Received: 23 August 2018 | Revised: 26 April 2019 | Accepted: 13 May 2019

DOI: 10.1002/smr.2184

RESEARCH ARTICLE - EMPIRICAL

WILEY Journal of Software: Evolution and Process

Identifying and predicting key features to support bug reporting

Md. Rejaul Karim[1] | Akinori Ihara[2] | Eunjong Choi[3] | Hajimu Iida[1]

[1] Graduate School of Information Science, Nara Institute of Science and Technology, Nara, Japan

[2] Faculty of Systems Engineering, Wakayama University, Wakayama, Japan

[3] Faculty of Information and Human Sciences, Kyoto Institute of Technology, Kyoto, Japan

Correspondence
Md. Rejaul Karim, Graduate School of Information Science, Nara Institute of Science and Technology, 8916-5 Takayama, Ikoma, Nara, Japan.
Email: rejaul.karim.qw4@is.naist.jp

Funding information
JSPS KAKENHI, Grant/Award Number: JP18H04094; JSPS Program for the Grant-in-Aid for Young Scientists, Grant/Award Number: 16K16037

Abstract

Bug reports are the primary means through which developers triage and fix bugs. To achieve this effectively, bug reports need to clearly describe those features that are important for the developers. However, previous studies have found that reporters do not always provide such features. Therefore, we first perform an exploratory study to identify the key features that reporters frequently miss in their initial bug report submissions. Then, we propose an approach that predicts whether reporters should provide certain key features to ensure a good bug report. A case study of the bug reports for Camel, Derby, Wicket, Firefox, and Thunderbird projects shows that Steps to Reproduce, Test Case, Code Example, Stack Trace, and Expected Behavior are the additional features that reporters most often omit from their initial bug report submissions. We also find that these features significantly affect the bug-fixing process. On the basis of our findings, we build and evaluate classification models using four different text-classification techniques to predict key features by leveraging historical bug-fixing knowledge. The evaluation results show that our models can effectively predict the key features. Our comparative study of different text-classification techniques shows that naïve Bayes multinomial (NBM) outperforms other techniques. Our findings can benefit reporters to improve the contents of bug reports.

KEYWORDS

bug report, open-source projects, prediction models

1 | INTRODUCTION

One of the key activities in the software development process is fixing bugs reported by developers, testers, and end users.[1] To fix these bugs, developers rely on the contents of bug reports.[2] A typical bug report contains input fields (eg, a summary and description), which contain unstructured features such as what the reporters saw happen (Observed Behavior), what they expected to see happen (Expected Behavior), and a clear set of instructions that developers can use to reproduce the bugs (Steps to Reproduce). These unstructured features are crucial for the developers to triage and fix the bugs.[3] However, reporters often omit these features in their bug reports.[4-7] In addition, 78.1% of bug reports have a short description that contains fewer than 100 words.[8] Thus, developers often receive bug reports with a short description such as "Various minor edits" (Camel-4820),[*] "What is it" (Derby-893),[†] "See subject" (Wicket-1159),[‡] and "See title" (Ambari-9083).[§] The lack of unstructured features in bug

[*] https://issues.apache.org/jira/browse/CAMEL-4820.

[†] https://issues.apache.org/jira/browse/DERBY-893.

[‡] https://issues.apache.org/jira/browse/WICKET-1159.

[§] https://issues.apache.org/jira/browse/AMBARI-9083.

reports is regarded as one of the key reasons for the amount of time taken to triage and fix bugs[4,9] because developers have to spend much time and effort in order to understand the bugs based on features provided or need to ask reporters to provide additional features. [4,10,11]

Well-described bug reports help in comprehending the problem, consequently increasing the likelihood of a bug being fixed.[3] Precise, well-described bug reports are more likely to gain the triager's attention.[9] Existing studies have proposed automated techniques to triage bugs, select appropriate developers, and localize bugs based on bug reports.[12-16] However, incomplete bug reports adversely affect the performance of these automated techniques.[14] Kim et al[16] found that almost half of all bug reports are unusable in terms of building a prediction model for localizing bugs. Hence, writing a well-described bug report is crucial to improving the bug-fixing process.

One of the main reasons for the lack of unstructured features in bug reports is inadequate tool support. [1,4,17,18] In order to support reporters, researchers have focused on detecting the presence/absence of unstructured features in bug reports.[17,19,20] Zimmerman et al[19] revealed 10 unstructured features that are important for developers in general. However, there is currently no consensus among software projects and bug reporting systems on the essential mandatory or optional unstructured features that should be part of a bug report.[7] Not all unstructured features are necessarily appropriate for all bug reports.[5] For example, Stack Trace is not generated for all bug reports. Previous empirical studies have found that bug reports contain only between two and six of the top 10 unstructured features.[5,10] This indicates that not all unstructured features are equally important to fix all bugs. However, selecting those features that should be provided in a bug report is not easy for reporters, especially for novices and end users. Thus, a tool that only detects the presence/absence of unstructured features is insufficient. Therefore, this study attempts to build an automated feature recommendation model to support reporters when writing new bug reports.

Bug-tracking systems for large open-source software (OSS) projects have more than 7000 bug reports for each project. Thus, examining historical bug reports can be a good way to understand which features developers require to fix bugs. To better understand which features are important, we perform an exploratory study on five OSS projects using qualitative and quantitative analyses. In particular, we first perform a qualitative analysis to identify the key features by examining those that are provided initially (ie, features that reporters provide in the initial bug reports), as well as additional required features (ie, features that reporters missed in their initial submission but that were required by developers to fix bugs). We manually investigate each bug report and identify the provided unstructured features from the initial bug reports. Then, we identify the additional required features that reporters provided in the comment sections after submitting the bug reports. In our previous study, we conducted a kick-off qualitative analysis of the Apache Camel project in order to better understand the key features of high-impact bug reports.[10] To generalize our findings, for now, we conduct a qualitative analysis of the Apache (Camel, Derby, and Wicket) and the Mozilla (Firefox and Thunderbird) ecosystems' projects. Through qualitative analysis, we identify five key features that reporters often miss in their initial bug reports and developers require them for fixing bugs.

The summary is a mandatory field when writing a bug report. Here, reporters briefly describe the detected bug. The summary text has been used successfully to detect similar and duplicate bugs.[21,22] By examining the contents of bug reports, we can determine which features are required to fix each bug. By applying machine learning techniques, reporters of new bug reports know which key features need to provide based on the summary text by leveraging historical bug-fixing activities. Thus, in order to help reporters, in our quantitative analysis, we build prediction models using naïve Bayes (NB), naïve Bayes multinomial (NBM), K-nearest neighbors (KNN), and support vector machine (SVM) text-classification techniques, based on the summary text. Existing studies have found that the performance of prediction models varies between the text-classification techniques[23,24] depending on the context. Hence, we use the aforementioned four popular text-classification techniques to build and compare prediction models. We evaluate our models using the bug reports of Camel, Derby, Wicket, Firefox, and Thunderbird projects. Our models achieve promising f1-scores when predicting key features, except for Stack Trace of the Wicket project and Code Example of the Firefox project. Our comparative study of the classification techniques reveals that NBM outperforms the other techniques in terms of predicting key features. Our contributions are twofold:

- **We perform an exploratory study on five OSS projects to investigate the initially provided and additional required features.** We identify the initially provided features from the submitted bug reports and additional required features during bug fixing. Through our qualitative analysis, we identify three features of the nine important features (ie, Observed Behaviour, Expected Behaviour, and Code Example) that reporters frequently provide (other than the summary). Then, the most common additional features required during bug fixing are Steps to Reproduce, Test Case, Code Example, Stack Trace, and Expected Behavior.
- **We build classification models using popular text-classification techniques to predict key features based on the summary text in bug reports.** We use four popular classifiers to predict whether a reporter should provide certain features based on the summary text in the bug reports. Our models achieve the best f1-scores for Code Example, Test Case, Steps to Reproduce, Stack Trace, and Expected Behavior of 0.7 (Wicket), 0.7 (Derby), 0.70 (Firefox), 0.65 (Firefox), and 0.76 (Firefox), respectively, which are promising.

The rest of the paper is organized as follows. Section 2 introduces the bug-fixing process and contents of a bug report, briefly describes related work with respect to our work, and presents the motivation for our study, including practical examples. Section 3 describes the design of our study and the qualitative and quantitative analyses. Sections 4 and 5 present and discuss the results of our qualitative and quantitative analyses, respectively. Section 6 presents the threats to validity, and Section 7 concludes the paper.

2 | BACKGROUND

This section first introduces the concept of bug report and explains how these reports contribute to different steps in the bug-fixing process. Next, we introduce related works and the motivation for our study.

2.1 | Bug-fixing process

A bug-tracking system (BTS), such as Bugzilla[¶] or Jira,[#] is commonly used to manage and facilitate the bug-fixing process in software development. The bug reports submitted to a BTS play a key role in sharing information about the bugs and the bug-fixing progress among developers. There are four basic steps in the bug-fixing process.

1. **Bug report submission.** A bug reporter describes an identified bug in a bug report. This report includes fields for providing a short summary and other features (eg, Affected Version, Component, Observed Behavior, Steps to Reproduce, and Test Case) in the bug report. Then, the report is submitted to the BTS.
2. **Triaging and assignment.** A bug triager (eg, project manager) decides whether the project should fix the bug because it may already have been submitted by other reporters (eg, duplicate bug[25]). If the triager decides to fix the reported bug, he or she assigns it to an appropriate developer.
3. **Localizing and fixing.** The assigned developer identifies the source code files that contain the reported bug. Here, the developer may request additional features, if needed. Then, the developer fixes the code.
4. **Verification.** A different developer (eg, tester) verifies whether the corrected code now satisfies the reporter's requirements. If it does, then the developer closes the bug report. Otherwise, the bug report is reopened and the triager reassigns it to the developer for correction.

2.2 | Contents of a bug report

The bug report plays an important role in fixing a bug in all steps of the bug-fixing process. The contents of a bug report are a collection of structured and unstructured features. The structured features usually represent the text defined by the project, such as the Component, Affected Version, and Priority. The unstructured features represent the text not defined by the project, such as Observed Behavior, Stack Trace, and Code Example. The unstructured features are the most important for the developers attempting to localize and fix the bugs.[19] Hence, reporters should write detailed descriptions of the bugs for the developers. In order to support the description, reporters sometimes submit files such as Screenshot and Error Report with the unstructured features. Table 1 briefly describes the top 10 important features[3] of a bug report.

2.3 | Related work

Many empirical studies have proposed ways in which to improve the contents of bug reports. Bettenburg et al[3] conducted a survey of 156 experienced developers and reporters from three OSS projects to examine what features developers expect to see in bug reports. As a result of their survey, they revealed 16 important structured and unstructured features for fixing bugs. They also developed a prototype tool called "CUEZILLA" to measure the quality of bug reports. To validate their prototype tool, they randomly selected 289 bug reports and then asked developers to assess their quality on a 5-point Likert scale ranging from very poor to very good.[26] Then, they used these 289 bug reports to train and evaluate CUEZILLA by building supervised learning models. Their models achieved 45% accuracy when measuring the quality of the bug reports. In contrast, we build prediction models based on the titles/summaries of bug reports to notify reporters on which features to provide in a new bug report.

Davies and Roper[5] conducted a case study on four OSS projects based on the top 10 important features (see Table 1) to understand which features reporters provide in bug reports. They found that bug reports do not always provide all important features. Furthermore, they found that 12% of all of features are provided after the initial submissions of the bug reports. As a result, developers spend valuable time collecting the required features. In order to inform the design of new bug-reporting tools, Ko et al[27] conducted a linguistic analysis of the titles of bug reports. They observed a large degree of regularity and a substantial number of references to visible software entities, physical devices, or user actions. Their results suggest that future BTSs should collect data in a more structured way. Our study focuses on revealing the additional required features that reporters often omit from bug reports. These additional requirements during bug fixing might increase the time required to fix the bugs.

[¶]http://www.bugzilla.org/.

[#]https://issues.apache.org/.

TABLE 1 The top 10 most important features of a bug report

Feature	Description
Steps to Reproduce (STR)	A clear set of instructions that the developer can use to reproduce the bug
Stack Trace (ST)	A stack trace produced by the application, most often when the bug reports a crash in the application
Test Case (TC)	One or more test cases that developers can use to determine whether they have fixed the bug
Observed Behavior (OB)	What the user saw happen in the application as a result of the bug
Screenshot (SS)	A screenshot of the application while the bug is occurring
Expected Behavior (EB)	What the user expected to happen, usually contrasted with Observed Behavior
Code Example (CE)	An example of code that can cause the bug
Summary (S)	A short (usually one sentence) summary of the bug
Environment (EN)	The operating system and version the user was using at the time of the error
Error Report (ER)	An error report produced by the application as the bug occurred

Thus, to reduce the additional requirements and to improve the contents of bug reports, we build prediction models using popular text-mining techniques (ie, NB, NBM, KNN, and SVM).

Chaparro et al[17] conducted an empirical study to understand the extent to which Observed Behavior, Steps to Reproduce, and Expected Behavior are reported in bug reports and what discourse patterns (ie, rules that capture the syntax and semantics of the text, eg, "To reproduce" and "STR" are the discourse patterns of Steps to Reproduce) reporters use to describe such information. Then, they designed an automated approach to detect the absence or presence of Steps to Reproduce and Expected Behavior in bug descriptions. Their approach intends to warn reporters if they forget to provide these features in the descriptions. In contrast, our approach helps reporters to understand which features should be provided in the descriptions when writing bug reports. Thus, reporters can create effective bug reports by providing the minimum number of features.

In addition, many existing studies use the features of bug reports to improve the bug-fixing process. Hooimeijer et al[28] presented a basic linear regression model that predicts whether bug reports are resolved within a given period. Their model is based on structured features (ie, Daily Load, Submitter Reputation, Readability, and Severity) that can be readily extracted from a bug report within a day of its initial submission to the repository. In contrast, our models are based on unstructured features (eg, Summary, Steps to Reproduce, Test Case, and Code Example). Another important difference is that our models intend to recommend which features reporters should provide in the description to create a good bug report. Several other studies have applied automated techniques to select appropriate developers,[12] localize bugs,[14] and predict bug-fixing effort.[15]

These works motivate us to conduct an exploratory study to understand the key features by examining the bug reports of OSS projects and developers' activities during bug fixing. Then, we build prediction models by leveraging popular text-mining techniques that enable reporters to know which key features are required by developers during bug fixing.

2.4 | Motivating example

Existing studies[3,5,10] have shown that developers rely on bug reports when fixing bugs. However, it is sometimes difficult to reproduce and localize these bugs.

For example, Figure 1 shows an example of a bug report[‖] where the developer was required additional features about the detected bug in the Camel project. The reporter, David J.M. Karlsen, provided Observed Behavior and Stack Trace as the unstructured features in the description of the bug report. When the developer, Willem Jiang, received the assignment to fix the bug, he failed to reproduce the bug and asked for a Test Case from the reporter one day later. The reporter then responded 77 days later. Such delays can be costly in terms of performance and may influence reports of new bugs. To avoid these kinds of delay, we propose an approach that suggests which unstructured features developers will require in order to fix the bugs. The approach will be especially helpful for novices and end users when writing a new bug report.

Figure 2 shows another example of a bug report[**] from the Apache Camel project. This bug was reported and fixed 1 month earlier than that shown in Figure 1. The reporter, Julien Graglia, provided the similar structured features as the first example. However, he clearly mentioned Observed Behavior, Expected Behavior, Test Cases, and Stack Trace as unstructured features in the description of the bug report. The developer, Christian Muller, found sufficient features in the bug report to localize and fix the bug. Finally, the developer fixed the bug within 1 day.

[‖]https://issues.apache.org/jira/browse/CAMEL-5860.

[**]https://issues.apache.org/jira/browse/CAMEL-5782.

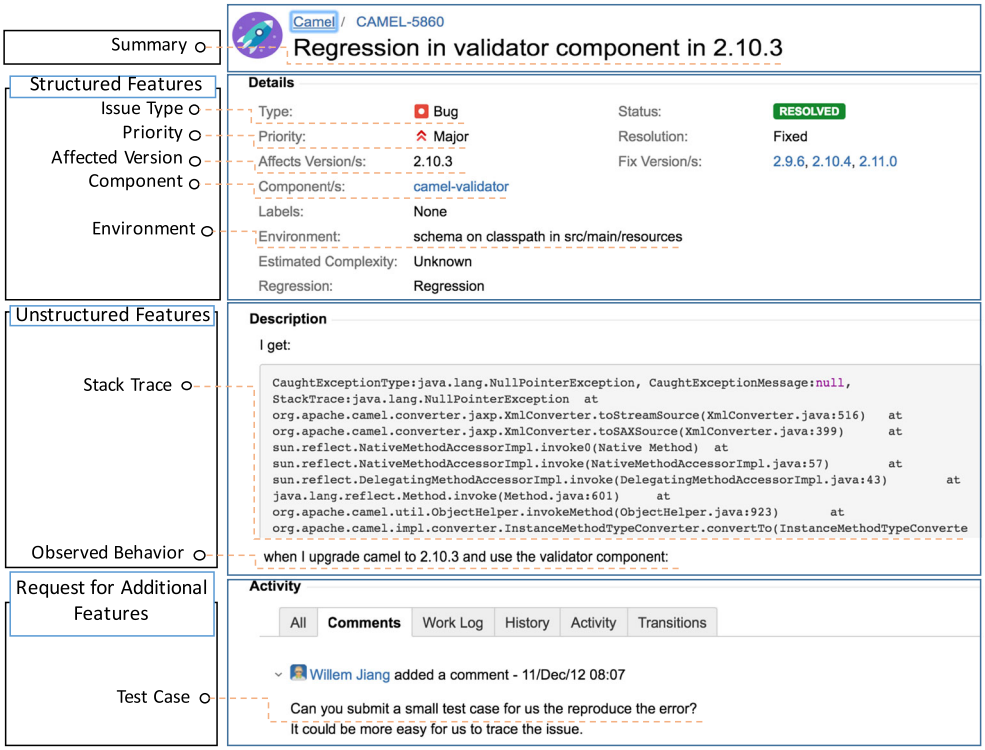


FIGURE 1 Example of a bug report that required additional features to fix the bug

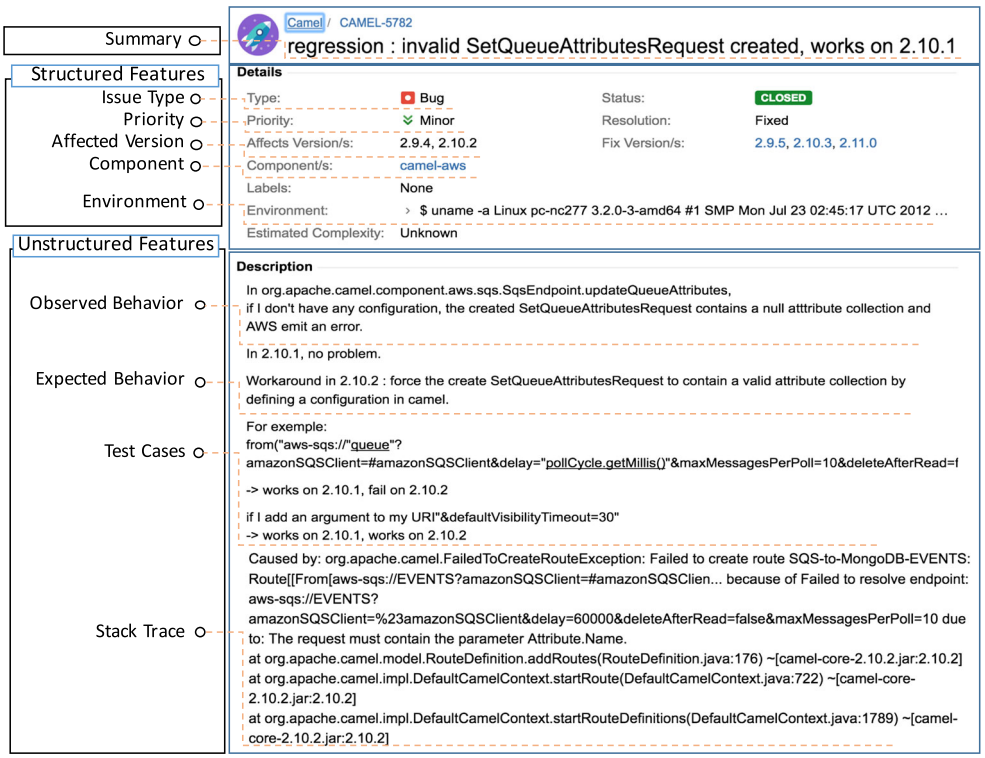


FIGURE 2 Example of a bug report without a request for additional features

In both examples, the bug reports are for regression-related bugs and both contain almost similar structured features except priority. Although the priority of the bug report CAMEL-5860 was "Major," the developer took a long time to fix the bug. On the other hand, the priority of the bug report CAMEL-5782 was "Minor," the developer fixed within 1 day after the assignment. Usually, a bug report with the higher priority gets more

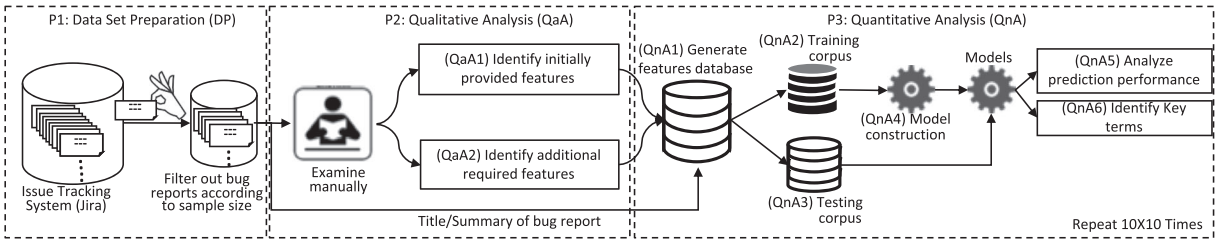


FIGURE 3 An overview of our study design

attention than the lower priority by the developers because of its impact on the software project. After analyzing the bug-fixing activities, we find that the Test Case is essential in both cases. However, the reporter of the bug report CAMEL-5860 (see Figure 1) did not provide this feature in the initial submission, which delayed fixing the bug, in contrast to the bug report CAMEL-5782 (see Figure 2). The first report, CAMEL-5860, was created 1 month after the bug in CAMEL-5782 was fixed. The reporter of CAMEL-5860 might have known which features are essential to fixing the bug because a similar type of bug in CAMEL-5782 was already fixed. Thus, the reporter could have saved valuable time for the developers by providing these features in the initial submission. However, it is very difficult for reporters to know what features are required based on fixed bug reports from the repository without any automated techniques. This motivates us to develop classification models to predict the key features reporters should provide in initial bug reports based on reports of bugs that have already been fixed. In the future, we will work on the closed projects.

3 | STUDY DESIGN

We conduct an exploratory study on bug reports of OSS projects to determine how to write a good bug report. To do so, we extract fixed bug reports (ie, marked as FIXED) from JIRA (Camel, Derby, and Wicket projects) and BugZilla (Firefox and Thunderbird projects) repositories. We then conduct a qualitative analysis to identify the key features of a bug report. We manually examine each bug report and identify the initially provided features. We also examine bug-fixing activities, especially the comments section, and identify additional required features. Then, we conduct a statistical analysis to understand the impact of the additional features on the bug-fixing process. Furthermore, we build classification models to predict the key features that reporters should provide in order to create good bug reports. Figure 3 provides an overview of our study. Our study comprises three phases: P1: Data Set Preparation (DP); P2: Qualitative Analysis (QaA); and P3: Quantitative Analysis (QnA). We describe each phase below.

3.1 | Data set preparation

In order to prepare data sets, we first set up three essential criteria for selecting target projects. Then, we generate the sample size and randomly select bug reports to prepare the sample data set for each of the selected projects.

- **Criterion 1 - Projects have a large number of bug reports in the issue-tracking system.** A previous study[29] found that a data set containing few bug reports is difficult to use when building data-mining or machine learning models. Thus, we target projects with a large number of bug reports because this indicates that the project is more mature and stable.
- **Criterion 2 - Projects have well-structured bug-fixing histories.** This study analyzes the historical communication logs between developers and reporters to understand what features were required to fix each bug.
- **Criterion 3 - Projects differ in terms of their application domains.** The contents of bug reports may vary between different application domains. To increase the generalizability of our results, we need to select projects from different application domains for our study.

We initially selected four projects from the Apache Software Foundation (ASF)[††] and two projects from the Mozilla Foundation that met the above criteria. The projects are Ambari,[‡‡] Camel,[§§] Derby,[¶¶] Wicket,[##] Firefox,[‖‖] and Thunderbird.[***] However, we found that the majority of the

[††] https://www.apache.org/.

[‡‡] https://issues.apache.org/jira/projects/AMBARI.

[§§] https://issues.apache.org/jira/projects/CAMEL.

[¶¶] https://issues.apache.org/jira/projects/DERBY.

[##] https://issues.apache.org/jira/projects/WICKET.

[‖‖] https://bugzilla.mozilla.org/buglist.cgi?quicksearch=Firefox.

[***] https://bugzilla.mozilla.org/buglist.cgi?quicksearch=Thunderbird.

bug reports in the Ambari project are self-reported (ie, reporters fix the detected bugs themselves). Self-reported bug reports have an impact on the bug-fixing process.[30] Because we conjecture that self-reported bug reports are likely to have incomplete descriptions, we exclude the Ambari project from our case study. The following is the brief description of each of the selected projects.

Apache Camel: An open-source framework for message-oriented middleware with a rule-based routing and mediation engine, written in Java.
Apache Derby: A relational database management system developed, written in Java.
Apache Wicket: A component-based web application framework for the Java programming language, written in Java.
Mozilla Firefox: A free and open-source web browser, written mainly in C++.
Mozilla Thunderbird: A free and open-source cross-platform email client, written mainly in C++.

The JIRA and BugZilla contains a large number of bug reports for each of the selected projects. To make our manual analysis simple and rational, we generate a statistically representative sample size for each project. To obtain proportion estimates that are within 5% bounds of the actual proportion, with a 95% confidence level, we randomly select a sample of size $s = \frac{z^2 p(1-p)}{0.05^2}$, where p is the proportion we want to estimate and z=1.96. Because we do not know the proportion in advance, we use p = 0.5. We further correct for the finite population of bug reports P using $ss = \frac{s}{1 + \frac{s-1}{P}}$ to obtain the sample for our qualitative analysis. Table 2 shows the statistics for the analyzed bug reports.

In order to prepare the data set for each target project according to the sample size, we first filter out bug reports that satisfy criterion i, described below. Then, we randomly select bug reports from the set of all reports according to the sample size. In our analysis, we also exclude bug reports that satisfy criterion ii and replace them with other, randomly selected bug reports.

- **Criterion i - The bug reporter and the fixer is the same person.** If the bug reporter and the bug fixer is the same person, then the report may not include all of the features required to fix the bugs. Therefore, we exclude these bug reports.
- **Criterion ii - The provided URL no longer exists.** Some bug reporters provide a URL of a website instead of writing features in the description. At the time of our analysis, we could not access some of those URLs. Therefore, we exclude these bug reports from our analysis.

3.2 | Study design of qualitative analysis

Motivation: A typical bug report contains 16 structured and unstructured features.[19] The reporters need to report the features that are suitable for localizing bugs. However, the reporters often omit unstructured features needed by the developers when fixing the bugs. Consequently, the developers need to additionally collect these features, as shown in the motivating example in Section 2.4. In our previous study, we conducted a kick-off qualitative analysis of the Apache Camel project of Ohira et al[31] data set to understand the key features for high-impact bug reports, based on 10 unstructured features. As the name implies, high-impact bugs are bugs that have high impact on software processes, products, or end users. Software engineering researchers have introduced different types of high-impact bugs based on their impact such as security bugs,[32] performance bugs,[33] breakage bugs,[34] surprise bugs,[34] dormant bugs,[35] and blocker bugs.[36] Although the data set contains six types of high-impact bug reports in the four OSS projects, some high-impact bugs contain very few bug reports. For example, there are six and three bug reports for the blocking bug of the Camel and Wicket projects, respectively. We did not find any other robust data sets for the high-impact bug reports. Thus, to generalize our findings, for now, we conduct a qualitative analysis for five OSS projects from Apache (Camel, Derby, and Wicket projects) and Mozilla (Firefox and Thunderbird projects) ecosystems without considering high-impact bugs. In particular, we investigate (a) the features reporters frequently provide in an initial bug report, (b) the features reporters frequently omit, but that developers require after the initial submission, and (c) the impact of the additional required features on bug-fixing process.

TABLE 2 Statistics of bug reports and the sample size for each target project

Ecosystems	Projects	# Total Issue Reports	# Fixed Bug Reports	# Sample Size
Apache	Camel	11798	3964	350
Apache	Derby	6955	4063	351
Apache	Wicket	6466	3946	350
Mozilla	Firefox (General)	10000	5353	358
Mozilla	Thunderbird (General)	8693	1614	310

Approach: Our qualitative analysis focuses on 10 unstructured features (see Table 1) for the sampled bug reports described in Section 3.1. These unstructured features are crucial to developers when fixing bugs.[19] Figure 3 provides an overview of our qualitative analysis process (QaA1 and QaA2). In QaA1 and QaA2, we identify the initially provided and additional required unstructured features. Then, we divide the bug reports into two groups, namely, those without and those with additional required features. Finally, we investigate the differences in bug-fixing time, the number of comments (ie, comments made by developers and reporters during bug fixing), and the number of commentators (ie, developers and reporters who participated in discussions during the bug fixing) between the two groups in order to understand the impact of the additional required features on the bug-fixing process.

(QaA1) **Identify the unstructured features in the initial submission.** We manually identify the unstructured features in the initial bug reports because these features are provided using natural language text in the description. Then, we double check the identified features for all sampled bug reports. If there are differences in the identified features for the same bug report, we attempt to reach a consensus on the features.

(QaA2) **Identify the additional required features:** We manually examine bug fixing activities and identify the additional unstructured features that developers requested during bug fixing after the initial submission. Then, we double check the identified features and reach a consensus on the features if there are differences for the same bug report.

3.3 | Study design of quantitative analysis

Motivation: Developers expect reporters to provide useful unstructured features in their bug reports in order to localize and fix bugs. However, reporters, especially novices and end users, sometimes find it difficult to do so, because they might not know which features will help developers to fix the bugs.[19] An automated technique that recommends which features to include in bug reports can reduce the number of missing features.[5,9,17-19] Thus, we propose a model that predicts which key features reporters should provide in bug reports that help developers to fix the bugs.

Approach: To predict these key features, we build classification models using four popular text-classification techniques. These models are trained using the "summary," which is one of the unstructured features reporters often use to include key instructions on the detected bugs. Figure 3 provides an overview of our quantitative analysis process (QnA1, QnA2, QnA3, QnA4, and QnA5).

(QnA1) Construct a database of the unstructured features: In our qualitative analysis, we manually identify the unstructured features provided in the initial bug reports and comments section during bug fixing. In our quantitative analysis, we build models that predict the key features reporters should provide in their initial bug reports. To train our prediction models, we construct a database based on the summaries and the identified unstructured features of the bug reports.

(QnA2 & QnA3) Divide the database into a training & a testing corpus: To validate our prediction models, we use 10-fold cross-validation. First, we split the key features database into 10 equal parts to create a training corpus and a testing corpus. Then, we use nine parts for the training corpus to construct the prediction models in the first round, setting aside one part for the testing corpus. We continue this process until we complete 10 rounds to ensure that each part of the database is used for training and testing corpus. We repeat the whole 10 rounds process of generating training and testing corpus 10 times to ensure the robustness of our approach.

(QnA4) Build prediction model: The performance of the prediction models varies between the different text-classification techniques[23,24,37] depending on the context. Hence, this study uses four popular text-classification techniques, namely, NB,[38] NBM,[38] KNN,[39] and SVM,[39] to build models. The reason to choose these techniques is they are classic and diverse algorithms. They represent features in different ways from each other. Existing studies show that they perform well in many text-classification tasks.[23,24,40,41] In addition, the learning strategies of these techniques are different from each other. Although both NB and NBM are based on the Bayes theorem, they represent features in different ways. SVM is a supervised learning model based on the structural risk-minimization principle. Unlike NB or NBM, SVM is a nonprobabilistic binary linear classifier. KNN is a distance-based classification algorithm and differs from NB and NBM. For these reasons, we build prediction models using these techniques and conduct a comparative study to understand which technique performs well in our context. Class imbalance is always a problem in machine learning and can lead to a classifier exhibiting poor performance. Imbalanced learning strategies can be employed to balance an initially imbalanced data set and help a trained classifier not to be biased to the majority class. Thus, in most cases, they improve the performance of the classifier.[42,43] There are many imbalanced learning strategies. In our study, we use the synthetic minority oversampling technique (SMOTE). SMOTE is a more sophisticated oversampling technique.

(QnA5) Analyze prediction performance: To evaluate the performance of our prediction models, we use traditional evaluation metrics, namely, precision, recall, and the f1-score. These metrics are commonly used to evaluate classification performance[44] and can be derived from a confusion matrix. A confusion matrix lists all four possible classification results. If a feature of a bug report predicts correctly, it is a true positive (Tp). If it predicts incorrectly, then it is a false positive (Fp). Similarly, there is a false negative (Fn) and a false positive (Fp) outcome. Based on Tp, Fp, Fn, and Tn, we calculate the precision, recall, and f1-score. Precision is the proportion of correctly predicted features for all bug reports

that predicted the feature. Mathematically, precision P is defined as $P = \frac{Tp}{Tp + Fp}$. Recall is the proportion correctly predicted features in bug reports to the actual number of features in the bug reports. Mathematically, recall R is defined as $R = \frac{Tp}{Tp + Fn}$. The f1-score is a summary measure that combines the precision and recall. It evaluates whether an increase in precision (recall) outweighs a reduction in recall (precision). Mathematically, the f1-score F is defined as $F = \frac{2*P*R}{P+R}$.

Baseline model: Software engineering researchers use random predictors to compare the performance of their prediction models.[34,40,45] Since our model is the first to predict key features, we also use a random predictor as a baseline model to evaluate the prediction performance of our model. The precision of a particular feature for random prediction is the percentage of the feature present in the data set. Since the random prediction is a random classifier with two possible outcomes (eg, feature provided or not provided), its recall is 0.5.

(QnA6) Identify key terms: To identify the key or best discriminator terms, we extract all the terms from the summary of bug reports and then exclude stop words. After performing pre-processing and stemming terms into root forms, we found 963, 1024, 931, 825, and 847 unique terms for Camel, Derby, Wicket, Firefox, and Thunderbird projects, respectively. The primary assumption is among these terms some are highly discriminatory for each key feature than others. The most discriminator terms could provide a deep insight into each feature. Recently, a large-scale study[46] conducted on 30 feature selection techniques and apply on 18 data sets. They found that correlation based ranking search technique perform well to select the important features. We also use a correlation based ranking search technique with 10-fold cross-validation to identify the top 10 discriminator terms for each key feature.

4 | IDENTIFICATION OF KEY FEATURES

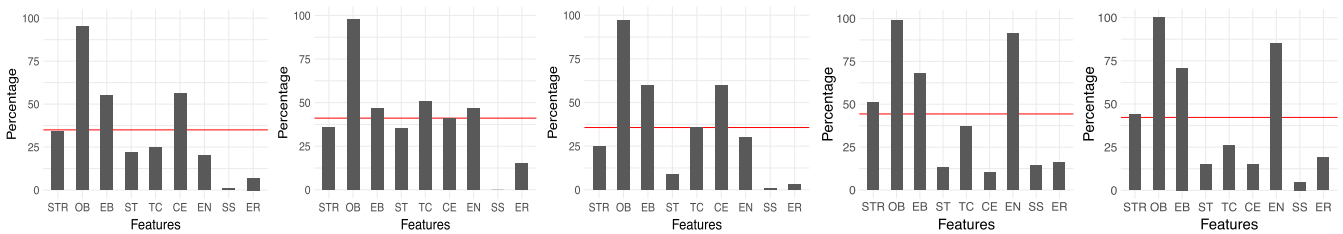
This section presents and discusses the results of our qualitative analysis to identify the features provided in the initial submission of the bug reports, as well as those provided later through discussions between the reporters and the developers during bug fixing.

4.1 | Analysis result

In the qualitative analysis QaA1, we focus on the features that are frequently provided in initial bug reports. Figure 4 shows the percentages of each of these features provided in the initial bug reports for the Camel, Derby, Wicket, Firefox, and Thunderbird projects. The red line shows the average percentages of the provided features. They are 35%, 41%, 36%, 44%, and 42% in each project, respectively. We found that Observed Behavior frequently exists in the bug reports of the projects. On the other hand, Screenshot and Error Report rarely exist in the bug reports. The rankings of the other features vary across the projects. In these five projects, the most frequently provided features are Observed Behavior, Expected Behavior, Code Example, Steps to Reproduce, Test Case, and Environment. These findings are almost similar to those of existing studies.[5,10]

In the qualitative analysis QaA2, we focus on the additional unstructured features that developers require during bug fixing. Figure 5 shows the percentages of these features for each project. The percentages are calculated as follows: $x_i = \frac{\#additional\,requirements\,for\,x_i}{\#bug\,reports\,that\,were\,required\,additional\,features} * 100$.

In Figure 5, we find that Steps to Reproduce is the additional feature most often required in the Derby, Wicket, and Firefox projects, whereas Test Case is in the Camel and Stack Trace is in Thunderbird projects are most often required features. For all of the projects, Steps to Reproduce and Test Case are the most additional required features during bug fixing. We find that Code Example is less often requested in the Derby project compared with other projects. Stack Trace and Expect Behavior are often requested in all five projects. For the other features, we find developers make a very few additional requests during bug-fixing, except Screenshot and Error Report for the Thunderbird project. This suggests that Steps to Reproduce, Test Case, Code Example, Stack Trace, and Expect Behavior are the features requested most often after the initial submission.



(A) Camel (B) Derby (C) Wicket (D) Firefox (E) Thunderbird

FIGURE 4 The percentages of initially provided features across different projects

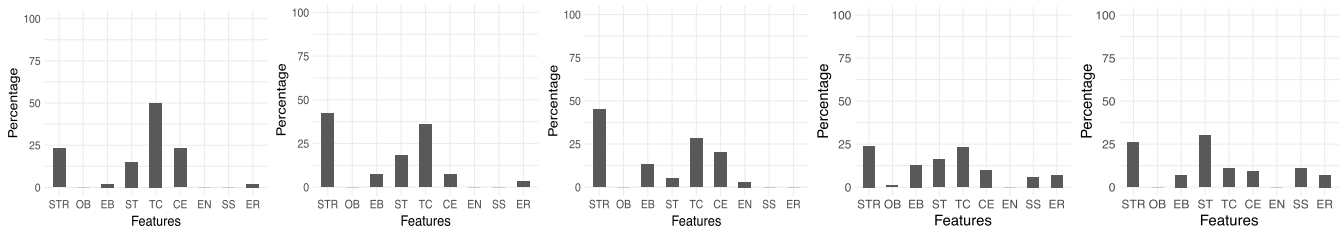


FIGURE 5 The percentages of additional required features during bug fixing across different projects

It is obvious that requests for additional features tend to increase bug-fixing time. However, we do not know how the additional features impact the bug-fixing process. To determine this effect, we divide the bug reports into two groups (ie, bug reports with additional required features and those without additional required features) for each project. Then, we analyze the bug-fixing time, the number of comments made by developers during bug fixing, and the number of commentators who participated in the bug-fixing process for each group.

Figure 6A shows the distribution of the bug-fixing time using bean plots. The distributions shown in gray and black are those of the bug-fixing times without additional required features and with additional required features, respectively. We find that the median bug-fixing time with the additional required features is much higher than that without additional features across all projects. We observe a significantly different ($P << .05$) bug-fixing time between these two groups using the Mann-Whitney U test with a large (eg, Derby and Wicket projects) and a medium (eg, Camel, Firefox, and Thunderbird projects) effect size (see Table 3). Figure 6B shows that the median value of the number of comments (ie, the number of comments made by developers and reporters during bug fixing) with additional required features is higher than that without additional features. We observe a significant difference in the comment count with a large effect size for the Camel and Wicket projects, a medium effect size for the Derby project, and a small effect size for the Firefox and Thunderbird projects. This indicates that the additional features needed to fix bugs tend to increase the number of comments. In Figure 6C, we observe a significant difference in the number of commentators (ie, the number of developers and reporters who participated in discussions during bug fixing) between the two groups. This indicates that the additional features needed to fix bugs tend to increase the number of developers required to fix the bugs. These findings suggest that the additional required features have a significant impact on the bug-fixing process. Therefore, the features requested most often, such as Steps to Reproduce, Test Case, Code Example, Stack Trace, and Expected Behavior, might be key features in writing a bug report.

4.2 | Discussion

The purpose of our qualitative analysis is to understand the key features needed to write a good bug report. In QaA1, we identify the frequently provided unstructured features based on initial bug reports. However, we do not yet know whether these features are key features. Reporters might frequently provide unstructured features that are easy to produce. For example, the most frequently reported unstructured feature is Observed Behavior. Sasso et al[7] found that this feature is easy for reporters to provide. Thus, to obtain a deeper understanding of the key features, we analyze the discussions between the developers and the reporters during bug fixing in QaA2. We find that the reporters often omit five unstructured features (ie, Steps to Reproduce, Test Case, Code Example, Stack Trace, and Expected Behavior) from their initial submissions. Thus, the developers requested that the reporters provide these features during bug fixing. We note that Steps to Reproduce and Test Case are requested most often. In reality, Steps to Reproduce is useful because it enables developers to reproduce[11,19] and understand the bugs. Sometimes, the developers cannot fix a bug without Steps to Reproduce.[11] Test Case is also useful to developers when checking whether the fixed patches are working as expected.[1] A previous survey revealed that 83% and 51%, respectively, of developers consider these two features as helpful when fixing bugs.[19] This suggests that the additional features are particularly important to developers when fixing bugs.

To generalize our findings, we set up our case study on the bug reports of different application domains projects from two ecosystems. In our QaA1, we notice that reporters provided Code Example less often in the Mozilla projects compared with the Apache projects. After a closer look, we see that reporters of the selected Apache projects are mostly developers. They sometimes encounter problems during writing codes, eg, bug reports-Wicket-5220[†††] and Wicket-5237.[‡‡‡] Thus, they could easily include Code Example in the description of the bug reports. On the other hand, the selected Mozilla projects are client applications and reporters are mostly end users. They frequently use application and encounter problems, eg, bug reports-bug#216608,[§§§] bug#220181,[¶¶¶] and bug#247128.[###] Thus, they could capture Screenshot and error message (Error

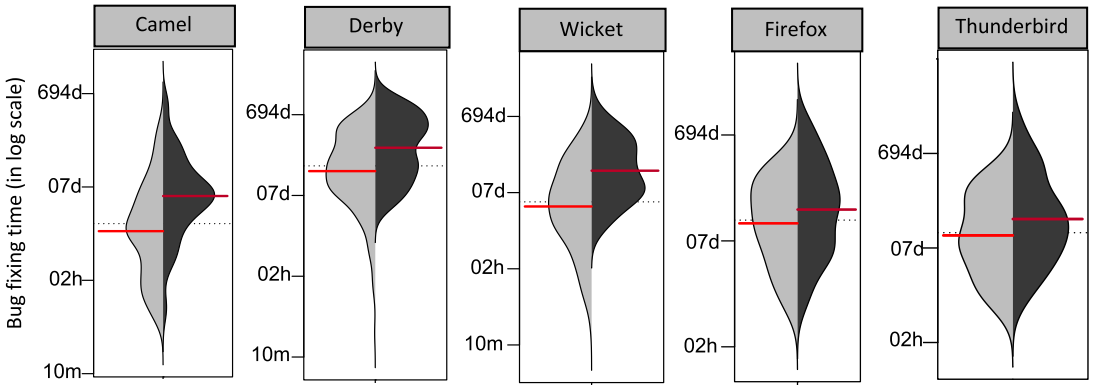
[†††]https://issues.apache.org/jira/browse/WICKET-5220

[‡‡‡]https://issues.apache.org/jira/browse/WICKET-5237

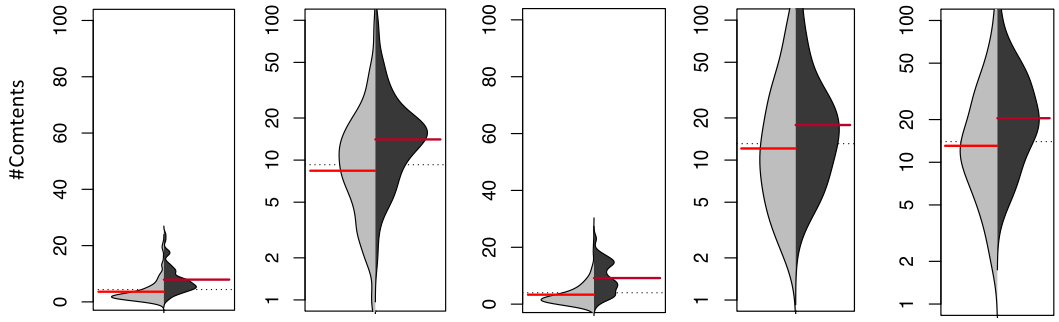
[§§§]https://bugzilla.mozilla.org/show_bug.cgi?id=216608

[¶¶¶]https://bugzilla.mozilla.org/show_bug.cgi?id=220181

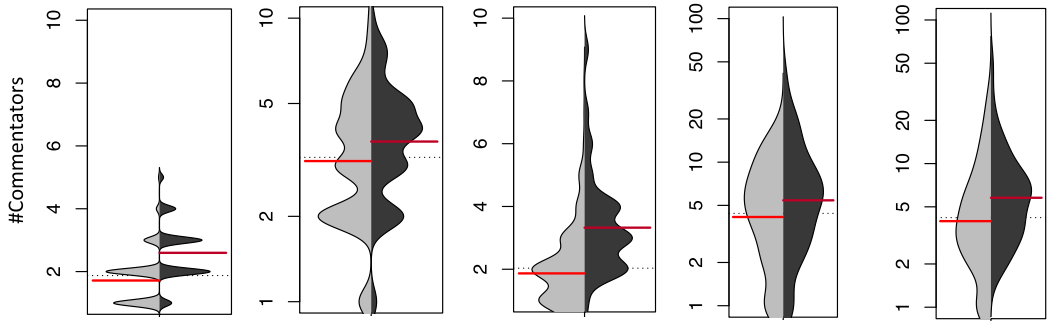
[###]https://bugzilla.mozilla.org/show_bug.cgi?id=247128



(A) The distribution of bug fixing time between bug reports with additional required features (black) and without additional required features (gray) groups



(B) The distribution of number of comments between bug reports with additional required features (black) and without additional required features (gray) groups



(C) The distribution of number of commentators between bug reports with additional required features (black) and without additional required features (gray) groups

FIGURE 6 Impact of additional required features on bug-fixing process

Report) and include in the bug reports. However, Code Example is difficult for them to provide in the bug reports.[19] In our QaA2, we see that Screenshot and Error Report requested more often requested in the Mozilla projects compared with the Apache projects. The conversation between the developers (ie, bug fixers) and the reporters of the bug report-bug#522459[IIII] and bug#370401[****] in Thunderbird project show that these two features help the developers to understand and fix the bugs. Interestingly, Steps to Reproduce and Test Case are the most often additional required features for both ecosystems except the Thunderbird project. We also notice that the top five additional required features slightly vary in the Thunderbird project compared with the other four projects. However, Steps to Reproduce, Test Case, Code Example, Stack Trace, and Expected Behavior are the additional required features requested most in these five projects.

A recent study found the textual difference between the bug reports written by an expert (ie, anyone who has contributed in the source code of a project) and a nonexpert reporter (ie, anyone who has not contributed in the source code of a project).[47] Thus, our primary assumption is that the key features such as Code Example, Test Case reporting might also depend on the reporter's experiences. In order to understand whether the

[IIII] https://bugzilla.mozilla.org/show_bug.cgi?id=522459

[****] https://bugzilla.mozilla.org/show_bug.cgi?id=370401

TABLE 3 Result of Mann-Whitney *U* test and effect size test on with additional features required and without additional features required groups during bug fixing

Projects	Bug Fixing Time		Comments During Bug Fixing		Commentators in Bug Fixing	
	P Value	Cliff δ	*P* Value	Cliff δ	*P* Value	Cliff δ
Camel	$P << .05$	M	$P << .05$	L	$P << .05$	L
Derby	$P << .05$	L	$P << .05$	M	$P << .05$	S
Wicket	$P << .05$	L	$P << .05$	L	$P << .05$	L
Firefox	$P << .05$	M	$P << .05$	S	$P << .05$	M
Thunderbird	$P << .05$	M	$P << .05$	S	$P << .05$	S

Note. Effect size: (L) Cliff $\delta \geq 0.474$, (M) $0.33 \leq \delta < 0.474$, (S) $0.147 \leq \delta < 0.33$, (N) $\delta < 0.147$.

key features reporting depends on the type of bug reporters, we perform a simple qualitative analysis from the two perspectives of bug reporters such as bug reporting experiences and the bug-fixing experiences. To perform our analysis, we classify the reporters of our target sampled bug reports as an experienced reporter if they have submitted at least one bug report before otherwise a nonexperienced. Indeed, for each reporter of our target sampled bug reports, we look for his/her past submitted bug reports. If the reporter submitted at least one bug report in the past, we classify the reporter as an experienced reporter. Otherwise, we classify the reporter as a nonexperienced reporter. Similarly, we classify the reporters of our target sampled bug reports as a contributor if they have fixed at least one bug report before otherwise an end user. We take the feature "Code Example" as an example for this analysis because it is one of the difficult features for the reporter.[19] We first investigate whether the bug reporting experiences (experienced vs nonexperienced) of bug reporters affect the key features reporting. Table 4 shows that the experienced reporter tends to report comparatively higher percentages of Code Example in both ecosystem projects than the nonexperienced reporter do. For example, in the case of the Camel project, the experienced reporter provided 46% more Code Example than the nonexperienced reporter. In the case of the Thunderbird project, the experienced reporter provided 153% more Code Example that the nonexperienced reporter. Then, we investigate whether the bug fixing experiences (contributor vs end user) of the bug reporters affects the key features reporting. From the Table 5, we do not find any relationship, which ascertains that the Code Example reporting depends on the type of bug reporters for the Apache projects. However, we find that the Code Example provided mostly by the contributor in the Mozilla projects. For example, the contributor provided 85% of the total Code Example in the Thunderbird project whereas the end user provided only 15%. From this analysis, we see that both bug reporting and bug fixing experiences of bug reporters may affect the key features reporting. Zaman et al[48] conducted a case study on security- and performance-related bugs of the Mozilla Firefox project and observed different characteristics between them. This indicates that the different type of bugs may require different key features to fix. This remains as our future work.

We found that the bug-fixing time with additional required features increases significantly. In reality, the bug-fixing time depends on many factors, such as the complexity of a bug and the developer's expertise. We find that the additional features also affected the bug-fixing time. In the case of the bug report CAMEL-5860,[††††] explained in Section 2.4, the developer had to wait many days to get the requested feature to reproduce the bug. In another case, when the bug report[‡‡‡‡] was assigned, the developer tried to reproduce the bug in different ways using the provided features. However, after failing to do so, the developer had to ask for Steps to Reproduce in order to reproduce the bug. In both cases, the bug-fixing time might have been reduced by providing the feature as part of the initial submission. Zimmermann et al[19] also claim that missing features are one of the biggest causes of delays in fixing bugs.

Based on these findings, we conclude that the additional features requested most often, such as Steps to Reproduce, Test Case, Code Example, Stack Trace, and Expected Behavior, might be key features when writing a bug report. Therefore, developing an automated approach to predict and suggest these key features to reporters may improve the bug-fixing process. That leads us to develop an approach to predict such features.

5 | PREDICTION OF KEY FEATURES

This section presents and discusses the results of our models in predicting the key features that reporters should provide in their initial bug reports.

5.1 | Analysis result

To predict the key features, we build classification models using four text-classification techniques. Our models are trained based on the summary text of bug reports. Table 6 shows the total number of unique terms that extracted from the summary of the bug reports (UT), percentage of

[††††]https://issues.apache.org/jira/browse/CAMEL-5860.

[‡‡‡‡]https://issues.apache.org/jira/browse/CAMEL-1199.

TABLE 4 The Code Example reporting statistics for each project based on the bug reporting experiences of the reporters

Projects	Reporter Type	Code Example (Provided/Not Provided)	# Bug Reports	Percentages
Camel	Experienced	No	88	43%
		Yes	118	57%
	Nonexperienced	No	63	44%
		Yes	81	56%
Derby	Experienced	No	186	63%
		Yes	109	37%
	Nonexperienced	No	21	38%
		Yes	35	63%
Wicket	Experienced	No	74	38%
		Yes	123	62%
	Nonexperienced	No	51	34%
		Yes	101	66%
Firefox	Experienced	No	131	86%
		Yes	21	14%
	Nonexperienced	No	192	83%
		Yes	14	7%
Thunderbird	Experienced	No	138	81%
		Yes	33	19%
	Nonexperienced	No	123	90%
		Yes	13	10%

TABLE 5 The Code Example reporting statistics for each project based on the bug fixing experiences of the reporters

Projects	Reporter Type	Code Example (Provided/Not Provided)	# Bug Reports	Percentages
Camel	Contributor	No	60	44%
		Yes	77	56%
	End user	No	95	46%
		Yes	112	54%
Derby	Contributor	No	132	56%
		Yes	104	44%
	End user	No	75	40%
		Yes	40	35%
Wicket	Contributor	No	46	48%
		Yes	50	52%
	End user	No	108	43%
		Yes	145	57%
Firefox	Contributor	No	184	88%
		Yes	26	12%
	End user	No	139	94%
		Yes	9	6%
Thunderbird	Contributor	No	132	77%
		Yes	39	23%
	End user	No	129	95%
		Yes	7	5%

TABLE 6 The structure of training and testing data sets (total number of unique terms extracted from the summary of the bug reports [UT], % of initially provided features [IPF], and % of additionally provided features [APF])

	Camel			Derby			Wicket			Firefox			Thunderbird		
Features	UT	IPF	APF	UT	IPF	APF	UT	IPF	APF	UT	IPF	APF	UT	IPF	APF
CE	963	53%	4%	1040	40%	1%	931	60%	2%	825	10%	2%	847	15%	1%
TC	963	22%	7%	1040	46%	6%	931	33%	3%	825	37%	4%	847	26%	2%
STR	963	32%	4%	1040	32%	7%	931	23%	5%	825	51%	5%	847	44%	4%
ST	963	20%	3%	1040	33%	3%	931	9%	1%	825	13%	3%	847	15%	5%
EB	963	54%	1%	1040	46%	1%	931	60%	1%	825	68%	3%	847	71%	1%

initially provided features (IPF) in the description of bug reports, and percentage of additionally provided features (APF) in the comment section. We consider IPF + APF as the total required features for fixing a bug. Table 7 shows the median precision, recall, and f1-score of the key feature prediction for the various text-classification techniques for each project.

The f1-scores shown in bold represent the best f1-score among the various techniques for each project and each feature. In the case of Code Example, our models achieve the best f1-score of 0.70 with NB for the Wicket project. Comparing this result with the baseline model (0.55), we observe that our prediction model provides a 27% improvement. The best f1-scores for the Camel, Derby, Firefox, and Thunderbird projects are 0.65, 0.66, 0.29, and 0.37, respectively, with NBM, whereas the f1-scores of the baseline models are 0.53, 0.45, 0.19, and 0.24, respectively. Thus, our models achieve a 23% to 51% improvement over the baseline models for predicting Code Example. In the case of the Test Case prediction, our models achieve the best f1-score of 0.41 and 0.51 with NBM for the Camel and Firefox projects, 0.70 with NB for the Derby project, and 0.53 and 0.47 with KNN for the Wicket and Thunderbird projects. Comparing these results with those of the baseline models (0.37, 0.45, 0.51,0.42, and 0.36), we observe that our models achieve a 12% to 36% improvement over the baseline models. Similarly, our models achieve a 12% to 33% improvement for Steps to Reproduce, a 36% to 115% improvement for Stack Trace, and a 16% to 33% improvement in Expected Behavior over the baseline models in terms of the f1-score. Thus, the improvement of our models over the baseline models is substantial in terms of the f1-score.

Our best performing model achieves the lowest precision of 0.30 with SVM to predict the Stack Trace for the Wicket project. Comparing this result with the baseline model (0.10), we observe that our prediction model provides a 200% improvement. Similarly, NBM produces a 93% better precision than the baseline model for the Stack Trace prediction for the Camel project. Thus, the improvement of our models over the baseline models is substantial in terms of precision. In some cases, we find that the recall values of our best predictor are lower than the values of the baseline models. In practice, there is a trade-off between precision and recall. We can increase precision by sacrificing recall. The trade-off causes difficulties when comparing the performance of prediction models using precision or recall alone.[44] Thus, the f1-score, which is a trade-off between precision and recall, is used as the main metric to evaluate the performance of our prediction model and a random prediction. In very few cases, the f1-scores of our models are lower than those of the baseline models. For example, our model achieves a lower f1-score with KNN when predicting Expected Behavior for the Wicket project than that of the baseline model. This is because of low recall. However, in most cases, our prediction models achieve a much better f1-score compared with those of the baseline models. The f1-scores of our best-performing models range from 0.29 to 0.70 for Code Example, 0.41 to 0.70 for Test Case, 0.38 to 0.70 for Steps to Reproduce, 0.29 to 0.65 for Stack Trace, and 0.56 to 0.76 for Expected Behavior across the projects.

We notice that in some cases, our models achieve low f1-score to predict the key features. To get a better understanding of the prediction performance of our models, we calculate the prediction accuracy (ACC) and area under the receiver operator characteristic curve (AUC). Table 8 shows that our models achieve promising ACC and AUC values of different classification techniques to predict the key features for the different projects. For example, in the case of Code Example for the Derby project, our best model (NBM) achieves 78% accuracy score, which is 51% improvement over the baseline model. The AUC value is 0.85, which is 70% improvement over the baseline model. Table 7 shows that our models achieve the lowest f1-score for the Stack Trace of Wicket project. However, in this case, our models achieve better ACC and AUC values. The ROC curve in Figure 7 also shows that our models can predict the key features more accurately than the baseline model.

In order to investigate the best performing text-classification technique to build the key features prediction model, we conduct a comparative analysis of the four text-classification techniques. Figure 8 shows the distribution of the f1-scores among the different text-classification techniques across the projects. We see that our best-performing classifiers significantly outperform over the baseline classifier. However, there is no single best-performing classification technique for predicting key features. For example, in the case of Code Example, NBM performs better than other techniques for the Camel and Derby projects. NB performs better than the other techniques for the Wicket project, but the f1-score of NBM is very close to the f1-score of NB. In order to determine whether the difference is statistically significant, we conduct a statistical significance test between the different classification techniques. To perform this test, we first select the best-performing technique for each feature of each project. Then, we calculate the Mann-Whitney *U* test (also called Wilcoxon rank sum test) result for the best-performing technique compared with the others, one by one, to observe whether the *P* value is less than .05. Table 9 shows the significant test results between the best

TABLE 7 Precision (P), recall (R), and f1-score (F1) of different classification techniques and projects for the key features prediction

Code Example															
	Camel			Derby			Wicket			Firefox			Thunderbird		
Classifiers	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
NBM	0.67	0.63	**0.65**	0.64	0.67	**0.66**	0.68	0.63	0.65	0.23	0.39	**0.29**	0.31	0.45	**0.37**
KNN	0.65	0.42	0.51	0.48	0.81	0.61	0.66	0.45	0.54	0.15	0.65	0.24	0.30	0.41	0.34
NB	0.61	0.64	0.62	0.63	0.52	0.57	0.70	0.70	**0.70**	0.18	0.74	0.29	0.23	0.65	0.34
SVM	0.64	0.55	0.59	0.58	0.65	0.61	0.66	0.60	0.63	0.28	0.26	0.27	0.33	0.32	0.33
Mean	0.64	0.56	0.59	0.58	0.66	0.61	0.67	0.59	0.63	0.21	0.51	0.27	0.29	0.46	0.34
Baseline	0.57	0.50	0.53	0.41	0.50	0.45	0.62	0.50	0.55	0.12	0.50	0.19	0.16	0.50	0.24
Test Case															
NBM	0.37	0.45	**0.41**	0.64	0.64	0.64	0.42	0.49	0.45	0.50	0.52	**0.51**	0.43	0.45	0.44
KNN	0.33	0.52	0.41	0.59	0.43	0.50	0.45	0.66	**0.53**	0.49	0.52	0.50	0.41	0.52	**0.47**
NB	0.33	0.42	0.36	0.62	0.80	**0.70**	0.47	0.50	0.48	0.56	0.47	0.50	0.34	0.61	0.44
SVM	0.36	0.39	0.37	0.64	0.57	0.60	0.46	0.52	0.49	0.49	0.52	0.51	0.42	0.45	0.43
Mean	0.35	0.44	0.39	0.62	0.61	0.61	0.45	0.54	0.49	0.51	0.51	0.51	0.40	0.51	0.44
Baseline	0.29	0.50	0.37	0.52	0.50	0.51	0.36	0.50	0.42	0.41	0.50	0.45	0.28	50.00	0.36
Steps to Reproduce															
NBM	0.43	0.47	0.45	0.50	0.51	0.51	0.34	0.42	**0.38**	0.67	0.72	**0.70**	0.62	0.65	**0.64**
KNN	0.39	0.60	**0.47**	0.49	0.69	**0.57**	0.31	0.45	0.36	0.59	0.38	0.47	0.57	0.65	0.61
NB	0.42	0.41	0.41	0.49	0.37	0.41	0.31	0.39	0.35	0.60	0.80	0.68	0.66	0.49	0.56
SVM	0.41	0.49	0.45	0.46	0.52	0.49	0.32	0.36	0.34	0.66	0.58	0.62	0.59	0.66	0.62
Mean	0.41	0.49	0.44	0.49	0.52	0.49	0.32	0.40	0.36	0.63	0.62	0.62	0.61	0.61	0.61
Baseline	0.36	0.50	0.42	0.39	0.50	0.44	0.26	0.50	0.34	0.55	0.50	0.52	0.48	0.50	0.49
Stack Trace															
NBM	0.43	0.54	**0.48**	0.54	0.59	**0.57**	0.19	0.42	0.26	0.46	0.57	0.51	0.47	0.59	0.52
KNN	0.32	0.55	0.41	0.42	0.87	0.57	0.20	0.51	0.29	0.69	0.64	0.65	0.31	0.80	0.44
NB	0.35	0.38	0.36	0.52	0.55	0.54	0.13	0.40	0.19	0.22	0.80	0.34	0.29	0.77	0.42
SVM	0.43	0.49	0.47	0.51	0.65	0.57	0.30	0.29	**0.29**	0.69	0.62	**0.65**	0.62	0.60	**0.61**
Mean	0.38	0.49	0.43	0.50	0.66	0.56	0.20	0.40	0.26	0.51	0.66	0.54	0.42	0.69	0.50
Baseline	0.23	0.50	0.32	0.36	0.50	0.42	0.10	0.50	0.17	0.16	0.50	0.24	0.20	0.50	0.29
Expected Behavior															
NBM	0.62	0.61	0.62	0.54	0.49	0.52	0.64	0.61	0.63	0.77	0.76	**0.76**	0.76	0.74	0.75
KNN	0.54	0.21	0.31	0.50	0.62	0.55	0.70	0.16	0.26	0.76	0.51	0.61	0.79	0.35	0.48
NB	0.59	0.82	**0.68**	0.55	0.35	0.43	0.64	0.64	**0.64**	0.74	0.68	0.71	0.76	0.76	**0.76**
SVM	0.63	0.54	0.58	0.53	0.59	**0.56**	0.67	0.55	0.60	0.75	0.70	0.72	0.77	0.74	0.76
Mean	0.59	0.55	0.55	0.53	0.51	0.51	0.66	0.49	0.53	0.76	0.66	0.70	0.77	0.65	0.69
Baseline	0.55	0.50	0.52	0.47	0.50	0.48	0.61	0.50	0.55	0.68	0.50	0.58	0.71	0.50	0.59

classifier and other techniques for each key feature. We find that in some cases there are no best performing techniques for identifying key features. For example, there is no significant difference in performance between NBM and SVM for the Camel and Derby projects to identify Stack Trace. This indicates that we can choose either technique to build the prediction model. In some cases, we find that the best-performing technique (eg, NB for Test Case for the Derby project) significantly outperform the other techniques. In these cases, the best-performing technique is highly recommended for building the prediction model. For each feature, we identify the best-performing technique for each project. Thus, for five features, we identify 25 best-performing cases (see column 3 of Table 9). Of these, we find that NBM appears in 12 cases as the best-performing technique and four cases as no significant difference with the best-performing technique (see the highlighted cells in Table 9). Considering all cases, NBM outperforms the other techniques when predicting the key features.

TABLE 8 The accuracy (ACC) and area under the receiver operator characteristic curve (AUC) of different classification techniques and projects for the key features prediction

Code Example										
	Camel		Derby		Wicket		Firefox		Thunderbird	
Classifiers	ACC	AUC	ACC	AUC	ACC	AUC	ACC	AUC	ACC	AUC
NBM	0.68	0.68	0.78	0.85	0.64	0.66	0.80	0.75	0.77	0.80
KNN	0.57	0.61	0.61	0.66	0.56	0.63	0.55	0.71	0.78	0.71
NB	0.60	0.56	0.74	0.74	0.67	0.63	0.61	0.58	0.61	0.64
SVM	0.60	0.57	0.72	0.70	0.61	0.56	0.86	0.56	0.81	0.58
Mean	0.61	0.60	0.71	0.74	0.62	0.62	0.71	0.65	0.74	0.69
Baseline	0.51	0.50	0.52	0.50	0.53	0.50	0.79	0.50	0.73	0.50
Test Case										
NBM	0.66	0.69	0.65	0.70	0.64	0.68	0.68	0.72	0.76	0.73
KNN	0.56	0.62	0.59	0.63	0.65	0.70	0.66	0.69	0.74	0.74
NB	0.61	0.62	0.66	0.65	0.67	0.66	0.72	0.61	0.64	0.63
SVM	0.62	0.61	0.65	0.64	0.69	0.66	0.67	0.65	0.76	0.65
Mean	0.61	0.63	0.64	0.65	0.66	0.68	0.68	0.67	0.73	0.69
Baseline	0.59	0.50	0.50	0.50	0.54	0.50	0.52	0.50	0.60	0.50
Steps to Reproduce										
NBM	0.63	0.64	0.64	0.68	0.64	0.62	0.74	0.78	0.73	0.78
KNN	0.56	0.61	0.64	0.68	0.62	0.63	0.60	0.72	0.68	0.73
NB	0.66	0.62	0.65	0.61	0.61	0.52	0.68	0.69	0.72	0.69
SVM	0.61	0.59	0.57	0.55	0.68	0.59	0.70	0.68	0.70	0.70
Mean	0.62	0.61	0.62	0.63	0.64	0.59	0.68	0.72	0.71	0.73
Baseline	0.54	0.50	0.52	0.50	0.59	0.50	0.51	0.50	0.50	0.50
Stack Trace										
NBM	0.74	0.79	0.74	0.80	0.80	0.82	0.86	0.84	0.82	0.86
KNN	0.63	0.68	0.56	0.66	0.74	0.73	0.93	0.91	0.64	0.76
NB	0.68	0.66	0.72	0.70	0.70	0.59	0.56	0.68	0.62	0.59
SVM	0.77	0.75	0.71	0.70	0.87	0.69	0.93	0.84	0.88	0.84
Mean	0.70	0.72	0.68	0.72	0.78	0.71	0.82	0.81	0.74	0.76
Baseline	0.65	0.50	0.54	0.50	0.82	0.50	0.73	0.50	0.68	0.50
Expected Behavior										
NBM	0.64	0.69	0.63	0.67	0.65	0.65	0.76	0.77	0.73	0.71
KNN	0.54	0.59	0.56	0.60	0.52	0.64	0.63	0.63	0.55	0.67
NB	0.63	0.70	0.60	0.59	0.61	0.57	0.70	0.60	0.74	0.63
SVM	0.68	0.71	0.64	0.62	0.63	0.61	0.72	0.70	0.74	0.73
Mean	0.62	0.67	0.61	0.62	0.60	0.61	0.70	0.67	0.69	0.68
Baseline	0.51	0.50	0.50	0.50	0.52	0.50	0.56	0.50	0.59	0.50

5.2 | Discussion

One of the purposes of our quantitative analysis is to build an accurate prediction model to predict the key features based on the summary of the bug reports. Table 7 shows that the performance of the prediction models varies across the projects. For example, in the case of the Test Case prediction, our best-performing model achieves f1-score of 0.41 for the Camel and f1-score of 0.70 for the Derby project. One of the reasons can be the terms present in the description of the bug reports. To obtain deeper insight, we identify the top-10 discriminative terms for each key feature based on the correlation value. We can see from Tables 10 and 11 that the different application domain projects share the set of discriminative terms for each key feature. The correlation value of a particular term determines how much the summary of the bug report correlates

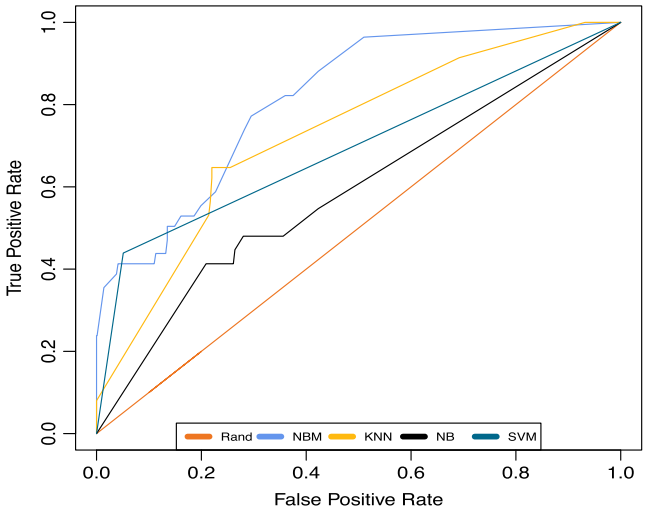


FIGURE 7 The Area under the receiver operator characteristic curve (ROC) for Stack Trace of Wicket project

with the feature. We see that the term "failure" appears in both the Camel and the Derby projects for the Test Case. We further perform a simple qualitative analysis to investigate each of the top-10 terms and examine how many bug reports (ie, the summaries of the bug reports) contain each of the top-10 discriminative terms in the training documents. We find that the term "failure" appears in 99 bug reports, or 28% of all bug reports for the Derby project. On average, 30 bug reports (the average is calculated as $\frac{1}{10}\sum_{i=1}^{i=10} x_i$, where x_i is the number of summaries that contain the ith term in the training documents) contain the top-10 discriminative terms in the Derby project. On the other hand, the term "failure" appears in only 24 bug reports, or 7% of all bug reports for the Camel project. On average, 13 bug reports contain the top-10 discriminative terms. We find a large difference between the average number of top-10 terms contained in the Derby project and those in the Camel project. Closer inspection reveals that the Derby project contains a large percentage of bug reports that are related to testing failures. Thus, reporters included the terms "failure" (99 times) and "test"(95 times) more often. This might be why the Derby project contains more of the top-10 discriminative terms, on average than the Camel project does. We see in Table 7 that the prediction performance of Test Case in terms of precision, recall, and f1-score is much better for the Derby project than for the Camel project. This indicates that the terms contained in the summaries of the bug reports might affect the performance of our prediction models.

In the case of Stack Trace, we find that the term "nullpointer" appears in 8%, 14%, and 5% of the summaries for the Camel, Derby, and Wicket projects, respectively. On average, 11, 29, and eight bug reports contain the top-10 discriminative terms in the summaries for the Camel, Derby, and Wicket projects, respectively. We find that our models achieve comparatively lower precision, recall, and f1-scores when predicting Stack Trace for the Wicket project than for the Derby project. We also find that a smaller percentage of bug reports share the top-10 discriminative terms in the Wicket project than in the Derby project. This indicates that the presence of the discriminative terms in the summary of the bug reports affect the performance of our prediction models.

We also find that, in some cases, reporters include very short summaries. For example, the description of the bug report[§§§§] contains Stack Trace, Test Case, and Expected Behavior. During the model evaluation (testing), our prediction model should correctly predict, Stack Trace, Test Case, and Expected Behavior for this bug report. The summary of this report is "MinaConsumerTest Failure," which contains only two terms, "MinaConsumerTest" and "failure." In Table 10, we find that the term "failure" appears among the top-10 most discriminative terms for Test Case. Thus, our model might predict Test Case because it is trained using the summaries of the bug reports. For Stack Trace and Expected Behavior, our models might provide an incorrect prediction because the summaries do not contain terms correlated with Stack Trace and Expected Behavior. In another example, the description of the bug report[¶¶¶¶] contains Steps to Reproduce, Stack Trace, and Code Example. Thus, during the model evaluation, our prediction model should predict these features correctly. The summary of the bug report is "NPE from came:run with below route codes." After removing stop words, the summary contains the terms "NPE (NullpointerExpection)," "Came," "run," "route," and "code." In Table 10, we find "NPE," "run," and "route" appear in the top-10 most discriminative terms for Stack Trace, Steps to Reproduce, and Code Example, respectively. That might help to predict these features. Therefore, the performance of our models might rely on the terms contained in the summaries of the bug reports.

To demonstrate the usefulness of our prediction models in practice, we introduce two cases from Camel project such as bug reports-CAMEL-4171[####] and CAMEL-2909.[‖‖‖‖] In the case of CAMEL-4171, the reporter, Sergey Zhemzhitsky, provided Observed Behavior, Stack Trace, and

[§§§§]https://issues.apache.org/jira/browse/CAMEL-795.

[¶¶¶¶]https://issues.apache.org/jira/browse/CAMEL-550.

[####]https://issues.apache.org/jira/browse/CAMEL-4171.

[‖‖‖‖]https://issues.apache.org/jira/browse/CAMEL-2909.

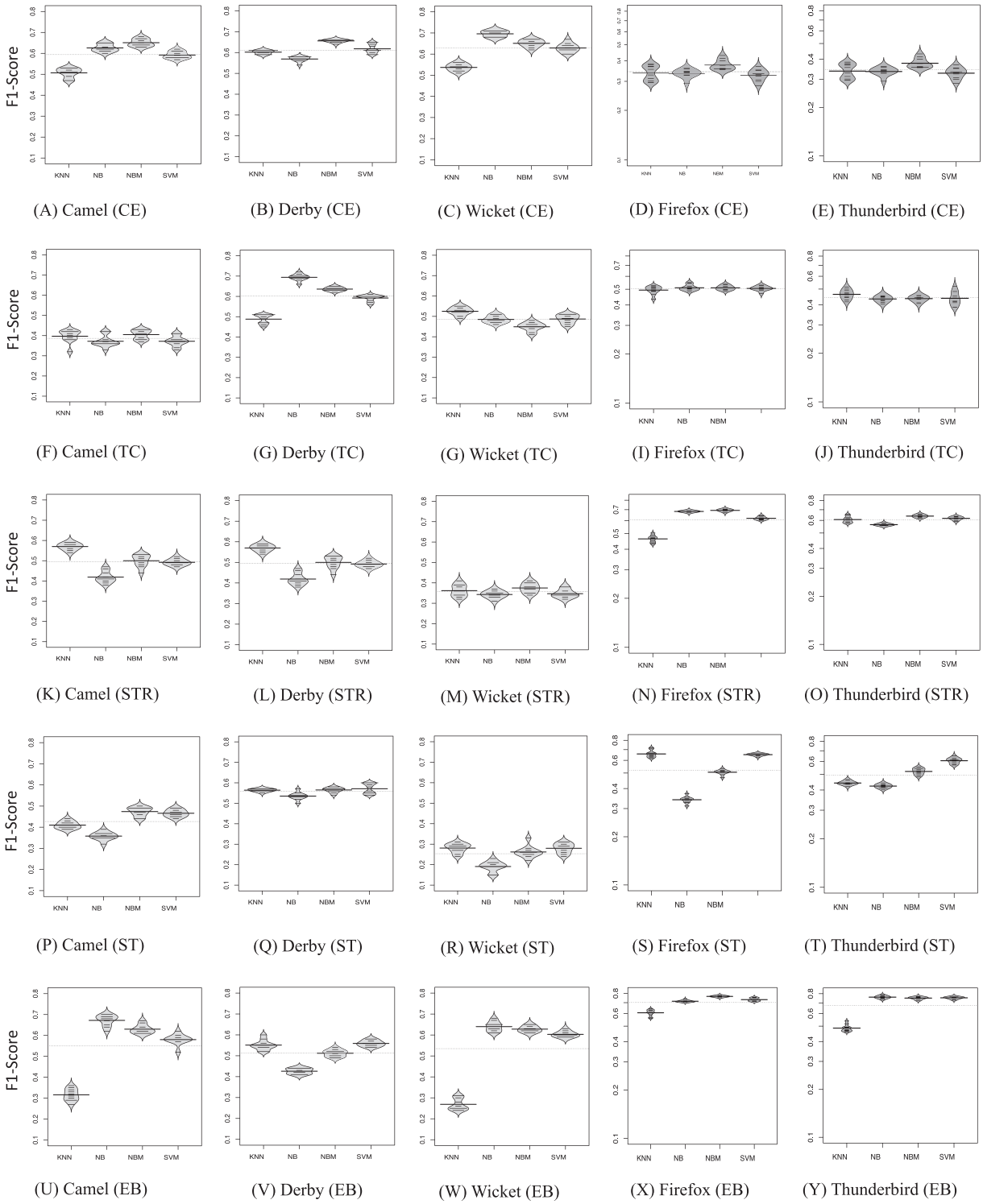


FIGURE 8 The difference in performance (f1-score) for predicting the key features among the different text-mining techniques across the projects

Code Example as the unstructured features in the description of the bug report. When this bug report was assigned to the developer, Claus Ibsen, to fix, he failed to reproduce the bug and asked for the additional feature. One day later, another developer, Freeman Fang, made a clarification question about the bug. Even after 4 months, the reporter did not respond with the requested feature. In the meantime, a new version was released. Then, the developer made a follow-up question, ie, "Any update? Did you try with a later release or created a unit test." Later, the bug was also identified in the new version. One year and 3 months later, the developer received the requested feature and finally resolved the bug. In the evaluation phase of our prediction model, we see that our model correctly predicted "Steps to Reproduce" as the required feature for this bug.

TABLE 9 Significant test result between the best classifier and other classifiers for each key feature.

			Other Classifiers				
Projects	**Features**	**Best Classifier**	**NBM**	**KNN**	**NB**	**SVM**	**Baseline**
	CE	**NBM**	–	***	*	***	***
	TC	**NBM**	–	–	**	**	***
Camel	STR	KNN	–	–	***	*	***
	ST	**NBM**	–	***	***	–	***
	EB	NB	***	***	–	***	***
	CE	**NBM**	–	***	***	***	***
	TC	NB	***	***	–	***	***
Derby	STR	KNN	***	–	***	***	***
	ST	**NBM**	–	–	**	–	***
	EB	SVM	***	–	***	–	***
	CE	NB	***	***	–	***	***
	TC	KNN	***	–	***	***	***
Wicket	STR	**NBM**	–	–	**	*	***
	ST	SVM	–	–	***	–	***
	EB	NB	–	***	–	**	***
	CE	**NBM**	–	***	–	–	***
	TC	**NBM**	–	–	–	–	***
Firefox	STR	**NBM**	–	***	*	***	***
	ST	KNN	***	–	***	–	***
	EB	**NBM**	–	***	***	***	***
	CE	**NBM**	–	*	***	***	***
	TC	KNN	*	*	–	–	***
Thunderbird	STR	**NBM**	–	*	***	**	***
	ST	SVM	***	***	***	–	***
	EB	NB	–	***	–	–	***

*P<.05

**P<.01

***P<.001

In the case of CAMEL-2909, the reporter, Max Matveev, submitted this report on 02 March 2012. When this bug report was assigned to the developer, Claus Ibsen to fix, he requested to the reporter to provide Stack Trace and Code Example as additional required features to fix the bug. In this case, our model correctly predicted Code Example successfully. However, our model made the wrong prediction for the Stack Trace.

Another purpose of our quantitative analysis is to determine the best-performing classification technique for building the prediction model. Our significance test (sec Table 9) shows there is no single best-performing technique. However, NBM performs comparatively better than the other techniques do. We do not know exactly why NBM outperforms the other techniques. One possible reason is our study design. We build prediction models based on the summaries of the bug reports. The summary contains a limited number of terms. SVM performs better in full-length documents, whereas NBM performs better in short documents.[49] NB assumes that each of the features is conditionally independent.[50] However, in reality, this assumption of independence is rarely true. Instead, NBM uses a multinomial distribution and works better than NB.[49,51] In spite of its better prediction performance, NBM is a simple and fast technique for training and testing the model.[50] The purpose of building key features prediction model is to develop an automated features recommendation tool that works in real time. Thus, a simple and quick technique is essential. Therefore, NBM may be effective in building a classification model for predicting key features.

Our quantitative analysis has shown that NBM outperforms over other classification techniques to predict the key features within-project setting. We want to further investigate whether our models with NBM can work for predicting the key features in the cross-project setting. To build prediction models in cross-project setting, we use one data set for the training the model and another data set for the testing the performance of the model.

TABLE 10 Top-10 discriminative terms based on the correlation value of each key feature for different projects

Projects	Features	Top-10 Discriminative Terms
	CE	camel (0.175), property (0.128), sftp (0.123), package (0.119), concurrentmodification (0.119), route (0.117), org (0.111), camelcontext (0.104), specific (0.104), configure (0.104)
	TC	window (0.152), failure (0.140), simple (0.131), problem (0.131), reference (0.131), raise (0.131), main (0.131), content (0.119), bean (0.115), procedure (0.109)
Camel	STR	bean (0.138), ftp (0.121), bug (0.117), run (0.117), read (0.117), issue (0.114), stream (0.114), fill (0.113), url (0.110), endpoint (0.105)
	ST	nullpointer (0.271), nullpointerexcept (0.179), configure (0.162), concurrentmodification (0.162), defaultcamelcontext (0.162), osg (0.153), provider (0.132), regression (0.132), version (0.132), header (0.113)
	EB	java (0.135), pipeline (0.120), attribute (0.120), throw (0.109), minimize (0.109), incorrect (0.108), content (0.108), example (0.104), run (0.104), match (0.104)
	CE	column (0.255), test (0.214), serial (0.192), failure (0.186), order (0.178), insert (0.172), clause (0.166), result (0.156), cause (0.155), apache (0.147)
	TC	language (0.167), make (0.164), failure (0.163), expect (0.141), correct (0.138), remove (0.138), test (0.133), found (0.133), drop (0.124), cause (0.116)
Derby	STR	nullpointer (0.160), select (0.141), insert (0.136), ignore (0.135), sql (0.121), distinct (0.121), expression (0.121), unexpect (0.117), derbynet (0.117), timestamp (0.107)
	ST	nullpointerexcept (0.229), incorrect (0.196), test (0.194), assertionfailederror (0.192), nullpointer (0.185), fail (0.178), ibm (0.163), wait (0.159), read (0.151), weme (0.138)
	EB	incorrect (0.180), nullpointerexcept (0.166), sql (0.146), server (0.133), derbytest (0.132), functiontest (0.132), include (0.124), privilege (0.124), java (0.111), example (0.111)
	CE	encode (0.134), object (0.130), tomcat (0.130), service (0.118), issue (0.118), filterpath (0.106), zip (0.106), double (0.106), log (0.105), catch (0.103)
	TC	wickettester (0.199), path (0.155), render (0.137), component (0.135), model (0.132), enclosure (0.122), datepicker (0.122), setenable (0.118), cookie (0.118), label (0.118)
Wicket	STR	modalwindow (0.218), before (0.164), pagelink (0.147), order (0.142), contribute (0.142), redirect (0.130), datepicker (0.130), call (0.121), iframe (0.116), clear (0.116)
	ST	debug (0.227), safari (0.227), nullpointer (0.224), service (0.201), child (0.176), java (0.176), tomcat (0.176), regression (0.161), reguestlogger (0.161), interrupt (0.161)
	EB	work (0.140), failure (0.138), redirect (0.127), resource (0.127), trigger (0.120), return (0.119), validatoradapter (0.105), issue (0.105), construct (0.109), call (0.099)

Table 12 shows the f1-scores of predicting the key features in the cross-project setting. We notice that in most of the cases, our models achieve in the cross-project setting only a bit worse than those achieved in the within-project setting. For example, the best f1-score of our models in the within-project for predicting Test Case of Camel project is 0.410 while in the cross-project setting the f1-score is 0.409 (eg, Camel->Derby setting). Therefore, we conclude that our models can work for predicting the key features in the cross-project setting.

In our qualitative analysis, we found that the features missing from the initial submission are unstructured features. Reporters provide these unstructured features in the descriptions of the bug reports as unstructured natural language text. Existing studies have revealed 10 unstructured features that are important to developers when fixing bugs. However, these features are not all equally important for all types of bug reports.[5,10,52] By examining the bug-fixing activities in our qualitative analysis, we know which unstructured features are required to fix each bug report. Thus, machine learning techniques could help to predict essential unstructured features when writing new bug reports by leveraging historical bug-fixing activities. Therefore, we motivate building a prediction model based on the summary text so that reporters of new bug reports might know which key features should be included. Thus, in order to help reporters, we build prediction models using the NB, NBM, KNN, and SVM text-classification techniques, based on the summary text. We evaluate our models using the bug reports of the Camel, Derby, Wicket, Firefox, and Thunderbird projects. Our models achieve promising f1-scores when predicting key features, except for Stack Trace of the Wicket project and Code Example of Firefox project. Given that our models are the first automated support of their kind, such performance can make a difference when writing bug reports.

6 | THREATS TO VALIDITY

In this section, we discuss some potential threats to the validity of our study, based on the guidelines proposed by Runeson and Höst.[53]

TABLE 11 Top-10 discriminative terms based on the correlation value for each key feature of the Camel project

Projects	Features	Top-10 Discriminative Terms
	CB	handler (0.231), javascript (0.201), bug (0.197), filter (0.188), cover (0.188), read (0.188), protocol (0.142), bind (0.142), event(0.142), html (0.138)
	TC	javascript (0.291), intermittent (0.261), browser (0.251), expect (0.213), chrome (0.192), uncaught (0.181), test (0.177), add(0.162), type (0.128), call (0.127)
Firefox	STR	browser (0.255), javascript (0.246), test (0.23), intermittent (0.226), content (0.18), remove (0.157), user (0.146), window (0.146), unexpect (0.139), perform (0.127)
	ST	browser (0.598), javascript (0.59), intermittent (0.576), test (0.391), expect (0.369), uncaught (0.351), crash (0.35), error (0.256), unexpect (0.243), np (0.23)
	EB	javascript (0.346), intermittent (0.344), browser (0.326), test (0.251), content (0.238), uncaught (0.216), error (0.205), chrome (0.191), build (0.180), perform (0.170)
	CE	bust (0.231), port (0.225), bug (0.204), source (0.188), define (0.188), http (0.188), script (0.188), error (0.161), folder (0.146), remove (0.145)
	TC	unexpect (0.350), fail (0.345), test (0.342), javascript (0.311), mozmil (0.283), xpcshel (0.243), toolkit (0.182), component (0.160), mail (0.157), content (0.157)
Thunderbird	STR	test (0.292), unexpect (0.251), javascript (0.234), xpcshel (0.174), mozmil (0.174), delete (0.173), filter (0.169), bug (0.167), fail (0.160), remove (0.149)
	ST	crash (0.448), test (0.393), mozmil (0.326), javascript (0.311), unexpect (0.304), fail (0.266), mail (0.211), dbview (0.175), init (0.175)
	EB	test (0.309), unexpect (0.266), javascript (0.256), fail (0.243), component (0.228), mozmil (0.223), crash (0.212), failure (0.168), build (0.166), error (0.145)

TABLE 12 F1-scores to predict the key features in the cross-project setting for the Apache projects

Projects	Code Example	Test Case	Steps to Reproduce	Stack Trace	Expected Behavior
Camel −> Derby	0.536	0.409	0.370	0.271	0.505
Camel −> Wicket	0.664	0.328	0.308	0.183	0.572
Derby −> Camel	0.485	0.444	0.395	0.271	0.495
Derby −> Wicket	0.496	0.475	0.374	0.301	0.590
Wicket −> Camel	0.647	0.311	0.402	0.137	0.609
Wicket −> Derby	0.573	0.375	0.382	0.274	0.536

External Validity: Threats to external validity relate to the generalizability of our results. We have studied five OSS projects from two ecosystems. Thus, our results may not be generalizable to all software systems especially proprietary systems. To combat the potential bias, we first followed strong selection criteria (see Section 3) to select the studied projects from two ecosystems such as Apache and Mozilla. This ensured that our case study included projects from different application domains. Second, we follow specific criteria (see Section 3) to select bug reports from each project in order to design data sets that exclude noisy and biased bug reports. The contents of bug reports might depend on the types of bug reporters (eg, experienced, nonexperienced reporters, contributor, and end user). Our data sets include a rational proportion of bug reports for each type of bug reporters, which might mitigate such a threat. Third, we use the standard sampling technique with a 95% confidence level to calculate the sample size. This ensured a rational subset for each studied project.

Internal Validity: Our concerns related to internal threats are the correctness of our manual analysis and experimental bias and errors. We manually analyzed the bug reports of five projects and identified frequently provided and additional required features for fixing bugs. Like any human activity, the features identification may be prone to human error or bias. To alleviate this threat, we double-check the identified features for all sampled bug reports. If there are differences in the identified features for the same bug report, we attempt to reach a consensus on the features. We also computed Cohen κ value to evaluate the interrater agreement. The Cohen κ values are 82%, 80%, 79%, 80%, and 87% for Camel, Derby, Wicket, Firefox, and Thunderbird, respectively, which showed excellent interrater agreement. However, there could still be errors that we may not notice. In our quantitative analysis, we constructed training and testing data sets with which to build and test our prediction models. We found that our data sets have a class imbalance problem. To mitigate this problem, we applied SMOTE, a popular class imbalance learning technique. In addition, to reduce the training data set selection bias, we applied 10-fold cross-validation and repeated the experiment

10 times to report the average performance. Our prediction models are trained and tested based using the summaries of the bug reports. Therefore, they rely on well-written summaries. An inaccurate summary may degrade the performance of our approach.

7 | CONCLUSION AND FUTURE WORK

Our goal is to understand the key features that reporters frequently omit from initial bug report submissions and to propose an approach that predicts whether reporters should provide certain features in their reports. To achieve this goal, we first perform a qualitative analysis of five OSS projects to investigate the key features of a bug report by examining bug-fixing activities. Then, we perform a quantitative analysis to develop an approach to support reporters while writing new bug reports. We build classification models to predict the key features using four popular text-classification techniques. Our models are trained using the summaries of the bug reports so that reporters may know which features to provide in the descriptions of new bug reports. Our qualitative analysis reveals that Steps to Reproduce, Test Case, Code Example, Stack Trace, and Expected Behavior are the additional required features that reporters most often omit from their initial submissions. We evaluate our prediction models using the bug reports of the Camel, Derby, Wicket, Firefox, and Thunderbird projects. Our models achieve the best f1-score for Code Example, Test Case, Steps to Reproduce, Stack Trace, and Expected Behavior of 0.7 (Wicket), 0.7 (Derby), 0.70 (Firefox), 0.65 (Firefox), and 0.76 (Firefox), respectively, which are promising. Our comparative study of the different classifications techniques reveals that NBM outperforms the other techniques when predicting key features. We also compare the performance of our models with the baseline models. The results show that our models provide a 12% to 115% improvement over the baseline models. In the future, we plan to valid our findings with more ecosystems' projects and advanced machine learning techniques. We also plan to do details impact analysis of each key feature on the bug-fixing process. We would also like to develop an approach to recommend features that reporters should provide in the description of bug reports based on types.

ACKNOWLEDGMENTS

This work was supported by the JSPS Program for the Grant-in-Aid for Young Scientists (B) (no. 16K16037) and JSPS KAKENHI grant number JP18H04094.

ORCID

Md. Rejaul Karim https://orcid.org/0000-0002-5532-2504

REFERENCES

1. Zimmermann T, Nagappan N, Guo PJ, Murphy B. Characterizing and predicting which bugs get reopened. In: Proceedings: 34th International Conference on Software Engineering , ICSE '12; 2012.
2. Just S, Premraj R, Zimmermann T. Towards the next generation of bug tracking systems. In: Proceedings: IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), ICSE '09; 2008.
3. Bettenburg N, Just S, Schroter A, Weiss C, Premraj R, Zimmermann T. What makes a good bug report? In: Proceedings: 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '08/FSE-16; 2008.
4. Breu S, Premraj R, Sillito J, Zimmermann T. Information needs in bug reports: improving cooperation between developers and users. In: Proceedings: 2010 ACM Conference on Computer Supported Cooperative Work, CSCW '10; 2010.
5. Davies S, Roper M. What's in a bug report? In: Proceedings: 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measuremen, ESEM '14; 2014.
6. Aranda J, Venolia G. The secret life of bugs: going past the errors and omissions in software repositories. In: 31st International Conference on Software Engineering, ICSE '09; 2009.
7. Dal Sasso T, Mocci A, Lanza M. What makes a satisficing bug report? In: Proceedings: IEEE International Conference on Software Quality, Reliability and Security (QRS), QRS '16; 2016.
8. Zhang T, Chen J, Jiang H, Luo X, Xia X. Bug report enrichment with application of automated fixer recommendation. In: Proceedings: 25th International Conference on Program Comprehension, ICPC '17; 2017.
9. Guo PJ, Zimmermann T, Nagappan N, Murphy B. Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In: Proceedings: 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10; 2010.
10. Karim Md. Rejaul, Ihara A, Yang X, Iida H, Matsumoto K. Understanding key features of high-impact bug reports. In: Proceedings: 8th International Workshop on Empirical Software Engineering in Practice, IWESEP '17; 2017.
11. Erfani Joorabchi M, Mirzaaghaei M, Mesbah A. Works for me! characterizing non-reproducible bug reports. In: Proceedings: 11th Working Conference on Mining Software Repositories, MSR '14; 2014.
12. Anvik J, Hiew L, Murphy GC. Who should fix this bug? In: Proceedings: 28th International Conference on Software Engineering, ICSE '06; 2006.
13. Canfora G, Cerulo L. Fine grained indexing of software repositories to support impact analysis. In: Proceedings: 2006 International Workshop on Mining Software Repositories, MSR '06; 2006.

14. Zhou J, Zhang H, Lo D. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In: Proceedings: 34th International Conference on Software Engineering, ICSE '12; 2012.
15. Weiss C, Premraj R, Zimmermann T, Zeller A. How long will it take to fix this bug? In: Proceedings: Fourth International Workshop on Mining Software Repositories, MSR '09; 2009.
16. Kim D, Tao Y, Kim S, Zeller A. Where should we fix this bug? A two-phase recommendation model. *IEEE Trans. Softw. Eng.* 2013;39(11).
17. Chaparro O, Lu J, Zampetti F, Moreno L, Di Penta M, Marcus A, Bavota G, Ng V. Detecting missing information in bug descriptions. In: Proceedings: 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE '17; 2017.
18. An open letter to GitHub from the maintainers of open source projects. available online: https://github.com/dear-github/dear-github; 2016.
19. Zimmermann T, Premraj R, Bettenburg N, Just S, Schroter A, Weiss C. What makes a good bug report? *IEEE Transactions on Software Engineering*. 2010;36(5).
20. Bettenburg N, Premraj R, Zimmermann T, Kim S. Extracting structural information from bug reports. In: Proceedings: 2008 International Working Conference on Mining Software Repositories, MSR '08; 2008.
21. Wang X, Zhang L, Xie T, Anvik J, Sun J. An approach to detecting duplicate bug reports using natural language and execution information. In: Proceedings: 30th International Conference on Software Engineering, ICSE '08; 2008.
22. Rocha H, Valente M, Marques-Neto H, Murphy GC. An empirical study on recommendations of similar bugs. In: Proceedings: 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER '16; 2016.
23. Yang X-L, Lo D, Xia X, Huang Q, Sun J-L. High-impact bug report identification with imbalanced learning strategies. *Computer Science and Technology*. 2017;32.
24. Xia X, Lo D, Qiu W, Wang X, Zhou B. Automated configuration bug report prediction using text mining. In: Proceedings: IEEE 38th Annual International Computers, Software and Applications Conference, COMPSAC '14; 2014.
25. Sun C, Lo D, Wang X, Jiang J, Khoo C. A discriminative model approach for accurate duplicate bug report retrieval. In: Proceedings: 32Nd ACM/IEEE International Conference on Software Engineering, ICSE '10; 2010.
26. Rensis L. A technique for the measurement of attitudes; 1932.
27. Ko AJ, Myers BA, Chau DH. A linguistic analysis of how people describe software problems. In: Proceedings: Visual Languages and Human-Centric Computing, VLHCC '06; 2006.
28. Hooimeijer P, Weimer W. Modeling bug report quality. In: Proceedings: Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ASE '07; 2007.
29. Kashiwa Y, Yoshiyuki H, Kukita Y, Ohira M. A pilot study of diversity in high impact bugs. In: Proceedings: 30th International Conference on Software Maintenance and Evolution, ICSME '14; 2014.
30. Ohira M, Hassan AE, Osawa N, Matsumoto K. The impact of bug management patterns on bug fixing: a case study of eclipse projects. In: Proceedings: 28th IEEE International Conference on Software Maintenance, ICSM '12; 2012.
31. Ohira M, Kashiwa Y, Yamatani Y, Yoshiyuki H, Maeda Y, Limsettho N, Fujino K, Hata H, IharaA, Matsumoto K. A dataset of high impact bugs: manually-classified issue reports. In: Proceedings: 12th Working Conference on Mining Software Repositories, MSR '15; 2015.
32. Gegick M, Rotella P, Xie T. Identifying security bug reports via text mining: an industrial case study. In: Proceedings: 7th Working Conference on Mining Software Repositories, MSR '10; 2010.
33. Nistor A, Jiang T, Tan L. Discovering, reporting, and fixing performance bugs. In: Proceedings: 10th Working Conference on Mining Software Repositories, MSR '13; 2013.
34. Shihab E, Mockus A, Kamei Y, Adams B, Hassan AE. High-impact defects: a study of breakage and surprise defects. In: Proceedings: 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11; 2011.
35. Chen T-H, Nagappan M, Shihab E, Hassan AE. An empirical study of dormant bugs. In: Proceedings: 11th Working Conference on Mining Software Repositories, MSR '14; 2014.
36. Valdivia Garcia H, Shihab E. Characterizing and predicting blocking bugs in open source projects. In: Proceedings: 11th Working Conference on Mining Software Repositories, MSR '14; 2014.
37. Karim MR, Farid DM. An adaptive ensemble classifier for mining complex noisy instances in data streams. In: Proceedings: 2014 International Conference on Informatics, Electronics Vision, (ICIEV); 2014.
38. McCallum A, Nigam K. A comparison of event models for naive Bayes text classification. *in AAAI-98 Workshop on Learning for Text Categorization*. 1998;752.
39. Wu X, Kumar V, Ross Quinlan J, Ghosh J, Yang Q, Motoda H, McLachlan GJ, Ng A, Liu B, Yu PS, Zhou Z-H, Steinbach M, Hand DJ, Steinberg D. Top 10 algorithms in data mining. *Knowl. Inf. Syst.* 2007;14(1).
40. Xin X, David L, Emad S, Xinyu W, Xiaohu Y. Elblocker: predicting blocking bugs with ensemble imbalance learning. *Information and Software Technology*. 2015;61.
41. Cubranic D. Automatic bug triage using text categorization. In: Proceedings: Sixteenth International Conference on Software Engineering and Knowledge Engineering, SEKE '04; 2004.
42. Kamei Y, Monden A, Matsumoto S, Kakimoto T, ichi Matsumoto K. The effects of over and under sampling on fault-prone module detection. In: Proceedings: First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007), ESEM '07; 2007.
43. Wang S, Yao X. Using class imbalance learning for software defect prediction. *IEEE Trans. Reliability*. 2013;62.

44. Han J, Kamber M. *Data Mining: Concepts and Techniques*. O'Reilly Media, Inc.; 2009.
45. Zhou M, Mockus A. Who will stay in the FLOSS community? Modeling participants' initial behavior. *IEEE Transactions on Software Engineering*. 2015;41.
46. Ghotra B, Mcintosh S, Hassan AE. A large-scale study of the impact of feature selection techniques on defect classification models. In: Proceedings: 14th International Conference on Mining Software Repositories, MSR '17; 2017.
47. Rodeghero P, Huo D, Ding T, McMillan C, Gethers M. An empirical study on how expert knowledge affects bug reports. *Journal of Software Evolution and Process*. July 2016;28(7).
48. Zaman S, Adams B, Hassan AE. Security versus performance bugs: a case study on Firefox. In: Proceedings: 8th Working Conference on Mining Software Repositories, MSR '11; 2011.
49. Wang S, Manning ChristopherD. Baselines and bigrams: simple, good sentiment and topic classification. In: Proceedings: 50th Annual Meeting of the Association for Computational Linguistics: Short Papers - Volume 2, ACL '12; 2012.
50. Russell J. S, Norvig P. Artificial intelligence: a modern approach (2nd). *Artificial Intelligence and Machine Learning Book*. 2003.
51. Rennie JasonD. M., Shih L, Teevan J, Karger DavidR. Tackling the poor assumptions of naive Bayes text classifiers. In: Proceedings: Twentieth International Conference on International Conference on Machine Learning, ICML '03; 2003.
52. Yusop NSM, Grundy J, Vasa R. Reporting usability defects: do reporters report what software developers need? In: Proceedings: 20th International Conference on Evaluation and Assessment in Software Engineering, EASE '16; 2016.
53. Runeson P, Höst M. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*. 2009;14(2).

How to cite this article: Karim M, Ihara A, Choi E, Iida H. Identifying and Predicting Key Features to Support Bug Reporting. *J Softw Evol Proc*. 2019;e2184. https://doi.org/10.1002/smr.2184