two sequences is the minimum number of edit operations (insertions, deletions, and substitutions) required to transform the first sequence into the second [9, 16]. Note that hamming is only applicable on

Table 1: Projects Under Study

Projects	#Faults (#Versions)	#Test Cases	Median number of Test cases per version
JFreeChart	26	2,205	1,751
Closure Compiler	133	7,927	7,066
Commons Math	106	3,602	1,976
Commons Lang	65	2,245	1,757
Joda Time	27	4,130	3,748

identical length inputs and is equal to the number of substitutions required in one input to become the second one [9]. However, in our case, the sequences of method calls may have different lengths. Therefore, to force them to have an identical length, at first, we have made a vector V that contains all the method calls from two comparing sequences. Then, we have encoded the each test case using a binary vector of the length of V, where a bit is true if the corresponding method call appears in the test case. Finally, the hamming distance between two sequences of method calls is measured by applying XOR operation between their binary encoded representations and then normalized between 0 and 1 by dividing the sum of XOR values by the length of V [26]. The low hamming distance value of a test case means the test case has high similarity with the previous failing test cases and should be ranked higher.

3.2.4 Improved Basic Counting (IBC). We proposed an Improved Basic Counting (IBC) similarity function that combines the Basic Counting (BC) and Hamming Distance (HD) similarity functions. IBC is a weighted version of BC and HD, where BC values are used as the default similarity value of IBC and therefore, it is called the improved BC. However, if the BC similarities are very low (i.e., less than a threshold) then the IBC will use the HD similarity value (unless HD's values are also lower than the threshold, which in this case the default BC value would be used) [26]. Therefore, test cases having higher IBC values will be prioritized.

3.3 Building Regression Model

In this paper, we propose to combine our similarity-based metrics with existing quality metrics into a regression model and prioritize the test cases based on the estimated failure probabilities. But first we build a regression model using only the traditional metrics (ED, HD, BC, IBC, TM, MC, CMC, ST) of each test case from previous releases as independent variables. Basically, to predict the ranks of the test cases in the current version of a project, the regression model is trained using different quality measures from all the previous versions. For example, to rank the test cases in a latest version (e.g., version 10) of a project, the model is trained using all quality measure values from version 1 to 9.

Since a test case can either pass or fail in the actual execution, the dependent variable of the regression model is binary (pass or fail), which makes the model a logistic regression model. Now given these quality measures of each test case from previous releases, the model is actually trained based on the actual status of each test case (i.e., whether it was passed or failed).

After applying the trained model on the current release's test cases, they are ranked based on the probability value of the test being failed. The overall process of combining quality metrics using a regression model is shown in Figure 1, where the higher rank of test case TC_y indicates that it should be executed earlier than other test cases (e.g., TC_z, TC_x) to reveal faults in the current release.

It is worth mentioning that there might be several independent variables in the model that are highly correlated, which is referred as *multicollinearity* problem. In addition, all the variables might not be statistically significant for the prediction model. To deal with these problems, we have removed highly correlated variables (i.e., any variables with correlation higher than 0.5) as suggested by Shihab *et al.* [35]. We have performed this removal in an iterative manner, until all the factors left in the model has a Variance Inflation Factor (VIF) below 2.5, as recommended by previous works [5, 35]. Thus, the regression model is built with the least number of uncorrelated factors. In addition, we have also measured the p-value of the independent variables for statistical significance and ensured that it is less than 0.05. Note that the remaining number of uncorrelated factors in the model might be different for different versions of a project [35].

So far we only used the traditional metrics for prediction, but the regression model can also contain our similarity-based metrics. We will discuss this in RQ2 of the Empirical Study section.

4 EMPIRICAL STUDY

In this section, we explain our empirical study and discuss the results of our experiments.

4.1 Research Questions

We have investigated the following two research questions in this study:

RQ1: Can combining the traditional test quality metrics using a regression model improve test prioritization compared to ranking the test cases using the individual metrics?

RQ2: Can we improve test prioritization results by adding similarity-based quality metrics into the traditional model of RQ1?

4.2 Subjects Under Study

We have used five different Java projects in our experiment. The projects have been retrieved from the *defects4j* database [8]. The database provides 357 faults and 20,109 Junit tests from five different open-source Java projects as mentioned in Table 1 [26]. All the faults are real, reproducible and have been isolated in different versions [19]. There is a fixed version and a faulty version for each fault of the program source code. The faulty source code is modified to remove the fault in the fixed version. The test cases are exactly the same in both of the faulty and the fixed versions. In each version, there is at least one test case (a Junit test method) that fails on the faulty version but passes on the fixed version [25] [26].

Table 2: Number of Studied Version

Projects	#Studied Versions	#Versions (# new/modified tests >=5)	#Versions (TM works)
Commons Lang	60	41	20
JFreeChart	24	16	2
Commons Math	43	36	12
Joda Time	25	16	0
Closure Compiler	95	85	0

4.3 Data Collection

We have extracted the basic quality metrics, i.e., Size of Testcase (ST), Method Coverage (MC), Changed Method Coverage (CMC) and Traditional historical fault-based Metric (TM) directly from the projects.

ST is the number of uncommented statements in a test method that has been derived from Java Abstract Syntax Tree (AST) parser using Eclipse JDT API [11]. MC of a test case is the number of unique methods called during the test execution. We have calculated this measure from the execution traces produced by Daikon and AspectJ [26]. CMC of a test case is the number of unique method calls that are called by the test case and have been changed since the previous version. To calculate CMC, we first identified the list of source code method names in the current version that have been changed from the immediate previous version. Then we count the change methods that also appear in the execution trace of modified/new test cases of the current version. The final CMC value of a test case has been normalized between 0 and 1 by dividing the total number of changed methods by the total number of unique method calls from the test execution trace.

TM measures how many times the tests from the current release failed previously. However, in our study, we found that in two of the projects (i.e., Commons Math and Closure Compiler), TM is not available in any version, as the fault revealing test cases in any given release did not fail in the previous releases. Therefore, test cases could not be ranked using TM in these two projects and in JFreeChart project, TM only works for 2 versions. Table 2 shows the detailed number of versions where TM works for each project .

Other than those 4 basic quality metrics, we have also extracted the four similarity-based metrics (BC, HD, ED, IBC), which are explained in section 3.2, to build a better prediction model, in RQ2. We have used Daikon [12] to produce the execution traces from three projects (Commons Lang, Joda Time and JFreeChart). Daikon dynamically detects likely program invariants and it allows to detect properties in C, C++, C#, Eiffel, F#, Java, Perl, and Visual Basic programs [12]. The Java front end (instrumenters) of Daikon, named Chicory, executes the target Java programs and generates the *.dtrace* file that contains the program execution flow along with the variable values in each program point. We have extracted the method sequence calls from the *.dtrace* file generated by Daikon [26].

For the other two projects (Commons Math and Closure Compiler), we have used AspectJ [10] instead of using daikon to produce the trace of method sequence calls, as the test cases from these projects generated very large *.dtrace* files.

Note that although the trace extraction process is different, the format of the extracted method sequence calls is exactly the same. We have collected the method sequences for all modified tests in the current release and also all the failed tests traces from the previous releases [26]. For all the modified test cases in a current release, we have predicted their probability of failing and ranked accordingly.

We have implemented the similarity functions using Java and we have used R [31] to build the regression models and to evaluate the results. We have used “glm" function in R [13] to build the logistic regression model. Further, we have used “predict" function [30] from R to get the response probability value (i.e., probability of a test being passed/failed, in our case) of the observations from the test set. Note that all the raw data of the experiment is available online. [1]

4.4 Evaluation Metric

To evaluate a list of prioritized test cases, we look at the rank of the first failing test case in the list. An effective prioritization technique is supposed to rank the failing test case higher. Note that this metric is useful when there is only one fault that the test cases are looking for (but there might be still several tests that may detect the fault). This was the case in our study and thus we chose this metric. The more general evaluation metric would be APFD [33], which can be applied for cases with multiple faults.

The defects4j dataset mentions which test case fails in the current version [8]. Therefore, we can easily extract the ranks of first failing test cases provided by different approaches (in this study using regression models and each individual metrics) for each version of a project. Furthermore, we then divided the rank of the first failing test by the total number of modified/new tests in each version, separately. We have done this to calculate the percentage of the test cases that need to be executed in order to catch the first fault in each version.

In case of tied ranks using any measure, we have ranked the tied test cases randomly, by applying the *rank* function in R with a parameter *ties.method=“random"* [34]. We have calculated the rank 30 times for each version of the projects to deal with this randomness.

We obtained one normalized rank value between 0 and 1 that represents the percentage of tests required to execute in order to identify the first fault. Since we have calculated the rank 30 times for each version, we have considered the median of these 30 values as the rank of first failing test for each version. We have also combined these median ranks from all the versions of a project and represented them using a boxplot distribution where the lower median line indicates better ranking for the project.

Whenever we compare two distributions of the results and say one outperforms the other, we have first conducted a non-parametric statistical significance test (U-test) [22] to make sure the differences are not due to the randomness of the algorithms. Besides measuring statistical significance, it is also crucial to assess the magnitude of the differences [4]. Effect size measures are used to analyze such a property [4, 15]. In our study, we have used a non-parametric effect size measure, called Vargha and Delaney’s $\hat{A}_{12}$ statistics [4, 15, 36]. Given a performance measure M, the $\hat{A}_{12}$ statistics measures the probability that running algorithm X yields higher M values than

[1] https://bitbucket.org/tanzeem_noor/promise17_data/src/

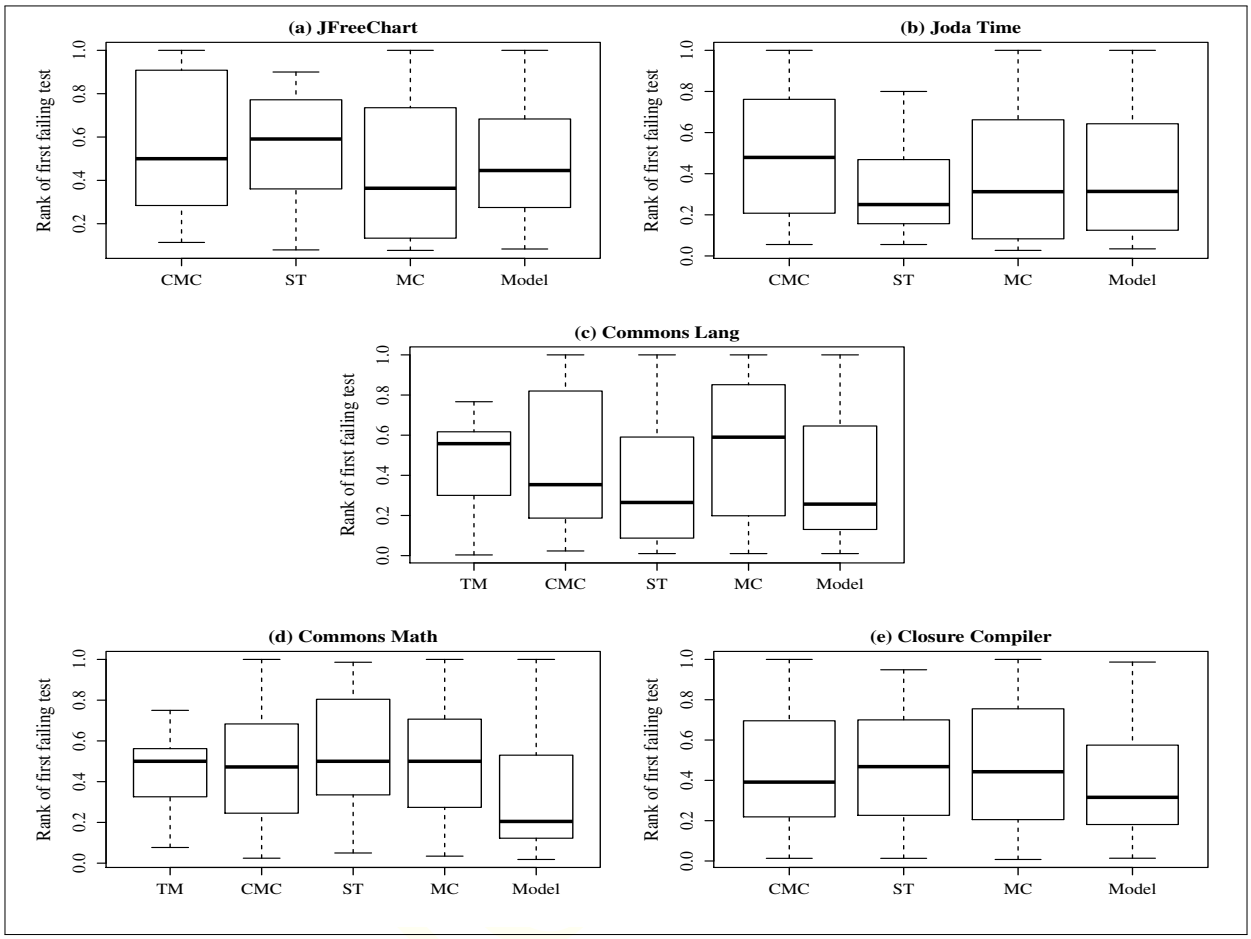


Figure 2: The ranks of the first failing test cases using TM, CMC, MC, ST, and regression model for each version of the project.

running another algorithm Y. When the two algorithms are equivalent, then $\hat{A}_{12}$ is 0.5. Therefore, while comparing algorithm X and Y, $\hat{A}_{12} = 0.8$ represents that we would obtain higher results in 80% of the time with algorithm X compared to the algorithm Y [4, 26].

In our context, the given performance measure M is the percentage of the tests need to be executed (i.e., normalized rank of the first failing test) in order to catch the first fault in a version. Therefore, we get a $\hat{A}_{12}$ value for each version of the project, while comparing two prioritization approaches. Thus, in our case, $\hat{A}_{12} = 0.8$ indicates that approach X ranks the first failing test higher than approach Y in 80% of the time. In other words, approach X can detect the fault faster than approach Y in 80% of time [26]. We have used this $\hat{A}_{12}$ measure to evaluate all of our results.

This is worth mentioning that we have considered only the versions with at least 5 modified/new test cases for the evaluation. We assume that in the versions having less than 5 modified/new test cases, executing all test cases is not costly and hence the prioritization is not actually beneficial [26]. The details of our number of studied versions is mentioned in Table 2.

4.5 Results and Discussion

In the rest of this section, we answer the research questions by comparing both the ranks of the first failing test cases and the $\hat{A}_{12}$ measure. We have also made sure that the differences of the results are statistically significant.

4.5.1 Experimental Results for RQ1. In RQ1, we have compared the ranks of the first failings test cases provided by each individual traditional metric (TM, CM, CMC, ST) and their combined set in a regression model (called Traditional Model). To answer RQ1, we have calculated median rank from 30 different runs for each version and represented median ranks of all versions of a project in a boxplot. Therefore, the lower median line of a boxplot represents better ranking, which indicates the average percentage of tests required to execute in order to catch the first fault of the projects, in each version.

Figure 2 compares the boxplot distribution of the first failing test's rank per project using different approaches. It can be seen that none of the metrics among ST, MC, CMC and TM completely outperforms the others in all the projects. For example, the ranking

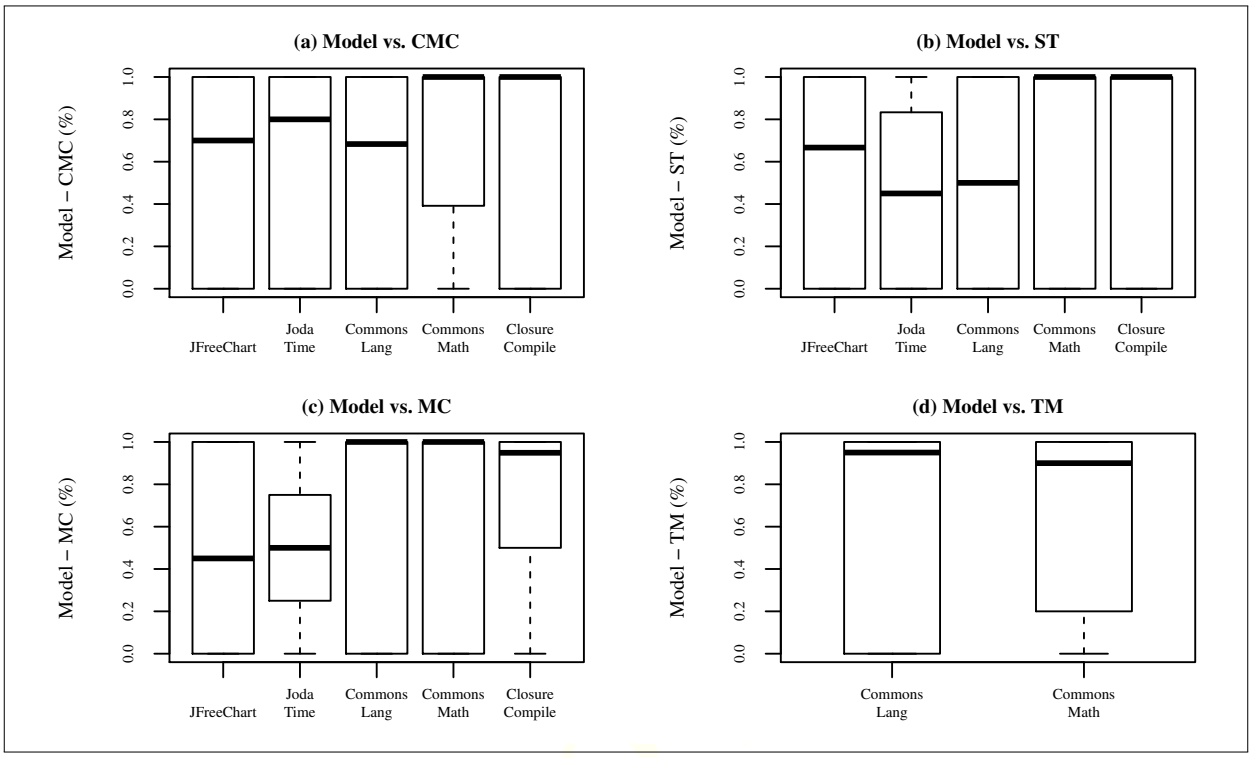


Figure 3: The boxplots of the effect size measures for finding the first fault using CMC, MC, ST, TM and regression model when comparing 30 runs of each versions of each project.

using ST is better than MC and CMC in Joda Time and Commons Lang project, however, for the other three projects, ST is falling behind the MC and CMC. Similarly, between CMC and MC, the ranking using MC provides better prioritization in JFreeChart and Joda Time project but the opposite behavior is observed for the other three projects except the Commons Math project, where ranking using MC and CMC looks similar.

Looking at the ranks of the first failing tests using the failure probability from the traditional regression model clearly outperforms all the other individual metrics in Commons Math and Closure Compiler project. On average, it requires only 20% and 25% tests to be executed in order to catch the first fault, in Commons Math and Closure Compiler project, respectively . Besides, in Commons Lang project, although the ranking provided by the regression model is better than the others (e.g., MC, CMC), it is equally good as the ST. However, the ranks using the regression model is only falling behind the ST and MC, in Joda Time and JFreeChart project, respectively.

We have also demonstrated the median ranks using the traditional historical fault based metric (TM) for the Commons Lang and Commons Math project. Note that TM does not work in any version of the Closure Compiler and Joda Time project (as mentioned in section 4.3 and Table 2). In addition, in the JFreeChart project, TM works only for 2 versions. Therefore, we have excluded these 3 projects while comparing their performance using TM with the other measures. However, for the other 2 projects, the prioritization using TM also falls behind the other metrics and the prediction model.

Besides showing the median rank of the first failing test, we have also shown the $\hat{A}_{12}$ measure comparing the performance of different prioritization approaches. Figure 3 shows the $\hat{A}_{12}$ measure distribution while comparing the performance of the regression model-based metric with other individual metrics for 30 runs of each version of a project. In the boxplots, the higher median line (i.e., higher than 0.5) represents better prioritization. Therefore, it can be seen from Figure 3(a) that the ranks of first failing test case using the regression model is always better than the CMC in Commons Math and Closure Compiler project, as the median line is in 1.0. The higher than 0.7 median line for the other 3 projects represent that on average the prioritization using the regression model is better than the CMC in 70% of the times.

According to Figure 3(b), prioritization using the regression model is clearly better than the ST in 3 of the projects (e.g., Commons Math, Closure Compiler and JFreeChart). However, the difference is not much significant in the Commons Lang project, as the median line is in 0.5. The only case where the ranking using model slightly falls behind the ST (i.e., the median line is just below the 0.5) is in Joda Time project.

In Figure 3(c), we observe that only for the JFreeChart project, MC performs slightly better than the model (i.e., in 60% of the cases, as the median line is in 0.4). However, the model clearly

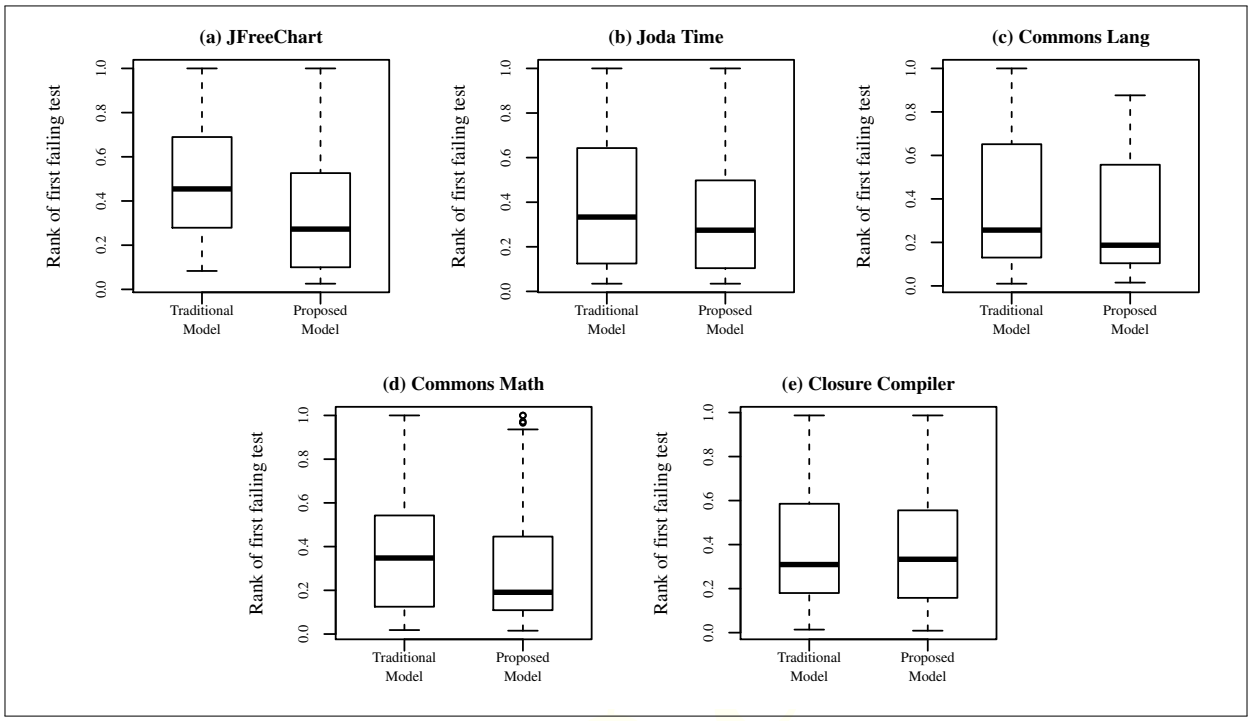


Figure 4: The boxplots of the average rank of the first failing test rank using two prediction models for all versions of each project (Traditional Model: uses all traditional metrics - Proposed Model: uses all traditional and similarity-based metrics).

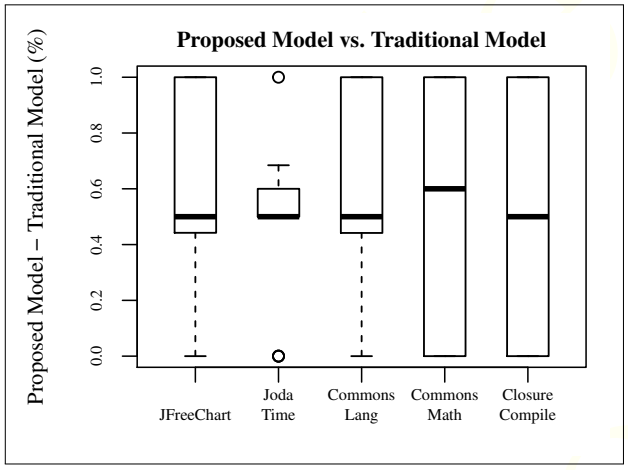


Figure 5: The boxplots of the effect size measures for finding the first fault using the two prediction models, when comparing 30 runs of each versions of each project.

outperforms the MC in the other projects, except in Joda Time, where they look similar. In addition, the median line leaning to 1.0 in Figure 3(d) represents that the ranking using the regression model is better than the TM in almost all the cases for Commons Lang and Commons Math project.

Therefore, the results shown in 2 and 3 represents that neither of the traditional individual metrics (e.g., TM, ST, CMC and MC) completely outperforms others for all the projects. However, the prioritization using the combined set of these metrics in a regression model is much more consistent in all the projects.

It can be also seen that in Commons Math and Closure Compiler project, the result using regression model is much better than any other metrics. One plausible reason would be the higher amount of historical data has been used to train the regression model for these 2 projects, which provides better ranks than the use of individual metrics. Note that Commons Math and Closure Compiler projects have data from more versions (106 and 133 versions respectively, mentioned in Table 1) compared to the JFreeChart and Joda Time projects (26 and 27 versions respectively). Therefore, the results suggest that the regression model could provide much better results when it is trained using enough data from longer history.

4.5.2 Experimental Results for RQ2. RQ2 investigates that how well can we predict the first failing test case when combining the similarity based metrics with the traditional metrics. To answer RQ2, we have compared the ranking using predicted ranks from two different logistic regression models. Recall that in the regression model using traditional metrics, we have considered all the traditional metrics (TM, ST, MC and CMC) as the predictors (i.e., independent variables). We call this regression model as the “Traditional Model". In our proposed regression model, we have appended the list of traditional predictors by adding our newly proposed

similarity-based metrics (ED, HD, BC and IBC) and therefore, we refer to this regression model as the "Proposed Model".

Figure 4 compares the distribution of the predicted ranks of the first failing tests for all versions of the projects using two models — the traditional and the proposed logistic regression model. Again, we have used the median rank from 30 different runs for each version of the projects. It can be seen that on average, the predicted ranks of the first failing tests from the proposed regression model are always lower than the ranks from the traditional regression model in four projects (JFreeChart, Joda Time, Commons Lang and Commons Math). However, the ranks provided by these two models look similar (the median line is at 0.5) for the Closure Compiler project. The same behavior is also observed in Figure 5 that compares the $\hat{A}_{12}$ measure distribution of the first failing tests ranks from the two regression models.

To summarize, we can say that in our study, combining traditional quality metrics in a regression model improved the prioritization effectiveness. In addition, supplementing the traditional model with similarity-based metrics boosted the effectiveness even more.

4.6 Threats to Validity

In terms of conclusion validity, we have conducted solid experiments to ensure that the results are statistically significant and the magnitude of differences are significant, as well (effect size).

In terms of internal validity, we have used existing libraries and tools as much as possible (e.g., Daikon [12], AspectJ [10]). However, there is a threat to validity of the extracted data (pass and fails) due to flaky tests. In terms of construct validity, we use a pretty well-known evaluation metric, which is the rank of first test that catches the fault. The other alternative would be APFD (Average Percentage of Faults Detected) [33], which is commonly used for test prioritization evaluation. However, in our study, each version contains only one fault. So, using APFD would not make sense [26].

In terms of external validity, we have conducted our empirical study based on five real-world open source java libraries from *defects4j* [8] database with several versions and faults [26]. However, generalizing the results to different types of systems may still require further experiments.

5 RELATED WORK

The most related work to this study are those that introduce new test quality metrics. In the rest of this section, we mention some of the related works.

In [37], Xie and Memon mentioned the characteristics for developing a "good" test case/suite that significantly affects the fault-detection effectiveness of the test suite. However, their specific concentration was the test case effectiveness in GUI testing. They found that the tests with more diversity and higher event coverage are better in detecting bugs from the GUI. In another study, Arcuri evaluated the effect of test case sequence length in testing programs with higher internal states [3]. The author showed that the longer test case sequence tend to lead to a higher level of coverage and hence are more effective in detecting faults. We have also considered the test case length (size and executed method sequence length) while combining metrics using the model.

A number of researchers proposed to use historical fault detection as a quality metric, in the context of regression test case selection, prioritization and minimization that makes the regression testing cost-efficient. Kim and Porter [20] proposed to prioritize tests based on historical test execution that also improves the overall regression testing. They considered the number of previous faults exposed by a test as the key prioritizing factor

Park *et al.* considered the test case execution costs and the severity of detected faults to prioritize tests in regression testing [28]. However, instead of specific time frame, the total history of the test execution was used to determine the historical value of the test case. Huang *et al.* also used historical record of test cases execution cost and severity of detected faults [17], however, they applied a search-based approach, i.e., genetic algorithm to generate prioritized test execution order in the current release.

All these traditional historical fault detection measures quality of the test cases and prioritize tests based on their previous fault detection capability retrieved from history. However, we identified the shortcomings of these traditional approaches in our previous work [26] and proposed a better approach that measures quality (and prioritize) of test cases using their similarity to the previously failing tests cases. Our similarity-based metrics also use the historical data; more specifically, it uses failing execution traces from the history. In this study, we have proposed to build the regression model using our proposed similarity-based and other traditional metrics.

In the fault prediction domain, a number of metrics and approaches have been used in predicting faults in source code. In a large survey [32], Radjenović *et al.* categorized the fault prediction metrics based on size, complexity, OO metrics and process metrics. According to their empirical study, process metrics were found most successful in predicting post-release defects compared to the traditional source code metrics (e.g., LOC and complexity metrics) and OO metrics (e.g., CK metrics [6]). Process metrics are usually extracted from the combination of source code and repository. Some of the process metrics are number of revisions, bug fixes, refactoring, code, delta, code churn (i.e., sum of added and deleted lines of code over all revisions) etc.

Nagappan *et al.* proposed a set of 9 test quantification and complexity metrics and Object-Orientation (OO) metrics to evaluate Junit test cases in terms of early estimation of software defects [24]. Test quantification metrics evaluate the test cases by the amount of the tests (e.g., number of assertions or LOC) written to check the program thoroughly. We have also used the size of tests (i.e., LOC) and coverage (i.e., method coverage) for our comparison.

In another work [23], Nagappan *et al.* combined the complexity metrics using a regression model in order to predict the post-release defects. They found that there is no single metric that works good in different projects, in terms of detecting post release defects. Therefore, they combined the metrics using a prediction model to predict post-release defects. They also mentioned about the *multicollinearity* problem while combining several metrics and they applied principal component analysis (PCA) to overcome the problem. In our study, we have also dealt with the *multicollinearity* problem, however, we have applied different approach (considered Variance Inflation Factor (VIF)) of the model is below 2.5) as suggested by Shihab *et al.* [5, 35]. We have also considered another

findings from [23], which is predictors should be obtained from the same or similar projects. Therefore, we have used different quality measures of a project to build the regression model for the same project. Similar to [23], we have also proposed to combine several metrics using a regression model, however, we have used the model to predict the failing probability of test cases and prioritized the tests based on higher failing probability value.

6 CONCLUSION

Test case prioritization ranks the tests based on some quality measures to detect the potential faults earlier. In this paper, we have studied the prioritization effectiveness using some individual quality metrics. We have also prioritized the test cases based on their failure probability using a regression model that combines the individual metrics. We have conducted a large empirical study (247 versions from five real-world java projects with real faults) and compared the ranks of fault revealing tests using different approach in the context of test case prioritization. We have not found any individual metric that is always good in prioritizing the fault reveling tests, however, the combined set of the metrics in a regression model preformed better in all the projects. We have also proposed to add our recently introduced similarity-based metrics while building the regression model, as the results show adding the similarity-based metrics provides better prioritization.

In the future, we will extend our study by adding more quality measures to prioritize test cases. In addition, feature selection techniques can be applied on the measures. Moreover, we are also interested to apply our proposed combined set of metrics as criteria to build an automatic test generation tool that can generate high-quality tests.

REFERENCES

[1] Jeff Anderson, Saeed Salem, and Hyunsook Do. 2014. Improving the effectiveness of test suite through mining historical data. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 142–151.

[2] J. Anderson, S. Salem, and H. Do. 2015. Striving for Failure: An Industrial Case Study about Test Failure Prediction. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. 49–58.

[3] Andrea Arcuri. 2010. Longer is better: On the role of test sequence length in software testing. In *Third International Conference on Software Testing, Verification and Validation (ICST), 2010*. IEEE, 469–478.

[4] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *33rd International Conference on Software Engineering (ICSE), 2011*. IEEE, 1–10.

[5] Marcelo Cataldo, Audris Mockus, Jeffrey Roberts, James D Herbsleb, et al. 2009. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering* 35, 6 (2009), 864–878.

[6] Shyam R Chidamber and Chris F Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 20, 6 (1994), 476–493.

[7] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2010. An extensive comparison of bug prediction approaches. In *7th IEEE Working Conference on Mining Software Repositories (MSR), 2010*. IEEE, 31–41.

[8] defects4j. 2014. https://github.com/rjust/defects4j. (2014). [Online; last accessed 26-Dec-2015].

[9] G. Dong and J. Pei. 2007. *Sequence Data Mining*. Springer US. https://books.google.ca/books?id=GESJmZpkePIC

[10] Eclipse-AspectJ. 2014. The AspectJ Project. https://eclipse.org/aspectj/. (2014). [Online; last accessed 26-Dec-2015].

[11] Eclipse-JDT. 2014. Eclipse java Development Tool (JDT). https://eclipse.org/jdt//. (2014). [Online; last accessed 26-Dec-2015].

[12] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1 (2007), 35–45.

[13] GLM. 2015. https://cran.r-project.org/web/packages/glm2/glm2.pdf. (2015). [Online; last accessed 26-Dec-2015].

[14] John B. Goodenough and Susan L. Gerhart. 1975. Toward a theory of test data selection. *IEEE Transactions on Software Engineering* 2 (1975), 156–173.

[15] Keith J Goulden. 2006. Effect Sizes for Research: A Broad Practical Approach. (2006).

[16] D. Gusfield. 1997. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press. https://books.google.ca/books?id=Ofw5w1yuD8kC

[17] Yu-Chi Huang, Kuan-Li Peng, and Chin-Yu Huang. 2012. A history-based cost-cognizant test case prioritization technique in regression testing. *Journal of Systems and Software* 85, 3 (2012), 626–637.

[18] Yue Jia and Mark Harman. 2011. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678.

[19] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A Database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), 2014*. ACM, 437–440.

[20] Jung-Min Kim and Adam Porter. 2002. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the 24rd International Conference on Software Engineering (ICSE), 2002*. IEEE, 119–129.

[21] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. 2007. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering (ICSE), 2007*. IEEE Computer Society, 489–498.

[22] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

[23] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. 2006. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*. ACM, 452–461.

[24] Nachiappan Nagappan, Laurie Williams, Mladen Vouk, and Jason Osborne. 2007. Using in-process testing metrics to estimate post-release field quality. In *18th International Symposium on Software Reliability (ISSRE), 2007*. IEEE, 209–214.

[25] Tanzeem Noor and Hadi Hemmati. 2015. Test case analytics: Mining test case traces to improve risk-driven testing. In *IEEE 1st International Workshop on Software Analytics (SWAN), 2015*. IEEE, 13–16.

[26] Tanzeem Bin Noor and Hadi Hemmati. 2015. A similarity-based approach for test case prioritization using historical failure data. In *26th International Symposium on Software Reliability Engineering (ISSRE), 2015*. IEEE, 58–68.

[27] M.A. P. 2008. *Foundations of Software Testing*. Pearson Education. https://books.google.ca/books?id=yU-rTcurys8C

[28] Hyuncheol Park, Hoyeon Ryu, and Jongmoon Baik. 2008. Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing. In *Second International Conference on Secure System Integration and Reliability Improvement (SSIRI), 2008*. IEEE, 39–46.

[29] M. Pezzè and M. Young. 2008. *Software testing and analysis: process, principles, and techniques*. Wiley.

[30] Predict. 2015. https://stat.ethz.ch/R-manual/R-devel/library/stats/html/predict.glm.html. (2015). [Online; last accessed 26-Dec-2015].

[31] R-Project. 2015. The R Project for Statistical Computing. http://www.r-project.org/. (2015). [Online; last accessed 26-Dec-2015].

[32] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. 2013. Software fault prediction metrics: A systematic literature review. *Information and Software Technology* 55, 8 (2013), 1397–1418.

[33] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test case prioritization: An empirical study. In *Proceedings of IEEE International Conference on Software Maintenance, 1999.(ICSM'99)*. IEEE, 179–188.

[34] R:Ranks. 2014. R:Sample Ranks. https://stat.ethz.ch/R-manual/R-devel/library/base/html/rank.html. (2014). [Online; last accessed 26-Dec-2015].

[35] Emad Shihab, Audris Mockus, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2011. High-impact defects: a study of breakage and surprise defects. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 300–310.

[36] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[37] Qing Xie and Atif M Memon. 2006. Studying the characteristics of a "Good" GUI test suite. In *17th International Symposium on Software Reliability Engineering (ISSRE), 2006*. IEEE, 159–168.

[38] Shin Yoo, Robert Nilsson, and Mark Harman. 2011. Faster fault finding at Google using multi objective regression test optimisation. In *8th European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'11), Szeged, Hungary*.

[39] Hong Zhu, Patrick AV Hall, and John HR May. 1997. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)* 29, 4 (1997), 366–427.

[40] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting defects for eclipse. In *International Workshop on Predictor Models in Software Engineering, PROMISE'07: ICSE Workshops 2007*. IEEE, 9–9.

Clustering Dycom

An Online Cross-Company Software Effort Estimation Study

Leandro L. Minku
Department of Informatics, University of Leicester
University Road
Leicester LE1 7RH, UK
leandro.minku@leicester.ac.uk

Siqing Hou
Computer, Electrical and Mathematical Sciences and Engineering Division, King Abdullah University of Science and Technology
Thuwal 23955-6900, Saudi Arabia
siqing.hou@kaust.edu.sa

ABSTRACT

Background: Software Effort Estimation (SEE) can be formulated as an online learning problem, where new projects are completed over time and may become available for training. In this scenario, a Cross-Company (CC) SEE approach called Dycom can drastically reduce the number of Within-Company (WC) projects needed for training, saving the high cost of collecting such training projects. However, Dycom relies on splitting CC projects into different subsets in order to create its CC models. Such splitting can have a significant impact on Dycom's predictive performance.
Aims: This paper investigates whether clustering methods can be used to help finding good CC splits for Dycom.
Method: Dycom is extended to use clustering methods for creating the CC subsets. Three different clustering methods are investigated, namely Hierarchical Clustering, K-Means, and Expectation-Maximisation. Clustering Dycom is compared against the original Dycom with CC subsets of different sizes, based on four SEE databases. A baseline WC model is also included in the analysis.
Results: Clustering Dycom with K-Means can potentially help to split the CC projects, managing to achieve similar or better predictive performance than Dycom. However, K-Means still requires the number of CC subsets to be pre-defined, and a poor choice can negatively affect predictive performance. EM enables Dycom to automatically set the number of CC subsets while still maintaining or improving predictive performance with respect to the baseline WC model. Clustering Dycom with Hierarchical Clustering did not offer significant advantage in terms of predictive performance.
Conclusion: Clustering methods can be an effective way to automatically generate Dycom's CC subsets.

CCS CONCEPTS

• **Software and its engineering → Software creation and management**; • **Computing methodologies → Supervised learning by regression**; **Transfer learning**; **Lifelong machine learning**; **Online learning settings**; **Ensemble methods**;

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PROMISE , November 8, 2017, Toronto, Canada
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5305-2/17/11…$15.00
https://doi.org/10.1145/3127005.3127007

KEYWORDS

Software effort estimation, cross-company learning, concept drift, online learning, ensembles

ACM Reference format:
Leandro L. Minku and Siqing Hou. 2017. Clustering Dycom. In *Proceedings of PROMISE , Toronto, Canada, November 8, 2017,* 10 pages.
https://doi.org/10.1145/3127005.3127007

1 INTRODUCTION

Software Effort Estimation (SEE) is the process of estimating the effort required to develop a software project. The use of machine learning approaches for creating SEE models based on data describing completed projects has been studied for many years [5, 7, 15]. The process of creating such SEE models faces several challenges, such as the heterogeneity of software projects and the high cost associated to collecting Within-Company (WC) projects for training SEE models. These challenges can hinder the resulting SEE model's predictive performance.

However, SEE models could form useful tools to help experts with performing and/or re-thinking their estimations, which in turn can be used to inform many important project decisions, such as project bidding, requirements selection, task allocation, etc. Therefore, many studies have been attempting to tackle such challenges in order to improve the predictive performance of SEE models and facilitate their use. In particular, several studies have investigated the use of Cross-Company (CC) projects to reduce the cost of collecting WC projects for training SEE models [17, 26, 36].

The different processes and environments underlying different companies render CC projects potentially heterogeneous with respect to the projects being estimated [30], requiring solutions to tackle heterogeneity. Therefore, several studies proposed ways to tackle heterogeneity in CC SEE [20, 22, 26, 35, 36]. In particular, the approach Dycom [35] managed to drastically reduce the number of WC projects used for training while maintaining or slightly improving predictive performance in comparison with WC models. Dycom is the only approach able achieve that while treating the online learning nature of the SEE problem, i.e., the fact that new projects arrive over time and that SEE models must be able to adapt to changes that could otherwise affect their suitability.

In order to use CC projects for SEE, Dycom splits the CC projects into different subsets according to their productivity. Specifically, pre-defined productivity thresholds are used to decide the number of CC subsets and their composing projects. As each subset contains projects within the same productivity range, each subset is expected

to contain projects that are more homogeneous to each other. These subsets are thus used to create different CC SEE models. Mapping functions are then learned to dynamically map the estimations given by the CC models to estimations reflecting the current context (in particular, the relationship between project input attributes and effort) of a given company. These mapping functions are learned by using a small number of WC projects received over time.

However, the CC splits can have a significant impact on Dycom's predictive performance [31], and it is unclear how to best split the CC projects into different subsets. This paper aims at investigating whether clustering methods can be used to help finding good CC splits for Dycom. The following research questions are answered:

- RQ1 Can clustering methods help to improve Dycom's predictive performance? Which ones?
- RQ2 Can clustering methods facilitate the creation of CC splits by automatically deciding not only the content, but also the number of CC subsets? Which ones?

In order to answer these questions, Dycom is extended to use clustering methods for splitting the CC projects. The new approach is called *Clustering Dycom*. Three different clustering methods are investigated, namely Hierarchical Clustering, K-Means and Expectation-Maximisation (EM). The analysis reveals that K-Means can potentially help to split the CC projects. Its best and worst case predictive performances were similar or better than those obtained by the original Dycom (RQ1). However, K-Means requires the number of CC subsets to be pre-defined. A poor choice can cause the predictive performance to become worse than that of a baseline WC model. EM enables Dycom to automatically set the number of CC splits, while still maintaining or improving predictive performance with respect to the baseline WC model (RQ2). Clustering Dycom with Hierarchical Clustering did not offer significant advantage in terms of predictive performance over the original Dycom.

This paper is further organised as follows. Section 2 explains related work. Section 3 explains the original Dycom. Section 4 explains how to use Dycom with clustering methods. Section 5 explains the databases used in the study. Section 6 explains the experimental setup used to answer the RQs. Section 9 discusses threats to validity. Sections 7 and 8 present the analysis to answer RQ1 and RQ2, respectively. Section 10 presents the conclusions and future work.

2 RELATED WORK

This section discusses related work on CC SEE and on tackling heterogeneity in SEE. Many studies have investigated the predictive performance of CC versus WC SEE models. A systematic literature review published by Kitchenham et al. [17] found ten studies, seven of which were independent. Three of these concluded that the predictive performance of CC and WC models was similar, whereas the other four concluded that the predictive performance of CC models was worse. McDonell and Shepperd [24] also performed a systematic literature review to investigate whether CC models have similar predictive performance to WC models, and found no strong evidence in support of either type of model. Ordinary least squares regression, stepwise regression, robust regression, regression trees and k-nearest neighbours were among the machine learning approaches investigated.

Several of these studies used CC projects in an attempt to completely eliminate the need for WC projects [17, 24] . They compared WC models against models created only with projects from other companies (e.g., [6, 44]). Some other studies used both CC and WC projects in at least part of the procedure of building SEE models [14, 16, 23]. For example, Lefley and Shepperd [23] trained SEE models with both CC and WC projects, in an attempt to overcome the small number of WC projects [23].

Both systematic reviews mention that the level of heterogeneity of the single-company being estimated may vary and influence the WC models' performance. This has influenced more recent studies that use local learning approaches to deal with the heterogeneity of WC and CC projects. For example, Kocaguneli et al. [20] used a tree-based filtering mechanism, obtaining very encouraging results. Models trained solely with different types of projects (or projects from different centres of a company) from the ones being estimated obtained overall similar performance to models trained on projects of the same type (or from the same centre). Turhan and Mendes [42] investigated the use of a filtering mechanism based on k-nearest neighbours, obtaining very encouraging results in the area of web effort estimation. Specifically, filtered stepwise regression CC models achieved similar performance to WC models in seven out of eight data sets, and worse performance in only one data set.

Menzies et al. [25] proposed to cluster WC+CC projects using a method called WHERE, which splits projects according to the input attributes of highest variability. Prediction rules are then created for each cluster based on a method called WHICH. Given a certain cluster, its neighbouring cluster with the lowest required efforts was referred to as the *envied* cluster. When making predictions for WC projects belonging to a cluster, rules created using only the CC projects from the envied cluster were better than rules created using only the WC projects from the envied cluster. So, the authors recommended to cluster WC+CC projects, but to learn rules using solely the CC projects from the envied cluster. This study was in the context of both SEE and software defect prediction, but this particular conclusion was based only on the defect prediction data.

Other studies also investigated the use of clustering methods to tackle heterogeneity. For example, Huang et al. [12] investigated K-Means and Scheffe's method to cluster CC projects based on their input attributes. An ordinary least squares model was created for each cluster. When estimating a new project, the cluster to which this project belongs is determined and its corresponding SEE model is used for performing the estimation. The predictive performance obtained using clustering was similar to that obtained without clustering. Gallego et al. [10] partitioned CC projects into different subsets based on certain input attributes of interest. Each of the partitions was then further clustered using EM. Linear or exponential regression models were created for each cluster. Predictions were made in a similar way to Huang et al. [12]'s method. The authors concluded that clustering can improve predictive performance, even though no statistical tests were used in this comparison. These methods were evaluated for making predictions for CC projects, i.e., no WC data set was used in these studies.

It is important to note that, even though CC/WC projects could be considered as potentially more/less heterogeneous with respect to the projects being estimated, WC projects may also be heterogeneous. Therefore, the terms CC/WC should not be considered as

synonyms of heterogeneous/homogeneous [30], and the possible heterogeneity of WC projects should be tackled. Menzies et al. [21] and Minku and Yao [34] investigated the use of tree-based SEE models to tackle heterogeneity in general, i.e., not restricted to CC projects. Other local approaches such as k-nearest neighbours [2, 40] could also be seen as tackling heterogeneity.

The studies above did not take into account the fact that SEE is an online learning problem, where new projects need to be predicted over time, and changes suffered by the company may affect the quality of existing SEE models [33, 36]. With that in mind, Dynamic Cross-company Learning (DCL) [33, 36] creates an ensemble of WC and CC models and dynamically identifies which of them is currently beneficial for SEE. The beneficial models are emphasised when the ensemble is asked to estimate a new WC project. This approach managed to improve predictive performance in comparison with a WC model. However, DCL can only benefit from CC projects when they match the current WC context reasonably well. It still requires a fair amount of WC projects to achieve good performance during the periods when the CC models are not beneficial.

Kocaguneli et al.[22] investigated a tree-based filtering mechanism called TEAK [21] to tackle heterogeneity. This mechanism creates trees to represent training projects and provide effort estimations. CC or WC training projects corresponding to sub-trees of high variance are assumed to be detrimental and filtered out. Besides investigating TEAK for CC web effort estimation, Kocaguneli et al. [22] also investigated its ability to transfer knowledge from the past to the present in conventional WC SEE. The approach managed to obtain similar performance when using only training projects from the same time period as the project being estimated, and when using a mix of training projects from the same and different time periods. However, similar to DCL [33, 36], this approach still relies on a good number of projects from the same time period to be available for the process of generating the SEE model.

The approach Dycom [35] has been proposed to overcome this limitation. Similar to DCL, it maintains an ensemble of WC and CC models. However, instead of just identifying which past WC or CC models are more beneficial, it maps the estimations given by the CC models to the WC context. In this way, it can significantly reduce the number of WC projects needed for training while maintaining or slightly improving predictive performance in comparison with a WC model. However, Dycom requires CC projects to be split into different subsets to create its CC models, and the choice of CC split can significantly affect Dycom's predictive performance. Therefore, it would be desirable to have a method to facilitate the splitting process. As explained in section 1, this paper aims to investigate whether clustering methods can help with that. Section 3 explains Dycom in more detail.

3 DYCOM

Dycom (a.k.a. Dynamic Cross-company Mapped Model Learning) is an SEE online learning approach based on the observation that there is a relationship between the SEE context of a certain company and other companies. Minku and Yao [35] formalise the relationship between two companies C_A and C_B as follows:

$$f_A(\mathbf{x}) = g_{BA}(f_B(\mathbf{x})) \quad (1)$$

where C_A is the company in which we are interested; C_B is another company (or a subset of this other company); f_A and f_B are the true functions providing C_A's and C_B's required efforts, respectively; g_{BA} is a function that maps the effort from C_B's context to C_A's context; and $\mathbf{x} = [x_1, x_2, \cdots, x_n]$ are the input attributes describing a software project. The functions f_A, f_B and g_{BA} can be of any type. An illustrative example where $f_A(\mathbf{x}) = g_{BA}(f_B(\mathbf{x})) = 1.2 \cdot f_B(\mathbf{x})$ can be found in [35].

Given equation 1, the task of learning an SEE model for a company C_A can involve the task of learning the relationship between C_A and other companies. Therefore, Dycom uses CC training projects to learn one or more CC models, and uses a very limited number of WC training projects to learn a function that maps the estimations given by each CC model to estimations in the WC context. It is hoped that the task of learning the mapping functions is less difficult than the task of learning a whole WC model based solely on the WC training projects, as this would considerably reduce the amount of WC projects required for learning. As summarised by Minku et al. [32], Dycom works as follows.

CC training projects: Dycom uses CC training projects that are available beforehand. In [35], they were split into M different training subsets C_{Bi}, $1 \leq i \leq M$, of similar size based on productivity thresholds. For example, if the productivity of the CC training projects in terms of effort (in person-months) divided by size varies from 3 to 38, one may wish to separate these projects into three subsets, one containing projects with productivity below 8, one containing projects with productivity between 8 and 16, and one for projects with productivity higher than 16, if these productivity thresholds lead to subsets containing a similar number of projects. Each subset C_{Bi} is considered as a separate CC training set. The reason for splitting CC projects will be explained later in the paragraph on mapping functions. Note that we use the term CC loosely herein. For example, projects from different departments within the same company could be considered as CC projects if such departments employ largely different practices.

CC SEE models: Each of the M CC training sets is used to create a different CC model $\hat{f}_{Bi}$ ($1 \leq i \leq M$).

WC training projects: Dycom considers that the WC projects arrive in order of completion, i.e., online. In particular, it considers that one WC project arrives at each *time step*. WC projects that arrive at every p ($p > 1$) time steps contain both information on their input attributes and actual effort. All remaining WC projects contain only the information on the input attributes, whereas their actual effort, which is more difficult and costly to collect [19], is missing. So, even though an effort estimation is required for all WC projects, only a few of them (those arriving at time steps multiple of p) can be used as training projects.

Mapping functions: Whenever a new WC training project arrives, each model $\hat{f}_{Bi}$ is asked to perform an SEE. Each SEE is then used to create a mapping training example $(\hat{f}_{Bi}(\mathbf{x}), y)$. A mapping function $\hat{g}_{BiA}$ that receives estimations $\hat{f}_{Bi}(\mathbf{x})$ in the context of C_{Bi} as input and maps them to estimations in the context of C_A is trained with the mapping training example. Dycom considers that the relationship formalised in equation 1 can be modelled reasonably well by linear functions of the format $\hat{g}_{BiA}(\hat{f}_{Bi}(\mathbf{x})) = \hat{f}_{Bi}(\mathbf{x}) \cdot b_i$ when different subsets containing relatively more similar CC training

projects are considered separately. This is the reason to split CC training projects into different subsets.

Learning a function of the format $\hat{g}_{BiA}(\hat{f}_{Bi}(\mathbf{x})) = \hat{f}_{Bi}(\mathbf{x}) \cdot b_i$ is equivalent to learning the factor b_i. This is done using equation 2:

$$b_i = \begin{cases} 1, & \text{if no mapping training example has been received yet} \\ \dfrac{y}{\hat{f}_{Bi}(\mathbf{x})}, & \text{if } (\hat{f}_{Bi}(\mathbf{x}), y) \text{ is the first mapping training example} \\ lr \cdot \dfrac{y}{\hat{f}_{Bi}(\mathbf{x})} + (1 - lr) \cdot b_i, & \text{otherwise.} \end{cases} \quad (2)$$

where $(\hat{f}_{Bi}(\mathbf{x}), y)$ is the mapping training example being learnt, lr ($0 < lr < 1$) is a pre-defined smoothing factor and the factor b_i in the right side of the equation is the previous value of b_i. This equation enables the mapping function to dynamically adapt to potential changes affecting software effort over time. For a more detailed explanation of this equation, we refer the reader to [35].

WC SEE model: Whenever a new WC training project arrives, it is used to train a WC model $\hat{f}_{W_A}$. This model is not expected to perform very well, because it will be trained on a limited number of projects. However, its effort estimations may be helpful when used in an ensemble together with the mapped estimations, given that ensembles have been showing to improve SEE considerably [33, 34, 36]. It is also worth noting that, despite being potentially more homogeneus than CC projects, the WC projects could possibly also present a significant level of heterogeneity. In order to tackle this heterogeneity, it is recommended to use Dycom with WC models that are local approaches, such as decision trees [34].

Dycom's SEEs: Both the mapped models $\hat{g}_{BiA}(\hat{f}_{Bi})$ and the WC model $\hat{f}_{W_A}$ can provide an SEE in the WC context when required. The SEE given by Dycom is the weighted average of these $M + 1$ estimations:

$$\hat{f}_A(\mathbf{x}) = \left[\sum_{i=1}^{M} w_{Bi} \cdot \hat{g}_{BiA}(\hat{f}_{Bi}(\mathbf{x})) \right] + w_{W_A} \hat{f}_{W_A}(\mathbf{x}), \quad (3)$$

where the weights w_{Bi} and w_{W_A} represent how much we trust each of the models, are positive and sum to one. So, Dycom uses an ensemble of mapped and WC SEE models.

Weights: The weights are initialised so that they have the same value for all models being used in the ensemble and are updated as as follows: whenever a new WC training project is made available, the model which provided the lowest absolute error is considered to be the *winner* and the others are the *losers*. The losers have their weights multiplied by a pre-defined parameter β ($0 < \beta \leq 1$), and then all weights are normalised in order to sum up to one.

Dycom's pseudocode can be found in [35], and is omitted from here due to space limitations.

4 CLUSTERING DYCOM

CC projects could potentially be split into different CC subsets based on clustering algorithms, rather than pre-defined productivity thresholds. For that, each cluster of CC projects can be considered as a CC subset. We will refer to this approach as *Clustering Dycom*. Clustering Dycom offers potential advantages over pre-defined productivity thresholds. Clustering algorithms could automatically find out which CC projects are most similar to each other and group them together. In this way, project managers using Clustering Dycom would not need to analyse the CC projects and their productivity in order to determine good thresholds to be used. A good clustering algorithm may be able to further boost Dycom's predictive performance, or avoid low predictive performance resulting from poor thresholds. Moreover, certain clustering algorithms may be able to automatically decide the number of CC subsets to be used. This would further facilitate the use of Dycom, as the number of CC subsets would not need to be chosen beforehand.

Different features can be used to describe training projects for clustering. We investigate three different sets of clustering features:

- Productivity, measured by effort divided by size. We investigate productivity because the original Dycom achieved good results when splitting CC projects based on productivity thresholds.
- Size and effort. Together, size and effort represent the productivity associated to a project. However, two projects with similar productivities could still have very different sizes. Given that size is an important factor for estimating effort [36], we investigate the use of clustering based on the size and effort of projects.
- All project input and output attributes. We have investigated the use of all attributes of a project to check whether a more detailed characterisation of projects could lead to better results than using only size and effort or productivity.

Please note that it is ok to use the effort as (part of) a feature for clustering, because (1) clustering is applied only to the CC *training* projects, and (2) Dycom does not require to determine the cluster to which a project being estimated belongs.

Three different clustering methods were investigated: Hierarchical Clustering, K-Means, and EM. These three clustering methods, as well as the reasons for choosing them for the investigation, are explained below.

4.1 Hierarchical Clustering

Hierarchical clustering methods [37] build a hierarchy of clusters by putting together projects that are similar to each other. In this work, we have used an agglomerative hierarchical method. This method uses a bottom-up approach in which each project is considered to be a cluster in the lowest level of the hierarchy. Then, at each step of the algorithm, the two clusters that are closest to each other are merged together. In the end of the procedure, a single cluster containing all the projects is created. In order to use the clusters provided by this method, the level of the hierarchy corresponding to the desired number M of CC subsets is chosen.

Different ways to measure the similarity between clusters can be used. In this work, the distance between two clusters is defined as the Manhattan distance between the two most distant projects, one from each cluster. The Manhattan distance between two projects $\mathbf{x}_1$ and $\mathbf{x}_2$ is defined as $\sum_{i=1}^{F} |x_{1i} - x_{2i}|$, where F is the number of clustering features, and x_{1i} and x_{2i} represent the ith feature of projects $\mathbf{x}_1$ and $\mathbf{x}_2$, respectively. If a given feature i is nominal, $|x_{1i} - x_{2i}|$ is set to 1 if x_{1i} and x_{2i} have different values, and to 0 otherwise. As the Manhattan distance is affected by the scale of the features, all numeric features are normalised to be within $[0, 1]$.

This clustering method has been chosen because hierarchical SEE models have obtained promising results for tackling heterogeneity [21, 34]. It also has the advantage of being a deterministic method, i.e., it will always retrieve the same clusters when the same projects and features are used.

4.2 K-Means

K-Means [37] is an iterative clustering method that tries to minimise the distance of the projects to their cluster centres. It first randomly assigns projects to a pre-defined number k of clusters, where k equals to the desired number M of CC subsets to be generated. Then, in each iteration, projects are reassigned to the clusters with the closest centres, and the clusters centres are re-computed. This iterative procedure is typically halted when projects stop moving between clusters, or when a maximum number of iterations is reached. Similar to hierarchical clustering, Manhattan distance has been used as the distance measure.

This clustering method has been chosen because it is one of the most popular and well known clustering methods in the literature.

4.3 EM

EM [4] is a density-based clustering method. It assumes that the projects belonging to each cluster are drawn from a specific distribution. Therefore, this method tries to identify the clusters and their probability distributions. Given a number of clusters M, EM first randomly assigns random values for the parameters θ_j of the probability distributions associated to each cluster C_j. It then performs two steps: expectation and maximisation. In the expectation step, the posterior probability $p(C_j|\mathbf{x})$ that a given project $\mathbf{x}$ belongs to a given cluster C_j is estimated based on the current parameters of the probability distributions. In the maximisation step, the parameters that maximise the log likelihood of the projects $ln\ p(\mathbf{x}|\theta)$ are calculated (note that $p(C_j|\mathbf{x})$ is used to compute that) and used to replace the previous parameters. The algorithm iterates through the expectation and maximisation steps until a stopping criterion is met. This could be when a pre-defined maximum number of iterations is reached, or when the the changes in the log likelihood $ln\ p(\mathbf{x}|\theta)$ become smaller than a threshold. The distributions are assumed to be Gaussian for numeric features, whereas discrete estimators (i.e., frequency counts) are used to characterise the distribution of nominal features.

This algorithm can automatically determine the number of clusters by using 10-fold cross-validation on the CC training projects being clustered. This is done through the following procedure [11]:

(1) The number of clusters (which is equal to the number M of CC subsets) is set to 1.
(2) The CC training projects are divided randomly into 10 folds.
(3) EM is performed 10 times using the 10 folds in the usual cross-validation way. Specifically, in each time, a different fold is used to compute the log likelihood and the remaining folds are used for clustering.
(4) The log likelihood is averaged over all 10 results.
(5) The process above is repeated by increasing the number of clusters by 1 while improvements in the log likelihood are observed. If no improvements are observed, the procedure stops and retrieves the number of clusters with the maximum log likelihood so far.

If there are less than 10 projects, the number of folds is set to the number of projects in order to perform cross-validation.

This algorithm has been chosen because of its potentially useful procedure to automatically determine the number of clusters.

5 DATABASES

The databases used in the experiments to answer the RQs outlined in section 1 are the same as the ones used in Dycom's original paper [35]. Part of their descriptions is transcribed here to make this paper more self-contained. The key difference between the use of these databases in Dycom's original [35] and current paper is that the former fixed the CC subsets based on specific thresholds. The current paper will investigate different CC splits, as explained in sections 4 and 6.

Five different databases were used: KitchenMax, CocNasaCoc81, ISBSG2000, ISBSG2001 and ISBSG. These include both data sets derived from the former PROMISE Repository [27] (now SEACRAFT Repository [28]) and the ISBSG Repository [13]. Each database contains a WC data set and a CC data set.

5.1 KitchenMax

The database KitchenMax is composed of Kitchenham[1] and Maxwell[2], which are two SEE data sets available from the SEACRAFT Repository [28]. Kitchenham's detailed description can be found in [18]. It comprises 145 maintenance and development projects undertaken between 1994 and 1998 by a single software development company. Maxwell's detailed description can be found in [38]. It contains 62 projects from one of the biggest commercial banks in Finland, covering the years 1985 to 1993 and both in-house and outsourced development. In order to make these data sets compatible, a single input attribute (functional size) was used. Still, Maxwell uses functional size, whereas Kitchenham uses adjusted functional size. An appropriate mapping function should be able to overcome this problem. There were no functional size attribute values missing. The output attribute is the effort in person-hours.

Kitchenham was considered as the WC data, and was sorted according to the actual start date plus the duration. This sorting corresponds to the exact completion order of the projects. Maxwell's projects were considered as the CC projects. Figure 1a shows the productivity of the CC projects in terms of effort divided by size, which is used by the original Dycom to create the CC subsets.

5.2 CocNasaCoc81

The database CocNasaCoc81 is composed of Cocomo Nasa and Cocomo 81, which are two SEE data sets available from the former PROMISE Repository [27]. Cocomo Nasa contains 60 Nasa projects from 1980s-1990s and Cocomo 81 consists of the 63 projects analysed by Boehm to develop the software cost estimation model COCOMO [5] first published in 1981. Both data sets contain 16 input attributes (15 cost drivers [5] and number of lines of code) and one output attribute (software effort in person-months). Cocomo 81 contains an additional input attribute (development type) not

[1] https://doi.org/10.5281/zenodo.268457
[2] https://doi.org/10.5281/zenodo.268461

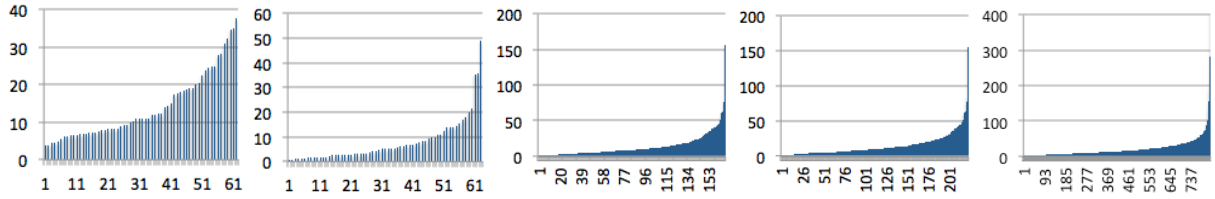


Figure 1: CC projects' productivities (effort divided by size is shown in the y-axis). Projects are sorted from the most productive to the least productive.

present in Cocomo Nasa, which was thus removed. These data sets contain no missing values.

Cocomo Nasa's projects were considered as the WC and Cocomo 81's as the CC projects. The productivity of the CC projects is shown in figure 1b. Cocomo Nasa provides no information on whether the projects are sorted in chronological order. The original order of the Cocomo Nasa projects was preserved in order to simulate a given WC projects chronology scenario. Even though this may not be the true chronological order, it is still useful to evaluate whether approaches are able to make use of mapped CC models when/if they are beneficial. Therefore, this is adequate for the purpose of answering the RQs outlined in section 1.

5.3 ISBSG Databases

Three SEE databases were derived from ISBSG [13] Release 10, which contains software project information from several companies. Information on which projects belong to a single company for composing a WC data set has been provided to us upon request. The databases are:

- ISBSG2000 – 119 WC projects implemented after the year 2000 and 168 CC projects implemented up to the end of 2000.
- ISBSG2001 – 69 WC projects implemented after the year 2001 and 224 CC projects implemented up to the end of 2001.
- ISBSG – no date restriction to the 184 WC and 826 CC projects, meaning that CC projects with implementation date more recent than WC projects are allowed. This database simulates the case where some other companies are likely to be be more evolved than the single company analysed.

Information on how these databases were preprocessed can be found in [33] and is not included here due to space limitations. Four input attributes (development type, language type, development platform and functional size) and one output attribute (software effort in person-hours) were used. The WC projects were sorted based on the implementation date to compose a stream of incoming projects. Figures 1c, 1d and 1e show the productivities of the CC projects of these databases.

6 EXPERIMENTAL SETUP

In order to answer the RQs outlined in section 1, Clustering Dycom was compared against the original Dycom with different CC splits. A baseline WC approach was also used to complement the analysis. The experiments followed the experimental setup from the paper proposing Dycom [35]. In particular, Regression Trees (RTs) were used as Dycom's base learners. As explained in [35], RTs were chosen because they have shown competitive performance in comparison with several other SEE approaches [34]. They are also local approaches, which can help dealing with the heterogeneity within each data set. The approaches used to answer the RQs are explained over the next three paragraphs.

Dycom: Dycom was used with RTs as the CC and WC models with $p = 10$, i.e., a new WC training project was provided only at every 10 time steps. So, Dycom uses only 10% of the WC projects for training, as in [35]. Whenever a new WC training project was made available, the WC RT used by Dycom was rebuilt from scratch using all WC training projects so far. Minku and Yao [35] suggested that the CC splits are created in such a way that different CC subsets have similar size. Therefore, different CC splits were created in the following way [31]. Given a desired number M of CC subsets, productivity thresholds were set to the values that lead to M CC subsets with the same number of projects. Least productive CC subsets may have one project less than the most productive CC subsets if the total number of CC projects is not a multiple of M. The experiments used the RT implementation *REPTree* provided by WEKA [11], where splits are created so as to minimise the variance of the output attributes of the training projects in the nodes.

Clustering Dycom: In order to provide a fair comparison with Dycom, Clustering Dycom also used RTs as the CC and WC models with $p = 10$. The experiments used the WEKA [11] implementation of the clustering methods.

RT: RTs were created to reflect WC online learning, forming a baseline for the analysis, as in [35]. Whenever a new WC training project was provided, the current RT was discarded and a new RT was trained on all projects so far. This approach considers that all WC completed projects have their true effort known, i.e., every time step receives a WC training project. Therefore, it uses ten times more WC training projects than Dycom and Clustering Dycom.

The parameters of all approaches were set to the same values as in previous work [35], except for the number M of CC subsets, and the clustering parameters. Dycom and Clustering Dycom were run with six different values $M = \{1, \cdots, 6\}$ in order to answer RQ1 and RQ2, except for Clustering Dycom with EM, which automatically determines an appropriate value for M. The clustering parameters, which were not needed in [35], were set so as to run the methods described in section 4. The stopping criteria for K-Means and EM were set to the same default value of 500 iterations, and EM's minimum standard deviation was set to the default value of 1.0E-6. The use of default values avoids giving an unfair advantage to Clustering Dycom over Dycom in the experiments. Future work

will investigate whether Clustering Dycom's performance could be improved further by fine tuning the clustering parameters. Dycom's and RT's parameters from [35] were the following. Dycom's parameter β was set to the default value of 0.5. Dycom's parameter lr was set to 0.1 after some preliminary investigation with 0.1 and 0.05. The parameters used with each RT were the ones more likely to obtain good results in previous work [34]: minimum total weight of 1 for the instances in a leaf, and minimum proportion of the variance on all the data that need to be present at a node in order for splitting to be performed 0.0001.

As in [35], at each time step, effort estimations given for the next ten WC projects were performed and evaluated. None of these projects were used for training before being used for evaluation. In this work, Mean Absolute Error (MAE) and Standardised Accuracy (SA) were used to evaluate the estimations. The equations to calculate MAE and SA are the following: $MAE = \frac{1}{T}\sum_{i=1}^{T}|\hat{y}_i - y_i|$, and $SA = (1 - MAE/MAE_{rguess}) \cdot 100$, where T is the number of projects used for evaluating the performance, y_i is the actual effort for the project i, $\hat{y}_i$ is the estimated effort for project i, MAE is the MAE of the approach being evaluated, and MAE_{rguess} is the MAE of 1000 runs of random guess. Random guess is defined as sampling uniformly at random the true effort over all the WC projects received up to the current time step. Therefore, it has access to the same number of WC training projects as RT, which is 10 times higher than the number of WC projects used by Dycom and Clustering Dycom.

MAE has been recommended by Shepperd and McDonell [39] for being unbiased towards under or overestimations. SA is an unbiased measure that allows for interpretability – it is viewed as the ratio of how much better an approach is than random guess [39]. So, it can be used to give a better idea of the magnitude of the differences in performance. Mean Magnitude of the Relative Error (MMRE) and percentage of predictions within 25% of the actual value (PRED(25)) were not used because they are biased towards underestimations [39] and could lead to misleading conclusions. The comparisons involving MAE are supported by Wilcoxon Sign-Rank tests with Holm-Bonferroni corrections at the overall level of significance of 0.05, and A12 [43], which is one of the measures of effect size recommended by Arcuri and Briand [3]. Both Wilcoxon and A12 are non-parametric. As suggested by Vargha and Delaney [43], A12 of 0.50 indicates no difference, up to 0.56 indicates a small difference, up to 0.64 indicates medium difference and over 0.71 indicates large difference. Wilcoxon and A12 have not been used for comparing SA because SA is an interpretable equivalent of MAE.

Dycom and Clustering Dycom with Hierarchical Clustering were run a single time for each data set, as they are deterministic when using deterministic RTs. Clustering Dycom with K-Means and EM were run 30 times, and the MAE obtained at each time step was averaged across these 30 runs.

7 THE IMPACT OF CLUSTERING DYCOM ON PREDICTIVE PERFORMANCE

This section investigates if the use of clustering methods could help to create CC splits that lead to better predictive performance (RQ1). For that, the predictive performance obtained by Clustering Dycom with the 3 different clustering methods explained in section 4 is analysed and compared with Dycom and the baseline RT.

Each clustering method was analysed when using three different sets of project features (productivity, size and effort, and all attributes), as explained in section 4. The MAEs obtained when using size and effort, and when using all attributes, were worse than those obtained by using productivity in most cases. This further supports the fact that Dycom achieved good results when splitting CC projects based on productivity in [35]. Therefore, this section concentrates on the analysis of Clustering Dycom based on productivity. The results obtained with size and effort, and all attributes, are omitted due to space constraints.

Table 1 shows the results achieved by productivity-based Clustering Dycom, Dycom and the baseline RT. For Clustering Dycom with Hierarchical Clustering and K-Means, and for Dycom, the results correspond to the number M^* of CC subsets which achieved the top ranked MAE. For EM, the number of CC subsets was determined automatically (see section 4.3). RT does not use CC projects.

The statistical tests found no significant difference between the MAE of Dycom and Clustering Dycom with Hierarchical Clustering for KitchenMax, CocNasaCoc81 and ISBSG2001. For ISBSG2000 and ISBSG, significant difference was found. Even though Clustering Dycom's MAE was better than Dycom's for ISBSG2000, the A12 of the difference was small (0.53). Clustering Dycom's MAE was worse than Dycom's for ISBSG and A12 was medium-large (-0.69). So, the best MAE achieved by Hierarchical Clustering is similar or worse than that of Dycom. EM's results were also similar or worse than those of Dycom, with A12 varying from small to medium.

According to the MAEs and statistical tests, Clustering Dycom with K-Means obtained better MAE than Dycom for CocNasaCoc81. A12 was medium-large (0.70) and the difference in SA was of 21.55 units. Such difference has considerably large magnitude, and is thus likely to have an effect in practice. For the other databases, no significant difference was found. Therefore, Clustering Dycom with K-Means was able to maintain or improve Dycom's MAE. This suggests that grouping together projects in terms of the distance of their productivities could potentially be a helpful alternative to the original Dycom. Therefore, we investigate the results obtained with K-Means further.

Table 2 shows the results obtained when using K-Means with $k \in \{1, \cdots, 6\}$ and RT. The MAEs and statistical tests show that the top ranked Ms always obtained significantly better MAE than the RTs. The corresponding A12s varied from medium to large and differences in SA varied from 7.07 to 33.75, being very considerable and likely to have an effect in practice. Therefore, Clustering Dycom with K-Means can drastically reduce the number of WC training projects, while significantly improving predictive performance.

However, similar to the original Dycom [31], the number of CC subsets M can significantly affect Clustering Dycom with K-Means. A poor choice of M can lead to significantly worse MAE than the top ranked M, as shown by the statistical tests from table 2. The A12s of such differences varied from small to very large. Moreover, a poor choice of M could lead to worse results than the baseline RT. For instance, Clustering Dycom with K-Means led to significantly worse MAE than RT when using $M = 3$ for CocNasaCoc81 and $M = 1$ for ISBSG (p-values of 2.99E-09 and 6.09E-12, respectively), with very large differences in SA. Therefore, using K-Means involves some risk in terms of predictive performance.

Table 1: Results obtained by productivity-based Clustering Dycom, RT, and Dycom.

Approach / M^*	Database	MAE	SA	P-value	A12
Hierar. / 6	KitchenMax	2163	38.13	6.72E-01	-0.52
Hierar. / 5	CocNasaCoc81	224	53.15	2.04E-01	0.60
Hierar. / 3	ISBSG2000	2146	50.94	3.35E-03	0.53
Hierar. / 6	ISBSG2001	2344	43.01	6.89E-01	-0.52
Hierar. / 6	ISBSG	4137	31.71	1.35E-14	-0.69
K-means / 1	KitchenMax	2176	37.75	1.13E-01	-0.51
K-means / 2	CocNasaCoc81	158	66.89	8.03E-05	0.70
K-means / 6	ISBSG2000	2094	52.12	6.69E-01	0.51
K-means / 6	ISBSG2001	2315	43.70	1.60E-01	0.51
K-means / 3	ISBSG	2826	53.36	9.34E-01	0.51
EM / 2	KitchenMax	2333	33.27	2.26E-06	-0.61
EM / 2	CocNasaCoc81	308	35.57	2.30E-02	-0.64
EM / 3	ISBSG2000	2256	48.42	5.05E-01	-0.51
EM / 4	ISBSG2001	2640	35.82	7.66E-07	-0.58
EM / 5	ISBSG	2937	51.53	2.78E-03	-0.53
RT	KitchenMax	2441	30.18	3.01E-11	0.62
RT	CocNasaCoc81	319	33.14	1.41E-01	0.52
RT	ISBSG2000	2753	37.05	1.11E-05	0.62
RT	ISBSG2001	3622	11.93	1.83E-07	0.76
RT	ISBSG	3253	46.29	6.37E-06	0.58
Dycom / 6	KitchenMax	2165	38.06	–	–
Dycom / 2	CocNasaCoc81	261	45.34	–	–
Dycom / 6	ISBSG2000	2215	49.35	–	–
Dycom / 4	ISBSG2001	2353	42.79	–	–
Dycom / 3	ISBSG	2806	53.69	–	–

M^* is the number of CC subsets with the best ranked MAE, except for EM, where M^* is the median of the number of clusters automatically set by EM across 30 runs. The p-values are the results of the Wilcoxon Sign Rank tests to compare each approach's MAE against Dycom's. P-values highlighted in yellow (light grey) represent statistically significant difference at the overall level of significance of 0.05, when using Holm-Bonferroni corrections considering the 5 comparisons made for a given approach. Positive A12 (or absolute values of negative A12) represent the probability that the corresponding approach has better (worse) MAE than Dycom.

Table 2: Results obtained by Clustering Dycom with K-Means based on productivity, and RT.

	M	MAE	SA	P-value	A12
KitchenMax	1	2176	37.75	–	–
	2	2443	30.13	8.87E-06	0.62
	3	2269	35.11	3.26E-02	0.55
	4	2278	34.85	1.89E-01	0.55
	5	2269	35.10	2.66E-01	0.54
	6	2249	35.68	5.02E-01	0.53
	RT	2441	30.18	3.50E-16	0.60
CocNasaCoc81	1	541	-13.31	3.37E-09	0.81
	2	158	66.89	–	–
	3	818	-71.13	7.56E-10	0.99
	4	675	-41.33	7.56E-10	0.99
	5	598	-25.06	7.56E-10	0.98
	6	612	-28.19	7.56E-10	0.99
	RT	319	33.14	7.16E-07	0.66
ISBSG2000	1	2789	36.24	3.95E-14	0.70
	2	2342	46.44	1.37E-02	0.55
	3	2124	51.43	7.31E-01	0.51
	4	2372	45.76	1.48E-08	0.57
	5	2201	49.67	1.69E-02	0.51
	6	2094	52.12	–	–
	RT	2753	37.05	1.06E-06	0.65
ISBSG2001	1	2435	40.80	8.43E-03	0.55
	2	2663	35.25	1.71E-06	0.61
	3	2749	33.16	7.37E-07	0.58
	4	2483	39.61	8.53E-02	0.53
	5	2353	42.79	2.58E-01	-0.50
	6	2315	43.70	–	–
	RT	3622	11.93	6.76E-09	0.77
ISBSG	1	4961	18.11	2.14E-27	0.72
	2	3695	39.01	1.83E-14	0.64
	3	2826	53.36	–	–
	4	2921	51.78	3.09E-04	0.53
	5	2916	51.87	2.18E-08	0.53
	6	4145	31.58	5.54E-27	0.70
	RT	3254	46.29	2.02E-07	0.59

The column M indicates the number of CC subsets for Clustering Dycom with K-Means, or the baseline RT approach. The top and bottom ranked MAEs for each database are highlighted in lime (light grey) and orange (dark grey), respectively. The p-values are the results of the Wilcoxon Sign Rank tests to compare the MAE of each configuration/approach against the top ranked one. P-values highlighted in yellow (light grey) represent statistically significant difference at the overall level of significance of 0.05, when using Holm-Bonferroni corrections considering the 6 comparisons made for a given database. A12 represents the probability that the corresponding approach has worse MAE than the top ranked one.

Still, the worst MAEs obtained by Clustering Dycom with K-Means were similar or better than those obtained by the original Dycom. The results of the statistical tests and A12s to support this analysis are shown in table 3. When Clustering Dycom obtained significantly better MAE than Dycom (ISBSG2001 and ISBSG), A12 was medium and the differences in SA were of 9.97 and 15.02 units, being of considerable magnitude. Therefore, even though the number of clusters can affect the MAE achieved with K-Means, Clustering Dycom with K-means could be considered as a safer method than the original Dycom in the case of a poor choice of M.

In summary, this section shows that K-Means can help to create CC splits that lead to better predictive performance, giving a positive answer to RQ1.

8 THE EFFECT OF AUTOMATICALLY DETERMINING THE NUMBER M OF CC SUBSETS

K-Means obtained encouraging results in terms of predictive performance, as shown in section 7. However, a poor choice of the number M of CC subsets could lead to worse MAE than the baseline RT. Choosing the right M for Clustering Dycom with K-Means may not be an easy task. Therefore, this section investigates whether clustering methods can facilitate the creation of the CC splits by automatically deciding not only the content, but also the number of CC subsets (RQ2).

The clustering method EM can automatically decide the number of clusters, as explained in section 4. We have seen in section 7 that the MAE obtained by EM was similar or worse than the MAE

Table 3: A12 and p-values of Wilcoxon Sign-Rank tests for comparing the MAEs obtained by Clustering Dycom with K-Means and Dycom, using the bottom ranked M.

Database	P-value	A12
KitchenMax	2.99E-01	0.52
CocNasaCoc81	4.78E-01	0.52
ISBSG2000	6.95E-01	-0.53
ISBSG2001	1.07E-07	0.57
ISBSG	2.50E-05	0.57

P-values in yellow (light grey) indicate statistically significant difference at the overall level of 0.05 with Holm-Bonferroni corrections considering the 5 comparisons made.

Table 4: A12 and p-values of Wilcoxon Sign-Rank tests for comparing the MAEs obtained by RT and Clustering Dycom with EM.

Database	P-value	A12
KitchenMax	1.31E-01	0.52
CocNasaCoc81	9.81E-01	-0.57
ISBSG2000	6.46E-05	0.62
ISBSG2001	7.61E-06	0.70
ISBSG	2.49E-04	0.55

P-values in yellow (light grey) indicate statistically significant difference at the overall level of 0.05 with Holm-Bonferroni corrections considering the 5 comparisons made.

obtained by the original Dycom with its best number of CC subsets. However, this clustering method may still be a good option if it offers less risk than K-Means, i.e., if its MAE is no worse than that of the baseline RT. That is because this would mean that Clustering Dycom with EM can drastically reduce the number of WC training projects needed to be collected, while at least maintaining the predictive performance obtained by a WC model.

Table 4 shows the results of the comparison between the MAE of Clustering Dycom with EM and the MAE of the baseline RT. No significant difference was found for KitchenMax and CocNasaCoc81. For the other databases, Clustering Dycom obtained significantly better MAE than RT. A12s were 0.62 (medium), 0.70 (medium-large) and 0.55 (small-medium) for ISBSG2000, ISBSG2001 and ISBSG, respectively. The differences in SA were 11.37, 23.89 and 5.24, respectively. These differences are of considerable magnitude.

Therefore, when using Clustering Dycom with EM, it is possible to drastically reduce the number of WC training projects, while maintaining or even improving MAE. This means that EM can not only facilitate the creation of the CC subsets by deciding their content, but also their number, giving a positive answer to RQ2.

9 THREATS TO VALIDITY

The experiments used the same databases and use mostly the same setup as Dycom's original paper [35]. Therefore, they have similar threats to validity as follows. When using machine learning approaches, it is important that the approaches being compared use fair parameter choices in comparison to each other in order to address internal validity [29, 41]. In this paper, both the RTs used as WC learners and within Dycom and Clustering Dycom used the same parameters, which were the ones more likely to obtain good results in the literature [34]. (Clustering) Dycom has two extra parameters (β and lr) which were set to the same values for all databases used in this study, i.e., they were not fine tuned for each database and therefore should not lead to an unfair advantage to (Clustering) Dycom. The number M of CC subsets was varied from 1 to 6 to answer the RQs. The other parameters of the clustering methods were set to default values in order not to give Clustering Dycom an unfair advantage over Dycom or the baseline RTs.

ISBSG, ISBSG2000 and ISBSG2001 have an overlap of projects. Therefore, these databases are not independent. However, in online learning, the conclusions that may be obtained with different numbers of preceding and following projects can be very different [36]. For example, as will be shown in section 7, Hierarchical Clustering with 6 clusters appeared to be good for ISBSG2000, but led to poor results for ISBSG. Therefore, it is important to include all versions of the ISBSG database in the analysis, as in previous work [33, 35, 36].

In order to address construct validity, this study used MAE. This is a measure unbiased towards over or underestimations, and has been recommended for SEE studies [39]. SA [39] was also used in order to give a better idea of the magnitude of the differences in performance and the impact that they are likely to have in practice. Wilcoxon Sign-Rank statistical tests with Holm-Bonferroni corrections were used to check the statistical significance of the differences in terms of MAE. Effect size A12 was used to support the comparison, as it is independent of the number of observations in the groups being compared.

Besides never using a WC project for training before using it for testing, we used five databases to handle external validity. Four databases with known WC chronological order were used in the evaluation. Even though the WC chronological order is unknown for the other database (CocNasaCoc81), it can still be used to evaluate whether (Clustering) Dycom is able to successfully make use of CC models, contributing to the generalisation of our results. For ISBSG, some future CC projects were used to simulate the case where it is known that some other companies are likely to be more evolved than the single company being analysed. The use of simulation or synthetic data is common practice in online learning studies [8]. Obtaining additional databases for this study is difficult due to our need for non-proprietary data sets with information on which projects belong to a single-company among the projects of a cross-company data set. However, the databases used in this study can be made available through SEACRAFT [28] and ISBSG [13]. So, researchers and companies willing to use Dycom could use the same CC data sets used in this study. Future work should also investigate Dycom with other base learners, clustering methods, input attributes and clustering features.

As in [35, 36], CC data were considered as fixed CC datasets. As shown in [36], such fixed CC datasets can be useful for prolonged periods of time. So, Clustering Dycom as investigated here is applicable in practice. If the weights of all CC mapped models become too low for a prolonged period of time, this may be an indication that Dycom needs to be re-built with updated CC datasets. Future work should investigate a more streamlined approach to update CC models.

10 CONCLUSIONS

Dycom is a promising approach for SEE. It is able to reduce the amount of WC training projects required for training SEE models

while maintaining or improving the predictive performance of a corresponding WC approach. However, its good predictive performance depends on the choice of a CC split to create its CC models. A poor choice can lead to worse predictive performance than WC approaches. Therefore, this paper investigated whether clustering methods can help to create good CC splits for use with Dycom.

Three different clustering methods were investigated. Among them, K-Means was able to improve predictive performance over the original Dycom strategy for creating the CC splits (RQ1). However, the predictive performance obtained when using K-Means depends on the prior choice of a suitable number M of CC subsets. A poor choice can still lead to worse predictive performance than a baseline WC approach. Even though EM did not achieve so good predictive performance as Clustering Dycom with K-Means with the best M, it can automatically determine the number of CC subsets while maintaining or improving the predictive performance of a corresponding baseline WC approach (RQ2).

Therefore, if the company has the necessary machine learning expertise and an initial set of WC training projects that is large enough to tune Dycom's parameters, we recommend to use Clustering Dycom with K-Means in order to boost Dycom's predictive performance. Otherwise, we recommend to use Clustering Dycom with EM, in order to avoid the risk of obtaining worse results than a WC approach.

This work has several possible future research directions. In particular, other clustering methods, base learners, project input attributes, project features for clustering, parameter values [41] and (automated) tuning procedures [1, 9] could be investigated, besides the proposal of a more streamlined approach to update CC models.

ACKNOWLEDGEMENTS

Part of this work has been conducted during Siqing Hou's internship at the University of Birmingham (UK).

REFERENCES

[1] A. Agrawal and T. Menzies. 2017. "Better Data" is Better than "Better Data Miners" (Benefits of Tuning SMOTE for Defect Prediction). *ArXiv preprint arXiv:1705.03697* (2017).
[2] S. Amasaki, Y. Takahara, and T. Yokogawa. 2011. Performance Evaluation of Windowing Approach on Effort Estimation by Analogy. In *IWSM-MENSURA*. Nara, Japan, 188–195.
[3] A. Arcuri and L. Briand. 2011. A Practical Guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE*. 1–10.
[4] C.M. Bishop. 2006. *Pattern Recognition and Machine Learning*. Springer.
[5] B. Boehm. 1981. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs.
[6] L.C. Briand, T. Langley, and I. Wieczorek. 2000. A Replicated Assessment of Common Software Cost Estimation Techniques. In *ICSE*. Como, Italy, 377–386.
[7] K. Dejaeger, W. Verbeke, D. Martens, and B. Baesens. 2012. Data Mining Techniques for Software Effort Estimation: A comparative study. *IEEE TSE* 38, 2 (2012), 375–397.
[8] G. Ditzler, M. Roveri, C. Alippi, and R. Polikar. 2015. Learning in Nonstationary Environments: A Survey. *CIM* 10, 4 (2015), 12–25.
[9] W. Fu, T. Menzies, and X. Shen. 2016. Tuning for Software Analytics: is it Really Necessary? *ArXiv preprint arXiv:1609.01759* (2016).
[10] J.J.C. Gallego, D. Rodriguez, M.A. Sicilia, M.G. Rubio, and A.G. Crespo. 2007. Software Project Effort Estimation Based on Multiple Parametric Models Generated Through Data Clustering. *JCST* 22, 3 (2007), 371–378.
[11] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. 2009. The WEKA Data Mining Software: An update. *SIGKDD Explorations* 11, 1 (2009), 10–18.
[12] S.-J. Huang, N.-H. Chiu, and Y.-J. Liu. 2008. A comparative evaluation on the accuracies of software effort estimates from clustered data. *IST* 50 (2008), 879–888.
[13] ISBSG. 2011. The International Software Benchmarking Standards Group. (2011). http://www.isbsg.org
[14] R. Jeffery, M Ruhe, and I. Wieczorek. 2010. A Comparative Study of Two Software Development Cost Modeling Techniques Using Multi-Organizational and Company-Specific Data. *IST* 42, 14 (2010), 1009–1016.
[15] M. Jørgensen and M. Shepperd. 2007. A Systematic Review of Software Development Cost Estimation Studies. *IEEE TSE* 33, 1 (2007), 33–53.
[16] B. Kitchenham and E. Mendes. 2004. A Comparison of Cross-Company and Within-Company Effort Estimation Models for Web Applications. In *METRICS*. Chicago, 348–357.
[17] B.A. Kitchenham, E. Mendes, and G.H. Travassos. 2007. Cross versus Within-Company Cost Estimation Studies: A Systematic Review. *IEEE TSE* 33, 5 (2007), 316–329.
[18] B. Kitchenham, S. L.. Pfleeger, B. McColl, and S. Eagan. 2002. An empirical study of maintenance and development estimation accuracy. *JSS* 64 (2002), 57–77.
[19] E. Kocaguneli, B. Cukic, T. Menzies, and H. Lu. 2013. Building a Second Opinion: learning cross-company data. In *PROMISE*. 12.1–10.
[20] E. Kocaguneli, G. Gay, T. Menzies, Y. Yang, and J. W. Keung. 2010. When to Use Data from Other Projects for Effort Estimation. In *ASE*. Antwerp, Belgium, 321–324.
[21] E. Kocaguneli, T. Menzies, A. Bener, and J. W. Keung. 2012. Exploiting the Essential Assumptions of Analogy-Based Effort Estimation. *IEEE TSE* 38, 2 (2012), 425–438.
[22] E. Kocaguneli, T. Menzies, and E. Mendes. 2015. Transfer Learning in Effort Estimation. *Empirical Software Engineering Journal* 20, 3 (2015), 813–843.
[23] M. Lefley and M. Shepperd. 2003. Using Genetic Programming to Improve Software Effort Estimation Based on General Data Sets. In *GECCO*, Vol. LNCS 2724. Chicago, 2477–2487.
[24] S. G. McDonell and M.J. Shepperd. 2007. Comparing Local and Global Software Effort Estimation Models – Reflections on a Systematic Review. In *ESEM*. Madrid, 401–409.
[25] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmerman. 2013. Local vs. Global Lessons for Defect Prediction and Effort Estimation. *IEEE TSE* 39, 6 (2013), 822–834.
[26] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann. 2013. Local vs. Global Lessons for Defect Prediction and Effort Estimation. *IEEE TSE* 39, 6 (2013), 822–834.
[27] T. Menzies, R. Krishna, and D. Pryor. 2015. The Promise Repository of Empirical Software Engineering Data. (2015). http://openscience.us/repo
[28] T. Menzies, R. Krishna, and D. Pryor. 2017. The SEACRAFT Repository of Empirical Software Engineering Data. (2017). tiny.cc/seacraft
[29] T. Menzies and M. Shepperd. 2012. Special Issue on Repeatable Results in Software Engineering Prediction. *Empirical Software Engineering Journal* 17 (2012), 1–17.
[30] L. Minku. 2016. On the Terms Within- and Cross-Company in Software Effort Estimation. In *PROMISE*. Ciudad Real, Spain, 4.1–4.4.
[31] L.L. Minku. 2017. *An Investigation of Dycom's Sensitivity to Different Cross-Company Splits*. Technical Report. Department of Informatics, University of Leicester. http://www.cs.le.ac.uk/people/llm11/publications/dycom-cc-splits.pdf
[32] L. Minku, F. Sarro, E. Mendes, and F. Ferrucci. 2015. How to Make Best Use of Cross-Company Data for Web Effort Estimation?. In *ESEM*. Bergamo, Italy.
[33] L.L. Minku and X. Yao. 2012. Can Cross-company Data Improve Performance in Software Effort Estimation?. In *PROMISE*. Lund, Sweden, 69–78.
[34] L.L. Minku and X. Yao. 2013. Ensembles and Locality: Insight on Improving Software Effort Estimation. *IST* 55, 8 (2013), 1512–1528.
[35] L. Minku and X. Yao. 2014. How to Make Best Use of Cross-company Data in Software Effort Estimation?. In *ICSE*. Hyderabad, 446–456.
[36] L. Minku and X. Yao. 2017. Which Models of the Past Are Relevant to the Present? A software effort estimation approach to exploiting useful past models. *Automated Software Engineering Journal* 24, 3 (2017), 499–542.
[37] L. Rokach and O. Maimon. 2005. *Clustering Methods*. Springer, 321–352.
[38] P. Sentas, L. Angelis, I. Stamelos, and G. Bleris. 2005. Software Productivity and Effort Prediction with Ordinal Regression. *IST* 47 (2005), 17–29.
[39] M. Shepperd and S. McDonell. 2012. Evaluating Prediction Systems in Software Project Estimation. *IST* 54, 8 (2012), 820–827.
[40] M. Shepperd and C. Schofield. 1997. Estimating Software Project Effort Using Analogies. *IEEE TSE* 23, 12 (1997), 736–743.
[41] L. Song, L.L. Minku, and X. Yao. 2013. The Impact of Parameter Tuning on Software Effort Estimation Using Learning Machines. In *PROMISE*. Baltimore, USA, Article No. 9, 10p., doi: 10.1145/2499393.2499394.
[42] B. Turhan and E. Mendes. 2014. A Comparison of Cross- versus Single- Company Effort Prediction Models for Web Projects. In *SEAA*. Verona, Italy, 285–292.
[43] A. Vargha and H. D. Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. In *Journal of Educational and Behavioral Statistics*, Vol. 25. 101–132.
[44] I. Wieczorek and M. Ruhe. 2002. How Valuable Is Company-Specific Data Compared to Multi-Company Data for Software Cost Estimation?. In *METRICS*. Ottawa, 237–246.

Scripted GUI Testing of Android Apps: A Study on Diffusion, Evolution and Fragility

Riccardo Coppola, Maurizio Morisio and Marco Torchiano
Dipartimento di Informatica e Automatica
Politecnico di Torino
Turin, Italy
name.surname@polito.it

ABSTRACT

Background. Evidence suggests that mobile applications are not thoroughly tested as their desktop counterparts. In particular GUI testing is generally limited. Like web-based applications, mobile apps suffer from GUI test fragility, i.e. GUI test classes failing due to minor modifications in the GUI, without the application functionalities being altered.

Aims. The objective of our study is to examine the diffusion of GUI testing on Android, and the amount of changes required to keep test classes up to date, and in particular the changes due to GUI test fragility. We define metrics to characterize the modifications and evolution of test classes and test methods, and proxies to estimate fragility-induced changes.

Method. To perform our experiments, we selected six widely used open-source tools for scripted GUI testing of mobile applications previously described in the literature. We have mined the repositories on GitHub that used those tools, and computed our set of metrics.

Results. We found that none of the considered GUI testing frameworks achieved a major diffusion among the open-source Android projects available on GitHub. For projects with GUI tests, we found that test suites have to be modified often, specifically 5%-10% of developers' modified LOCs belong to tests, and that a relevant portion (60% on average) of such modifications are induced by fragility.

Conclusions. Fragility of GUI test classes constitute a relevant concern, possibly being an obstacle for developers to adopt automated scripted GUI tests. This first evaluation and measure of fragility of Android scripted GUI testing can constitute a benchmark for developers, and the basis for the definition of a taxonomy of fragility causes, and actionable guidelines to mitigate the issue.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PROMISE , Toronto, Canada
© 2017 ACM. 978-1-4503-5305-2/17/11...$15.00
DOI: 10.1145/3127005.3127008

KEYWORDS

Mobile Development, Automated Software Testing, GUI Testing, Software Evolution, Software Maintenance

ACM Reference format:
Riccardo Coppola, Maurizio Morisio and Marco Torchiano. 2017. Scripted GUI Testing of Android Apps:
A Study on Diffusion, Evolution and Fragility. In *Proceedings of PROMISE , Toronto, Canada, November 8, 2017,* 11 pages.
DOI: 10.1145/3127005.3127008

1 INTRODUCTION

Android has reached a very significant market share with respect to other mobile systems (86.2% in Q2 '16[1]), and mobile devices have largely overtaken desktop ones in terms of shipped units (1.91 to 0.25 billion in 2015[2]). Mobile devices offer their users a large number of applications, capable of performing tasks that just a few years ago were exclusively available on high-end desktop computers.

One of the characteristics that have brought Android to its success is the availability of marketplaces (e.g., the Play Store) where developers can sell -or release for free- their applications. The huge quantity of software published on those platforms, and the resulting competition, makes crucial for the applications to behave as promised to their users.

Thus, testing applications and their GUI (i.e., Graphical User Interface), through which most of the interaction with the final user is performed, becomes a valuable practice to ensure that no crashes and no undesired behaviours happen during a typical execution.

However, there is evidence that Android applications -and mobile applications in general- are not deeply tested as they should be. Although a variety of testing tools (open-source or not) are available, most Android developers rely just on manual testing. Some developers do not perform testing at all, leaving the recognition of faults and bugs to the feedback of their users. Evidence about this lack of testing is given in [18], where only 14% of the set of applications considered featured any kind of test classes.

In addition to this need for testing, Android development comes with a set of domain-specific challenges, that have consequences for testing. The main differences between traditional software and Android applications are: the great quantity of different context events to which the apps have

[1] https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/
[2] http://www.gartner.com/newsroom/id/3187134

to react properly; the diversity of devices and configurations where apps will eventually be deployed; the very fast pace of evolution of the operating system; the lack of resources that has been intrinsic for a long time for mobile devices[29].

Among those peculiarities of Android testing, we focus on the problem of fragility of GUI test classes. We consider the fragility of test classes among the main factors that may discourage the adoption of GUI testing for Android applications, since developers may decide to not exercise any testing if even small changes in the user interface may break entire test suites. In our previous study [8] on a popular open-source Android application, K-9 mail, a small test suite was developed and adapted to different releases. We found that up to 75% of the tests developed had to be modified because of modifications in the GUI.

With this work we aimed at gathering information about test suites in released open-source projects. We collected statistics about the level of penetration of six popular tools that are used for Android GUI testing, among open-source applications whose source code is available on GitHub. For those projects that leveraged the tools, we measured the quantity of test code they featured, and counted the modifications performed on test classes during their lifespan.

We define the concept of fragility of test classes, and provide metrics to estimate the fragility of a project by automated inspection of its test suite. This allows us to give a characterization and quantification of the fragility issue on a large set of Android projects, and can be an aid to developers to evaluate the maintenance effort needed by their scripted test suites. This evaluation can serve, in the future, as a base for a taxonomy of fragility causes, a set actionable guidelines to help developers to avoid it, and finally automated tools capable of adapting the test methods to modificatons made in the user interfaces.

2 BACKGROUND AND RELATED WORK

This section provides an introduction to Android application testing, and a survey of existing papers about the challenges it presents.

Mobile apps are defined [29] as mobile software (i.e., applications that run on mobile devices) taking input from the context where they are executed (for instance, contextual sensing and adaptation, and context-triggered actions). They can be distinguished between native apps, if they are designed to run on a specific mobile platform according to its design patterns, or web-based apps, if they are based on web sites engineered to be loaded by a browser application, with partial or no use of the specific functionalities of the mobile device [16].

2.1 Testing Android apps

Mobile testing can be defined as "testing native and Web applications on mobile devices using well-defined software test methods and tools to ensure quality in functions, behaviours, performance, and quality of service"[9] .

Testing of mobile apps can be performed on a series of different levels: in addition to the traditional unit testing, integration testing, system testing and regression testing, scopes that are specific to the mobile scenario must be considered. In [15] compatibility testing (i.e., to ensure that the application works on different handheld models and/or OS versions), performance testing (i.e., to ensure that the mobile devices do not consume too many of the resources available) and security testing are discussed. GUI testing is identified as a very prominent testing need for all mobile applications. For Android applications, GUI testing is focused on testing the *Activities* (i.e. the components in charge of managing the graphical user interfaces) and the transitions between the screens they are composed from.

The first and most immediate option for testing Android applications and their GUIs is the execution of manual test cases. In [23], a study conducted in the field of performance testing, manual testing is identified as the option preferred by developers, along with an examination of reports and feedback from users. The technique, as discussed in [19], is however not exhaustive, error prone and not reproducible.

The approaches for automated GUI testing of Android applications can be classified as follows [22]: fuzzy (or random) testing, model-based testing techniques, capture and replay, white-box scripted testing. Most of them allow, in some cases without having any access to the source code (i.e., only the .apk package of the application is needed), to generate test scripts that can be therefore executed quickly and repeatedly.

Without any additional information about the AUT (Application Under Test) random and fuzzy testing techniques give random sequences of inputs to activities, in order to trigger potential defects and crashes. Monkey[3] is the random tester supported by Android. Random testers can be applied after a model of the user interface is created (like it is done in [25], [28] and [39]) to distribute the input given to the interface in a more intelligent way.

Model-based testing techniques leverage models (typically Finite State Machines or Event-Flow Graphs) of the GUI of the apps under test, that can be created manually or extracted automatically with a process called GUI ripping. Such models are therefore used to generate systematic test cases traversing the GUI. The tools and studies in [2], [3], [36] can serve as examples of this approach.

Capture & Replay testing tools (examples are presented in [11], [14] and [24]) record the operations performed on the UI to generate repeatable test sequences. Event-sequence generation tools are based on the construction of test cases as streams of events, that then can be inserted in repeatable scripts: [5] and [13] are examples of this paradigm.

Less coverage (two examples are given in [19] and [32]) is present in literature about white-box approaches and scripted testing techniques, which require the developer to have access to the code and manually write down testing code with sequences of operations to be performed on the AUT.

[3] https://developer.android.com/studio/test/monkey.html

Several studies (like [15] and [29]) are focused on the peculiarities of Android apps that make testing them properly a complex challenge: limited energy, memory and bandwidth; rapid changes of context and connectivity type; constant interruptions caused by system and communication events; the necessity to adapt the input interface to a wide set of different devices; very short time to market; very high multitasking and interaction with other apps.

The authors in [18] find that time constraints, compatibility issues, complexity and lack of documentation of available testing tools are among the most relevant challenges experienced by the interviewed developers, that may therefore be discouraged from testing their applications.

2.2 Test Fragility

Test fragility (defined for not GUI-based testing by Garousi et al. [10]) represents a problem for different kind of software: Leotta et al. [20, 21] report a study on web application UI tests. A list of the possible causes of fragilities for mobile applications is reported in [8]: identifier and text changes inside the visual hierarchy of activities; deletion or relocation of their elements; usage of physical buttons; layout and graphics change; adaptation to different hardware and device models; activity flow variations; execution time variability.

For our purposes, which is an evaluation of GUI testing of Android apps, we will use the following definition of fragile GUI tests.

> A GUI test class is said to be fragile when:
> - it needs modifications when the application evolves;
> - the need is not due to the modification of the functionalities of the application, but to changes in the interface arrangement and/or definition.

Modifications performed in test code may be due to different reasons and therefore divided into four categories [37]: perfective maintenance, when test code is refactored to enhance its quality (e.g. to increase coverage or to adopt well-known test patterns); adaptive maintenance, to make test code evolve according to the evolutions of the production code; preventive maintenance, to change aspects of the code that may require intervention in future releases; corrective maintenance, to perform bug fixes. According to our definition of GUI testing fragility, we are interested in cases of adaptive maintenance, in which the modifications in the production code are GUI-related.

The manual identification of test fragility occurrences in the history of software projects is time consuming and requires a careful inspection of different version of the test code together with the application production code. For those reasons we propose an automatic classification approach: any time a pre-existent method of a GUI test class is modified we assume the change is due to test fragility. Other test class modifications are not attributed to fragility; for instance, the modifications may involve only import statements and class constructors, or the addition and removal of test methods. We suppose, in fact, that the addition of a new method should reflect the introduction of new functionalities or new use cases to be tested in the application, and not the modification of existing elements of the already tested activities. On the other hand, if some lines of code inside a single test method had to be changed or added, it is more likely that tests had to be modified due to minor changes in the application and possibly in its user interface (e.g. modifications in the screen hierarchy and in the transitions between activities).

3 STUDY DESIGN

The goals of this work can be described following the Goal-Question-Metric template [35]: the main objective is to estimate and assess the quantity of fragility-induced changes in test code, in the context of automated scripted GUI testing of Android applications. The goal entails answering the following research questions:

RQ1 *Diffusion: how many projects use automated testing tools, and how much test code do they produce?*

RQ2 *Evolution: how much test code is modified over different releases?*

RQ3 *Fragility: how fragile are Android UI tests?*

The first step of our research was to estimate the diffusion of Android UI testing. We started from a repository of Android open-source applications – we selected GitHub for this purpose – and we performed a code search in order to detect the usage of a set of six testing tools that are frequently cited in literature.

Then, we studied how applications (and their test classes) were changed throughout their release history, by means of file-by-file comparisons. Finally, with the aid of an automated shell script, we tracked the modifications of individual test classes and methods to compute a set of change indicators. The script cycles over all the releases of each project, for each of them performing the respective *git clone* command to locally download all the files. Then the files are locally investigated to compute size statistics, about Project and Test code. The *git diff* command is leveraged by the script to compute modification statistics between each pair of consequent releases of the project history. Metrics are then computed as explained in detail in section 3.3.

Since we are interested in the evolution of test cases and classes, we excluded from our analysis the projects featuring less than two tagged releases (including master).

Finally, to validate the accuracy of the fragility measures, we selected a random sample of the analyzed projects, and checked the precision of the measures compared to the outcome of a manual inspection of the modified test classes.

3.1 Metrics definition

We defined a set of metrics that can be divided into three groups according to the research question they address. Table 1 reports the metrics togehter with the relative descriptions. The metrics are explained in detail in the following subsections.

Table 1: Metrics definition

Group	Name	Explanation
Diffusion and size (RQ1)	TD	Tool Diffusion
	NTR	Number of Tagged Releases
	NTC	Number of Test Classes
	TTL	Total Test LOCs
	TLR	Test LOCs Ratio
Test evolution (RQ2)	MTLR	Modified Test LOCs Ratio
	MRTL	Modified Relative Test LOCs
	TMR	Test Modification Relevance Ratio
	MRR	Modified Releases Ratio
	TSV	Test Suite Volatility
Fragility (RQ3)	MCR	Modified Test Classes Ratio
	MMR	Modified Test Methods Ratio
	FCR	Fragile Classes Ratio
	RFCR	Relative Fragile Classes Ratio
	FRR	Fragile Releases Ratio
	ADRR	Releases with Added-Deleted Methods Ratio
	TSF	Test Suite Fragility

13 out of the 17 metrics we defined are normalized, to allow comparison across projects of different sizes. Most of them can be defined on top of lower-level metrics for the quantification of absolute changes in test classes and test cases. For instance, Tang et al. [34] report eighteen basic metrics for the description of bug-fixing change histories (e.g., number of added or removed files, classes, methods or dependencies).

3.1.1 Diffusion and size (RQ1). To estimate the diffusion of Android automated UI testing tools and of the size of test suites using them, we defined the following five metrics:

TD (Tool Diffusion) is defined as the percentage, among the set of Android projects in our context, of those featuring a given testing tool.

NTR (Number of Tagged Releases) is the number of tagged releases of an Android project (i.e., the ones that are listed by using the command *git tag* on the GIT repository).

NTC (Number of Test Classes) is the number of test classes featured by a release of an Android project, relatively to a specific tool.

TTL (Total Test LOCs) is the number of lines of code that can be attributed to a specific testing tool in a release of an Android project.

TLR (Test LOCs Ratio) defined as $TLR_i = TTL_i/Plocs_i$ where $Plocs_i$ is the total amount of Program LOCs for release i. This metric, lying in the $[0, 1]$ interval, allows us to quantify the relevance of the testing code.

3.1.2 Test suite evolution (RQ2). The metrics addressing RQ2 aim to describe the evolution of Android projects and the relative test suites; they have been computed for each pair of consecutive tagged releases.

MTLR (Modified Test LOCs Ratio) defined as $MTLR_i = Tdiff_i/TTL_{i-1}$, where $Tdiff_i$ is the amount of added, deleted or modified test LOCs between tagged releases $i-1$ and i, and TTL_{i-1} is the total amount of test LOCs in release $i-1$. This quantifies the amount of changes performed on existing test LOCs for a specific release of a project.

MRTL (Modified Relative Test LOCs) defined as $MRTL_i = Tdiff_i/Pdiff_i$, where $Tdiff_i$ and $Pdiff_i$ are the amount of modified (or added, or deleted) test and project LOCs, in the transition between release $i-1$ and i. It is computed only for releases with test code (i.e., $TRL_i > 0$). This metric lies in the $[0, 1]$ range. Values close to 1 imply that a significant portion of the total effort in making the application evolve is needed to keep test methods up to date.

TMR (Test Modification Relevance Ratio) defined as $TMR_i = MRTL_i/TLR_{i-i}$. This ratio can be an indicator of the proportion of effort needed to adapt test classes during the evolution of the application. It is computed only when $TLR_{i-1} > 0$. We consider a value greater than 1 as an index of greater effort needed in modifying the test code than the actual relevance of test code inside the application.

MRR (Modified Releases Ratio), computed as the ratio between the number of tagged releases in which at least a test class has been modified, and the total amount of tagged releases. This metric lies in the range $[0, 1]$ and bigger values indicate a minor adaptability of the test suite -as a whole- to changes in the AUT.

TSV (Test Suite Volatility), is defined for each project as the ratio between the number of test classes that are modified at least once in their lifespan, and the total number of test classes of the project history.

3.1.3 Fragility of tests (RQ3). With an automatic inspection of test code, information about modified methods and classes can be obtained. Based on such data, the metrics answering RQ3 aim to give an approximated characterization of the fragility of test suites.

The number of modified classes with modified methods can be different from the total number of modified classes in three different cases (and their combinations): (i) when the modifications performed to the classes involve non-significant portions of code like comments, imports, declarations; (ii) when the modifications performed to the classes involve only additions of test methods; (iii) when the modifications performed to the classes involve only removal of test methods. Additions and removals of test methods are considered the consequence of a new functionality or a new use case of the application, hence they are not considered as an evidence of fragility of test classes. On the other hand, modifications of test methods may be strictly linked with fragilities.

MCR (Modified test Classes Ratio) defined as $MCR_i = MC_i/NTC_{i-1}$, where MC_i is the number of modified test classes in the transition between release $i-1$ and i, and NTC_{i-1} the number of test classes in release $i-1$ (the metric is not defined when $NTC_{i-1} = 0$). The metric lies in the $[0, 1]$ range: the larger the values of MCR, the less test classes are stable during the evolution of the app.

MMR (Modified test Methods Ratio) defined as $MMR_i = MM_i/TM_{i-1}$, where MM_i is the number of modified test methods between releases $i-1$ and i, and TM_{i-1} is the total number of test methods in release $i-1$ (the metric is not defined when $TM_{i-1} = 0$). The metric lies in the $[0, 1]$ range: the larger the values of MMR, the less test methods are stable during the evolution of the app they test.

FCR (Fragile Classes Ratio) defined as $FCR_i = MCMM_i/NTC_{i-1}$, where $MCMM_i$ is the number of test classes that

are modified, and that feature at least one modified method between releases $i-1$ and 1. The metric is not defined when $NTC_{i-1} = 0$. This metric represents an estimate of the percentage of fragile classes, upon the entire set of test classes featured by a tagged release of the project. The metric is upper-bounded by *MCR*, since by its definition $MCR_i = MC_i/TC_i$, and $MCMM_i \leq MC_i$.

RFCR (Relative Fragile Classes Ratio) defined as $RFCR_i = MCMM_i/MC_i$, where $MCMM_i$ and MC_i are defined as above.

FRR (Fragile Releases Ratio), computed as the ratio between the number of tagged releases featuring at least a fragile class, and the total amount of tagged releases featuring test classes. This metric lies in the range $[0, 1]$ and is upper-bounded by *MRR*.

ADRR (Releases with Added-Deleted Methods Ratio), computed as the ratio between the number of tagged releases in which at least a test method has been added or removed, and the total amount of tagged releases featuring test classes. This metric lies in the range $[0, 1]$, and higher values should imply more frequent changes in application functionalities and defined use cases to be tested.

TSF (Test Suite Fragility), is defined for each project as the ratio between the number of test classes that feature fragilities at least once in their lifespan, and the total number of test classes of the project history.

To validate the metrics defined for fragile classes and fragile methods (since we may consider as fragile tests that are modified for reasons different from GUI modifications) we adopt the following metric:

P (Precision), is defined as $P = TP/(TP + FP)$, where *TP* is the number of True Positives, in our case the test classes (or methods) that feature changed test code, and whose modifications reflect changes in the GUI of the AUT; *FP* is the number of False Positives, i.e., the test classes (or methods, according to which is being validated) that feature changed test code, but due to different reasons. *P* is defined in the range $[0, 1]$: values closer to 1 are an evidence that the presence of modified lines in test methods is a dependable proxy to identify modifications in test classes due to changes related to the user interface of the application. As our oracle for the computation of Precision, we leverage a manual inspection of a set of selected test classes, before and after they undergo modifications.

3.2 Selected Testing Tools

We have chosen six different popular scripted testing tools for our investigations. We selected open-source testing tools that were already considered in similar explorations of the testing procedure of Android applications. All those testing tools give the possibility to write test scripts manually.

The first two tools we have searched for are part of the official *Android Instrumentation Framework*[4]. *Espresso* [17] is an open-source automation framework that allows to test the UI of a single application, leveraging a gray-box approach (i.e., the developer has to know the internal disposition of elements inside the view tree of the app, to write scripts exercising them). *UI Automator*[6, 22]adds some functionalities to those provided by Espresso: it allows to check the device status and performance, to perform testing on multiple applications at the same time, and operations on the system UI. Both tools can be used only to test native applications.

Selendroid[5] [33] is a testing framework based on Selenium, that allows to test the UI of native, hybrid and web-based applications; the tool allows to retrieve elements of the application and to inspect the current state of the app's UI without having access to its source code, and to execute the test methods on multiple devices at the same time.

Robotium[12, 38] is an open-source extension of JUnit for testing Android apps, that has been one of the most used testing tools since the beginning of the diffusion of Android programming; it can be used to write black-box test scripts or function tests (if the source code is available) of both native and web-based apps.

Robolectric[6] [1, 26, 27] is a tool that can be used to perform black-box testing directly on the Java Virtual Machine, without the use of a real device or an emulator; it can be considered as an enabler of Test-Driven Development for Android applications, since the instrumentation of Android emulators is significantly slower than the direct execution on the JVM.

Appium[31, 32] leverages WebDriver and Selendroid for the creation of black-box test methods that can be run on multiple platforms (e.g., Android and iOS); test methods can be created via an inspector that enables basic functions of recording and playback, via image recognition, or via code. It can be used to test both native and web-based applications. Test scripts can be data-driven.

3.3 Procedure

Three main phases can be identified in the study, each relative to one of the three research questions defined. The following paragraphs describe the steps performed in detail.

3.3.1 Test code Search (RQ1). The approach we adopted for the selection of the context (i.e., the set of projects that we used for the subsequent study) is a sequence of different steps, the first one being a search for the word "Android" in descriptions, readmes and names of projects. The Repository Search API of Git has been leveraged to this purpose. All the projects extracted this way were cloned locally.

All the projects that have no tagged releases are cut out from the context. This is done because the aim of the experiment is to track the evolutions of the projects, by means of computing differences between tagged releases (as it is explained later). That considered, projects without at least a single tagged release (which allows for a single comparison, made between it and the master release) are not of interest. To know how many releases were featured by each cloned

[4] https://developer.android.com/studio/test/index.html

[5] https://github.com/selendroid/selendroid

[6] http://robolectric.org/

repository, we leveraged the *Git tag* command, which outputs the names of all the tagged releases.

The keyword "Android" alone would include libraries, utilities, and applications intended to interface with Android counterparts. Since it is mandatory for any Android app to have a Manifest file in its root directory. We excluded projects that do not contain any manifest file.

Once a filtered list of Android projects is obtained, they are searched for the presence of JUnit test classes. The amount of JUnit test classes can serve as a comparison to evaluate the diffusion of other tools and testing techniques. To search for the considered testing tools, a GitHub Code Search, with the names of the tools as keywords, has been performed on the remaining repositories. For each tool its adoption has been estimated by means of the TD metric. Sets of projects featuring different testing tools are not disjoint: it is possible that a repository features more than just one scripted testing tool. Even though some of the chosen tools are based on JUnit, the researches have been conducted independently and in parallel. Obviously, if a tool is based on JUnit, the set of projects featuring JUnit will be a superset of the set of projects featuring that specific tool. The data extraction has been performed between September and December 2016.

We consider any ".java" file featuring the name of a testing technique in its code as a test class (for instance, a class featuring the statement "import static android.support.test.espresso. Espresso.onView;" is considered as a class featuring Espresso). For each test class the lines of test code are counted, so that TTL and NTC can be computed for each project, on the master release. The use of the *git tag* command allows to obtain the NTR metric.

3.3.2 Test LOCs analysis (RQ2). To answer RQ2, for each pair of consecutive tagged releases of any project, the total amount of modified LOCs is computed.

Then, the total amount of LOCs added, removed or modified in the test files previously identified is computed. Throughout all our study, we have considered moved or renamed files as different test files.

Those values allow to compute TLR, $MTLR$, $MRTL$ and TMR for each tagged release of the project.

Finally, when the exploration of the project history is complete, global averages are computed: $\overline{TLR} = Avg_i\{TLR_i\}$, $\overline{MTLR} = Avg_i\{MTLR_i\}$, $\overline{MRTL} = Avg_i\{MRTL_i\}$, $\overline{TMR} = Avg_i\{TMR_i\}$ with $i \in [2, NTR]$, being NTR the number of tagged releases featured by the project.

Volatile classes (i.e., classes featuring modifications throughout their lifespan) have been identified inside each project, in order to compute the TSV value.

3.3.3 Test classes history tracking, Fragility (RQ3). We have finally tracked the evolution of single test classes and methods, taking into account the tagged releases in which each test class has been added, modified or deleted.

Then, for each tagged release we have obtained the number of modified classes and methods, i.e. MCR and MMR, and the derived metrics $RFCR$ and FCR. Also in this case, at the end of the exploration averages have been computed as $\overline{MCR} = Avg_i\{MCR_i\}$, $\overline{MMR} = Avg_i\{MMR_i\}$, $\overline{FCR} = Avg_i\{FCR_i\}$, with $i \in [1, NTR]$.

Since $RFCR$ makes sense only when modifications are actually present, $\overline{RFCR}$ has been computed as an average of $RFCR$ only for release transitions in which test classes have been modified (i.e., $MCR \neq 0$).

At the end of the exploration of the tagged releases of each project, FRR and $ADRR$ have been computed to quantify the percentage of them featuring, respectively, fragile and non-fragile modifications.

Based on the recognition of classes affected by fragilities, the overall TSF value has been computed for each project.

A manual inspection of a set of modified test classes with modified methods has been conducted, in order to verify the dependability of the metrics defined to identify fragile methods and fragile classes (i.e., MMR and FCR).

30 pairs of consecutive releases of different classes have been selected randomly, and manually inspected before and after they were modified. The modifications performed were characterized under three categories: (i) test code refactoring, syntactical correction and formatting; (ii) adaptation to changes in program code not related to GUI; (iii) adaptation to changes in program code related to GUI.

Only the modifications belonging to the last category are considered as true positives for our analysis; the others are considered as false positives. Based on that subdivision, the precision of the metrics is computed for the percentage of fragile classes, and the percentage of fragile methods.

4 RESULTS AND DISCUSSION

In the following paragraphs, we report the results we obtained by applying the procedure described in the previous section. The results measured for the metrics defined in section 3.1 are detailed, along with the conclusions we can base on them. The full set of intermediate data about classes and releases of each project has been made available online[7].

We initially gathered a total of 280,447 GitHub repositories featuring the term *Android* in their names, descriptions or readmes. Then, a significant amount of projects were pruned because of their lack of tagged releases (so they had no history to be investigated), or Manifest files. A final set of 18,930 Android projects was obtained (6.75% of the initial number of projects).

4.1 Diffusion and size (RQ1)

Table 2 summarizes the metrics gathered to answer RQ1. The columns show: the total number of projects featuring each of the six tools considered; the TD metric; the average and median values for NTR, NTC, TTL and TLR, computed on the sets of projects featuring each testing tool.

As a comparison for the diffusion of other testing tools, we counted the number of projects featuring the JUnit testing framework. We counted 3,669 projects (with tagged releases and manifest files) featuring JUnit, among the total set of Android projects we extracted (the 19.38%).

Table 2: *NTR*, *NTC*, *TTL*, *TLR* per testing tool: average and median (in parentheses) values for master release.

Tool	n	TD	*NTR*	*NTC*	*TTL*	*TLR*
Espresso	423	2.23%	15 (6)	5 (2)	588 (190)	8.8% (4.1%)
UIAutomator	134	0.71%	60 (25)	12 (3)	3,155 (1,134)	8.6% (0.6%)
Selendroid	6	0.03%	46 (17)	76 (1)	8,627 (126)	19.4% (0.2%)
Robotium	150	0.79%	44 (7)	5 (1)	873 (227)	8.7% (3.3%)
Robolectric	842	4.44%	22 (6)	11 (3)	1,448 (399)	16.4% (11.4%)
Appium	18	0.09%	27 (15)	38 (4)	4,469 (1096)	37.3% (6.0%)

Considering the overestimation due to possible overlaps (since the sets for the individual tools are not necessarily disjoint) about 8.5% of the set of projects feature tests belonging to one of the six selected tools. None of the testing frameworks reached by itself a significant level of diffusion. The absolute number of projects featuring Selendroid and Appium test classes is practically irrelevant. A higher number (the 4.44% of the total) of projects featuring Robolectric has been found, but the tool has been available for a longer time with respect to other ones (especially Espresso and UI Automator) and is often used solely for Unit Testing.

Although the total number of Android projects extracted can take into account some projects that are not likely to feature test classes (e.g. experiments, duplicates, exercises, prototypes, projects that are abandoned at very early stages) the statistics extracted about the metric TD give evidence of the lack of an extensive usage of scripted automated UI testing on Android. However, it must be taken into account that the study we performed is limited to the testing tools we considered, i.e. it is possible that different scripted testing tools are used by some other projects of the context.

The average and median number of test classes can be quite small (e.g., just 5 and 2, respectively, in the case of Espresso) due to the typical coding patterns for Android applications, in which -usually- one GUI testing class is written specifically for each Activity featured by the application. Most applications -this is particularly evident in the case of small and even experimental open-source projects- do not feature many screens to be shown to their users, and therefore they do not feature many activities to be tested.

Average TTL and TLR values are very large for both Selendroid and Appium; however, the result is heavily influenced by the small size of the sets of projects featuring these tools (respectively 6 and 18 projects) and by the presence of the full Selendroid framework for Android (selendroid/selendroid, with 47,436 LOCs) and of a very large set of Appium API demos (appium/android-apidemos, with 48,868 LOCs).

The fact that the set of projects featuring Espresso has the lowest average TTL can be explained with the following reasons: (i) using a white-box testing technique allows to exercise the functionalities of the application with little coding effort; (ii) the framework is quite accessible even to non-experienced developers, and its usage is encouraged by Android, leading it to be used also in very small projects, in tryouts, and even for experimental and partial coverage of applications use cases. On the other hand, the mean TTL for projects featuring UI Automator is very high, and also significantly higher with respect to the sets featuring Robotium, Robolectric and Espresso. This is mainly due to the cross-application features of UIAutomator, that make it recommended for the testing of whole firmwares and application bases, which are typically very big projects.

Table 3: Measures of the evolution of test code (averages on the sets of repositories)

Tool	$\overline{TLR}$	$\overline{MTLR}$	$\overline{MRTL}$	$\overline{TMR}$	MRR	TSV
Espresso	7.3%	2.6%	4.7%	0.68	22.2%	28.6%
UI Automator	9.6%	1.4%	3.5%	1.17	16.5%	35.9%
Selendroid	19.4%	4.3%	11.5%	0.15	39.6%	33.7%
Robotium	7.8%	3.8%	5.3%	0.56	22.1%	36.3%
Robolectric	13.4%	2.9%	9.5%	0.79	28.2%	30.4%
Appium	31.9%	1.8%	16.6%	0.27	27.3%	36.2%
Average	11.1%	2.8%	7.4%	0.76	25.2%	30.6%

The considered GUI testing tools reach a diffusion that is always lower than 4.5%. Projects that have their GUI tested feature on average 9 test classes, with a total of 1,361 LOCs (13.2% of the whole project code).

4.2 Test suite evolution (RQ2)

Table 3 shows the statistics collected about the average evolution of test code, for the six selected testing tools. For every set, $\overline{TLR}$, $\overline{MTLR}$, $\overline{MRTL}$, $\overline{TMR}$, MMR and TSV have been averaged on all the projects. The values in last row are obtained as averages of the six values above, weighted by the respective sizes of the six sets.

The values reported for average Test LOCs Ratio ($\overline{TLR}$) show that -when present- GUI testing can be an important portion of the project during its lifecycle, if compared to the number of LOCs of program code. The average values range from about 7.3% (for the set of Espresso projects) to 31.9% (for the set of Appium projects). For the largest set of projects considered (the ones featuring Robolectric) the mean $\overline{TLR}$ is 13.4%.

Average Modified Test LOCs Ratio ($\overline{MTLR}$) measures show that typically around 2.8% of test code is modified between consecutive releases. Very small values were obtained for the projects featuring UIAutomator. In general, this should be a consequence of bigger test suites, in terms of absolute LOCs, with respect to the ones written with other testing frameworks. Hence, the influence of a similar amount of absolute modified LOCs would result in a lower $\overline{MTLR}$ value. The highest value was found for the set of projects featuring Selendroid: this can be explained by the very high percentage of total LOCs belonging to testing code for these repositories. However, the set of projects featuring Appium did not exhibit the same trend, having a lower $\overline{MTLR}$: this should mean that, even though the important ratio of testing code above project code, few modifications (in both production and test code) were made between subsequent releases.

The measures about Modified Relative Test LOCs ($\overline{MRTL}$) show that, on average, when UI testing tools are used, the 7.4% of the modified LOCs belong to test classes. With this metric, however, we are still unable to discriminate what is

the reason behind the modifications to be performed on test classes. The higher $\overline{MRTL}$ values for the sets of projects featuring Appium and Selendroid can be justified by the small size of the two sets, and by the nature of the projects examined. For instance, the Selendroid framework, on GitHub as selendroid/selendroid, is subject to heavy modifications.

The mean values of Test Modification Relevance Ratio ($\overline{TMR}$) stayed in the range between 0.56 and 1.17 for big-sized sets of projects, with lower values for sets featuring Selendroid and Appium. In general, those values imply that the effort to spend in modifying test code is not linear with the relevance of test code inside the application: in our case, on average, the ratio between the intervention on test code and the intervention on program code is about 3/4 of the ratio between test and program code. The higher $\overline{TMR}$ value for UIAutomator is due to some projects (e.g. Lanchon/android-platform-tools-base) in which TLR is rather small, and where in some releases all modified LOCs belong to test classes (thus leading to $MRTL$ values very close to 1).

The Modified Releases Ratio (MRR) metric gives an indication about how often the developers had to modify any of their test classes when they published new releases of their projects. On average, 25.2% of releases needed modifications in the test suite (with a maximum of 39.6% for the set of projects featuring Selendroid). Since releases may be frequent and numerous for GitHub projects, this result explains that the need for updating test classes is a common issue for Android developers that are leveraging scripted testing. The average 30.6% value for the Test Suite Volatility (TSV) metric, which characterizes the phenomenon from the point of view of whole test suites, highlights that on the lifespan of a project, about one third of test classes require at least one modification.

On average, near 3% of testing code is modified between consecutive tagged releases. 7.4% of the overall LOCs modified between consecutive tagged releases belong to testing code. On average, one fourth of tagged releases require modifications in the test suite, and one third of the test suites needs modifications during the project history.

4.3 Fragility of tests (RQ3)

Table 4 shows the fragility estimations that we have computed for each project, and then averaged over the six sets: $\overline{MCR}$, $\overline{MMR}$, $\overline{FCR}$, $\overline{RFCR}$. Based on them, we computed three additional derived metrics: FRR, $ADRR$ and TSF. The values in last row are obtained as averages of the six values pertaining to the individual sets of projects, weighted by the respective sizes of the six sets.

The first column about the Modified Classes Ratio ($\overline{MCR}$) metric shows that, on average, 14.8% of test classes are modified between consecutive tagged releases in our set of Android projects. The only value significantly different from the average is the one obtained for the set of projects featuring UIAutomator, but it can be justified with the bigger amount of test classes that they feature on average (see table 2).

Table 4: Measures for RQ3 (averages on the sets of repositories)

Tool	$\overline{MCR}$	$\overline{MMR}$	$\overline{FCR}$	$\overline{RFCR}$	FRR	$ADRR$	TSF
Espresso	15.2%	3.5%	8.3%	59.7%	14.4%	17.7%	18.8%
UI Automator	9.0%	1.8%	4.6%	54.4%	10.2%	8.2%	16.6%
Selendroid	16.5%	2.7%	4.9%	42.2%	28.2%	23.2%	11.9%
Robotium	16.4%	3.5%	9.3%	53.1%	15.2%	21.2%	22.8%
Robolectric	15.1%	3.8%	8.5%	60.7%	20.6%	25.8%	19.4%
Appium	15.2%	4.6%	7.7%	48.2%	17.1%	23.5%	19.6%
Average	14.8%	3.6%	8.2%	59.1%	17.7%	21.9%	20.2%

The 3.6% average value found for the Modified Methods Ratio ($\overline{MMR}$) metric highlights that the percentage of modified methods is -as expected- smaller than the percentage of modified classes: this is obviously due to the fact that multiple test methods are contained in single test classes.

Not all modified test classes could be defined as fragile classes. The Relative Fragile Classes Ratio ($\overline{RFCR}$) metric gives a statistic about the possibility of a modified class to contain modified methods. The results collected show that more than half of the classes having modified lines featured modifications inside the code of test methods as well, hence they could be defined as fragile according to the heuristic definition given in section 2.4. The Fragile Classes Ratio ($\overline{FCR}$) metric gives the ratio between the classes that we define fragile upon all the classes contained by each project. On average, 8.2% of the classes were fragile in the transition between consecutive releases of the same project.

The Fragile Releases Ratio (FRR) metric gives an indication of how many releases of the considered project contained test classes that we identify as fragile. The value is upper-bounded by MRR, which is the frequence of releases featuring any kind of modification. The average value for $\overline{FRR} = 17.7\%$ means that about one every five releases records fragility-induced changes in test methods. The Releases with Added-Deleted Methods Ratio ($ADRR$) metric quantifies the probability that there is the need – between two subsequent releases – to add or delete test methods inside existing test classes. In general (with the only exception of the set of projects featuring UIAutomator) $ADRR$ is higher than FRR. This result is in accordance with the findings by Pinto et al.[30], who observed – in the context of traditional desktop applications – that the sum of test deletions and additions is higher, on average, than the number of test modifications. However, we can observe that the two values are generally close to each other: during the evolution the need for fragility induced test changes (FRR) occurs roughly as often as the definition of new test methods ($ADRR$).

Upper-bounded by TSV (the overall volatility for test suites), the average value for Test Suite Fragility (TSF) provides information about the amount of test classes, in each project, that need modifications because of fragilities. The average value of 20.2% tells us that one fifth of the classes in test suites face at least a fragility during its entire lifespan.

On each new release 14.8% of test classes and 3.6% of test methods are modified. Fragility-induced changes concern 8.2% of the classes. One every five releases feature fragile test classes, and 20.2% of the classes inside test suites are affected by fragilities at least once in their lifespan.
Overall the changes induced by fragility requires an effort comparable to the definition of tests for new features: both in terms of frequency (17.7% of releases with fragility-induced changes vs. 21.9% of releases with new or removed test methods) and number of classes interested (among the modified test classes 59.1% are affected by fragilities).

It must also be considered that the averages are significantly lowered by projects in which test classes have been added – at the beginning or at some point in their history – but never modified: in practice tests fell into oblivion. For instance, among the projects of 423 projects featuring Espresso, we detected modifications in test classes in 181 projects (43%), and modifications in test method in 144 projects (34%).

4.4 Fragility Metrics Validation

Table 5 shows the results of the validation procedure for RQ3, described in section 3.3.3. We found that about 69% of the modifications of methods are true positives if we consider them as proxies of modifications performed to the GUI. Hence, we can consider that modifications in the GUI of the AUT are involved in the majority of the modifications to test methods and classes.

Considering fragility at class level – i.e. classes containing at least a fragile method –, we found 21 such classes, and hence 21 true positives among 30 samples (70%). Such classification performance is comparable to that achieved with widely adopted approaches for fix-commit identification [4].

In 70% of the samples analyzed, a modification in a test method (or class) corresponds to an actual GUI test fragility being addressed.

Table 5: Precision for Fragile Methods and Classes

Metric	Measured	TP	FP	P
Fragile Methods	65	45	20	69%
Fragile Classes	30	21	9	70%

5 THREATS TO VALIDITY

Threats to internal validity. The test class identification process is based on the search of the name of the tool as keyword: any file containing one of such keyword is considered as a test file without further inspection; this procedure may miss some test classes, or consider a file as a test file mistakenly. The number of tagged releases is used as a criterion to identify a project as worth to be investigated; it is not assured that this check is the most dependable one for pruning negligible projects. The release level has been selected as the granularity of our inspections. We have considered that the commit level would have been an excessively fine granularity, taking in consideration very small and/or temporary modifications, and non-relevant dynamics. However, average metrics computed on the release level can be slightly different from the ones computed on the commit level, and it may be the case that changes between releases cancel each other out. The scripts and tools we used assume that no syntactic errors are present inside the test classes on which they operate, and that the names of those files are properly spelled (e.g., without the presence of special characters or blank spaces); the correctness of the metric extraction technique is not assured in different circumstances.

Threats to external validity. Our findings are based only on the GitHub open-source project repository. Even though it is a very large repository, it is not assured that such findings can be generalized to closed-source Android applications, neither to ones taken from different repositories. The applications we extracted are not necessarily released to final users. Nevertheless, we selected a subset of projects that were released on the Play Store, and the average metrics computed on them were not significantly different from the ones computed on the whole sample. We have collected measures for six scripted GUI automated testing tools. It is not certain that such selection is representative of other categories of testing tools or different tools of the same category, which may exhibit different trends of fragilities throughout the history of the projects featuring them.

Threats to construct validity. We link the GUI test fragility to any change in the interface that requires an adaptation of the test. The proxy we used - a change in any test method - is not perfectly linked to a change in the GUI. The magnitude of this threat has been evaluated with a Precision measure equal to 70%. This might reduce our fragility estimate but not change its order of magnitude.

6 CONCLUSION AND FUTURE WORK

In this paper we aimed at taking a snapshot of the usage of automated GUI testing frameworks in the Android ecosystem. We analyzed the use of some of the most important tools – Espresso, UI Automator, Selendroid, Robotium, Robolectric, and Appium – in the projects hosted on the GitHub portal.

The level of adoption of any GUI testing framework is about the 8% of the Android projects having at least one tagged release. This value can be compared to the 20% diffusion for the JUnit framework in the same context. Overall, automated GUI testing is not widely adopted. This result is slightly lower of the one about the F-Droid repository by Kochar et al. [18], who found that only 14% of apps contained test classes, and only 9% of apps had executable test classes. On average, when present, the GUI testing code represent about 11% of the whole project code.

Concerning the evolution of test code, in each release, on average, about 7.5% of the changed lines are in the GUI test code and about 3% of test code is modified.

The fragility of the tests can be estimated with two metrics based on the raw count of classes and methods modified.

Overall we can estimate fragility of the analyzed test classes around 8% (meaning that there is such probability that a test class may include a modified test method). On average, one out of five classes in each test suite needs modifications in its code because of fragilities. The association between modified test methods and fragility has been proved dependable in 70% of the samples examined. These results show that developers need rather frequently to adapt their GUI scripted testing suites, and suggest that state of the art tools should profit of additional features reducing the amount of effort needed by users to keep their script up to date and running. The results can also be used as a benchmark for practitioners and developers.

Based on these evaluations and insights about the fragility issue, we plan as future work to define a taxonomy of the causes of fragilities, produce a set of actionable guidelines to help developers avoiding them, and finally develop automated tools capable of adapting the test classes and methods to modificatons made in the user interfaces. An extension of the study to other databases of open-source projects, to take into account different testing frameworks or to other software platforms (like iOS) is also possible.

Acknowledgment. This work was supported by a fellowship from TIM.

REFERENCES

[1] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Gennaro Imparato. 2012. A toolset for GUI testing of Android applications. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 650–653.

[2] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. 2012. Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 258–261.

[3] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M Memon. 2015. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Software* 32, 5 (2015), 53–59.

[4] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2008. Is It a Bug or an Enhancement?: A Text-based Approach to Classify Change Requests. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds (CASCON '08)*. ACM, New York, NY, USA, Article 23, 15 pages.

[5] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided gui testing of android apps with minimal restart and approximate learning. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 623–640.

[6] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for android: Are we there yet?(e). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 429–440.

[7] Riccardo Coppola, Maurizio Morisio, and Marco Torchiano. 2017. Testing Fragility Data. Figshare. https://figshare.com/articles/Testing_Fragility_data/4595362. (2017).

[8] Riccardo Coppola, Emanuele Raffero, and Marco Torchiano. 2016. Automated mobile UI test fragility: an exploratory assessment study on Android. In *Proceedings of the 2nd International Workshop on User Interface Test Automation*. ACM, 11–20.

[9] Jerry Gao, Xiaoying Bai, Wei Tek Tsai, and Tadahiro Uehara. 2014. Mobile application testing. *Computer* 47, 2 (2014), 46–55.

[10] V. Garousi and M. Felderer. 2016. Developing, Verifying, and Maintaining High-Quality Automated Test Scripts. *IEEE Software* 33, 3 (May 2016), 68–75.

[11] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. 2013. Reran: Timing-and touch-sensitive record and replay for android. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 72–81.

[12] Robi Grgurina, Goran Brestovac, and Tihana Galinac Grbac. 2011. Development environment for Android application development: An experience report. In *MIPRO, 2011 Proceedings of the 34th International Convention*. IEEE, 1693–1698.

[13] Casper S Jensen, Mukul R Prasad, and Anders Møller. 2013. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 67–77.

[14] Jouko Kaasila, Denzil Ferreira, Vassilis Kostakos, and Timo Ojala. 2012. Testdroid: automated remote UI testing on Android. In *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia*. ACM, 28.

[15] Anureet Kaur. 2015. Review of Mobile Applications Testing with Automated Techniques. *interface* 4, 10 (2015).

[16] B. Kirubakaran and V. Karthikeyani. 2013. Mobile application testing #x2014; Challenges and solution approach through automation. In *2013 International Conference on Pattern Recognition, Informatics and Mobile Engineering*. 79–84.

[17] Thomas W Knych and Ashwin Baliga. 2014. Android application development and testability. In *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems*. ACM, 37–40.

[18] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo. 2015. Understanding the Test Automation Culture of App Developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 1–10.

[19] Martin Kropp and Pamela Morales. 2010. Automated GUI testing on the Android platform. *on Testing Software and Systems: Short Papers* (2010), 67.

[20] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. 2013. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution.. In *WCRE*. 272–281.

[21] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. 2014. Visual vs. DOM-based web locators: An empirical study. In *International Conference on Web Engineering*. Springer, 322–340.

[22] Mario Linares-Vásquez. 2015. Enabling testing of android apps. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 763–765.

[23] Mario Linares-Vásquez, Christopher Vendome, Qi Luo, and Denys Poshyvanyk. 2015. How developers detect and fix performance bottlenecks in android apps. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. IEEE, 352–361.

[24] Chien Hung Liu, Chien Yu Lu, Shan Jen Cheng, Koan Yuh Chang, Yung Chia Hsiao, and Weng Ming Chu. 2014. Capture-replay testing for Android applications. In *Computer, Consumer and Control (IS3C), 2014 International Symposium on*. IEEE, 1129–1132.

[25] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 224–234.

[26] Diego Torres Milano. 2011. *Android application testing guide*. Packt Publishing Ltd.

[27] Nariman Mirzaei, Sam Malek, Corina S Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. 2012. Testing android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–5.

[28] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2017. Crashscope: A practical tool for automated testing of android applications. In *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 15–18.

[29] H. Muccini, A. Di Francesco, and P. Esposito. 2012. Software testing of mobile applications: Challenges and future research directions. In *2012 7th International Workshop on Automation of Software Test (AST)*. 29–35.

[30] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. 2012. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th International Symposium*

on the Foundations of Software Engineering. ACM, 33.
[31] Gaurang Shah, Prayag Shah, and Rishikesh Muchhala. 2014. Software testing automation using appium. *International Journal of Current Engineering and Technology* 4, 5 (2014), 3528–3531.
[32] Shiwangi Singh, Rucha Gadgil, and Ayushi Chudgor. 2014. Automated Testing of mobile applications using scripting Technique: A study on Appium. *International Journal of Current Engineering and Technology (IJCET)* 4, 5 (2014), 3627–3630.
[33] Ming-xin TAN and Pei CHENG. 2016. Research and implementation of automated testing framework based on Android. *Information Technology* 5 (2016), 035.
[34] Xinye Tang, Song Wang, and Ke Mao. 2015. Will This Bug-Fixing Change Break Regression Testing?. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–10.
[35] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.
[36] Wei Yang, Mukul R Prasad, and Tao Xie. 2013. A grey-box approach for automated GUI-model generation of mobile applications. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 250–265.
[37] Vahid Garousi Yusifoğlu, Yasaman Amannejad, and Aysu Betin Can. 2015. Software test-code engineering: A systematic mapping. *Information and Software Technology* 58 (2015), 123–147.
[38] Hrushikesh Zadgaonkar. 2013. *Robotium Automated Testing for Android*. Packt Publishing Ltd.
[39] Yury Zhauniarovich, Anton Philippov, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci. 2015. Towards Black Box Testing of Android Apps. In *Availability, Reliability and Security (ARES), 2015 10th International Conference on*. IEEE, 501–510.

delegate copy

Code Authorship and Fault-proneness of Open-Source Android Applications : An Empirical Study

John Businge
Mbarara University of Science and Technology
P.O. Box 1410
Mbarara, Uganda
johnxu21@gmail.com

Simon Kawuma
Mbarara University of Science and Technology
P.O. Box 1410
Mbarara, Uganda
simon.kawuma@must.ac.ug

Engineer Bainomugisha
Makerere University
P.O. Box 7106
Kampala, Uganda
baino@cis.mak.ac.ug

Foutse Khomh
SWAT Lab., École Polytechnique de Montréal
Montréal, Canada
foutse.khomh@polymtl.ca

Evarist Nabaasa
Mbarara University of Science and Technology
P.O. Box 1410
Mbarara, Uganda
enabaasa@must.ac.ug

ABSTRACT

Context: In recent years, many research studies have shown how human factors play a significant role in the quality of software components. Code authorship metrics have been introduced to establish a chain of responsibility and simplify management when assigning tasks in large and distributed software development teams. Researchers have investigated the relationship between code authorship metrics and fault occurrences in software systems. However, we have observed that these studies have only been carried on large software systems having hundreds to thousands of contributors. In our preliminary investigations on Android applications that are considered to be relatively small, we observed that applications systems are not totally owned by a single developer (as one could expect) and that cases of no clear authorship also exist like in large systems. To this end, we do believe that the Android applications could face the same challenges faced by large software systems and could also benefit from such studies.
Goal: We investigate the extent to which the findings obtained on large software systems applies to Android applications.
Approach: Building on the designs of previous studies, we analyze 278 Android applications carefully selected from GitHub. We extract code authorship metrics from the applications and examine the relationship between code authorship metrics and faults using statistical modeling.
Results: Our analyses confirm most of the previous findings, i.e., Android applications with higher levels of code authorship among contributors experience fewer faults.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PROMISE , November 8, 2017, Toronto, Canada
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5305-2/17/11…$15.00
https://doi.org/10.1145/3127005.3127009

KEYWORDS

Software faults; Minor Contributors; Major Contributors; Most Values Contributors; Total Contributors

ACM Reference format:
John Businge, Simon Kawuma, Engineer Bainomugisha, Foutse Khomh, and Evarist Nabaasa. 2017. Code Authorship and Fault-proneness of Open-Source Android Applications : An Empirical Study. In *Proceedings of PROMISE , Toronto, Canada, November 8, 2017,* 10 pages.
https://doi.org/10.1145/3127005.3127009

1 INTRODUCTION AND MOTIVATION

In recent years, a number of studies have shown that human factors play a significant role in software quality [2, 3, 7–9, 12, 14, 15, 19–21]. Code authorship metrics were introduced by Bird et al. [3] to capture developers' contributions in large and distributed software development teams, with the aim to establish a clear chain of responsibility (who to blame in case there is a problem) and simplify management (to whom to assign a task or a bug-fix). Using code authorship metrics, researchers have investigated how different levels of developers' activities on components affect the quality of a software. For example, Bird et al. [3] found that a module with weak code authorship (i.e., that is written by many minor authors) is more likely to have faults in the future. Bird et al. used code authorship metrics to split developers of a software component into two distinct groups: major developers, corresponding to people who authored more than 5% of the contributions and minor developers who are people who authored less than 5% of the code of the component. Greiler et al. [9] argue that a lack of clear code authorship is likely to cause a lack of responsibility on the parts of the code that an engineer does not own.

Although there has been some research examining the relationship between code authorship metrics and the quality of software components, all these studies have been performed on a few (maximum of ten) medium to large sized systems, that were developed by hundreds of developers. We are not aware of any study that investigated code authorship in small sized systems, such as mobile applications. Yet, our preliminary analysis of Android applications,

revealed that most applications are not totally owned by a single developer (as one could expect) and that cases of no clear authorship also exist like it was reported for large systems (see Bird et al. [3]). While the findings reported by these previous works are actionable in large software systems, the role of (a lack of) clear code authorship is still unclear for small sized systems. We do believe that relatively small software systems like Android applications could face similar challenges as large software systems and could also possibly benefit from the findings of code authorship studies. For example, Bird et al. [3] observed that high values of minor contributors are associated with more faults in a software system. In this paper, in the context of Android applications, *we investigate whether applications with few major contributors are more–or–less fault-prone than applications with larger numbers of developers that do minor contributions.*

The remainder of this paper is organized as follows: Section 2 presents the background information. Section 3 discusses the experimental setup of our study. Section 4 discusses the results and findings of our study. Section 5 presents threats to the validity, while Section 6 provides an overview of the related work. Finally, Section 7 concludes the paper and outlines some avenues for future work.

2 BACKGROUND & DEFINITIONS

In this section, we give a brief overview of code authorship metrics and explain how previous studies related the metrics to software quality.

2.1 Code Authorship Metrics and Software Quality

Previous studies used the authorship of code changes to estimate the code authorship of a developer for a module [3, 7, 20]. Bird et al. [3] computed a developer's code authorship for a module by calculating the proportion of code changes that the developer has authored within that module. Bird et al. showed that weakly owned Windows binaries where many engineers contributed small amounts of code were more likely to be fault-prone than strongly owned Windows binaries, for both Windows Vista and Windows 7. Furthermore, Bird et al. also observed that the more minor code authors contributed to a software module, the more faults it contained. Rahman and Devanbu [20] computed code authorship values at a finer level of granularity, by calculating the proportion of changed lines that each developer has authored. Rahman and Devanbu observed that code implicated in faults were strongly associated with situations of single developer's contribution. Foucault et al. [7] revisited the theory formulated by Rahman and Devanbu [20] on seven open source software systems, using code authorship metrics proposed by Bird et al. [3] and confirmed the existence of a relationship between code authorship and software quality. However, they also argue that the usefulness of code authorship metrics is debatable since in all their studied systems, they found independent variables to be highly collinear. The main difference between these previous studies and our study is the size of the systems that are investigated, i.e., large vs. small sized systems.

In our study, we use the metrics proposed by Bird et al. [3], i.e., we use the amount of code changes to estimate the code authorship of a developer for an application. Below we state four code authorship metrics that capture the magnitude of the contributions of developers in the applications and one metric that captures software quality. We also provide the rationale for choosing these metrics to assess the potential relation between code authorship and the fault-proneness of applications.

Most valued author (MVA): This metric measures the highest percentage of contributions that a code author has made to a software module. If the MVA of a software project is close to 100%, this means that one author performed almost all the changes in that project. A high value of MVA reveals that the software project has strong authorship while a low value reveals that it has a shared authorship. We expect that an increase in the value of MVA metric may result in a decrease of the number of faults in the project.

Minor Author (Minor-A): The metric counts how many code authors have a ratio of contributions that is lower than a given threshold. If there are lots of minor authors, this implicitly means that many contributions are made by minor contributors and therefore the software project is shared between many code authors. The work is thus fragmented between many code authors with little knowledge of the project they are working on, and therefore overseeing all these contributions becomes an obstacle. We expect that an increase in the value of the Minor-A metric may result in an increase of the number of faults in the project.

Major Author (Major-A): The metric counts how many code authors have a ratio of contributions that is bigger than a given threshold. We rely on Bird et al's. [3] analogy stating that "too many cooks spoil the broth". This means that if there are lots of major authors, they all perform a significant amount of contributions and therefore the software project has a shared authorship which implies that coordinating the work of developers is more difficult. We expect that an increase in the value of the Major-A metric to result in an increase of the number of faults in the project.

Total Authors (Total-A): This metric counts the total number of contributors that have made changes in a software project. It examines the effect of team size on software quality. If the size of a team is too large, coordination may be difficult, which may result in poor quality and faulty code.

Faults: This is the number of faults reported in a given Android project.

Later in our analysis, we will build regression models to identify the relationship between faults and code authorship metrics in the Android applications.

3 EXPERIMENTAL SETUP

This section details the experimental setup. We also define the research goal and questions, descriptions of how we collected the data and how we extracted the metrics.

3.1 Goal & Research Questions

Like the study by Bird et al. [3] we also adopt the Basili's goal question metric approach [1] to frame our study of code authorship and fault-proneness. Our goal is to understand the relationship between code authorship and fault-proneness in Android applications (which are relatively small software systems). In order to reach this goal, we ask two research questions:

RQ1: Are Android applications with *higher values of Minor-A* associated with more *faults* in comparison to those with *lower values of Minor-A*?

RQ2: Are Android applications with *higher values of MVA* less fault-prone than applications with *lower values of MVA*?

To answer the two research questions, we use regression models on a dataset collected from Android open-source applications hosted on Github. Below we present the methodology that was employed to process the dataset. We present the corpus of applications used, as well as how we computed the different metrics used in the models.

3.2 Data Collection and Extraction of Code Authorship and Control Metrics

3.2.1 Data Collection. The data used in the study was carefully extracted from GitHub using the GitHub API. The selection of the GitHub data was based on the following criteria: First, in early March, 2016 when we collected the data, we typed the keyword *Android applications* in the GitHub search engine, which returned about 45,000 repositories. About 40,000 of these were written in the *Java* language. Because we are aware that some of the projects on GitHub are for example written by students as assignments, we wanted to eliminate these repositories as much as we could so as not to pollute our results. For this reason, we carefully collected *repository names* on GitHub with a "*description*" containing the word "*Android application*", having *at least five releases, at least three stars* and written in *Java*. Starring a repository allows a developer to keep track of projects that he finds interesting as well as showing appreciation to the repository maintainer for their work[1]. The rationale for using three stars was because we wanted to get as many applications as possible, for statistical significance.

To compute the code authorship metrics, we processed the different JSON objects returned by the Github API. As an illustration example, we use the Android application project `eglaysher/life-counter`. The Github API `https://api.github.com/repos/eglaysher/lifecounter/commits` returned a JSON object with all the commits of this project. Each commit has got a unique 40 character identifier called `SHA`. For each commit SHA, we use the Github API, for example `https://api.github.com/repos/eglaysher/lifecounter/commits/a807848a5d426aabea59dae4b355706e60228e7a` which returns a JSON object with specific commit details that include: author details (login ID, email address, and full names), all the files that have been modified as well as the number of lines of code (LOC) in each file that have been modified. For each file, we summed up the changed LOC. We thereafter summed up the changed LOC for all the files in a commit. Additionally, for each commit, we also kept track of the author details for the commits. Finally, we summed up the changed LOC of all the commits in a project made by different authors. We also collected other statistics like `longevity`, and `inactivity` to help us summarize the studied projects using descriptive statistics in Section 4.1. `Longevity`–is the time interval in months between the application's "first release date" to the "last commit date" or March 06, 2016 (the last day of collection of the application statistics on GitHub). `Inactivity`—The time interval in months between the application last commit and March 06, 2016. Because we want to build meaningful models, in our data sets we only used projects that have existed for at least one year since we collected this data (i.e., first release dates before March 06, 2015). Using the aforementioned criteria, we retained a total of 278 applications for our study. We share our data set on-line[2], to allow the community to replicate our work.

While collecting the data from GitHub, we observed that some of the applications received a lot of commits from certain contributors in their first release, but no commits from these contributors later on. This possibly means that the development of the application started elsewhere and it was just imported into GitHub. Our reasoning is that, the more contributors an application has after its first release, the higher the likelihood that the application will be maintained frequently on GitHub. To compute code authorship metrics for a given application, we collect information about the commits of all the developers that have contributed to the application *after the first release* of the application until *March, 6 2016*. We also downloaded the source files of the applications from Github and extracted the number of Lines of source code (LOC), using the *cloc*[3] tool.

3.2.2 Name Merging. During data collection, we discovered that some contributors of the applications used more than one account, which makes them appear as different contributors. To address this issue, we performed name merging to ensure that our data is not polluted with duplicate informations that would introduce noise. We merged the details of two contributors into one using the following heuristics in the order mentioned: 1) if they possess the same `login ID`, 2) posses different `login ID` but posses the same `full names`, and 3) possess both different `login ID` and `full names` but have the same `e-mail` prefix.

3.2.3 Code Authorship and Control Metric Extraction. We computed code authorship metric values following the definition proposed by Bird *et al.* [3] and used by Rahman and Devanbu [20], which consists in calculating the proportion of contribution of each author. If C^b lines are changed on a repository rp_1 in a time interval t, and there are a total number of m distinct authors, and the number of lines contributed by author a in the time interval t is C^b_t, then the contribution ratio of a is $r^C_a = \frac{C^b_t}{C^b}$. We then sort the ratios in descending order and thereafter we sum them up as illustrated in Equation 1. $sum_{0.8}$ is the summation of the ratios starting with the largest ratio r^C_1 (highest ratio) to r^C_n, where r^C_n is the first ratio where $sum_{0.8} \geqslant 0.8$ and $n \leqslant m$.

Bird *et al.* who studied large scale software systems (i.e., Windows Vista and Windows 7) with nearly one hundred code authors, used a threshold value of 5% to categorize major and minor code

[1] https://help.github.com/articles/about-stars/

[2] https://sites.google.com/site/coauthorship2017/dataset

[3] https://github.com/AlDanial/cloc

contributors (i.e., they considered an author to be a major contributor of a module if they contributed more than 5% of the code of that module, otherwise they are considered to be a minor contributor to the module). Since our work focuses on applications, which are of smaller size, and have fewer developers than these Windows projects, computed differently the threshold value used to decide about major and minor contributors.

A threshold of 5% would be inappropriate because, for example, if an application has two code authors with contribution ratios of 0.93 and 0.07, respectively, applying the threshold of 5% used by Bird et al., would characterize both authors as major contributors, which would be misleading. In our study, using the heuristics presented in Equation 1, we label major authors if they have been categorized in $sum_{0.8}$ and minor authors in the remaining $sum_{0.2}$. In terms of thresholds values, we state that the labeling of major and minor authors is considered at threshold–0.8. To compute the `MVA` metric, we consider the author with the highest ratio. For example, if an application has two code authors having contribution ratios of 0.93 and 0.07, the `MVA` value will be 0.93. However, as shown in Figure 1 (in Section 4.1), we do have cases of low MVA metric meaning that there are cases of shared-authorship of the applications.

$$sum_{0.8} = \sum_{i=1}^{n} r_i \tag{1}$$

We carried out a *sensitivity analysis* on the experiments by varying the thresholds from 0.7, 0.75, 0.8, 0.85 and 0.9 but the results did not yield significant differences. We decided to use the threshold of 0.8.

Also, we would like to emphasize that we computed all the aforementioned metrics at the project level, i.e., we computed the authorship (respectively the level of contribution) of a developer for the whole application. A detailed explanation of this design decision (i.e., project level vs. module level analysis) can be found at the end of Section 3.3.

3.3 Data Collection and Extraction of Code Quality Metrics

We used the `SonarQube` tool[4] to extract information about faults and security vulnerability experienced by the studied applications. `SonarQube` (previously called Sonar) is an open source quality management platform, dedicated to continuously analyze and measure technical quality, from project portfolio to methods. `SonarQube` is a popular code quality measurement tool that has gone through a number evolutionary versions having its first version released in 2007. The tool is actively maintained on Github as of January 16, 2016 having 21,739 `commits`, 107 `releases`, 55 `contributors`, 160 `watches`, 1,461 `stars`, and 566 `forks`[5]. All the statistics of `SonaQube` on Github show that the tool is very popular among developers and is actively being maintained. The `SonarQube` tool analyzed the files in each application looking for faults, and reported specific points in the file where a fault was observed. The tool categorizes the identified faults into five types: blocker, critical, major, minor, and info.

- *Blocker*: A fault of this kind might make the whole application unstable in production. For example, calling garbage collector, not closing a socket, etc.
- *Critical*: A fault of this kind might lead to an unexpected behavior in production without impacting the integrity of the whole application. For example, NullPointerException, badly caught exceptions.
- *Major*: A fault of this kind might have a substantial impact on productivity. For example, too complex methods, package cycles.
- *Minor*: A fault of this kind might have a potential but minor impact on productivity. For example, finalizer does nothing but call superclass finalizer.
- *Info*: Unknown or not yet well defined security risk which can impact productivity.

The tool counts the number of faults reported in each file and aggregates them to obtain the total number of faults contained in the project. Additionally, the tool further rates the fault-proneness of the whole application as follows: A–Zero faults, B–at least one minor fault, C–at least one major fault, D–at least one critical fault and E–at least one blocker fault. For our dependent variable metric, we consider the total number of faults reported for each of the Android application. We extracted the faults reported on the last release of each of the studied applications.

We would like to state that our study differs slightly from the previous studies of Foucault et al. [7] and Bird et al. [3] on the artifact that was considered. While the previous studies investigate the relationship between code authorship metrics and fault-proneness in a module, our study investigates the relationship between code authorship metrics and fault-proneness in a project. Software modules are units of development within a software project for example file or package. There are two main reasons for the above stated difference in the investigated artifact in the software system: 1) As earlier stated, unlike in the previous studies with relatively large software systems having hundreds of developers, it made sense extracting code authorship metrics and building module-level models. However, since we are investigating Android applications that are much smaller as well as having few authors, extracting code authorship metrics and building for example file-level models would not make a lot of sense. 2) Again, because of the large sizes as well as the very few number of the software systems investigated in the previous studies (i.e., Bird et al.–two software systems and Foucault et al.–seven software systems), models were built for each software system. For example, the data points of the independent variables in the models are the number of code authorship metrics per software module in a software system (if a software system has 100 software modules, then 100 code authorship metrics data points were extracted). As opposed to the way models were built in previous studies, in this study because of the limited size and number of authors in the Android applications, we build models where we consider each application as a data point. Another difference with previous works concerns the faults investigated. We rely on SonarQube to identify faults in the code of applications while previous works extracted faults reported in bug tracking systems.

[4]https://www.sonarqube.org/
[5]https://github.com/SonarSource/sonarqube

3.4 Multiple Linear Regression Models Tuning

In this section, we discuss how we build multiple linear regression models to uncover the relationship between faults and code authorship metrics. Similar to previous related works [3, 4, 14, 21, 23], our main goal for building fault-proneness models is not to predict fault-prone applications, but to understand the relationship between the explanatory variables and the fault-proneness of the applications. Specifically, we use linear regression to enable us to examine the effect of one or more code authorship metrics and source code metrics (when controlling the other variables). In the regression models with faults as the dependent variable, one can observe which variables have an effect on faults, how large the effect is, in what direction (i.e., if number of faults go up when a metric goes up or when it goes down), and how much of the variance in the number of failures is explained by the metrics. We compare the amount of variance in failures explained by a model that includes the code authorship metrics to a model that does not include them. In the models, we use size (i.e., LOC) and complexity metrics (Mc Cabe complexity[6]) as control variables.

Before building the models, we conducted a number of model tuning and preparations in order to have model results that can be trusted. The following are some of the diagnostics that we carried out: First, we standardized the variables by subtracting the values from the mean and dividing by the standard deviation. This allowed our variables to be relatively on the same scale (which is very important since we are building models across different applications). Second, because the interpretation of the models' results can be influenced by the presence of redundant variables. We checked for redundant variables using the `redun` function in the `rms` `R` package [11]. However, we found that none of the explanatory variables that survived our correlation analysis were redundant. Third, we also removed the variables that introduced multicollinearity and over-fitting by considering the Variance Inflation Factor (VIF). All variables in the final models had VIFs of under 5, as guided by standard rule of thumb [17]. Fourth, for the ordinary least squares (OLS) regression to be reliably interpreted, we had to look at normally distribution of the residuals. Non-normality of the residuals is attributable to the skewness of the variables. In our data set, variables that are found to be skewed are log transformed to stabilize the variance and improve the model fit, whenever appropriate [17]. Fifth, to overcome the issue of the ordering of regressors, we assess and report the relative importance of regressors in the multiple regression model using the PMVD technique developed by Feldman [6], which is implemented in the R package relimpo [10]. Finally, we take special care to make sure that the most important modeling assumptions of OLS regression are met, namely: 1) homoscedasticity (by examining *residual vs. fitted* plots), 2) linear independence (mentioned above, by removing highly correlated variables according to VIF), and 3) normality of errors (by examining *normal qq plots*). In addition, we also consult the *Cook's distance vs. leverage plots* to identify any potentially overly influential outliers to examine for validity. This resulted in the removal of six points in our data set which improved the model fit (based on R^2 value) while having minimal effect on the estimated model coefficients (*i.e.,* no variable coefficients changed signs or significance).

[6]https://docs.sonarqube.org/display/SONAR/Metrics+-+Complexity

Table 1 – Descriptive statistics of the study variables.

Variable	Mean	Min	1st Quartile	Median	3rd Quartile	Max
MVA	82%	20%	66%	95%	100%	100%
Total-A	12.3	1	2	3	8	465
Minor-A	10.5	0	0.25	2	6	457
Major-A	1.8	1	1	1	2	16
faults	71.8	0	11.3	32.5	78.5	679
Complexity	2053.1	11	350.8	889.0	2397.8	20511
SizeF	5680.6	14	719.5	2194.5	5601.3	82655
SizeL	9604.6	67	1927.25	4437	10928.5	84265
Longevity	24.4	0.7	14	22.8	34.675	79.2
Inactivity	10.1	0	0.525	4.7	16.425	67.6
Cd'LOC	184160.5	15	8071	24459	95337.5	10331255

faults—Number of faults.
SizeF—Size in Lines of Code of the first release of a project.
SizeL—Size in Lines of Code of the last release of a project.
Cd'LOC—Added + Deleted LOC.

To build the models, we follow a traditional hierarchical approach where we start with a base model that contains only control factors. In subsequent models, we add the various independent measures associated with code authorship metrics. This modeling approach allows us to understand the independent and relative impact on faults, of each set of factors. In order to assess the fit of each model, we report the percentage of variance explained by the model (commonly referred to as the R-squared). We examine the improvement in percentage of variance in the dependent variable explained when we add code authorship metrics to the base model.

4 RESULTS AND DISCUSSION

In this section, we present the results of our experiments. As shall be seen in this section, in our analysis we used a number of methods to examine the relationship between code authorship and the fault-proneness of applications.

4.1 Descriptive Statistics

The first step in our analysis consisted of examining various descriptive statistics of the measures described earlier. Table 1 presents the descriptive statistics of the variables used in the models as well as other variables describing the studied applications. One can draw a number of insights from these descriptive statistics, about the common characteristics exhibited by the studied applications. First, looking at the column values of the *Median* and *Mean* for the variables `Size-F` and `Size-L`, we observe that these are small software systems; as the values are below 10,000 LOC. Second, looking at values of the variables `Size-F` and `Size-L` in the *3rd Quartile* column we also observe that very few applications have a size above 10,000 LOC. Third, looking at values in the column of the *3rd Quartile* for the variables `Total-A`, the values of the column *Median* for `Major-A`, and `Minor-A`, we observe that the software applications considered, indeed comprise few contributors. Although the values of code authorship metrics are low (expected for relatively small systems), studying such a situation is still important in the sense that there are still authors with low contributions, hence with limited knowledge of certain parts of the applications.

Table 2 – Spearman correlation between the different variables used in the study.

Variable	MVA	Minor-A	Major-A	Total-A		SizeF	SizeL	faults	Complexity	Cd'LOC
MVA	1.00	-0.33	-0.75	-0.33		-0.15	-0.22	-0.23	-0.21	-0.19
Minor-A	-0.33	1.00	0.46	0.68		0.31	0.28	0.34	0.28	0.32
Major-A	-0.75	0.46	1.00	0.59		0.11	0.13	0.17	0.12	0.33
Total-A	-0.33	0.68	0.59	1.00		0.19	0.19	0.17	0.18	0.44
SizeF	-0.15	0.31	0.11	0.19		1.00	0.78	0.43	0.80	0.10
SizeL	-0.22	0.28	0.13	0.19		0.78	1.00	0.56	0.99	0.09
faults	-0.23	0.34	0.17	0.17		0.43	0.56	1.00	0.54	0.19
Complexity	-0.21	0.28	0.12	0.18		0.80	0.99	0.54	1.00	0.09
Cd'LOC	-0.19	0.32	0.33	0.44		0.10	0.09	0.19	0.09	1.00

These low contributors could introduce regressions in the applications if they modify areas of the code for which they have little knowledge. Fourth, comparing the values of size (SizeF and SizeL) and Cd'LOC, gives us an indication that the applications have gone through multiple releases along their lifetime, which reduces the possibility that they are student class assignments and not real products. Fifth, looking at the values of variables of `Longevity` and `Inactivity`, we can also observe that the applications have been in existence for a fairly good amount of months. Lastly, looking at the column values of Min, 1st Quartile, Median, 3rd Quartile and Max for all other variables apart from MVA, we observe that the distribution of the variables is heavily rightly skewed. We confirmed this observation on the distribution by drawing box-plots (cf. Figure 1) using the R software. We also observe that the values of code authorship obtained on our studied apps at the project level is similar to the values of code authorship reported by Greiler et al. [9] for the files contained in the four Microsoft projects. This similarity reinforced our decision to conduct our study at the project level instead of component or file levels.

4.2 Correlations

Here we computed Spearman rank correlations presented in Table 2. From the table, we observe that the metric MVA is negatively correlated with most of the other metrics including the number of faults. This implies that the more the code authorship is shared among multiple developers of an application, the higher is the likelihood that the application will contain faults. This observation is reinforced by the fact that for all the studied applications, Total-A and Minor-A metrics are positively correlated with the number of faults (the more people are involved in the development of an application, the higher is the risk of faults). Looking at the individual trends of relationships between code authorship metrics and number of faults, we observe that Minor-A–(0.34) correlates highest with the number of faults, followed by MVA–(-0.23), Major-A, and Total-A–(0.17). Additionally, we also observe high correlations between code attributes like SizeF, SizeL, and complexity with number of faults. The high correlations of size and complexity is not surprising since previous studies have shown that the two variables have a very strong correlation with fault-proneness [5].

Considering the correlations of both code authorship metrics and those of code attributes, as discussed by Bird et al. [3], it is not clear if the increase in number of faults in the applications is attributable to more Minor-A or to measures such as size and

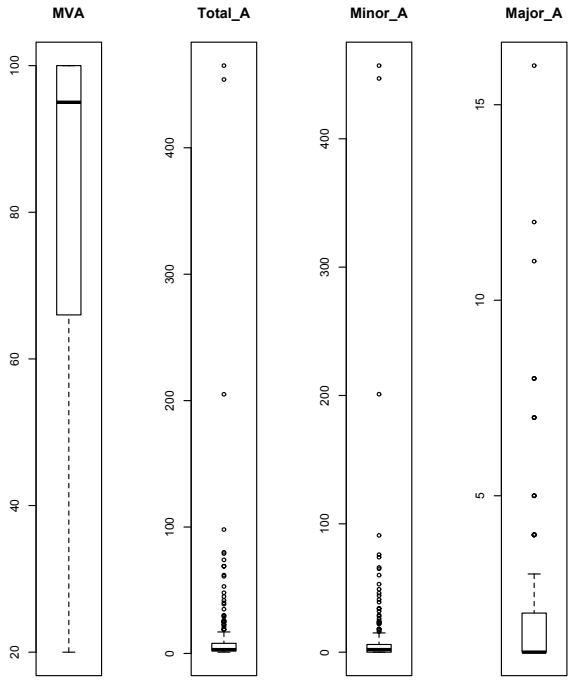


Fig. 1 – Box plots showing the distribution of the code authorship metrics used in the study.

complexity that are also known to be related to faults. To clear this dilemma, we build models of multiple linear regression to observe the effect of each metric on the number of faults when the code authorship and code attribute metrics are used together. Furthermore, prior research has shown that when characteristics such as size are not considered, the omission can affect the validity of observations made for other software metrics [5]. In the next section, we discuss how we used regression models to overcome the anticipated problem. Furthermore, looking at the correlations from Table 2, we observe high correlations between number of `faults` and `SizeL` (> 0.5). In building the models in the next section, we decided to use SizeF instead of SizeL as the control variable since using SizeL introduced over-fitting in the models.

Table 3 – Linear Models coefficients and the sum of squares (ANOVA) for the different variable combinations. The models include standard metrics of size and complexity, as well as the models with code authorship metrics added. An asterisk* in column–*Variance* denotes that a model showed statistically significant improvement when the additional variable was added. Significance codes: 0.000 (***), 0.001 (**), 0.01 (*), 0.05 (.), >= 0.1 ()

Model	Description	Variable	Coeff	Pr(> |t|)	Sum Sq	Importance	Variance
Base Model							
1	Base	SizeF	0.049	0.473	1.258***	29.6%	39.5%
		Complexity	0.121	0.000***	0.876***	70.4%	
Base Model + code authorship Heuristics							
2	Base + MVA	SizeF	0.053	0.426	1.129***	26.9%	42.0% (+2.5%)
		Complexity	0.115	0.000***	0.777***	63.0%	
		MVA	-0.023	0.000***	0.364***	10.1%	
3	Base + Total-A	SizeF	0.041	0.538	1.258***	26.5%	41.3%* (+1.8%)
		Complexity	0.115	0.000***	0.780***	63.4%	
		Total-A	0.056	0.005**	0.193***	10.1%	
4	Base + Minor-A	SizeF	0.033	0.619	1.258***	23.5%	43.8%* (+4.3%)
		Complexity	0.109	0.000***	0.680***	56.1%	
		Minor-A	0.056	0.000***	0.429***	20.4%	
5	Base + Minor-A + Major-A	SizeF	0.034	0.606	1.258***	23.0%	44.0%* (+0.2%)
		Complexity	0.109	0.000***	0.679***	55.1%	
		Minor-A	0.048	0.001**	0.429***	16.9%	
		Major-A	0.005	0.369	0.008	5.0%	

4.3 OLS Model Results

In this section, we present the results of the multivariate linear regression analysis for the 5 models we built in our experiments. Table 3 illustrates the results of our analysis. The table presents the values of the different constructs in the models that we built, which includes: 1) *Model*–the models comprising different variable combinations. 2) *Description*–how the models were incrementally built, 3) *Variable*–the different variable combinations considered in each of the models. 4) *Coeff* –the values of the regression coefficients corresponding to each of the variables in the models. The values of the coefficients tell us the change in the number of faults for every unit increase in the independent variable. For example, a value of 0.049 for the variable SizeF in Model–1 tells us that the predicted number of faults in an application will increase by 0.049 for every increase of one LOC in SizeF. 5) $Pr(>|t|)$–The $p-value$ showing the statistical significance of that variable given all the other variables have been entered into the model. The significant codes indicate: (***)–$p = 0.000$, (**)–$p = 0.01$, (*)–$p = 0.05$, and (.)–$p = 0.1$. 6) *Sum Sq*–Sum of Squares associated with the sources of variance of the variables (total variance partitioned into the variance which can be explained by the independent variables) and the residual. The significance code for the *Sum Sq*, like $Pr(>|t|)$, indicate the variables that significantly contribute to the estimate of the dependent variable. 7) *Importance*–This is the relative importance for each of the predictors in the model provided by R `relaimpo` package. For example, in Model–1, Complexity has the highest importance in estimating the dependent variable with a value of 70.4%. 8) *Variance*–This measures the proportion of variance of the dependent variable (i.e., number of faults) explained by the regressors (i.e., code authorship and control metrics) in the model. For example, in Model-1, the proportion of variance explained by the regressors `SizeF` and `Complexity` is 39.5%.

Table 3, column–*Variance*, the asterisk* denotes cases where the goodness-of-fit F-test indicated that the addition of variable improved the model by a statistically significant degree. The value in parenthesis indicates the percentage of increase in variance explained over the model without the added variable. For example, in Model-5–Base + Minor-A + Major-A explains 44.0% of the variance in the number of faults which is 0.2% more than Model-4–Base + Minor-A which explains 43.8%. As stated in Section 3.4, our model building followed a traditional hierarchical approach where we started with a base model containing only control factors–SizeF and Complexity (we refer to this model as the Base model). From Table 3, (*Model-1*:*Importance*), the Base model shows that SizeF and Complexity both have significant effects on the number of faults. In addition, these metrics are able to explain 39.5% of the variance in the number of faults in the software projects we considered. Looking at the *p-values* of SizeF in the column–Pr(> |t|), we observe that they are all > 0.05. This seems to suggest that SizeF do not contribute to the model.

The reason for the insignificant contribution of SizeF in terms of *p-values* is because SizeF and Complexity are highly correlated (c.f., Table 2). However, when considered in isolation of Complexity, the contribution of SizeF on the number of faults is very significant. This can also be observed from column–Importance in Table 3. Furthermore, in order to determine the relative importance of the code authorship metrics, one needs at least two variables in the *Base model* to run the `relaimpo` package in R. In subsequent models (models 2–5), we incrementally added the various independent variables associated with the different research questions. This modeling approach allowed us to understand the importance of the different independent variables on the number of faults in the applications.

RQ1: Are Android applications with *higher values of Minor-A* associated with more *faults* in comparison to those with *lower values of Minor-A*?

From the detailed results—Table 3, we use Model-3, Model-4 and Model-5 to answer RQ1. In Model-3 we add the metric Total-A to the set of predictor variables of the base model to examine the effect of team size on fault-proneness an application. In Model-4 we add the

metric Minor-A to the set of predictor variables of the base model to examine the effect of minor contributors on the fault-proneness in an application. In Model-5 we add the metric Major-A to the variables in Model-4 to examine what effect the major contributors have on the fault-proneness of an application. We compare the results of pairs of Model-3 and Model-4 to determine if the total number of code authors has a different effect on the number of faults than the number of minor code authors in the studied applications. The statistics show that the predictor Minor-A has a higher relative importance of 20.4% compared to that of Total-A–10.1%. We also observe that the addition of the Minor-A metric increases the variance explained by 4.3%, whereas the addition of the Total-A metric increases the variance explained by 1.8% only. The gains shown by Minor-A are stronger than those shown by Total-A in estimating the number of faults in the Android applications. This indicates that the number of minor contributors in Android applications have a stronger effect on the fault-proneness of the applications, in comparison to the total number of contributors. Furthermore, the addition of Major-A in Model-5 showed smaller gains, but was still statistically significant. We left out the results of the model Base + Minor-A + Major-A + Total-A since this model produced VIF > 5. Overall, the most significant contributor to the variance explained is Minor-A, followed by Total-A, and lastly by Major-A.

Conclusion RQ1: In Section 2.1 we stated that we expected that an increase in the values of the minor Author metric may result in an increase of the number of faults. The modeling discussed above has revealed that the number of minor code authors have a strong positive relationship with the number of faults even when controlling for classical metrics such as size and complexity. This implies that Android applications with few major contributors are more reliable than applications with larger numbers of contributors where developers do minor contributions. These findings obtained on relatively small sized open source software projects (i.e., the applications) concur with the findings of Bird et al. [3] which were obtained on large commercial software projects. However, while we agree with the finding of Foucault et al. [7] (also conducted on large open source projects) that Minor-A and Total-A metrics are highly collinear, we differ in the conclusion that Minor-A is highly redundant and could be ignored. As we have discussed in our results above, we do state that although Total-A and Minor-A are collinear, we have shown that the individual contribution of each of these variables to the variance of the number of faults, is statistically significant. The difference between our results and those reported by Foucault et al. [7] could possibly be attributed to the differences in the artifact and the experimental design used in the two studies (i.e., module vs project level granularity).

RQ2: Are Android applications with *higher values of MVA* less fault-prone than applications with *lower values of MVA*?

From the results presented in Table 3, we use Model-2 to address RQ2. We observe from the table that the addition of MVA variable in the Base model, i.e., Model-2, significantly improves the variance explained. The added MVA metric examines the impact of the most valuable author on the number of faults. We can also observe that MVA has a significant relative importance of about 10.1%. This statistic tells us that a change in the value of authorship levels in an Android application relates to the number of faults in that Android application. However, we also observe that the MVA metric has a negative coefficient. This tells us that Android applications that have a high value of MVA (i.e., less shared code authorship) are less fault-prone than those with low values of code authorship.

Conclusion RQ2: In Section 2.1 we stated that we expect an increase in the value of the MVA metric to result in a decrease of the number of faults. From the discussion of the results presented above, we can conclude that Android applications with higher levels of code authorship are less fault-prone than applications with lower levels of code authorship. Again, our study's findings concur with the findings of Bird et al. [3] which were obtained on large commercial software projects. Regarding the work of Foucault et al. [7] which was conducted on seven large open source projects, our findings still disagree with their claim that the contribution of the MVA metric seem to be incidental. Our analysis shows that the metric MVA indeed has a relationship with faults in the applications.

From the results of RQ1 and RQ2 discussed above, we make the following recommendations to the developers of Android applications regarding the development process:

(1) *Changes made by minor contributors should be reviewed with more scrutiny before being committed on respective Github projects.* Development teams in Android applications should apply additional scrutiny to files authored by minor developers before these files are committed to the respective Android projects on Github. Since Github differentiates the author and the committer of a software change, we recommend that before a commit is performed, the committer should request major contributors on the project to perform an inspection on the changes made by minor contributors.
(2) *Android applications with lower levels of code authorship should be reviewed with more scrutiny.* Android application project teams on Github can make use of the MVA metric to scrutinize applications with low values of MVA as they may contain many faults and hence experience many failures.

5 THREATS TO VALIDITY

Though we have sought to make sure that all our data was gathered and linked correctly, and that our models are statistically robust, we note some potential threats to validity.

There are *construct validity* threats related to our data collection approach. In the description of the software project contributors in Section 3, we mentioned that some projects might have been started elsewhere, therefore GitHub does not provide the full history of these projects. We therefore decided to extract the code authorship metrics from GitHub in the project, by collecting only the contributions that were made *after the first release* of the project until *March, 6 2016*. This design decision is likely to affect the authorship metrics values obtained on some projects. However, since we analyzed a large number of projects (including projects started on Github), we believe the effect on our conclusions to be minimal. Another *construct validity* threat concern our use of heuristic to merge the names of contributors, in order to avoid polluting our data set with duplicate informations that would introduced noise. It is possible that some names that we merged represent different contributors. There is also a *construct validity* threat related to the fact that we considered faults on the entire projects and not at commit level. It

is possible that one/few developer(s) may be writing most of the faulty code. Nevertheless, since we analyzed a large number of projects from diverse domains and diverse team sizes, we believe the effect of this skewness on our conclusions to be minimal.

There is an *internal validity* threat related to the tool (i.e., SonarQube) used to extract faults information, and compute size and complexity metrics. SonarQube is a well maintained tool that is extensively used by practitioners. However, as most static analysis tools, it doesn't have a 100% precision. It is possible that some of the faults considered in this study will never be experienced either by developers or the users of the applications.

Finally, although the observed correlations and the model results indicate that the phenomenons are related, i.e., low authorship and fault introduction, we cannot claim causation.

6 RELATED WORK

As mentioned earlier, the work presented in this paper builds on previous studies. To this end, throughout the paper we have discussed a number of studies that relate to ours. In this section, we shall discuss other studies related to our study that we have not yet discussed.

A number of other studies have investigated the relationship shared between code authorship metrics and fault-proneness. Matsumoto et al. [13] studied the effects of developer characteristics on software reliability. The authors proposed developer metrics such as the number of code churns made by each developer and the number of developers for each module. The authors analyzed the relationship between the number of faults and developer metrics. The authors reported that modules touched by more developers contained more faults.

Nagappan et al. [18] proposed a metric scheme to quantify organizational complexity, in relation to the product development process. They conducted a case study to identify if the metrics impact failure-proneness. For the organizational metrics, the number of developers was one of the metrics. The authors found that the precision and recall measures for identifying failure-prone binaries, using the organizational metrics, was significantly high.

Weyuker et al. [22] investigated the impact on predictive accuracy of using data about the number of developers who accessed individual code units. The authors found only moderate improvements of fault prediction models that included the cumulative number of developers as prediction factor. Meneely et al. [15] studied the effects of the number of contributors on security vulnerabilities focusing on the Linux software system. They reported that files with changes from nine or more developers were 16 times more likely to have a vulnerability than files changed by fewer than nine developers. In our study we also use the number of contributors as one authorship metric.

Mockus et al. [16] observed two code authorship patterns in the open source projects Apache and Mozilla. In the Apache project, they found that almost every source code file with more than 30 changes had several contributors who authored more than 10% of the changes. In the Mozilla project they found that code authorship decisions were enforced by project development guidelines, which stated that all contributions should be reviewed and approved by the module owner. The authors investigated authorship but did not attempted to examine the connection between the authorship patterns and fault-proneness.

7 CONCLUSIONS

In this study, we investigated whether applications with few major contributors are more reliable than applications larger number of contributors where developers do minor contributions. We carefully selected 278 Android applications from GitHub, from which we extracted metrics related to code authorship. We measured the reliability of the applications using information about fault occurrences, i.e., the number of faults. Using statistical modeling, we examined the relationship between code authorship metrics and faults. We observed that Android applications with higher levels of code authorship among contributors experience fewer faults.

We formulate the following two recommendations to development teams of Android applications projects:

(1) Changes made by minor contributors should be reviewed with more scrutiny before being committed.
(2) Android applications with lower levels of code authorship should be reviewed with more scrutiny as they may contain faults.

To generalize our findings, in the future we plan to expand this work to investigate other relatively small sized projects in other domains.

REFERENCES

[1] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. 1994. The Goal Question Metric Approach. In *Encyclopedia of Software Engineering*. Wiley.
[2] Christian Bird, Nachiappan Nagappan, Premkumar Devanbu, Harald Gall, and Brendan Murphy. 2009. Does Distributed Development Affect Software Quality?: An Empirical Case Study of Windows Vista. *Communication ACM* 52, 8 (2009), 85–93.
[3] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. 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*.
[4] Marcelo Cataldo, Audris Mockus, Jeffrey A. Roberts, and James D. Herbsleb. 2009. Software Dependencies, Work Dependencies, and Their Impact on Failures. *IEEE Transactions Software Engineering* 35, 6 (Nov. 2009), 864–878.
[5] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai. 2001. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering* 27, 7 (2001), 630–650.
[6] Barry E. Feldman. 2005. Relative Importance and Value. *Available at SSRN* (2005).
[7] Matthieu Foucault, Cédric Teyton, David Lo, Xavier Blanc, and Jean-Rémy Falleri. 2015. On the Usefulness of Ownership Metrics in Open-source Software Projects. *Inf. Softw. Tech.* 64, C (Aug. 2015), 102–112.
[8] Thomas Fritz, Gail C. Murphy, and Emily Hill. 2007. Does a Programmer's Activity Indicate Knowledge of Code?. In *Pro. of the 6th Joint Meeting of the European Soft. Eng. Conf. and ACM SIGSOFT Symposium on The Foundations of Software Engineering*.
[9] Michaela Greiler, Kim Herzig, and Jacek Czerwonka. 2015. Code Ownership and Software Quality: A Replication Study. In *Pro. of the 12th Working Conference on Mining Software Repositories*.
[10] Ulrike Grömping. 2006. Relative Importance for Linear Regression in R: The Package relaimpo. *Journal of Statistical Software* 17, 1 (2006), 1–27.
[11] F. E. Harrell Jr. 2015. *Regression Modeling Strategies*.
[12] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W. Godfrey. 2015. Investigating Code Review Quality: Do People and Participation Matter?. In *Proceedings of International Conference on Software Maintenance and Evolution*.
[13] Shinsuke Matsumoto, Yasutaka Kamei, Akito Monden, Ken-ichi Matsumoto, and Masahide Nakamura. 2010. An Analysis of Developer Metrics for Fault Prediction. In *Proeedings of the 6th International Conference on Predictive Models in Software Engineering*.
[14] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 2016. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering* 21, 5 (2016), 2146–2189.

[15] Andrew Meneely and Laurie Williams. 2009. Secure Open Source Collaboration: An Empirical Study of Linus' Law. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*.
[16] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. 2002. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Trans. Softw. Eng. Methodol.* 11, 3 (July 2002), 309–346.
[17] Christopher J Nachtsheim, John Neter, Michael H Kutner, and William Wasserman. 2004. Applied linear regression models. *McGraw-Hill Irwin* (2004).
[18] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. 2008. The Influence of Organizational Structure on Software Quality: An Empirical Case Study. In *Pro. of the 30th Int. Conf. on Software Engineering*.
[19] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. 2008. Can Developer-module Networks Predict Failures?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*.
[20] Foyzur Rahman and Premkumar Devanbu. 2011. Ownership, Experience and Defects: A Fine-grained Study of Authorship. In *Pro. of the 33rd International Conference on Software Engineering*.
[21] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. 2016. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Pro. of the 38th International Conference on Software Engineering*.
[22] Elaine J. Weyuker, Thomas J. Ostrand, and Robert M. Bell. 2008. Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models. *Empirical Software Engineering* 13 (2008), 539–559.
[23] Thomas Zimmermann, Nachiappan Nagappan, Philip J. Guo, and Brendan Murphy. 2012. Characterizing and Predicting Which Bugs Get Reopened. In *Proceedings of the 34th International Conference on Software Engineering*.

delegate
COPY

Ghera: A Repository of Android App Vulnerability Benchmarks

Joydeep Mitra
Kansas State University
USA
joydeep@k-state.edu

Venkatesh-Prasad Ranganath
Kansas State University
USA
rvprasad@k-state.edu

ABSTRACT

Security of mobile apps affects the security of their users. This has fueled the development of techniques to automatically detect vulnerabilities in mobile apps and help developers secure their apps; specifically, in the context of Android platform due to openness and ubiquitousness of the platform. Despite a slew of research efforts in this space, there is no comprehensive repository of up-to-date and lean benchmarks that contain most of the known Android app vulnerabilities and, consequently, can be used to rigorously evaluate both existing and new vulnerability detection techniques and help developers learn about Android app vulnerabilities. In this paper, we describe *Ghera*, an open source repository of benchmarks that capture 25 known vulnerabilities in Android apps (as pairs of exploited/benign and exploiting/malicious apps). We also present desirable characteristics of vulnerability benchmarks and repositories that we uncovered while creating Ghera.

KEYWORDS

Security, Collection, Benchmarking, Mobile, Applications

ACM Reference format:
Joydeep Mitra and Venkatesh-Prasad Ranganath. 2017. Ghera: A Repository of Android App Vulnerability Benchmarks. In *Proceedings of PROMISE, Toronto, Canada, November 8, 2017,* 10 pages.
https://doi.org/10.1145/3127005.3127010

1 INTRODUCTION

1.1 Motivation

With increased use of mobile devices (including mobile phones), mobile apps play a pivotal role in our lives. They have access to and use huge amounts of sensitive and personal data to enable various services banking, shopping, social networking, and even two-step authorization. Hence, security of mobile devices and apps is crucial to guarantee the security and safety of their users.

This observation is further amplified on the Android platform as it is widely adopted by both consumers (users) and developers – Android has captured 88% of the global market share as of third quarter of 2016 [3] and Google Play, the official store for Android apps, has 2.8 million apps as of March 2017 [22].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PROMISE, November 8, 2017, Toronto, Canada
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5305-2/17/11…$15.00
https://doi.org/10.1145/3127005.3127010

One approach to secure Android devices is to keep malicious apps out of Android devices. This approach is supported by numerous tools and techniques that detect malicious behaviors in apps [14, 19, 23, 26, 27]. However, these techniques cannot detect every malicious behavior. Further, malicious behaviors often depend on vulnerabilities in the Android platform and the apps executed on the platform.

Consequently, a complementary approach is to *secure (harden) Android apps*. This approach is also supported by tools and techniques to detect known vulnerabilities in apps (referred to as *app vulnerabilities*) that stem from incorrect use of or bugs in Android framework[1] and can be used to carry out malicious actions such as data theft.[2]

Like all automation-based approaches, this approach is useful only if developers can trust the verdicts of the employed tools and techniques. Such trust can be established only by rigorous and reproducible evaluations of tools and techniques. By *rigorous*, we mean the verdict of a technique can be verified to be true and to be caused only by the reasons/explanations provided by the technique (within the context considered in the evaluation). By *reproducible*, we mean evaluations can be repeated to verify the reproduction of (same) results.

Further, if such evaluations are based on *a common baseline – a set of benchmarks (apps)*[3] *containing specific vulnerabilities*, then their results will be fair and comparable. By *fair*, we mean evaluations will not favor any specific technique. By *comparable*, we mean the results from evaluations of different techniques can be compared (as the evaluations are controlled for subjects). Consequently, developers can use the results from such evaluations to easily compare and select techniques based on aspects such as efficacy, efficiency/scale, and ease-of-use. Also, a common baseline can simplify comparison of future techniques with existing techniques and help make robust claims about future techniques.

Another approach to secure Android apps is to *educate developers about securing apps*. This approach is enabled by the extensive official documentation provided by Google about security in Android and the best practices for security and privacy in Android apps [10, 11]. It is also enabled by documentation available in the form of white papers, guides, and books about how to secure Android apps [7, 12, 18]. Most of these resources provide code snippets (at times, even apps) that showcase good practices to secure apps.

Despite the availability of such resources, Android apps still have vulnerabilities [6, 17]. The presence of known vulnerabilities in apps

[1] Android framework is a set of APIs available to Android apps to interact with the Android platform.
[2] This paper does not explore the application of this approach to harden the Android platform.
[3] A benchmark is a standard or point of reference used for evaluation/comparison. So, in this paper, we refer to an app that embodies a specific vulnerability X (along with an app that exploits X) as a benchmark for X.

can only stem from developer's lack of knowledge about known vulnerabilities, benefits of securing apps, or how security can easily go awry. Like bad code examples are interspersed between good code examples in programming books to highlight good practices via contrasting, a way to help address such lack of knowledge is to *offer apps that showcase known vulnerabilities and demonstrate how security can easily go awry.* Such a repository of vulnerable apps can also serve as an channel to learn about known vulnerabilities in Android apps.

Finally, while there are numerous efforts to develop tools and techniques to secure Android apps, there is no single benchmark repository that captures most of the known vulnerabilities in Android apps in a technique/tool agnostic manner.

1.2 Contributions

Motivated by above observations and the need of an ongoing effort to assess existing vulnerability detection tools for Android, we created *Ghera*, an open source repository of vulnerability benchmarks[4]. Currently, Ghera comprises 25 benchmarks. Each benchmark is a pair of exploited/benign and exploiting/malicious apps that captures a unique known vulnerability in Android apps. The benchmarks span four areas of Android framework: *Inter Component Communication (ICC), Storage, System,* and *Web*. Ghera is open sourced under BSD 3-clause license and publicly available at http://bitbucket.org/secure-it-i/android-app-vulnerability-benchmarks.

While creating these benchmarks, we found very little guidance in the literature regarding how to create good benchmarks or the desirable characteristics of good benchmarks. So, after creating the benchmarks, we did a retrospection and identified various desirable characteristics of vulnerability benchmarks, which are applicable to benchmarks in general.

In this paper, we present Ghera in detail along with the desirable characteristics of vulnerability benchmarks.

The remainder of the paper is organized as follows. Section 2 lists desirable characteristics of vulnerability benchmarks. It also explores related work in the light of these characteristics. Section 3 describes Ghera in terms of its design choices, structure and content, characteristics, and limitations. Section 4 lists future possibilities with Ghera. Section A catalogs the vulnerabilities captured in Ghera.

2 DESIRABLE CHARACTERISTICS OF VULNERABILITY BENCHMARKS

2.1 Context

When we started an effort to assess various vulnerability detection techniques, we searched for suites of tests or benchmarks to fairly evaluate the techniques. We did not find such suites. Most existing efforts used applications from app stores, e.g., Google Play. While few efforts constructed small dedicated example apps, almost all of the example apps were geared towards demonstrating the associated techniques and did not explicitly capture known vulnerabilities, e.g., DroidBench. Hence, we took a detour in our effort to *collect and catalog known Android app vulnerabilities in an informative and comprehensive repository to enable rigorous and reproducible evaluation of vulnerability detection tools and techniques.*

As we started collecting vulnerabilities, we searched for guidance to create good benchmarks. To our surprise, while there were numerous benchmarks, there was very little information about how to design good benchmarks or even characteristics of a good benchmark. Hence, after we collected vulnerabilities, we did a retrospection to identify characteristics that we considered while collecting vulnerabilities along with the reasons why we considered them.

2.2 Vulnerability Benchmark Characteristics

Here are the characteristics identified during our retrospection. Besides describing the characteristics, we also describe how they were influenced by existing efforts in the space of detecting vulnerabilities in Android apps.

2.2.1 Tool and Technique Agnostic. The benchmark is agnostic to tools and techniques and how they detect vulnerabilities. This characteristic enables the use of benchmarks for fair evaluation and comparison of tools and techniques.

We uncovered this characteristic when we explored *DroidBench*[5], one of the first benchmark suites created in the context of efforts focused on detecting vulnerabilities in Android apps [2]. DroidBench is tailored to evaluate the effectiveness of taint-analysis[6] tools to detect information leaks in Android apps. So, the benchmarks are geared towards testing the influence of various program structures and various aspects of static analysis (e.g., field sensitivity, trade-offs in access-path lengths) on the effectiveness of taint analysis to detect information leaks (vulnerabilities). Further, it is unclear if program structures considered in the benchmarks reflect program structures that enable vulnerabilities in real world apps or program structures that push the limits of static analysis techniques. Hence, these benchmarks are not tool and technique agnostic.

In contrast, repositories such as AndroZoo[7] [1] and PlayDrone[8] [24] provide real world Android apps available in various app stores. Further, the selection of apps is independent of their intended use by any specific tool or technique. Hence, any vulnerable apps (benchmarks) in these repositories are tool and technique agnostic.

2.2.2 Authentic. If the benchmark claims to contain vulnerability X, then it truly contains vulnerability X. This characteristic enables benchmarks to serve as ground truths when evaluating accuracy of tools and techniques. Consequently, the comparison of tools and techniques is simplified.

To appreciate this characteristic, consider the evaluation of MalloDroid, a tool to detect the possibility of Man-in-the-Middle (MitM) attacks on Android apps due to improper TLS/SSL certificate validation [8]. To evaluate the accuracy of MalloDroid, 13,500 apps were analyzed using Mallodroid and 8% of them were flagged as potentially vulnerable. However, since the potential vulnerability does not imply the vulnerability can be exploited to steal information, 266 apps from Google Play were selected based on app categories

[4] We refer to a benchmark containing at least one vulnerability as *a vulnerability benchmark.*

[5] https://github.com/secure-software-engineering/DroidBench

[6] Taint analysis identifies parts of the program affected (tainted) by specific data sources such as user input.

[7] https://androzoo.uni.lu/

[8] https://github.com/nviennot/playdrone

that are likely to be most affected by detected vulnerabilities. Of these apps, 100 of the most popular apps were manually audited and 41 of them were found to have exploitable vulnerabilities. Further, not all aspects of apps were exercised during the evaluation. This evaluation could have been simpler, easier, more complete, and more accurate using authentic vulnerable benchmarks.

In terms of comparing tools and techniques, when EdgeMiner [4] was compared to FlowDroid [2], 9 apps were flagged with vulnerabilities by EdgeMiner but not by FlowDroid. To verify this outcome, the 9 flagged apps were analyzed using TaintDroid, yet another tool, and 4 apps were flagged by TaintDroid. So, it was concluded that EdgeMiner did better than FlowDroid in 4 cases. With authentic benchmarks, the comparison would have been simpler and extensive.

Of the various repositories we explored, AndroZoo provides some evidence of malicious behavior (exploits) in apps in the form of verdicts from running analysis tools on apps. However, no evidence is provided about presence of vulnerabilities. Hence, these benchmarks cannot be used authentic vulnerability benchmarks. In contrast, the benchmarks in DroidBench are authentic as they are seeded with vulnerability during construction.

2.2.3 Feature Specific. If the benchmark uses only features F of a framework to create vulnerability X, then the benchmark does not contain other features of the framework that can be used to create X. This characteristic helps evaluate if tools and techniques can detect vulnerabilities that stem only due to specific reasons (features). In other words, it helps assess if and how the cause of a vulnerability affects the ability of tools and techniques to detect the vulnerability. Often, this could translate into being able to verify the explanations provided by a tool when it detects X.

As discussed above, EdgeMiner detected 4 more vulnerable apps than FlowDroid. However, there was no explanation for the better performance of EdgeMiner in terms of features (causes) that EdgeMiner handled better than FlowDroid. Such explanations could have been easily uncovered with feature specific benchmarks.

Often, real world apps serving as benchmarks (as in case of AndroZoo and PlayDrone) lack this characteristic as the causes of app vulnerabilities in them are most likely unknown to the public. In contrast, benchmarks in repositories such as DroidBench exhibit this characteristic as they are feature specific by construction.

2.2.4 Contextual. The benchmark capturing vulnerability X in a context C is distinct from benchmarks capturing X in other contexts. This characteristic helps evaluate the efficacy of tools and techniques to detect vulnerabilities in specific contexts, e.g., real world scale, experimentation, use of specific libraries.

To understand this characteristic, consider the size of benchmarks. While evaluating a tool (or a technique) to detect vulnerabilities, we can first evaluate it on *lean* benchmarks – *A benchmark containing vulnerability X that can be created by any of the features F of a framework is lean if it makes minimal use of features of the framework not in F* – . Due to leanness, such benchmarks can speed up the evaluation process and enable easy comprehension when debugging/understanding the behavior of the tool. Further, such benchmarks can be used as tests while building tools. When in doubt, authenticity of such benchmarks can be easily verified with manual effort. After such evaluations, we can consider *fat (non-lean)* benchmarks to evaluate how the tool performs (scales) on larger inputs. Hence, with contextual benchmarks, efficacy evaluations can be more focused and streamlined.

In our explorations, we found tool evaluations that use both lean and fat benchmarks [2, 25] and tool evaluations that use only fat benchmarks [8, 15, 20][9]. In comparison, establishing veracity of tools was easier in former tool evaluations.

As for repositories, DroidBench offers custom apps as lean benchmarks while AndroZoo and PlayDrone offer real world apps as fat benchmarks.

Like size, connectivity, resource availability, complexity, number of vulnerabilities in a benchmark, and number of kinds of vulnerabilities in a benchmark can be considered to identify useful contexts when designing benchmarks.

2.2.5 Ready-to-Use. The benchmark is composed of artifacts that can be used as is to reproduce the vulnerability. This characteristic precludes the influence of external factors (e.g., interpretation of instructions, developer skill) in realizing a benchmark, e.g., starting from its textual description or skeletal form. Hence, it enables fair evaluation and comparison of tools and techniques.

DroidBench, AndroZoo, and PlayDrone repositories provide benchmarks as ready-to-use APKs (Android app bundles).

In comparison, SEI provides a set of guidelines for development of secure Android apps [21]. The descriptions of many guidelines are accompanied by illustrative good and bad code snippets. While the code snippets are certainly helpful, they are not ready-to-use in the above sense. This is also true of many security related code snippets available as part of Android documentation.

2.2.6 Easy-to-Use. The benchmark is easy to set up and reproduce the vulnerability. Benchmarks with this characteristic help expedite evaluations. Consequently, this characteristic can help usher wider adoption of the benchmarks. This characteristic is desirable of benchmarks that require some assembling, e.g., build binaries from source, extensive set up after installation.

As with ready-to-use characteristic, DroidBench, AndroZoo, and PlayDrone cater binary benchmarks that are easy to install and conduct evaluations. The source form of benchmarks provided by DroidBench also have this characteristic as they contain Eclipse project files required to build them.

2.2.7 Version Specific. The benchmark is associated only with the versions of the framework in which the contained vulnerability can be reproduced. With this characteristic, benchmarks can be chosen for an evaluation based on the version of the framework being used in the evaluation. Hence, it helps evaluations to be version specific.

To appreciate this characteristic, consider a vulnerability that was affected by the rapid evolution of Android framework – evolved from level (version) 1 thru 25 from 2008 to 2017. In 2011, when the latest version of Android was 4.0.4, Chin et al. revealed that a background service in an Android app could be hijacked by a malicious app installed on the device if the service allowed clients to start the service via an implicit intent [5]. Starting with Android 7.0 in 2016, the vulnerability was invalidated as starting of services via implicit intent were prohibited.

[9] While lean benchmarks may have been used, such uses were not reported.

Now, suppose an evaluation uses a benchmark that captures this vulnerability but is not version specific. This can lead to two undesirable situations. In the first situation, a tool targeting Android 7 apps will be flagged as incorrect when it (correctly) fails to detect this vulnerability. In the second situation, a tool that detects the vulnerability will be flagged as correct but it will incorrectly detect non-existent vulnerabilities in Android 7 apps. Both these situations can be avoided by using version specific benchmarks.

In terms of maintenance and keeping benchmarks current, version specific benchmarks can be easily updated to reflect any changes (such as end of life support) to associated versions of Android framework/platform. This will prevent accidental evaluations of benchmarks with unsupported versions of Android.

The benchmarks in AndroZoo, PlayDrone, and DroidBench are not version specific as these benchmarks have no information about compatible versions of Android framework/platform. Consequently, DroidBench can enable a situation similar to that one described above. Many Droidbench benchmarks use log files as information sink. Such use was valid prior to Android 4.1 as every app on a phone could read log files of any apps on the phone. However, since Android 4.1, apps can read only their log files. Hence, these benchmarks can lead to incorrect evaluations depending on Android version being considered.

2.2.8 Well Documented. The benchmark is accompanied by relevant documentation. Such documentation should contain description of the contained vulnerability and the features used to create the vulnerability. It should also mention the target (compatible) versions of the framework/platform and provide instructions to both surface the vulnerability and exploit the vulnerability. When possible, the source code of the benchmark should be included as part of the documentation. This characteristic obviously helps expedite evaluations that use the benchmarks and contributes to ease of use of benchmarks. With source code, it can help developers understand the vulnerability.

The benchmarks provided by DroidBench are in some ways well documented as they contain source code along with binaries and there is brief documentation on the web site and in the source code about captured vulnerabilities. This is not the case with benchmarks catered by AndroZoo and PlayDrone.

2.2.9 Dual. The benchmark contains both the vulnerability and a corresponding exploit (dual). This characteristic simplifies evaluations that depend on exercising the vulnerability, e.g., dynamic analysis. It allows benchmarks to be used to demonstrate vulnerabilities and even evaluate exploits. Benchmarks with this characteristic can help developers understand the vulnerability; specifically, when the source code is available. Also, duality helps verify the authenticity of benchmarks.

In our exploration, we did not find any dual benchmarks.

2.3 Vulnerability Benchmark Repository Characteristics

Similar to the above benchmark characteristics, here are two desirable characteristics of vulnerability benchmark repositories.

2.3.1 Open. The benchmark repository should open to the community both in terms of consumption and contribution. The benchmarks should be available with minimal restrictions (e.g., permissive licence) and preferably at no or very low cost to the community. The repository should have a well-defined yet accessible process for the community to contribute new benchmarks. This characteristic helps with reproducibility of results and community wide consolidation of benchmarks. The latter effect reduces duplication efforts in the community.

In this regard, DroidBench is more open than AndroZoo and PlayDrone repositories. DroidBench is hosted as a public repository on GitHub and it welcomes contributions. PlayDrone is hosted as multiple public archives on Internet Archive with no explicit guidance for contributions. AndroZoo is hosted as a web service that can be accessed only by approved users. This is most likely to manage and track access to a large corpus of data in AndroZoo. Like PlayDrone, there is no explicit guidance for contributions. This may be due to how AndroZoo is populated – with real world apps collected from different app stores.

2.3.2 Comprehensive. The benchmark repository should have benchmarks that account for (almost) all known vulnerabilities of the target framework/platform. This characteristic simplifies evaluations as they can rely on a single repository (or very few repositories) to consider all vulnerabilities. Further, evaluations can be more thorough as they can consider most of the known vulnerabilities.

In our explorations, we did not find a single repository that covered almost all vulnerabilities in Android apps. While DroidBench does a pretty good job covering information leak vulnerabilities stemming from ICC, it does not cover information leaks due to other reasons such as misuse of WebView component [13]. In terms of evaluation, Reaves et al. [16] used DroidBench along with 6 mobile money apps [17] and 10 most widely used financial apps in Google Play to evaluate Android security analysis tools. While DroidBench and 6 mobile money apps had certain known vulnerabilities, they did not cover all kinds of vulnerabilities. With a comprehensive repository, this evaluation could have been simpler and more thorough.

2.4 Discussion

Reality Check. Of the above characteristics, some are easier to achieve than others. For example, creating an open repository is easier than creating a comprehensive repository. Similarly, creating a tool agnostic benchmark is easier than creating an authentic benchmark. Further, while it is desirable for benchmarks/repositories to have all of the characteristics, it is hard to achieve them all as we have seen in examples. Nevertheless, given the benefits of these characteristics, we believe that the community should strive to create benchmarks/repositories with these characteristics.

Wider Relevance. While we uncovered and described these characteristics in the context of vulnerabilities, we believe they apply to benchmarks in general; say, in other contexts such as performance. For example, in the context of performance benchmarking, *agnostic* characteristic can be restated as *The benchmark is agnostic to techniques and how they achieve performance.*

3 GHERA

We created Ghera, an Android app vulnerability benchmarks repository, because we needed vulnerable benchmarks to evaluate existing tools that aid in the development of secure Android apps. We initially explored existing repositories like DroidBench, AndroZoo, and PlayDrone. Except for DroidBench, there was little to no information about the presence and kind of vulnerabilities in apps from these repositories. In case of DroidBench, the benchmarks/apps were specific to information flow-based vulnerabilities.

So, our goals while creating Ghera were to ensure the benchmarks contained unique Android app vulnerabilities and were tool agnostic, easy to use, and well documented with source code. In addition, we wanted the repository to be open and as comprehensive as possible.

3.1 Design Choices

Given our goals for the repository, we decided to group vulnerabilities based on the features (capabilities) that cause the vulnerabilities. To decide on the features that we wanted to explore, we looked at features commonly used by Android apps and discussed in various Android security related resources [5, 8, 9, 18]. Almost all Android apps use one or more of the following capabilities:

- Communicate with components in apps installed on the device.
- Store data on the device.
- Interact with the Android platform.
- Use web services.

Based on these capabilities, for the initial version of the repository, we identified *Inter Component Communication (ICC), Storage, System,* and *Web* as the categories of vulnerabilities.

In each category, we identified APIs pertinent to that category and studied them. To determine potential vulnerabilities stemming from an API, we primarily explored prior research efforts, Stack Overflow discussions, Android documentation, and the source code in the Android Open Source Project (AOSP).[10] When we uncovered a potential vulnerability X related to the API, we developed an app M with the vulnerability X along with an app N to exploit vulnerability X in app M. We then verified the vulnerability by executing apps M and N and checking if the vulnerability was indeed exploited. This verification was carried out on Android versions 4.4 thru 7.1.

We decided to name each benchmark based on the feature causing the vulnerability captured by the benchmark and the exploit used to confirm the vulnerability. So, a benchmark is named as *P_Q* when feature P causes the vulnerability that enables exploit Q.

3.2 Structure and Content

The repository contains top-level folders corresponding to various categories of vulnerabilities: *ICC, Storage, System,* and *Web*. We refer to these top-level folders as *category folders*. Each category folder contains subfolders corresponding to different benchmarks. We refer to these subfolders as *benchmark folders*. Each category folder also contains a README file that briefly describes each benchmark in the category.

There is one-to-one correspondence between benchmark folders and benchmarks. Each benchmark folder is named as *P_Q* where P is the specific feature that causes a vulnerability of interest and Q is the exploit enabled by the vulnerability. Each benchmark folder contains two *app folders*: *Benign* folder contains the source code of an app that uses feature P to exhibit a vulnerability and *Malicious* folder contains the source code of an app that exploits the vulnerability exhibited by the app in the *Benign* folder. A README file in each benchmark folder summarizes the benchmark, describes the contained vulnerability and the corresponding exploit, provides instructions to build both Benign and Malicious apps (refer to Section 3.3), and lists the versions of Android on which the benchmark has been tested.

In case of *Web* category, benchmark folders do not contain a *Malicious* folder in them because the captured vulnerabilities can be exploited by Man-in-the-Middle (MitM) attacks. This requires a web server that the Benign apps can connect to. Consequently, code and instructions to set up local web server are provided in a top-level folder named *Misc/LocalServer*. README file of each Benign app contain instructions to configure the app to talk to the local web server. As for the MitM attack in this set up, the users are free to choose how to mount such an attack.

Currently, the repository contains 25 benchmarks, each capturing a unique vulnerability. There are 13 ICC benchmarks, 2 Storage benchmarks, 4 System benchmarks, and 6 Web benchmarks. The smallest and the largest benchmarks contains 490 and 1510 lines of code and configuration, respectively. In terms of bytes, Storage benchmarks are larger as they rely on two external APK files of size 13,144 KB in total. Table 1 provides basic size based statistics about the benchmarks.

3.3 Work Flow

To illustrate the steps involved in using Ghera, we consider the *ICC/DynamicallyBroadcastReceiverRegistration-UnrestrictedAccess* benchmark. In this benchmark, the benign app dynamically registers a service with Android platform. This action exposes the service to any app on the same device, including malicious apps. Following are the instructions to use the benchmark to reproduce and exploit this vulnerability.

(1) Execute the following commands to create an Android Virtual Device (AVD) if an AVD does not already exist.

```
1 avdmanager list target
2 avdmanager list avd
3 avdmanager create avd -n <name> -k <target>
```

`<name>` should be an identifier not in the list of existing AVD names displayed by the command in line 2. `<target>` should be a target identifier of a system image listed by the command in line 1.

(2) Start the emulator.

```
emulator -avd <name>
```

(3) Build Benign and Malicious apps and install them on to the emulator using the following commands.

[10] We explored public facing lists of vulnerabilities and exploits maintained by organizations such as Mitre [6]. While these lists are useful to understand different kinds of common vulnerabilities on a platform or a framework, they often lack details about app vulnerabilities such as primary causes of a vulnerability or how to reproduce a vulnerability. Also, many of these vulnerabilities do not have accompanying known exploits. So, to quickly bootstrap the benchmark repository, we did not consider such lists. However, since these lists serve as excellent starting points to create benchmarks for app vulnerabilities, we are exploring them to expand the repository.

	Total Size (KB)			Size of Text Files (KB)			# Lines of Code+Config		
Category	*Min*	*Median*	Max	*Min*	*Median*	Max	*Min*	*Median*	Max
ICC	364	380	700	164	196	336	635	1182	1510
Storage	1,000	7,500	14,000	156	160	164	936	955	974
System	364	404	444	164	172	180	968	995.5	1035
Web	192	224	228	84	86	100	490	522	590

Table 1: Basic Source Code Size Statistics of Benchmarks in Ghera

```
cd Benign
./gradlew installDebug
cd ../Malicious
./gradlew installDebug
```

(4) In the emulator, launch Benign app followed by Malicious app.
(5) Execute the following command.
```
adb logcat -d | grep "UserLeftBroadcastRecv"
```
If the exploit was successful, you should see a message "An email will be sent to rookie@malicious.com with the text: I can send email without any permissions".

The number of steps are few and they are simple. However, they assume the user is familiar with Android development in terms of various tools (e.g., virtual devices, emulators) used while developing Android apps.

Further, instructions assume users will use these benchmarks on emulators. If a user wants to use a benchmark on a device, then she should skip step 2 and ensure an Android device is connected to the development machine and configured as a device.

The workflow for every benchmark is similar to the above workflow barring minor changes associated with specific aspects of the benchmark, if any.

3.4 What about characteristics?

While we identified the benchmark and repository characteristics in retrospective, we believe it did influence the creation of Ghera and its benchmarks. So, we will now examine if and why Ghera and its benchmarks exhibits the identified characteristics.

The creation of each benchmark in Ghera focused on the specific features of Android framework and how it could lead to a vulnerability. Based on this information, we created a Benign app that exhibited the vulnerability. We tested the vulnerability in the Benign app by using the Malicious app that exploited the vulnerability in the Benign app. This process did not involve the use of any vulnerability (or exploit) detection tools to confirm the presence of the vulnerability. Hence, the benchmarks are *tool and technique agnostic*.

As described above, for each benchmark, we tested the presence of vulnerability in the Benign app by exploiting the vulnerability via the Malicious app. As for Web apps, we used a local web server with self-signed certificates and static HTML containing malicious JavaScript code to exploit and verify the presence of the vulnerability. Hence, the benchmarks are *authentic*.

When creating the Benign app of each benchmark capturing a vulnerability X due to features F, we only used features F to create X. Further, we made very little use of other features of Android framework in the Benign app. Hence, we claim the benchmarks are *feature specific*.

The benchmarks focus on reproducing vulnerabilities while being minimal in size and the features used from Android framework. This observation is partially supported by the basic source code size statistics of the benchmarks in Table 1. Hence, we claim the benchmarks are *contextual*.

Every Ghera benchmarks come with instructions to build, install, and execute its Benign and Malicious apps to exercise captured vulnerabilities. Also, we have verified the instructions when testing the benchmarks on different versions of Android. Further, the work flow associated with each benchmark as illustrated in Section 3.3 is short, mostly automated, simple, and easy. Hence, the benchmarks are both *ready-to-use* and *easy-to-use*.

We tested each benchmark on different supported versions of Android on emulators. Based on the success of reproducing the vulnerabilities in these tests, we have documented the versions of Android on which a benchmark reproduces the captured vulnerability. Hence, the benchmarks are *version specific*. Further, the benchmarks cover all of the supported versions of Android.

Each benchmark is accompanied by documentation that describes the vulnerability and the associated exploit along with instructions to reproduce and exploit the vulnerability. In addition, each benchmark is available in source form. Hence, the benchmarks are *well documented*.

Since each benchmark (where possible) is composed of two apps: one exhibiting the vulnerability and another exploiting the exhibited vulnerability, the benchmarks exhibit the *dual* characteristic.

Ghera is hosted as a public repository that accepts contribution from the community. Hence, it is an *open* repository.

While Ghera does have benchmarks covering four different areas (capabilities) of Android framework, there are many more areas of the Android framework that may be associated with known vulnerabilities and are not covered by Ghera benchmarks. Hence, Ghera is *not yet a comprehensive* repository.

3.5 Limitations & Threats to Validity

Currently, Ghera only caters lean benchmarks. Consequently, it cannot be used to evaluate the scalability of vulnerability detection tools, which would require fat benchmarks.

Ghera benchmarks currently capture 25 known vulnerabilities while covering four areas of Android framework API: ICC, Storage, System, and Web. However, sources such as CVE [6] suggest vulnerabilities exists in other areas of the Android framework (e.g., Networking, Camera) not covered by Ghera. Likewise, in the four areas covered by Ghera, there may be known vulnerabilities that are not captured by any Ghera benchmarks.

In the previous section, we claimed Ghera benchmarks were feature specific and contextual based on our diligent process of creating benchmarks. However, our claim does not account for any bias or oversight.

4 FUTURE WORK

Here are few possibilities to extend and use Ghera to help secure Android apps.

(1) Add new benchmarks with vulnerabilities stemming from ICC, System, Storage, and Web related Android framework API but not present in existing benchmarks. Also, add new benchmarks with vulnerabilities stemming from Android APIs such as Networking and Camera that have not been considered by current benchmarks. These additions will make the repository more comprehensive.
(2) Extend the repository with real world apps (or links to apps in repositories such as AndroZoo) that have vulnerabilities captured by existing benchmarks as new context-specific benchmarks. These additions will help evaluate techniques for scale.
(3) Extract code patterns from these benchmarks to identify other instances of these benchmarks. These patterns can be codified as IDE plugins to help developers avoid vulnerabilities while coding apps. Also, these patterns could be used to measure the prevalence of corresponding vulnerabilities in real-world apps.
(4) Create benchmarks based on Android app vulnerabilities listed on CVE [6]. These additions will help developers understand the reported vulnerabilities and explore solutions to avoid them.
(5) Use these benchmarks to evaluate existing vulnerability analysis tools. Such an evaluation will help compare existing tools and possibly provide hints to new possibilities such as combining existing techniques to improve detection accuracy. (We are currently pursuing such an evaluation of existing tools.)

5 SUMMARY

While Android security has been the focus of research efforts in the past few years, there are hardly any benchmarks to test and evaluate vulnerability detection tools that help develop secure Android apps. Our search for such benchmarks led us to create Ghera, an open repository of vulnerability benchmarks (http://bitbucket.org/secure-it-i/android-app-vulnerability-benchmarks). Currently, Ghera captures 25 known vulnerabilities in Android apps spanning four different capabilities of Android platform. We plan to extend it and use it in an ongoing tools evaluation effort. We hope the community will use and contribute to Ghera to improve the evaluation of Android app vulnerability detection tools and techniques.

During the creation of Ghera, we uncovered various desirable characteristics of vulnerability benchmarks and benchmark repositories. Given the increasing interest in rigorous empirical evaluation, we have documented these characteristics in the hope that it can serve as guidance to create benchmarks to assist with rigorous and reproducible evaluations.

REFERENCES

[1] Kevin Allix, Tegawéndé F. Bissyande, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 468–471.
[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 259–269.
[3] Ananya Bhattacharya. 2016. Android just hit a record 88% market share of all smartphones. (2016). https://qz.com/826672/android-goog-just-hit-a-record-88-market-share-of-all-smartphones/ Accessed: 07-Jun-2017.
[4] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruege land Giovanni Vigna, and Yan Chen. 2015. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In *Proceedings of the Network and Distributed System Security Symposium*.
[5] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. Analyzing Inter-application Communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*. ACM, 239–252.
[6] Mitre Corporation. 2017. Common Vulnerabilities and Exposures. (2017). https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=Android Accessed: 08-Jun-2017.
[7] Nikolay Elenkov. 2015. *Android Security Internals*. No Starch Press.
[8] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. ACM, 50–61.
[9] Google Inc. 2017. Android Security Tips. https://developer.android.com/training/articles/security-tips.html. (2017). Accessed: 07-Jun-2017.
[10] Google Inc. and Open Handset Alliance. 2017. Best Practices for Security & Privacy. (2017). https://developer.android.com/training/best-security.html Accessed: 08-Jun-2017.
[11] Google Inc. and Open Handset Alliance. 2017. Security. (2017). https://source.android.com/security/overview/implement Accessed: 08-Jun-2017.
[12] Japan Smartphone Security Association (JSSEC). 2016. *Android Application Secure Design/Secure Coding Guidebook*.
[13] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. 2011. Attacks on WebView in the Android system. In *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 343–352.
[14] Oberheide and Miller. 2012. Dissecting Google Bouncer. https://jon.oberheide.org/files/summercon12-bouncer.pdf. (2012). Accessed: 08-Jun-2017.
[15] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective Inter-component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis. In *Proceedings of the 22Nd USENIX Conference on Security*. USENIX Association, 543–558.
[16] Bradley Reaves, Jasmine Bowers, Sigmund Albert Gorski III, Olabode Anise, Rahul Bobhate, Raymond Cho, Hiranava Das, Sharique Hussain, Hamza Karachiwala, Nolen Scaife, Byron Wright, Kevin Butler, William Enck, and Patrick Traynor. 2016. *Droid: Assessment and Evaluation of Android Application Analysis Tools. *ACM Comput. Surv.* 49, 3, Article 55 (Oct. 2016), 30 pages.
[17] Bradley Reaves, Nolen Scaife, Adam Bates, Patrick Traynor, and Kevin R.B. Butler. 2015. Mo(bile) Money, Mo(bile) Problems: Analysis of Branchless Banking Applications in the Developing World. In *24th USENIX Security Symposium*. USENIX Association, 17–32.
[18] Naveen Rudrapp. 2015. Secure Coding for Android Applications. http://www.mcafee.com/us/resources/white-papers/foundstone/wp-secure-coding-android-applications.pdf. (2015). Accessed: 08-Jun-2017.
[19] Alireza Sadeghi, Hamid Bagheri, Joshua Garcia, and Sam Malek. 2016. *A Taxonomy and Qualitative Comparison of Program Analysis Techniques for Security Assessment of Android Apps*. Technical Report. University of California, Irvine.
[20] A. Sadeghi, H. Bagheri, and S. Malek. 2015. Analysis of Android Inter-App Security Vulnerabilities Using COVERT. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. 725–728.
[21] SEI. 2016. Android Secure Coding Standard. (2016). https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=111509535 Accessed: 17-Jun-2017.
[22] Statista. 2017. Number of available applications in the Google Play Store from December 2009 to March 2017. (2017). https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/ Accessed: 08-Jun-2017.
[23] Sufatrio, Darell J. J. Tan, Tong-Wei Chua, and Vrizlynn L. L. Thing. 2015. Securing Android: A Survey, Taxonomy, and Challenges. *ACM Comput. Surv.* 47, 4 (2015),

58:1–58:45.
[24] Nicolas Viennot, Edward Garcia, and Jason Nieh. 2014. A Measurement Study of Google Play. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*. ACM, 221–233.
[25] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2014. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1329–1341.
[26] Y. Zhou and X. Jiang. 2012. Dissecting Android Malware: Characterization and Evolution. In *2012 IEEE Symposium on Security and Privacy*. 95–109.
[27] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. 2012. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*.

A CATALOG OF BENCHMARKS

In this section, we catalog the current benchmarks in Ghera according to the vulnerability categories identified in Section 3.1. For each benchmark, we provide a short description of the vulnerability and the exploit that uses the vulnerability.

A.1 Inter Component Communication

Android apps are composed of four basic kinds of components: 1) *Activity* components display the user interface, 2) *Service* components perform background operations, 3) *Broadcast Receiver* components receive event notifications and act on those notifications, and 4) *Content Provider* components manage app data. Communication between components in an app and in different apps is facilitated via exchange of *Intent*s. Components specify their ability to process specific kinds of intents by using *intent-filters*.

A.1.1 Dynamically registered broadcast receiver provides unrestricted access.

Vulnerability: When a broadcast receiver is dynamically registered with the Android platform, a non-null intent filter is provided. As a result, the component is automatically exported to be accessible from other apps, including malicious apps.

Exploit: A malicious app broadcasts a message to a dynamically registered broadcast receiver. This triggers the broadcast receiver to process the intent and unintentionally perform an action on behalf of the malicious app.

A.1.2 Empty pending intent leaks privilege.

Vulnerability: An app X can allow another app Y to perform an action on its behalf at a future time via a *pending intent*; these intents are saved in the system. When no action is specified in a pending intent, the recipient of the pending intent can set any action and execute it in the context of the app that sent the pending intent.

Exploit: A malicious app specifies its interest in the pending intent via an intent-filter. Upon receiving an empty pending intent, the malicious app associates a malicious action with the pending intent. Conseuqently, when the pending intent is processed, the malicious action will be executed in the context of app X.

A.1.3 Low priority activity prone to hijacking.

Vulnerability: A *priority* can be specified for an activity in the app's manifest file. When an activity is started, Android displays all activities with the same intent-filter as a list to the user in the order of priority (high to low).

Exploit: A malicious app registers an activity X with the same *intent-filter* as that of an activity Y registered by a benign app and with higher priority than Y. Consequently, the malicious app's activity X will be displayed before the benign app's activity Y.

A.1.4 Service started by implicit intent is prone to hijacking.

Vulnerability: Android platform uses intent-filters to identify the service to process *implicit intents*, i.e., intents dedicated to a class of targets (as opposed to specific target). When multiple services have the same intent-filter, the service with higher priority is chosen to process corresponding intents.

Exploit: A malicious app has a service X with the same intent-filter as that of the service Y in a benign app and with higher priority than Y. When an app requests the start of service Y by relying on the intent-filter, service X in the malicious app will be started.

A.1.5 Implicit pending intent leaks information.

Vulnerability: A app X can create a pending intent containing an implicit intent. When the pending intent is processed, the containing implicit intent will be processed by a component identified based on the intent-filter. When multiple components have the same intent-filter, the component with higher priority is chosen to process corresponding intents.

Exploit: A malicious app has a component X with an intent-filter same as that of the component Y in the benign app and X has higher priority than Y. So, component X is chosen (over component Y) to process the implicit intent in the pending intent.

A.1.6 Content provider with inadequate path-permission leaks information.

Vulnerability: An app can use *path-permissions* to control access to the data exposed by a content provider. When an app protects a folder by permissions, only the files in the folder are protected by the permissions; none of the subfolders and their descendants are protected by the permissions.

Exploit: A malicious app calls methods of a content provider to access and modify sub-directories and contained files that are not protected by path-permissions.

A.1.7 Apps have unrestricted access to Broadcast receivers registered for system events.

Vulnerability: When a Broadcast receiver registers to receive (system) intents from the Android platform, it needs to be exported. Consequently, it is accessible by any app without restrictions.

Exploit: A malicious app sends an intent to a broadcast receiver that is registered to receive system intents and possibly forces it to perform unintended operations.

A.1.8 Ordered broadcasts allow malicious data injection.

Vulnerability: When an ordered broadcast is sent, broadcast receivers respond to it in the order of priority. Broadcast receivers

with higher priority respond first and forward it to receivers with lower priority.

Exploit: A malicious receiver with high priority receives the intent, changes it, and forwards it to lower priority receivers.

A.1.9 Sticky broadcasts are prone to leaking sensitive information and malicious data injection.

Vulnerability: When a *sticky broadcast message (intent)* is sent, it is delivered to every registered receiver and is saved in the system to be provided to receivers that register for the message in the future. When the message is re-broadcasted with modification, the modified message replaces the original message in the system.

Exploit: A malicious broadcast receiver registers for the message at later time and retrieves any sensitive information in the message. Further, it can modify the contents of the message and re-broadcast to provide incorrect information to future receivers of the message.

A.1.10 Task affinity makes an app vulnerable to phishing attacks.

Vulnerability: A *task* is a collection (stack) of activities. When an activity is started, it is launched in a task. An activity can request that it be started in a specific task. This is known as *task affinity*. The task containing the displayed activity is moved to the background if none of the activities in that task are being displayed. When any activity from a task in the background is resumed, then the activity at the top of the task (and not the resumed activity) is displayed.

Exploit: An activity X in a malicious app requests to start itself in the same task as an activity Y in a benign app. When activity X is at the top of the task, any call to activity Y will cause activity X to be displayed to the user.

A.1.11 Task affinity and task re-parenting enables phishing and denial-of-service.

Vulnerability: An activity can request to always be at the top of a task. This is called *task re-parenting*. In such cases, when an activity from that task resumed, activity at the top of the task will be displayed to the user.

Exploit: An activity in a malicious app uses task affinity and task re-parenting to supersede activities from other apps in a task and launch a denial-of-service attack or a phishing attack.

A.1.12 Content Provider API allow unauthorized access.

Vulnerability: Content provider API provides a method `call` to call any provider-defined method. With a reference to the content provider, this method can be invoked without any restrictions.

Exploit: A malicious app uses `call` method to invoke content provider methods to access the underlying data even when it does not have specific permissions to access this data.

A.2 Storage

Android provides numerous options for storing application data. It provides

(1) *Internal Storage* to store data that is private to apps. Every time an application is uninstalled, its internal storage is emptied. Starting from Android 7.0, files stored in internal storage cannot be shared with other apps.
(2) *External Storage* as a data storage area that is common to apps. Its public partition is accessible to any app without any restrictions. Its private partition is only accessible to apps with a specific permission.

A.2.1 External storage allows data injection attack.

Vulnerability: Files stored in external storage can be modified by an app with (appropriate) access to external storage.

Exploit: A malicious app modifies external storage (e.g., add files) and the content in external storage (e.g., change files).

A.2.2 Writing sensitive information to external storage enables information leak.

Vulnerability: Files stored in external storage can be accessed by an app with (appropriate) access to external storage.

Exploit: A malicious app reads content from external storage.

A.3 System

System APIs help Android apps access low level features of the Android platform like process management, thread management, runtime permissions etc.

Every Android app runs in its own process with a unique Process ID (PID) and a User ID (UID). All components in an app run in the same process. A permission can be granted to an app at installation time or at run time. If an app is granted a specific permission at installation time, then all components of the app are granted the same permission. If component in an app is protected by a permission, only components that have been granted this permission can communicate with the protected component. If the permission is checked at runtime, then all components have to request for the required permission.

A.3.1 checkCallingOrSelfPermission method leaks privilege.

Vulnerability: Before servicing a request, a component protected by a permission uses `checkCallingOrSelfPermission` to check if the requesting component has the permission. This method returns true if the app containing the requesting component or the app containing the protected component has the given permission. When the app containing the protected component has the permission, the method will always return true.

Exploit: A malicious app accesses a component that is protected by permission P, is in an app that has permission P, and uses `checkCallingOrSelfPermission` to check for permission.

A.3.2 checkPermission method leaks privilege.

Vulnerability: Before servicing a request, a component protected by a permission uses `checkPermission` to check if the given PID

and UID pair has the permission. Typically, `getCallingPID` and `getCallingUID` methods of Binder API are used to retrieve PID and UID, respectively. When these methods are invoked in the main thread of an app, they return the IDs of the app and not the IDs of the calling app.

Exploit: A malicious app accesses a component that is protected by permission P, is in an app that has permission P, and uses `checkPermission` to check for permission in the main thread of the containing app.

A.3.3 enforceCallingOrSelfPermission method leaks privilege.

Vulnerability: Before servicing a request, a component protected by a permission uses `enforceCallingOrSelfPermission` to check if the requesting component has the permission. This method raises `SecurityException` if the app containing the requesting component or the app containing the protected component does not have the given permission. When the app containing the protected component has the permission, the method will complete without any exceptions.

Exploit: A malicious app accesses a component that is protected by permission P, is in an app that has permission P, and uses `enforceCallingOrSelfPermission` to enforce the permission.

A.3.4 enforcePermission method leaks privilege.

Vulnerability: Before servicing a request, a component protected by a permission uses `enforcePermission` to check if the given PID and UID pair has the permission. Typically, `getCallingPID` and `getCallingUID` methods of Binder API are used to retrieve PID and UID, respectively. When these methods are invoked in the main thread of an app, they return the IDs of the app and not the IDs of the calling app.

Exploit: A malicious app accesses a component that is protected by permission P, is in an app that has permission P, and uses `enforcePermission` to enforce the permission in the main thread of the containing app.

A.4 Web

Web APIs allow Android apps to interact with web servers both insecurely and securely (via SSL/TLS), display web content through *WebView* widget, and control navigation between web pages via *WebViewClient* class.

A.4.1 Incorrect hostname verification enables Man-in-the-Middle (MitM) attack.

Vulnerability: Android apps that use SSL/TLS for secure communication employ custom implementations of the `HostnameVerifier` interface. Such implementations perform custom checks on the given hostname in the `verify` method. When these custom checks are incorrect or weak (e.g., does not check hostname), then apps can end up connecting to malicious servers.

Exploit: An application takes advantage of incorrect/weak hostname verification and mounts a MitM attack.

A.4.2 Incorrect trust validation enables Man-in-the-Middle attack.

Vulnerability: An app uses custom implementation of the *TrustManager* interface to check if the presented certificates are valid and can be trusted. When these implementations are incorrect (e.g., an empty stub implementation), unknown certificates may be trusted.

Exploit: An application takes advantage of incorrect trust validation and mounts a MitM attack.

A.4.3 Allowing execution of unverified JavaScript code in WebView exposes app's resources.

Vulnerability: When an app uses `WebView` to display web content and any JavaScript code embedded in the web content is executed, the code is executed with the same permission as the `WebView` instance used in the app.

Exploit: An app injects malicious JavaScript code into the web content loaded in `WebView`, e.g., modify static web page stored on the device.

A.4.4 Ignoring SSL errors in WebViewClient enables Man-in-the-Middle (MitM) attack.

Vulnerability: When an app loads web content from a SSL connection via `WebView` and is notified of an SSL error while loading the content (via `onReceivedSslError` method of `WebViewClient`), the app ignores the error.

Exploit: An application takes advantage of ignored errors and mounts a MitM attack.

A.4.5 Lack of validation of resource load requests in WebView allows loading malicious content.

Vulnerability: When a resource (e.g., CSS file, JavaScript file) is loaded in a web page in `WebView`, the app does not validate the resource load request in `shouldInterceptRequest` method of `WebViewClient`. Consequently, any resource will be loaded into `WebView`.

Exploit: An application takes advantage of lack of validation of resource load requests and mounts a MitM attack.

A.4.6 Lack of validation web page load requests in WebView allows loading malicious content.

Vulnerability: When a web page is to be loaded into `WebView`, the app does not validate the web page load request in `shouldOverridUrlLoading` method of `WebViewClient`. Consequently, any web page provided by the server will be loaded into `WebView`.

Exploit: An application takes advantage of lack of validation of web page load requests and mounts a MitM attack.

Multi-objective search-based approach to estimate issue resolution time

Wisam Haitham Abbood Al-Zubaidi
University of Wollongong
Australia
whaa807@uowmail.edu.au

Hoa Khanh Dam
University of Wollongong
Australia
hoa@uow.edu.au

Aditya Ghose
University of Wollongong
Australia
aditya@uow.edu.au

Xiaodong Li
RMIT University
Australia
xiaodong.li@rmit.edu.au

ABSTRACT

Background: Resolving issues is central to modern agile software development where a software is developed and evolved incrementally through series of issue resolutions. An issue could represent a requirement for a new functionality, a report of a software bug or a description of a project task.

Aims: Knowing how long an issue will be resolved is thus important to different stakeholders including end-users, bug reporters, bug triagers, developers and managers. This paper aims to propose a multi-objective search-based approach to estimate the time required for resolving an issue.

Methods: Using genetic programming (a meta-heuristic optimization method), we iteratively generate candidate estimate models and search for the optimal model in estimating issue resolution time. The search is guided simultaneously by two objectives: maximizing the accuracy of the estimation model while minimizing its complexity.

Results: Our evaluation on 8,260 issues from five large open source projects demonstrate that our approach significantly ($p < 0.001$) outperforms both the baselines and state-of-the-art techniques.

Conclusions: Evolutionary search-based approaches offer an effective alternative to build estimation models for issue resolution time. Using multiple objectives, one for measuring the accuracy and the other for the complexity, helps produce accurate and simple estimation models.

1. INTRODUCTION

In software projects, an issue represents description of a bug or a security vulnerability (e.g. bug report issues), or a description of a new functionality (e.g. feature request or user story issues) or enhancements of an existing functionality, or a project task. Most of today's software projects are issue-driven where a project consists of a number of past issues (i.e. issues that have been closed), ongoing issues (i.e. issues that the team are working on), and new issues (i.e. issues that have just been created). Knowing when an issue will be resolved is important for many stakeholders. For example, the end-users may want to know when the new functionality they requested will be implemented. The bug reporters may be interested in learning when a particular bug will be fixed. The project managers may need to estimate the time it will take for completing a project task since these estimates are critical to their plan for costing and timing future releases.

Predicting when a particular issue will be resolved is however difficult. Existing practices in the industry often use the average resolution time of past issues, combined with a certain margin of error, as an estimate for the resolution time of the new issues. However, software issues may be significantly different from one another in their nature and the complexity of resolving them. Hence, the quality of the average resolution time as an estimator is often poor. Other existing practices heavily rely on experts' (e.g. project managers or experienced developers) subjective assessment to arrive at an estimate for the time and effort of resolving an issue. Relying on expert knowledge is however sometimes based on outdated experience and an underlying bias, thus may lead to inaccuracy in estimation. A number of software analytics techniques have recently been proposed to address this problem. These work (e.g. [16, 32]) mine the historical data generated when issues were reported and resolved. They identify features which characterize an issue and also influence on its resolution time. They then build machine learning models, train them using historical issues with known resolution time, and use them for future estimations.

Machine learners have also widely used for estimating the effort required for developing a complete software system (e.g. [17]). Recent approaches (e.g. [10, 28]) have employed a evolutionary, search-based approach to this problem. Largely inspired by *Sarro et. al.*'s work [28] done for effort estimation for the whole project, we employ a multi-objective search-based evolutionary approach to estimate the resolution time of each single issue in the project. Specifically, we leverage a meta-heuristic technique, namely

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PROMISE, November 8, 2017, Toronto, Canada

© 2017 ACM. ISBN 978-1-4503-5305-2/17/11…$15.00
DOI: https://doi.org/10.1145/3127005.3127011

genetic programming (GP) [18], to generate a large number of candidate estimation models, and search for the ones that are optimal with respect to a number of objectives. Differing from Sarro *et. al.* [28], we do *not* impose a fixed structure and depth of candidate estimation models.

We explore two objectives guiding our search algorithms. Similarly to *Sarro et. al.* [28], the *first objective* is to minimize the Sum of Absolute Errors, which measures the accuracy of an estimation model in terms of the differences between values (i.e. issue resolution time) estimated by the model and the values actually observed. The pressure of minimizing the estimation errors may however cause the solution model to adhere precisely to noisy data in the training set, which potentially make the model be excessively large and complex (hence, overfitting problems). While accuracy is critical for an estimation model, the Occam's Razor principle of *parsimony* also plays an important role here: the model needs to be expressed in a simple way, easy for software practitioners to interpret [23]. Hence, our *second objective* is to minimize the *complexity* of an estimation model, which can be measured in terms of the size of an expression tree representing the model. This is also another key difference from *Sarro et. al.*'s approach [28]. This second objective also leads to reduced computational costs since it encourages parsimonious (thus, computationally efficient) candidate solutions be generated. We name our approach *Multi-Objective Issue Resolution Time Estimator* (MOIRTE).

Our search-based approach outperforms the three common baselines (random guessing, and mean and median) and state-of-the-art techniques (linear regression, case-based reasoning, and random forests) in predicting issue resolution time. The evaluation was performed against a dataset of 8,260 issues which we collected from five different projects including four Apache Hadoop projects (Common, HDFS, MapReduce, Yarn) and one Apache Mesos project. Similarly to *Sarro et. al.*'s work [28], we use two standardized measures, Mean Absolute Error and Standardized Accuracy, to evaluate the performance of estimation models, and also use a non-parametric Wilcoxon test [4] and Vargha and Delaney's statistic [31] to demonstrate both the statistical significance and the effect size of the results.

The remainder of this paper is organized as follows. Section 2 formulates the problem of estimating issue resolution time. Section 3 describes our multi-objective approach to solve this problem using evolutionary algorithms. Section 4 reports on the experimental evaluation of our approach. Related work is discussed in Section 5 before we conclude and outline future work in Section 6.

2. ISSUE RESOLUTION TIME

In modern software development settings, software is developed through repeated cycles (iterative) and in smaller parts at a time (incremental), allowing for adaptation to changing requirements at any point during a project's life. A project has a number of iterations, in each of which, the development team resolves a number of *issues*. An issue could be requesting the implementation of a new functionality, fixing a bug or a security vulnerability, or refactoring the code.

Figure 1 shows the report of issue HADOOP-13353 in the Hadoop Common project. This issue report was recorded in the widely-used JIRA project management system. As can be seen from the report, this issue was created on 08 July

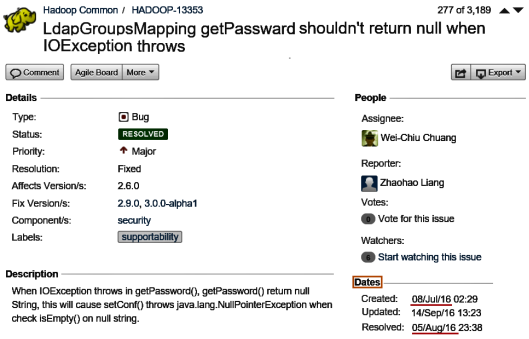


Figure 1: An example of a Hadoop Common issue recorded in JIRA. Note that we highlighted the dates when the issue were created and resolved.

2016 and resolved on 05 August 2016. This issue was a bug and its resolution was set to "Fixed", indicating that it was in fact a valid bug and has been fixed.

We would like to estimate how long it will take to resolve an issue using the following information provided with an issue report. These are common information which must be provided at the time an issue is created:

1. *Type*: each issue is assigned with a type (e.g. Bug, Task, Improvement, New feature, etc.) which indicates the nature of the task associated with resolving with the issue. For example, a "bug" issue reports a defect while a "new feature" describes a request for implementing a new functionality.

2. *Priority*: The issue's priority presents the order in which an issue should be attended with respect to other issues. In the projects we studied, there are 5 common types of priority: Blocker, Critical, Major, Minor, and Trivial. Issues with blocker priority should be more concerned than issues with major or minor priority since the former block other issues to be completed.

3. *Reporter*: We use reporter's reputation, a common feature which has been studied in previous work in mining bug reports. The intuition here is that poor issue reports may take longer time to resolve and reporters who frequently write them will accumulate such a reputation. We use the widely-used Hooimeijer's reporter reputation [14] as follows:

$$reputation(D) = \frac{|opened(D) \cap fixed(D)|}{|opened(D)| + 1}$$

   The reputation of a reporter D is measured as the ratio of the number of issues that D has opened and fixed to the number of issues that D has opened plus one.

4. *Title and description*: The title and description of an issue explains its nature and thus can be a good feature. We employ a common approach to translate an issue's title and description into the number of word

counts and use this as a feature. In addition, we also use the readability of the issue description as another feature. Readability is a quality indicator for issue reports [14]. We hypothesize that issues that are more difficult to understand will be more difficult to deal with and thus potentially take longer time to resolve. We use the Gunning fog readability metric [22] to measure an issue description's readability score. The lower Gunning fog score is, the easier to understand an issue description.

Note that other information associated with an issue report (e.g. assignee, fix versions, votes, watchers, etc.) can be used. However, some of these information are not mandatory (e.g. fix versions), or do not have any value at the time when an issue is created (e.g. assignee or the number of votes or watchers), which is the time we would like to make a prediction. Some of them are specific to an issue tracking system, while the features that we use here are commonly found in many issue-tracking systems.

3. APPROACH

3.1 Overview

Our approach falls under the search-based software engineering umbrella and is largely inspired by *Sarro et. al.*'s search-based approach [28] to estimate the effort required for completing a whole project. We however focus on the issue-level, i.e. estimating the resolution time for each issue in a project. Figure 2 gives an overview of our approach. We build a training set by collecting completed issues from a given project, and extracting their actual resolution time. The resolution time is the elapsed time when an issue was created and when it was resolved. We design a set of features characterizing an issue (see Section 2) and extract the values of these features at the time when the issue was created. We then iteratively generate candidate estimation models (by using a set of mathematical operators to combine the issue features) and search for the "best" estimation model with respect to the training set. This search process employs evolutionary algorithms which work based on the principle that a population of candidate solutions (also referred to as individuals) to an optimization problem is evolved toward better solutions, following Darwin's evolution theory. Each candidate solution has a number of properties (i.e. chromosomes or genotype) which can be mutated and altered to derive new candidate solutions. In our context of estimating issue resolution time, a candidate solution is an estimation model.

The estimation model found at the end of the search process is used for estimating the resolution time of new issues in the same project (within-project estimation) or in a different project (cross-project estimation). We will now describe the details of our approach.

3.2 Symbolic regression

Estimating issue resolution time can be considered as a regression problem: we need to model the relationship between the time required for resolving an issue and a set of features characterizing the issue. Here, we measure issue resolution time as a *continuous* number of days, and rely on five basic information associated with an issue: type, priority, reporter, title and description (see Section 2) to extract

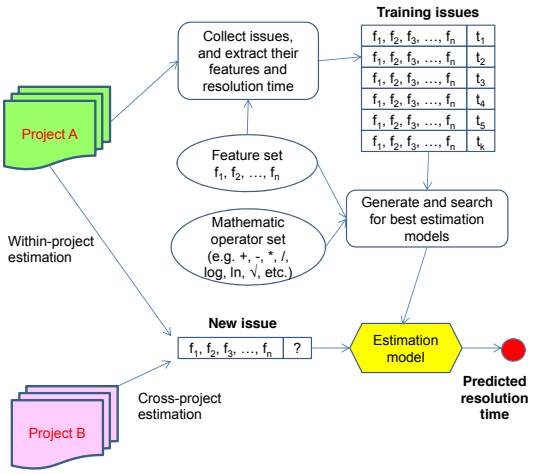


Figure 2: An overview of our approach

the features. Since the issue type and priority are categorical features, we need to perform additional steps to ensure the results from a regression model be interpretable. Specifically, we use one hot encoding to transform the issue type in a number of features (corresponding to the number of issue types), each of which represents one type and has a value of either 0 or 1. Since there are 7 issue types (i.e. bug, task, improvement, test, sub-task, new feature, and wish), this results in 7 features. For issue priority, we convert it into an ordinal value from 1 to 5 where 1 represents the least priority and 5 represents the most priority. In total, we have 13 numerical features (7 derived from issue type, 1 from priority, 1 from reporter reputation, 2 from the title and 2 for the description) characterizing an issue.

An estimation model can be viewed as a mathematical expression which combines those features of an issue and a set of mathematical operators to output a scalar value representing the time required for resolving an issue. We use a training dataset of past issues (with known resolution time) to *search* the space of those mathematical expressions to find the estimation model that best fits the training dataset. This model is then used to predict the time required for resolving new issues. This approach is commonly referred to as *symbolic regression* and has been previously used by Sarro *et. al.*'s [28] in estimating effort for the whole project. We adapted this approach from Sarro *et. al.*, but also made several key differences: (i) not imposing a fixed structure nor depth on candidate models; (ii) using a different second objective function to explicitly control the model's complexity; and (iii) using a wider range of mathematical operators.

Since each candidate estimation model is a mathematical expression, we represent it as an expression tree to facilitate the application of genetic operators (described in details later) to derive new candidates. Issue features are encoded as leafs of the tree and mathematical operators as its internal nodes. We thus use genetic programming [18], a metaheuristic algorithm in the family of evolutionary algorithms, which specifically deals with tree representation. We employ a wide range of thirteen mathematical operators ($+$, $-$, $*$, $/$,

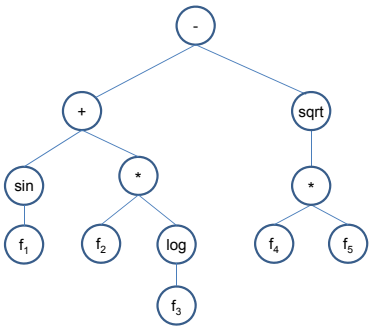


Figure 3: An example of expression tree representing a candidate estimation model. Note that f_i represents a feature of an issue.

exp, log, log_{10}, sin, cos, tan, $power$, $square$, $squareroot$), as opposed to only three operators used in Sarro *et. al.*'s [28]. Figure 3 shows an example of an expression tree representing a estimation model in which the resolution time of an issue is calculated as $(sin(f_1) + (f_2 * log(f_3))) - \sqrt{f4 * f5}$ where f_1, f_2, f_3, f_4 and f_5 are some of the thirteen issue features.

3.3 Fitness functions

The search for the best estimation model is guided by a number of fitness functions, which are used to compare if a candidate model is "better" than another one. We employ two fitness functions: one reflecting the accuracy of an estimation model in predicting issue resolution time and the other representing the model's complexity. The details of these fitness functions are described as below.

3.3.1 Sum of Absolute Errors

We use a training set of past issues (with known resolution time) to evaluate the accuracy of a candidate estimation model. A number of measures have been used to evaluate the predictive performance of an estimation model and can be classified in two groups: relative measures such as Mean of Magnitude of Relative Error or Root Mean Square Error (RMSE), or absolute measures like the Sum of Absolute Error (SAE). Previous work (e.g. [20, 28]) have suggested that the predictive performance of estimation models found by genetic algorithms is affected by the use of those different measures as a fitness function. Specifically, using relative measures as a fitness function has a negative impact on the model accuracy, while absolute measures appear to not have damaging effect [28]. Hence, similarly to [28], we chose to use the Sum of Absolute Error (defined below) as our first fitness function.

$$SAE = \sum_{i=1}^{N} |ActualTime_i - EstimatedTime_i| \quad (1)$$

where N is the number of issues in the training set, $ActualTime_i$ is the actual resolution time for issue i in the training set, and $EstimatedTime_i$ is the time estimated by a candidate model.

3.3.2 Tree size

Using an accuracy measure as the sole fitness function may result in excessively complex estimation models. The pressure of minimizing the estimation errors may lead to solution models that "overfit" the training data, which thus negatively affects the generalization performance of the models. It also takes more computational resources to store and evaluate complex models during the evolution process. In addition, software practitioners usually find complex estimation models difficult to understand and interpret [23].

The simplest method to control the complexity of estimation models is imposing a fixed limit on the depth or size (i.e. the number of nodes) of expression trees representing those models. Sarro *et. al.* [28] employed this approach by ensuring all the trees in the population having a fixed depth. This approach however suffers from a number of limitations. Determining a good value for the limit is very challenging. A small limit may prevent good solutions from being generated, while a large limit may still result in over-complex solutions. In addition, the process of eliminating non-conformance individuals from the population may create bias and adverse affect [11].

To control the balance between accuracy and complexity, we employ a second fitness function which measures the complexity of a solution estimation model. Since we represent an estimation model as an expression tree, the size of the tree can be used as a complexity indicator. Hence, the second fitness function returns the *size of a solution tree*, i.e. the number of nodes in the tree. For example, there are 12 nodes in the tree in Figure 3, hence its size is 12. The tree size reflects to some extends both the depth and width of a tree.

3.4 Evolutionary search

The search for an estimation model starts with an initial population in which each individual in the population is a candidate estimation model. The initial population is created by randomly generating a number of expression trees (each represents an individual) using the thirteen mathematical operators and thirteen issue features. The fitness values of each individual with respect to each of two fitness functions (see Section 3.3) are computed. The population is then undergone a selection process.

Selected individuals form parents to generate a new generation of individuals through the crossover and mutation operators. These genetic operators act directly on the expression trees to form new valid trees. The mutation operator chooses a node in the expression tree and substitutes the sub-tree at that node by a randomly generated sub-tree. Figure 4 shows an example of how the tree in Figure 3 is mutated. The crossover operators involve two parent trees and generate two offspring trees by exchanging selected branches (see Figure 5). Generated trees which give negative or invalid (e.g. division by zero) estimated time are assigned with a very large (infinite) sum of absolute errors. This would prevent those trees from being selected in the evolution process. This evolution process continues until a fixed number of generations has been reached.

We seek for estimation models that meet both objectives: high accuracy and low complexity. These solutions would belong to a Pareto front of estimation error and tree size (see Figure 6). To find such a Pareto front, we employ a widely-used multi-objective optimization algorithm, the non-dominated sorting genetic algorithm (NSGA-II) [6]. The NSGA-II algorithm works based on the principle of non-

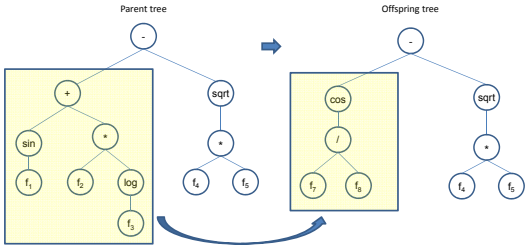


Figure 4: An example of the mutation operator

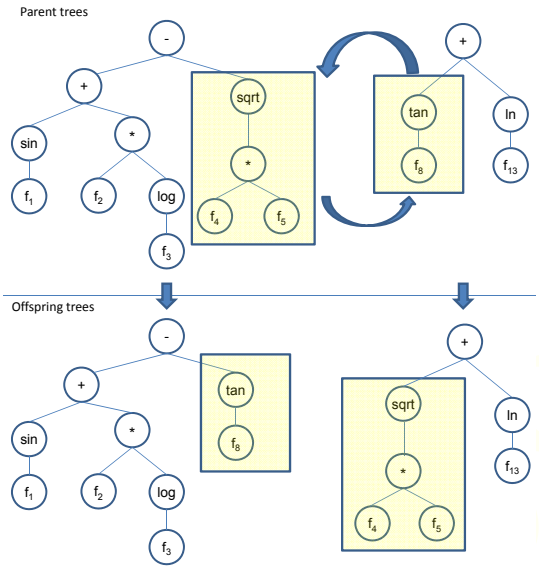


Figure 5: An example of the crossover operator

dominated sorting (Pareto dominance). In multi-objective optimization, an individual is said to *dominate* another individual if the former is better than the latter with respect to at least one objective, and not worse in the remaining objectives. For example, in Figure 6 estimation model E_1 does *not* dominate E_2 since the former is smaller than the latter in tree size but has greater sum of absolute errors. On the other hand, E_1 dominates E_4 since the E_1 is smaller than E_4 in tree size and also has smaller sum of absolute errors.

At each generation, NSGA-II sorts the current population into a number of non-dominated fronts (e.g. fronts 1, 2 and 3 in Figure 6). Each non-dominated front contains individuals which do not dominate each other. Individuals in the first non-dominated front dominate those in the second front, which in turn dominate individuals in the third front, and so on. Individuals in the same non-dominated front are assigned the same rank, which is the index of its front. For example, in Figure 6 estimation models E_1 and E_2 have the same rank 1, while E_4 has rank 2. The crowding distance of each individual is then computed as the sum of the distance between itself and its nearest neighbours on the same front. For example, the crowding distance for estimation model E_1 is the sum of the distances between E_1 and E_2 and between E_1 and E_3 (see Figure 6). NSGA-II [6] uses the rank and the crowding distance to define the crowded-comparison operator $\prec_c$, which is central to the operation of the algorithm.

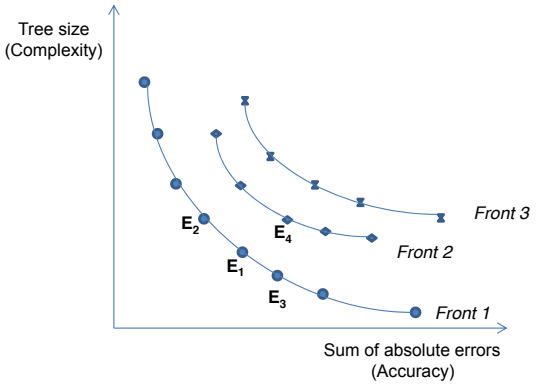


Figure 6: An example of non-dominated fronts

The partial order $\prec_c$ is defined as follows: for two individuals E_i and E_j, $E_i \prec_c E_j$ if and only if: (a) the rank of E_i is less than that of E_j; or (b) their ranks are the same (i.e. they are on the same non-dominated front) and the crowd distance of E_i is greater than that of E_j. The intuition here is that individuals with lesser domination rank are favoured in case when they are on different fronts, and individuals in a less dense region (i.e. higher crowding distance) are preferred in case when they are on the same front. This necessitates the pressure for the population to move towards the Pareto Front and spread along it. NSGA-II uses the crowded-comparison operator $\prec_c$ to guide the selection process for forming the offspring population. The next generation is selected from a combination of the parent and offspring operation. This is to minimize the possibility of losing a high quality solution. In the final generation, NSGA-II returns a set of non-dominated solutions. Choosing which one of these solutions to use is usually a user-specific decision. In our case, we choose to use the solution which has the lowest sum of absolute errors with respect to the training set. For more details on NSGA-II, we refer to the reader to [6].

Following the common practice [15], we used the following parameters. The size of the initial population is set to $10 * V$ where V is the number of features ($V = 13$ in our case, hence 130). The number of generations was set to $1,000 * V$, i.e. 13,000. Crossover probability was set to 0.9, mutation probability was 0.1, and reproduction probability was 0.2. We used tournament selection method. These are common values used in previous studies [15, 28]. We used an implementation of NSGA-II in the MOEA Framework[1].

4. EVALUATION

In this section, we report an empirical evaluation of our approach. We first describe the datasets we have built for

[1]http://moeaframework.org/index.html

our evaluation. We then discuss the experimental settings and performance measures. We then present the evaluation results which answer a range of research questions.

4.1 Datasets

We chose five projects, namely Common, HDFS, MapReduce, Yarn and Mesos from the well-known Apache to build a dataset of issues with known creation and resolved times. Those issues were recorded in the widely-used JIRA tracking system. We used the Representational State Transfer (REST) API provided by JIRA for querying the issues. After that we collected issue reports in JavaScript Object Notation (JSON) format. In terms of Common, HDFS, MapReduce and Yarn, the collected issues have the created date and resolved date up to September 9, 2016 and for Mesos is up to March 24, 2017. From these projects, we extracted a total of 16,858 issues. We excluded issues with a status other than "Resolved" to avoid collecting uncompleted issues and a resolution other than "Fixed" (e.g. "Duplicate" or "Not Fix", or "Invalid") in order to collect only "real" issues. In the end, we included 8,260 issues into our dataset. We have calculated the duration time for each issue by subtracting its resolved time from its created time. The resolution time is measured in days.

Table 1: Descriptive Statistics of our Dataset

Projects	# selected issues	Mean	Median	Mode	Std
Common	1,402	37.19	10	2	62.24
HDFS	2,334	28.66	7	2	55.49
MapReduce	635	45.68	14	2	72.62
Yarn	930	46.82	17	2	69.32
Mesos	2959	52.97	18	1	77.13
Total	8,260				

Table 1 summarizes the descriptive statistics of our dataset in terms of mean, median, mode and standard deviation of resolution time. The median resolution times range from 7 to 18 days in the five projects, while the mean resolution times were from 28 to 52 days. Although the resolution times varied quite substantially (standard deviation in all projects were above 55), most of the issues were resolved within 2 days. We will make the data sets used in this study be publicly available for the research community once the paper has been accepted.

4.2 Experimental settings and measures

For each project, we applied a cross validation process which is a widely employed technique to validate an estimation or prediction model. Specifically, we divided our dataset into 10 folds and applied cross-validation (i.e. used nine folds for training and the remaining one fold for testing) to reduce the estimation instability and bias. We ran each algorithm 30 times and took the median result. Similarly to previous work in effort estimation (e.g. [8, 19, 28, 29]), we employed two widely-used standardised measures, Mean Absolute Error (MAE) and the Standardized Accuracy (SA). They are defined as below.

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|ActualTime_i - EstimatedTime_i| \quad (2)$$

where N is the total number of issues used in the test set. Estimation models with lower MAE are better in terms of accuracy.

Standardized Accuracy (SA) measures how good an estimation model is with respect to random guessing:

$$SA = (1 - \frac{MAE}{MAE_{rguess}}) \times 100 \quad (3)$$

where MAE_{rguess} is the MAE averaging a large number of random guesses. Estimation models with larger SA are more useful.

To compare the performance between different estimation models, we followed [28] and tested the statistical significance of the absolute errors produced from those estimation models using the Wilcoxon Signed Rank Test [4]. Wilcoxon test does not require the data be normally distributed, which is suitable to our data. We set the confidence limit at 0.05 (i.e. $p < 0.05$). We also applied the Vargha and Delaney's $\hat{A}_{12}$ non-parametric effect size measure [31]. This non-parametric effect size measure is suitable for testing randomized algorithms in software engineering, especially in the context of effort estimation [1, 28]. The $\hat{A}_{12}$ measures the probability that, estimation model E_i achieves better accuracy (i.e. smaller absolute errors) than estimation model E_j using the following formula [31]:

$$\hat{A}_{12} = (r_1/m - (m+1)/2)/n \quad (4)$$

where r_1 is the rank sum of (test) issues where E_i produces smaller absolute error than E_j does, and m and n are the number of issues tested for E_i and E_j respectively. Note that $\hat{A}_{12} = 0.5$ when the two estimation models perform in an equivalent way, while $\hat{A}_{12} > 0.5$ when E_i perform better than E_j.

4.3 Results

In this section, we present the results of our experimental evaluation[2] in terms of answering the following research questions.

RQ1. *Is the multi-objective search-based approach suitable for estimating issue resolution time?*

To answer this question, we compared our multi-objective search-based approach against three common baselines: *Random Guessing*, and *Mean* and *Median*, which are often used in effort estimation [28]. Random guessing is a naive technique for estimation [28, 29]. It performs random sampling over a set of issues with a known resolving time, choosing randomly one issue from the sample, and uses the resolving time of that issue as the estimate of the target issue. Random guessing does not use any information associated with the target issue. Thus, any useful estimation model should outperform random guessing. Mean and Median estimations are also commonly used as baseline benchmarks for effort estimation. They use the mean or median resolving time of the past issues to estimate the resolving time of the target issue.

As can be seen from Table 2, our approach MOIRTE produced better estimations in terms of MAE and SA than

[2]All the experiments were run on a Microsoft Windows 10 Home PC with an Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz and 8.00 GB RAM.

the Random, Mean and Median did. MOIRTE consistently outperformed all the three baselines in all five projects. Averaging across all projects, MOIRTE achieved an accuracy of 24.17 MAE and 58.45 SA, while the best of the baselines (Median) achieved 38.65 MAE and 33.49 SA.

Table 2: Mean Absolute Error (MAE) and Standard Accuracy (SA) values achieved by our approach MOIRTE, the baselines (Mean and Median Time), and state-of-the-art techniques: Linear Regression (LR), Case-Based Reasoning (CBR) Weiss *et. al.*, and Random Forests (RF). MAE and SA results are also included for the the single objective search with unlimited depth (GP-SAE) – discussed later in RQ3.

Project	Measures	MOIRTE	GP-SAE	Mean	Median	LR	CBR	RF
Common	MAE	**22.35**	30.13	41.52	34.25	40.68	44.63	33.89
	SA	**57.14**	42.23	20.38	33.80	21.99	14.42	35.02
HDFS	MAE	**17.80**	23.36	33.88	27.19	31.25	33.46	26.10
	SA	**58.02**	44.91	20.09	35.88	26.29	21.07	38.44
Mapreduce	MAE	**23.37**	37.52	49.99	42.40	50.46	58.06	39.38
	SA	**63.26**	41.02	21.41	33.34	20.67	8.72	38.10
Yarn	MAE	**23.97**	36.05	49.89	41.53	44.84	49.39	39.18
	SA	**60.01**	39.85	16.76	30.72	25.18	17.59	34.62
MESOS	MAE	**33.35**	42.71	57.08	47.88	54.82	59.35	44.35
	SA	**53.82**	40.86	20.95	33.69	24.08	17.81	38.58

LR:Linear Regression, CBR:Case-Based Reasoning, RF:Random Forest

Table 3: Comparison between our approach MOIRTE and the three baseline techniques using Wilcoxon test and $\hat{A}_{12}$ effect size (in brackets). This comparison is also done for the single-objective genetic programming algorithm with unlimited depth (GP-SAE), which is discussed in RQ3.

Project	Technique	Mean	Median	Random Guessing
Common	MOIRTE	<0.001 [0.76]	<0.001 [0.56]	<0.001 [0.75]
	GP-SAE	<0.001 [0.72]	0.0167 [0.52]	<0.001 [0.72]
HDFS	MOIRTE	<0.001 [0.79]	<0.001 [0.55]	<0.001 [0.78]
	GP-SAE	<0.001 [0.76]	<0.001 [0.52]	<0.001 [0.75]
MapReduce	MOIRTE	<0.001 [0.80]	<0.001 [0.60]	<0.001 [0.72]
	GP-SAE	<0.001 [0.73]	0.4423 [0.52]	<0.001 [0.68]
Yarn	MOIRTE	<0.001 [0.78]	<0.001 [0.59]	<0.001 [0.79]
	GP-SAE	<0.001 [0.75]	0.0033 [0.54]	<0.001 [0.76]
Mesos	MOIRTE	<0.001 [0.76]	<0.001 [0.59]	<0.001 [0.78]
	GP-SAE	<0.001 [0.72]	<0.001 [0.54]	<0.001 [0.73]

Table 3 shows the results of the Wilcoxon test and the corresponding $\hat{A}_{12}$ effect size to measure the statistical significance and effect size (in brackets) of the improved accuracy achieved by MOIRTE over the three baselines. In all cases, our approach MOIRTE significantly outperforms the baselines with $p < 0.001$ and effect sizes greater than 0.5. We also compared the performance of the single-objective genetic programming algorithm using the sum of absolute error as the single objective function (GP-SAE), which will be discussed later in *RQ*3. The results clearly suggest that our approach outperforms the naive benchmarks. The next step is thus assessing if it outperforms the existing techniques in predicting issue resolving time, which leads us to the second research question.

RQ2. *Does the search-based approach provide more accurate estimates than existing techniques used in predicting issue resolution time?*

Table 4: Comparison between our approach MOIRTE and the state-of-the-art techniques (LR, CBR Weiss *et. al.* and RF) using Wilcoxon test and $\hat{A}_{12}$ effect size (in brackets. This comparison is also done for the single-objective genetic programming algorithm with unlimited depth (GP-SAE), which is discussed in RQ3.

Project	Technique	LR	CBR	RF
Common	MOIRTE	<0.001 [0.75]	<0.001 [0.67]	<0.001 [0.72]
	GP-SAE	<0.001 [0.71]	<0.001 [0.62]	<0.001 [0.68]
HDFS	MOIRTE	<0.001 [0.74]	<0.001 [0.64]	<0.001 [0.70]
	GP-SAE	<0.001 [0.71]	<0.001 [0.61]	<0.001 [0.67]
Mapreduce	MORITE	<0.001 [0.79]	<0.001 [0.72]	<0.001 [0.73]
	GP-SAE	<0.001 [0.72]	<0.001 [0.66]	<0.001 [0.67]
Yarn	MORTIE	<0.001 [0.75]	<0.001 [0.70]	<0.001 [0.72]
	GP-SAE	<0.001 [0.70]	<0.001 [0.66]	<0.001 [0.68]
Mesos	MOIRTE	<0.001 [0.74]	<0.001 [0.66]	<0.001 [0.68]
	GP-SAE	<0.001 [0.69]	<0.001 [0.62]	<0.001 [0.62]

To answer this question, we compare our search-based approach against the three existing techniques that have been widely used in effort estimation: Linear Regression (LR), Case-Based Reasoning (CBR) and Random Forests. For case-based reasoning, we used k-nearest-neighbour(kNN) as done in the seminal work of Weiss *et. al.* [32]). Random Forests (RF) is chosen since it is currently the most effective method for effort estimation [17]. RF is an ensemble method which combines the estimates from multiple estimators. RF achieves a significant improvement over the decision tree approach by generating many classification and regression trees. Each tree is built on a random resampling of the data, with a random subset of variables at each node split. Then through averaging, tree predictions can be aggregated. Note that all these three prediction models use the same set of features as in our approach.

We used the implementation of linear regression and kNN provided with Weka [13]. Since it is tedious to find the optimal hyperparameters for these classification or regression algorithms, we automated this process using Weka's MultiSearch, a meta-classifier for tuning hyperparameters of a given base classifier or regressor. Specifically, for linear regression, we focused on tuning two hyperparameters: ridge (ranging from 1e-7 to 10) and selection method (with three methods: none, greedy and M5). For kNN, we used the brute force search algorithm (i.e. LinearNNSearch), Euclidean distance for the distance function, and tuned k number ranging from 1 to 64. We experimented with two implementations of Random Forests: one provided in Weka and the other written in Matlab[3], and found that the Matlab implementation produced better results with our data. Thus we chose this implementation to compare against our approach. We tuned Random Forests from 1 to 500 trees. All tuning was done using training data.

The MAE and SE results (see Table 2) shows that Random Forests is the best performer amongst the three existing techniques. However, when comparing against our approach, Random Forests consistently produced higher MAE and lower SA than MOIRTE in all five projects. The improvement brought by our approach over Random Forests was from 24% to 40% in MAE, with an average improvement of 34% in the five projects we studied. The Wilcoxon

[3]https://github.com/ami-GS/randomforest-matlab

test (see Table 4) also confirms this: the improvement of MOIRTE over LR, CBR, and RF is significant ($p < 0.001$) with the effect size greater than 0.7 in most cases. These results suggest that our multi-objective search-based approach offers an alternative and effective technique to issue resolution time estimation. The improvement may be due to its capability of capturing the nonlinear relationship between issue features and resolution time. Also, our approach does not carry any human biases nor affected by unknown domain-specific knowledge by not imposing any prior model structure and size.

RQ3. *Does the multi-objective approach produce more accurate estimates than the single-objective approach in predicting issue resolution time, and at the same time produce solutions of lower complexity?*

This question aims to investigate whether using the tree size as the second objective offers significant benefits in terms of both accuracy and complexity. To do so, we also implemented the traditional single-objective genetic programming algorithm using the sum of absolute error as the objective function. We name this alternative approach as GP-SAE. We experimented with two variants of this algorithm, one with unlimited depth tree and the other with a limited depth tree of 10. The former represents a method with no control on the complexity of the solutions, while the latter represents a technique with a constant limit on the tree size.

Table 5: Comparison between our multi and single objective (MOIRTE vs. GP-SAE) using Wilcoxon test and $\hat{A}_{12}$ effect size (in brackets). The second row reports the average tree size (i.e. the number of nodes) of a solution estimation model – the first number produced by MOIRTE while the second number produced by GP-SAE.

	GP-SAE (unlimited depth)				
	Common	**HDFS**	**Mapreduce**	**Yarn**	**Mesos**
MOIRTE	<0.001 [0.55]	<0.001 [0.54]	<0.001 [0.61]	<0.001 [0.57]	<0.001 [0.58]
Tree size	13 vs. 415	14 vs. 498	18 vs. 328	17 vs. 306	24 vs. 386
	GP-SAE (depth = 10)				
	Common	**HDFS**	**Mapreduce**	**Yarn**	**Mesos**
MOIRTE	<0.001 [0.52]	<0.001 [0.53]	<0.001 [0.59]	<0.001 [0.56]	<0.001 [0.56]
Tree size	13 vs. 38	14 vs. 46	18 vs. 42	17 vs. 30	24 vs. 81

Results from Tables 2, 3 and 4 show that the single objective approach with unlimited depth tree (GP-SAE) even outperforms all the baselines and state-of-the-art techniques. However, using tree size as the second objective has brought significant improvement in estimation accuracy. Across the five projects we studied, MOIRTE achieved from 22% – 37% improvement over GP-SAE in MAE (see Tables 2). The Wilcoxon test also confirmed that the improvement brought by using our multi-objective approach is significant ($p < 0.001$) in all cases with effect size from 0.54 to 0.61 (see the first row in Table 5).

Our approach of using tree size as the second objective is effective not only in improving the accuracy of the estimation model but also in the reducing its complexity. As can be seen in Table 5, the average tree sizes of the solution estimation models produced by MOIRTE were significantly less than those produced GP-SAE (e.g. 13 nodes versus 415 nodes for the Hadoop Common project). The approach of setting a fixed depth limit (depth = 10) is also not as effective as the multi-objective approach. Although using this approach reduced the tree size of the solution estimation models, it is still inferior to the multi-objective approach in terms of accuracy (see the bottom part of Table 5). These results clearly demonstrate the benefit of using our multi-objective approach in terms of producing both accurate and simple estimation models.

RQ4. *Is the proposed approach suitable for cross-project estimation?*

Estimating issue resolution time in new projects is often difficult due to lack of training data. One common technique to address this problem is training a model using data from a (source) project and applying it to the new (target) project. We employed this technique and performed 20 cross-project estimation experiments.

Table 6 reports the MAE produced by our approach in cross-project settings. For example, when we used the issues from Hadoop Common for training to obtain an estimation model and then applied this model for the issues in Hadoop HDFS, our approach achieved 19.31 MAE. In the within-project setting, i.e. the training was done using Hadoop HDFS, our approach achieved 17.80 MAE (see Table 2). The decreased performance in this case was relatively small (only 7%), which is also observed in the other cases. These results suggest that our approach can be used for cross-project estimation with a small sacrifice in accuracy. We also observe that estimations done cross the four projects in Hadoop were more accurate than those performed cross one of the four projects in Hadoop and the Apache Mesos project. This may be due to the fact that the four Hadoop projects share many commonalities than those in a totally different project like Mesos.

Table 6: MAE produced by our approach MOIRTE in cross-project settings, trained using a project in the row and tested on a project in the column.

Project	**Common**	**HDFS**	**Mapreduce**	**Yarn**	**Mesos**
Common		19.31	27.54	34.20	42.08
HDFS	28.30		30.23	33.94	42.09
Mapreduce	29.45	21.86		32.72	40.47
Yarn	27.21	22.90	27.56		39.90
Mesos	30.10	26.87	35.63	35.15	

4.4 Threats to validity

To mitigate threats to construct validity, we used real world data from issues reported in large open source projects. We collected the common issue features and the actual time took to resolve issues. In terms of the conclusion validity, we carefully selected unbiased error measures and applied a number of statistical tests to verify our assumptions [2]. Our study was performed on five datasets of different sizes. Furthermore, we carefully followed recent best practices in evaluating effort estimation models (e.g. [25]) to miminize conclusion instability. Another threat is related to the random initialization of the first generation. Therefore, a single run of an experimental study may deliver results that can be affected either by the favorable initial random selection or the bad randomly selected point [5]. To avoid this problem, we have conducted a multiple run (30 runs in this study) and chose the median result.

To overcome the external validity threat, we have considered a large number of issues from five different projects.

The size and the complexity of these issues are also significantly diverse. By this way, different contexts can be characterised by some specific projects and human factors (e.g. team structure and communication, time and other constraints, and so on). However, we cannot claim that our datasets are representative of all kind of software projects, and that our results can generalize to all software projects, especially those in commercial settings.

5. RELATED WORK

Predicting issue resolution time could be considered as a form of software effort estimation. Research in software effort estimation dates back several decades and they can generally be divided into model-based and expert-based methods [24]. Model-based approaches leverages data from old projects to make predictions about new projects. Expert-based methods rely on human expertise to make such judgements. Most of the existing work (e.g. [10, 17, 28]) in effort estimation focus on waterfall-like software development. These approaches estimate the effort required for developing a complete software system, relying on a set of features manually designed for characterizing a software project. This contrasts with our work since we estimate a single issue in the project at a time.

There is an emerging interest in predicting the fixing time of a bug, which was initiated by the work of Weiss *et. al.* [32]. These work (e.g.[7, 30, 33, 34] use machine learning techniques (e.g. kNN in [32] or Random Forests in [3, 16, 21]) to build their prediction models. For example, the work in [32] estimates the fixing time of a bug by finding the previous bugs that have similar description to the given bug (using text similar techniques) and using the known time of fixing those previous bugs. Using decision trees and other machine learning techniques, the work in [27] predict the lifetime of Eclipse bugs based on several primitive features of a bug such as severity, component, and number of comments. The work in [21] explored a different set of issue features including location, reporter and description. The time when the prediction is made also affect the predictive performance as shown in the study in [12]. They tested the predictive models with initial bug report data as well as those with post-submission information and found that inclusion of post-submission bug report data of up to one month can further improve prediction models. Most of those techniques used classifiers which do not deal with continuous response variables, they need to discretize the fix-time into categories, e.g. within 1 month, 1 year and more than 1 year as in [21]. This is one of the key difference to our work since we are able to predict the exact resolution time. In addition, our work offers an alternative in which we propose a search-based evolutionary approach to the problem.

Our work falls in the area of search-based software engineering where substantial work has been done and a range of them focused on software effort estimation. Most of the recent work in the context of effort prediction can be found in the review paper of Ferrucci et. al. [10]. While most of existing work in this space use single-objective search, a few of them (e.g. [26, 28]) has recently proposed multi-objective search approach to effort estimation. For example, a recent study done by Sarro *et. al.* [28] employed NSGA-II with two objectives sum of absolute error and confidence interval. Our work is largely inspired by Sarro *et. al.*'s work in the use of multi-objective search, sum of absolute error as an objective function, and standardized performance measures such as MAE, SA, Wilcoxon and effect size tests. There are however several key differences from our work and Sarro *et. al.*'s work. First, we built effort (time) estimation models for a single software issue rather than for the whole project (as done by Sarro *et. al.*). This makes our work more relevant and applicable to the modern agile software development settings where the focus is at the issue level. Second, we used the tree size as the second objective function to simultaneously manage the parsimonious and accuracy of generated estimation models. Hence, our approach does not impose any fixed structure or depth on the candidate models, which is different from Sarro *et. al.*'s approach. In addition, while Sarro *et. al.* used only three mathematical operators, we used a wider range of thirteen mathematical operators and thus accommodate a larger search space.

6. CONCLUSIONS AND FUTURE WORK

We have proposed a multi-objective search-based approach to estimate issue resolution time. Our approach leverages evolutionary algorithms to find robust estimation models. The search is guided simultaneously by two contrasting objectives: maximizing the accuracy of an estimation model and minimizing the complexity of the estimation model. A comprehensive evaluation done in five large open source projects with 8,260 issues demonstrates that our approach significantly (i.e. with $p < 0.001$) outperforms not only the three common naive baselines but also three state-of-the-art techniques. Our results also demonstrate the benefit of using the complexity measure as the second fitness function since this approach produced more accurate but less complex estimation model than the single-objective approach did. Results from our cross-project experiments also suggest that our approach is also applicable for cross-project estimation.

Future work would involve validating these results with some additional projects, especially those in the commercial settings. We also plan to investigate the use of other objectives such as confidence interval (as used in previous work [28]) and other complexity measures (e.g. order of nonlinearity). To compare the Pareto front of solutions returned by those approaches against those by our approach, we plan to use the evaluation measures recently proposed by Ferrucci et al.[9] such as contributions, hypervolume, and generational distance. We will also explore the use of other multi-objective evolutionary algorithms as part of our future work.

7. REFERENCES

[1] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 1–10. IEEE, 2011.

[2] A. Arcuri and L. Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[3] M. Choetkiertikul, H. K. Dam, T. Tran, and A. Ghose. Characterization and prediction of issue-related risks in software projects. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR)*, pages 280–291. IEEE, 2015.

[4] J. Cohen. Statistical power analysis for the behavioral sciences. 1988, hillsdale, nj: L. *Lawrence Earlbaum Associates*, 2.

[5] M. de Oliveira Barros and A. C. Dias-Neto. 0006/2011-threats to validity in search-based software engineering empirical studies. *RelaTe-DIA*, 5(1), 2011.

[6] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[7] A. Dehghan, K. Blincoe, and D. Damian. A hybrid model for task completion effort estimation. In *Proceedings of the 2nd International Workshop on Software Analytics*, pages 22–28. ACM, 2016.

[8] J. J. Dolado, D. Rodriguez, M. Harman, W. B. Langdon, and F. Sarro. Evaluation of estimation models using the minimum interval of equivalence. *Applied Soft Computing*, pages 1–12, 2016.

[9] F. Ferrucci, M. Harman, J. Ren, and F. Sarro. Not going to take this anymore: Multi-objective overtime planning for software engineering projects. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 462–471, Piscataway, NJ, USA, 2013. IEEE Press.

[10] F. Ferrucci, M. Harman, and F. Sarro. Search-based software project management. In *Software Project Management in a Changing World*, pages 373–399. Springer, 2014.

[11] C. Gathercole and P. Ross. An adverse interaction between crossover and restricted tree depth in genetic programming. In *Proceedings of the 1st Annual Conference on Genetic Programming*, pages 291–296, Cambridge, MA, USA, 1996. MIT Press.

[12] E. Giger, M. Pinzger, and H. Gall. Predicting the fix time of bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, pages 52–56. ACM, 2010.

[13] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.

[14] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *Proceedings of the 22 IEEE/ACM international conference on Automated software engineering (ASE)*, pages 34 – 44. ACM Press, nov 2007.

[15] S.-J. Huang and N.-H. Chiu. Optimization of analogy weights by genetic algorithm for software effort estimation. *Information and software technology*, 48(11):1034–1045, 2006.

[16] R. Kikas, M. Dumas, and D. Pfahl. Using dynamic and contextual features to predict issue lifetime in github projects. In *Proceedings of the 13th International Workshop on Mining Software Repositories*, pages 291–302. ACM, 2016.

[17] E. Kocaguneli, T. Menzies, and J. W. Keung. On the value of ensemble effort estimation. *IEEE Transactions on Software Engineering*, 38(6):1403–1416, 2012.

[18] J. R. Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[19] W. B. Langdon, J. Dolado, F. Sarro, and M. Harman. Exact mean absolute error of baseline predictor, marp0. *Information and Software Technology*, 73:16–18, 2016.

[20] C. Lokan. What should you optimize when building an estimation model? In *11th IEEE International Software Metrics Symposium (METRICS'05)*, pages 1–10. IEEE, 2005.

[21] L. Marks, Y. Zou, and A. E. Hassan. Studying the fix-time for bugs in large open source projects. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, page 11. ACM, 2011.

[22] D. R. McCallum and J. L. Peterson. Computer-based readability indexes. In *Proceedings of the ACM'82 Conference*, pages 44–48. ACM, 1982.

[23] T. Menzies. Occam's razor and simple software project management. In G. Ruhe and C. Wohlin, editors, *Software Project Management in a Changing World*, pages 447–472. Springer, 2014.

[24] T. Menzies, Z. Chen, J. Hihn, and K. Lum. Selecting best practices for effort estimation. *IEEE Transactions on Software Engineering*, 32(11):883–895, 2006.

[25] T. Menzies and M. Shepperd. Special issue on repeatable results in software engineering prediction. *Empirical Software Engineering*, 17(1):1–17, 2012.

[26] L. L. Minku and X. Yao. Software effort estimation as a multiobjective learning problem. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4):35, 2013.

[27] L. D. Panjer. Predicting eclipse bug lifetimes. In *Proceedings of the Fourth International Workshop on mining software repositories*, page 29. IEEE Computer Society, 2007.

[28] F. Sarro, A. Petrozziello, and M. Harman. Multi-objective software effort estimation. In *Proceedings of the 38th International Conference on Software Engineering*, pages 619–630. ACM, 2016.

[29] M. Shepperd and S. MacDonell. Evaluating prediction systems in software project estimation. *Information and Software Technology*, 54(8):820–827, 2012.

[30] Y. Tian, D. Lo, and C. Sun. Drone: Predicting priority of reported bugs by multi-factor analysis. In *ICSM*, pages 200–209, 2013.

[31] A. Vargha and H. D. Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[32] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 1. IEEE Computer Society, 2007.

[33] X. Xia, D. Lo, E. Shihab, X. Wang, and B. Zhou. Automatic, high accuracy prediction of reopened bugs. *Automated Software Engineering*, 22(1):75–109, 2015.

[34] X. Xia, D. Lo, X. Wang, X. Yang, S. Li, and J. Sun. A comparative study of supervised learning algorithms for re-opened bug prediction. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 331–334. IEEE, 2013.

Context-Centric Pricing: Early Pricing Models for Software Crowdsourcing Tasks

Turki Alelyani
Stevens Institute of Technology
Hoboken NJ, USA
talelyan@stevens.edu

Ke Mao
University College London
London, UK
k.mao@cs.ucl.ac.uk

Ye Yang
Stevens Institute of Technology
Hoboken, NJ, USA
Ye.Yang@stevens.edu

ABSTRACT

In software crowdsourcing, task price is one of the most important incentive to attract broad worker participation and contribution. Underestimating or overestimating a task's price may lead to task starvation or resource inefficiency. Nevertheless, few studies have addressed pricing support in software crowdsourcing. In this study, we propose Context-Centric Pricing approach to support software crowdsourcing pricing based on limited information available at early planning phase, i.e. textual task requirements. In the proposed approach, the global models include a list of 6 pricing factors and employ different natural language processing techniques for prediction; in addition, local models can be derived w.r.t. more relevant context, i.e. a set of similar tasks identified based on Topic modeling and we evaluate 7 predictive models. The proposed approach is evaluated on 450 software tasks extracted from TopCoder, the largest software crowdsourcing platform. The results show that: 1) the proposed models can be used at early crowdsourcing planning phase, when information on traditional metrics are not available; 2) the best model achieves 65% in accuracy measure; 3) the local model exhibits 27% increases superior to the global model. The proposed work can stimulate future research into crowdsourcing pricing estimation and inform ideas for crowdsourcing decision-makers.

CCS CONCEPTS

D.2.9 [**Software Engineering**]: Management – *life cycle, productivity.*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions fromPermissions@acm.org.
PROMISE , November 8, 2017, Toronto, Canada
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5305-2/17/11…$15.00
https://doi.org/10.1145/3127005.3127012

KEYWORDS

Crowdsourcing, pricing, software measurement

1 INTRODUCTION

New advances in information and communication technology have fostered more innovative crowdsourcing systems that can harness the intelligence of online communities. Thus, software production is no longer centralized or limited to specific corporations, times of day, or numbers of developers. Instead, decentralized systems draw from large groups of people connecting through the Internet.

The term 'crowdsourcing' was coined by business journalist Jeff Howe in 2006 [4] to define a development model that taps into a potentially large online market in which hundreds of thousands of geographically disparate developers can contribute services or input to a project. Requesters in crowd-sourcing markets typically outsource their tasks, which vary from micro-tasks [1] (i.e., image tagging) to developing and testing software [2,3], to anonymous workers.

Several crowdsourcing platforms give workers monetary incentives in exchange for their project participation: TopCoder (the largest software crowdsourcing platform), Amazon Mechanical Turk, eLance, oDesk, vWorker, Guru, Upwork, TaskCity, Taskcn, etc. Despite the success of these platforms in attracting talented individuals to participate and produce different products, studies show that crowdsourcing platforms are still challenged to design effective pricing systems for their different tasks, sometimes due to requesters' constraints and the online market setting of a particular project [2]. Optimal pricing mechanism is needed to maximize the benefits of tasks requesters and workers. Underpricing projects may undermine workers willingness to join and contribute which could affect the entire system.

Recent studies show that appropriate pricing mechanism is a key issue facing software crowdsourcing platforms [2,3,5]. So far most of the existing pricing options seem quite broad and need further investigation [2]. This partly

[1] https://www.mturk.com

[2] https://www.topcoder.com
[3] https://www.utest.com

due to the fact that some of the previously proposed pricing techniques pay little attention to the variations in crowdsourcing contexts. One major variation can be exhibited in the micro-tasks versus complex software tasks. Furthermore, there is a lack of tools to effectively price different tasks based on their complexity and size. This can establish a limitation to the possible value of the existing pricing techniques for software crowdsourcing projects.

Software crowdsourcing has no particular standard for price prediction, and some crowdsourcing pricing is based on rule-of-thumb strategies that cannot price tasks effectively [1]. This may cause the task to be canceled or to undergo what is called "task starvation". Thus, providing an accurate pricing tool/technique may limit some uncertainties which might diminish the productivity of software crowdsourcing platforms. One related work to ours that attempted to address software crowdsourcing pricing was done by Mao et al. [3]. The study introduced 16 cost-drivers, Table 1, and proposed 8 different machine learning techniques to estimate the price for crowdsourcing component development tasks. However, the major limitation of Mao's study is that each component development task is associated with an earlier component design task. In the early crowdsourcing planning phase, such information is generally unavailable which greatly constraints the applicability of this model. Especially, as crowdsourcing platforms evolve, more and more recent typical crowdsourcing processes do not employ this kind of design-development task mapping.

Table 1. 16 Cost Drivers in Four Categories [3]

Category	Variables	Meaning
DEV	*ISUP*	whether the task at component update
	ISJA	whether the language is java
	ISCS	whether the language is C#
QLY	*SCOR*	score of winner is design phase
	WRAT	rating of the winner in design phase
	REGI	number of registrants in design phase
	SUBM	number of submissions in design phase
CPX	*TECH*	number of technologies to be used
	DEPE	number of component dependencies
	REQU	number of pages of requirements specs
	COMP	number of pages of component specs
	SEQU	number of sequence diagrams of the design
	EFRT	winner's effort (in days) in design phase
	SUML	size of UML design file in KB
	SIZE	estimated size in KSLOC
PRE	*AWARD*	winner's award in $ of design phase

In this paper, a new approach called Context-Centric Pricing (CCP) is proposed to overcome its limitation and provide an effective way to estimate crowdsourced software task price in early stages. Our CCP approach provides models that learn from the early information in the task description. It is a context centric since we aim at constructing a more relevant local context for task pricing using similar historical tasks. Specifically, we show how the context of a task can facilitate task price estimation so crowdsourcing project managers can make an initial estimate of the costs to develop the software. Here we will examine task features from the requirements perspective on TopCoder. Then we will apply and evaluate nine predictive machine-learning algorithms on TopCoder to show how software requirements can be a practical estimation method for crowdsourcing. Our approach can help developers and platforms to have a better strategy when estimating software crowdsourcing tasks. Through this research, we aim to answer the following questions:

RQ1: How task context can be used to facilitate the pricing estimation for crowdsourced software development task?

RQ2: How machine learning techniques can be employed to effectively estimate tasks pricing? Can we employ Topic Modeling to estimate tasks pricing?

RQ3: How tasks pricing ranges can be mapped to different topics?

The rest of this paper is structured as follows: Section 2 summarizes the related literature. Section 3 presents an overview of the proposed Context-Centric Pricing approach. Section 4 introduces our study evaluation. In section 5, we present the reported results. Section 6 and 7 discusses the study results and some limitations. Finally, section 8 summarizes our conclusion and implications for future studies.

2 RELATED LITERATURE

2.1 Software Cost Estimation

Over the past two decades, researchers have explored artificial intelligence-based approaches to software estimation: Artificial Neural Network (ANN), Case-Based Reasoning (CBR), Rule Induction (RI), Classification and Regression Trees (CART), etc. [9]. Several studies have also applied machine-learning techniques on Global Software Development (GSD) projects [10], including a proposed analogy-based model for software estimation. Azzeh and Nassif [11] used the Fuzzy Model Tree to derive effort estimates from the Use Case Points (UCP) measure. The result was validated using the Treeboost model, multiple linear regression, and classical effort estimation. Several related studies [12-16] have applied machine-learning techniques and concluded that they provide adequate estimation models compared to traditional ones.

2.2 Pricing for General Crowdsourcing

In crowdsourcing, platforms need to provide effective pricing strategies that are adequately attractive to the crowd, to avoid task cancelation or task starvation.

Several studies have tried to address the problem of pricing tasks on crowdsourcing. For instance, Singer and Mittal [1] proposed constant-competitive incentive compatible mechanisms for maximizing the number of tasks under a budget, and minimizing payments, given a fixed number of tasks. The study created the Mechanical Perk (MPerk) platform, which uses various pricing mechanisms for efficient allocation of tasks. This study was based on the bidding model and the posted price model.

Difallah et al. [6] studied crowdsourcing pricing for worker retention and latency improvement on Amazon Mechanical Turk. Their study showed how increasing or decreasing the monetary reward over time influences the number of tasks a worker is willing to complete. The result was compared against a traditional pricing method for software crowdsourcing. Based on different bonus schemes to maximize retention —fixed, random, training, increasing, milestone-based, etc.—the study results show that various pricing schemes performed differently depending on the type of task involved. However, the best pricing scheme was based on milestone bonuses. The study concludes that new pricing schemes can be adopted to motivate workers to continue the crowdsourcing process and receive results faster.

Wang et al. [7] studied pricing on crowdsourcing platforms from a workers' quality perspective. The authors proposed methods to measure the quality of workers based on their expected contribution so an appropriate pricing scheme can be concept of reservation wage, the study assumes that the platform knows or can estimate the joint distribution of workers' quality. Most of these mentioned studies are applicable only on platforms such as Amazon Mechanical Turk, the micro-task crowdsourcing platform, which often deals with simple tasks. In our study, we apply our approach on the TopCoder platform, which allows for more diverse, complex software tasks.

2.3 Pricing for Software Crowdsourcing

Mao et al. [3] addressed the pricing problem on TopCoder by introducing 16 cost-drivers grouped in four different categories: (1) Development Type (DEV), (2) Quality of Input (QLY), (3) Input Complexity (CPX), and (4) Previous Phase Decision (PRE) (Table 1). The study concluded that high predictive quality is achievable by outperforming existing available pricing techniques and providing actionable insights for software engineers working on crowdsourced projects. This study, however, lacks some efficiency as some of the proposed variables are not practical in estimating software projects. More specifically, 6 out of the 16 cost drivers are depending on information from prior component design tasks, i.e. 4 in quality category (QLY), 1 in complexity category (CPX) and 1 in previous phase category (PRE), as shown in Table 1. In addition, the majority drivers from the CLX require more elaborated input to development tasks, such as sequence diagrams, UML design files, etc. For instance, winner rating, number of registrants, number of submissions and winner's effort are not always available prior the development phase.

These concerns motivate our study to develop a new pricing technique based on the limited information available in the received requirements. In software development, the main measure for the success of a product is to which degrees it meets the purpose for which it was aimed [25]. This also can be the case in software crowdsourcing as requirement tends to provide the online developers a complete, detailed, textual and/or graphical, description about the task to be designed or developed. This study reports how software requirements [8] can be more effectively used to estimate task pricing, which have been overlooked in existing studies.

3 OVERVIEW OF CONTEXT-CENTRIC PRICING

In the early phase, task requirement is the only source for the platform to make a decision. The CCP approach advocates for evoking some features from the task

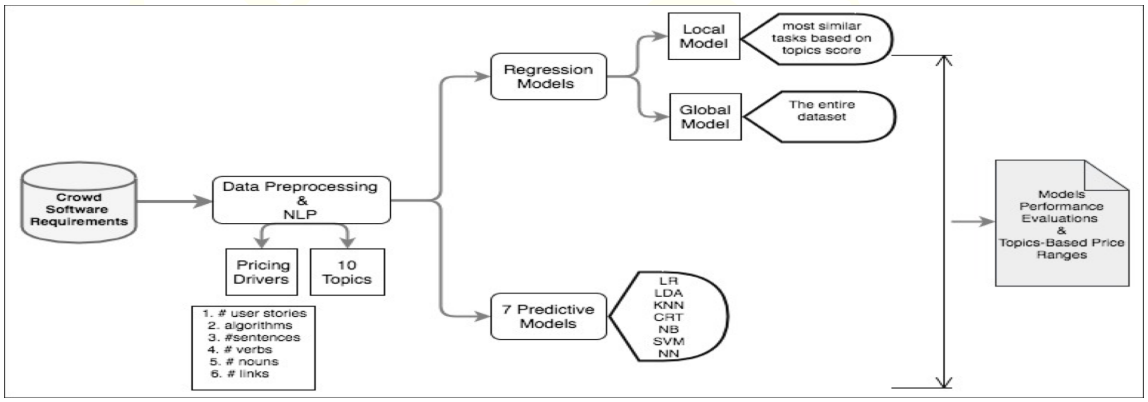


Figure 1. Context-centric pricing(CCP)

requirements. We believe that our approach can provide a systematic way to better estimate the task price from the available requirements. Figure 1 shows the overview of the CCP approach. It consists of step 1: the preprocessing and Natural Language Processing (NLP) step which allowed for preparing the data, extracting drivers, and 10 Topics (the most prevalent to each documents). Step 2 is proposing two regression based models, i.e. global model and local model, and step 3 building 7 machine learning based models as described in Table 7. Natural language processing techniques are employed to extract the following pricing features: number of sentences, verbs, nouns, links, and whether there is an algorithm included or not. User stories were extracted manually. The main objective is to produce accurate estimates of task prices as well as to reduce the need for expert knowledge in the task estimation process. We will introduce the details of CCP next.

3.1 Identifying Pricing Drivers from Crowdsourcing Task Requirements

The focus of this study is to identify and use requirements-based factors in task pricing that can always be obtained before starting any project. For that purpose, we first identify number of user stories and number of algorithms from crowdsourcing task description as two main pricing factors, which help to get into the details of task requirements. In the case of software crowdsourcing, user stories represent the measure of tasks in terms of the required functionality. To ensure consistency, we employed a manual process to count the number of user stories per each project. Two research assistants manually counted user stories separately to ensure accuracy. Algorithms can be described as a prerequisite input for some tasks provided by task requesters. It requires crowd workers to use these algorithms when developing their products. To extract whether there is an algorithm included within a project or not, we applied text mining technique using Python 2.7.12. We hypothesize that this type of feature might play a role in the estimated task price. Other requirement-centric features considered in CCP are the textual drivers such as the numbers of verbs, nouns, links and sentences. These features summarize the textual description per each task and weight each feature a potential impact on the overall estimation. Table 2 and Table 3 show summary of the description and statistics of these drivers which will be explained in more details later on.

3.2 Regression Models

Based on the drivers identified in Table 2, regression based pricing models can be established from historical data. Equation 1 at below shows the general format of the multiple linear regression models:

$$PRICE = \beta_0 + \beta_1 (User\ Story) + \beta_2 (Algorithm) + \beta_3 (Verbs) + \beta_4 (Nouns) + \beta_5 (Links) + \varepsilon \quad (1)$$

Furthermore, to better characterize other factors that may influence on pricing strategies in software crowdsourcing, such as task types, complexity, and urgency, we propose the local models which can be established with respect to a smaller set of historical data similar to the task to be predicted. In this study, we propose to apply topic modeling to construct such local context for establishing local models. Topic modeling technique was applied to extract the hidden semantic structure in each task. Topic models are "probabilistic models for uncovering the underlying semantic structure of a document collection based on a hierarchical Bayesian analysis of the original texts" [17]. In this study, we apply Topic Modeling to extract and represent the technical or functional context of the entire task description. MALLET [18] is applied to transform textual task description documents into numerical representation of task context that can then be processed effectively. MALLET handles distinct tasks such as tokenizing strings, removing stop-words, and converting texts into count vectors. The results of this process are sets of topics that show a document's most frequently used words. The process uses the "weight," a statistical measure that evaluates each word's significance to a specific document or collection of documents. The results show the probability of the occurrence of each topic within documents (Table 4). The distribution of topics within projects shows the probability of the

Table 2. Multiple Regression Model (Global Model)

Variable/ Drivers	Description	Regression Coefficients			Regression Coefficients		
		Step 1 Model (AIC =389.599)			Step 2 Model (AIC =387.600)		
		B	*T*	*P*	*B*	*T*	*P*
Intercept	Price per project	4.114	18.231	0.00	4.114 .	18.274	0.00
Algorithm	A binary outcome 0,1. If there is an algorithm included within the description or not	0.056	1.090	0.276	0.056	1.091	0.275
# Sentences	Sentences per each document	0.002	0.034	0.973	-	-	-
# Verbs	Verbs per each document	0.114	1.486	0.138	0.114	0.076	0.133
# Nouns	Nouns per each document	-0.145	-1.512	0.131	-0.143	-1.886	0.060
# Links	Links per each document	-0.045	-1.843	0.066	-0.045	-1.853	0.064
User stories	Number of user stories per project	1.912	19.735	0.00	1.912	19.759	0.00

Table 3. Summary of Raw Results

Variable	Descriptive Statistics					
	Min	*1st Q.*	*Median*	*Mean*	*3rd Q.*	*Max*
Price	112.50	750.00	750.00	773.30	750.00	3000.00
Sentences	140.00	281.00	386.50	455.10	560.00	2464.00
Nouns	340.00	725.80	1114.50	1261.10	1582.00	8299.00
Verbs	135.00	273.20	416.00	469.6	613.8	1969.00
Links	0.00	0.00	0.00	1.767	2.00	38.00
User Stories	2.00	4.00	4.00	4.18	5.00	9.00
Algorithm	-	-	-	-	-	-

occurrence of a given topic in a specific project, which helps to shape and establish the contextual information for each task. Finally, in order to construct the local model based on topic modeling analyses and the extracted pricing drivers, we select a subset of our dataset which tends to be the most similar regarding the topics scores. Pricing features including user stories, algorithm, verbs, nouns, and links were extracted for the same subset to build the local model.

Table 4. Samples of Topics Distributions Within Projects

Project	Top Topics	Score	Top Topics	Score	Top Topics	Score
1st	7	0.377	6	0.229	1	0.195
2nd	8	0.561	3	0.173	1	0.096
3rd	5	0.625	1	0.242	3	0.062
4th	9	0.776	5	0.051	7	0.526
5th	10	0.347	6	0.343	1	0.134
6th	8	0.694	4	0.086	6	0.067
7th	6	0.703	9	0.119	4	0.091
8th	9	0.405	1	0.311	3	0.12
9th	6	0.615	7	0.178	3	0.024
10th	9	0.513	6	0.21	1	0.149

3.3 Local Pricing Context

To effectively validate the conclusion of our results, we followed the ***Local Vs. Global Contexts*** introduced by Menzies et al. [26] in the context of empirical Software Engineering. Local Vs. Global context emphasizes the local regions of the data with similar properties where the lessons learned across a population may be the same to the lessons learned from individuals within the population.
So, in the generality of conclusions, it is recommended to test the validation of general conclusion within subsets of the data.
To find the best local conclusion within our data, we applied the following:

1. After applying the topic modeling technique, a subset of the collected data found to be similar in their properties. The probability of each Topic in each project determined the closest projects in terms of their Topics score distribution.
2. We applied cosine similarity matrix on the topic modeling scores to group the most similar software projects and extracted the following properties for each project: user story, algorithm, number of sentences, number of verbs, number of nouns and number of links.
3. Multiple-regression model, eq.2, was introduced to allow a general conclusion of the reported results and identify how variance can change from modeling the whole data, equation 1, VS. subsets with similar properties, equation 2.

$$PRICE = \beta_0 + \beta_1 (User\ Story) + \beta_2 (Algorithm) + \beta_3 (Nouns) + \varepsilon \quad (2)$$

3.4 Learning-based models

To provide more pricing options, we integrate seven machine-learning algorithms to predict software crowdsourcing pricing. These include six classification algorithms—Logistic Regression, Naïve Bayes, Classification and Regression Trees, K-Nearest Neighbors, Support Vector machines, Linear Discriminant Analysis—and one Neural Network Learner.

4 ANALYSIS

This section will present the evaluation of the proposed CCP approach.

4.1 Dataset

The data used in this study was collected by Mao et al. [3]. It consists of 450 crowdsourcing software development tasks with complete textual descriptions. Tasks vary in their types including but not limited to (security, database, web, financial, data management, analysis) and sizes (lines of Codes) between 310 LOC to 21925 LOC. Task description consists of overview of the problem, the required technology, links for external resources. Tasks requirements range between 2 and 23 pages. Table 4 shows basic descriptive statistics for the extracted pricing drivers.

4.2 Methodology

4.2.1 Data Preprocessing

We used NLTK[4], a platform for building the Python program to work with human language to remove from each document both stop-words and frequently used words we deemed inefficient for topic modeling applications, see Figure 1 (Data Preprocessing and NLP). Documents with textual description usually contain stop-words that are irrelevant to our context, e.g., prepositions, conjunctions, articles. We used a MALLET list to remove stop words so we knew which misleading words to remove from the extracted topics. We followed an iterative process to remove some frequently used words that would mislead our models by giving more weight to

[4] https://www.nltk.org

these specific keywords. Some of these keywords appeared too often in project descriptions, including: *summary, overview, design, tag, class, PDF, row, column, table, package, file, installation, thread name.*

4.2.2 Regression Modeling

We employ stepwise regression procedure on log-transformed data, which eliminates non-significant factors. Each step reduces the Akaike Information Criterion (AIC). The Step 1 model includes algorithm (*A*), number of verbs (*V*), number of nouns (*N*), number of links (*L*), and number of userstories (*US*).
As Table 1 shows, Step 2 model eliminates the number of sentences (*S*) predictor, which contributes little variance.

4.2.3 Topic Modeling in MALLET

Topics are collections of word tokens (see Fig.2) that show the contents of what topics represent. MALLET identifies the most relevant topic for each document. Table 4 shows a sample of topic distributions among 10 projects and their associated probability. For instance, the first project has topics 7, 6, 1 as the most relevant topics to its content; these keywords are the most frequent in the corpus. Some projects, however, have more topics associated with them than other projects do. In the data-cleaning process, projects missing one topic or more receive a value of zero for that topic. For instance, after applying topic modeling on the whole dataset, Project 2 in Table 4 shows that it has three topics (8, 3, 1), each of which has some probability scores associated with it greater than zero. The other topics for this project (2, 4, 5, 6, 7, 9, 10) equal zero.

Figure 2 illustrates the a visualized density of 10 topics and their occurrence within software projects. The density is between 0 and 1. The *x*-axis shows that the project ID is between 1 and 450. The first 5 topics in the first 5 rows are spread throughout the projects. Topic 6 in row 6 is concentrated in the first 150 projects and spread evenly throughut the last 100 projects.

4.2.4 Machine Learning Classifiers

The second part of our study aims to investigate how machine learning techniques help in estimating software crowdsourcing price. To do so, we first converted the target variable "Task Price" into discrete/categorical values so it can be treated as a classification problems.
Then we applied six classifiers on the topic modeling results including Logistic Regression, Naïve Bayes, Classification and Regression Trees, K-Nearest Neighbors, Support Vector machines, Linear Discriminant Analysis—and one Neural Network Learner. This allows us to compare our results to previous work done by Mao et al. [3]. Python 2.7.12 was used in for the analysis of this study with the default parameter except the models' evaluation which we introduce in the next section. Figure 1 shows that after applying NLP, we derived 10 topics which are more prevalent in each tasks document. These topics used later to train the model in order to predict software price.

4.2.5 Models Performance Measure Evaluation

To quantify and assess the quality of the applied machine-learning algorithms, we have used several model evaluation measures, most of which have been extensively used in the literature. Cross-validation, applied to assess the accuracy of the proposed models, is a common practice when preformatting supervised machine-learning algorithms to hold out part of the available data [22]. In our study, the data are divided into 10 parts; each single part is used as a test while the others are used to train the model. This process is repeated for all combinations of train-test splits. This type of assessment was used in previous studies in which the goal was to predict future events [20, 21].

To evaluate model performance, we use an accuracy classification score (see Equation 3)—the ratio of the number of correctly predicted instances divided by the total number of instances in the dataset, all multiplied by 100 to render the ratio as a percentage. Remember that the target variable (task price) was converted to discrete values to allow classification. Finally, two measures of *recall* and *precision are also employed*. *Precision* (see Equation 4) is "the number of true positives divided by the total number of elements labeled as belonging to the class" [19].

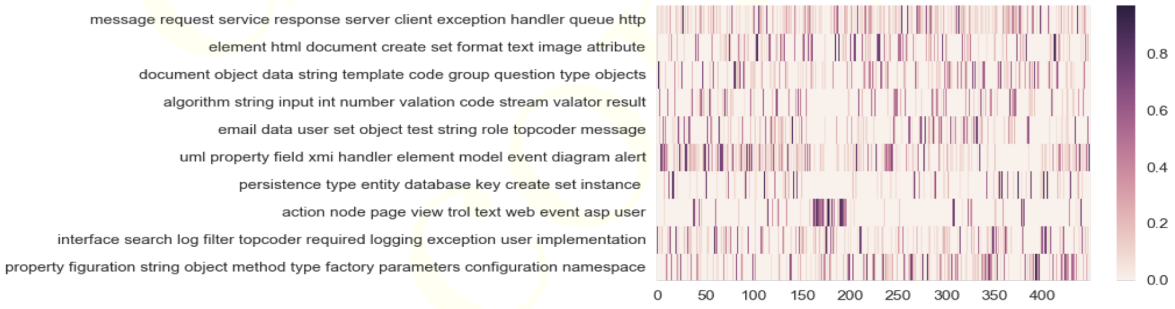


Figure 2. 10 Topics and their density among projects

Recall (see Equation 5) is "the number of true positives divided by the total number of elements that actually belong to the class" [20].

$$Accuracy = \frac{tp+tn}{tp+tn+fp+fn} \quad (3)$$

$$Percision = \frac{tp}{(tp+fp)} \quad (4)$$

$$Recall = \frac{tp}{(tp+fn)} \quad (5)$$

Precision and recall are combined into a single measure called F-Measure—the weighted average of the precision and recall. It reaches its best at 1 and its worst at 0. Equation 6 shows how F-Measure can be computed:

$$F1 = 2 * \frac{(precision*recall)}{(precision+recall)} \quad (6)$$

5 RESULTS

This section presents the obtained results of multiple linear regression (Eq. 1 and Eq. 2) and the seven-trained machine-learning algorithms. Multiple linear regression was used to develop a model for predicting crowdsourcing tasks pricing based on five log-transformed predictors extracted from tasks' requirements (Table 2). Topic modeling analysis produced 10 topics, which can be fed into different machine learning classifiers for predicting price (Fig. 3).

A. *How task context can be used to facilitate the pricing estimation for crowdsourced software development task (RQ1)?*

Table 2 shows a stepwise regression procedure on log-transformed data, which eliminates non-significant factors. Each step reduces the Akaike Information Criterion (AIC). The Step 1 model includes algorithm (*A*), number of verbs (*V*), number of nouns (*N*), number of links (*L*), and number of user stories (*US*). As Table 1 shows, Step 2 model eliminates the number of sentences (*S*) predictor, which contributes little variance.

In Equation 1, where price (*P*) was predicted by algorithms (*A*), verbs (*V*), nouns (*N*), links (*L*), and user stories (*US*), results show that the effect of user stories (*US*) (*t*-statistics = 20.80, *p*-values = 0.00), the number of links (*L*) (*t*-statistics = -1.876, *p*-values > 0.05), and the number of nouns (N) (*t*-statistics = -1.871, *p*-values > 0.05) are statically significant. However, two predictors—algorithms (*A*) and number of verbs (*V*) turned out to be less significant, with *t*-statistics = 1.091 and *p*-values = 0.275 for (*A*), and *t*-statistics = 1.503 and *p*-values = 0.133 for (*V*). R-squared for the proposed model equals to 50%.

The regression results in Equation 1 reveal that the coefficient $\beta_1 = 0.056$, which signifies whether or not algorithm is included in the task description, is increased where the intercept is $\beta_0 = 4.114$ (pricing). This result indicates that algorithm factor contributes to increased pricing. In other words, it might reveal more complexity or some constraints within the task. The coefficient $\beta_2 = 0.114$, which represents the number of verbs, is increased where the intercept is $\beta_0 = 4.114$ (pricing). This also shows that, when the number of verbs increased, the price also increased. $\beta_3 = -0.143$, which represents the number of nouns that appeared in the task description, is decreased where the intercept is $\beta_0 = 4.114$ (pricing). This is the opposite of the number of verbs effect, in which the number of nouns decreased when the price increased. Finally, the effect of user stories on pricing was statistically significant.

The coefficient $\beta_4 = 1.912$ shows that tasks with higher numbers of user stories contribute to increasing task pricing. This finding agrees with our assumption that more user stories mean higher pricing. A more thorough research of crowdsourcing pricing can provide an array of clues to include the user stories in future pricing strategies.

The regression results in Table 5 investigates the difference between local and global treatment. The local treatment employed by the second regression model is eq2. expects to reveal low variance as suggested by prior research [24]. In this model, the price (P) was predicted by algorithms (*A*), nouns (*N*), and user stories (*US*), results show that the effect of user stories (*t*-statistics = -1,506, *p*-values = 0.00), number of nouns (*t*-statistics = 1,256, *p*-values = 0.144), and algorithm (A) (*t*-statistics = 9.038 , *p*-values = 0.22) are statistically significant. The R-squared for the proposed model(local model) equals to, $R^2 = \%79$ which was significantly increased compared to the global model.

Table 5. Multiple Regression Model (Local Model)

Variable	Regression Coefficients $R^2 = \%79$		
	B	*T*	*P*
Intercept	-784.43	-3.085	0.004
Algorithm	76.55	9.038	0.22
# Nouns	-86.16	1.256	0.144
User stories	2762.33	-1.506	0.0

B. *How machine learning techniques can be employed to effectively estimate tasks pricing (RQ2)?*

Tables 6 and 7 summarize the findings of our empirical study on topic modeling data. The applied machine learning techniques are treated as a classification problem where, based on the proposed cost-drivers, the output depended upon whether or not the model estimated the right price. For this analysis, we used 80% of the dataset for training and 20% for testing, and we reported the results on 10-fold cross validation. (Recall that we have validated our results using 450 complete projects from TopCoder, the largest software crowdsourcing platform.) We analyzed the results from the perspectives of accuracy measure, precision, recall, and F-Measure. Table 6 presents the pricing models performance in terms of their accuracy. SVM scored 65%, the highest mean accuracy, or ratio of the number of correctly predicted instances divided by the total number of instances.

This finding shows that the Support Vector Machine (SVM) substantially outperformed the other models. This finding confirmed previous studies' conclusions that SVM is an outstanding classifier in text classifications [22-24]. Logistic Regression (LR) and KNN scored %63 and %61, respectively. These results confirm the

improvement this study made on Mao et al. [3]. As mentioned earlier, the authors used some factors we consider impractical in software cost estimation, i.e. "number of registrants for a task, number of submissions for a task and score of winners."

Table 6. Performance Accuracy of The Pricing Models

Algorithm	Accuracy (Mean)	Accuracy (STD)
Logistic Regression	0.63	0.65
Linear Discriminant Analysis	0.55	0.10
K-Nearest Neighbors	0.61	0.06
Classification and Regression Trees	0.55	0.08
Gaussian Naive Bayes	0.16	0.06
Support Vector Machines	0.65	0.06
Neural Network	0.35	0.16

Linear Discriminant Analysis (LDA) and Classification and Regression Trees (CART) performed %55 in the accuracy measure, which suggest LDA has some linearity. Naïve Bayes (NB) and Neural Network (NN) produced the worst results, which confirms previous research findings that Naïve Bayes is very sensitive to our topic modeling dataset.

The boxplot in Figure 3 provides a better insight into the classification accuracy of the six classifiers plus one Neural Network (NN) model. This boxplot clearly shows that Naïve Bayes (NB) produced the worst estimates, with significant differences from those produced by Support Vector Machines (SVM), which appeared to be the best estimate model. Neural Network (NN) shows high variance between ~0.27 and –0.73. In contrast, Naïve Bayes (NB) shows low variance with less accuracy.

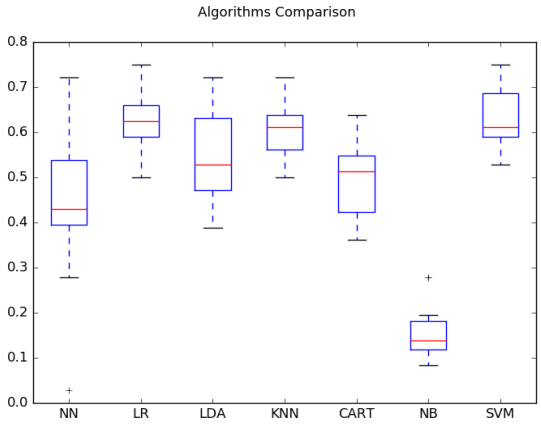


Figure 3. Machine learning model accuracy comparison

Table 7 shows the F-measure of each proposed classifier. As in previous studies, the F-measure was used as a confidence level in the prediction of new instances. In Table 6, the success of Linear Discriminant Analysis regarding its high F1-measure (63%) may reveal that our data is well preprocessed; that is, it has no outliers with low correlations between predictor variables. However, Naïve Bayes and Neural Network still report low performance compared to the other models. By looking at the results reported by the accuracy measure and F1-measure, we can see a discrepancy between them, possibly due to the sample size and, more importantly, our use of 10-way cross validation, which usually overestimates errors in models [20]. In other words, the real performance of the models in F1-measure exceeds the values reported in Table 6. This is consistent with previous studies applying machine-learning algorithms on text data [19].

C. How Tasks Pricing Ranges Can be Mapped to Different Topics(RQ3)?

To answer (RQ3), we applied Kernel Density Estimation (KNE) on the tasks pricing, which revealed three groups. Table 8 shows that the first group, which has the range between $112.5 and $525, contributes by 9.52% to the whole data. The second group, which represents the majority of the pricing range in our dataset between $600 and $1,200, contributes by 83.80%. The third group, ranging between $1,275 and $3,000, contributes by 6.66%.

Table 7. Comparison of Confusion Matrix Results

Algorithm	Precision	Recall	F1 Score
Linear Discriminant Analysis	0.64	0.64	0.63
Logistic Regression	0.48	0.67	0.54
K-Nearest Neighbors	0.42	0.60	0.50
Classification and Regression Trees	0.59	0.56	0.57
Gaussian Naive Bayes	0.67	0.21	0.27
Support Vector Machines	0.43	0.66	0.52
Neural Network	0.40	0.39	0.39

Figure 4 gives more insights into the relationship between pricing ranges and the 10 extracted topics. Topic 7, *"Data persistence entity database key set instance entity,"* and Topic 9, "*Interface search log filter exception user implementation,*" represent the topics belonging to the first group. Topic 1, *"Message request service response server client exception handler queue,*" and Topic 6, *"Field xmi handler element model event diagram alert,"* tend to be in the second group, with the mean of $781. Finally, Topic 2, "*Element html data set format text image attribute,*" and Topic 8, "*Action node view troll text web event ASP user*" turned out to be within the last group. This can be explained by a recent study showing that HTML-related projects are one of the most demanding on TopCoder [25]. Our findings can provide crowdsourcing platforms with a better understanding of the current types of projects and their associated prices. Future research can build upon the reported results for more accurate pricing schemes.

Table 8. Pricing Groups and Assigned Topics

	First	Second	Third
Max	$525	$1200	$3000
Min	$112.5	$600	$1275
Mean	$408.75	$781	$1878
STD	130.40	109	536

Count	40(9.52%)	352(83.80%)	28(6.66%)
(Assigned Topics)	Topic7 Topic9	Topic1 Topic6	Topic2 Topic8

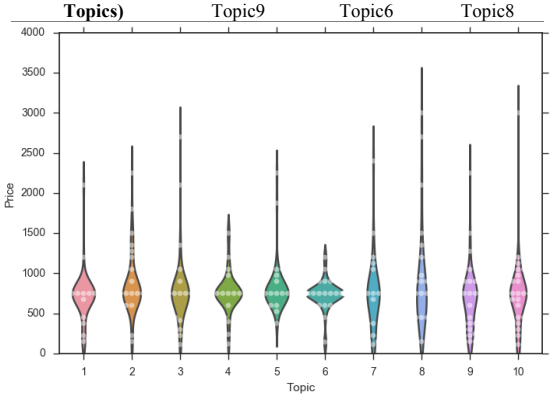


Figure 4. Pricing Ranges among 10 Topics

6 DISCUSSION

Our results show that requirements can be used to estimate software crowdsourcing project prices. Throughout our study, we showed the process of using natural language processing techniques on software requirements. We did so throughout introducing Context-Centric Pricing (CCP) approach by studying how different models can be trained on such datasets. We began by preprocessing the texts to achieve sufficient topics, which helped us to achieve some reasonable accuracy. Future work should use large datasets with more variance in price so we can compare our current study to future findings.

Our results show that machine-learning methods on natural language processing analyses are useful estimation tools. Our analysis provides more insights into some of the proposed drivers that can assist decision-makers in pricing crowdsourcing projects. For instance, in Table 2, we were able to estimate the pricing through multiple regression model. Predictors were statistically significant. Our results reveal that the local model exhibits significant results superior to the global model allowing for more validation within subsets of the data.

The topics derived from this study were mapped to the estimated prices as shown in Table 8. The first group, which includes Topic 7 and Topic 9, shows that most of the tasks in this group are related to Data Management. This can be clearly seen by going back to the dataset and investigating projects context. By looking at projects description in this group, we found that most of these projects are related to filtering or control systems which require some of the keywords in Topic 7 and Topic 9. In addition, most of these projects do not require complicated features. We quote from one project:

"This interface provides methods for writing the data to some output in fixed format. Its methods allow to write instances of basic types".

The second group in Table 8 shows that most of its projects belong to Topic 1 and Topic 6. By looking at projects description, we found out that these projects are regarding developing communication systems. For instance, the following quotes were extracted from one project:

"Email is a central part of many applications. The Email WCF Service will provide an SOA friendly manner for sending and receiving emails. It will leverage the Email Client component to actually send the emails, and will also provide its own implementation of the Email Client for other users to integrate into their own applications".

"All exception handling in this component is done using the SelfDocumentingException. Configuration is done via Configuration API and Object Factory Configuration API Plugin components".

Finally, the third group which belongs to Topic 2 and Topic 8 turned out to be the most sophisticated and complex projects. Most of these projects require advanced Graphical User Interface" GUI" with more features as it appeared in previous study [25]. The following quote was extracted from one project:

"The diagram should have a text input tool that will be displayed as a popup. The text field's text will be configurable through the API. It will disappear if Enter or Escape keys are pressed (setting or ignoring the input), or when the mouse is pressed outside the text input (in which case the input is accepted)".

These findings can reveal more insights into the platform in which projects with specific keywords have higher pricing. Furthermore, our topics from topic modeling analysis can generate meaningful labels: user interface design, email server, web development, etc. (Fig. 3) This is important when applying Topic Modeling to some domain knowledge, e.g., software engineering.

To link the current results with previous study [3], we can make note of the following:

1. In this study, we employed 10-way cross validation to evaluate our models, dividing the data into 10 equally sized subsamples. Each subsample is retrained as the validation data for testing the model; the remaining data are used as a training dataset. Previous studies have used or recommended this approach [20]. In contrast, the previous study related to ours used Leave-One-Out Cross-Validation.

2. The previous study [3] included some drivers mentioned earlier which we deem impractical for estimating software pricing. However, we based our drivers on software requirements that can always be obtained.

3. The previous study used four performance measures, including Pred (30), to evaluate the classifiers. The results show that C4.5 is the best performer, where 84% of the estimates have an error lower than 30%. In our study, 3 out of 7 models had a relatively good performance greater than 60% accuracy—a promising result, given the limitations mentioned. Future work will compare classification results against random guessing as a baseline comparison.

7 THREATS TO VALIDITY

Several limitations should be considered while interpreting and reusing the reported results. First, our Topic Modeling Parameter *K*—the amount of topics that can be produced—was set to be 10, based on the "Rule of Thumps". This result can change in the case of a different parameter, say, K = 20. Second, our analysis was based on a sample of 450 projects with small price ranges. Larger numbers of projects with broader pricing ranges could produce different results in future studies. Third, our study focuses solely on the TopCoder platform, so results could differ in other crowdsourcing platforms.

Of the above threats, some can be resolved in future studies with larger amounts of projects and cost-drivers. However, other threats relate to the nature of the topic-modeling process.

8 CONCLUSION AND FUTURE WORK

Over the years, researchers have studied and built different models to accurately estimate software effort. Yet more effort to investigate their implications on real software projects is needed. This study shed light on a subject rarely pursued: the application of machine-learning techniques such as Topic Modeling on software requirements for price estimation.

We empirically studied this phenomenon on software crowdsourcing using real projects, and we proposed different features, all based on task descriptions. From these pricing features we derived seven machine-learning algorithms, most described as classifiers. We performed 10-way cross validation to evaluate the proposed models. Our results show evidence that analyzing task descriptions can assist in estimating software prices. We also showed evidence that this study overcame some limitations the previous study had encountered.

Future research will focus more on domain knowledge type, particularly the categorization of a large number of projects according to their domain knowledge to attain more insights into variation of crowdsourcing platform performance based on different domains. One direction of future work can be to study different crowdsourcing platforms and compare the results of different models.

REFERENCES

[1] Singer, Yaron, and Manas Mittal. "Pricing Tasks in Online Labor Markets." Human computation. 2011.

[2] Singer, Yaron. "Budget feasible mechanisms." Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on. IEEE, 2010.

[3] Ke Mao, Ye Yang, Mingshu Li, Mark Harman. "Pricing Crowdsourcing-based Software Development Tasks*". In Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. NIER Track, pp. 1221-1224, 2013.

[4] Howe, Jeff. "The rise of crowdsourcing." *Wired magazine* 14.6 (2006):1-4

[5] Gonen, Rica, et al. "Increased efficiency through pricing in online labor markets." *Journal of Electronic Commerce Research* 15.1 (2014): 58.

[6] Difallah, Djellel Eddine, et al. "Scaling-up the crowd: Micro-task pricing schemes for worker retention and latency improvement." *Second AAAI Conference on Human Computation and Crowdsourcing*. 2014.

[7] Wang, Jing, Panagiotis G. Ipeirotis, and Foster Provost. "Quality-based pricing for crowdsourced workers." (2013).

[8] Harwell, Richard, et al. "What is a Requirement?" *INCOSE International Symposium*. Vol. 3. No. 1. 1993.

[9] Humayun, Mamoona, and Cui Gang. "Estimating effort in global software development projects using machine learning techniques." *International Journal of Information and Education Technology* 2.3 (2012): 208.

[10] El Bajta, Manal. "Analogy-based software development effort estimation in global software development." *2015 IEEE 10th International Conference on Global Software Engineering Workshops*. IEEE, 2015

[11] Azzeh, Mohammad, and Ali Bou Nassif. "Fuzzy Model Tree For Early Effort Estimation." *Machine Learning and Applications (ICMLA), 2013 12th International Conference on*. Vol. 2. IEEE, 2013.

[12] B. Baskeles, B. Turhan, and A. Bener, "Software effort estimation using machine learning methods," In Proc. of 22nd international symposium on computer and information sciences, 2007.

[13] A. Heiat, "Comparison of artificial neural network and regression models for estimating software development effort," Information and Software Technology,pp 911–922, 2004.

[14] E. Jun. and J. Lee, "Quasi-optimal case-selective neural network model for software effort estimation," Expert Systems with Applications, pp. 1–14,2001.

[15] A. Oliveira, "Estimation of software project effort with support vector regression," Neurocomputing, pp.1749–1753, 2006.

[16] H. Park. and S. Baek, "An empirical validation of a neural network model for software effort estimation," Expert Systems with Applications vol 35 no 3 pp 929–937 2008

[17] Srivastava, Ashok N., and Mehran Sahami, eds. *Text mining: Classification, clustering, and applications*. CRC Press, 2009.

[18] McCallum, Andrew Kachites. "MALLET: A Machine Learning for Language Toolkit." http://mallet.cs.umass.edu. 2002.

[19] Menzies, Tim, and Andrian Marcus. "Automated severity assessment of software defect reports." *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*. IEEE, 2008.

[20] Quinlan, J. Ross. C4. 5: programs for machine learning. Elsevier, 2014.

[21] Pedregosa, Fabian, et al. "Scikit-learn: Machine learning in Python." *Journal of Machine Learning Research* 12.Oct (2011): 2825-2830.

[22] Forman, George. "An extensive empirical study of feature selection metrics for text classification." *Journal of machine learning research* 3.Mar (2003): 1289-1305.

[23] Yang, Yiming, and Xin Liu. "A re-examination of text categorization methods" *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 1999.

[24] Joachims, Thorsten. "Text categorization with support vector machines:Learning with many relevant features." *European conference on machine learning*. Springer Berlin Heidelberg, 1998.

[25] Alelyani, Turki, and Ye Yang. "Software crowdsourcing reliability: an empirical study on developers behavior." *Proceedings of the 2nd*International Workshop on Software Analytics. ACM, 2016.

[26] Menzies et al. "Local vs. global models for effort estimation and defect prediction." *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*. IEEE, 2011.

The Characteristics of False-Negatives in File-level Fault Prediction

Harold Valdivia-Garcia
Bloomberg LP
hvaldiviagar@bloomberg.net

Meiyappan Nagappan
University of Waterloo Waterloo, Ontario Canada
mei.nagappan@uwaterloo.ca

ABSTRACT

Over the years, a plethora of works has proposed more and more sophisticated machine learning techniques to improve fault prediction models. However, past studies using product metrics from closed-source projects, found a ceiling effect in the performance of fault prediction models. On the other hand, other studies have shown that process metrics are significantly better than product metrics for fault prediction. In our case study therefore we build models that include both product and process metrics taken together. We find that the ceiling effect found in prior studies exists even when we consider process metrics. We then qualitatively investigate the bug reports, source code files, and commit information for the bugs in the files that are false-negative in our fault prediction models trained using product and process metrics. Surprisingly, our qualitative analysis shows that bugs related to false-negative files and true-positive files are similar in terms of root causes, impact and affected components, and consequently such similarities might be exploited to enhance fault prediction models.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

KEYWORDS

Process Metrics, Code Metrics. Post-release Defects

ACM Reference format:
Harold Valdivia-Garcia and Meiyappan Nagappan. 2017. The Characteristics of False-Negatives in File-level Fault Prediction . In *Proceedings of PROMISE , Toronto, Canada, November 8, 2017,* 10 pages.
https://doi.org/10.1145/3127005.3127013

1 INTRODUCTION

In modern society, software has become ubiquitous in most aspects of our working and social life. We use software systems in applications ranging from entertainment (e.g., games, multimedia), to finance (e.g., e-commerce, internet-banking) as well as mission critical systems (e.g., aircraft navigation). Simultaneously, the ever growing demand for high quality and new features in shortened intervals is skyrocketing the software production and maintenance cost [12].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PROMISE , November 8, 2017, Toronto, Canada
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5305-2/17/11…$15.00
https://doi.org/10.1145/3127005.3127013

In order to cope with this pressure, the researchers and practitioners have put a large amount of effort in developing fault prediction techniques that predict the location of fault prone software entities (e.g., subsystems, files, function, etc) in the next release of a software system [3, 23, 35, 49]. Such models can be used to guide practitioners to produce dependable software systems with fewer faults, on schedule and within the budget.

For the past decade, there has been a plethora of publications on this subject. For example, in 2009, Catal and Diri [7] identified 74 studies related to fault prediction. Later, a systematic literature review of fault prediction performance by Hall et al. [19] identified 208 studies between 2000 and 2010. As a research community, we have explored a large number of approaches in order to improve the performance of fault prediction models. We investigated various **machine learning techniques** ranging from simple models (e.g., linear regression [21, 47, 64]) to sophisticated and complex models (e.g., random forest [18, 25, 38]). We also used a wide variety of **software metrics** derived from source code (e.g., LOC, Cyclomatic, Call dependency metrics), code history (e.g., Churn, pre-release bugs, developer's contribution).

However, past research by Menzies et al. [42] and Lessmann et al. [14, 32] using product metrics (static metrics) extracted from closed-source projects, found a ceiling effect in the performance of fault prediction models. On the other hand, other studies have shown that process metrics are significantly better than product metrics for fault prediction models [45, 50]. Therefore, to extend the scope of prior work and to better understand the limitations of fault prediction, we investigate the extent of the ceiling effect in prediction models trained on both product and process metrics from eleven open-source projects. More precisely, we analyze the outcomes of six well-known machine learning techniques trained on eight software metrics that have being reported to be good predictors of buggy files. In a further analysis, and to shed light on the ceiling effect of fault prediction models, we investigate what are those buggy files that prediction models incorrectly predict as non-buggy (i.e., false negative)? The inability of identifying a large number of buggy files might limit the adoption of fault prediction models. Therefore, we quantitatively and qualitatively analyze what aspects of buggy files (false negative and true positive files) make them different/similar.

There are certainly many papers that report the number or percentage of such incorrectly predicted files. But nobody has looked at them to better understand what characteristics make them unable to be correctly predicted by the fault prediction models. **Terminology.** There are a few terms that will be used throughout the remainder of this paper. Thus, before we proceed any further, let us briefly define these terms. When predicting buggy and non-buggy files, there are four possible outcomes. If a buggy file is correctly

predicted as buggy, we refer to it as ***True-Positive file* (*TP-file*)**; however if it is incorrectly predicted as non-buggy, we refer to it as ***False-Negative file* (*FN-file*)**. If a non-buggy file is correctly predicted as non-buggy, we call it ***True-Negative file* (*TN-file*)**, otherwise we call it ***False-Positive file* (*FP-file*)**. In our study, we also quantitatively and qualitatively analyze the post-release bugs related to buggy files. We split such bugs into two different groups. Bugs that affect only *TP-files* are referred to as ***True-Positive bugs* (*TP-bugs*)**, whereas bugs that affect only *FN-files* are referred to as ***False-Negative bugs* (*FN-bugs*)**. In this paper, we make the following contributions:

- By analyzing the outcomes of six fault prediction models, we found that approximately 79% of buggy files in our case studies are incorrectly predicted as non-buggy files. In other words, the True-Positive Rate of the our prediction models has an upper bound of about 21%.
- We compare the distributions of *FN-files*, *TP-files* and *TN-files* across 8 software metrics. **First**, we found that even though both *FN-files* and *TP-files* are both buggy files, they are statistically significantly different in each of the 8 metrics. **Second**, we found that *FN-files* are closer to *TN-files* than to *TP-files*.
- We quantitatively and qualitatively investigate the bug reports, source code and commit information for the post-release bugs related to buggy files incorrectly and correctly predicted (*FN-bugs* and *TP-bugs*).
- For the **quantitative analysis**, we compare *FN-bugs* and *TP-bugs* across metrics collected from bug tracking systems. Our results show that at bug report level both groups of bugs are similar.
- For the **qualitative analysis**, we manually inspect subsets of *FN-bugs* and *TP-bugs*. We find the most common root causes for these two groups are: missing cases (25%-23%), control flow issues (9%-10%) and wrong functional implementations (40%-42%). Similarly, in terms of impact, we find that crashes (44%-45%) and incorrect functionalities (40%-42%) are the most frequently occurring ones.

Our study confirms previous findings about the ceiling effect in the performance of fault prediction models even when we include process metrics. Surprisingly, our qualitative analysis shows that bugs related to *FN-files* and *TP-files* are similar in terms of root causes, impact and affected components, and consequently such similarities might be exploited to enhance fault prediction models.

The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 presents our case study design, including the data collection at bug and file level. Section 4 discusses the findings of our case study. Section 5 highlights the threats to validity and Section 6 concludes the paper and discusses future work.

2 RELATED WORK

Software Metrics: Past research has used a wide variety of software metrics in order to improve the performance of fault prediction models. Many studies have extensively investigated the usefulness of product metrics such: LOC, Cyclomatic, static analysis defect density, object oriented metrics, etc [6, 10, 40, 46, 48, 67]. Other studies have reported that process metrics such as Churn, number of pre-release faults, number of prior modifications, co-changed files, developer-related metrics, etc have good explanative and predictive power when predicting faults [17, 43, 45, 47, 55]. Beside product and process metrics, there are also studies that used more complex metrics such as code dependency graph, change genealogy, socio-technical network metrics, etc [3, 22, 65]. In our paper, we build our fault prediction models using two code metrics and six process metric that have been reported to be good indicators of fault-proneness [17, 40, 43, 45, 50]. We did not consider network based metrics because, first, there has been considerable disagreement about their effectiveness [49, 65] and second, they are more expensive to extract than code and process metrics.

Fault Prediction: Prior studies on fault prediction have investigated different approaches (statistical techniques and machine learning techniques), settings (forward-release [49], cross-project [66]) and granularity levels (commit [26, 28, 54], method[15, 29], file [16, 39, 55], subsystem [3, 20, 41], etc). A wide range of techniques from simple (e.g., Logistic Regression, Naive-Bayes, decision tree, etc [11, 21, 45]) to more sophisticated (e.g., Random Forest, ensemble methods, multivariate adaptive regression splines (MARS), etc [2, 18, 33, 62, 63]) have been used in order to improve the predictive power of fault prediction.

Many comprehensive studies have compared the predictive power of different prediction techniques and found conflicting results. Lessmann et al. [32] proposed a comparative framework to evaluated 20 different prediction techniques on the NASA dataset. Among the top 17 top techniques, the authors found no significant differences in their performance. Similarly, Arisholm et al. [1] performed a systematic investigation of fault prediction models using different techniques, metrics and evaluation criteria. Their results suggest that the choice of prediction technique has limited impact in terms of accuracy and cost-effectiveness. Menzies et al. [42] studied the limited improvement (*ceiling effect*) in fault prediction offered by product (static) metrics. The authors performed different sampling techniques to improve the prediction performance. They suggest that improvement may not come from new machine learning techniques, but from considering the software business knowledge through case-based reasoning tools. In a subsequent work Menzies et al. [41] proposed a meta-learner called WHICH that can include different business goals. Their results shows that the ceiling effect may not hold when using domain specific goals instead of traditional performance metrics.

Most recently, Ghotra et al. [14] replicated and extended the analysis conducted by Lessmann et al. The authors used a cleaned version of the NASA dataset for their evaluation. Their results showed that different prediction techniques are statistically distinct than others and therefore the choice of prediction techniques matters. Other studies showed that automated parameter optimization significantly improve the performance of fault prediction models [13, 58]. Tantithamthavorn et al. [58] used the R package Caret[31] to tune the parameters of 26 machine learning techniques. In their study, the authors were able to improve the AUC performance of the prediction models by up to 40%. Within a similar vein of research, Fu et al. [13] investigated the usefulness of a parameter optimization technique named *Differential Evolution* [56]. They reported an improvement in the precision of 5%-20%. In our work, we also use several prediction techniques in a forward-release setting

to predict buggy files and perform parameter tuning to complement our findings. The way we differ from prior mentioned studies is in the fact that we are not advocating for one technique/metric over another. We are also not proposing a new technique to improve the performance of fault prediction. Instead, we examine if a ceiling effect exists when using product and process metrics, and investigate the nature (characteristics) of *FN-files* and *TP-files* from six prediction techniques

3 CASE STUDY DESIGN

3.1 Studied Projects

For our analysis of false negatives in fault prediction, we collect data from 11 open-source projects namely: Accumulo, Bookkeeper, Camel, Cassandra, CXF, Derby, Felix, Hive, OpenJPA, Pig and Wicket. Theses projects are written in Java and cover a wide range of domains (i.e., SQL and NoSQL databases, Enterprise, Service and Web frameworks, Data Analytics platforms, etc). We selected these projects because they are long-lived and large projects that have been used in prior works [9, 52, 53]. We leveraged code, process and bug metrics from the bug tracking system and source-code repository of each project. In total, we extracted 22,803 bug reports and 98,297 commits. We selected these projects because they are long-lived and large projects that have been used in prior works [9, 53].

3.2 Data Collection

We queried the JIRA issue repository for all the issues using the JIRA REST APIs in order to extract the bugs (i.e., issues of the type Bug) in the projects. We only consider closed or verified bugs that have been fixed (i.e., bug's resolution = 'Fixed'). We also used the GIT repositories to identify the commits related to the bugs.

To associate the bugs and their bug-fix commits, we follow an approach similar to previous studies [52, 53]. First, we checked out all the release-branches from the GIT repositories. Then, we extracted all commits that contain bug-related words (e.g., bug, fixed, failed, etc) and potential bugs identifiers (e.g., BOOKKEEPER-667) in their commit messages. In order to validate the collected commits, we checked that the bug-identifier in the commits are present in the bugs extracted from JIRA. Since, we are dealing with more than one branch, we are at risk of including duplicate commits in our data set. For example, we found two identical commits fixing bug-issue CAMEL-8091 in release-branches *camel-2.13.x* and *camel-2.14.x*. The aforementioned situation can inflate our metrics and therefore bias/invalidate our conclusions. Therefore, we removed all duplicate commits that have the same message and diff information. Additionally, bug-fix commits were filtered to remove non-source code (e.g., XML, html, log, documentation files, etc) and unit-test files that were part of the bug-fix commit. Out of the 22,803 bug reports extracted, 17,032 were successfully linked to at least one commit. Similarly, out of the 98,297 commits extracted, 21,899 were successfully linked to at least one bug report. It is important to emphasize that in our experiments we only consider bug reports and commits that were explicitly linked. While such bug reports were not considered, and they could potentially be bugs, we cannot know for sure since we are not experts of developers of the case study projects. By considering only bugs that can be linked to commits, we ensure a lower bound for correctness.

Table 1: Information on the releases studied in each of the projects

Project	Releases	# Pre-release Commits	# Post-release Commits	# Files	# Buggy Files
Accumulo	1.4.0	481	206	801	55
	1.5.0	577	178	907	86
	1.6.0	487	204	1,052	70
Bookkeeper	4.0.0	32	24	176	24
	4.1.0	51	32	227	32
	4.2.0	82	21	311	25
Camel	2.9.0	396	93	2,340	100
	2.10.0	320	158	2,570	147
	2.11.0	466	114	2,852	113
Cassandra	1.0.0	529	167	692	151
	1.1.0	339	154	684	114
	1.2.0	470	169	922	156
CXF	2.2	489	256	2,152	287
	2.3	483	106	2,406	91
	2.4	351	217	2,613	170
Derby	10.1.1.0	185	187	1,456	139
	10.2.1.6	520	151	1,558	85
	10.3.1.4	493	166	1,629	130
Felix	scr-1.4.0	1,190	169	2,622	110
	scr-1.6.0	413	400	2,577	209
	scr-1.8.0	965	229	3,412	154
Hive	0.11.0	184	108	1,833	98
	0.12.0	297	243	2,246	222
	0.13.0	365	199	2,550	210
OpenJPA	2.0.0	308	119	1,305	84
	2.1.0	135	86	1,489	56
	2.2.0	102	51	1,509	35
Pig	0.9.0	147	69	985	59
	0.10.0	75	89	1,041	56
	0.11	105	76	1,083	51
Wicket	1.3.0-final	2,137	242	1,278	70
	1.4.0	426	168	1,338	113
	1.5.0	1,502	373	1,530	107
Total	–	15,102	5,224	52,146	3,609

Table 2: Software metrics collected at file-level

Metrics	Description
LOC	Lines of Code
Cyclomatic	Cyclomatic Complexity
Churn	Cumulative code churn
Added Churn	Added lines in the churn
Deleted Churn	Deleted lines in the churn
# Commits	Number of pre-release commits
# Dev	Num. Developers
# Pre-release Bugs	Num. fixed pre-release bugs

Software Metrics. We use the source-code and commit history from the GIT repositories along with the JIRA information to extract code and process metrics from files for three major releases of each project. Not all the projects deal with the releases in the same way. Some projects use only tags, whereas others use only branches (although they periodically merge back into master). In the latter case, GIT records neither the creation date nor the starting point of a branch. To identify the releases and their release-dates, we use the release information posted in the JIRA repositories. We use the closest revisions (i.e., commits) to the release-dates as the identifiers of the releases in the GIT repositories. We selected the three consecutive major releases with the largest number of post-release bug-fixes, in each project (see Table 1 for a detailed list of the major

releases). Given two consecutive releases *A* and *B*, we consider the middle point between them as the end of the post-release period for release *A* and the start of the pre-release period for release *B*. In total, we ended up with a dataset of 52,146 files from all the projects, out of which only 3,609 were buggy files (≈7%).

For each of the releases, we collected code and process metrics during the **pre-release period**. More precisely, we checked out the code repository at each release and extracted various *code metrics* of all the source-code files using the tool UNDERSTAND from Scitools[1]. To extract *process metrics* from the commits in the pre-release, we developed several Python and Bash scripts. For the **post-release period**, we collected the fault-proneness of the files from the bug-fixes. Table 2 provides a brief description of the collected metrics. While other metrics could have been used, we choose the metrics that have been consistently shown in the literature [19] to be the best predictors of faults.

3.3 Identifying False Negative Files

We train six different machine learning techniques using the data from one release (i.e., training-data) and used them to predict defective files in the next-release (i.e., testing-data). Since we have the ground truth of the testing-data, we were able to identify whether a file was correctly or incorrectly predicted. Then, to identify ***FN-files*** (and the other outcomes) with high confidence, we used a majority vote approach. The machine learning techniques used in this works are: Naive Bayes, Classification And Regression Trees (CART) [5], Random Forests [4], Logistic Regression, latent Dirichlet allocation (LDA) and Quadratic discriminant analysis (QDA). In Table 3, we report the number of files for each of the prediction outcomes. We found that 5.3% of the files were identified as ***FN-files***, 1.4% as ***TP-files***, 90.8% as ***TN-files*** and 2.5% as ***FP-files***. On the other hand, if we only consider buggy files, we find that ***FN-files*** and ***TP-files*** account for approximately 79% and 21% on median respectively respectively. In other words, 79% of all buggy files in one release are not identified by prediction models trained in a previous release.

So far, we found that a large number of buggy files cannot be identified by fault prediction models in a forward-release setting. However, in the above experiments we did not perform any data pre-processing step that could potentially improve the models' performance. As a consequence, the validity of our current findings might be compromised. For example, two difficulties associated to fault prediction that could lead to poor generalization performance are the presence of *multi-collinearity between metrics* and the *class imbalance distribution* in SE data sets. Prior works have used two data pre-processing methods namely *Principal Component Analysis* (PCA) and *over/under sampling* in order to alleviate the negative effects of the aforementioned issues respectively [16, 42, 48]. Therefore, in this subsection, we also assess how our findings differ when we consider these data pre-processing methods.

For the PCA pre-processing, we retain the most important components that account for at least 95% of the cumulative variance. For the imbalance issue, we employ a state-of-the-art sampling method called SMOTE[8], which combines both under-sampling and over-sampling. The under-sampling step randomly eliminates instances of the majority class, while the over-sampling step generates new synthetic instances by extrapolating the metrics (i.e features) from the k-nearest neighbors of each of the existing instances in the minority class. In our experiments, we use the SMOTE implementation available in the R package DMwR [59] and set $k = 5$ (i.e., 5-nearest neighbors) as recommended in [8]. It is important to note that the testing data are not re-sampled.

[1]http://www.scitools.com

Table 3: Fault prediction outcomes using majority vote

Project	Training → Testing release	FN	TP	TN	FP
Accumulo	1.4.0 → 1.5.0	70	16	792	29
	1.5.0 → 1.6.0	58	12	943	39
Bookkeeper	4.0.0 → 4.1.0	19	13	165	30
	4.1.0 → 4.2.0	16	9	260	26
Camel	2.9.0 → 2.10.0	136	11	2388	35
	2.10.0 → 2.11.0	97	16	2661	78
Cassandra	1.0.0 → 1.1.0	78	36	552	18
	1.1.0 → 1.2.0	107	49	725	41
CXF	2.2 → 2.3	48	43	2221	94
	2.3 → 2.4	149	21	2420	23
Derby	10.1.1.0 → 10.2.1.6	61	24	1395	78
	10.2.1.6 → 10.3.1.4	118	12	1482	17
Felix	scr-1.4.0 → scr-1.6.0	199	10	2363	5
	scr-1.6.0 → scr-1.8.0	85	69	3120	138
Hive	0.11.0 → 0.12.0	187	35	1998	26
	0.12.0 → 0.13.0	156	54	2271	69
OpenJPA	2.0.0 → 2.1.0	42	14	1405	28
	2.1.0 → 2.2.0	29	6	1448	26
Pig	0.9.0 → 0.10.0	47	9	969	16
	0.10.0 → 0.11	40	11	999	33
Wicket	1.3.0-final → 1.4.0	110	3	1223	2
	1.4.0 → 1.5.0	86	21	1344	79
All	--	1938 [5.3%]	494 [1.4%]	33144 [90.8%]	930 [2.5%]

Another aspect that can impact the performance of fault prediction models is the selection of the parameter values of the machine learning techniques used during the training phase [37, 60]. Since, we used default parameter values in our experiments, our prediction models might show suboptimal performance as pointed out by Tosun et al. [60]. Therefore, in addition to PCA and SMOTE sampling, we re-build our prediction models using the parameter tuning functionality provided by the R package Caret [31] that was recommended in a recent work by Tantithamthavorn et al. [58]. However, unlike Tantithamthavorn et al., we employed 100-times repeated 10-fold cross-validation instead of out-of-sample bootstrap as our validation technique. In a prior study by Mende et al. [36], the authors reported that cross-validation may generate small and imbalanced test-folds that could lead to unstable results. Our experiments did not suffer from this issue because we over-sampled the buggy files in the training data using SMOTE. Although, out-of-sample bootstrap is an attractive method for small datasets (due to its low produced variance), it has a substantial more (no less) bias than k-fold cross-validation [30]. Furthermore, repeated k-fold cross-validation is recommended over other approaches due to its acceptable variance and small bias [27]. Finally, to select the model with the optimal parameter values (if any), we use the AUC measure as suggested in [58].

Due to space limitations, the detailed results for the outcomes of the new prediction models are listed in our online appendix [61]. Here, we only report the average results. We found that 4.4% of the files were identified as ***FN-files***, 2.3% as ***TP-files***, 86.3% as ***TN-files*** and 7.1% as ***FP-files***. In terms of only buggy files, *FN-files* account

for ≈65% of them. This last result represents a moderate improvement of 14% over our original False-negative rate (≈79%). Therefore, even though we tuned the parameters of our prediction models, pre-processed our dataset (through PCA and SMOTE sampling), the prediction cannot identify a large portion of buggy files (65%). Therefore, even tough our results would be slightly different if we pre-processed our dataset, we do not think there would be a marked change in our final results.

4 CASE STUDY RESULTS

4.1 RQ1. Are FN-files different from TP-files?

Motivation. In Section 3.3, we trained six well known prediction techniques on 8 software metrics that have been reported to be good predictors of buggy files. Yet, we found that a large proportion of buggy files are *FN-files* (≈ 70%). In this RQ, we want to understand how ***FN-files*** and ***TP-files*** are different and to what extent. Since ***FN-files*** are buggy files predicted as non-buggy files, we also want determine the similarities between ***FN-files*** and ***TN-files***. More precisely, we compare ***FN-***, ***TP-*** and ***TN-files*** along the software metrics defined in Section 3.2.

Approach. First, to determine whether the metrics for ***FN-files*** and ***TP-files*** are statistically different, we perform a Wilcoxon rank-sum test for each software metric between these two groups of files (e.g., $H_a : LOC_{FN} \neq LOC_{TP}$). We compare ***FN-files*** and ***TN-files*** in a similar manner. We also use boxplots to visualize the comparisons. Second, we analyze the covariance matrices of ***FN-files***, ***TP-files*** and ***TN-files*** to better understand the differences of their joint distributions. In detail, we calculate the covariance matrices: Σ_{FN}, Σ_{TN} and Σ_{TP} and compute the similarity distances $d_1 = \|\Sigma_{FN} - \Sigma_{TN}\|$ and $d_2 = \|\Sigma_{FN} - \Sigma_{TP}\|$ by using the Frobenius norm[2] (which treats matrices like vectors). Then, we compare d_1 and d_2 to determine whether ***FN-files*** are closer to ***TP-files*** or to ***TN-files***.

Results. We found that in most of the cases, the metrics of ***FN-files*** were statistically different from the metrics of ***TP-files*** at p-value << 0.001. The only exception was *# Pre-release Bugs* in *OpenJPA-2.2.0*, for which we obtained a p-value = 0.052. On the other hand, with respect to the metrics of ***FN-files*** and ***TN-files***, we found that only in four projects (Cassandra, Derby, Hive and Wicket) they are significantly different. For the other projects, we found mixed results. Some metrics are statistically different among ***FN-files*** and ***TN-files*** in some projects and releases, whereas in others not. For example, we found that $Churn_{FN}$ and $Churn_{TP}$ are statistically different in *OpenJPA-2.1.0*, but not in *OpenJPA-2.2.0*. We also plot the value of the metrics to visually understand the extent to which our three groups of files differ among each other. We choose Cassandra (not only in this RQ, but in our two RQs), as a case study project to look more closely. The plot for the rest of the projects can be found in our online appendix [61]. Figure 1 shows the boxplots for each of our metrics. We used log-scale to better visualize the values of *LOC*, *Cyclomatic* and *Churn* due to presence of massive outliers. From these boxplots, we clearly observe that ***FN-files*** (green boxes) are far from ***TP-files*** (blue boxes) across all of the metrics (having higher values for the latter). This corroborates the results of the Wilcoxon test showing that ***FN-files*** and ***TP-files*** are very different. When comparing the metrics of ***FN-files*** and ***TN-files***, we observe that the values of *LOC*, *Cyclomatic* and *Churn* for ***FN-files*** are greater than for ***TN-files***. With respect to the other metrics (shown at the bottom end of Figure 1), we found that their boxplots look slightly different, although these differences are small. Therefore, even though ***FN-files*** and ***TN-files*** are different, the values of the metrics for FN-files are closer to those for ***TN-files*** than to those for ***TP-files***. In other words, ***FN-files*** are more similar to ***TN-files*** than to ***TP-files*** in terms of the individual distributions their metrics.

These results do not necessarily imply that the joint distribution for ***FN-files*** is closer to the joint distribution for ***TN-files*** than that for ***TP-files***. Therefore, to quantitatively compare the joint distributions among these three groups of files, we calculate the distance d_1 between the covariance matrices of ***FN-files*** and ***TN-files***, and the distance d_2 between the covariance matrices of ***FN-files*** and ***TP-files*** (see Table 4). We found that for 7 of the projects, the covariance matrix of ***FN-files*** is closer to the covariance matrix of ***TN-files***. The only exceptions are Accumulo, Bookkeeper, Cassandra and Hive, for which the covariance matrix of ***FN-files*** is closer to that of ***TP-files***.

Table 4: Covariance Comparison[†]

Project	Training release → Testing release	d_1	d_2	$FN \rightsquigarrow TN$ $d_1 < d_2$	$FN \rightsquigarrow TP$ $d_1 > d_2$
Accumulo	1.4.0 → 1.5.0	4.71	12.65	X	
	1.5.0 → 1.6.0	133.82	7.42		X
Bookkeeper	4.0.0 → 4.1.0	2.19	1.34		X
	4.1.0 → 4.2.0	0.06	1.26	X	
Camel	2.9.0 → 2.10.0	0.06	4.09	X	
	2.10.0 → 2.11.0	0.12	4.71	X	
Cassandra	1.0.0 → 1.1.0	0.25	2.96	X	
	1.1.0 → 1.2.0	27.60	4.52		X
CXF	2.2 → 2.3	0.17	1.96	X	
	2.3 → 2.4	0.52	2.18	X	
Derby	10.1.1.0 → 10.2.1.6	0.62	401.29	X	
	10.2.1.6 → 10.3.1.4	0.81	57.05	X	
Felix	scr-1.4.0 → scr-1.6.0	0.58	6.69	X	
	scr-1.6.0 → scr-1.8.0	0.09	13.84	X	
Hive	0.11.0 → 0.12.0	54.47	71.71	X	
	0.12.0 → 0.13.0	73.28	32.38		X
OpenJPA	2.0.0 → 2.1.0	0.91	19.45	X	
	2.1.0 → 2.2.0	1.59	26.53	X	
Pig	0.9.0 → 0.10.0	0.18	4.06	X	
	0.10.0 → 0.11	0.30	3.99	X	
Wicket	1.3.0-final → 1.4.0	0.85	18.27	X	
	1.4.0 → 1.5.0	0.13	6.85	X	

[†] The distances are reported in 10^5 scale

Discussion. Fault prediction models try to learn the decision boundary between buggy and non-buggy files, based on the empirical distribution of their metrics. However, if the distribution of a large portion of buggy files is similar to the distribution of non-buggy files (like in our case), then we have reached the limits of the True Positive Rate (≈30% in our case) that can be obtained by any machine learning technique. These results confirm the findings (ceiling effect) by Menzies et al. [42].

Summary. Although ***FN-files*** and ***TP-files*** are both buggy files, we find that the individual distribution as well as the joint distribution of the software metrics for these files are different. At the

[2] http://mathworld.wolfram.com/FrobeniusNorm.html

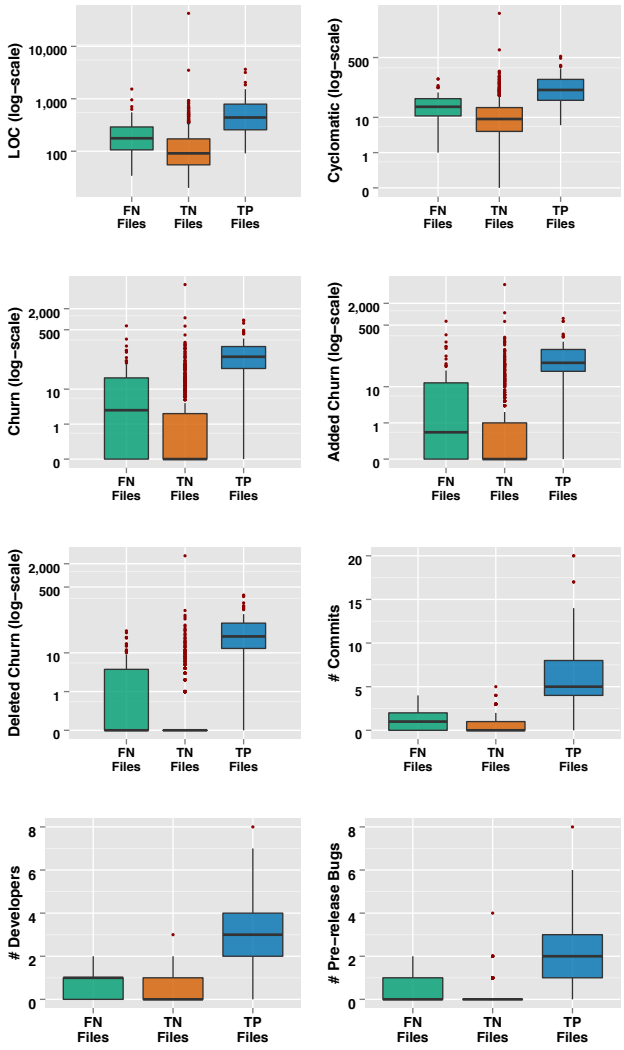


Figure 1: Boxplots of the metrics for FN-files, TN-files and TP-files in Cassandra 1.2.0

same time, we find that the ***FN-files*** and ***TN-files*** (non-buggy files) are more similar in terms of their distributions. Therefore, we require better metrics in order to improve the performance of current fault prediction modeling techniques.

4.2 RQ2. Are the bugs affecting FN-files different from bugs affecting TP-files?

Motivation. In RQ1, we found that these ***FN-files*** are different from ***TP-files***, even though they both are buggy files. In this research question, we want to examine the bugs in these two group of files to see if there are metrics at bug report level that can help to improve the performance of bug prediction models. Therefore, we want to (a) quantitatively examine the bugs along metrics collected from their bug tracking system and (b) qualitatively analysis whether the bugs affecting ***FN-files*** are different from bugs affecting ***TP-files*** in three dimensions: root causes, impacts and affected components. Such analysis will help researchers to collect better metrics for bug prediction models.

RQ2-a. Quantitative Analysis

Approach. For the quantitative analysis, we first use the post-release bug-fix commits of the last two releases in order to link the buggy files to their corresponding post-release bug. Second, given that we know whether a buggy file is either a *FN-file* or *TP-file*, we split the post-release bugs into ***FN-bugs*** and ***TP-bugs***. Third, from the bug reports related to both ***FN-bugs*** and ***FN-bugs***, we

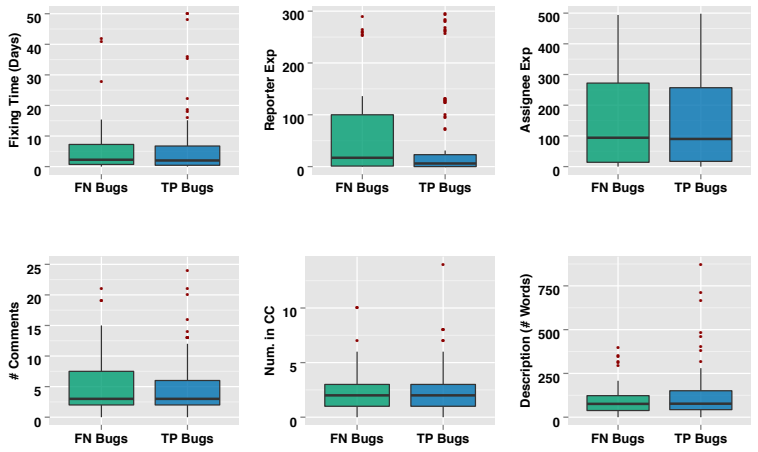


Figure 2: Boxplots of the metrics for FN-bugs and TN-bugs in Cassandra

collect six metrics: Fixing Time, Reporter Experience, Assignee Experience, Number of Comments, Number of Developers in the CC and Description Size. Then, we use Wilcoxon rank-sum test and boxplots to examine if these two groups of bugs are different or not. It is important to note that unlike ***FN-files*** and ***TP-files*** that are related to ***FN-bugs*** and ***TP-bugs*** respectively, ***TN-bugs*** do not have a counterpart in this RQ, since it is a self-contradictory concept (bugs affecting non-buggy files).

Results. In Figure 2, we compare ***FN-bugs*** and ***TP-bugs*** along six bug-report metrics for Cassandra. From Figure 2, we observe that for most of the metrics ***FN-bugs*** are similar to ***TP-bugs***. The only exception was Reporter Experience. It seems that ***FN-bugs*** are reported by reporter with more experience than ***TP-bugs*** are. Additionally, the results of the Wilcoxon rank-sum test show that in most of the cases, there is no statistically significant difference between the metrics of ***FN-bugs*** and ***TP-bugs*** (p-value >> 0.05).

Summary. ***FN-files*** and ***TP-files*** are different. However, we find no evidence to show that ***FN-bugs*** and ***TP-bugs*** are different at bug report level.

RQ2-b. Qualitative Analysis

Approach. For the qualitative analysis, we study three dimensions for both the *FN-bugs* and *TP-bugs*: **Root cause**, **Impact** and affected **Component**. More precisely, we use the bug categories proposed by Tan et al. [57]. Our dataset is comprised of 1025 ***FN-bugs*** and 675 ***TP-bugs***. Therefore, we randomly select 280 ***FN-bugs*** and 235 ***TP-bugs*** in order to have representative samples at 95% confidence level and 5% confidence interval.[3] We used stratified sampling to preserve the original proportions of the projects. For example, ≈10% of the ***FN-bugs*** belongs to Camel, so we randomly select 28 ***FN-bugs*** from Camel. Additionally, invalid bugs were filtered out during the categorization. In this work, an invalid bugs refers to a feature request, maintenance request, compilation error, documentation issue, unit-test error (only affecting unit-test files), or if there is insufficient information. In total, we categorized 242 ***FN-bugs*** and 223 ***TP-bugs***. While in the ideal case, such invalid bugs (38 *FN-bugs* and 12 *TP-bugs*) should be removed in the previous research questions too, there is no automated way to determine them. In this research question we found them through manual analysis. Hence, we remove them in this research question and assume that since they are quite low in number that they do not considerably impact the results of the other questions.

For the identification of the root cause, impact and affected component of a bug, we used the following sources of information:

- **Bug textual information.** The textual information contained in bug reports is a valuable source to diagnose and identify the impact caused by a bug. We primarily use the ***summary*** and ***description*** of the bug report. If the bug report refers to external sources such ***mailing-lists*** or ***forums***, we also include such information. The most popular external sources used by our projects are: Mail-Archive, Nabble, Apache-Reviews and StackOverflow[4]. When the information of the above sources is not clear or partially incomplete, we also use the ***comments*** of the bug report.
- **Commit messages and patches.** The ***message of a fix-commit*** provides detailed information about the changes performed to fix a bug. We mainly use this information to identify the root causes of the bug. In addition, we also use the ***commit patch***, if the message of the commit is not clear.
- **Affected files and Architecture Documentation.** We examine the fullpath of the affected files looking for key words (e.g., `/storage/`, `/net/`, `/shell/`, etc) that can help us to identify the component. When we cannot identify the component using the fullpaths, we look at the files themselves in order to determine

[3] https://www.surveysystem.com/sscalc.htm

[4] https://www.mail-archive.com/, http://www.nabble.com/, https://reviews.apache.org/, http://stackoverflow.com/

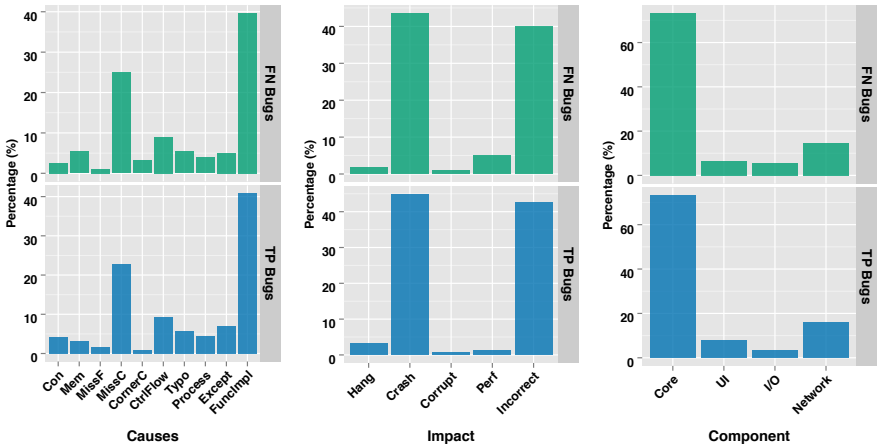


Figure 3: All categories all projects

to which component belong to. It is important to say that before categorizing the bugs, we read the architecture documentation of each project (whenever available) in order to better understand the project structure.

For each bug in our two samples, we collect and aggregate the aforementioned sources of information. To support the categorization process of the bugs, we used the QDA Miner software from Provalis Research[5], which is a tool for qualitative analysis of text-based datasets.

Results. In Figure 3, we show the histograms for ***FN-bugs*** and ***TP-bugs*** along our root cause, impact and component dimensions. From this Figure, we can see that in all of the three dimensions, ***FN-bugs*** and ***TP-bugs*** have similar distributions.

Now, if for example, the distributions of root cause were different for ***FN-bugs*** and ***TP-bugs***, then, we should focus on examining the causes that make ***FN-bugs*** different from ***TP-bugs***. However, we found that the distributions are almost the same, which means that we should focus on examining bugs in the largest root causes. Based on the histograms for root causes, we observe that for both ***FN-bugs*** and ***TP-bugs***, the three largest causes are: Missing Cases (25%-23%), Control Flow (9%-10%) and Wrong Functionality Implementation (40%-42%).

Similarly, we found that the two most common impacts are: crashes (44% and 45%) and incorrect functionalities (40% and 42%), whereas the most commonly affected component is the Core with approximately 66% of the bugs.

Discussion. So far, we found that fault prediction models have limitations (about 70% of buggy files are incorrectly predicted). In addition, we also found in this RQ that a large percentage of ***FN-bugs*** and ***TP-bugs*** are caused by Missing Cases and Control Flow issues (≈34%). Therefore, our research community should investigate methods to identify these kind of semantic bugs. For example, static and dynamic analysis are two promising techniques that can be used to detect them. These techniques have been used in the detection of vulnerabilities [24], malware [44] as well as semantic and concurrent bugs [34]. These approaches might help us to identify buggy files that cannot be correctly predicted by bug prediction models. Rahman et al. [51] have done some initial work along this line of research. They found that prioritizing the warnings of *Static Bug Finder* based on the results of bug prediction models can slightly improve their performance. However, there is still a lot more work and research that needs to be done in this area.

Summary. Based on our qualitative analysis, we find that ***FN-bugs*** and ***TP-bugs*** similar. For both cases, the most common root causes are Missing Cases, Control Flow issues and Wrong Functionality Implementation. We also find that Crashes and Incorrect Functionalities are the most common impacts. Finally, we find that the majority of the bugs affect Core components.

5 THREATS TO VALIDITY

Internal Validity. In order to reduce the introduction of mistakes into our experiments, we used standard tools, statistical libraries and methods. For example, we used Scikit-learn (a machine learning library in Python) and the R package Caret to build our fault prediction models. We also used standard libraries in R to perform our statistical analysis (e.g., Wilcoxon rank-sum test, covariance analysis, etc). Although these tools are not free from errors, they have been used by other researchers in the past for fault prediction [2, 25, 53]. Our qualitative analysis was performed on random samples of bugs and therefore their distribution of the categories may be different from the whole sets of bugs. That said, the sample sizes that we chose for the study are statistically representative at 95% confidence level and 5% confidence interval. We did not define the categories for root causes, impacts and components used in this work. Such categories were borrowed from prior work [57]. During the manual categorization process, we may introduce some bias. In order to alleviate this issue, the first author based his decision on information from a large number of sources (bug reports, com-

[5]http://provalisresearch.com/

mits, mailing list, forums, architecture documentation and even content of the files). Prior work has only considered bug reports and commits[9, 57]. Additionally, we provide all the data of our qualitative analysis (sources of information and the annotation of the categories) in our online appendix [61].

External Validity. Although, we used collected datasets from 11 open source projects written in Java that cover a wide variety of domains and have been used in prior works [9, 52, 53], there are other commercial and open source projects that use different software processes, programming language, bug tracking systems, etc and therefore we cannot assume that our conclusions will generalize to all of them.

Construct Validity. We calculated *# Developers* in a file as the total number of unique emails from the commits related to such file. Sometimes, the same developer might have more than one email and therefore our metric might be inflated. The *# Pre-release bugs* in a file and the identification of buggy files are based on the Jira issues reported as bug that already have being fixed. However, our dataset of bugs may not have all bugs present in the systems (e.g., dormant bugs) or may contains invalid bugs (e.g., enhancement/new-feature issues mistakingly reported as bugs) and therefore our findings might be impacted. The identification of *FN-files* might change from classifier to classifier. In order to mitigate this problem, we use majority vote from 6 classifiers. However, even in this case, the identification may change if a different set of classifiers are considered.

6 CONCLUSION

Prior work on fault prediction have proposed a wide variety of machine learning techniques and software metrics. In this study, we investigated the ceiling effect (reported in previous studies) in terms of the false negative files. By analyzing the outcomes of six machine learning techniques, we found that on average 79% of the buggy files in our eleven case studies are predicted as *FN-files*. Furthermore, we found that the distribution of these *FN-files*is similar to the distribution of non-buggy files (*TN-files*). Our results confirms previous findings about the ceiling effect in fault prediction models trained using product and process metrics. The implications of these results are on the performance of fault prediction models. Prediction models try to learn the decision boundary between buggy and non-buggy files, based on the empirical distributions of the independent variable. Therefore, such models cannot correctly differentiate the two groups of files if the groups have very similar distributions along the independent variables. As a result, we have reached the limits of the True Positive Rate (correctly predicting only 21% of the buggy files in our case studies) based on our metrics.

Surprisingly, our analysis of the ***FN-bugs*** and ***TP-bugs*** shows that they have similar root causes, impact, locations (effected components). For example, among the most common root causes in these two groups of bugs we have: Missing Cases, Control Flow and Incorrect Implementation issues. Additionally, our results show that the percentages of ***FN-bugs*** and ***TP-bugs*** caused by Missing Cases and Control Flow issues are not negligible ($\approx$ 36% and 33% respectively). Therefore, as a research community, we should investigate methods that can exploit the aforementioned similarities (by analyzing the data in bug reports further) in order to determine better software metrics that can enhance fault prediction models. For instance, to detect files affected by these kind of semantic bugs ahead of time, we should consider doing static and dynamic analysis on such files. With this new approach, we might be able to overcome the limits of current fault prediction models. To that end, we should work closely with the Program Analysis Community to come up with better metrics that can help predict faulty files. Therefore, while considerable research has been carried out in the Promise community, there is still a lot of potential to to improve the fault prediction.

REFERENCES

[1] Erik Arisholm, Lionel C. Briand, and Eivind B. Johannessen. 2010. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software* 83, 1 (1 2010), 2–17.

[2] N. Bettenburg, M. Nagappan, and A. E. Hassan. 2012. Think locally, act globally: Improving defect and effort prediction models. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 60–69.

[3] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. 2009. Putting it all together: Using socio-technical networks to predict failures. In *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*. IEEE, 109–119.

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

[5] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. 1984. *Classification and regression trees*. CRC press.

[6] G. Carrozza, D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo. 2013. Analysis and Prediction of Mandelbugs in an Industrial Software System. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. 262–271.

[7] Cagatay Catal and Banu Diri. 2009. A systematic review of software fault prediction studies. *Expert Systems with Applications* 36, 4 (5 2009), 7346–7354.

[8] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* (2002), 321–357.

[9] Tse-Hsun Chen, Meiyappan Nagappan, Emad Shihab, and Ahmed E Hassan. 2014. An empirical study of dormant bugs. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, Vol. undefined. ACM Press, New York, New York, USA, 82–91.

[10] Domenico Cotroneo, Roberto Natella, and Roberto Pietrantuono. 2013. Predicting Aging-related Bugs Using Software Complexity Metrics. *Perform. Eval.* 70, 3 (March 2013), 163–178.

[11] Marco DâĂŹAmbros, Michele Lanza, and Romain Robbes. 2011. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering* 17, 4-5 (8 2011), 531–577.

[12] L. Erlikh. 2000. Leveraging legacy system dollars for e-business. *IT Professional* 2, 3 (0 2000), 17–23.

[13] Wei Fu, Tim Menzies, and Xipeng Shen. 2016. Tuning for Software Analytics: is it Really Necessary? *Information and Software Technology* (4 2016).

[14] Baljinder Ghotra, Shane McIntosh, and Ahmed E. Hassan. 2015. Revisiting the impact of classification techniques on the performance of defect prediction models. (5 2015), 789–800.

[15] Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C. Gall. 2012. Method-level bug prediction. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement - ESEM '12*. ACM Press, New York, New York, USA, 171.

[16] Emanuel Giger, Martin Pinzger, and Harald C. Gall. 2011. Comparing fine-grained source code changes and code churn for bug prediction. In *Proceeding of the 8th working conference on Mining software repositories - MSR '11*. ACM Press, New York, New York, USA, 83.

[17] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy. 2000. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering* 26, 7 (7 2000), 653–661.

[18] Lan Guo, Yan Ma, Bojan Cukic, and Harshinder Singh. 2004. Robust Prediction of Fault-Proneness by Random Forests. In *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*. IEEE, 417–428.

[19] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. 2012. A Systematic Literature Review on Fault Prediction Performance in Software Engineering. *IEEE Transactions on Software Engineering* 38, 6 (11 2012), 1276–1304.

[20] A.E. Hassan and R.C. Holt. 2005. The top ten list: dynamic fault prediction. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, 263–272.

[21] Ahmed E. Hassan. 2009. Predicting faults using the complexity of code changes. In *Proceedings - International Conference on Software Engineering*. IEEE, 78–88.

[22] Kim Herzig, Sascha Just, Andreas Rau, and Andreas Zeller. 2013. Predicting defects using change genealogies. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 118–127.

[23] Kim Sebastian Herzig. 2013. *Mining and untangling change genealogies.* Ph.D. Dissertation. SaarbruÍLcken, UniversitaÍLt des Saarlandes, Diss., 2013.

[24] N. Jovanovic, C. Kruegel, and E. Kirda. 2006. Pixy: a static analysis tool for detecting Web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 6 pp.–263.

[25] Yasutaka Kamei, Shinsuke Matsumoto, Akito Monden, Ken-ichi Matsumoto, Bram Adams, and Ahmed E. Hassan. 2010. Revisiting common bug prediction findings using effort-aware models. In *2010 IEEE International Conference on Software Maintenance*. IEEE, 1–10.

[26] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (6 2013), 757–773.

[27] Ji-Hyun Kim. 2009. Estimating classification error rate: Repeated cross-validation, repeated hold-out and bootstrap. *Computational Statistics & Data Analysis* 53, 11 (9 2009), 3735–3745.

[28] Sunghun Kim, E. James Whitehead, and Yi Zhang. 2008. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering* 34, 2 (3 2008), 181–196.

[29] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. 2007. Predicting Faults from Cached History. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 489–498.

[30] Max Kuhn. 2014. Futility Analysis in the Cross-Validation of Machine Learning Models. (5 2014), 22.

[31] Max Kuhn. 2016. Caret package. (2016). http://cran.r-project.org/package=caret

[32] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. 2008. Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. *IEEE Transactions on Software Engineering* 34, 4 (7 2008), 485–496.

[33] Yi Liu, Taghi M. Khoshgoftaar, and Naeem Seliya. 2010. Evolutionary Optimization of Software Quality Modeling with Multiple Repositories. *IEEE Transactions on Software Engineering* 36, 6 (11 2010), 852–864.

[34] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca a. Popa, and Yuanyuan Zhou. 2007. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. *ACM SIGOPS Operating Systems Review* 41, 6 (10 2007), 103.

[35] Wanwangying Ma, Lin Chen, Yibiao Yang, Yuming Zhou, and Baowen Xu. 2015. Empirical analysis of network measures for effort-aware fault-proneness prediction. *Information and Software Technology* (9 2015).

[36] Thilo Mende. 2010. Replication of defect prediction studies. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering - PROMISE '10*. ACM Press, New York, New York, USA, 1.

[37] Thilo Mende and Rainer Koschke. 2009. Revisiting the evaluation of defect prediction models. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering - PROMISE '09*. ACM Press, New York, New York, USA, 1.

[38] T Mende and R Koschke. 2010. Effort-Aware Defect Prediction Models. In *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 107–116.

[39] Thilo Mende, Rainer Koschke, and Marek Leszak. 2009. Evaluating Defect Prediction Models for a Large Evolving Software System. In *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE, 247–250.

[40] Tim Menzies, Jeremy Greenwald, and Art Frank. 2007. Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Transactions on Software Engineering* 33, 1 (1 2007), 2–13.

[41] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and AyÅ§e Bener. 2010. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering* 17, 4 (5 2010), 375–407.

[42] Tim Menzies, Burak Turhan, Ayse Bener, Gregory Gay, Bojan Cukic, and Yue Jiang. 2008. Implications of ceiling effects in defect predictors. In *Proceedings of the 4th international workshop on Predictor models in software engineering - PROMISE '08*. ACM Press, New York, New York, USA, 47.

[43] Audris Mockus and David M. Weiss. 2000. Predicting risk of software changes. *Bell Labs Technical Journal* 5, 2 (8 2000), 169–180.

[44] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Limits of Static Analysis for Malware Detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. IEEE, 421–430.

[45] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. 2008. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 13th international conference on Software engineering - ICSE '08*. ACM Press, New York, New York, USA, 181.

[46] Nachiappan. Nagappan and Thomas. Ball. 2005. Static analysis tools as early indicators of pre-release defect density. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* ACM Press, New York, New York, USA, 580.

[47] N. Nagappan and T. Ball. 2005. Use of relative code churn measures to predict system defect density. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* IEEe, 284–292.

[48] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. 2006. Mining metrics to predict component failures. In *Proceeding of the 28th international conference on Software engineering - ICSE '06*. ACM Press, New York, New York, USA, 452.

[49] Rahul Premraj and Kim Herzig. 2011. Network Versus Code Metrics to Predict Defects: A Replication Study. In *2011 International Symposium on Empirical Software Engineering and Measurement*. IEEE, 215–224.

[50] Foyzur Rahman and Premkumar Devanbu. 2013. How, and why, process metrics are better. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 432–441.

[51] Foyzur Rahman, Sameer Khatri, Earl T. Barr, and Premkumar Devanbu. 2014. Comparing static bug finders and statistical prediction. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*. ACM Press, New York, New York, USA, 424–434.

[52] Foyzur Rahman, Daryl Posnett, and Premkumar Devanbu. 2012. Recalling the "imprecision" of cross-project defect prediction. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*. ACM Press, New York, New York, USA, 1.

[53] Foyzur Rahman, Daryl Posnett, Israel Herraiz, and Premkumar Devanbu. 2013. Sample size vs. bias in defect prediction. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*. ACM Press, New York, New York, USA, 147.

[54] Emad Shihab, Ahmed E. Hassan, Bram Adams, and Zhen Ming Jiang. 2012. An industrial study on the risk of software changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*. ACM Press, New York, New York, USA, 1.

[55] Emad Shihab, Audris Mockus, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 2011. High-impact defects: a study of breakage and surprise defects. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM Press, New York, New York, USA, 300.

[56] Rainer Storn and Kenneth Price. 1997. Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces. *Journal of Global Optimization* 11, 4 (1997), 341–359.

[57] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. 2013. Bug characteristics in open source software. *Empirical Software Engineering* 19, 6 (6 2013), 1665–1705.

[58] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. Automated Parameter Optimization of Classification techniques for Defect Prediction Models. In *Proc. of the International Conference on Software Engineering (ICSE)*. To appear.

[59] Luis Torgo. 2013. Package DMwR. (2013). https://cran.r-project.org/package=DMwR

[60] Ayse Tosun and Ayse Bener. 2009. Reducing false alarms in software defect prediction by decision threshold optimization. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE, 477–480.

[61] Harold Valdivia-Garcia and Meiyappan Nagappan. 2017. The Characteristics of False-Negatives in File-level Fault Prediction. (2017). https://github.com/harold-valdivia-garcia/fp-in-bug-pred/blob/master/false-negative-dp-appx.pdf

[62] Xin Xia, David Lo, Sinno Jialin Pan, Nachiappan Nagappan, and Xinyu Wang. 2016. HYDRA: Massively Compositional Model for Cross-Project Defect Prediction. *IEEE Transactions on Software Engineering* 42, 10 (2016), 977–998.

[63] Xinli Yang, David Lo, Xin Xia, and Jianling Sun. 2017. TLEL: A two-layer ensemble learning approach for just-in-time defect prediction. *Information and Software Technology* 87 (2017), 206 – 220.

[64] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. 2011. Security versus performance bugs. In *Proceeding of the 8th working conference on Mining software repositories - MSR '11*. ACM Press, New York, New York, USA, 93.

[65] Thomas Zimmermann and Nachiappan Nagappan. 2008. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 13th international conference on Software engineering - ICSE '08*. ACM Press, New York, New York, USA, 531.

[66] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. 2009. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM Press, New York, New York, USA, 91.

[67] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting Defects for Eclipse. In *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*. IEEE, 9–9.

A Large-Scale Study of Modern Code Review and Security in Open Source Projects

Christopher Thompson
University of California, Berkeley
cthompson@cs.berkeley.edu

David Wagner
University of California, Berkeley
daw@cs.berkeley.edu

ABSTRACT

Background: Evidence for the relationship between code review process and software security (and software quality) has the potential to help improve code review automation and tools, as well as provide a better understanding of the economics for improving software security and quality. Prior work in this area has primarily been limited to case studies of a small handful of software projects. ***Aims:*** We investigate the effect of modern code review on software security. We extend and generalize prior work that has looked at code review and software quality. ***Method:*** We gather a very large dataset from GitHub (3,126 projects in 143 languages, with 489,038 issues and 382,771 pull requests), and use a combination of quantification techniques and multiple regression modeling to study the relationship between code review coverage and participation and software quality and security. ***Results:*** We find that code review coverage has a significant effect on software security. We confirm prior results that found a relationship between code review coverage and software defects. Most notably, we find evidence of a negative relationship between code review of pull requests and the number of security bugs reported in a project. ***Conclusions:*** Our results suggest that implementing code review policies within the pull request model of development may have a positive effect on the quality and security of software.

CCS CONCEPTS

• **Security and privacy** → **Economics of security and privacy**; *Software security engineering*; • **Software and its engineering** → **Software development methods**; *Collaboration in software development*;

KEYWORDS

mining software repositories, software quality, software security, code review, multiple regression models, quantification models

1 INTRODUCTION

Modern code review takes the heavyweight process of formal code inspection, simplifies it, and supports it with tools, allowing peers and others to review code as it is being added to a project [37]. Formal software inspection generally involves a separate team of inspectors examining a portion of the code to generate a list of defects to later be fixed [2]. In contrast, modern code review is much more lightweight, focusing on reviewing small sets of changes before they are integrated into the project. In addition, modern code review can be much more collaborative, with both reviewer and author working to find the best fix for a defect or solution to an architectural problem [37]. Code review can help transfer knowledge, improve team awareness, or improve the quality of solutions to software problems [3]. It also has a positive effect on understandability and collective ownership of code [6].

There has been much research into code review process and how it relates to software quality in general [2, 3, 25, 40], but the connection to the security of software has been less thoroughly explored. Software security is a form of software quality, but there is reason to believe that software vulnerabilities may be different from general software defects [8, 26, 41].

Empirical evidence for the relationship between code review process and software security (and software quality) could help improve decision making around code review process. Prior work in this area has primarily been limited to case studies of a small handful of software projects [25, 27, 40]. To the best of our knowledge, we present the first large-scale (in terms of number of repositories studied) analysis of the relationship between code review and software security in open source projects.

GitHub[1], the largest online repository hosting service, encourages modern code review through its pull request system. A pull request (or "PR") is an easy way to accept contributions from outside developers. Pull requests provide a single place for discussion about a set of proposed changes ("discussion comments"), and for comments on specific parts of the code itself ("review comments").

However, not all projects have such a strictly defined or enforced development process, or even consistently review changes made to their code base. Code review coverage is the proportion of changes that are reviewed before being integrated into the code base for a project. Code review participation is the degree of reviewer involvement in review. Previous studies have examined the effects of code review coverage and participation on software quality [25, 40] and software security [27] among a handful of large software projects. We extend and generalize these prior studies by performing quantitative analysis of these effects in a very large corpus of open source software repositories on GitHub.

Our contributions in this paper are as follows:

- We create a novel neural network-based quantification model (a model for predicting the class distribution of a dataset)

[1] https://github.com

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PROMISE'17, November 8, 2017, Toronto, Canada
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5305-2/17/11…$15.00
https://doi.org/10.1145/3127005.3127014

which outperforms a range of existing quantification techniques at the task of estimating the number of security issues in a project's issue tracker.
- We perform a large-scale analysis of projects on GitHub, using multiple regression analysis to study the relationship between code review coverage and participation and: (1) the number of issues in a project, and (2) the number of issues that are security bugs in a project, while controlling for a range of confounding factors.

2 DATA PROCESSING

We focused our investigation on the population of GitHub repositories that had at least 10 pushes, 5 issues, and 4 contributors from 2012 to 2014. This is a conservatively low threshold for projects that have had at least some active development, some active use, and more than one developer, and thus a conservatively low threshold for projects that might benefit from having a set code review process (we hypothesize, but do not investigate, that a higher threshold would yield a population that exhibits *stronger* effects from code review). We used the GitHub Archive[2], a collection of all public GitHub events, to generate a list of all such repositories. This gave us 48,612 candidate repositories in total. From this candidate set, we randomly sampled 5000 repositories.

We wrote a scraper to pull all non-commit data (such as descriptions and issue and pull request text and metadata) for a GitHub repository through the GitHub API[3], and used it to gather data for each repository in our sample. After scraping, we had 4,937 repositories (due to some churn in GitHub repositories).

We queried GitHub to obtain the top three languages used by each repository. For each such language, we manually labeled it on two independent axes: Whether it is a programming language (versus a markup language like HTML, etc.), and whether it is memory-safe or not. A programming language is "memory-safe" if its programs are protected from a variety of defects relating to memory accesses (see Szekeres et al. [39] for a systematization of memory safety issues).

Starting from the set of repositories we scraped, we filtered out those that failed to meet our minimum criteria for analysis:

- Repositories with a `created_at` date later than their `pushed_at` date (these had not been active since being created on GitHub; 13 repositories)
- Repositories with fewer than 5 issues (these typically had their issue trackers moved to a different location; 264 repositories)
- Repositories with fewer than 4 contributors (these typically were borderline cases where our initial filter on distinct committer e-mail addresses over-estimated the number of GitHub users involved; 1008 repositories)
- Repositories with no pull requests (406 repositories)
- Repositories where the primary language is empty (GitHub did not detect any language for the main content of the repository, so we ruled these as not being software project repositories; 83 repositories)

[2] https://www.githubarchive.org/
[3] https://developer.github.com/v3/

Table 1: Summary of our sample of GitHub repositories.

	median	min	max
Stars	42.5	0	338880
Pull Requests	34	1	9060
Contributors	11	4	435
Issues	46	5	8381
Size (B)	4163	22	6090985
Age (days)	1103	0.3078	3048
Avg Commenters per PR	0.9274	0	5
Unreviewed PRs	2	0	1189
% Unreviewed PRs	5.56	0	100
Security Bugs	1	0	175
% Security Bugs	0	0	68

Table 2: Top primary languages in our repository sample.

Language	# Repositories	Total Primary Size (MB)
JavaScript	660	1083.27
Python	418	282.98
Ruby	315	93.62
Java	305	755.75
PHP	268	540.87
C++	172	1147.48
C	140	2929.24
CSS	110	23.13
HTML	107	476.02
C#	83	177.83

- Repositories where none of the top three languages are programming languages (a conservative heuristic for a repository not being a software project; 37 repositories)

This left us with 3,126 repositories in 143 languages, containing 489,038 issues and 382,771 pull requests. Table 1 shows a summary of the repositories in our sample. Table 2 lists the top 10 languages used in our repository sample and the total number of bytes in files of each. We use this dataset for our regression analysis in Section 4.

We found that, in our entire sample, security bugs make up 4.6% of all issues (as predicted by our final trained quantifier, see Section 3 for how we derived this estimate). This proportion is close to the results of Ray et al. [36] where 2% of bug fixing commits were categorized as security-related.

Our dataset and full analysis code is available online[4].

[4] https://doi.org/10.6078/D14X0T

3 QUANTIFYING SECURITY ISSUES

Ultimately, our basic approach involves applying multiple linear regression to study the relationship between code review and security. We count the number of security bugs reported against a particular project and use this as a proxy measure of the security of the project; we then construct a regression model using our code review metrics and other variables as the independent variables. The challenge is that this would seem to require examining each issue reported on the GitHub issue tracker for each of the 3,126 repositories in our sample, to determine whether each issue is security-related or not.

Unfortunately, with 489,038 issues in our repository dataset, it is infeasible to manually label each issue as a security bug or non-security bug. Instead, we use machine learning techniques to construct a *quantifier* that can estimate, for each project, the proportion of issues that are security-related. Our approach is an instance of *quantification*, which is concerned with estimating the distribution of classes in some pool of instances: e.g., estimating the fraction of positive instances, or in our case, estimating the fraction of issues that are security-related. Quantification was originally formalized by Forman [13] and has since been applied to a variety of fields, from sentiment analysis [15] to political science [21] to operations research [13]. We build on the techniques in the literature and extend them to construct an accurate quantifier for our purposes.

One of the insights of the quantification literature is that it can be easier to estimate the fraction of instances (out of some large pool) that are positive than to classify individual instances. In our setting, we found that accurately classifying whether an individual issue is a security bug is a difficult task (reaching at best 80-85% classification accuracy). In contrast, quantification error can be much smaller than classification error (the same models achieved around 8% average absolute quantification error—see "RF CC" in Figure 1). Intuitively, the false positives and the false negatives of the classifier cancel each other out when the proportion is being calculated.

For our research goals, the crucial insight is that we are only concerned with estimating the *aggregate* proportion of security issues in a repository, rather than any of the individual labels (or predicting the label of a new issue). In particular, our regression models only require knowing a count of how many security issues were reported against a particular project, but not the ability to identify which specific issues were security-related. Thus, quantification makes it possible to analyze very large data sets and achieve more accurate and generalizable results from our linear regression models.

Using our best methods described below, we were able to build a quantifier that estimates the fraction of issues that are security-related with an average absolute error of only 4%.

We distinguish our task of quantification from prior work in vulnerability *prediction* models (as in work by Gegick et al. [16] and Hall et al. [19]). We are concerned about textual issues reported in a project's issue tracker, rather than identifying vulnerable components in the project's source code. Our goal is not *prediction*, where we would want to correctly label each new instance we see (such as has been addressed in work by Gegick et al. [17]). Instead, the goal of our models is to estimate the proportion of positive (security-related) instances (issues) in an existing population.

3.1 Basic Quantification Techniques

We start by reviewing background material on quantification. Quantification is a supervised machine learning task: we are given a training set of labeled instances (x_i, y_i). Now, given a test set S, the goal is to estimate what fraction of instances in S are from each class. Quantification differs from standard supervised learning methods in that the class distribution of the training set might differ from the class distribution of the test set: e.g., the proportion of positive instances might not be the same.

Many classifiers work best when the proportion of positive instances in the test set is the same as the proportion of positive instances in the training set (i.e., the test set and training set have the same underlying distribution). However, in quantification, this assumption is violated: we train a single model on a training set with some fixed proportion of positives, and then we will apply it to different test sets, each of which might have a different proportion of positives. This can cause biased results, if care is not taken. Techniques for quantification are typically designed to address this challenge and to tolerate differences in class distribution between the training set and test set [13]; a good quantification approach should be robust to variations in class distribution.

Several methods for quantification have been studied in the literature. The "naive" approach to quantification, called *Classify and Count (CC)* [13], predicts the class distribution of a test set by using a classifier to predict the label y_i for each instance and then counting the number of instances with each label to estimate the proportion of positive instances:

$$\hat{p} = \frac{1}{N} \sum_i y_i .$$

In other words, we simply classify each instance in the test set and then count what fraction of them were classified as positive.

The *Adjusted Count (AC)* method [13] tries to estimate the bias of the underlying classifier and adjust for it. Using k-fold cross-validation, the classifier's true positive rate (tpr) and false positive rate (fpr) can be estimated. For our experiments, we used $k = 10$. The adjusted predicted proportion is then

$$\hat{p}_{AC} = \frac{\hat{p} - fpr}{tpr - fpr}.$$

Some classifiers (such as logistic regression) output not only a predicted class y, but also a probability score—an estimate of the probability that the instance has class y. The *Probabilistic Adjusted Classify and Count (PACC)* method builds on the AC method by using the probability estimates from the classifier instead of the predicted labels [5]. It also uses estimates of the expected true positive and false positive rates (computed using cross-validation, as with AC). The adjusted predicted proportion is then

$$\hat{p}_{PACC} = \frac{\hat{p} - \mathrm{E}[fpr]}{\mathrm{E}[tpr] - \mathrm{E}[fpr]}.$$

3.2 Quantification Error Optimization

More recently, researchers have proposed training models to optimize the quantification error directly instead of optimizing the

classification error and then correcting it post-facto [4, 12, 29]. Forman was the first to use Kullback-Leibler Divergence (KLD), which measures the difference between two probability distributions, as a measure of quantification error [13]. For quantification, KLD measures the difference between the true class distribution and the predicted class distribution. Given two discrete probability distributions P and $\hat{P}$, the KLD is defined as

$$\mathrm{KLD}(P|\hat{P}) = \sum_i P(i) \log \frac{P(i)}{\hat{P}(i)}.$$

The KLD is the amount of information lost when $\hat{P}$ is used to approximate P. A lower KLD indicates that the model will be more accurate at the quantification task. Thus, rather than training a model to maximize accuracy (as is typically done for classification), for quantification we can train the model to minimize KLD.

Esuli and Sebastiani [12] use structured prediction (based on $\mathrm{SVM}_{\mathrm{perf}}$ [22, 23]) to train an SVM classifier that minimizes the KLD loss. They call their quantifier SVM(KLD), and it has been used for sentiment analysis tasks [15]. However, we were unable to reproduce comparable KLD scores on simple test datasets, and found the existing implementation difficult to use. Other researchers report subpar performance from SVM(KLD) compared to the simpler CC, AC, or PACC quantification methods for sentiment analysis tasks [33].

3.3 Our Quantifier

Building on the idea of optimizing for quantification error instead of accuracy, we construct a neural network quantifier trained using TensorFlow [1] to minimize the KLD loss. TensorFlow allows us to create and optimize custom machine learning models and has built-in support for minimizing the cross-entropy loss. We express the KLD in terms of the cross entropy via

$$\mathrm{KLD}(P|\hat{P}) = H(P, \hat{P}) - H(P),$$

that is, the difference of the cross entropy of P and $\hat{P}$ and the entropy of P. Because for any given training iteration the entropy of the true class distribution $H(P)$ will be constant, minimizing the cross entropy $H(P, \hat{P})$ will also minimize the KLD.

We implement a fully-connected feed-forward network with two hidden layers of 128 and 32 neurons, respectively. The hidden layers use the ReLU activation function. The final linear output layer is computed using a softmax function so that the output is a probability distribution. Training uses stochastic gradient descent with mini-batches, using the gradient of the cross entropy loss between the predicted batch class distribution and the true batch class distribution.

Naive random batching can cause the neural network to simply learn the class distribution of the training set. To combat this, we implemented random proportion batching: for each batch, we initially set the batch to contain a random sample of instances from the training set; then we randomly select a proportion of positives p (from some range of proportions) and select a maximum-size subset of the initial set such that the sub-batch proportion is p; finally, we evaluate the model's KLD on that sub-batch. This objective function is equivalent to minimizing the model's average KLD, where we are averaging over a range of proportions p for the true proportion of positives. This training procedure forces the model to be accurate at the quantification task over a wide range of values for the true proportion p of positives, and thus provides robustness to variations in the class distribution.

Our network architecture is kept intentionally simple, and as shown below, it performs very well. We leave heavy optimization of the network design or testing of alternative architectures to future work.

3.4 Feature Extraction

In all of our quantification models we used the following features. We extract features from the text of the issue using the "bag-of-words" approach over all 1- and 2-grams. We extract all of the text from each issue, remove all HTML markup and punctuation from the text, stem each word (remove affixes, such as plurals or "-ing") using the WordNet [35] lemmatizer provided by the Natural Language Toolkit (NLTK) [24], compute token counts, and apply a term-frequency inverse document frequency (TF-IDF) transform to the token counts. Separately, we also extract all of the labels (tags) from each issue, normalize them to lowercase,[5] and apply a TF-IDF transform to obtain additional features. We also count the number of comments on each issue, and extract the primary language of the repository. The combination of all of these were used as our features for our quantifiers.

3.5 Methodology

To train and evaluate our quantifiers, we hand-labeled 1,097 issues to indicate which ones were security issues and which were not. We reserved 10% of them (110 issues) as a test set, and used the remaining 987 issues as a training set.

We selected the issues in our dataset to reduce class imbalance. Because security issues are such a small fraction of the total population of issues, simply selecting a random subset of issues would have left us with too few security issues in the training and test sets. Therefore, we used the tags on each issue as a heuristic to help us find more issues that might be security-related. In particular, we collected a set of issues with the "security" tag, and a set of issues that lacked the "security" tag; both sets were taken from issues created from January 2015 to April 2016, using the GitHub Archive (issue events in the GitHub Archive before 2015 did not have tag information). We restricted the non-security-tagged issues to one per repository, in order to prevent any single project from dominating our dataset. We did not limit the security-tagged issues, due to the limited number of such issues. This left us with 84,652 issues without the "security" tag and 1,015 issues with the "security" tag. We took all of the security-tagged issues along with a random sample of 2,000 of the non-security-tagged issues and scraped all of the text and metadata for each issue using the GitHub API:

- The owner and name of the repository
- The name of the user who created the issue
- The text of the issue
- The list of any tags assigned to the issue
- The text of all comments on the issue
- The usernames of the commenters
- The time the issue was created

[5] To avoid the potential for overfitting due to interaction with how we selected issues to be hand-labeled, we remove the tag "security" if present.

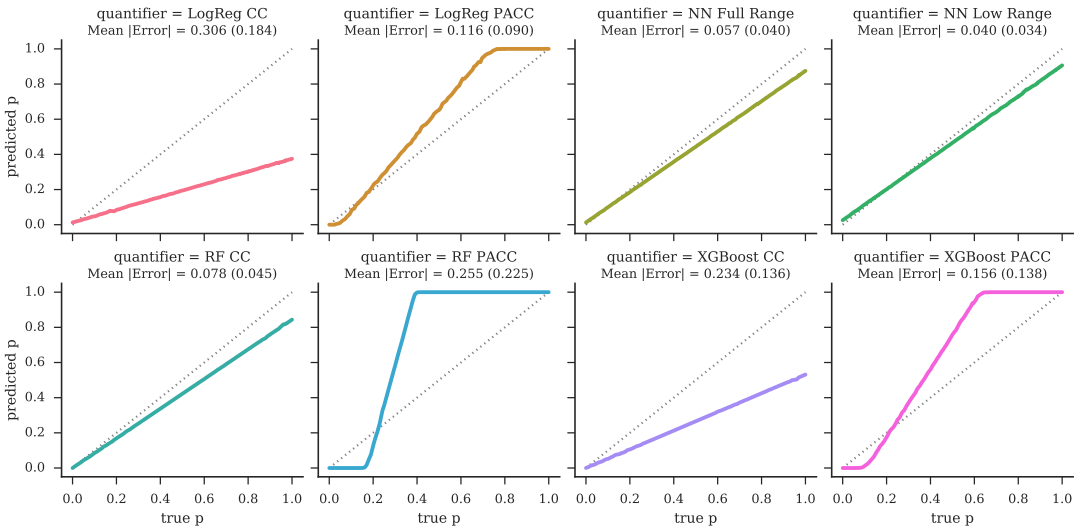


Figure 1: Plots of predicted proportion vs. true proportion for our quantifiers on our reserved test set. The dotted line marks the line $y = x$, which represents the ideal (a quantifier with no error); closer to the dotted line is better. Each quantifier is labeled with the mean absolute error over all proportions and the standard error of that mean. Our "low range" neural network quantifier trained over $p \in [0.0, 0.1]$ shows the best performance, with a mean absolute error of only 4%.

- The time the issue was last updated
- The time the issue was closed (if applicable)

We then hand-labeled 1,097 of these issues, manually inspecting each to determine it was a "security bug" (the uneven number is an artifact of our data processing pipeline and issues in our overall archival sample that had since been deleted). We considered an issue filed against the repository to be a security bug if it demonstrated a defect in the software that had security implications or fell into a known security bug class (such as buffer overruns, XSS, CSRF, hard-coded credentials, etc. [10]), even if it was not specifically described in that way in the bug report. We treated the following as not being security bugs:

- Out-of-date or insecure dependencies
- Documentation issues
- Enhancement requests not related to fundamental insecurity of existing software

We compensated for the generally low prevalence of security bugs by hand-labeling more of the issues from our "security"-tagged set. After hand-labeling we had 224 security bug issues and 873 non-security bug issues.

The "security" tag on GitHub had a precision of 37% and a recall of 99% when compared to our hand-labeling. This very low precision validates our decision to hand-label issues and develop quantification models to analyze our main repository corpus.

3.6 Evaluation

We implemented and tested a variety of quantifiers. We tested CC, AC, and PACC with logistic regression, SVM, random forest, and XGBoost [9] classifiers under a variety of settings, along with various configurations of our neural network-based quantifier. Fig. 1 shows the relative error over all proportions $p \in [0.0, 1.0]$ for the top-performing quantifiers. Our neural network quantifier, when trained on proportions of positives in the range $[0.0, 0.1]$ (the "low range"), performed the best on our test set, with the lowest mean absolute error (0.04) and the lowest mean KLD (0.01), so we adopt it for all our subsequent analysis.

4 REGRESSION DESIGN

In our study design, we seek to answer the following four research questions:

- ***(RQ1)*** *Is there a relationship between code review coverage and the number of issues in a project?*
- ***(RQ2)*** *Is there a relationship between code review coverage and the number of security bugs reported to a project?*
- ***(RQ3)*** *Is there a relationship between code review participation and the number of issues in a project?*
- ***(RQ4)*** *Is there a relationship between code review participation and the number of security bugs reported to a project?*

Our research questions are similar to McIntosh et al. [25] and Ray et al. [36], and we use similar model construction techniques. We use multiple linear regression modeling to describe the relationship between code review coverage and participation (our explanatory variables) and both the number of issues and the number of security bugs filed on each project (our response variables). In our models we also include a number of control explanatory variables (such as the age, size, churn, number of contributors, and stars for each repository). Table 3 explains each of our explanatory variables.

We manually inspect the pairwise relationships between our response variable and each explanatory variable for non-linearity. Following standard regression analysis techniques for improving

Table 3: Description of the control (a), code review coverage (b), and code review participation (c) metrics.

(a) Control Metrics

Metric	Description	Rationale
Forks	Number of repository forks	The more forks a repository has, the more users are contributing pull requests to the project.
Watchers	Number of repository watchers	Watchers are users who get notifications about activity on a project. The more watchers a repository has, the more active eyes and contributors it likely has.
Stars	Number of repository stars	On GitHub, users interested in a project can "star" the repository, making the number of stars a good proxy for the popularity of a project. More popular projects, with more users, will tend to have more bug reports and more active development.
Size	Size of repository (in bytes)	Larger projects have more code. Larger code bases have a greater attack surface, and more places in which defects can occur.
Churn	Lines added and removed in pull requests	The sum of added and removed lines of code in all merged pull requests. Code churn has been associated with defects [31, 32].
Age	Age of repository (seconds)	The difference (in seconds) between the time the repository was created and the time of the latest commit to the repository. Ozment and Schechter [34] found evidence that the number of foundational vulnerabilities reported in OpenBSD decreased as a project aged, but new vulnerabilities are reported as new code is added.
Pull Requests	Number of pull requests	The number of pull requests is used as a proxy for the churn in the code base, which has been associated with both software quality [30, 31] and software security [38].
Memory-Safety	Whether all three of the top languages for a project are memory-safe	Software written in non-memory-safe languages (e.g., C, C++, Objective-C) are vulnerable to entire classes of security bugs (e.g., buffer-overflow, use-after-free, etc.) that software written in memory-safe languages are not [39]. Therefore, we might expect that such software would inherently have more security bugs.
Contributors	Number of authors that have committed to a project	The number of contributors to a project can increase the heterogeneity of the code base, but can also increase the number and quality of code reviews and architectural decisions.

(b) Coverage Metrics (RQ 1, 2)

Metric	Description	Rationale
Unreviewed Pull Requests	The number of pull requests in a project that were merged without any code review	A pull request merged by the same author who created it, without any discussion, implies that the changes have not been code reviewed. Such changes may be more likely to result in both general defects [25] and security bugs [27].
Unreviewed Churn	The total churn in a project from pull requests that were merged without any code review	While churn may induce defects in software, code review may help prevent some defects introduced by churn. We would expect that the lower the amount of unreviewed churn, the lower the number of defects introduced.

(c) Participation Metrics (RQ 3, 4)

Metric	Description	Rationale
Average Commenters	Mean number of commenters on pull requests in a project	Prior work has shown that too many distinct commenters on change requests can actually have a negative impact on software quality [27].
Mean Discussion Comments	Mean number of general discussion comments on pull requests in a project	We expect that increased discussion on a pull request may be indicative of more thorough code review.
Mean Review Comments	Mean number of comments on specific lines of code in pull requests in a project	We expect that more review comments mean more specific changes are being requested during code review, which may be indicative of more thorough code review.

linearity [14], we apply a log transformation ($\log(x+1)$) to each metric with natural number values.

To reduce collinearity, before building our regression models we check the pairwise Spearman rank correlation (ρ) between our explanatory variables. We use Spearman rank correlation since our explanatory variables are not necessarily normally distributed. For any pair that is highly correlated ($|\rho| > 0.7$ [28]), we only include one of the two in our model. To determine whether the coefficients for each explanatory variable are significantly different from zero, we perform a t-test on each to determine a p-value. If a coefficient is not significantly different from zero ($p > 0.05$), we do not report the coefficient in our model summary.

We inspect the resulting QQ plots for each model, and find that all of our models appear to have roughly normal residuals. Full regression diagnostics are included in our supplementary dataset[6].

5 RESULTS

5.1 RQ1: Is there a relationship between code review coverage and the number of issues in a project?

Table 4: Review coverage and overall issues model.

Adjusted R^2	0.5459	
F(8,3117)	470.6***	
log issues	Coef.	Std. Err.
(Intercept)		◇
log forks		†
log watchers		†
log size	0.1972	(0.0087)***
log churn	-0.0209	(0.0089)*
log age	0.0464	(0.0198)*
log contributors		◇
log stars	0.2065	(0.0105)***
log pull requests	0.2822	(0.0238)***
memory safety	0.2306	(0.0435)***
log unreviewed pull requests	0.0880	(0.0169)***
log unreviewed churn		†

† Discarded during correlation analysis ($|\rho| > 0.7$)
◇ $p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

Prior work has found significant effects between code review coverage and defects in an analysis of three large software projects [25]. To investigate this relationship on a more general sample, we build a model using the log number of overall issues in a repository as the response variable, and the log number of unreviewed (but integrated) pull requests as the explanatory variable. The model is presented in Table 4.

[6]https://doi.org/10.6078/D14X0T

The amount of unreviewed churn is too highly correlated with the number of unreviewed pull requests to include in the model. We chose to keep the number of unreviewed pull requests as it is a simpler metric, and we argue is easier to reason about as part of an operational code review process. For completeness, we analyzed the model that used the amount of unreviewed churn instead and found that it had no noticeable effect on model performance. The same was true for RQ2.

We find a small but significant positive relationship between the log number of unreviewed pull requests in a project and the log number of issues the project has. Projects with more unreviewed pull requests tend to have more issues. Holding other variables constant, with a 1% decrease in the number of unreviewed pull requests we would expect to see a 0.08% decrease in the number of issues. Halving the number of unreviewed pull requests we would expect to see 5% fewer issues.

5.2 RQ2: Is there a relationship between code review coverage and the number of security bugs reported to a project?

Table 5: Review coverage and security issues model.

Adjusted R^2	0.3663	
F(8,3117)	226.8***	
log security issues	Coef.	Std. Err.
(Intercept)	-2.1603	(0.3555)***
log forks		†
log watchers		†
log size	0.1691	(0.0086)***
log churn		◇
log age	0.0396	(0.0198)*
log contributors		◇
log stars	0.0883	(0.0103)***
log pull requests	0.2046	(0.0234)***
memory safety	0.2322	(0.0428)***
log unreviewed pull requests	0.0957	(0.0167)***
log unreviewed churn		†

† Discarded during correlation analysis ($|\rho| > 0.7$)
◇ $p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

To explore this question, we replace the response variable of our previous model with the log number of security bugs (as predicted by our quantifier for each project). The model is presented in Table 5.

We find a small but significant positive relationship between the log number of integrated pull requests that are unreviewed and the log number of security bugs a project has. Projects with more unreviewed pull requests tend to have a greater number of security bugs, when controlling for the total numbers of pull requests and issues. Holding other variables constant, with a 1% decrease in the

number of unreviewed pull requests we would expect to see a 0.09% decrease in the number of security bugs. Halving the number of unreviewed pull requests we would expect to see 6% fewer security bugs.

5.3 RQ3: Is there a relationship between code review participation and the number of issues in a project?

Table 6: Review participation and overall issues model.

Adjusted R^2		0.543
F(9,3116)		413***
log issues	Coef.	Std. Err.
(Intercept)		◇
log forks		†
log watchers		†
log size	0.2036	(0.0090)***
log churn		◇
log age		◇
log contributors		◇
log stars	0.1881	(0.0107)***
log pull requests	0.3471	(0.0212)***
memory safety	0.2436	(0.0436)***
log mean commenters per pr		◇
log mean review comments per pr	-0.1105	(0.0477)*
log mean discussion comments per pr		†

† Discarded during correlation analysis ($|\rho| > 0.7$)
◇ $p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

To explore this question, we alter our model to use a response variable of the log number of issues in a project, and we replace our main explanatory variable with the log mean number of commenters on pull requests and the log mean number of review comments per pull request in each project. The model is presented in Table 6. As noted in Table 6, we did not include the mean number of discussion comments in the model as it was highly correlated with the mean number of commenters. We chose to keep the mean number of commenters and the mean number of review comments, to capture both aspects of participation.

We do not find a significant relationship between the average number of commenters on pull requests and the total number of issues. However, we did find a small but significant negative relationship between the log mean number of review comments per pull request and the log number of issues a project has. Projects that, on average, have more review comments per pull request tend to have fewer issues. Holding other variables constant, with a 1% increase in the average number of review comments per pull request we would expect to see a 0.11% decrease in the number of issues. Doubling the average number of review comments per pull request we would expect to see 5.5% fewer issues.

5.4 RQ4: Is there a relationship between code review participation and the number of security bugs reported to a project?

Table 7: Review participation and security issues model.

Adjusted R^2		0.36
F(9,3116)		196.3***
log security bugs	Coef.	Std. Err.
(Intercept)	-2.0655	(0.3569)***
log forks		†
log watchers		†
log size	0.1704	(0.0088)***
log churn		◇
log age		◇
log contributors		◇
log stars	0.0797	(0.0105)***
log pull requests	0.2718	(0.0209)***
memory safety	0.2416	(0.0429)***
log mean commenters per pr		◇
log mean review comments per pr		◇
lot mean discussion comments per pr		†

† Discarded during correlation analysis ($|\rho| > 0.7$)
◇ $p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

To explore this question, we change the response variable in our previous model to be the log number of security bugs reported in a project. The model is presented in Table 7.

We do not find a significant relationship between the average number of commenters on pull requests and the number of security bugs. This result is in contrast with that found by Meneely et al. [27]. While they found that vulnerable files in the Chromium project tended to have more reviewers per SLOC and more reviewers per review, we were unable to replicate the effect. (We note that we looked for an effect across projects, taking a single average for each project, instead of across files within a single project.)

We also did not find a significant relationship between the average number of review comments per pull request and the number of security bugs reported.

6 THREATS TO VALIDITY

6.1 Construct Validity

In order to handle the scale of our sample, we used a machine-learning based quantifier to estimate the dependent variable in our security models (the number of security bugs reported in a project). A quantifier with low precision (high variance in its error)

or one that was skewed to over-predict higher proportions could cause spurious effects in our analysis. We tested our quantifiers on a sample of real issues from GitHub repositories and selected a quantifier model that has good accuracy and high precision (low variance) in its estimations across a wide range of proportions.

This also means that the dependent variable includes some noise (due to quantifier error). We do not expect errors in quantification to be biased in a way that is correlated to code review practices. Linear regression models are able to tolerate this kind of noise. Statistical hypothesis testing takes this noise into account; the association we found is significant at the $p < 0.001$ level (Table 5).

We manually label issues as "security bugs" to train our quantifier. We have a specific notion of what a security bug is (see Section 2), but we have weak ground truth. We used one coder, and there is some grey area in our definition. Our use of quantification should mitigate this somewhat (particularly if the grey area issues are equally likely to be false positives as false negatives, and thus cancel out in the aggregate).

Ideally, we would prefer to measure the security of a piece of software directly, but this is likely impossible. Security metrics are still an area of active research. In this study, we use the number of security issues as an indirect measure of the security of a project, rather than trying to directly assess the security of the software itself. It could be possible to test the predictive validity of our operationalization (the quantifier) on a real-world dataset with known ground truth (such as the Chromium issue tracker[7]).

Our main metric of review coverage (whether a pull request has had any participation from a second party) is somewhat simplistic. One concern is if open source projects tend to "rubber stamp" pull requests (a second participant merges or signs off on a pull request without actually reviewing it): our metric would count this as code review. Our metric is an upper bound on code review coverage.

6.2 External Validity

We intentionally chose a broad population of GitHub repositories in order to try to generalize prior case study-based research on code review and software quality. Our population includes many small or inactive repositories, so our sample may not be representative of security critical software or very popular software. Looking at top GitHub projects might be enlightening, but would limit the generalizability of the results, and might limit the ability to gather a large enough sample.

While GitHub is the largest online software repository hosting service, there may be a bias in open source projects hosted on GitHub, making our sample not truly representative of open source software projects. One concern is that many security critical or very large projects are not on GitHub, or only mirrored there (and their issue tracking and change requests happen elsewhere). For example, the Chromium and Firefox browsers, the WebKit rendering engine, the Apache Foundation projects, and many other large projects fall into this category. Additionally, sampling from GitHub limits us to open source software projects. Commercial or closed source projects may exhibit different characteristics.

[7] https://bugs.chromium.org/p/chromium/

6.3 Effect of Choice of Quantifier

Prior work in defect prediction has found that the choice of machine learning model can have a significant effect on the results of defect prediction studies [7, 18]. We repeated our regression analysis using the naive classify-and-count technique with a random forest classifier model ("RF CC"). This was the best performing of our non-neural network quantification models (see "RF CC" in Figure 1). The regression models produced using the predictions from this quantifier had the same conclusions as our results in Section 5, but with smaller effect sizes on the explanatory variables, and some differences in the effects of the controls. This is likely due to the fact that the RF CC model tends to under-predict the number of security issues compared to our chosen neural network model.

7 RELATED WORK

Heitzenrater and Simpson [20] call for the development of an economics of secure software development, give an overview of how information security economics can lay the foundations, and put forth a detailed research agenda to allow for reasoned investment decisions into software security outcomes. We believe this paper helps to elucidate a small piece of the secure software development economics problem.

Edmundson et al. [11] examined the effects of manual code inspection on a piece of web software with known and injected vulnerabilities. They found that no reviewer was able to find all of vulnerabilities, that experience didn't necessarily reflect accuracy or effectiveness (the effects were not statistically significant), and that false positives were correlated with true positives ($r = 0.39$). It seems difficult to predict the effectiveness of targeted code inspection for finding vulnerabilities.

McIntosh et al. [25] studied the connection between code review coverage and participation and software quality in a case study of the Qt, VTK, and ITK projects. For general defects, they found that both review coverage and review participation are negatively associated with post-release defects.

Meneely et al. [27] analyzed the socio-technical aspects of code review and security vulnerabilities in the Chromium project (looking at a single release). They measured both the thoroughness of reviews of changes to files, and socio-technical familiarity—whether the reviewers had prior experience on fixes to vulnerabilities and how familiar the reviewers and owners are with each other. They performed an association analysis among all these metrics, and found that vulnerable files tended to have many more reviews. In contrast to the results of McIntosh et al., vulnerable files also had more reviewers and participants, which may be evidence of a "bystander apathy" effect. These files also had fewer security-experienced participants.

Ray et al. [36] looked at the effects of programming languages on software quality. They counted defects by detecting "bug fix commits"—commits that fix a defect, found by matching error-related keywords. They found that some languages have a greater association with defects than other languages, but the effect is small. They also found that language has a greater impact on specific categories of defects than it does on defects in general.

8 CONCLUSIONS

We have presented the results of a large-scale study of code review coverage and participation as they relate to software quality and software security. Our results indicate that code review coverage has a small but significant effect on both the total number of issues a project has and the number of security bugs. Additionally, our results indicate that code review participation has a small but significant effect on the total number of issues a project has, but it does not appear to have an effect on the number of security bugs. Overall, code review appears to reduce the number of bugs and number of security bugs.

These findings partially validate the prior case study work of McIntosh et al. [25] and Meneely et al. [27]. However, we did not replicate Meneely's finding of increased review participation having a positive relationship with vulnerabilities. More work would be required to determine if this is a difference in our metrics or a difference in the populations we study. Our results suggest that implementing code review policies within the pull request model of development may have a positive effect on the quality and security of software. However, our analysis only shows correlation—further work is needed to show if there is a causative effect of code review on quality and security.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). Software available from tensorflow.org.

[2] Aybuke Aurum, Håkan Petersson, and Claes Wohlin. 2002. State-of-the-art: software inspections after 25 years. *Software Testing, Verification and Reliability* 12, 3 (2002). https://doi.org/10.1002/stvr.243

[3] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *International Conference on Software Engineering*.

[4] Jose Barranquero, Jorge Díez, and Juan José del Coz. 2015. Quantification-oriented learning based on reliable classifiers. *Pattern Recognition* 48, 2 (2015). https://doi.org/10.1016/j.patcog.2014.07.032

[5] Antonio Bella, Cesar Ferri, Jose Hernandez-Orallo, and Maria Jose Ramirez-Quintana. 2010. Quantification via probability estimators. In *IEEE International Conference on Data Mining*.

[6] M. Bernhart and T. Grechenig. 2013. On the understanding of programs with continuous code reviews. In *International Conference on Program Comprehension*. https://doi.org/10.1109/ICPC.2013.6613847

[7] David Bowes, Tracy Hall, and Jean Petrić. 2017. Software defect prediction: do different classifiers find the same defects? *Software Quality Journal* (2017). https://doi.org/10.1007/s11219-016-9353-3

[8] F. Camilo, A. Meneely, and M. Nagappan. 2015. Do Bugs Foreshadow Vulnerabilities? A Study of the Chromium Project. In *Mining Software Repositories*. https://doi.org/10.1109/MSR.2015.32

[9] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. *CoRR* abs/1603.02754 (2016). http://arxiv.org/abs/1603.02754

[10] Steve Christey. 2011. CWE/SANS Top 25 Most Dangerous Software Errors. (2011). https://cwe.mitre.org/top25/

[11] Anne Edmundson, Brian Holtkamp, Emanuel Rivera, Matthew Finifter, Adrian Mettler, and David Wagner. 2013. An Empirical Study on the Effectiveness of Security Code Review. In *International Symposium on Engineering Secure Software and Systems*. https://doi.org/10.1007/978-3-642-36563-8_14

[12] Andrea Esuli and Fabrizio Sebastiani. 2015. Optimizing text quantifiers for multivariate loss functions. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 9, 4 (2015).

[13] George Forman. 2005. Counting positives accurately despite inaccurate classification. In *European Conference on Machine Learning*.

[14] John Fox. 2008. *Applied regression analysis and generalized linear models* (2nd ed.). Sage.

[15] Wei Gao and Fabrizio Sebastiani. 2015. Tweet sentiment: from classification to quantification. In *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*.

[16] M. Gegick, P. Rotella, and L. Williams. 2009. Predicting Attack-prone Components. In *International Conference on Software Testing Verification and Validation*.

[17] M. Gegick, P. Rotella, and T. Xie. 2010. Identifying security bug reports via text mining: An industrial case study. In *Mining Software Repositories*. https://doi.org/10.1109/MSR.2010.5463340

[18] Baljinder Ghotra, Shane McIntosh, and Ahmed E. Hassan. 2015. Revisiting the Impact of Classification Techniques on the Performance of Defect Prediction Models. In *International Conference on Software Engineering*.

[19] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. 2012. A Systematic Literature Review on Fault Prediction Performance in Software Engineering. *IEEE Transactions on Software Engineering* 38, 6 (Nov 2012). https://doi.org/10.1109/TSE.2011.103

[20] Chad Heitzenrater and Andrew Simpson. 2016. A Case for the Economics of Secure Software Development. In *New Security Paradigms Workshop (NSPW '16)*. https://doi.org/10.1145/3011883.3011884

[21] Daniel J Hopkins and Gary King. 2010. A method of automated nonparametric content analysis for social science. *American Journal of Political Science* 54, 1 (2010).

[22] Thorsten Joachims, Thomas Finley, and Chun-Nam John Yu. 2009. Cutting-plane training of structural SVMs. *Machine Learning* 77, 1 (2009).

[23] Thorsten Joachims, Thomas Hofmann, Yisong Yue, and Chun-Nam Yu. 2009. Predicting structured objects with support vector machines. *Commun. ACM* 52, 11 (2009).

[24] Edward Loper and Steven Bird. 2002. NLTK: The Natural Language Toolkit. In *Proceedings of the Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics*. https://doi.org/10.3115/1118108.1118117 Software available from nltk.org.

[25] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2014. The impact of code review coverage and code review participation on software quality: a case study of the Qt, VTK, and ITK projects. In *Mining Software Repositories*. https://doi.org/10.1145/2597073.2597076

[26] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates. 2013. When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. https://doi.org/10.1109/ESEM.2013.19

[27] A Meneely, ACR Tejeda, B Spates, and S Trudeau. 2014. An empirical investigation of socio-technical code review metrics and security vulnerabilities. In *International Workshop on Social Software Engineering*. https://doi.org/10.1145/2661685.2661687

[28] Lawrence S. Meyers, Glenn Gamst, and Anthony J. Guarino. 2006. *Applied Multivariate Research: Design and Interpretation* (1st ed.). Sage.

[29] L. Milli, A. Monreale, G. Rossetti, F. Giannotti, D. Pedreschi, and F. Sebastiani. 2013. Quantification Trees. In *IEEE International Conference on Data Mining*. https://doi.org/10.1109/ICDM.2013.122

[30] J. C. Munson and S. G. Elbaum. 1998. Code churn: a measure for estimating the impact of code change. In *International Conference on Software Maintenance*. https://doi.org/10.1109/ICSM.1998.738486

[31] N. Nagappan and T. Ball. 2005. Use of relative code churn measures to predict system defect density. In *International Conference on Software Engineering*. https://doi.org/10.1109/ICSE.2005.1553571

[32] Nachiappan Nagappan and Thomas Ball. 2007. Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study. In *International Symposium on Empirical Software Engineering and Measurement*. https://doi.org/10.1109/ESEM.2007.13

[33] Preslav Nakov, Alan Ritter, Sara Rosenthal, Veselin Stoyanov, and Fabrizio Sebastiani. 2016. SemEval-2016 Task 4: Sentiment Analysis in Twitter. In *Proceedings of the 10th International Workshop on Semantic Evaluation*.

[34] Andy Ozment and Stuart E Schechter. 2006. Milk or Wine: Does software security improve with age?. In *USENIX Security*.

[35] Princeton University. 2010. About WordNet. (2010). http://wordnet.princeton.edu

[36] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A large scale study of programming languages and code quality in GitHub. In *International Symposium on Foundations of Software Engineering*. https://doi.org/10.1145/2635868.2635922

[37] Peter Rigby, Brendan Cleary, Frederic Painchaud, Margaret-Anne Storey, and Daniel German. 2012. Contemporary peer review in action: Lessons from open source development. *IEEE Software* 29, 6 (2012).

[38] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne. 2011. Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. *IEEE Transactions on Software Engineering* 37, 6 (Nov 2011). https://doi.org/10.1109/TSE.2010.81

[39] L. Szekeres, M. Payer, T. Wei, and D. Song. 2013. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy*. https://doi.org/10.1109/SP.2013.13

[40] P Thongtanunam and S McIntosh. 2015. Investigating code review practices in defective files: an empirical study of the Qt system. In *Mining Software Repositories*. https://doi.org/10.1109/msr.2015.23

[41] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. 2011. Security Versus Performance Bugs: A Case Study on Firefox. In *Mining Software Repositories*. https://doi.org/10.1145/1985441.1985457

On Applicability of Cross-project Defect Prediction Method for Multi-Versions Projects

Sousuke Amasaki
Okayama Prefectural University
Department of Systems Engineering
Soja, Okayama, Japan
amasaki@cse.oka-pu.ac.jp

ABSTRACT

Context: Cross-project defect prediction (CPDP) research has been popular, and many CPDP methods have been proposed so far. As the straightforward use of Cross-project (CP) data was useless, those methods filter, weigh, and adapt CP data for a target project data. This idea would also be useful for a project having past defect data. **Objective**: To evaluate the applicability of CPDP methods for multi-versions projects. The evaluation focused on the relationship between the performance change and the proximity of older release data to a target project. **Method**: We conducted experiments that compared the predictive performance between using older version data with and without Nearest Neighbor (NN) filter, a classic CPDP method. **Results**: NN-filter could not make clear differences in predictive performance. **Conclusions**: NN-filter was not helpful for improving predictive performance with older release data.

CCS CONCEPTS

• **Software and its engineering → Software defect analysis**;

KEYWORDS

Defect Prediction, Cross-Project, Experiment

ACM Reference format:
Sousuke Amasaki. 2017. On Applicability of Cross-project Defect Prediction Method for Multi-Versions Projects. In *Proceedings of PROMISE , Toronto, Canada, November 8, 2017,* 4 pages.
https://doi.org/10.1145/3127005.3127015

1 INTRODUCTION

Software quality is a primary factor for product success and must be attained efficiently for cost and time-to-market constraints. Software defect prediction is one of most active research areas for that purpose. The software defect prediction utilizes metrics data with actual bug records to train a prediction model. The cost of collecting metrics data is considered as a barrier to its popularization in industry [8, 9]. Insufficient data for projects in early stage of development also inhibits its penetration.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PROMISE , November 8, 2017, Toronto, Canada
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5305-2/17/11…$15.00
https://doi.org/10.1145/3127005.3127015

Cross-project defect prediction (CPDP) is a promising solution for the problems. CPDP trains a defect prediction model with cross-project (CP) data, which is available from a past project in an organization or possibly open access data hosted at the PROMISE Repository, for instance. Recent studies demonstrated that CPDP methods could achieve comparable performance to conventional within-project (WP) defect prediction (e.g. [5, 7]).

CPDP methods filter [10], weigh [4], and adapt [5] CP data for a target project data as the straightforward use of CP data is useless. This nature of CPDP methods could also be useful for a situation of projects going through multiple releases; while using older release data as WPDP could be more suitable for defect prediction than using CP data, such data would be less representative of current status for "concept drift". It is thus expected that applying CPDP methods to metrics data from older releases could improve predictive performance.

This study investigates the applicability of CPDP methods with NN-filter [10] for multi-versions projects. For that purpose, we set the following research questions:

RQ1 Can NN-filter improve the performance of defect prediction with *one* older release data?
RQ2 Can NN-filter improve the performance of defect prediction with *multiple* older release data?

RQ1 asks whether using any version of older release data is helpful for defect prediction by using CPDP methods. As CPDP methods gain some advantages from the volume of CP data, RQ2 asks whether augmenting CP data affects the predictive performance.

2 MATERIALS

2.1 Datasets

Table 1 shows basic statistics of the datasets used in this study. All datasets were collected by Jureczko and Madeyski [3] and served on the PROMISE repository. They are 41 versions of 11 open-source software projects having more than 100 instances. Those projects mainly use Java, and one instance corresponds to one class. They have 20 static code metrics measured using ckjm[1].

2.2 Nearest Neighbor filter

This study adopted Nearest Neighbor (NN) filter [10] as a CPDP method for its popularity and simplicity. Although the authors connected it with Naive Bayes Classifier, the filter method was independent of prediction models. The procedure of NN-filter is as follows:

[1] http://www.spinelis.gr/sw/ckjm

Table 1: Datasets for Experiments

project	sample size	% of defects
ant-1.3	125	16
ant-1.4	178	22
ant-1.5	293	11
ant-1.6	351	26
ant-1.7	745	22
camel-1.0	339	4
camel-1.2	608	36
camel-1.4	872	17
camel-1.6	965	19
ivy-1.1	111	57
ivy-1.4	241	7
ivy-2.0	352	11
jedit-3.2	272	33
jedit-4.0	306	25
jedit-4.1	312	25
jedit-4.2	367	13
jedit-4.3	492	2
log4j-1.0	135	25
log4j-1.1	109	34
log4j-1.2	205	92
lucene-2.0	195	47
lucene-2.2	247	58
lucene-2.4	340	60
poi-1.5	237	59
poi-2.0	314	12
poi-2.5	385	64
poi-3.0	442	64
synapse-1.0	157	10
synapse-1.1	222	27
synapse-1.2	256	34
velocity-1.4	196	75
velocity-1.5	214	66
velocity-1.6	229	34
xalan-2.4	723	15
xalan-2.5	803	48
xalan-2.6	885	46
xalan-2.7	909	99
xerces-init	162	48
xerces-1.2	440	16
xerces-1.3	453	15
xerces-1.4	588	74

(1) Find k neighbors from CP data for each project of the target project data based on Euclidean distance, and
(2) Collect the selected neighbors without duplication into a new cross-project data.

NN-filter has a parameter k that is the number of neighbors to be selected. This study used the same $k = 10$ as well as the past studies.

2.3 Prediction Models

CPDP studies used several prediction models: Naive Bayes Classifier for NN-filter [10] and Transfer Naive Bayes [4], Logistic Regression for TCA+ [5]. Some studies compared multiple prediction models and recommended the best model among them; Peters et al. [6] recommended Random Forests and Herbold [2] recommended SVM with RBF-kernel.

This study is a preliminary one, and we focused on the predictive performance of Naive Bayesian Classifier as well as the original study [10]. `scikit-learn`[2], a machine learning library for Python, was used in this study. Log-transformation was applied to all features in a dataset before training as well as some studies did [1].

2.4 Experiment Procedure

We conducted an experiment of performance comparisons for each multi-versions project following the steps below:

(1) Select the latest project data as a testing data from the multi-versions project,
(2) Select (for RQ1) or combine (for RQ2) older release data from previous versions as training data,
(3) Perform defect prediction with the training data and a filtered training data by NN-filter,
(4) Measure predictive performance with prediction models.

Predictive performance was evaluated with g-measure and AUC. The procedure has no repetition, and only a direct comparison of performance measure values was adopted for evaluation.

3 RESULTS AND DISCUSSIONS

3.1 CPDP with single older release data

Tables 2 and 3 show predictive performance using combined older release data. SP(x) signifies the use of xth recent version for predicting the latest one. For instance, SP(2) means that ant-1.5 was used for training for predicting ant-1.7. The second column denotes whether NN-filter was applied or not. The bold figures denote the best results in each project. If raw and NN show the same value, raw is preferred. Table 3 rarely holds values less than 0.5 and more than 0.8, and we can see that Naive Bayes Classifier achieved moderate performance.

Regarding defect predictions without NN-filter (i.e. "raw"), the tables show that the closest old version SP(1) did not necessarily provide the best training data. The improvements due to using older release data are almost less than 5 points. Only for lucene, SP(2) improved 14 points from SP(1) in g-measure but just 2 points in AUC. Therefore, these differences are negligible in practice. Rather, the performance degradation by using single older version are often observed. For instance, g-measure values dropped suddenly at SP(3) for ant and SP(2) for camel. We can also observe downward trends along with versions for 5 out of 11 projects: jedit, synapse, velocity, xalan, and xerces. For other projects, the predictive performance was retained through multiple releases. These observations imply that some projects suffered from a kind of concept drift while others were stable.

Regarding defect predictions with NN-filter (i.e. "NN"), the tables show no clear performance improvement by NN-filter. Only for lucene, SP(1) with NN-filter showed better g-measure by 8 points

[2] http://scikit-learn.org/

Table 2: Results for using single older release data (G-measure)

name	CPDP	SP(1)	SP(2)	SP(3)	SP(4)
	raw	0.636	0.628	0.446	**0.667**
ant	NN	0.636	0.628	0.446	0.667
	raw	0.529	0.354	**0.534**	
camel	NN	0.527	0.354	0.534	
	raw	**0.689**	0.686		
ivy	NN	0.689	0.686		
	raw	**0.617**	0.554	0.573	0.522
jedit	NN	0.617	0.554	0.572	0.527
	raw	0.63	**0.65**		
log4j	NN	0.63	0.65		
	raw	0.458	0.596		
lucene	NN	0.536	**0.601**		
	raw	0.593	**0.631**	0.539	
poi	NN	0.598	0.631	0.557	
	raw	**0.64**	0.575		
synapse	NN	0.64	0.575		
	raw	**0.507**	0.322		
velocity	NN	0.507	0.321		
	raw	**0.868**	0.845	0.681	
xalan	NN	0.868	0.845	0.678	
	raw	0.734	0.173	0.154	
xerces	NN	**0.745**	0.173	0.153	

Table 3: Results for using single older release data (AUC)

name	CPDP	SP(1)	SP(2)	SP(3)	SP(4)
	raw	0.683	0.678	0.581	**0.691**
ant	NN	0.683	0.678	0.581	0.691
	raw	0.570	0.570	0.534	
camel	NN	0.567	**0.572**	0.534	
	raw	0.692	**0.727**		
ivy	NN	0.692	0.727		
	raw	**0.618**	0.564	0.579	0.540
jedit	NN	0.618	0.564	0.578	0.543
	raw	0.63	**0.651**		
log4j	NN	0.63	0.651		
	raw	0.581	0.604		
lucene	NN	**0.605**	0.601		
	raw	0.672	**0.694**	0.656	
poi	NN	0.676	0.694	0.665	
	raw	**0.64**	0.576		
synapse	NN	0.64	0.576		
	raw	**0.599**	0.522		
velocity	NN	0.599	0.516		
	raw	**0.884**	0.866	0.758	
xalan	NN	0.883	0.866	0.757	
	raw	0.766	0.488	0.155	
xerces	NN	**0.775**	0.488	0.154	

than that without NN-filter. Therefore, the performance degradation and the downward trends due to using older release data are still observed in the results.

A possible reason for this invalidity of NN-filter is that projects with downward trends dramatically changed so that NN-filter could not find a representative subset enough to defect prediction. The difference in the volume of CP data can be another reason. CPDP methods usually take multiple CP data as input while this experiment only took single older release data. The use of multiple older release data may contribute to performance improvements, and we conducted another experiment described in the next section.

In summary, we answer for RQ1 as follows: while the use of single older release data can be helpful, NN-filter can rarely give additional improvements in predictive performance.

3.2 CPDP with combined older release data

Tables 4 and 5 show predictive performance using combined older release data. MP(x) signifies the use of a dataset combining x recent versions for predicting the latest one. For instance, MP(2) is the combination of ant-1.6 and ant-1.5 for predicting ant-1.7. The bold figures denote the best results in each project. If raw and NN show the same value, raw is preferred. If the bold values are better than the best values in Tables 2 and 3, they are underlined. Note that MP(1) is equivalent to SP(1). They are shown in the tables for easy observation. Table 5 holds values more than 0.5 but rarely more than 0.8, and we can see that Naive Bayes Classifier still achieved moderate performance.

Regarding defect predictions without NN-filter (i.e. "raw"), the tables show that the performance degradation and the apparent downward trends were diminished. It could be interpreted that combined data contains the most recent release data and could mitigate the bad effects of older ones observed in Section 3.1. A notable observation is subtle upward trends for some projects. For g-measure, we can see the trend with ant, camel, ivy, jedit, lucene, velocity, and xerces. For AUC, we can see the trend with ant, jedit, and velocity. These observations made a little surprise that the simple combination might contribute to performance improvement.

Regarding defect predictions with NN-filter (i.e. "NN"), NN-filter achieved the best performance with combined older release data for jedit, lucene, velocity, and xerces. With three of the four projects, the performance was better than the best with single older release data (bold figures in Tables 2 and 3). From the results without NN-filter, we can also see that these results heavily depend on the use of combined older release data; NN-filter made a little contribution. In total, NN-filter had no firm effect on the results though it could give a change of performance improvement safely. In summary, we answer for RQ2 as follows: while combined older release data is more helpful than single older release data, NN-filter can rarely give additional improvements in predictive performance.

Table 4: Results for combined older release data (G-measure)

name	CPDP	MP(1)	MP(2)	MP(3)	MP(4)
	raw	0.636	0.655	**0.673**	0.663
ant	NN	0.636	0.655	0.673	0.663
	raw	0.529	0.540	**0.543**	
camel	NN	0.527	0.527	0.535	
	raw	0.689	**0.691**		
ivy	NN	0.689	0.689		
	raw	0.617	0.622	0.628	0.631
jedit	NN	0.617	0.622	0.628	**0.633**
	raw	**0.63**	0.627		
log4j	NN	0.63	0.627		
	raw	0.458	0.467		
lucene	NN	0.536	**0.543**		
	raw	0.593	0.597	0.591	
poi	NN	**0.598**	0.597	0.588	
	raw	**0.64**	0.621		
synapse	NN	0.64	0.626		
	raw	0.507	0.550		
velocity	NN	0.507	**0.556**		
	raw	**0.868**	0.868	0.868	
xalan	NN	0.868	0.863	0.865	
	raw	0.734	0.732	0.741	
xerces	NN	0.745	0.732	**0.757**	

Table 5: Results for combined older release data (AUC)

name	CPDP	MP(1)	MP(2)	MP(3)	MP(4)
	raw	0.683	0.697	**0.71**	0.704
ant	NN	0.683	0.697	0.71	0.704
	raw	0.570	0.572	**0.574**	
camel	NN	0.567	0.564	0.565	
	raw	0.692	**0.694**		
ivy	NN	0.692	0.692		
	raw	0.618	0.623	0.628	0.631
jedit	NN	0.618	0.623	0.628	**0.633**
	raw	**0.63**	0.627		
log4j	NN	0.63	0.627		
	raw	0.581	0.587		
lucene	NN	0.605	**0.608**		
	raw	0.672	0.672	0.669	
poi	NN	**0.676**	0.672	0.671	
	raw	**0.64**	0.631		
synapse	NN	0.64	0.637		
	raw	0.599	0.612		
velocity	NN	0.599	**0.616**		
	raw	**0.884**	0.883	0.883	
xalan	NN	0.883	0.880	0.881	
	raw	0.766	0.768	0.765	
xerces	NN	0.775	0.768	**0.782**	

4 CONCLUSION

This study investigated the use of NN-filter for projects experienced multiple releases. The experiment showed that there was no clear contribution by NN-filter on the predictive performance. In contrast, we can see a weak evidence that combined older release data could be helpful for performance improvement.

The external threats concern that the same effects can be expected on other datasets. This study only used datasets collected from open source software and that these datasets only contain static code metrics collected with automated software. The use of commercial software and different metrics can lead different implications.

The internal threats concern that only NN-filter with the predetermined number of neighbors was supplied as a representative of CPDP methods. The use of NN-filter and the number of neighbors are all often adopted as baselines for evaluation, and we could show a typical behavior of CPDP methods on the situation. As other CPDP methods such as TCA+ [5] have been supplied, further investigation needs to evaluate the effects of using those methods.

ACKNOWLEDGMENT

This work was partially supported by JSPS KAKENHI Grant #15K15975 and Wesco Scientific Promotion Foundation.

REFERENCES

[1] Lin Chen, Bin Fang, Zhaowei Shang, and Yuanyan Tang. 2015. Negative samples reduction in cross-company software defects prediction. *Info. and Softw. Technol.* 62, C (2015), 67–77.
[2] Steffen Herbold. 2013. Training data selection for cross-project defect prediction. In *Proc. of PROMISE '13*. ACM, 6:1–6:10.
[3] Marian Jureczko and Lech Madeyski. 2010. Towards identifying software project clusters with regard to defect prediction. In *Proc. of PROMISE '10*. ACM, 9:1–9:10.
[4] Ying Ma, Guangchun Luo, Xue Zeng, and Aiguo Chen. 2012. Transfer learning for cross-company software defect prediction. *Info. and Softw. Technol.* 54, 3 (2012), 248–256.
[5] Jaechang Nam, S J Pan, and Sunghun Kim. 2013. Transfer defect learning. In *Proc. of ICSE 2013*. IEEE, 382–391.
[6] Fayola Peters, Tim Menzies, and Andrian Marcus. 2013. Better cross company defect prediction. In *Proc. of MSR 2013*. IEEE, 409–418.
[7] Foyzur Rahman, Daryl Posnett, and Premkumar Devanbu. 2012. Recalling the "imprecision" of cross-project defect prediction. In *Proc. of ESEC/FSE '12*. ACM, 61:1–61:11.
[8] Rakesh Rana, Miroslaw Staron, Christian Berger, Jörgen Hansson, Martin Nilsson, and Wilhelm Meding. 2014. The Adoption of Machine Learning Techniques for Software Defect Prediction: An Initial Industrial Validation. In *Proc. of Joint Conference on Knowledge-based Software Engineering*. 270–285.
[9] Ayşe Tosun, Ayşe Bener, Burak Turhan, and Tim Menzies. 2010. Practical considerations in deploying statistical methods for defect prediction: A case study within the Turkish telecommunications industry. *Info. and Softw. Technol.* 52, 11 (2010), 1242–1257.
[10] Burak Turhan, Tim Menzies, Ayşe B Bener, and Justin Di Stefano. 2009. On the relative value of cross-company and within-company data for defect prediction. *Empir. Softw. Eng.* 14, 5 (2009), 540–578.

Boosting Automatic Commit Classification Into Maintenance Activities By Utilizing Source Code Changes

Stanislav Levin
Tel Aviv University
The Blavatnik School of Computer Science
Tel Aviv, Israel
stanisl@post.tau.ac.il

Amiram Yehudai
Tel Aviv University
The Blavatnik School of Computer Science
Tel Aviv, Israel
amiramy@tau.ac.il

ABSTRACT

Background: Understanding maintenance activities performed in a source code repository could help practitioners reduce uncertainty and improve cost-effectiveness by planning ahead and pre-allocating resources towards source code maintenance. The research community uses 3 main classification categories for maintenance activities: Corrective: fault fixing; Perfective: system improvements; Adaptive: new feature introduction. Previous work in this area has mostly concentrated on evaluating commit classification (into maintenance activities) models in the scope of a single software project.

Aims: In this work we seek to design a commit classification model capable of providing high accuracy and Kappa across different projects. In addition, we wish to compare the accuracy and kappa characteristics of classification models that utilize word frequency analysis, source code changes, and combination thereof.

Method: We suggest a novel method for automatically classifying commits into maintenance activities by utilizing source code changes (e.g, statement added, method removed, etc.). The results we report are based on studying 11 popular open source projects from various professional domains from which we had manually classified 1151 commits, over 100 from each of the studied projects. Our models were trained using 85% of the dataset, while the remaining 15% were used as a test set.

Results: Our method shows a promising accuracy of 76% and Cohen's kappa of 63% (considered "Good" in this context) for the test dataset, an improvement of over 20 percentage points, and a relative boost of ~40% in the context of cross-project classification.

Conclusions: We show that by using source code changes in combination with commit message word frequency analysis we are able to considerably boost classification quality in a project agnostic manner.

CCS CONCEPTS

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PROMISE , November 8, 2017, Toronto, Canada
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5305-2/17/11…$15.00
https://doi.org/10.1145/3127005.3127016

KEYWORDS

Software Maintenance, Mining Software Repositories, Predictive Models, Human Factors

1 INTRODUCTION

Three main classification categories for maintenance activities in software projects were identified by Mockus et al.[1]:

- Corrective: fixing faults, functional and non-functional.
- Perfective: improving the system and its design.
- Adaptive: introducing new features into the system.

Understanding these maintenance activities, performed in a source code repository, could help practitioners reduce uncertainty and improve cost-effectiveness [2] by planning ahead and pre-allocating resources towards source code maintenance. Maintenance activity profiles of software projects have therefore been a subject of research in numerous works [1–6]. To determine maintenance activity profiles, one must first classify the activities, which come in the form of developer commits to the version control system (VCS). A widely practiced method for commit classification has been inspecting the commit's textual comment field (a.k.a commit message) [1, 7–9]. Works employing comment based classification models reported the accuracy to average below 60% when used in the scope of a single project, and below 53% when used in the scope of multiple projects (i.e., when a single model was used to classify commits from multiple projects) [9, 10]. Our work is motivated by the following observations:

(1) Existing results rarely consider cross-project classification, which threatens external validity. Hindle et al. [10] explored cross-project classification and reported the accuracy to be ~52%, which is considerably lower than the ~60% range reported by studies dealing with a scope of a single project.
(2) Existing classification results rarely report Cohen's kappa (hence forth Kappa) metric, which accounts for cases where classification categories (a.k.a classes) are unevenly distributed. Such cases make the accuracy metric somewhat misleading. For example, if the corrective class accounted for 98% of the commits, and each of the remaining classes accounted for 1% of the commits, then a simple classification model which always classified commits as corrective would have an impressive accuracy of 98%. Its Kappa on the other hand, would be 0, making this model much less appealing.
(3) Our previous work [5] shows that source code change types as defined by Fluri et al. [11] are statistically significant in the context of maintenance activity categories defined by Mockus et al. [1]. We believe that boosting (i.e. increasing)

the accuracy and kappa characteristics of commit classification into maintenance activities could improve the quality and accuracy of developers' maintenance profiles and the prediction models thereof [5] (see also section 8).

In this work we seek to design a commit classification model capable of providing high accuracy and Kappa across different projects. Our intuition is to try and capture such information, that is not unique to commits made in one project or another, but is of a rather generic nature. Fluri's taxonomy of source code changes for object-oriented programming languages (OOPLs) [11] consists of 48 (47 + an "unknown type") different change types, all of which are project agnostic and describe a meaningful action performed by a developer in a commit (e.g., *statement_delete*, *statement_insert*, *removed_class*, *additional_class* etc). Fluri's taxonomy of source code changes is therefore a nice fit for our need to capture project agnostic information pertaining to developer commits.

RQ. 1. Can source code changes be utilized to boost commit classification into maintenance activities?

RQ. 2. How does the quality of models which utilize source code changes compare to that of traditional models which use word frequency analysis?

2 RELATED WORK

Classifying commits into maintenance activities is commonly accomplished by inspecting commits' comment text and searching for indicative keywords [1, 5, 7, 8, 10]. Such keywords can be obtained using various techniques, such as a word-frequency analysis with normalization (e.g., stemming). Mockus et al. [1] was the first to employ a comment based commit classification, and reported the accuracy to be ~60% when this method was applied in the scope of a single project - a large multi-million line real-time telecommunications software system.

Recent work explored using additional information such as commits' author and module, to classify commits both within a single software project, and cross-projects [10]. Within a single project, the reported accuracy ranged from ~35% to 70% (accuracy fluctuated considerably depending on the project), for cross-project classification the accuracy was ~52% [10].

A slightly different technique was used by Amor et al. [9] who explored classifying maintenance activities in the FreeBSD project by applying a Naive Bayes classifier on commits' comments without an apparent use of keywords. The reported accuracy of classifying a random sample was ~70%. The sample's size was not specified.

A summary of the exiting results for commit classification into maintenance activities can be found in table 1.

Table 1: Classifying commits into maintenance activities, existing results [1, 9, 10]

Scope	Max Accuracy
Single Project	70%
Cross-Project	52%

In this work we were able to achieve an accuracy of 76% and Cohen's kappa of 63% in the context of cross-project commit classification, an improvement of over 20 percentage points and a relative boost of ~40% in accuracy compared to previous results.

3 STATISTICAL METHODS

Picking the optimal classifier for a real-world classification problem is hardly a simple task [12], however, Random Forest (RF) [13, 14] and Gradient Boosting Machine (GBM) [15–19] classifiers are generally considered top performers [12, 18]. In addition, we also use J48 [20, 21], a variation of the C4.5 [22] algorithm. The RF and GBM models are most likely to outperform the simpler J48, but the latter, in contrast to the formers, is capable of providing a human readable representation of its decision tree. We find this ability valuable, since inspecting the decision tree can provide additional insights. An example of a decision tree produced by the J48 classifier can be found in figure 1. It is a decision tree for our keyword based commit classification model, described in section 6.

To evaluate the different commit classification models we employ common statistical measures for classification performance. For a given class $L \in \{\text{adaptive}, \text{corrective}, \text{perfective}\}$, TP_L is the number of commits correctly classified as class L; FP_L is the number of commits incorrectly classified as class L; FN_L is the number of commits of class L that were incorrectly classified.

- $Precision_L = \frac{\text{TP}_L}{\text{TP}_L + \text{FP}_L}$, the number of commits correctly classified as class L, divided by the total number of commits classified as class L.
- $Recall_L = \frac{\text{TP}_L}{\text{TP}_L + \text{FN}_L}$, the number of commits correctly classified as class L, divided by the actual number of L class commits in the dataset.
- $Accuracy = \frac{\sum_{L \in \{a,c,p\}} \text{TP}_L}{\sum_{L \in \{a,c,p\}} (\text{TP}_L + \text{FP}_L)}$, the proportion of correctly classified commits out of all classified commits.
- *No Information Rate (NIR)*, measures the accuracy of a trivial classifier which classifies all commits with using a single class, the one that is most frequent, in our case - corrective.
- *Kappa* - Cohen's kappa, often considered helpful as a measure that can handle both multi-class and imbalanced class problems (see section 1).
- *P-Value [Accuracy > NIR]*, the p-value for the null hypothesis that the "Accuracy ≤ NIR" (i.e., the accuracy of a given predictive model) . A low p-value allows one to reject the null hypothesis in favor of the alternative hypothesis that the "Accuracy > NIR".

4 RESEARCH METHOD

Our work consists of the following main procedures:

(1) Select candidate software projects and harvest their commit data such as commit message and source code changes performed as part of the commit (see sections 5.1, 5.2).
(2) Create a labeled commit dataset by sampling commits and manually labeling them. Each label is a maintenance category, i.e. one of the following: corrective, perfective, or adaptive (see section 5.3).

(3) Inspect the agreement level of the manual labeling procedure by sampling 10% of the labeled dataset created by the first author, and have the second author independently label it (see section 5.4).
(4) Devise predictive models that utilize source code changes for the task of commit classification into maintenance activites (see section 6).
(5) Evaluate the devised models using two mutually exclusive datasets obtained by splitting the labeled dataset into (1) a *training* dataset, consisting of 85% of the labeled dataset, and (2) a *test* dataset, consisting of the remaining 15% of the labeled dataset which was never employed as part of the training process (see section 7).

5 DATA COLLECTION

5.1 Selecting candidate software projects

We use GitHub [23] as the data source for this work due to its popularity and rich query options. Candidate repositories were selected according to the following criteria, which we designed to target data-rich repositories:

- Used the Java programming language
- Had more than 100 stars (i.e. more than 100 users have "liked" these repositories)
- Had more than 60 forks (i.e., more than 60 users have "copied" these repositories for their own use)
- Had their code updated since 2016-01-01 (i.e., these repositories are active)
- Were created before 2015-01-01 (i.e., these repositories have been around)
- Had size over 2MB (i.e. these repositories are of considerable size)

Out of all candidates we selected 11 projects which are well known in the open source community, and cover a wide range of software domains such as IDEs, programming languages (that were implemented in Java), distributed database and storage platforms, and integration frameworks.

(1) **RxJava** - a library for composing asynchronous and event-based programs for the Java VM.
(2) **Intellij Community Edition** - A popular IDE for the Java programming language.
(3) **HBase** - A distributed, scalable, big data store.
(4) **Drools** - A Business Rules Management System solution.
(5) **Kotlin** - A Statically typed programming language for the JVM, Android and the browser by JetBrains.
(6) **Hadoop** - A framework that allows for the distributed processing of large data sets across clusters of computers.
(7) **Elasticsearch** - A distributed search and analytics engine.
(8) **Restlet** - A RESTful web API framework for Java.
(9) **OrientDB** - A Distributed Graph Database with the flexibility of Documents in one product.
(10) **Camel** - An open source integration framework based on known Enterprise Integration Patterns.
(11) **Spring Framework** - An application framework and inversion of control container for the Java platform.

5.2 Distilling source code changes

The source code changes distilling was carried out using a designated VCS mining platform we have built on top of Spark [24, 25], a state of the art framework for large data processing.

After downloading (cloning) the repositories from GitHub, for each repository r where $1 \leq r \leq 11$ we created a series of patch files $\{p_i^r\}_{i=1}^{N_r}$, where N_r is the latest revision number for repository r. Each patch file p_i^r was responsible for transforming repository r from revision r_{i-1} to revision r_i, where r_0 is the empty repository. By initially setting repository r to revision 1 (i.e. the initial revision) and then applying all patches $\{p_i^r\}_{i=2}^{N_r}$ in a sequential manner, the revision history for that repository was essentially replayed. Conceptually, this was equivalent to the case of all developers performing their commits sequentially one by one according to their chronological order.

To distill source code changes as per the taxonomy defined by Fluri et al., we repeatedly applied ChangeDistiller ([26–29]) on every two consecutive revisions of every Java file in every repository we had selected to be part of the dataset, essentially extracting source code changes from the entire project's history.

5.3 Creating a labeled commit dataset

The first author *manually* classified a randomly sampled set of ~100 commits from each of the studied 11 repositories. To improve classification quality the projects' issue tracking systems (e.g. JIRA by Atlassian [30]) was often used. The JIRA contained the tickets occasionally referenced in developers' commits. Such tickets (a.k.a. issues) typically contain additional information about the feature or bug the referencing commit was trying to address. Moreover, tickets sometimes had their own classification categories such as "feature request", "bug", "improvement" etc., but unfortunately they were not very reliable as developers were not always consistent with their categories. For instance, in some cases bug fixes were labeled as "improvement", and while fixing a bug is indeed an improvement, according to the classification categories we use (Mocuks et. al. [1]), bug fixes should be classified corrective while improvements should be classified perfective. Some developers used the term "fix" even when they referenced feature requests, e.g. "fixed issue #N", where "issue #N" spoke of a new feature or an improvement that did not necessarily report a bug. These observations are consistent with Herzig et al [31] who reported that 33.8% of the bug reports they studied were misclassified.

In cases where the lack of supporting information (e.g., indescriptive ticket and / or commit message) prevented us from classifying a certain commit with satisfactory confidence, that commit was dropped from the dataset and replaced by a new one, selected randomly from the same project repository. If we were unable to classify the replacement commit as well, we would repeat this routine until we found a commit that we were able to confidently classify. Further rules of thumb we used for classifying were as follows:

- Javadoc and comment updates were considered perfective
- Fixing a broken unit test or build was considered corrective
- Adding new unit test(s) was considered perfective
- Performance improvements that resulted from an open ticket in the issue tracking system were considered corrective

- Performance improvements that did NOT result from an open ticket in the issue tracking system were considered perfective

We made efforts to prevent class starvation (i.e., not having enough instances of a certain class) which could in turn substantially degrade models' performance, and in case we detected a considerable imbalance in some project's classification categories (a.k.a classes), we added more commits of the starved class from the same project. This balancing was done by repeatedly sampling and manually classifying commits until a commit of the starved class was found. The final dataset consisted of 1151 manually classified commits and was made open access [32]. This dataset contained 100-115 from each project. 43.4% (500 instances) were corrective, 35% (404 instances) were perfective, and 21.4% (247 instances) were adaptive. These commits yielded 33,149 source code changes.

5.4 Inspecting manual labeling agreement

In order to inspect manual classification agreement, we randomly selected 110 commits out of the 1151 commits that had been labeled by the first author, 10 random commits from each of the 11 projects, and had these commits independently labeled again by the second author.

At first the agreement stood at 79%. After discussing the conflicts and sharing the guidelines used by the first author in more detail, the agreement level rose to 94.5%. According to the one sample proportion test [33], the error margin for our observed agreement level was 4.2%, and the estimated asymptotic 95% confidence interval was [90.3%, 98.7%]. This indicates that both authors were in agreement about the labels for the vast majority of cases once they employed the same guidelines (see section 5.3).

6 COMMIT CLASSIFICATION MODELS

We split the labeled dataset into a training dataset and a test dataset, 85% and 15% respectively. The model training phase consists of using 5 time repeated 10-fold validation for each compound model (which boils down to performing a 10-fold cross validation process 5 different times and averaging the results). Then, the trained models are evaluated using the test dataset - the 15% split that did not take part in the model training process. Our statistical computations are carried out in the R statistical environment [34], where we extensively use the R caret package [35] for the purpose of model training and evaluation.

6.1 Utilizing word frequency analysis

First we classified the test dataset (the 15% of the entire labeled dataset) using a naive method to set an initial baseline. The naive method is based solely on searching for pre-defined words gathered from previous work [5], and returning the most frequent class (i.e., corrective) in case none of the keywords were present in a commit's message, see table 2 for more details. The results showed that 34.8% of the commits in the test dataset (60 commits) did not have any of the keywords present in their commit message, and were therefore automatically classified corrective. In addition, we can note the low recall of the perfective class, as opposed to the high recall of the corrective class (which accounts for most of the commits in the classified dataset).

Table 2: Naive model's confusion matrix

classified as \ true class	adaptive	corrective	perfective
adaptive	**18**	2	16
corrective	18	**72**	37
perfective	1	1	**7**
Recall:	48%	96%	11%
Precision:	50%	56%	77%
Accuracy:	56%		
Kappa:	29%		
No Information Rate (NIR):	43%		
P-Value [Accuracy > NIR]:	0.0005		

Due to the high percentage of commits without any of the keywords we had defined, we then tried to fine-tune the keywords we search for. We performed an additional experiment using the same classification method, only this time the keywords were obtained by employing a word frequency analysis with normalization of the commit messages. This time 28% of the commits did not have any of the keywords present in their commit message. These findings led us to believe that the number of commits having none of the keywords used by keyword based classification models is quite considerable.

6.2 Utilizing source code changes (RQ. 1)

The subject of dealing with missing values in a classification problem is broadly covered by Saar-Tsechansky et al. [36], who describe two common methods employed to overcome this issue: (1) imputation, where the missing values are estimated from the data that are present, and (2) reduced-feature models, which employ only those features that will be known for a particular test case (i.e., only a subset of the features that are available for the entire training dataset), so that imputation is not necessary. Since our dataset consists of two different data types, keywords and source code changes, we use reduce-feature models, which are reported to outperform imputation [36] and represent our use-case more naturally. In addition, since the missing feature patterns in our dataset are known in advance, i.e., given a commit only the keywords can be missing, its source code changes are always present, we can precompute and store two models; one to be used when all features are present (keywords + source code changes), and the other when only a subset is available (source code changes only). We define the notion of a **compound** model (similarly to the "classifier lattice" [36]) which uses two separate models for classifying commits with, and without (pre-defined) keywords in their commit message. The classify routine of the compound model is pseudo-coded in listing 1.

Listing 1: Compound model's classify routine

```
classify(commit) {
    if(hasKeywords(commit.comment)) {
        classifyWith(model_KW, commit);
    } else {
        classify(model_¬KW, commit);
    }
}
```

Given a commit C, the compound model first checks if C's commit message has any keywords, if so, the model defined as model_{KW} is used to classify C, otherwise (i.e., no keywords found in C's commit message), the model defined as $\text{model}_{\overline{KW}}$ is used to classify C. Each of the models model_{KW} and $\text{model}_{\overline{KW}}$ may or may not be a reduced-feature model, depending on whether it employs the full set of features (both keywords and source code changes), or only a subset of it (either keywords or source code changes).
We define $\text{model}_{\overline{KW}}$ and model_{KW} to be one of the following model types:

- Keywords model, which relies solely on keywords to classify commits. The features used by this model are keywords obtained by performing the following transformations on the commit comment fields:
  (1) Removed special characters
  (2) Made lower case (case-folding)
  (3) Removed English stopwords
  (4) Removed punctuation
  (5) Striped white-spaces
  (6) Performed stemming
  (7) Adjusted frequencies so that each comment can contribute a given word only once
  (8) Removed custom words such as developer names, projects names, VCSs lingo (e.g., head, patch, svn, trunk, commit), domain specific terms (e.g., http, node, client): "patch", "hbase", "checksum", "code", "version", "byte", "data", "hfile", "region", "schedul", "singl", "can", "yarn", "contribut", "commit", "merg", "make", "trunk", "hadoop", "svn", "ignoreancestri", "node", "also", "client", "hdfs", "mapreduc", "lipcon", "idea", "common", "file", "ideadev", "plugin", "project", "modul", "find", "border", "addit", "changeutilencod", "clickabl", "color", "column", "cach", "jbrule", "drool", "coprocessor", "regionserv", "scan", "resourcemanag", "cherri", "gong", "ryza", "sandi", "xuan", "token", "contain", "shen", "todd", "zhiji", "tan", "wangda", "timelin", "app", "kasha", "kashacherri", "messag", "spr", "camel", "http", "now", "class", "default", "pick", "via".
  (9) We then selected the 10 most frequent words from each of the three maintenance activities in the test dataset:
  – Corrective: (1) *fix* (2) *test* (3) *issu* (4) *use* (5) *fail* (6) *bug* (7) *report* (8) *set* (9) *error* (10) *npe*
  – Perfective: (1) *test* (2) *remov* (3) *use* (4) *fix* (5) *refactor* (6) *method* (7) *chang* (8) *add* (9) *improv* (10) *new*
  – Adaptive: (1) *support* (2) *add* (3) *implement* (4) *new* (5) *allow* (6) *use* (7) *method* (8) *test* (9) *set* (10) *chang*
  It can be seen that some of the words (as obtained by our commit message word frequency analysis) overlap between categories. The words *"test"* and *"use"* appear in all three categories; the word *"fix"* appears in both the corrective and perfective categories; the words *"method"*, *"chang"*, *"add"* and *"new"* appear both in the perfective and adaptive categories; and the word *"set"* appears both in the corrective and adaptive categories. These word overlaps may indicate that keywords alone are insufficient to accurately classify commits into maintenance activities, and need to be augmented with additional information in order to improve classification accuracy.
  For the purpose of building the Keywords model type, we remove multiple occurrences of the same word (so that each word appears only once per maintenance category) and remain with the following set of words: (1) add (2) allow (3) bug (4) chang (5) error (6) fail (7) fix (8) implement (9) improv (10) issu (11) method (12) new (13) npe (14) refactor (15) remov (16) report (17) set (18) support (19) test (20) use.
- (Source Code) Changes based model, which relies solely on source code changes to classify commits. The features used by this model are source code change types [11] obtained by distilling commits, as described earlier in this section.
- Combined (Keyword + Source Code Change Types) model, which uses both keywords and source code change types to classify commits. The features used by this type of models consist of both keywords and source code change types.

A summary of the model components can be found in table 3.

Table 3: Reduced-feature model components

Model Type	Model Features
Keywords	Words
Changes	Source Code Change Types
Combined	Words + Source Code Change Types

For example, a commit where two methods were added (source code change type "additional_functionality"), and one statement was updated (source code change type "statement_updated") and has a commit message that says "Refactored blob logic into separate methods" will be treated differently by each of the model types indicated in table 3.
The Keywords model extracts features represented by tuples of size 20, and given the commit above would extract the following features: $\overbrace{(0 \ldots 1 \ldots 1 \ldots 0)}^{20}$ with "1" in the coordinates that represent the words *"refactor"* and *"method"*. The count of each keyword is at most one, i.e., duplicate keywords are counted only once. Source code changes are ignored, since the Keywords model type does not consider source code changes.
The Changes model extracts features represented by tuples of size 48 (since there are 48 different source code change types), and given the commit above would extract the following features: $\overbrace{(0 \ldots 2 \ldots 1 \ldots 0)}^{48}$ with "2" in the coordinate that represents the

source code change type *"additional_functionality"* and "1" in the coordinate that represents *"statement_updated"*. In contrast to the case of the Keywords model, all occurrences of every source code change type are counted in. Keywords in the commit message are ignored, since the Changes model type does not consider keywords. The Combined model extracts features represented by tuples of size 68 (= 48 source code change types + 20 keywords), and given the commit above would extract the following features: $\overbrace{(\underbrace{0\ldots1\ldots1\ldots0}_{20}\ldots\underbrace{0\ldots0\ldots2\ldots1\ldots0}_{48})}^{68}$, with "2" in the coordinate that represents the source code change type *"additional_functionality"*, and "1" in the coordinates that represent the source code change type *"statement_updated"*, the keyword *"refactor"*, and the keyword *"method"*. The Combined model type captures both keywords and source code change types - hence its name.

In the next sections we evaluate and compare different compound models by considering the different combinations of their model_{KW} and $\text{model}_{\overline{KW}}$ model components. The evaluation process consists of the following steps:

(1) Select the model component model_{KW}
(2) Select the model component $\text{model}_{\overline{KW}}$
(3) Select an underlying classification algorithm for the compound model, which determines the algorithm to be used by each of the model components Model_{KW} and $\text{Model}_{\overline{KW}}$ (J48, GBM, or RF see section 3).

7 RESULTS (RQ. 2)

Table 4 describes an exhaustive set of combinations for selecting the pair of $(\text{Model}_{KW}, \text{Model}_{\overline{KW}})$ models, given that each can be one of the three model types defined in table 3. Each row in table 4 represents a compound model, defined by the selection of $(\text{Model}_{KW}, \text{Model}_{\overline{KW}})$. The classification accuracy and Kappa achieved by a given compound model are reported in the corresponding Accuracy and Kappa columns. The best performing compound model for each classification algorithm is highlighted in lime-green, and the keywords based model (where both Model_{KW} and , $\text{Model}_{\overline{KW}}$ are of the Keywords model type) is highlighted in orange so that it can be easily compared to compound models that utilize source code changes.

Following our main research questions (see section 1), the accuracy and Kappa results for each compound model during the training (see table 4) reveal that the compound models that use either $\boxed{\text{Model}_{\overline{KW}} = \text{Combined}}$ or $\boxed{\text{Model}_{\overline{KW}} = \text{Changes}}$ achieve higher accuracy and Kappa when compared to models with the same Model_{KW} component but with $\text{Model}_{\overline{KW}}$ = Keywords, regardless of the underlying classification algorithm (J48, GBM or RF). This comes as no surprise, as one could expect keyword based models would have trouble accurately classifying commits that do not have any keywords in their commit message. Table 4 also reveals that models that rely solely on commit messages have higher accuracy and kappa than models that rely solely on source code changes (under all three algorithms).

Table 4: Training dataset compound models performance

Alg.	Model_{KW}	$\text{Model}_{\overline{KW}}$	Accuracy	Kappa
J48	Combined		69.0%	51.7%
	Combined	Keywords	67.7%	50.2%
	Combined	Changes	69.2%	51.9%
	Keywords	Combined	69.8%	53%
	Keywords		68.5%	51.5%
	Keywords	Changes	69.9%	53.2%
	Changes	Combined	48.7%	20.1%
	Changes	Keywords	47.4%	17.2%
	Changes		48.8%	18.6%
GBM	Combined		72.0%	56.2%
	Combined	Keywords	69.0%	51.8%
	Combined	Changes	72.0%	55.9%
	Keywords	Combined	71.6%	56.0%
	Keywords		68.5%	51.4%
	Keywords	Changes	71.5%	55.6
	Changes	Combined	54.1%	26.9%
	Changes	Keywords	51.0%	22.4%
	Changes		54.3%	26.9%
RF	Combined		73.1%	57.8%
	Combined	Keywords	69.5%	52.6%
	Combined	Changes	71.9%	55.7%
	Keywords	Changes	72.2%	56.4%
	Keywords	Combined	73.6%	58.9%
	Keywords		69.8%	53.4%
	Changes	Combined	54.5%	26.6%
	Changes	Keywords	50.6%	21.1%
	Changes		52.9%	23.4%

Table 5: Training dataset accuracy, best model per algorithm

Alg.	Min.	1-st Q.	Median	Mean	3-rd Q.	Max.
J48	60.8%	66.4%	70.1%	69.9%	73.4%	80.6%
GBM	60.8%	69.2%	72.1%	72.0%	75.2%	80.8%
RF	65.6%	70.4%	73.4%	73.6%	76.6%	82.8%

Further accuracy and Kappa statistics pertaining to the training stage of the best performing model for each algorithm can be found in table 5 and table 6 respectively. From table 5 and table 6 we can learn that during the training stage, the RF model consistently outperforms the J48 and even the GBM model, in both accuracy and Kappa, across all of the cuts: minimum, 1-st quartile (25-th percentile), median, mean, 3-rd quartile (75-th percentile) and maximum. In particular, the minimum accuracy and Kappa of the RF are notably higher than its competitors.

Table 6: Training dataset Kappa, best model per algorithm

Alg.	Min.	1-st Q.	Median	Mean	3-rd Q.	Max.
J48	38.4%	47.9%	53.4%	53.2%	58.8%	69.7%
GBM	38.3%	51.8%	56.9%	56.2%	60.6%	70.0%
RF	45.5%	54.1%	58.6%	58.9%	63.3%	73.5%

The top performing models were then used to classify the test dataset, consisting of 15% of the entire labeled dataset, see table 7. The ultimate winner was the RandomForest compound model with Model_{KW} = Keywords and $\text{Model}_{\overline{KW}}$ = Combined. A detailed confusion matrix for this champion model can be found in table 8.

Table 7: Test dataset classification performance

Algorithm	Model_{KW}	$\text{Model}_{\overline{KW}}$	**Accuracy**	**Kappa**
J48	Keywords	Changes	70.3%	53.9%
GBM	Combined		72.6%	57.2%
RF	Keywords	Combined	76.7%	63.5%

The decision tree built by the J48 algorithm for our keyword based model (see figure 1) provides some interesting insights regarding its classification process. The word "fix" is the single most indicative word of corrective commits, which aligns well with our intuition, according to which commits that fix faults are likely to include the "fix" noun or verb in the commit message. Given that "fix" did not appear, the words "support" and "allow" are most indicative of adaptive commits, presumably these words are used by developers to indicate the support of a new feature, or the fact that something new is now "allowed" in the system. The combination "implement chang" (stemmed), given that "fix", "support" and "allow" did not appear, is very indicative of either perfective or corrective commits, if however, "implement" is not accompanied by the word "chang" (stemmed), the commit is likely to be adaptive. The (stemmed) word "remov", given that the words "fix", "support", "allow" and "implement" did not appear, is very indicative of perfective commits, perhaps because developers often use it to describe a modification where they remove an obsolete mechanism in favor of a new one.

Table 8: Keywords-Combined RF compound model's confusion matrix for the test dataset

true class / classified as	**adaptive**	**corrective**	**perfective**
adaptive	**28**	5	5
corrective	6	**63**	14
perfective	3	7	**41**
Recall:	75%	84%	68%
Precision:	73%	75%	80%
Accuracy:	76%		
Kappa:	63%		
No Information Rate (NIR):	43%		
P-Value [Accuracy > NIR]:	$< 2e^{-16}$		

We also visualized the maintenance categories keyword (see section 6.2) frequency using a word-cloud [1], which revealed that the word "test" is particularly common in perfective commits, but is generally common in all three maintenance activity types. The word "use" is also common in all three maintenance activity types, but is particularly frequent in the perfective category. The words "fix", "remov" and "support" are quite distinctive of their corresponding maintenance activity types: corrective, perfective and adaptive categories (respectively). The word "add" is common in adaptive commits, as well as "allow".

Similarly, the source code changes frequencies can also be visualized using a source-code-change-type-cloud [2] which reveals that statement related changes, e.g., "statement_insert", "statement_update" and "statement_delete" are the most common change types in all three maintenance activities (corrective, perfective, adaptive). The semantic change type "additional_functionality" is common in both perfective and adaptive commits, but less so in corrective commits.

The term-cloud and J48 keyword based decision tree visualizations provide an intuition for why J48 is likely to outperform a simple word-frequency based classification. In contrast to the word-cloud, which provides "flat" frequencies, the J48 is capable of capturing information pertaining to the presence of multiple keywords in the same commit message, as indicated by the decision tree. In addition, the predictor importance analysis for the champion RF model (omitted for brevity), shows numerous change types rank high, confirming their viability for the classification process.

8 DISCUSSION & APPLICATIONS

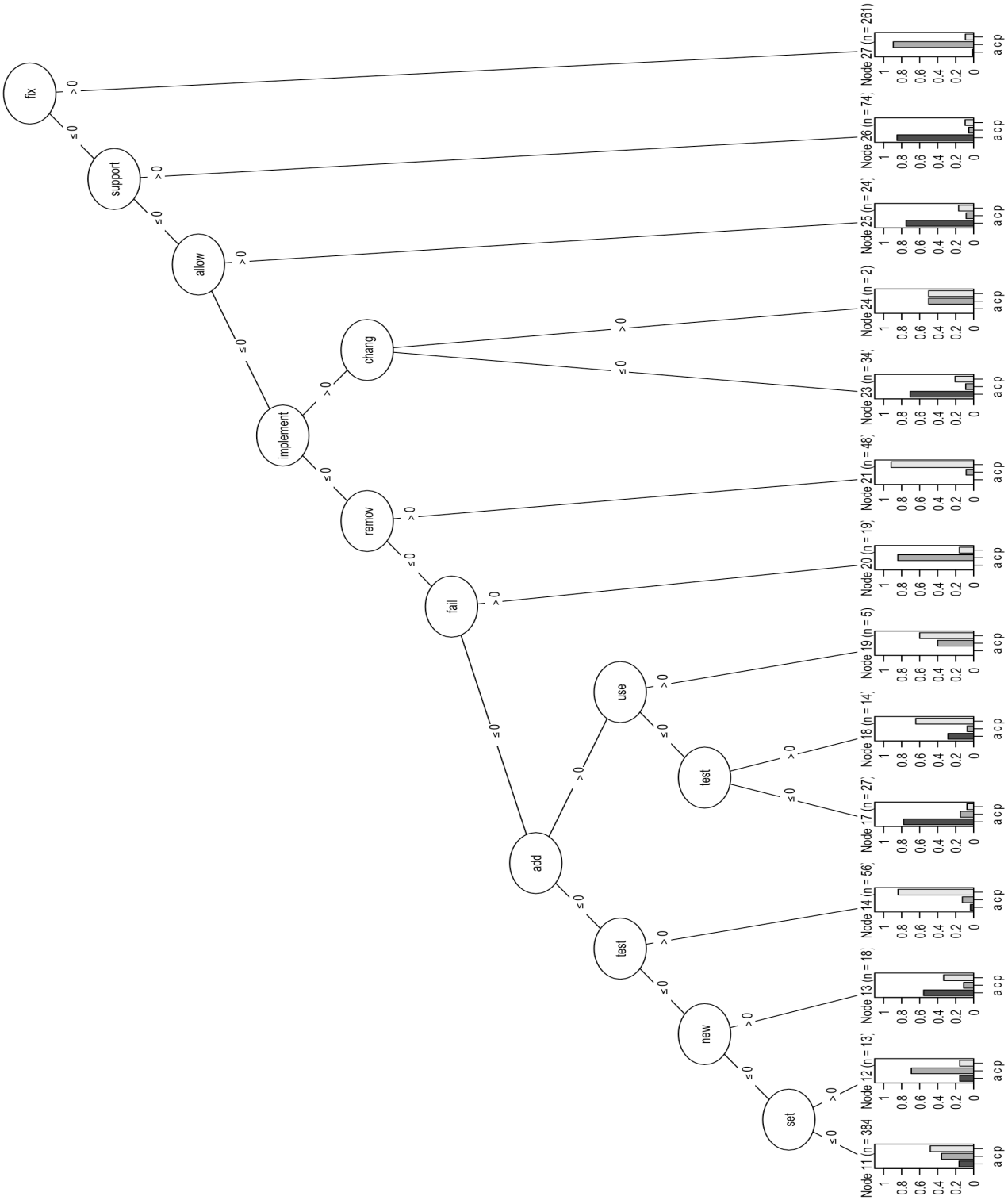
Improving developer's maintenance profile accuracy. Our previous work [5] suggested the notion of a developer's maintenance profile, which describes the amount of commits a given developer made in each of the maintenance categories (corrective, perfective, adaptive). The models we devised in order to predict developers maintenance profiles could benefit from a more accurate classification of commits into maintenance activities as part of their training stage, possibly yielding higher prediction quality.

Identifying anomalies in development process. The manager of a large software project should aim to control and manage its maintenance activity profiles, i.e., the volume of commits made in each maintenance activity. Monitoring for unexpected spikes in maintenance activity profiles and investigating the reasons (root cause) behind them would assist managers and other stakeholders to plan ahead and identify areas that require additional resource allocation. For example, lower corrective profiles could imply that developers are neglecting bug fixing. Higher corrective profiles could imply an excessive bug count. Finding the root cause in cases

[1] https://github.com/staslev/paper-resources/raw/promisedata-2017/Boosting-Automatic-Commit-Classification-Into-Maintenance-Activities-By-Utilizing-Source-Code-Changes/word-cloud.png

[2] https://raw.githubusercontent.com/staslev/paper-resources/promisedata-2017/Boosting-Automatic-Commit-Classification-Into-Maintenance-Activities-By-Utilizing-Source-Code-Changes/change-cloud.png

Figure 1: A J48 Keywords model type (”a“ stands for adaptive, ”c“ for corrective, and ”p“ for perfective)



of significant deviations from predicted values may reveal essential issues whose removal can improve projects' health. Similarly, exceptionally well performing projects can also be a good subject for investigation in order to identify positive patterns.

Improving development team's composition. Building a successful software team is hardly a trivial task as it involves a delicate balance between technological and human aspects [37, 38]. We believe that by using commit classification it would be possible to build reliable developer maintenance activity profiles [5] which could assist in composing balanced teams. We conjecture that composing a team that heavily favors a particular maintenance activity (e.g. adaptive) over the others could lead to an unbalanced development process and adversely affect the team's ability to meet typical requirements such as developing a sustainable number of product features, adhering to quality standards, and minimizing technical debt so as to facilitate future changes.

9 THREATS TO VALIDITY

Threats to Statistical Conclusion Validity are the degree to which conclusions about the relationship among variables based on the data are reasonable.

Our results are based on manually classifying 1151 commits, over 100 commits from each of the studied 11 projects. The projects originated from various professional domains such as IDEs, programming languages, distributed database and storage platforms, and integration frameworks. Each compound model was trained using 5-time repeated 10-fold cross validation. In addition, our commit classifications evaluations demonstrated p-value below 0.01, supporting the statistical validity of the hypothesis accuracy > NIR with high confidence.

Threats to Construct Validity consider the relationship between theory and observation, in case the measured variables do not measure the actual factors.

- Manual Commit Classification. We took the following measures to mitigate manual classification related errors:
  (1) Projects' issue tracking systems were used, and often provided additional information pertaining to commits.
  (2) Commits that did not lend themselves to classification due to lack of supporting information were removed from the dataset and replaced by other commits from the same repository.
  (3) Both authors independently classified 10% of the commits in the dataset used in this work. The observed agreement level was 94.5%, and the asymptotic 95% confidence interval for the agreement level was [90.3%, 98.7%] indicating that both authors agreed about the labels for the vast majority of cases.
- Source Code Change Extraction. ChangeDistiller and the VCS mining platform we have built and used are both software programs, and as such, are not immune to bugs which could result in inaccurate or incomplete source code change extraction.

Threats to External Validity consider the generalization of our findings.

- Programming Language Bias. All analyzed commits were in the Java programming language. It is possible that developers who use other programming languages, have different maintenance activity patterns which have not been explored in the scope of this work.
- Open Source Bias / GitHub. The repositories studied in this paper were all popular open source projects from GitHub, selected according to the criteria described in section 5.1. It may be the case that developers' maintenance activity profiles are different in an open source environment when compared to other environments.
- Popularity Bias. We intentionally selected the popular, data rich repositories. This could limit our results to developers and repositories of high popularity, and potentially skew the perspective on characteristics found only in less popular repositories and their developers.
- Limited Information Bias. The entire dataset, both the training and the test datasets, contained only those commits that we were able to manually classify. At the stage of VCS inspection it can be essentially impossible to actually ascertain the maintenance categories of commits that do not provide enough information traces (comment, ticket id, etc.). The true maintenance category for such commits may only be known to the developers who made them, and even they may no longer recall it soon after they have moved on to their next task.
- Mixed Commits. Recent studies [39, 40] report that commits may involve more than one type of maintenance activity, e.g. a commit that both fixes a bug, and adds a new feature. Our classification method does not currently account for such cases, but this is definitely an interesting direction to be considered for future work (see section 10).

10 CONCLUSIONS AND FUTURE WORK

We suggested a novel method for classifying commits into maintenance activities and used it to devise (and evaluate) a number of models that utilize commit message word frequency analysis and source code change extraction for the purpose of cross-project commit classification into maintenance activities. These models were then evaluated and compared using the accuracy and Kappa metrics with different underlying classification algorithms. Our champion model showed a promising accuracy of 76% and Kappa of 63% when applied on the test dataset which consisted of 172 commits originating from various projects. These results show an improvement of over 20 percentage points, and a relative boost of over 40% when compared to previous results (see table 1), see also table 2 vs. table 8 which depict the commonly used classifier and our champion classifier, respectively. Our work is based on studying 11 popular open source projects from various professional domains, from which we manually classified 1151 commits, ~100 from each of the studied projects. The suggested models were trained using repeated cross validation on 85% of the dataset, and the remaining 15% of the dataset were used as a test set.

We conclude that the answer to RQ 1. is that source code changes can indeed be successfully used to devise high quality models for commit classification into maintenance activities. The answer to

RQ 2. is that models that utilize source code changes are capable of outperforming the reported accuracy of word frequency based models ([9, 10]) from ~60% to ~75%, even when classifying cross-project commits.

In addition, we make the following observations based on our study:

- Using text cleaning and normalization, our word frequency based models were able to achieve an accuracy of 68-69% with Kappa of 51-53% for cross-project commits classification (see table 4).
- Compound models employing both (commit message) word frequency analysis and source code change types for the task of cross-project commit classification were able to achieve up to 73% accuracy with Kappa 59% during the training stage, and up to 76% accuracy with Kappa of 63% (considered "Good" [33]) for the test dataset.
- The RF algorithm outperformed the GBM and J48 in classifying cross-project commits (see table 7 and 8).

Having an accurate classification model and the ability to apply it at scale, it may be possible to automatically classify an unprecedentedly large number of projects and commit activities. This could facilitate the revisiting of the subject of maintenance activity distribution in software projects [4, 6].

It could also be beneficial to explore whether mixed commits ([39, 40]) could be automatically and accurately classified into hybrid categories, e.g. corrective+perfective to indicate that a given commit improves code's structure in addition to fixing a bug.

We believe that employing source code changes to help answer these questions may lead to useful insights for both practitioners and the research community.

REFERENCES

[1] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases," in *Software Maintenance, 2000. Proceedings. International Conference on.* IEEE, 2000, pp. 120–130.

[2] E. B. Swanson, "The dimensions of maintenance," in *Proceedings of the 2nd international conference on Software engineering.* IEEE Computer Society Press, 1976, pp. 492–497.

[3] W. Meyers, "Interview with wilma osborne," *IEEE Software*, vol. 5, no. 3, pp. 104–105, 1988.

[4] B. P. Lientz, E. B. Swanson, and G. E. Tompkins, "Characteristics of application software maintenance," *Communications of the ACM*, vol. 21, no. 6, pp. 466–471, 1978.

[5] S. Levin and A. Yehudai, "Using temporal and semantic developer-level information to predict maintenance activity profiles," in *Proc. ICSME.* IEEE, 2016, pp. 463–468.

[6] S. R. Schach, B. Jin, L. Yu, G. Z. Heller, and J. Offutt, "Determining the distribution of maintenance categories: Survey versus measurement," *Empirical Software Engineering*, vol. 8, no. 4, pp. 351–365, 2003.

[7] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on.* IEEE, 2003, pp. 23–32.

[8] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *ACM sigsoft software engineering notes*, vol. 30, no. 4. ACM, 2005, pp. 1–5.

[9] J. J. Amor, G. Robles, J. M. Gonzalez-Barahona, and A. Navarro, "Discriminating development activities in versioning systems: A case study," in *Proceedings PROMISE.* Citeseer, 2006.

[10] A. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt, "Automatic classication of large changes into maintenance categories," in *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on.* IEEE, 2009, pp. 30–39.

[11] B. Fluri and H. C. Gall, "Classifying change types for qualifying change couplings," in *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on.* IEEE, 2006, pp. 35–45.

[12] M. Fernández-Delgado, E. Cernadas, S. Barro, and D. Amorim, "Do we need hundreds of classifiers to solve real world classification problems," *J. Mach. Learn. Res*, vol. 15, no. 1, pp. 3133–3181, 2014.

[13] T. K. Ho, "The random subspace method for constructing decision forests," *IEEE transactions on pattern analysis and machine intelligence*, vol. 20, no. 8, pp. 832–844, 1998.

[14] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[15] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics*, pp. 1189–1232, 2001.

[16] "R gbm package," https://cran.r-project.org/web/packages/gbm/gbm.pdf, [Online; accessed Nov-2016].

[17] G. Ridgeway, "Generalized boosted models: A guide to the gbm package," *Update*, vol. 1, no. 1, p. 2007, 2007.

[18] R. Caruana and A. Niculescu-Mizil, "An empirical comparison of supervised learning algorithms," in *Proceedings of the 23rd international conference on Machine learning.* ACM, 2006, pp. 161–168.

[19] R. Caruana, N. Karampatziakis, and A. Yessenalina, "An empirical evaluation of supervised learning in high dimensions," in *Proceedings of the 25th international conference on Machine learning.* ACM, 2008, pp. 96–103.

[20] E. Frank, M. Hall, G. Holmes, R. Kirkby, B. Pfahringer, I. H. Witten, and L. Trigg, "Weka," in *Data Mining and Knowledge Discovery Handbook.* Springer, 2005, pp. 1305–1314.

[21] "R/weka interface," https://cran.r-project.org/web/packages/RWeka/index.html, [Online; accessed Nov-2016].

[22] J. R. Quinlan, *C4. 5: programs for machine learning.* Elsevier, 2014.

[23] "Github - the largest open source community in the world," https://github.com/open-source, [Online; accessed 18-April-2016].

[24] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, pp. 10–10, 2010.

[25] "Lightning-fast cluster computing," http://spark.apache.org/, [Online; accessed 11-April-2016].

[26] H. C. Gall, B. Fluri, and M. Pinzger, "Change analysis with evolizer and changedistiller," *IEEE Software*, vol. 26, no. 1, p. 26, 2009.

[27] B. Fluri, E. Giger, and H. C. Gall, "Discovering patterns of change types," in *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on.* IEEE, 2008, pp. 463–466.

[28] M. Martinez, L. Duchien, and M. Monperrus, "Automatically extracting instances of code change patterns with ast analysis," *arXiv preprint arXiv:1309.3730*, 2013.

[29] B. Fluri, M. Wursch, M. PInzger, and H. C. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *Software Engineering, IEEE Transactions on*, vol. 33, no. 11, pp. 725–743, 2007.

[30] Atlassian, "The #1 software development tool used by agile teams," https://www.atlassian.com/software/jira, [Online; accessed 20-Mar-2017].

[31] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: how misclassification impacts bug prediction," in *Proceedings of the 2013 International Conference on Software Engineering.* IEEE Press, 2013, pp. 392–401.

[32] S. Levin and A. Yehudai, "1151 commits with software maintenance activity labels (corrective,perfective,adaptive)," Jul. 2017. [Online]. Available: https://doi.org/10.5281/zenodo.835534

[33] D. G. Altman, *Practical statistics for medical research.* CRC press, 1990.

[34] R Development Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2008, ISBN 3-900051-07-0. [Online]. Available: http://www.R-project.org

[35] "The caret package," http://topepo.github.io/caret/index.html, https://cran.r-project.org/web/packages/caret/caret.pdf, [Online; accessed Nov-2016].

[36] M. Saar-Tsechansky and F. Provost, "Handling missing values when applying classification models," *Journal of machine learning research*, vol. 8, no. Jul, pp. 1623–1657, 2007.

[37] N. Gorla and Y. W. Lam, "Who should work with whom?: building effective software project teams," *Communications of the ACM*, vol. 47, no. 6, pp. 79–82, 2004.

[38] P. J. Guinan, J. G. Cooprider, and S. Faraj, "Enabling software development team performance during requirements definition: A behavioral versus technical approach," *Information Systems Research*, vol. 9, no. 2, pp. 101–125, 1998.

[39] H. A. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Filtering noise in mixed-purpose fixing commits to improve defect prediction and localization," in *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on.* IEEE, 2013, pp. 138–147.

[40] H. Kirinuki, Y. Higo, K. Hotta, and S. Kusumoto, "Hey! are you committing tangled changes?" in *Proceedings of the 22nd International Conference on Program Comprehension.* ACM, 2014, pp. 262–265.

An Extensive Analysis of Efficient Bug Prediction Configurations

Haidar Osman, Mohammad Ghafari, Oscar Nierstrasz
Software Composition Group, University of Bern
Switzerland
{osman,ghafari,oscar}@inf.unibe.ch

Mircea Lungu
University of Groningen
The Netherlands
m.f.lungu@rug.nl

ABSTRACT

Background: Bug prediction helps developers steer maintenance activities towards the buggy parts of a software. There are many design aspects to a bug predictor, each of which has several options, *i.e.*, software metrics, machine learning model, and response variable.

Aims: These design decisions should be judiciously made because an improper choice in any of them might lead to wrong, misleading, or even useless results. We argue that bug prediction *configurations* are intertwined and thus need to be evaluated in their entirety, in contrast to the common practice in the field where each aspect is investigated in isolation.

Method: We use a cost-aware evaluation scheme to evaluate 60 different bug prediction configuration combinations on five open source Java projects.

Results: We find out that the best choices for building a cost-effective bug predictor are change metrics mixed with source code metrics as independent variables, Random Forest as the machine learning model, and the number of bugs as the response variable. Combining these configuration options results in the most efficient bug predictor across all subject systems.

Conclusions: We demonstrate a strong evidence for the interplay among bug prediction configurations and provide concrete guidelines for researchers and practitioners on how to build and evaluate efficient bug predictors.

KEYWORDS

Bug Prediction, Effort-Aware Evaluation

ACM Reference format:
Haidar Osman, Mohammad Ghafari, Oscar Nierstrasz and Mircea Lungu. 2017. An Extensive Analysis of Efficient Bug Prediction Configurations. In *Proceedings of PROMISE , Toronto, Canada, November 8, 2017,* 10 pages.
https://doi.org/10.1145/3127005.3127017

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PROMISE , November 8, 2017, Toronto, Canada
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5305-2/17/11…$15.00
https://doi.org/10.1145/3127005.3127017

1 INTRODUCTION

The main promise of bug prediction is to help software engineers focus their testing and reviewing efforts on those software parts that most likely contain bugs. Under this promise, for a bug predictor to be useful in practice, it must be *efficient*, that is, it must be optimized to locate the maximum number of bugs in the minimum amount of code [31][32][2].[1] Optimizing a bug predictor requires making the right decisions for (i) the independent variables, (ii) the machine learning model, and (iii) the response variable.[2] We call this triple, *bug prediction configurations*.

These configurations are interconnected. The entire configuration should be evaluated in order to provide individual answers for each aspect reliably. However, the advice found in the literature focuses on each aspect of bug prediction in isolation and it is unclear how previous findings hold in a holistic setup. In this paper, we adopt the *Cost-Effectiveness* measure (*CE*), introduced by Arisholm *et al.* [2], to empirically evaluate the different options of each of the bug prediction configurations all at once, shedding light on the interplay among them. Consequently, we pose and answer the following research questions:

RQ1: What type of software metrics are cost-effective? We find that using a mix of source code metrics and change metrics yields the most cost-effective predictors for all subject systems in the studied dataset. We observe that change metrics alone can be a good option, but we advise against using source code metrics alone. These findings contradict the advice found in the literature that object-oriented metrics hinders the cost-effectiveness of models built using change metrics [2]. In fact although source code metrics are the worst metrics set, it can still be used when necessary, but with the right configuration combination.

RQ2: What prediction model is cost-effective? In this study we compare five machine learning models: Multilayer Perceptron, Support Vector Machines, Linear Regression, Random Forest, and K-Nearest Neighbour. Our results show that Random Forest stands out as the most cost-effective one. Support Vector Machines come a close second. While some previous studies suggest that Random Forest performs generally better than other machine learning models [18], other studies note that Random Forest does not perform as well [20]. Our findings suggest that Random Forest performs the best with respect to cost-effectiveness.

[1] Efficient bug prediction as we define it, is sometimes referred to as effort-aware bug prediction in the literature
[2] Also known as the dependent variable or the output variable

RQ3: What is the most cost-effective response variable to predict? We establish that predicting the number of bugs in a software entity is the most cost-effective approach and predicting bug proneness is the least cost-effective one. To our knowledge, this research question has not been investigated before in the literature.

RQ4: Is there a configuration combination that consistently produces highly cost-effective bug predictors? Here we evaluate all configurations at once to provide more reliable guidelines for building cost-effective bug predictors. We conclude that *both source code and change metrics as independent variables mixed, Random Forest as the prediction model, and bug count as the response variable*, form the configuration combination of the most cost-effective bug predictor across all subject systems in the studied dataset.

The rest of the paper is organized as follows: We explain and motivate the experimental setup of our empirical study in section 2. We demonstrate the results and answer the posed research questions in section 3, then we discuss the threats to validity in section 4. We survey the related work and compare our findings with the literature in section 5. Finally, in section 6, we conclude this study with specific guidelines on building cost-effective bug predictors.

2 EMPIRICAL STUDY SETUP

2.1 Evaluation Scheme

There is a strong relationship between what is expected from a model and how the model is evaluated. In the field of bug prediction, the desired value expected from a bug predictor is to enhance the efficiency of the quality assurance procedure by directing it to the buggy parts of a software system. This is possible only when the bug predictor can find most of the bugs in the least amount of code. Intuitively, the efficiency of a predictor increases inverse-proportionally with the number of lines of code in which it suspects a bug might appear because writing unit tests for large software entities or inspecting them requires more effort.

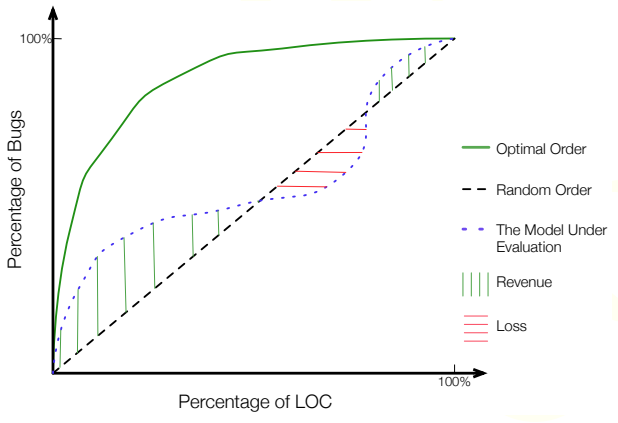


Figure 1: An overview of the *CE* measure as defined by Arisholm *et al.* [2].

Arisholm *et al.* state that *"... the regular confusion matrix criteria, although popular, are not clearly related to the problem at hand, namely the cost-effectiveness of using fault-proneness prediction models to focus verification efforts to deliver software with less faults at less cost"*[2]. Consequently, they proposed a cost-aware evaluation scheme called *Cost-Effectiveness* (*CE*) [2]. *CE* measures the benefit of using a certain bug prediction model. It summarizes the accuracy measures and the usefulness of a model by measuring how close the prediction model is to the optimal model, taking the random order as the baseline. This scheme assumes the ability of the prediction model to rank software entities in an ordered list. To demonstrate *CE*, we show in Figure 1 an example cumulative lift curves (Alberg diagrams [37]) of three orderings of software entities:

(1) **Optimal Order:** The green curve represents the ordering of the software entities with respect to the bug density from the highest to the lowest.
(2) **Random Order:** The dashed diagonal line is achieved when the percentage of bugs is equal to the percentage of lines of code. This is what one gets, on average, with randomly ordering the software entities.
(3) **Predicted Order:** The blue curve represents the ordering of the software entities based on the predicted dependent variable.

The area under each of these diagrams is called the Cost-Effectiveness (*CE*) area. The larger the *CE* area, the more cost-effective the model. However, two things need to be taken into account in this scenario. First, optimal models are different for different datasets. Second, the prediction model should perform better than the random ordering model to be considered valuable. That's why Arisholm *et al.* [2] took the optimal ordering and the random ordering into consideration in the Cost-Effectiveness measure as:

$$CE(model) = \frac{AUC(model) - AUC(random)}{AUC(optimal) - AUC(random)}$$

where $AUC(x)$ is the area under the curve x.

CE assesses how good the prediction model is in producing a total order of entities. The value of *CE* ranges from -1 to +1. The larger the *CE* measure is, the more cost-effective the model is. There are three cases:

(1) When *CE* is close to 0, it means that there is no gain in using the prediction model.
(2) When *CE* is close to 1, it means that the prediction model is close to optimal.
(3) When *CE* is between 0 and -1, it means that the cost of using the model is more than the gain, making the use of the model actually harmful.

In our experiments, we use *CE* to compare prediction results and draw conclusions.

2.2 Dataset

In our study, we use the "Bug Prediction Dataset" provided by D'Ambros *et al.* [11] in the form of publicly available software system metrics at the class level. The purpose of of this dataset is to provide a benchmark for researchers to run bug prediction experiments on. It contains source code metrics and change metrics of five popular Java systems (Table 1) and has been used previously

in many bug prediction studies. The dependent variable is the number of bugs on the Java class level.

Table 1: Details about the systems in the studied dataset, as reported by D'Ambros *et al.* [11]

System	KLOC	#Classes	% Buggy classes	% Buggy classes with more than one bug
Eclipse JDT Core	≈ 224	997	≈ 20%	≈ 33%
Eclipse PDE UI	≈ 40	1,497	≈ 14%	≈ 33%
Equinox	≈ 39	324	≈ 40%	≈ 38%
Mylyn	≈ 156	1,862	≈ 13%	≈ 28%
Lucene	≈ 146	691	≈ 9%	≈ 29%

2.3 Independent Variables

Source code metrics[3] are the metrics extracted from the source code itself. The most often-used metrics are the Chidamber and Kemerer (CK) metrics suite [10]. These more complex metrics are usually used in conjunction with simpler counting metrics like the number of lines of code (LOC), number of methods (NOM), or number of attributes (NOA). Source code metrics try to capture the quality (*e.g.*, LCOM, CBO) and complexity (*e.g.*, WMC, DIT) of the source code itself. The rationale behind using the source code metrics as bug predictors is that there should be a strong relation between source code features (quality and complexity) and software defects [47]. In other words, the more complex a software entity is, the more likely it contains bugs. Also the poorer the software design is, the more bug-prone it is.

Change metrics[4] are extracted from the software versioning systems like CVS, Subversion, and Git. They capture how and when software entities (binaries, modules, classes, methods) change and evolve over time. Change metrics describe software entities with respect to their age [44][5], past faults [27][55][41], past modifications and fixes [17][41][22][21][26][36], and developer information [53][54][43][33]. Using software history metrics as bug predictors is motivated by the following heuristics:

(1) Entities that change more frequently tend to have more bugs.
(2) Entities with a larger number of bugs in the past tend to have more bugs in the future.
(3) Entities that have been changed by new developers tend to have more bugs.
(4) Entities that have been changed by many developers tend to have more bugs.
(5) Bug-fixing activities tend to introduce new bugs.
(6) The older an entity, the more stable it is.

Many researchers argue that source code metrics are good predictors for future defects [23][42][3][8][48][7], but others show that change metrics are better than source code metrics at bug prediction [36][25] [17][26][54][41][54][16]. Moreover, Arisholm *et al.* states that models based on object-oriented metrics are no better than a model based on random class selection [2].

In this study, to take part in this debate, we consider source code metrics alone, change metrics alone, or both combined, and evaluate the cost-effectiveness of the resulted bug predictors.

[3]Source code metrics are also known as product metrics.
[4]Change metrics are also known as process metrics or history metrics

2.4 Response Variable

The chosen dependent variable is an important design decision because it sometimes determines the difference between a usable and an unusable model. We have to keep in mind that the final goal of the prediction is to prioritize the software entities into an ordered list to be able to apply the Cost-Effectiveness (*CE*) measure. There are multiple possible schemes to do it:

(1) predict the number of bugs as the response variable then order the software entities based on the *calculated* bug density.
(2) predict the bug density directly then order the software entities based on this *predicted* bug density.
(3) predict the bug proneness and order the software entities based on it. Small classes come before large ones in case of ties.
(4) classify software entities into buggy and bug-free, then order them as follows: buggy entities come before bug-free ones and small classes come before large ones (with respect to LOC).

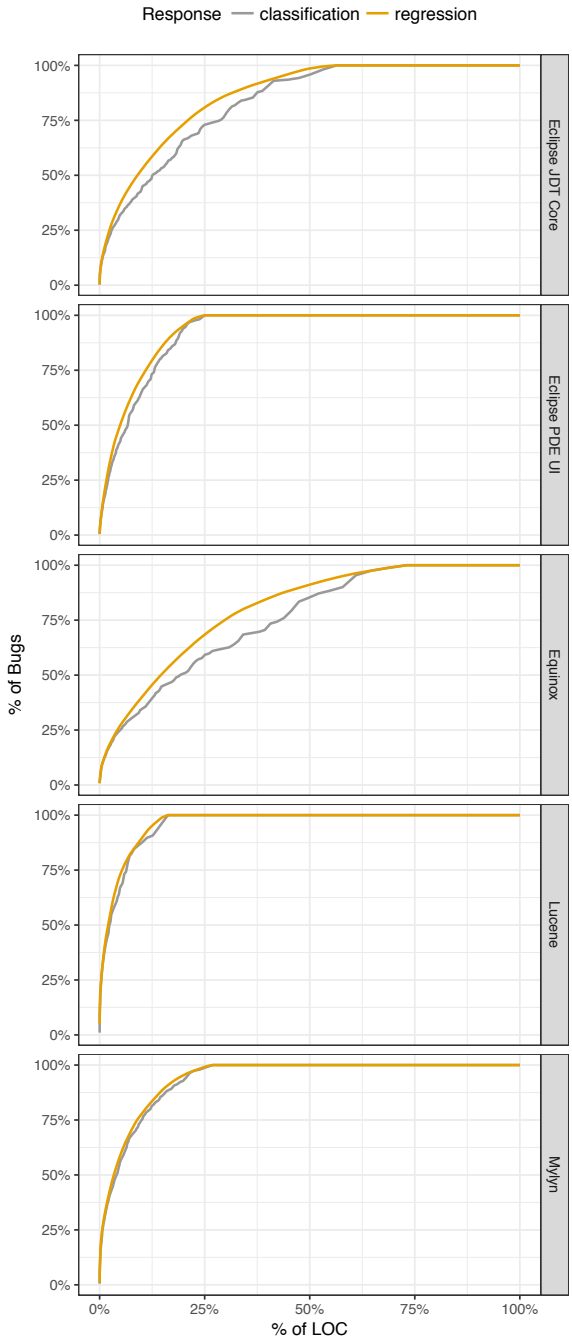
The optimal prediction of number of bugs (first scheme) is equivalent to the optimal prediction of bug density (second scheme), since bug density is calculated from the number of bugs as $bugdensity = (\#bugs/LOC)$. Also the optimal prediction of bug proneness (third scheme) is an optimal classification (fourth scheme). Obviously the optimal regression in the first or second schemes is more cost-effective than the optimal classification in the third or fourth scheme because it reflects exactly the optimal solution in the cost-effectiveness (*CE*) evaluation method. However, we need to verify whether classification is a valid approach in bug prediction and whether we should include the third and fourth schemes in the empirical study. In Figure 2, we evaluate the cost-effectiveness of the optimal classifier following the fourth scheme, for all five systems in our corpus. The cost-effectiveness of the fourth scheme is excellent for Equinox and JDT, and almost optimal for Mylyn, Lucene, and PDE. As shown in Table 1, the percentage of buggy classes with more than one bug ranges from 28% to 38%. This leads to the conclusion that in the set of buggy classes, the number of bugs is proportional to the number of lines of code. This is particularly interesting because it makes classification as good as predicting the number of bugs, in the ideal case. We empirically verify which response variable is better when we train our prediction models.

2.5 Machine Learning Models

We investigate the following machine learning models: Random Forest (RF), Support Vector Machine (SVM), Multilayer Perceptron (MLP), an implementation of the K-nearest neighbours algorithm called IBK, and Linear Regression (LinR) / Logistic Regression (LogR)[5]. We choose these machine learning models for two reasons: First, they are extensively used in the bug prediction literature [29]. Second, each one of them can be used as a regressor and as a classifier, making comparisons across different configurations possible. Classifiers are used to predict the bug proneness or the class (buggy, bug-free) and regressors are used to predict the bug count and

[5]Linear Regression and Logistic Regression are equivalent, but with different types of response variables. Linear Regression is a regressor and Logistic Regression is a classifier.

Figure 2: Commutative lift curves (Alberg diagrams [37]) comparing the optimal regressor (bug density predictor) and the optimal classifier with ranking based on LOC (smallest to largest). These diagrams show that optimal classification performs almost as well as optimal regression.



bug density. We use the Weka data mining tool [19] to build these prediction models.

2.6 Hyperparameter Optimization

Machine learning models may have configurable parameters that should be set before starting the training phase. This process is called hyperparameter optimization or model tuning, and can have a positive effect on the prediction accuracy of the models. However, different models have different sensitivities to this process. While model tuning improves IBK and MLP substantially, it has a negligible effect on SVM and RF [49][39]. In this study, we follow the same procedure proposed by Osman *et al.* for hyperparameter optimization [39]. The used model parameters are detailed in Table 2.

Table 2: The tuning results for the hyperparameters

RF	Number of Trees= 100
SVM	Kernel= RBF {Gamma=0.1} Complexity=10
MLP	Learning Rate=0.6 Momentum=0.6 Hidden Layer Size= 32
IBK	#Neighbours=5 Search Algorithm= Linear Search Evaluation Criterion=Mean Squared Error
LinR/LogR	No parameters to tune

2.7 Feature Selection

The prediction accuracy of machine learning models is highly affected by the quality of the features used for prediction. Irrelevant and correlated features can increase prediction error, increase model complexity, and decrease model stability. Feature selection is a method that identifies the relevant features to feed into machine learning models. We apply wrapper feature selection for SVM, MLP, and IBK as it has been shown that it leads to higher prediction accuracy [40]. We do not apply feature selection for RF because it performs feature selection internally. Following the guidelines by Osman *et al.* [38], we apply *l*2 regularization (Ridge) on LinR/LogR as the feature selection method.

2.8 Data Pre-Processing

Bug datasets are inherently imbalanced where most software entities are bug-free. This is called the class-imbalance problem and can negatively affect the performance of machine learning models [52][1]. To cope with this problem, we divide the data set for each project into two sets: test set (25%) and training set (75%). The samples in each set are taken at random but maintain the distribution of buggy classes similar to the one in the full data set. We then balance the training set by oversampling. This is important for training and for evaluating the prediction models. The machine learning models are then trained using the balanced training set and evaluated on the unseen test set.

3 RESULTS

In this study, we consider the following configurations:

(1) Independent Variables:
  (a) source code metrics (`src`)
  (b) change metrics (`chg`)
  (c) both of them combined (`both`).
(2) Machine Learning Model:
  (a) Support Vector Machines (`SVM`)
  (b) Random Forest (`RF`)
  (c) Multilayer Perceptron (`MLP`)
  (d) K-Nearest Neighbours (`IBK`)
  (e) Linear Regression (`LinR`) or Logistic Regression (`LogR`)
(3) Response Variable:
  (a) bug count (`cnt`)
  (b) bug density (`dns`)
  (c) bug proneness (`prs`)
  (d) classification (`cls`)

There are 60 possible configuration combinations. For each one, we pre-process the data, train the model on the training set, perform the predictions on the test data, and calculate the Cost Effectiveness measure (*CE*). We repeat this process 50 times to mitigate the threat of having outliers because of the random division of the dataset into a training set and a test set. We do not perform k-fold cross-validation method because calculating *CE* over a small set of classes can be misleading. Instead, we perform the repeated hold-out validation because it is known to have lower variance than k-fold cross validation making it more suitable for small datasets [4].

In this experiment, statistically speaking, the treatment is the configuration combination and the outcome is the *CE* score. Hence, we have 60 different treatments and one outcome measure. To answer the posed research questions, we need to compare the *CE* of different configuration combinations. Since there is a large number of treatments, traditional parametric tests (*e.g.*, ANOVA) or non-parametric tests (*e.g.*, Friedman) have the overlapping problem, *i.e.*, the clusters of treatment overlap. Therefore, we use the Scott-Knott (SK) cluster analysis for grouping of means [45], which is a clustering algorithm used as a multiple comparison method for the analysis of variance. SK clusters the treatments into statistically distinct non-overlapping groups (*i.e.*, ranks), which makes it suitable for this study. We apply SK with 95% confidence interval to cluster the configuration combinations for each project in the dataset.

Figure 3 shows box plots of the *CE* outcomes for each configuration combination. Each box plot represents the population of the 100 runs of the corresponding configuration. The box plots are sorted in an increasing order of the means of *CE*, represented by the red points. Alternating background colors indicate the Scott-Knott statistically distinct groups (*i.e.*, clusters or ranks).

The results in Figure 3 clearly demonstrate the interplay between the design choices in bug prediction. Changing one value in the configuration can transform a bug predictor from being highly cost-effective, to being actually harmful. For instance, while `both-RF-cnt` is the most cost-effective configuration in Figure 3(a), `both-RF-dns` is in the least cost-effective cluster. This means that although `RF` is the best machine learning model and `both` is the best choice of metrics, using them with the wrong response variable renders a bug predictor useless. There are many examples where changing one configuration parameter brings the bug predictor from one cluster to another. Examples for each configuration variable include:

(1) In Figure 3(a), `both-RF-cnt` is ranked 1^{st} whereas `both-RF-dns` is ranked 7^{th}.
(2) In Figure 3(b), `both-SVM-cnt` is ranked 2^{nd} whereas `both-MLP-cnt` is ranked 6^{th}.
(3) In Figure 3(e), `chg-SVM-cls` is ranked 2^{nd} where as `src-SVM-cls` is ranked 6^{th}.

These examples constitute a compelling evidence that bug prediction configurations are interconnected.

To answer the first research question (*RQ1*) regarding the choice of independent variables, we analyze the top cluster of configurations in Figure 3. We observe that for Eclipse PDE UI, Equinox, and Mylyn, the software metrics value in the top rank is either `both` or `chg`. In Eclipse JDT Core, `src` appears in one configuration out of three in the top rank. In Lucene, *src* appears in one configuration out of 27 in the top rank. These results suggest that the use of both source code and change metrics together is the most cost-effective option for the independent variables. Using only change metrics is also a good choice, but using only source code metrics rarely is. It was shown in the literature that adding source code metrics to change metrics hinders the cost-effectiveness and using source code alone is not better than random guessing [2]. Our results show that although less cost-effective, source code metrics can be used alone when necessary (*e.g.*, change metrics cannot be computed). There is always a cost-effective configuration combination with the source code metrics as the independent variables (*e.g.*, `src-RF-cnt`).

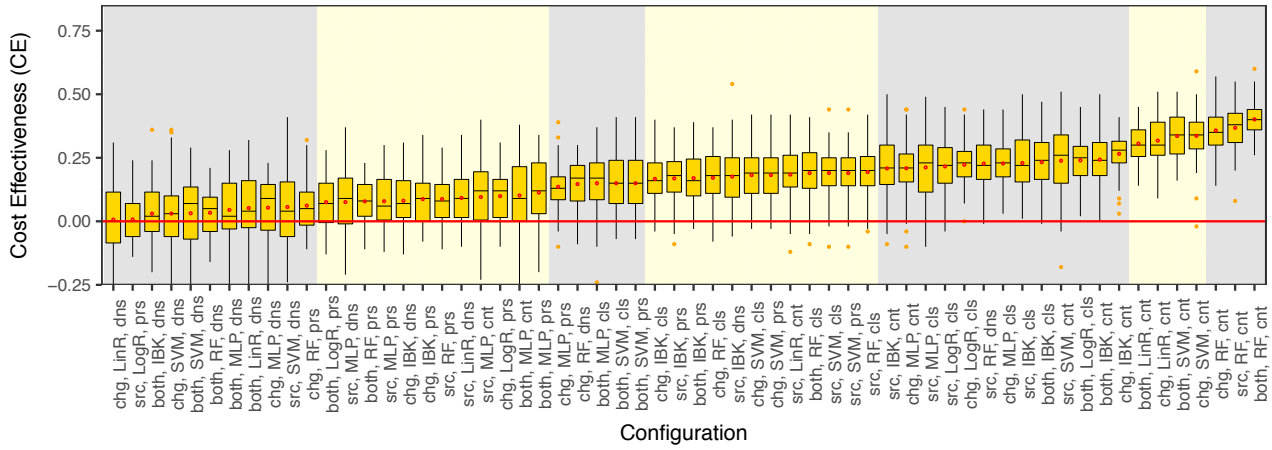
For the second research question (*RQ2*) regarding the choice of the machine learning model, we observe that `RF` is *the only* option in the top rank in Eclipse JDT Core, Eclipse PDE UI, and Equinox, and it is in the top rank of Lucene and Mylyn. `SVM` also performs well. It appears in the top rank in Lucene and Mylyn, and in the second rank in the rest of the projects. These results indicate the superiority of Random Forest and Support Vector Machines in producing cost-effective bug predictors. On the other hand, `MLP` and `IBK` made it to the top two clusters only in Lucene, suggesting that Multilayer Perceptron and K-Nearest Neighbour do not fit the bug prediction problem well.

For the third research question (*RQ3*) regarding the most cost-effective response variable, we observe that `cnt` is in the top rank in Lucene and is *the only* response variable in the top rank in the other projects. It is clear that predicting the bug count results in the most cost-effective bug predictors. Another observation is that the response variable configuration in the bottom two clusters is almost conclusively either `dns` or `prs`. This means that predicting bug density or bug proneness actually hinders the cost-effectiveness of the bug prediction.

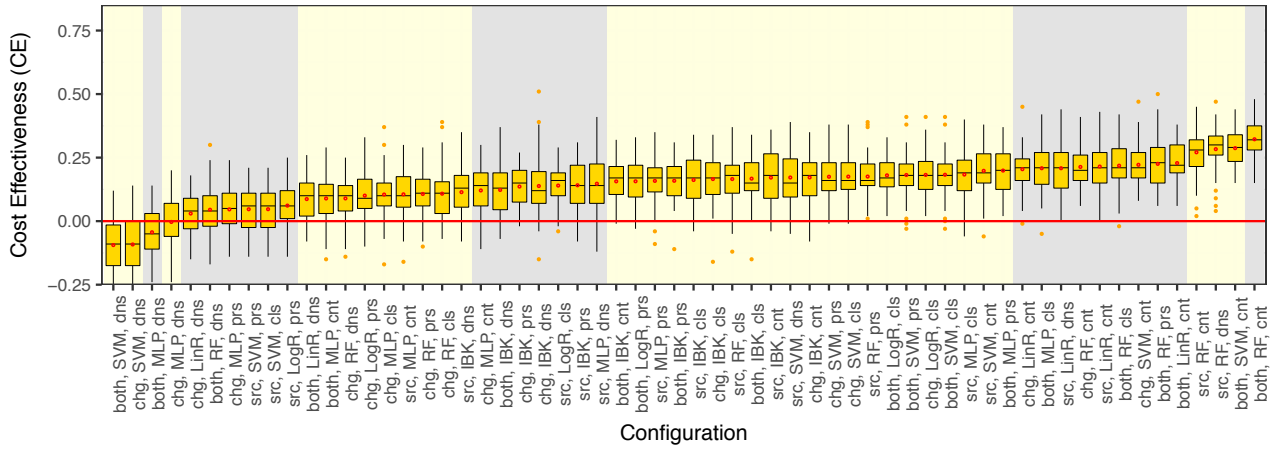
Overall, one result that stands out is that the configuration `both-RF-cnt` is in the top cluster across projects (*RQ4*). In fact, it is *the most* cost-effective configuration in Eclipse JDT Core, Eclipse PDE UI, and Equinox and it is in the top cluster in Lucene and Mylyn. This finding suggests that this configuration seems to be the best from the cost-effectiveness point of view.

Software projects differ in their domains, development methods, used frameworks, and developer experiences. Consequently,

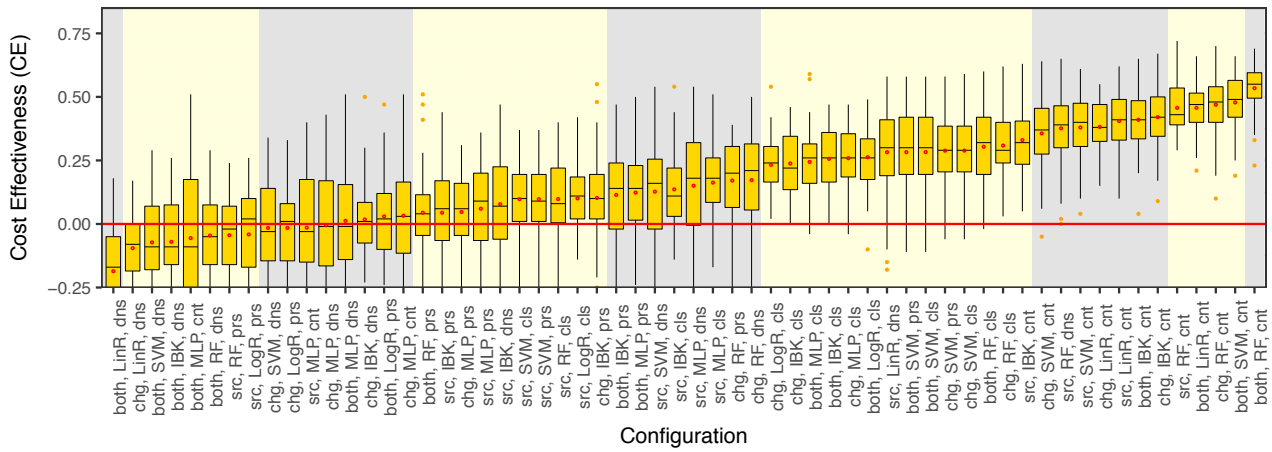
Figure 3: Boxplots of the *CE* outcome. Each box plot represents the *CE* obtained by 100 runs of the corresponding configuration combination on the x-axis. Different background colors indicate the statistically distinct groups obtained by applying the Scott-Knott clustering method with 95% confidence interval. The configurations on the x-axis are of the form `Metrics-Model-Response`.



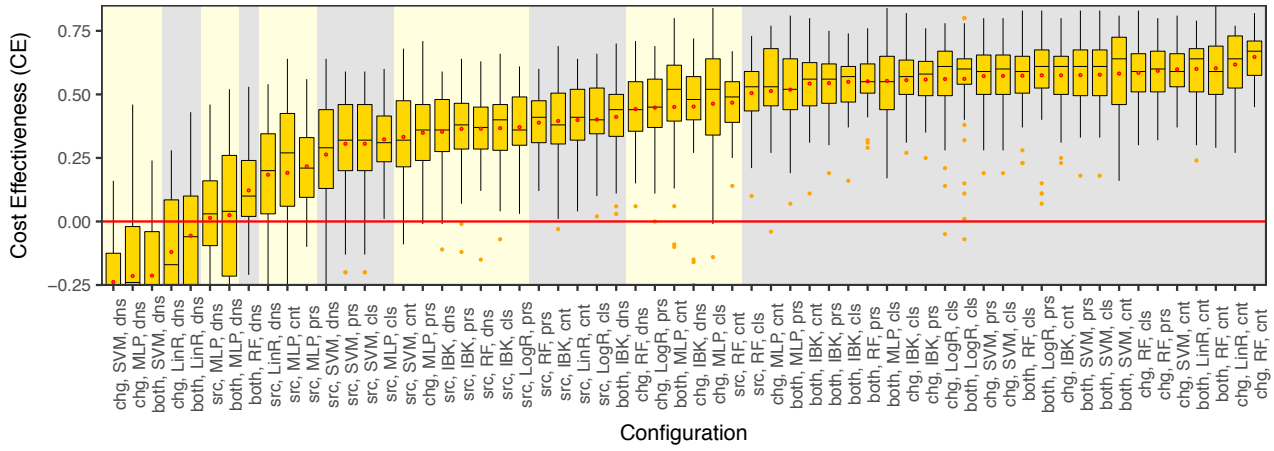
(a) Eclipse JDT Core



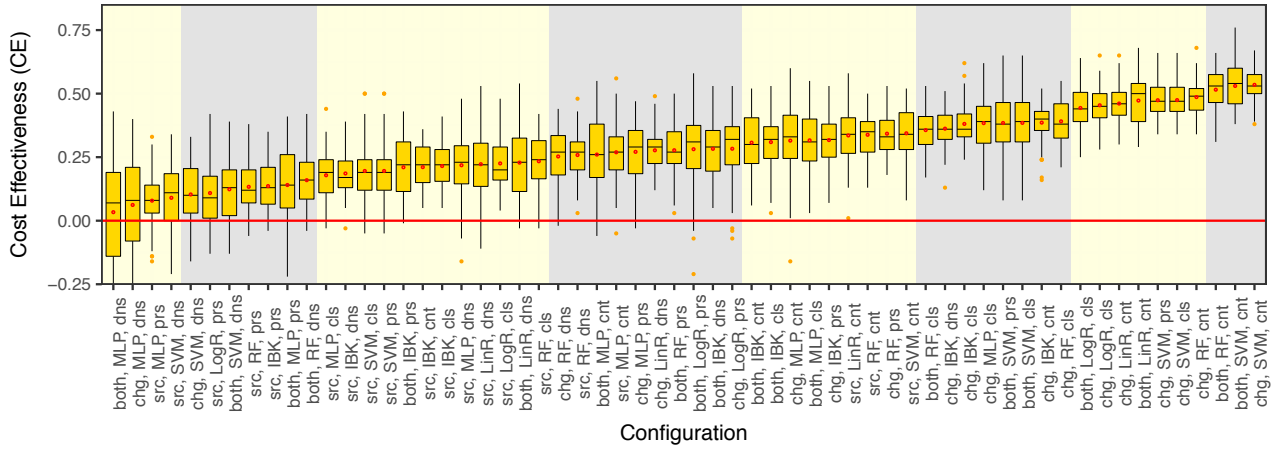
(b) Eclipse PDE UI



(c) Equinox

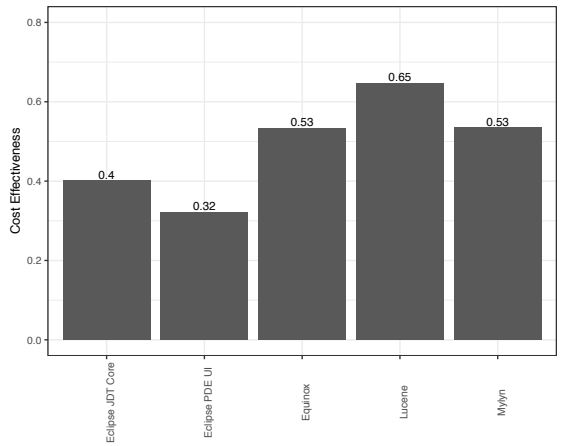


(d) Lucene



(e) Mylyn

Figure 4: The max mean values of *CE* obtained for each project



software metrics differ in the correlation with the number of bugs among projects. This is the reason why using both metrics came out as the best choice of independent variable. However, to deal with the inevitable noise and redundancy in using both metrics, the best configurations includes Random Forest as the machine learning model. Random Forest is an ensemble of decision trees created by using bootstrap samples of the training data and random feature selection in tree induction [6]. This gives RF the ability to work well with high-dimensional data and sift the noise away. This is the reason why feeding `both` types of metrics into Random Forest actually makes sense. Also bug count came out as the best option for response variable because it reflects the "gain" in *CE* better than classification or proneness. Bug density also reflects the "gain" but it is better to calculate it from bug count than to leave it to the prediction model to deduce. Therefore, bug count is a simpler and more appropriate response variable than bug density. All these factors contribute to the fact that `both-RF-cnt` is the most cost-effective configuration for bug prediction.

Finally, the results in Figure 4 also show that the cost-effectiveness of the best bug predictor varies among projects. Although the best bug predictor is never harmful to use (no negative *CE*) in our experiments, it can still be of little value for some projects, *e.g.*, $CE = 0.32$ for Eclipse PDE UI. This means that bug prediction should be evaluated as a technique in the context of a software project before putting it in use in that specific project.

4 THREATS TO VALIDITY

To minimize the threats to validity in our empirical study, we follow the guidelines of Mende [30] by

- using a large dataset to avoid large variance of performance measures,
- maintaining the same fault distribution in the test set and training set as the original set, to minimize bias,
- repeating the experiment 50 times to minimize the effect of outliers,
- and reporting on the dataset, data preprocessing procedure, and model configurations to enable replication.

In our study we use the “Bug Prediction Dataset" provided by D’Ambros *et al.* [11] as a benchmark. Although it is a well-known and studied dataset, the quality of our results is very dependent on the quality of that dataset.

Our dependence on WEKA [19] for building the artificial intelligence models, makes the quality of the models dependent solely on the quality of WEKA itself.

The fact that this dataset contains metrics only from open source software systems makes it hard to generalize to all Java systems. In the future, we plan to apply our study on more datasets and to use other data mining tools.

Another threat to validity comes from the use of LOC as a proxy for cost. As we explained before, reviewing code and writing unit tests take much more effort for large modules than small ones. However, we are aware of the fact that this proxy might introduce some bias. We use it because it has been used in several previous studies as such (*e.g.*, [31][2]) and it is widely accepted in the community as a good measure of effort.

Our study is on the Java-class level. Hence, our findings may not apply on other granularity levels such as method level or commit level.

Finally, in this study, we assume that the purpose of bug prediction is to locate the maximum number of bugs in the minimum amount of code in order to be a useful support to quality assurance activities. However, defect prediction models can be used for other purposes. For example, they can be used as tools for understanding common pitfalls and analyzing factors that affect the quality of software. In these cases, our findings do not necessarily apply. In the future, we plan to extend this study to the broader context of several defect prediction use cases.

5 RELATED WORK

Most studies comparing different machine learning models in bug prediction show no difference in performance [12][28] [51][35]. Menzies *et al.* [35] evaluate many models using the area under the curve of a probability of false alarm versus probability of detection “AUC(pd, pf)". They conclude that better prediction models do not yield better results. Similarly, Vandecruys *et al.* [51] compare the accuracy, specificity (true negative rate), and sensitivity (recall or true positive rate) between seven classifiers. Using the non-parametric Friedman test, as recommended in this type of problem [12], it is shown that there is no statistically significant difference at the 5% significance level. Lessmann *et al.* [28] study 22 classifiers and conclude that the classification model is less important than generally assumed, giving researchers more freedom in choosing models when building bug predictors. Actually simple models like naïve Bayes or C4.5 classifiers perform as well as more complicated models [13][34]. Other studies suggest that there are certain models which perform better than others in predicting bugs. Elish and Elish [14] compare SVM against 8 machine learning and statistical models and show that SVM performs classification generally better. Guo *et al.* [18] compare Random Forest with other classifiers and show how it generally achieves better prediction. Ghotra *et al.* [15] show that there are four statistically distinct groups of classification techniques suggesting that the choice of the classification model has a significant impact on the performance of bug prediction. Our findings confirm the superiority of certain models over others. Specifically, we show that Random Forest is indeed the best machine learning model, followed by Support Vector Machines.

Mende and Koschke [32] studied the concept of effort-aware defect prediction. They compared models with predicting defect density using only Random Forest trained only on source code metrics. They took the effort into account during the training phase by predicting bug density as a response variable. Although we agree with Mende and Koschke on the importance of building effort-aware prediction models, our results actually advise against using source code metrics and bug density as independent and dependent variables respectively. Canfora *et al.* also consider cost in the training phase [9]. Using genetic algorithms, they trained a multi-objective logistic regressor that, based on developer preferences, is either highly effective, with low cost, or balanced. They treated bug prediction as a classification problem and they used only source code metrics as independent variables. We agree with Canfora *et al.* that bug predictors should be tuned to be cost-effective, but our study shows that source code metrics are rarely a good choice of independent variables, and predicting bug count is more cost-effective than predicting bug proneness.

Kamei *et al.* [25] also evaluated the predictive power of history and source code metrics in an effort-sensitive manner. They used regression model, regression tree, and Random Forest as models. They found that history metrics significantly outperform source code metrics with respect to predictive power when effort is taken into consideration. Our results confirm their findings but also add that the use of both metrics is even more cost-effective.

Arisholm *et al.* [2] studied several prediction models using history and source code metrics in bug prediction. They also dealt with the class imbalance problem and performed effort-aware evaluation. They found that I) history metrics perform better than source code metrics and II) source code metrics are no better than random class selection. Our results confirm their first finding but contradict the second. We show that indeed using source code metrics is less cost-effective than using change metrics or a mix of both. However, using source code metrics with the right options of other

configurations is certainly better than random class selection. This contradiction with our findings comes from the fact that the dependent variable in their study is bug proneness and, as opposed to our study, they did not consider other dependent variables.

Jiang *et al.* [24] surveyed many evaluation techniques for bug prediction models and concluded that the system cost characteristics should be taken into account to find the best model for that system. The cost Jiang *et al.* considered is the cost of misclassification because they dealt with the bug prediction problem as a classification problem. In our study, we treat bug prediction as a regression problem (*i.e.*, bug count and bug density) and as a classification problem (bug proneness and entity class) and the cost of the model is the actual cost of maintenance using LOC as a proxy.

Finally, there are many other studies that look into the effect of experimental setup on bug prediction studies. Tantithamthavorn *et al.* show how different evaluation schemes can lead to different outcome in bug prediction research [50]. In this study we focus on configuring bug predictors to be cost-effective. Thus, we fix the evaluation scheme to the one the reflects our purpose (*i.e.*, *CE*). In a systematic literature review, Hall *et al.* [20] define some criteria that makes a bug prediction paper and its results reproducible. Surprisingly, out of 208 surveyed papers, only 36 were selected. In this paper we adhere to these criteria by extensively describing our empirical setup. Shepperd *et al.* [46] raised concerns about the conflicting results in previous bug prediction studies. They surveyed 42 primary studies. They found out that the choice of the classification technique, the metric family, and the data set have small impact on the variability of the results. The variability comes mainly from the research group conducting the study. Shepperd *et al.* conclude that who is doing the work matters more than what is done, meaning that there is a high level of researcher bias. We agree with the authors that there are many factors that might affect the outcome of a bug prediction study. In fact this is the main motivation behind our study. However, Shepperd *et al.* looked at studies that treat bug prediction as a classification problem, ignoring the fact that the response variable is itself a factor that affects the outcome. In our study we include more factors and emphasize on the interplay among them.

6 CONCLUSIONS & FUTURE WORK

Bug prediction is used to reduce the costs of testing and code reviewing by directing maintenance efforts towards the software entities that most likely contain bugs. From this point of view, a successful bug predictor finds the largest number of bugs in the least amount of code. Using the cost effectiveness evaluation scheme, we carry out a large-scale empirical study to find the most efficient bug prediction configurations, as building a bug predictor entails many design decisions, each of which has many options. We summarize the findings of this study as follows:

(1) Using a mix of source code and change metrics is the most cost-effective option for the independent variables. Change metrics alone is a good option.
(2) Random Forest results is the best machine learning model, followed by Support Vector Machines.
(3) Bug count is the most cost-effective option for the response variable. Bug density and bug proneness are the least cost-effective and should be avoided.
(4) The combination of the above configurations results in the most cost-effective bug predictor in all systems in the used dataset.

Finally, our findings reveal a compelling evidence that bug prediction configurations are interconnected. Changing one configuration can render a bug predictor useless. Hence, we advise that future bug prediction studies take this factor into account.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the financial support of the Swiss National Science Foundation for the project “Agile Software Analysis” (SNSF project No. 200020-162352, Jan 1, 2016 - Dec. 30, 2018).

REFERENCES

[1] Harald Altinger, Steffen Herbold, Friederike Schneemann, Jens Grabowski, and Franz Wotawa. 2017. Performance Tuning for Automotive Software Fault Prediction. In *2017 IEEE 24th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*.
[2] Erik Arisholm, Lionel C. Briand, and Eivind B. Johannessen. 2010. A Systematic and Comprehensive Investigation of Methods to Build and Evaluate Fault Prediction Models. *J. Syst. Softw.* 83, 1 (Jan. 2010), 2–17. https://doi.org/10.1016/j.jss.2009.06.055
[3] V.R. Basili, L.C. Briand, and W.L. Melo. 1996. A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on* 22, 10 (Oct. 1996), 751–761. https://doi.org/10.1109/32.544352
[4] Claudia Beleites, Richard Baumgartner, Christopher Bowman, Ray Somorjai, Gerald Steiner, Reiner Salzer, and Michael G Sowa. 2005. Variance reduction in estimating classification error using sparse datasets. *Chemometrics and intelligent laboratory systems* 79, 1 (2005), 91–100.
[5] Abraham Bernstein, Jayalath Ekanayake, and Martin Pinzger. 2007. Improving defect prediction using temporal features and non linear models. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting (IWPSE '07)*. ACM, New York, NY, USA, 11–18. https://doi.org/10.1145/1294948.1294953
[6] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.
[7] Lionel C. Briand, Jürgen Wüst, John W. Daly, and D. Victor Porter. 2000. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software* 51, 3 (2000), 245–273. https://doi.org/10.1016/S0164-1212(99)00102-8
[8] Lionel C. Briand, Jürgen Wüst, Stefan V. Ikonomovski, and Hakim Lounis. 1999. Investigating Quality Factors in Object-oriented Designs: An Industrial Case Study. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*. ACM, New York, NY, USA, 345–354. https://doi.org/10.1145/302405.302654
[9] G. Canfora, A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella. 2013. Multi-objective Cross-Project Defect Prediction. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. 252–261. https://doi.org/10.1109/ICST.2013.38
[10] Shyam R. Chidamber and Chris F. Kemerer. 1991. Towards a metrics suite for object oriented design. In *Conference proceedings on Object-oriented programming systems, languages, and applications (OOPSLA '91)*. ACM, New York, NY, USA, 197–211. https://doi.org/10.1145/117954.117970
[11] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2010. An Extensive Comparison of Bug Prediction Approaches. In *Proceedings of MSR 2010 (7th IEEE Working Conference on Mining Software Repositories)*. IEEE CS Press, 31–40.
[12] Janez Demšar. 2006. Statistical comparisons of classifiers over multiple data sets. *The Journal of Machine Learning Research* 7 (2006), 1–30.
[13] Pedro Domingos and Michael Pazzani. 1997. On the Optimality of the Simple Bayesian Classifier Under Zero-One Loss. *Mach. Learn.* 29, 2-3 (Nov. 1997), 103–130. https://doi.org/10.1023/A:1007413511361
[14] Karim O Elish and Mahmoud O Elish. 2008. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software* 81, 5 (2008), 649–660.
[15] Baljinder Ghotra, Shane McIntosh, and Ahmed E Hassan. 2015. Revisiting the impact of classification techniques on the performance of defect prediction models.

In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 789–800.

[16] Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C Gall. 2012. Method-level bug prediction. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 171–180.

[17] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. 2000. Predicting Fault Incidence Using Software Change History. *IEEE Transactions on Software Engineering* 26, 2 (2000).

[18] Lan Guo, Yan Ma, Bojan Cukic, and Harshinder Singh. 2004. Robust prediction of fault-proneness by random forests. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*. IEEE, 417–428.

[19] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. 2009. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter* 11, 1 (2009), 10–18.

[20] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. 2012. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering* 38, 6 (2012), 1276–1304.

[21] Ahmed E. Hassan and Richard C. Holt. 2005. The Top Ten List: Dynamic Fault Prediction. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*. IEEE Computer Society, Washington, DC, USA, 263–272. https://doi.org/10.1109/ICSM.2005.91

[22] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. 2012. Bug Prediction Based on Fine-grained Module Histories. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 200–210. http://dl.acm.org/citation.cfm?id=2337223.2337247

[23] Steffen Herbold. 2013. Training Data Selection for Cross-project Defect Prediction. In *Proceedings of the 9th International Conference on Predictive Models in Software Engineering (PROMISE '13)*. ACM, New York, NY, USA, Article 6, 10 pages. https://doi.org/10.1145/2499393.2499395

[24] Yue Jiang, Bojan Cukic, and Yan Ma. 2008. Techniques for Evaluating Fault Prediction Models. *Empirical Softw. Engg.* 13, 5 (Oct. 2008), 561–595. https://doi.org/10.1007/s10664-008-9079-3

[25] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A.E. Hassan. 2010. Revisiting common bug prediction findings using effort-aware models. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*. 1–10. https://doi.org/10.1109/ICSM.2010.5609530

[26] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. 2007. Predicting Faults from Cached History. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, Washington, DC, USA, 489–498. https://doi.org/10.1109/ICSE.2007.66

[27] M. Klas, F. Elberzhager, J. Munch, K. Hartjes, and O. von Graevemeyer. 2010. Transparent combination of expert and measurement data for defect prediction: an industrial case study. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, Vol. 2. 119–128. https://doi.org/10.1145/1810295.1810313

[28] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. 2008. Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. *IEEE Trans. Softw. Eng.* 34, 4 (July 2008), 485–496. https://doi.org/10.1109/TSE.2008.35

[29] Ruchika Malhotra. 2015. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing* 27 (2015), 504–518.

[30] Thilo Mende. 2010. Replication of defect prediction studies: problems, pitfalls and recommendations. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. ACM, 5.

[31] Thilo Mende and Rainer Koschke. 2009. Revisiting the Evaluation of Defect Prediction Models. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering (PROMISE '09)*. ACM, New York, NY, USA, Article 7, 10 pages. https://doi.org/10.1145/1540438.1540448

[32] Thilo Mende and Rainer Koschke. 2010. Effort-Aware Defect Prediction Models. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering (CSMR '10)*. IEEE Computer Society, Washington, DC, USA, 107–116. https://doi.org/10.1109/CSMR.2010.18

[33] Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. 2008. Predicting Failures with Developer Networks and Social Network Analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*. ACM, New York, NY, USA, 13–23. https://doi.org/10.1145/1453101.1453106

[34] T. Menzies, J. Greenwald, and A. Frank. 2007. Data Mining Static Code Attributes to Learn Defect Predictors. *Software Engineering, IEEE Transactions on* 33, 1 (Jan. 2007), 2–13. https://doi.org/10.1109/TSE.2007.256941

[35] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. 2010. Defect Prediction from Static Code Features: Current Results, Limitations, New Approaches. *Automated Software Engg.* 17, 4 (Dec. 2010), 375–407. https://doi.org/10.1007/s10515-010-0069-5

[36] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. 2008. A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 181–190. https://doi.org/10.1145/1368088.1368114

[37] Niclas Ohlsson and Hans Alberg. 1996. Predicting Fault-Prone Software Modules in Telephone Switches. *IEEE Trans. Softw. Eng.* 22, 12 (Dec. 1996), 886–894. https://doi.org/10.1109/32.553637

[38] Haidar Osman, Mohammad Ghafari, and Oscar Nierstrasz. 2017. Automatic Feature Selection by Regularization to Improve Bug Prediction Accuracy. In *1st International Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE 2017)*. 27–32. https://doi.org/10.1109/MALTESQUE.2017.7882013

[39] Haidar Osman, Mohammad Ghafari, and Oscar Nierstrasz. 2017. Hyperparameter Optimization to Improve Bug Prediction Accuracy. In *1st International Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE 2017)*. 33–38. https://doi.org/10.1109/MALTESQUE.2017.7882014

[40] Haidar Osman, Mohammad Ghafari, and Oscar Nierstrasz. 2017. The Impact of Feature Selection on Predicting the Number of Bugs. *Information and Software Technology* (2017), in review.

[41] T.J. Ostrand, E.J. Weyuker, and R.M. Bell. 2005. Predicting the location and number of faults in large software systems. *Software Engineering, IEEE Transactions on* 31, 4 (April 2005), 340–355. https://doi.org/10.1109/TSE.2005.49

[42] A Panichella, R. Oliveto, and A De Lucia. 2014. Cross-project defect prediction models: L'Union fait la force. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*. 164–173. https://doi.org/10.1109/CSMR-WCRE.2014.6747166

[43] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. 2008. Can Developer-module Networks Predict Failures?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*. ACM, New York, NY, USA, 2–12. https://doi.org/10.1145/1453101.1453105

[44] Joseph R. Ruthruff, John Penix, J. David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. 2008. Predicting Accurate and Actionable Static Analysis Warnings: An Experimental Approach. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 341–350. https://doi.org/10.1145/1368088.1368135

[45] ANDREW JHON Scott and M Knott. 1974. A cluster analysis method for grouping means in the analysis of variance. *Biometrics* (1974), 507–512.

[46] Martin Shepperd, David Bowes, and Tracy Hall. 2014. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering* 40, 6 (2014), 603–616.

[47] R. Subramanyam and M.S. Krishnan. 2003. Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects. *Software Engineering, IEEE Transactions on* 29, 4 (April 2003), 297–310. https://doi.org/10.1109/TSE.2003.1191795

[48] Mei-Huei Tang, Ming-Hung Kao, and Mei-Hwa Chen. 1999. An empirical study on object-oriented metrics. In *Software Metrics Symposium, 1999. Proceedings. Sixth International*. 242–249. https://doi.org/10.1109/METRIC.1999.809745

[49] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. 2016. Automated Parameter Optimization of Classification Techniques for Defect Prediction Models. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 321–332. https://doi.org/10.1145/2884781.2884857

[50] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2017. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering* 43, 1 (2017), 1–18.

[51] Olivier Vandecruys, David Martens, Bart Baesens, Christophe Mues, Manu De Backer, and Raf Haesen. 2008. Mining software repositories for comprehensible software fault prediction models. *Journal of Systems and software* 81, 5 (2008), 823–839.

[52] Shuo Wang and Xin Yao. 2013. Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability* 62, 2 (2013), 434–443.

[53] Elaine J. Weyuker, Thomas J. Ostrand, and Robert M. Bell. 2007. Using Developer Information As a Factor for Fault Prediction. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering (PROMISE '07)*. IEEE Computer Society, Washington, DC, USA, 8–. https://doi.org/10.1109/PROMISE.2007.14

[54] Elaine J. Weyuker, Thomas J. Ostrand, and Robert M. Bell. 2008. Do Too Many Cooks Spoil the Broth? Using the Number of Developers to Enhance Defect Prediction Models. *Empirical Softw. Engg.* 13, 5 (Oct. 2008), 539–559. https://doi.org/10.1007/s10664-008-9082-8

[55] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. 2009. Cross-project Defect Prediction: A Large Scale Experiment on Data vs. Domain vs. Process. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '09)*. ACM, New York, NY, USA, 91–100. https://doi.org/10.1145/1595696.1595713

bib file

```
@inproceedings{PROMISE2017-1-Hemmati,
 author = { Hemmati, Hadi and  Noor, Tanzeem Bin},
 title = {Studying Test Case Failure Prediction for Test Case Prioritization},
 booktitle = {Proceedings of the 13th International Conference on Predictive Modelling in S
 series = {PROMISE '2017},
 year = {2017},
 isbn = {978-1-4503-5305-2},
 location = {Toronto, Canada},
 articleno = {1},
 url = {http://doi.acm.org/10.1145/3127005.3127006},
 doi = {10.1145/3127005.3127006},
 publisher = {ACM},
 address = {New York, NY, USA}
}
@inproceedings{PROMISE2017-2-Minku,
 author = { Minku, Leandro},
 title = {Clustering Dycom: An Online Cross-Company Software Effort Estimation Study},
 booktitle = {Proceedings of the 13th International Conference on Predictive Modelling in S
 series = {PROMISE '2017},
 year = {2017},
 isbn = {978-1-4503-5305-2},
 location = {Toronto, Canada},
 articleno = {2},
 url = {http://doi.acm.org/10.1145/3127005.3127007},
 doi = {10.1145/3127005.3127007},
 publisher = {ACM},
 address = {New York, NY, USA}
}
@inproceedings{PROMISE2017-3-Coppola,
 author = { Coppola, Riccardo},
 title = {Scripted UI Testing of Android Apps: A Study on Diffusion, Evolution and Fragilit
 booktitle = {Proceedings of the 13th International Conference on Predictive Modelling in S
 series = {PROMISE '2017},
 year = {2017},
 isbn = {978-1-4503-5305-2},
 location = {Toronto, Canada},
 articleno = {3},
 url = {http://doi.acm.org/10.1145/3127005.3127008},
 doi = {10.1145/3127005.3127008},
 publisher = {ACM},
 address = {New York, NY, USA}
}
@inproceedings{PROMISE2017-4-Businge,
 author = { Businge, John and  Kawuma, Simon and  Bainomugisha, Engineer and  Khomh, Foutse
Nabaasa, Evarist},
 title = {Code Authorship and Fault-proneness of Open-Source Android Applications : An Emp
 booktitle = {Proceedings of the 13th International Conference on Predictive Modelling in S
 series = {PROMISE '2017},
 year = {2017},
 isbn = {978-1-4503-5305-2},
 location = {Toronto, Canada},
 articleno = {4},
```

```
 url = {http://doi.acm.org/10.1145/3127005.3127009},
 doi = {10.1145/3127005.3127009},
 publisher = {ACM},
 address = {New York, NY, USA}
}
@inproceedings{PROMISE2017-5-Mitra,
 author = { Mitra, Joydeep and  Ranganath, Venkatesh-Prasad},
 title = {Ghera: A Repository of Android App Vulnerability Benchmarks},
 booktitle = {Proceedings of the 13th International Conference on Predictive Modelling in S
 series = {PROMISE '2017},
 year = {2017},
 isbn = {978-1-4503-5305-2},
 location = {Toronto, Canada},
 articleno = {5},
 url = {http://doi.acm.org/10.1145/3127005.3127010},
 doi = {10.1145/3127005.3127010},
 publisher = {ACM},
 address = {New York, NY, USA}
}
@inproceedings{PROMISE2017-6-Al-Zubaidi,
 author = { Al-Zubaidi, Wisam Haitham Abbood and  Dam, Hoa Khanh and  Ghose, Aditya and
Li, Xiaodong},
 title = {Multi-objective search-based approach to estimate issue resolution time},
 booktitle = {Proceedings of the 13th International Conference on Predictive Modelling in S
 series = {PROMISE '2017},
 year = {2017},
 isbn = {978-1-4503-5305-2},
 location = {Toronto, Canada},
 articleno = {6},
 url = {http://doi.acm.org/10.1145/3127005.3127011},
 doi = {10.1145/3127005.3127011},
 publisher = {ACM},
 address = {New York, NY, USA}
}
@inproceedings{PROMISE2017-7-Alelyani,
 author = { Alelyani, Turki and  Yang, Ye},
 title = {Context-Centric Pricing: Early Pricing Models for Software Crowdsourcing Tasks},
 booktitle = {Proceedings of the 13th International Conference on Predictive Modelling in S
 series = {PROMISE '2017},
 year = {2017},
 isbn = {978-1-4503-5305-2},
 location = {Toronto, Canada},
 articleno = {7},
 url = {http://doi.acm.org/10.1145/3127005.3127012},
 doi = {10.1145/3127005.3127012},
 publisher = {ACM},
 address = {New York, NY, USA}
}
@inproceedings{PROMISE2017-8-Valdivia-Garcia,
 author = { Valdivia-Garcia, Harold and  Nagappan, Meiyappan},
 title = {The Characteristics of False-Negatives in File-level Fault Prediction for Eleven
 booktitle = {Proceedings of the 13th International Conference on Predictive Modelling in S
 series = {PROMISE '2017},
 year = {2017},
```

```
 isbn = {978-1-4503-5305-2},
 location = {Toronto, Canada},
 articleno = {8},
 url = {http://doi.acm.org/10.1145/3127005.3127013},
 doi = {10.1145/3127005.3127013},
 publisher = {ACM},
 address = {New York, NY, USA}
}
@inproceedings{PROMISE2017-9-Thompson,
 author = { Thompson, Christopher},
 title = {A Large-Scale Study of Modern Code Review and Security in Open Source Projects},
 booktitle = {Proceedings of the 13th International Conference on Predictive Modelling in S
 series = {PROMISE '2017},
 year = {2017},
 isbn = {978-1-4503-5305-2},
 location = {Toronto, Canada},
 articleno = {9},
 url = {http://doi.acm.org/10.1145/3127005.3127014},
 doi = {10.1145/3127005.3127014},
 publisher = {ACM},
 address = {New York, NY, USA}
}
@inproceedings{PROMISE2017-10-Amasaki,
 author = { Amasaki, Sousuke},
 title = {On Applicability of Cross-project Defect Prediction Method for Multi-Versions Pr
 booktitle = {Proceedings of the 13th International Conference on Predictive Modelling in S
 series = {PROMISE '2017},
 year = {2017},
 isbn = {978-1-4503-5305-2},
 location = {Toronto, Canada},
 articleno = {10},
 url = {http://doi.acm.org/10.1145/3127005.3127015},
 doi = {10.1145/3127005.3127015},
 publisher = {ACM},
 address = {New York, NY, USA}
}
@inproceedings{PROMISE2017-11-Levin,
 author = { Levin, Stanislav and Yehudai, Amiram},
 title = {Boosting Automatic Commit Classification Into Maintenance Activities By Utilizing
 booktitle = {Proceedings of the 13th International Conference on Predictive Modelling in S
 series = {PROMISE '2017},
 year = {2017},
 isbn = {978-1-4503-5305-2},
 location = {Toronto, Canada},
 articleno = {11},
 url = {http://doi.acm.org/10.1145/3127005.3127016},
 doi = {10.1145/3127005.3127016},
 publisher = {ACM},
 address = {New York, NY, USA}
}
@inproceedings{PROMISE2017-12-Osman,
 author = { Osman, Haidar and Ghafari, Mohammad and Nierstrasz, Oscar},
 title = {An Extensive Analysis of Efficient Bug Prediction Configurations},
 booktitle = {Proceedings of the 13th International Conference on Predictive Modelling in S
```

```
 series = {PROMISE '2017},
 year = {2017},
 isbn = {978-1-4503-5305-2},
 location = {Toronto, Canada},
 articleno = {12},
 url = {http://doi.acm.org/10.1145/3127005.3127017},
 doi = {10.1145/3127005.3127017},
 publisher = {ACM},
 address = {New York, NY, USA}
}
```

delegate copy