Singapore Management University

Institutional Knowledge at Singapore Management University

Research Collection School Of Information Systems | School of Information Systems

8-2018

Chaff from the wheat: Characterizing and determining valid bug reports

Yuanrui FAN
Zhejiang University

Xin XIA
Zhejiang University

David LO
Singapore Management University, davidlo@smu.edu.sg

Ahmed E. HASSAN
Queen's University - Kingston, Ontario

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

 Part of the Databases and Information Systems Commons, and the Software Engineering Commons

Citation

FAN, Yuanrui; XIA, Xin; LO, David; and HASSAN, Ahmed E.. Chaff from the wheat: Characterizing and determining valid bug reports. (2018). *IEEE Transactions on Software Engineering*. 1-30. Research Collection School Of Information Systems.
Available at: https://ink.library.smu.edu.sg/sis_research/4103

This Journal Article is brought to you for free and open access by the School of Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email library@smu.edu.sg.

Published in IEEE Transactions on Software Engineering,
August 2018,
DOI 10.1109/TSE.2018.2864217

Chaff from the Wheat: Characterizing and Determining Valid Bug Reports

Yuanrui Fan, Xin Xia, David Lo, Ahmed E. Hassan

Abstract—Developers use bug reports to triage and fix bugs. When triaging a bug report, developers must decide whether the bug report is *valid* (i.e., a real bug). A large amount of bug reports are submitted every day, with many of them end up being *invalid* reports. Manually determining *valid* bug report is a difficult and tedious task. Thus, an approach that can automatically analyze the validity of a bug report and determine whether a report is *valid* can help developers prioritize their triaging tasks and avoid wasting time and effort on *invalid* bug reports.

In this study, motivated by the above needs, we propose an approach which can determine whether a newly submitted bug report is *valid*. Our approach first extracts 33 features from bug reports. The extracted features are grouped along 5 dimensions, i.e., reporter experience, collaboration network, completeness, readability and text. Based on these features, we use a random forest classifier to identify *valid* bug reports. To evaluate the effectiveness of our approach, we experiment on large-scale datasets containing a total of 560,697 bug reports from five open source projects (i.e., Eclipse, Netbeans, Mozilla, Firefox and Thunderbird). On average, across the five datasets, our approach achieves an F1-score for *valid* bug reports and F1-score for *invalid* ones of 0.74 and 0.67, respectively. Moreover, our approach achieves an average AUC of 0.81. In terms of AUC and F1-scores for *valid* and *invalid* bug reports, our approach statistically significantly outperforms two baselines using features that are proposed by Zanetti et al. [99]. We also study the most important features that distinguish *valid* bug reports from *invalid* ones. We find that the textual features of a bug report and reporter's experience are the most important factors to distinguish *valid* bug reports from *invalid* ones.

Index Terms—Bug Report, Feature Generation, Machine Learning

1 Introduction

Software projects use issue tracking systems (e.g., Bugzilla[1] and JIRA[2]) to manage the process of bug reporting, assignment, tracking, resolution and archiving [6]. Due to the large amount of bug reports in software projects, bug report management is a challenging work. A number of automated bug report management techniques have been proposed. These techniques include bug assignee recommendation [4], [96], reopened bug prediction [71], [95], [103], bug fixing time prediction [11], [100], blocking bug prediction [83], [94], duplicate bug report detection [75], [84] and bug severity/priority assignment [47], [81].

In a typical bug triaging and fixing process, a developer/tester or an end user detects a bug, then he/she reports the bug to an issue tracking system. Next, bug triagers manually judge whether the bug is a *valid* bug (i.e., a real bug which contains complete description and can be reproduced). After that, if the bug is *valid*, the bug is assigned to the most appropriate developer to fix [4], [62]. In practice, a large number of bug reports are submitted every day in large-scale projects. For example, in Mozilla[3], on average, ∼307 new bug reports are submitted per day. Thus, manually determining the validity of a bug is time-consuming and tedious.

Invalid bugs (i.e., bugs that are not real bugs or cannot be reproduced) increase the difficulty of bug triaging. For instance, in our bug report datasets, 22% to 79% the of bug reports have a resolution of *DUPLICATE* (i.e., reported bug is a duplicate of a known bug), *INVALID* (i.e., reported bug is not a software defect), *WORKSFORME* (i.e., reported bug cannot be reproduced) or *INCOMPLETE* (i.e., the report lacks sufficient information to reproduce, e.g., no steps to reproduce).

Considering the substantial time and effort that is needed to determine the validity of a bug, an approach which determines early on (i.e., when a bug report is submitted) whether a bug report is *valid*, can help developers prioritize their triaging tasks. In such a case, developers can make better use of their limited resources so that they can focus their efforts on *valid* bugs. Furthermore, a deeper understanding of the factors that lead to *invalid* bugs might help in developing tools and guidelines that can assist bug reporters in submitting well written *valid* bug reports.

In this study, we focus on analyzing bug reports in Bugzilla, which is a popular issue tracking system. Bugzilla uses two fields to track a bug report's development process—namely status and resolution. The status of a bug report describes the current developement state of the bug. The resolution of a bug report describes how has this bug been resolved. Following Zanetti et al.'s study [99], we define a *valid* bug report as a bug report whose resolution is *FIXED* (i.e., reported bug is eventually fixed) or *WONTFIX* (i.e., reported bug is an actual bug but will not be fixed due

- *Yuanrui Fan and Xin Xia are with the College of Computer Science and Technology, Zhejiang University, Hangzhou, China. E-mail: yrfan@zju.edu.cn, xxia@zju.edu.cn*
- *David Lo is with the School of Information Systems, Singapore Management University, Singapore. E-mail: davidlo@smu.edu.sg*
- *Ahmed E. Hassan is with School of Computing, Queen's University, Canada. E-mail: ahmed@cs.queensu.ca*
- *Xin Xia is the corresponding author.*

1. http://www.bugzilla.org/
2. http://www.atlassian.com/JIRA/
3. For more details about the dataset, please refer to Section 3.1

to, for example, lack of resources or low priority). We define an *invalid* bug report as a bug report whose resolution is *DUPLICATE*, *INVALID*, *WORKSFORME* or *INCOMPLETE*. Note that in our definition, *invalid* bug reports include bug reports whose resolution is *WORKSFORME*, while Zanetti et al. did not define these bug reports as *valid* or *invalid* bug reports, i.e., they dropped such bugs in their analysis.

In this paper, we propose an approach to determine early on whether a bug report is *valid* or *invalid*. To do so, we extract 33 features from bug reports which are grouped along five dimensions: reporter experience, collaboration network, completeness, readability and text. In the reporter experience dimension, we mine the history of bug report submissions and updates, and we extract features based on the previous behaviors of the reporter of a bug report, e.g., the number of prior bug reports submitted by the bug reporter. In the collaboration network dimension, we first extract the creation time of the bug report, and construct a network based on the collaborations of reporters and developers (i.e., developers' comments on bug reports) in the preceding month of the creation time. Then we extract degree and centrality features of the reporter's node of that network. In the completeness dimension, we analyze the description of the bug report and extract features by checking whether the description contains needed information for developers, e.g., whether the description of the bug report contains stack traces or code samples. In the readability dimension, we use readability measures that are calculated based on the description of the bug report such as *Flecsh* and *Lix* scores of the description [3], [22], [26], [45], [53], [69]. In the text dimension, we first extract token features from summary and description of bug reports, then we apply four types of classifiers to convert the token features into numerical scores. Based on the extracted 33 features, we use random forest [15] to build models that can effectively determine whether a bug report is *valid* or not.

Zanetti et al.'s study [99] is the most related to ours. For each bug report, they extracted its creation time and constructed a collaboration network based on the *ASSIGN* and *CC* relations of bug reports in the preceding month of the creation time—an *ASSIGN* relation is formed when a user assigns a bug to another user, and a *CC* relation is formed when a user adds another user into the cc list of a bug report. Then, they proposed nine features to be extracted from the network. Based on these features, they built models using a Support Vector Machine (SVM) classifier to determine whether a bug report is *valid* or not. Our work is related to theirs but there are differences between our work and theirs:

1) Our datasets are more complete than Zanetti et al.'s. We analyze bug reports whose status is *CLOSED* or *VERIFIED*, while they did not analyze these bug reports. Moreover, Zanetti et al. also dropped bug reports whose resolution is *WORKSFORME*. In our bug report datasets, up to 43% of the bug reports have a status of *CLOSED* or *VERIFIED*, and up to 31% of *invalid* bug reports have a resolution of *WORKSFORME*. Zanetti et al.'s study excluded a large number of bug reports, which might introduce a threat to the validity of their study.
2) We extract more domain-specific features from bug reports—we not only extract features from the collaboration network, but also derive features from the reporter experience and bug report contents.

Zanetti et al. used SVM as the default underlying classifier of their models [99], and in this paper, we use random forest as the default underlying classifier of our models. To investigate whether our additional features improve the effectiveness of our models, we make a comparison with two baselines based on Zanetti et al.'s features. We refer to them as SVMZ and RFZ. The last letter "Z" in the names indicates that the two baselines are both using **Z**anetti et al.'s proposed features. SVMZ uses **SVM** as its underlying classifier, while RFZ uses random forest (**RF**) as its underlying classifier.

To evaluate the performance of our approach, we collect bug report datasets from five large open source projects—namely Eclipse, Netbeans, Mozilla, Firefox and Thunderbird, which contains a total of 560,697 bug reports. Experimental results show that across the five datasets, our approach achieves an average F1-score for *valid* and *invalid* bug reports of, 0.74 and 0.67, respectively. And our approach achieves an average AUC (Area Under the receiver operator characteristic Curve [38]) of 0.81. Across the five datasets, our approach statistically significantly outperforms SVMZ and RFZ in terms of AUC and F1-scores for *valid* and *invalid* bug reports. Of the 33 features that we extract, we find that *desc-dmnb-score*, *valid-rate* and *summary-dmnb-score* are the most important features to distinguish *valid* bug reports from *invalid* ones across the five studied datasets. Of these three features, *desc-dmnb-score* and *summary-dmnb-score* are textual features that we extract based on the summary and description of bug reports, and *valid-rate* is a feature that is used to quantify reporters' experience.

The main contributions of this paper are:

- We propose an approach which includes a total of 33 features to determine the validity of a bug report. We extract features from collaboration network, reporting history and bug report contents. We experiment on a broad range of datasets containing a total of 560,697 bug reports from five large-scale projects. Our experimental results show that our approach outperforms the baselines by a substantial margin.
- We investigate the most important characteristics that impact the validity of a bug report. Our experimental results show that the textual features of a bug report and reporter's experience are the most important factors that determine whether a bug report is *valid* or not.

Paper Organization. The remainder of this paper is organized as follows. Section 2 presents the usage scenario of our approach when it is integrated with Bugzilla. Section 3 presents the research questions that we investigate in this study, elaborates the details of our data collection process, and describes our experiment setup. Section 4 presents experimental results and answers to our research questions. Section 5 discusses performance of our approach and the baselines in various settings and threats to validity. Section 6 briefly reviews prior studies related to ours. Section 7 concludes the paper and discusses possible

avenues for future work.

2 Usage Scenario

Our approach can be integrated with Bugzilla as follows:

Without Our Approach. Bob is a bug triager in a large software project using Bugzilla as its issue tracking system. Daily, Bob receives more than 100 bug reports to be triaged and this is too many for him to process in a single day. Without our approach, he randomly selects bug reports to triage. He reads each selected bug report, checks whether it contains sufficient information and determines its validity. If he finds that the bug report does not contain sufficient information or he cannot decide its validity, he contacts the reporter for more information. Due to the large number of received bug reports daily, his processing of reports is often slow. Most of his processed reports are eventually determined as *invalid*. These reports waste much of Bob's time and effort, resulting in Bob having less time to triage other bug reports.

With Our Approach. Bob's project adopts our approach. When Bob needs to triage many bug reports in a single day, he first uses our approach to determine the validity of these bug reports. Then, Bob triages the bug reports that are most likely to be *valid*. In this case, Bob is able to prioritize his triaging tasks and pay more attention on bug reports that are more likely to be *valid*. And he can spend less time and effort on *invalid* bug reports. Additionally, for bug reports receiving a very low ranking score, those reports are sent back to the reporters requesting them to provide more information. This automated feedback can be done as soon as a bug report is received; the reporters would thus still have the bug triggering condition fresh in their minds, are still interested in the bug, and thus are more likely to respond. In such a way, some reports that would otherwise be deemed *invalid* can potentially be salvaged. In summary our proposed approach helps 1) triagers by reducing wasted time on *invalid* bug reports and 2) reporters by informing them immediately that their report is likely to be marked as *invalid* (hence they can enhance their reports while the reported problems are still fresh in their mind).

3 Experiment Setup

In this section, we first introduce our data collection process. Then we elaborate the details of 33 features that we extract from bug reports. The features are grouped along 5 dimensions: reporter experience, collaboration network, completeness, readability, and text. After that, we present the classifiers (i.e., random forest and SVM) that we use in this study. Next, we present our evaluation setup and our baselines, including two baselines that are based on features of Zanetti et al. [99]. Finally, we present the evaluation measures that we use to evaluate the classification performance of our approach and the baselines. In this paper, we would like to answer following four research questions:

RQ1 Can we effectively determine the validity of a bug report?

RQ2 How effective is our approach when all features are used than when a single dimension of features is used?

RQ3 Which features are most important for differentiating *valid* bug reports from *invalid* ones?

RQ4 How effective is our model when built on a subset of our features?

3.1 Data Collection

In Bugzilla, each bug report contains a number of fields (e.g., bug id, creation time, reporter, status and resolution). All updates of these fields are stored in the database of Bugzilla. Each update record contains its time stamp, the new field value, the original field value and the user who has changed the field. After a reporter submits a bug report, developers and the reporter can give comments on the bug report and each comment is recorded along with its time stamp and author. The comments of bug reports are also stored in the database of Bugzilla.

In this study, we collect all fields and comments of bug reports as well as full history of bug report updates stored in the Bugzilla system of Eclipse, Netbeans, Mozilla, Firefox and Thunderbird. We retrieve all the data using the Python package **python-bugzilla**[4]. This package can crawl bug reports data from Bugzilla via the Bugzilla's Webservice API[5]. Note that our goal is to determine the validity of an initially submitted bug report. In such a case, we should only use the initial field values of bug reports, i.e., the field values of bug reports when they are submitted. Bugzilla keeps the track of bug report updating history. To simulate the practical usage of our approach, for bug reports in the testing set, we use their updating histories to recover the initial field values.

In the Bugzilla system, the status of a bug report can be *NEW*, *UNCONFIRMED*, *ASSIGNED*, *REOPENED*, *CLOSED*, *RESOLVED*, or *VERIFIED*. Developers can reopen bug reports and change their status and resolution. Thus, status and resolution of a bug report may be changed many times in the bug report updating history. In this study, we focus on the final status and resolution of bug reports until the time we collected our data. Note that our bug report datasets are collected on May 2017. And we focus on the bug reports whose final status is *CLOSED*, *RESOLVED* or *VERIFIED*. For these bug reports, we extract their final resolution and label them as *valid* or *invalid* according to the definition of *valid* and *invalid* bug reports presented in Section 1.

All the five projects we study have large bug repositories and provide ample data for our evaluation experiment. Following Sun et al.'s study [76], we select a subset of each repository to set up bug report datasets for our evaluation experiment. Table 1 presents the statistics of the selected data. From Table 1, we find that the class distributions of different projects are different. For example, 78% of bug reports in the Eclipse dataset are *valid*, while 21% of bug reports in the Thunderbird dataset are *valid*.

A bug report may be reopened and its validity can be changed, which may make labels of bug reports in our datasets unreliable. For example, in Eclipse, the bug report whose bug id is 219748[6] preliminarily got a resolution of

4. https://github.com/python-bugzilla/python-bugzilla
5. https://goo.gl/XzaRby
6. https://bugs.eclipse.org/bugs/show-bug.cgi?id=219748

TABLE 1
Statistics of the selected data.

Project	Time Period	# Bug Report	# Valid	# Invalid
Eclipse	2010.1–2014.12	121,855	95,565 (78%)	26,290 (22%)
Netbeans	2010.1–2014.12	63,621	39,197 (62%)	24,424 (38%)
Mozilla	2013.1–2014.12	224,408	155,612 (69%)	68,796 (31%)
Firefox	2002.4–2014.12	126,967	28,603 (23%)	98,364 (77%)
Thunderbird	2000.1–2014.12	33,846	7,100 (21%)	26,746 (79%)
Total		**570,697**	**326,077 (57%)**	**244,620 (43%)**

INVALID. However, after a few days, the bug report was reopened and the resolution field was changed to *FIXED*. We perform an analysis on the threat to the labels of our datasets that is introduced by reopening bug reports. Notice that we focus on the final status and resolution of bug reports until the time that we collected our data (i.e., May 2017) in our study. We find that for 98%, 96%, 97% , 99% and 98% of the bug reports in our selected Eclipse, Netbeans, Mozilla, Firefox and Thunderbird datasets, their status and resolution have remained unchanged during the period between May 2016 (i.e., one year earlier) and the time that we collected our datasets (i.e., May 2017), respectively. It means that for at least one year, these bug reports have not been reopened, i.e., most of the bug reports in our datasets are unlikely to be reopened. Thus, labels of the bug reports in our datasets are reliable.

As mentioned earlier, our datasets are different from Zanetti et al.'s [99]. In our study, we analyze the bug reports whose final status is *CLOSED* or *VERIFIED*, while Zanetti et al. dropped such data. In our datasets, 6%–43% of the bug reports have a final status of *CLOSED* or *VERIFIED*. In contrast to Zanetti et al.'s study, we analyze a new type of bug reports that have a final resolution of *WORKSFORME*. These bug reports are determined to be non-reproducible by bug triaggers. We label reports with a final resolution of *WORKSFORME* as *invalid*. And we find that 21%–31% of *invalid* bug reports in the five datasets have a final resolution of *WORKSFORME*.

3.2 Studied Features

We extract 33 features which can potentially impact the validity of bug reports and differentiate *valid* bug reports from *invalid* ones. Table 2 summarizes the set of 33 features, which are grouped along 5 dimensions: reporter experience, collaboration network, completeness, readability and text.

We extract some features (e.g., reporter experience) based on all bug reports recorded in their issue tracking systems since we need to mine full history of bug report submissions and updates. Also, we find reporters usually comment on their bug reports immediately after they initially submit the bug reports. For example, in a bug report from Eclipse whose bug id is 116691[7], the reporter added a comment on his report in a few minutes after he initially reported the bug. We find that he attached a file in the comment. Ignoring the comment will cause some information loss about the bug report. Zimmermann et al. analyzed the attachments within 15 minutes after the creation of bug report [104]. Following their study, we use the same time window. When we extract completeness, readability and textual features from bug reports, we consider description of a bug report including not only its initial description submitted by the reporter, but also the comments which he/she adds in 15 minutes after the creation time of the bug report. In a bug report, each comment has its time stamp and author. We extract reporters' comments of bug reports and retrieve the comments that we need by comparing the time stamps of the comments and the creation time of bug reports.

To determine whether the features that we extract make sense for identifying *valid* bug reports, we randomly exclude 2,000 bug reports from each dataset. We have five datasets, and thus, in total, we have 10,000 bug reports. We find that 51% of these bug reports are *valid*. We denote these 10,000 bug reports as the mini-dataset. For each feature, we use the Wilcoxon rank-sum test [52] to analyze the statistical significance of the difference between *valid* and *invalid* bug reports in the mini-dataset (p-value < 0.05). And we compute Cliff's delta[8] [21]. Cliff's delta [21] is a non-parametric effect size measure that can evaluate the amount of difference between two variables. A higher level for a feature with a positive effect increases the likelihood of a bug report being *valid*, while a higher level for a feature with a negative effect decreases the likelihood of a bug report being *valid*.

Section 4.3 performs a more thorough analysis on the importance of our features in identifying *valid* bug reports for each project. The analysis in Section 4.3 considers the combinations of features, in contrast to the simplified per-feature analysis that we just presented.

Reporter Experience Dimension refers to features that are based on the experience of a bug reporter in his/her bug handling community. Developers' experience can influence their contributions to projects [46]. Just et al. observed that experienced reporters are better at providing information that is needed by for bug fixing [43]. Zimmermann et al. noted that inexperienced reporters are likely to submit duplicate bug reports (a type of *invalid* reports) [104]. Thus, we expect that a reporter's experience can help determine the possible validity of a bug report. We use three features to quantify reporters' experience—namely *bug-num*, *valid-rate* and *recent-bug-num*.

Intuitively, the more bug reports a reporter submits, he/she should be more experienced. Thus, we use the *bug-num* feature to quantify the experience of bug reporters.

7. https://bugs.eclipse.org/bugs/show-bug.cgi?id=116691

8. Cliff defines a delta of less than 0.147, between 0.147 and 0.33, between 0.33 and 0.474 and above 0.474 as negligible, small, medium, large effect size, respectively.

TABLE 2
Studied Features.

Dimension	Feature Name	Description
Reporter Experience	bug-num	Number of prior bug reports submitted by the reporter of this bug report
	recent-bug-num	Number of prior bug reports submitted by the reporter of this bug report in 90 days
	valid-rate	Valid rate of prior bug reports with known labels submitted by the reporter of this bug report
Collaboration Network	lcc-membership	These metrics are used to quantify a bug reporter's degree of activity in his/her bug handling community [99]
	in-degree	
	out-degree	
	total-degree	
	clustering-coefficient	
	k-coreness	
	closeness-centrality	
	betweenness-centrality	
	eigenvector-centrality	
Completeness	has-stack	Whether description of this bug report contains stack traces
	has-step	Whether description of this bug report contains steps to reproduce the bug
	has-code	Whether description of this bug report contains code examples
	has-patch	Whether description of this bug report contains patches
	has-testcase	Whether description of this bug report contains test cases
	has-screenshot	Whether description of this bug report contains screenshots
Readability	flesch	These metrics are measured by the number of syllables per word and the length of sentences, which are used to quantify the readability of a text [3], [22], [25], [30], [45], [53], [69]
	fog	
	lix	
	kincaid	
	ari	
	coleman-liau	
	smog	
Text	summary-nb-score	Likelihood scores to be valid of this bug report calculated based on its summary: *summary-nb-score*, *summary-mnb-score*, *summary-dmnb-score* and *summary-cnb-score* are output by the naive Bayes, multinomial naive Bayes , discriminative multinomial naive Bayes and complement naive Bayes classifiers that are learned using summary of bug reports, respectively
	summary-mnb-score	
	summary-dmnb-score	
	summary-cnb-score	
	desc-nb-score	Likelihood scores to be valid of this bug report calculated based on its description: *desc-nb-score*, *desc-mnb-score*, *desc-dmnb-score* and *desc-cnb-score* are output by the naive Bayes, multinomial naive Bayes , discriminative multinomial naive Bayes and complement naive Bayes classifiers that are learned using description of bug reports, respectively
	desc-mnb-score	
	desc-dmnb-score	
	desc-cnb-score	

The *bug-num* feature does not consider the impact of time. For example, for a reporter who has reported 500 bug reports, *bug-num* of his/her next report is always calculated as 500. But his/her next report may be submitted a year later, which decreases the likelihood of the new report to be *valid* since the reporter may have not been familiar with the issue tracking system and developing process of the project. Thus, *bug-num* may suffer from the concept drift issue [23], [86]. To deal with this issue, we use the *recent-bug-num* feature to characterize the recent behavior of a reporter. The *recent-bug-num* of a bug report is computed as the number of prior bug reports submitted by its reporter in 90 days. We expect that we can gain a better overview of a reporter's experience by combining *recent-bug-num* and *bug-num*.

Just et al. proposed using the reputation of a reporter to identify the experience of a reporter [43]. Guo et al. found that bug reports submitted by reporters with more reputation are more likely to be fixed [31]. As fixed bug reports are identified to be *valid* in this study, we believe that a reporter's reputation is a good indicator of the validity of a bug report. In Guo et al.'s study, the reputation score of a reporter is computed as the percentage of fixed bug reports over the bug reports submitted by the reporter. We use a similar computation method, and we use the percentage of *valid* bug reports over the bug reports submitted by the reporter, i.e., *valid-rate*, to quantify the reputation of reporters.

The calculation of the three features is as follows. Let us denote a newly submitted bug report as B. For B, we first extract its creation time and reporter. We denote B's creation time and reporter as T and R, respectively. Then we extract all bug reports submitted by R before T. The *bug-num* feature is computed as the number of these bug reports. The *recent-bug-num* feature is computed as the number of the bug reports which are submitted by R in 90 days before T. Next, to compute *valid-rate*, we retrieve the resolution of all the bug reports submitted by R. From these bug reports, we count the number of *valid* bug reports (i.e., bug reports whose resolution is *FIXED* or *WONTFIX*). We also count the total number of *valid* and *invalid* bug reports (i.e., bug reports whose resolution is *FIXED*, *WONFIX*, *DUPLICATE*, *INVALID*, *WORKSFORME* or *INCOMPLETE*). The *valid-rate* feature is computed as the ratio between the former number and the latter number.

TABLE 3
P-values and Cliff's delta for the three features in the reporter experience dimension comparing *valid* and *invalid* bug reports on the mini-dataset.

Features	P-value	Cliff's delta
bug-num	<0.001	0.55 (Large)
valid-rate	<0.001	0.65 (Large)
recent-bug-num	<0.001	0.53 (Large)

Table 3 presents the p-values and Cliff's delta for the three features comparing *valid* and *invalid* bug reports on the mini-dataset. From the table, we find that for all the three features, *valid* and *invalid* bug reports have statistically significant differences. Also, all the effect sizes are positive

and large—indicating that the three features are effective in determining the validity of a bug report.

Collaboration Network Dimension refers to features based on the collaboration activities of a bug reporter in his/her bug handling community. Bug tracking involves many collaboration and communication activities between reporters and developers [5]. Ehrlich et al. observed that the social activities of a developer in the collaboration network of the OSS community can impact his/her performance [24]. Bettenburg et al. also found that the social activities of reporters in the collaboration network can impact the software quality of a project [7], [8]. Inspired by these studies, we expect that bug reporters' degree of activity in the collaboration network can impact the validity of their submitted bug reports. To quantify the degree of activity for bug reporters, we apply the nine network measures that were proposed in Zanetti et al.'s study [99]. We refer to the nine features as *lcc-membership*, *in-degree*, *out-degree*, *total-degree*, *clustering-coefficient*, *k-coreness*, *closeness-centrality*, *betweenness-centrality* and *eigenvector-centrality*.

The collaboration network of a project changes over time [42]. For example, some reporters or developers may leave the project. Collaboration networks built based on the full history of a bug repository cannot reflect such changes. Effectiveness of the features that are extracted from such collaboration networks will be impacted by time, i.e., they will suffer from the issue of concept drift [23], [86]. To deal with this issue, we decided to consider recent collaboration activities (i.e., within a short time window prior to the creation of the bug report). We use a time window of 30 days to build our collaboration networks, since Zanetti et al. found that when using 30 days as a time window, features extracted from their collaboration networks are effective in determining *valid* bug reports [99].

We first construct monthly networks based on collaborations of reporters and developers in the bug handling community. Then, we extract features of reporters' nodes in the networks. However, the collaboration activities of reporters and developers that we use are different from Zanetti et al.'s study. Zanetti et al. constructed collaboration networks based on the *ASSIGN* and *CC* relations of bug reports. In Zanetti et al.'s network, users are represented as nodes, and two users are linked by a directed edge when one user assigns a bug to the other user or one user adds the other user to the cc list of a bug report. Bettenburg et al. used comments of bug reports to analyze the collaboration network of developers [8]. Compared with *ASSIGN* and *CC* relations of bug reports, comments of bug reports can record all the collaborative interactions of users in the bug handling community. In this study, we consider constructing collaboration networks based on the sequence comments given to bug reports. We expect that relations extracted from comments of bug reports contain more information than the *ASSIGN* and *CC* relations.

To elaborate further, let us denote a newly submitted bug report as B. To extract its collaboration network features, we first extract its reporter and submission date. We denote the reporter of B as R. Then, since each comment is recorded along with its creation time stamp, we can retrieve all the comments created in the preceding month of the date when B is reported. After that, for each of these comments, we extract its creator and the reporter of its corresponding bug report. Next, we construct a directed graph based on creators and corresponding bug reporters of the comments. In the graph, nodes denote creators and corresponding bug reporters of the comments. And for each comment, its creator and its corresponding bug reporter are connected by a directed edge. Subsequently, we quantify R's degree of activity using the nine features that are proposed by Zanetti et al. [99].

To calculate the nine features for a reporter, we follow Zanetti et al.'s study and focus our analysis on the largest connected component (LCC) of a collaboration network. The *lcc-membership* feature is a binary feature and it will be set as true if the reporter is in the largest connected component. The three features (i.e., *total-degree*, *in-degree* and *out-degree*) quantify the number of connections of a reporter to other reporters and developers. The *in-degree* feature is the number of people who are connected to the reporter by the reporter's incoming edges. The *out-degree* feature is the number of people who are connected to the reporter by the reporter's outgoing edges. The *total-degree* feature is the sum of *in-degree* and *out-degree*. A higher *total-degree*, *in-degree* or *out-degree* indicates that the reporter is more active in the collaboration network.

K-coreness is derived from network decomposition [16]. In a network, a *k-core* [16] is a maximal subgraph containing nodes of degree *k* or more. The *k-coreness* of a node is *k* if it belongs to the k-core but not to the (k+1)-core. A higher *k-coreness* of a reporter's node indicates that he/she has a higher influence in the collaboration network.

Clustering-coefficient of a node is defined as the ratio between the number of triangles connected to the node and the number of triples that are centered around the node, where a triple centered around a node is a set of two edges that are connected to the node [64]. *Clustering-coefficient* is a measure quantifying the degree to which nodes in the network tend to cluster together. A higher *clustering-coefficient* of a reporter's node indicates that the reporter and his/her neighbors in the network has higher dense of collaboration activities.

Closeness-centrality of a node is defined as the inverse of sum of all distances of the node to all the other nodes [27]. *Closeness-centrality* is a measure quantifying the degree to which a node is close to all other nodes in a network. A higher *closeness-centrality* of a reporter's node indicates that he/she is closer to all other people in the collaboration network.

Betweenness-centrality of a node is defined as the total number of shortest paths between all possible pairs of nodes that pass through that node [14]. A higher *betweenness-centrality* of a reporter's node indicates that the reporter has more control on the collaboration network since more information will pass through his/her node.

Eigenvector-centrality is a network measure that assigns scores to nodes in a network based on the concept that connecting to high centrality nodes increases the node's centrality [12]. The *eigenvector-centrality* of a node is recursively defined by the centrality of the node's direct neighbors as shown in Formula 1, in which $DN(n_i)$ denotes the set of direct neighbors of the node n_i and λ is the largest

TABLE 4
P-values and Cliff's delta for the nine features in the collaboration network dimension comparing *valid* and *invalid* bug reports on the mini-dataset.

Features	P-value	Cliff's delta
lcc-membership	<0.001	0.43 (Med)
in-degree	<0.001	0.44 (Med)
out-degree	<0.001	0.43 (Med)
total-degree	<0.001	0.48 (Large)
eigenvector-centrality	<0.001	0.40 (Med)
betweenness-centrality	<0.001	0.40 (Med)
closeness-centrality	<0.001	0.41 (Med)
clustering-coefficient	<0.001	0.42 (Med)
k-coreness	<0.001	0.47 (Med)

eigenvalue of the adjacency matrix of the network.

$$Ev(n_i) = \frac{1}{\lambda} \sum_{n_j \in DN(n_i)} Ev(n_j) \quad (1)$$

We construct the collaboration networks and extract features using the Python package **NetworkX** [33].

Table 4 presents the p-values and Cliff's delta for the nine features comparing *valid* and *invalid* bug reports on the mini-dataset. From the table, we find that for all the nine features, the *valid* and *invalid* bug reports are statistically significantly different. Moreover, all the effect sizes are positive and at least medium—indicating that the nine features are effective in determining the validity of a bug report.

Completeness Dimension refers to features based on completeness of the technical information (e.g., stack traces and code samples) present in the bug report. Schroter et al. observed that stack traces help developers fix bugs [66]. Weimer noted that bug reports accompanied by the generated patches of their tool were three times more likely to be addressed than standard bug reports [85]. Zimmermann et al. found that steps to reproduce, stack traces, code samples, test cases and screenshots are widely used by developers to handle and fix bugs [104]. They reported that missing technical information is a severe problem with bug reports and it is a problem that is commonly encountered by bug triagers. Following these studies, we use the existence of technical information in the description of a bug report as an indicator of the validity of a bug report. The technical information includes: stack traces, steps to reproduce, code samples, patches, test cases and screenshots. The binary features *has-stack*, *has-step*, *has-code*, *has-patch*, *has-testcase* and *has-screenshot* represent the existence of the six types of technical information in a bug report respectively.

In Bugzilla, two ways are provided to describe a bug: writing textual description or attaching description files. In our study, we consider that description of a bug report includes its textual description and attachments. In our datasets, we find that the textual description of a bug report may contain stack traces, steps to reproduce, code samples, patches and test cases. And we find that the attachments of a bug report may contain stack traces, patches, test cases and screenshots. We set a binary feature as true if its corresponding technical information is presented in the textual description or attachments of the bug report.

The method we apply to recognize the technical information in the textual description of a bug report is as follows. We recognize stack traces, steps to reproduce, code samples, patches and test cases in the textual description of a bug report. To recognize stack traces and patches, we follow Bettenburg et al.'s study [9] and identify the technical information using the regular expressions they proposed. To recognize code samples in the description, we look for a set of code constructs like *classes*, *functions*, *conditional statements* or *loop statements*. If the description contains one of these constructs, we consider that the bug report contains code samples. To recognize steps to reproduce, we first list several patterns such as "*steps to reproduce*" and "*reproduce steps*". Then, we identify steps to reproduce by matching regular expressions of these patterns in the description of a bug report. To recognize test cases, we also list several patterns which indicate that the description contains test cases. We find that reporters usually first write words like *"test case:"* or *a line of text only containing words like "test case"* to highlight that there follows a test case. Thus, we match regular expressions of these patterns to identify test cases in bug reports.

The method we apply to recognize the technical information in the attachments of a bug report is as follows. We recognize stack traces, patches, test cases and screenshots in the attachments of a bug report by analyzing description of the attachments. We notice that in Bugzilla, each attachment has an id number and corresponds to a two-line paragraph in the description of a bug report. The first line of the paragraph starts with words *"Created attachment"* and *the attachment id*. The second line of the paragraph is the description of the attachment. We use a regular expression to recognize the first line of the paragraph and retrieve its second line, i.e., the description of the attachment. To recognize each type of technical information, we observe the description of attachments containing the technical information and list several words or phrases appearing in the description of the attachments. Then, we identify the technical information by checking whether one of the words or phrases appears in the description of attachments. To recognize stack traces, we use the word *"trace"*. To recognize patches, we use the words *"fix"* and *"patch"*. To recognize test cases, we use the words and phrases such as *"test case"* and *"testcase"*. To recognize screenshots, we use the words such as *"window"*, *"view"*, and *"screenshot"*.

A document describing all the regular expressions, phrases and words that are used in our study for recognizing different kinds of technical information in a bug report is available from our accompanying GitHub repository[9].

Table 5 presents the p-values and Cliff's delta for the six features comparing *valid* and *invalid* bug reports on the mini-dataset. In the table, we find that for one feature (i.e., *has-testcase*), the two groups of bug reports do not show a statistically significant difference. And five features have negligible effect sizes. However, these features cannot be simply considered worthless for determining *valid* bug

9. https://github.com/YuanruiZJU/TSE-Valid-Bug/blob/master/recognizing-technical-information.md

TABLE 5
P-values and Cliff's delta for the six features in the completeness dimension comparing *valid* and *invalid* bug reports on the mini-dataset.

Features	P-value	Cliff's delta	
has-stack	<0.001	0.03	(Negligible)
has-step	<0.001	-0.42	(Med)
has-code	<0.001	0.02	(Negligible)
has-patch	<0.001	0.06	(Negligible)
has-testcase	>0.05	0.00	(Negligible)
has-screenshot	<0.001	-0.02	(Negligible)

reports. They may be weak indicators for determining *valid* bug reports. But using all these features and combining these features with features from other dimensions may improve the classification performance of our approach. For example, we find that in the mini-dataset, 96% of the 360 bug reports which contain patches (i.e., *has-patch* is true) are *valid*—indicating that bug reports with patches have a high likelihood to be *valid*. Hence, Section 4.3 considers the combinations of features (in contrast to the simplified per-feature analysis in the mini study).

Readability Dimension refers to features that measure the readability of the description present in the bug report. Readability of a text is measured based on the syllables per word and the length of sentences in the text—it can evaluate how many years of education required for understanding the text without difficulties. In general, a text with higher readability scores is more complex to read. Hooimeijer et al. found that bug reports with better readability of their description are resolved faster [36]. Zimmermann et al. found that readability of a bug report's description is an important factor impacting the quality of bug reports [104]. Based on these prior studies, we expect that the readability of a bug report's description is a good indicator of the validity of a bug report. To quantify the readability of the description in the bug report, we use the seven readability measures proposed by previous studies—namely *flesch* [25], *fog* [30], *lix* [3], *kincaid* [45], *ari* [69], *coleman-liau* [22], and *smog* [53].

The seven readability features are calculated using seven formulas [3], [22], [25], [30], [45], [53], [69]. To introduce the formulas, we need to first introduce the definition of *complex words*, *long words*, *period*, and *polysyllables*.

Complex words are defined as those with three or more syllables, which do not include proper nouns, familiar jargon or compound words, and *complex words* do not contain common suffixes (e.g., "-es") as a syllable [30]. *Long words* are defined as those with more than six characters [3]. A *period* is defined as a period (i.e., "."), colon (i.e., ":") or capital first letter [3]. *Polysyllables* are defined as the words with three or more syllables in three groups of ten sentences, which are chosen in a row near the beginning, in the middle and in the end of the analyzed text (i.e., a bug report description in our case) [53].

We denote the number of characters, words, syllables, sentences, complex words, long words, periods and polysyllables in the text as *Characters*, *Words*, *Syllables*, *Sentences*, *Complex Words*, *Long Words*, *Periods* and *Polysyllables*, respectively. Notice that words appearing more than once should be counted when counting polysyllables [53]. Based on the above definitions, the seven formulas for calculating the seven readability features are shown below:

$$flesch = 206.835 - 1.015\frac{Words}{Sentences} - 84.6\frac{Syllables}{Words} \quad (2)$$

$$fog = 0.4\frac{Words}{Sentences} + 40\frac{Complex\ Words}{Words} \quad (3)$$

$$lix = \frac{Words}{Periods} + 100\frac{Long\ Words}{Words} \quad (4)$$

$$kincaid = 0.39\frac{Words}{Sentences} + 11.8\frac{Syllables}{Words} - 15.59 \quad (5)$$

$$ari = 4.71\frac{Characters}{Words} + 0.5\frac{Words}{Sentences} - 21.43 \quad (6)$$

$$coleman - liau = 5.88\frac{Characters}{Words} + 29.6\frac{Sentences}{Words} \quad (7)$$

$$smog = 3 + \sqrt{Polysyllables} \quad (8)$$

We use the Python package **readability**[10] to calculate the readability features for the description of bug reports.

TABLE 6
P-values and Cliff's delta for the seven features in the readability dimension comparing *valid* and *invalid* bug reports on the mini-dataset.

Features	P-value	Cliff's delta	
flesch	<0.001	-0.19	(Small)
fog	<0.001	0.08	(Negligible)
lix	<0.001	-0.06	(Negligible)
kincaid	<0.001	0.10	(Negligible)
ari	<0.001	0.04	(Negligible)
coleman-liau	<0.001	0.10	(Negligible)
smog	<0.001	0.13	(Negligible)

Table 6 presents the p-values and Cliff's delta for the seven features comparing *valid* and *invalid* bug reports on the mini-dataset. In the table, we find that for all the features, *valid* and *invalid* bug reports are statistically significantly different. Six of the features have negligible effect sizes. Similar to the completeness dimension, we cannot simply consider that these features are useless. Combining these features with features from other dimensions may improve the classification performance of our approach. For example, 2171 bug reports in the mini-dataset have a *coleman-liau* larger than 20 and in these bug reports, the proportion of *valid* ones pertains to 65%, which is substantially larger than the proportion (51%) of *valid* reports in the mini-dataset. Thus, on the mini-dataset, *coleman-liau* is an effective feature to distinguish *valid* and *invalid* bug reports.

10. https://github.com/mmautner/readability

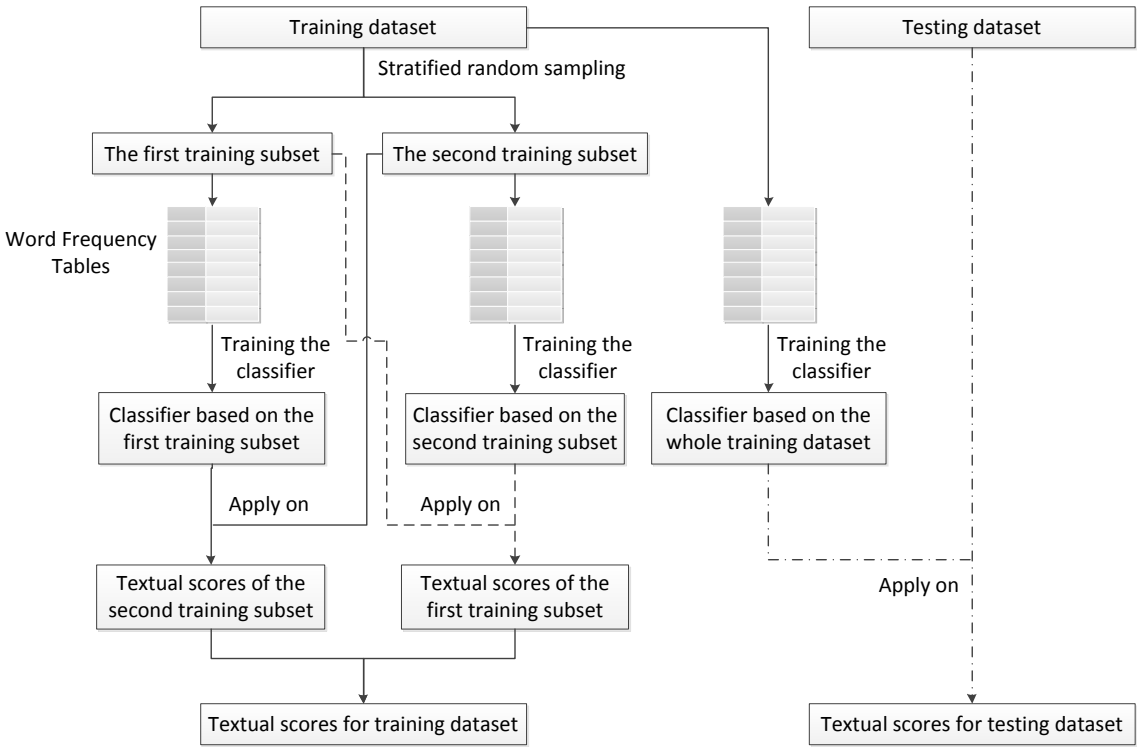

Fig. 1. Converting token features into textual scores for training and testing datasets.

Text Dimension refers to features that capture the textual characteristics based on text mining techniques. Prior work showed that using text mining techniques can help with bug classification [102]. Thus, we expect that analyzing the textual content of a bug report using text mining techniques can help us identify *valid* and *invalid* bug reports. A bug report consists of two important textual contents, i.e., its summary and description. Note that as mentioned earlier, we consider that the description of a bug report includes its initial description and comments added by its reporter within 15 minutes after its creation. In this paper, we separately extract textual features for summary and description of bug reports. And we extract textual features in two steps: extracting token features of bug reports and converting the token features into numerical scores.

1) Extracting Token Features. To leverage text mining techniques, we need to first extract token features from summary and description of bug reports. We use the same token feature extraction method applied to summary and description of bug reports. Without losing generality, we only present our feature extraction method for summary of the bug reports as follows. The same method is applied to description of bug reports.

We extract token features from summary of bug reports in four steps. First, we tokenize the text in the summary into words, phrases, symbols, or other meaningful name element tokens. Then, we remove the stop-words such as "the", "a", "of" which carry little value to distinguish *valid* bug reports from *invalid* ones. Next, we perform stemming on the tokens to reduce inflected or sometimes derived words into their stem, base or root form. For example the words "reading", "read", and "reads" would all be reduced to "read". After that, we use the resulting textual tokens and count the number of times each token appears in the summary of bug reports. By applying the above four steps, all the summary text of a bug report dataset can be represented as a word frequency table. In the word frequency table, summary of each bug report is represented as a word vector.

2) Converting Token Features into Numerical Scores. In our evaluation setup[11], there is a training dataset and testing dataset in each fold. Both training and testing datasets contain a large number of tokens that are extracted from the summary and description of bug reports. By contrast, the experience, collaboration network, completeness and readability dimensions contain 25 features only. The token features would crowd out the features from other dimensions if we directly combine these token features with other dimensions to learn models [92]. Moreover, a large number of features might cause the curse-of-dimensionality [35]. Thus, we follow prior studies [83], [97] and convert the large number of token features into a small number of numerical scores. These scores indicate how likely a report is *valid* using variants of naive Bayes when they are applied to the summary or description of bug reports. We refer to these numerical scores as *textual scores* of bug reports.

Here, we present the approach to converting token features (extracted from summary or description) of bug reports into textual scores.

Figure 1 presents our approach to converting token features into textual scores for training and testing datasets. The class labels of testing dataset in each fold are unknown for us. Thus, as shown in the figure, we can only learn

11. Please refer to Section 3.4 for our evaluation set up

TABLE 7
P-values and Cliff's delta for the eight features in the text dimension comparing *valid* and *invalid* bug reports on the mini-dataset.

Features	P-value	Cliff's delta
summary-nb-score	<0.001	0.41 (Med)
summary-mnb-score	<0.001	0.53 (Large)
summary-dmnb-score	<0.001	0.54 (Large)
summary-cnb-score	<0.001	0.41 (Med)
desc-nb-score	<0.001	0.38 (Med)
desc-mnb-score	<0.001	0.59 (Large)
desc-dmnb-score	<0.001	0.68 (Large)
desc-cnb-score	<0.001	0.49 (Large)

classifiers using bug reports in the training dataset. We elaborate the calculation of textual scores for training and testing datasets in one fold as follows.

The calculation of textual scores for the training dataset is as follows. We first split the training dataset into two subsets of equal sizes using stratified random sampling, so that the distribution and number of *valid* and *invalid* bug reports in both subsets are the same. Then, we train a classifier based on the first training subset and apply the classifier on the second training subset. For each bug report of the second training subset, the classifier outputs a "likelihood to be *valid*" score and we use the likelihood scores as textual scores for the second training subset. Also, we use the second training subset to train a classifier and use it to calculate the textual scores for the first training subset. By applying this strategy, we can calculate the textual scores for the training dataset without peeking into the testing dataset. Moreover, this strategy ensures that the used dataset for testing a classifier will not appear in the used dataset for training the classifier.

To calculate the textual scores for the testing dataset, we train a classifier based on the whole training dataset. Then, we apply the classifier on the testing dataset. For each bug report in the testing dataset, the classifier outputs a "likelihood to be *valid*" score and we use the likelihood scores as the textual scores of the testing dataset.

In this paper, we use four types of classifiers to convert token features into textual scores—namely naive Bayes classifier [54], multinomial naive Bayes classifier [54], discriminative multinomial naive Bayes classifier [74], and complement naive Bayes classifier [63]. Previous studies have shown that these four classifiers are fast and effective for text classification [54], [63], [74]. We use the implementation of the four classifiers in Weka [34]. We denote textual scores output by the four classifiers which are learned from summary of bug reports as *summary-nb-score*, *summary-mnb-score*, *summary-dmnb-score*, and *summary-cnb-score*, respectively. And we denote textual scores output by the four classifiers which are learned from description of bug reports as *desc-nb-score*, *desc-mnb-score*, *desc-dmnb-score*, and *desc-cnb-score*, respectively.

To calculate the eight textual features for the mini-dataset, we use the method introduced above for calculating the features for the training dataset of each fold. We split the mini-dataset into two subsets of equal sizes using stratified random sampling. We train models using the first subset and use the models to calculate the textual features for the second subset. Also, we train models using the second subset and use the models to calculate the textual features for the first subset. Then, we have the eight textual features for the whole mini-dataset. Table 7 presents the p-values and Cliff's delta for the eight features comparing *valid* and *invalid* bug reports on the mini-dataset. From the table, we find that for all the eight textual features, the *valid* and *invalid* bug reports are statistically significantly different. Moreover, all the effect sizes are positive and at least medium—indicating that the eight features are effective in determining the validity of a bug report.

3.3 Classifiers

We characterize a bug report using the 33 features that we extract. These features are then used to learn a model to determine the validity of a bug report when it is initially submitted. In this study, we use random forest [15] as default classifier to construct the model. Zanetti et al. used SVM as their default classifier [99]. We also use SVM as the underlying classifier for one of our baselines (i.e., SVMZ). In this section, we present the classifiers that we use in our study, i.e., random forest and SVM.

Random Forest: Random forest is proposed by Breiman [15]. It is an ensemble approach and specially designed for decision tree classifiers. Random forest is composed of multiple decision trees, each of which is built based on a random subset of the features. To label a sample, random forest adopts the mode of the class labels output by the decision trees. The major advantage of random forest is that it is generally highly accurate and it can automatically generate feature importance. Moreover, since random forest summarizes classification results of many trees that are learned differently, it is robust to noise and outliers. In this paper, we apply the default random forest implementation in Weka [34].

SVM: Support Vector Machines (SVMs) [90] are built based on statistical theory. The core idea of SVM is to construct a hyperplane or a set of hyperplanes in a high- or infinite-dimensional space, which are applied for classification. Given a set of training instances, an SVM model first selects a small number of critical boundary instances for each label as support vectors. In our case, the labels are *valid* and *invalid*. Then, the SVM model builds a linear or non-linear discriminative function to calculate classification boundaries using the principle of maximizing the margins among training instances of different labels. In this paper, we use the **LibSVM** package in Weka as our SVM implementation. LibSVM [18] is an integrated library for support vector classification, regression and distribution estimation. The library implements the SMO (Sequential Minimal Optimization) [60] algorithm for training SVMs. The SMO algorithm can efficiently solve the large quadratic programming (QP) problem that arises when training SVMs, since it breaks the problem into a series of smallest possible QP problems, which can be solved analytically in a short amount of time.

By default, when building SVM models for SVMZ, we use the default parameter setting of LibSVM in Weka. Parameter tuning may impact the classification performance of SVM [28]. Thus, in Section 5.1, we discuss the impact of tuning SVM parameters on the baseline.

3.4 Evaluation Setup and Baselines

In Section 3.2, we used the mini-dataset to investigate the effectiveness of the 33 extracted features. To avoid potential bias induced by the mini-dataset, we exclude the 10,000 bug reports of the mini-dataset from our datasets. In Section 4 and Section 5, we experiment on the remaining data which contains a total of 560,697 bug reports to evaluate the performance of our approach.

To simulate the usage of our approach in practical bug handling process, we use the longitudinal data setup described in previous software engineering studies [10], [77], [93]. First, the bug reports from each project presented in Table 1 are first sorted in chronological order according to their initially submission time. Then we put the bug reports into 11 non-overlapping frames (i.e., ordered set of bug reports), where each of them has the same number of bug reports. The process proceeds as follows: First, in fold 0, we train using bug reports in frame 0, and test the trained model using the bug reports in frame 1. Then, in fold 1, we train using bug reports in frame 1, and test the trained model using the bug reports in frame 2, and so on. In the final fold (fold-9), we train using bug reports in frame 9, and test using bug reports in frame 10. Notice that in each fold, we use one frame as training dataset and its next frame as testing dataset. We then calculate the average AUC scores across the 10 folds. In RQ1 and RQ2, we use this evaluation setup.

Zanetti et al. [99] proposed a set of features based on the collaboration networks they constructed, and used these features to determine whether a bug report is *valid* or not. They constructed collaboration network on the basis of dyadic relations *ASSIGN* and *CC*. Then for each bug report, they extracted nine features of its reporter's corresponding node in the network. After that, they used SVM as default underlying classifier to build models based on the extracted features. In this paper, we would like to investigate whether our features are more effective than the collaboration network features proposed by Zanetti et al.. To make a comparison, we choose two baselines which are all based on Zanetti et al.'s features. We name the two baselines as SVMZ and RFZ. SVMZ uses SVM as its underlying classifier, which is the original Zanetti et al.'s method as presented in their paper. RFZ uses random forest as its underlying classifier, which is the same as the classifier used by our approach. We introduce the second baseline to investigate whether the improvement that we achieve is due to the difference in the classifier or not.

3.5 Evaluation Metrics

For each bug report, there would be four possible determination outcomes: a bug report is determined as *valid* when it is truly *valid* (true positive, TP); it can be determined as *valid* when it is truly *invalid* (false positive, FP); it can be determined as *invalid* when it is truly *valid* (false negative, FN); it can be determined as *invalid* when it is truly *invalid* (true negative, TN). Based on these possible outcomes, we calculate precision, recall and F1-score to evaluate the performance of our approach in comparison with the baselines. We also use AUC as an evaluation metric in our study.

Valid Precision: is the proportion of bug reports that are correctly labeled as *valid* among those that are determined as *valid*, i.e.:

$$P(v) = \frac{TP}{TP + FP} \tag{9}$$

Valid Recall: is the proportion of *valid* bug reports that are correctly labeled, i.e.:

$$R(v) = \frac{TP}{TP + FN} \tag{10}$$

Invalid Precision: is the proportion of bug reports that are correctly labeled as *invalid* among those that are determined as *invalid*, i.e.:

$$P(i) = \frac{TN}{TN + FN} \tag{11}$$

Invalid Recall: is the proportion of *invalid* bug reports that are correctly labeled, i.e.:

$$R(i) = \frac{TN}{FP + TN} \tag{12}$$

F1-score: is a summary measure that combines both precision and recall—it evaluates if an increase in precision (recall) outweighs a reduction in recall (precision). For *valid* bug reports, the F1-score is calculated as the harmonic mean of $P(v)$ and $R(v)$, i.e.:

$$F1(v) = \frac{2 \times P(v) \times R(v)}{P(v) + R(v)} \tag{13}$$

For *invalid* bug reports, the F1-score is calculated as the harmonic mean of $P(i)$ and $R(i)$, i.e.:

$$F1(i) = \frac{2 \times P(i) \times R(i)}{P(i) + R(i)} \tag{14}$$

AUC: We use the Area Under the Curve (AUC) of receiver operator characteristic (ROC) [38] as a performance measure in our study. The AUC score is computed by plotting the ROC curve. In the ROC curve, the true positive rate (TPR) is plotted as a function of the false positive rate (FPR) across all thresholds. In our context, a threshold refers to the threshold that is needed by a classifier to label a bug report as *valid* or *invalid*. When determining the validity of a bug report, the classifier first outputs a likelihood score for the report to be *valid*, and then, the classifier needs to compare the score with a threshold to decide the label of the report. If the likelihood is higher than the threshold, the bug report is predicted as *valid*. Otherwise, the bug report is predicted as *invalid*. The threshold can be a value ranging from 0 to 1. AUC measures the prediction performance across all the thresholds. Thus, it is a threshold independent measure [13].

Also, AUC is robust towards the class imbalance problem [48], [78]. When calculating AUC, data imbalances that are inherent to the datasets can be automatically considered. Thus, AUC is not biased for classifiers when they are evaluated on datasets of different class distributions. AUC ranges from 0 to 1. Higher AUC values indicate better classification performance. An AUC of 0.5 indicates a classifier that is no better than random guessing.

As a general rule, a classifier with an AUC score that is more than 0.7 is considered to have acceptable performance [37].

AUC has a statistical interpretation: in our context, it evaluates the possibility that a classifier ranks a randomly chosen *valid* bug report higher than a randomly chosen *invalid* bug reports [48]. When developers use a classifier to prioritize their triaging tasks, AUC is an appropriate measure to evaluate the quality of the ranking list that is produced by the classifier.

4 Results

In this section, we presents answers of the four research questions proposed in Section 3.

4.1 RQ1: Can we effectively determine the validity of a bug report?

To evaluate the performance of our approach and the baselines, we calculate the average AUC, precision, recall, and F1-scores for *valid* and *invalid* bug reports across the 10 folds. Table 8 presents the average AUC, F1-scores, precision and recall for *valid* and *invalid* bug reports of our approach in comparison with the baselines. On average, across the five datasets, our approach achieves an AUC, F1(v) and F1(i) of 0.81, 0.74 and 0.67, respectively. As shown in Table 8, our approach consistently outperforms the baselines in terms of AUC, F1(v) and F1(i) across the five datasets. We use Wilcoxon signed-rank test [87] with a Bonferroni correction [1] to investigate whether the improvements of our approach over the baselines are statistically significant. We also compute Cliff's delta [21]. Table 9 presents the adjusted p-values and Cliff's delta comparing AUC, F1(v) and F1(i) for our approach with the baselines.

We notice that as shown in Table 8, our approach outperforms SVMZ and RFZ in terms of average F1(i) across the 10 folds on the Netbeans dataset, while in Table 9, our statistical tests show that the improvements of our approach over the two baselines are not significant. We look into the F1(i) scores of our approach, SVMZ and RFZ in the 10 folds. We find that in three folds, SVMZ and RFZ only achieve F1(i) scores which are around 0.1, indicating that both SVMZ and RFZ cannot handle imbalanced data in these folds. As for our approach, in these folds, our approach can still achieve F1(i) scores which are around 0.5. Hence, the average F1(i) scores of SVMZ and RFZ on the Netbeans dataset are much lower than our approach. However, in five folds, both SVMZ and RFZ achieve equal/larger F1(i) scores than our approach with much lower F1(v) scores than our approach. Hence, the improvements of our approach over the baselines are not significant in terms of F1(i).

Moreover, we also notice that on the Eclipse dataset, both SVMZ and RFZ achieve a much lower recall for *invalid* bug reports (i.e., R(i)) than our approach. And the two baselines achieve an average F1(i) score near to 0. Hence, the two baselines cannot handle imbalanced data. In comparison with the two baselines, our approach achieves a much better performance in terms of F1(i). The underlying classifier of RFZ is the same as the classifier used by our approach. It indicates that our features are more robust towards imbalanced data than Zanetti et al.'s features.

From Tables 8 and 9, we observe the following findings:

- In terms of AUC, our approach on average improves SVMZ and RFZ by 33% and 19%, respectively. Statistical tests show that the improvements are significant, and all the effect sizes are positive and large.
- In terms of F1-scores for *valid* bug reports, i.e., F1(v), our approach on average improves SVMZ and RFZ by 9% and 12%, respectively. Statistical tests show that the improvements are significant, and all the effect sizes are positive and at least medium.
- In terms of F1-scores for *invalid* bug reports, i.e., F1(i), our approach on average improves SVMZ and RFZ by 34% and 29%, respectively. Statistical tests show that only two of the improvements are not significant, and most of the effect sizes are positive and non-negligible.

In summary, our approach outperforms SVMZ and RFZ by a substantial margin. Thus, our features are more effective than the proposed features by Zanetti et al. [99] to determine the validity of a bug report.

4.2 RQ2: How effective is our approach when all features are used than when a single dimension of features is used?

By default, our approach combines 33 features from five dimensions (i.e., reporter experience, collaboration network, completeness, readability and text). In this research question, we investigate how effective is our model when built on a single dimension of features. We do this to find out whether our approach benefits from using multiple dimensions of features—as compared to a single dimension of features. If the answer is no, it means that we have made a simple problem too complex and we just need to use a feature dimension to identify *valid* bug reports. To achieve this goal, We compare performance of these models and our model learned using all features. For convenience, in this section, we denote our model learned using all the 33 features as All Features.

We build a random forest model using features from each dimension. In total, we build five random forest models for each dataset. We denote them as *reporter experience*, *collaboration network*, *completeness*, *readability* and *text* models, respectively. We then experiment the five random forest models on each dataset, and compute the average AUC, precision, recall and F1-scores for *valid* and *invalid* bug reports in the 10 folds.

Table 10 presents the AUC, precision, recall and F1-scores for *valid* and *invalid* bug reports of the All Features models in comparison with the models built using features in each dimension. As shown in the table, the All Features models achieve better AUC, F1(v) and F1(i) scores than the models that are learned using one dimension across the five datasets. Similar to RQ1 and RQ2, we use Wilcoxon signed-rank test with Bonferroni correction to investigate whether the improvements of the All Features models over the models learnt using one dimension are statistically significant. And we use Cliff's delta to measure effect size. Table 11 presents the adjusted p-values and Cliff's delta comparing AUC, F1(v) and F1(i) scores for the All Features models with the models learned using one dimension.

From Table 10 and 11, we have the following findings:

TABLE 8
AUC, precision, recall and F1-scores for *valid* and *invalid* bug reports of our approach in comparison with the baselines. The best results are in bold.

Project	Approach	AUC	Valid			Invalid		
			F1(v)	P(v)	R(v)	F1(i)	P(i)	R(i)
Eclipse	Ours	**0.76**	**0.88**	**0.82**	0.96	**0.35**	**0.63**	**0.24**
	SVMZ	0.50	**0.88**	0.78	**1.00**	0.00	0.00	0.00
	RFZ	0.63	0.87	0.78	0.99	0.01	0.15	0.01
Netbeans	Ours	**0.73**	**0.76**	**0.72**	**0.80**	**0.55**	**0.61**	0.49
	SVMZ	0.58	0.69	0.69	0.73	0.39	0.36	0.44
	RFZ	0.63	0.67	0.71	0.68	0.44	0.46	**0.50**
Mozilla	Ours	**0.83**	**0.86**	**0.82**	**0.90**	**0.62**	**0.71**	**0.55**
	SVMZ	0.58	0.81	0.74	0.89	0.33	0.54	0.27
	RFZ	0.64	0.78	0.74	0.84	0.36	0.47	0.32
Firefox	Ours	**0.87**	**0.62**	**0.70**	**0.56**	**0.90**	**0.88**	**0.92**
	SVMZ	0.70	0.55	0.63	0.50	0.88	0.87	0.90
	RFZ	0.77	0.50	0.61	0.43	0.88	0.85	0.91
Thunderbird	Ours	**0.84**	**0.59**	**0.74**	**0.51**	**0.92**	**0.89**	**0.95**
	SVMZ	0.68	0.49	0.66	0.41	0.90	0.87	0.94
	RFZ	0.75	0.50	0.62	0.43	0.90	0.87	0.94

TABLE 9
Adjusted P-values and Cliff's Delta comparing AUC, F1(v) and F1(i) for our approach with the baselines.

Project	Approach	AUC		F1(v)		F1(i)	
Eclipse	SVMZ	1.00	(Large)**	0.44	(Med)**	1.00	(Large)**
	RFZ	1.00	(Large)**	0.58	(Large)**	1.00	(Large)**
Netbeans	SVMZ	1.00	(Large)**	0.62	(Large)*	0.06	(Negligible)
	RFZ	1.00	(Large)**	0.80	(Large)**	-0.22	(Small)
Mozilla	SVMZ	1.00	(Large)**	0.84	(Large)**	1.00	(Large)**
	RFZ	1.00	(Large)**	1.00	(Large)**	1.00	(Large)**
Firefox	SVMZ	1.00	(Large)**	0.42	(Med)**	0.28	(Small)**
	RFZ	1.00	(Large)**	0.60	(Large)**	0.28	(Small)**
Thunderbird	SVMZ	0.96	(Large)**	0.42	(Med)**	0.46	(Med)**
	RFZ	0.84	(Large)**	0.44	(Med)**	0.60	(Large)**

***p<0.001, **p<0.01, *p<0.05

- In terms of AUC, the All Features models on average improve the *reporter experience*, *collaboration network*, *completeness*, *readability* and *text* models by 11%, 16%, 25%, 29% and 7%. Statistical tests show that the improvements are significant, and all the effect sizes are non-negligible.
- In terms of F1-scores for *valid* bug reports, i.e., F1(v), the All Features models on average improve the *reporter experience*, *collaboration network*, *completeness*, *readability* and *text* models by 7%, 9%, 19%, 25% and 4%, respectively. Statistical tests show that in most cases, the All Features models significantly improve the models built on a single dimension of features with non-negligible effect sizes.
- In terms of F1-scores for *invalid* bug reports, i.e., F1(i), the All Features models on average improve the *reporter experience*, *collaboration network*, *completeness*, *readability* and *text* models by 12%, 24%, 49%, 29% and 8%, respectively. Statistical tests show that in most cases, the All Features models significantly improve the models built on a single dimension of features with non-negligible effect sizes.

In summary, our approach that uses all features is more effective than the models using a single dimension of features in determining the validity of a bug report. Thus, our approach benefits from using multiple dimensions of features—as compared to a single dimension features.

4.3 RQ3: Which features are most important for differentiating *valid* bug reports from *invalid* ones?

In this research question, we would like to find out the most important features that differentiate *valid* bug reports from *invalid* bug reports in our Eclipse, Netbeans, Mozilla, Firefox and Thunderbird datasets. In different projects, the characteristics of bug reports that differentiate *valid* and *invalid* bug reports may be different. Thus, we conduct our experiment on each bug report dataset.

Note that in RQ1 and RQ2, the textual features we use are extracted based on training dataset and testing dataset in each fold. In this research question, we calculate the textual features for each dataset using all the bug reports in the dataset. First, we split the dataset into two subsets of equal sizes using stratified random sampling. Then, we create word frequency tables for each subsets. Next, we use the first subset to train models using the four types of classifiers presented in Section 3.2. We apply these models to calculate textual scores for the second subset. We also use the second subset to train models using the four types of classifiers and apply these models to calculate textual scores for the first subset. After the above steps, we have textual features for all the bug reports in the dataset.

Following Tian et al.'s study [82], we leverage the random forest classifier with 10-times 10-fold cross-validation to investigate the most important features. Comparing with the random forest models we build in RQ1,

TABLE 10
AUC, precision, recall and F1-scores for *valid* and *invalid* bug reports of All Features models in comparison with models built using features in each dimension. The best results are in bold.

Project	Approach	AUC	Valid			Invalid		
			F1(v)	P(v)	R(v)	F1(i)	P(i)	R(i)
Eclipse	All Features	**0.76**	**0.88**	**0.82**	0.96	**0.35**	**0.63**	**0.24**
	Reporter Experience	0.67	0.86	0.80	0.94	0.18	0.34	0.13
	Collaboration Network	0.64	0.87	0.78	0.98	0.04	0.20	0.02
	Completeness	0.58	**0.88**	0.78	**1.00**	0.00	0.21	0.00
	Readability	0.57	0.87	0.79	0.98	0.10	0.39	0.06
	Text	0.68	0.87	0.81	0.96	0.26	0.53	0.17
Netbeans	All Features	**0.73**	**0.76**	**0.72**	0.80	**0.55**	**0.61**	0.49
	Reporter Experience	0.64	0.69	0.70	0.68	0.52	0.51	**0.53**
	Collaboration Network	0.63	0.70	0.68	0.74	0.42	0.49	0.43
	Completeness	0.52	**0.76**	0.61	**1.00**	0.00	0.22	0.00
	Readability	0.57	0.70	0.64	0.77	0.37	0.46	0.31
	Text	0.66	0.73	0.68	0.80	0.47	0.56	0.41
Mozilla	All Features	**0.83**	**0.86**	**0.82**	**0.90**	**0.62**	**0.71**	**0.55**
	Reporter Experience	0.73	0.81	0.79	0.84	0.52	0.57	0.49
	Collaboration Network	0.69	0.81	0.77	0.87	0.47	0.57	0.41
	Completeness	0.68	0.83	0.77	**0.90**	0.48	0.62	0.39
	Readability	0.68	0.81	0.75	0.88	0.40	0.54	0.32
	Text	0.80	0.85	0.81	0.89	0.59	0.67	0.53
Firefox	All Features	**0.87**	**0.62**	0.70	**0.56**	**0.90**	**0.88**	0.92
	Reporter Experience	0.81	0.54	0.62	0.47	0.88	0.85	0.91
	Collaboration Network	0.79	0.50	0.59	0.44	0.88	0.85	0.91
	Completeness	0.74	0.28	**0.82**	0.23	0.87	0.83	**0.94**
	Readability	0.70	0.34	0.56	0.26	0.87	0.81	0.92
	Text	0.85	0.58	0.66	0.53	0.89	0.87	0.91
Thunderbird	All Features	**0.84**	**0.59**	**0.74**	**0.51**	**0.92**	**0.89**	**0.95**
	Reporter Experience	0.79	0.53	0.64	0.46	0.90	0.87	0.94
	Collaboration Network	0.77	0.51	0.63	0.44	0.90	0.87	0.93
	Completeness	0.74	0.33	0.82	0.26	0.89	0.84	**0.95**
	Readability	0.65	0.24	0.48	0.17	0.88	0.82	**0.95**
	Text	0.81	0.54	0.67	0.46	0.91	0.88	0.94

TABLE 11
Adjusted P-values and Cliff's Delta comparing AUC, F1(v) and F1(i) for All Features models with models built using features in each dimension.

Project	Approach	AUC		F1(v)		F1(i)	
Eclipse	Reporter Experience	1.00	(Large)**	0.80	(Large)**	0.90	(Large)**
	Collaboration Network	1.00	(Large)**	0.74	(Large)**	1.00	(Large)**
	Completeness	1.00	(Large)**	0.44	(Med)**	1.00	(Large)**
	Readability	1.00	(Large)**	0.62	(Large)**	1.00	(Large)**
	Text	1.00	(Large)**	0.50	(Large)**	0.82	(Large)**
Netbeans	Reporter Experience	1.00	(Large)**	0.88	(Large)**	0.40	(Med)*
	Collaboration Network	1.00	(Large)**	0.74	(Large)**	0.30	(Small)
	Completeness	1.00	(Large)**	-0.04	(Negligible)	1.00	(Large)**
	Readability	1.00	(Large)**	0.82	(Large)**	1.00	(Large)**
	Text	0.94	(Large)**	0.44	(Med)**	0.76	(Large)**
Mozilla	Reporter Experience	0.98	(Large)**	0.82	(Large)**	1.00	(Large)**
	Collaboration Network	1.00	(Large)**	0.70	(Large)**	1.00	(Large)**
	Completeness	1.00	(Large)**	0.54	(Large)**	1.00	(Large)**
	Readability	1.00	(Large)**	0.82	(Large)**	1.00	(Large)**
	Text	0.52	(Large)**	0.42	(Med)**	0.92	(Large)**
Firefox	Reporter Experience	0.90	(Large)**	0.50	(Large)**	0.26	(Small)**
	Collaboration Network	0.98	(Large)**	0.60	(Large)**	0.30	(Small)**
	Completeness	1.00	(Large)**	0.72	(Large)**	0.30	(Small)**
	Readability	1.00	(Large)**	0.94	(Large)**	0.32	(Small)**
	Text	0.60	(Large)**	0.26	(Small)**	0.18	(Small)**
Thunderbird	Reporter Experience	0.64	(Large)**	0.36	(Med)**	0.52	(Large)**
	Collaboration Network	0.68	(Large)**	0.44	(Med)**	0.52	(Large)**
	Completeness	0.84	(Large)**	0.84	(Large)**	0.46	(Med)**
	Readability	1.00	(Large)**	1.00	(Large)**	0.72	(Large)**
	Text	0.32	(Small)**	0.24	(Small)**	0.36	(Med)**

***p<0.001, **p<0.01, *p<0.05

we first apply feature selection and use the selected features to build another random forest models. Feature selection is aimed at removing correlated features which may lead to poor models [56].

Step 1: Correlation Analysis. For each bug report dataset, we first look for the correlations among the features by taking advantage of variable clustering analysis implemented in the R package **Hmisc**. Then, we construct a hierarchical overview of the 33 features. In the hierarchical overview, the correlated features are grouped into sub-hierarchies. To remove correlated features, we use the same setting in the previous study [82]—we randomly select one feature and remove the other features from a sub-hierarchy if the correlations of features in the sub-hierarchy are above 0.7. After this step, we remove 15, 18, 16, 19 and 20 features in the datasets of Eclipse, Netbeans, Mozilla, Firefox and Thunderbird, respectively. Note that we separately study the most important features for different projects and different projects may have different correlated features. Thus, we have different remaining features for different projects.

Step 2: Redundancy Analysis. After reducing collinearity among the features by correlation analysis, we detect redundant features which do not consist of unique signal as compared to the other features. To do the redundancy analysis, we apply the **redun** function provided by the R package **rms**. By redundancy analysis, we find that in the Eclipse, Mozilla and Firefox datasets, one of the remaining features can be represented by other features (i.e., the feature is redundant). Thus, in this step, we remove one feature from the Eclipse, Mozilla and Firefox datasets, respectively.

Step3: Important Features Identification. After the above two steps, there are 17, 15, 16, 13 and 13 features remaining in the Eclipse, Netbeans, Mozilla, Firefox and Thunderbird datasets, respectively. For each dataset, we build a random forest model based on the remaining features using the R package **bigrf**. In the training process, we use the **varimp** function in the R package **bigrf** to compute the importance of features. The feature importance evaluation is based on an internal error estimate of a random forest classifier, which is called out of the bag (OOB) estimate [89]. The key idea behind it is to see whether the OOB estimate will be reduced significantly or not when features are randomly permuted one by one.

We perform 10 times 10-fold cross-validation when we compute the importance of features. In each run of 10-fold cross-validation, we have 10 importance values for each feature. To determine which of the features are the most important, we apply the Scott-Knott Effect Size Difference (ESD) test [49], [79], [91] for the importance values taken from all 10 runs of 10-fold cross-validation. The Scott-Knott ESD test is different from the Scott-Knott test [67]. The Scott-Knott test assumes that the data is normally distributed. This might cause that the created groups are trivially different from one another. The Scott-Knott ESD test can correct the non-normal distribution of an input dataset and merge any two statistically distinct groups (i.e., the groups have a negligible effect size) into one group.

Tables 12, 13, 14, 15 and 16 present the importance of the remaining features as ranked according to the Scott-Knott

TABLE 12
Importance of the 17 features in the Eclipse dataset as ranked according to the Scott-Knott ESD test. The second and third columns show P-values, Cliff's Delta for the features. The features with non-negligible effect sizes are in bold.

Groups	Features	P-value	Cliff's delta	
1	**desc-dmnb-score**	**<0.001**	**0.52**	**(Large)**
2	**valid-rate**	**<0.001**	**0.45**	**(Med)**
3	**summary-dmnb-score**	**<0.001**	**0.40**	**(Med)**
4	**summary-nb-score**	**<0.001**	**0.26**	**(Small)**
5	flesch	<0.001	0.08	(Negligible)
6	lix	<0.001	-0.12	(Negligible)
7	**bug-num**	**<0.001**	**0.22**	**(Small)**
8	clustering-coefficient	<0.001	0.13	(Negligible)
9	**desc-mnb-score**	**<0.001**	**0.16**	**(Small)**
10	**desc-nb-score**	**<0.001**	**0.18**	**(Small)**
11	**lcc-membership**	**<0.001**	**0.20**	**(Small)**
12	has-step	<0.001	-0.09	(Negligible)
13	**has-stack**	**<0.001**	**-0.15**	**(Small)**
14	has-code	<0.001	-0.01	(Negligible)
15	has-screenshot	<0.001	-0.02	(Negligible)
16	has-patch	<0.001	0.06	(Negligible)
17	has-testcase	>0.05	0.00	(Negligible)

TABLE 13
Importance of the 15 features in the Netbeans dataset as ranked according to the Scott-Knott ESD test. The second and third columns show P-values, Cliff's Delta for the features. The features with non-negligible effect sizes are in bold.

Groups	Features	P-value	Cliff's delta	
1	**desc-dmnb-score**	**<0.001**	**0.43**	**(Med)**
2	**summary-dmnb-score**	**<0.001**	**0.33**	**Small**
3	**valid-rate**	**<0.001**	**0.37**	**(Med)**
4	**desc-mnb-score**	**<0.001**	**0.30**	**(Small)**
5	**summary-nb-score**	**<0.001**	**0.20**	**(Small)**
6	lix	<0.001	-0.02	(Negligible)
7	**bug-num**	**<0.001**	**0.28**	**(Small)**
8	**clustering-coefficient**	**<0.001**	**0.18**	**(Small)**
9	desc-nb-score	<0.001	0.07	(Negligible)
10	has-stack	<0.001	-0.05	(Negligible)
11	has-step	<0.001	0.02	(Negligible)
12	has-screenshot	>0.05	0.00	(Negligible)
13	has-code	<0.001	0.02	(Negligible)
14	has-testcase	<0.001	0.00	(Negligible)
15	has-patch	<0.001	0.01	(Negligible)

TABLE 14
Importance of the 16 features in the Mozilla dataset as ranked according to the Scott-Knott ESD test. The second and third columns show P-values, Cliff's Delta for the features. The features with non-negligible effect sizes are in bold.

Groups	Features	P-value	Cliff's delta	
1	**desc-dmnb-score**	**<0.001**	**0.64**	**(Large)**
2	**summary-dmnb-score**	**<0.001**	**0.58**	**(Large)**
3	**valid-rate**	**<0.001**	**0.55**	**Large**
4	**bug-num**	**<0.001**	**0.30**	**(Small)**
5	lix	<0.001	-0.12	(Negligible)
6	coleman-liau	<0.001	-0.03	(Negligible)
7	**clustering-coefficient**	<0.001	**0.25**	**(Small)**
8	eigenvector-centrality	<0.001	0.13	(Negligible)
9	**has-step**	**<0.001**	**-0.30**	**(Small)**
10	**desc-nb-score**	**<0.001**	**0.20**	**Small**
11	**lcc-membership**	**<0.001**	**0.21**	**(Small)**
12	has-screenshot	<0.001	-0.06	(Negligible)
13	has-patch	<0.001	0.06	(Negligible)
14	has-testcase	<0.001	0.00	(Negligible)
15	has-code	>0.05	0.00	(Negligible)
16	has-stack	<0.001	0.00	(Negligible)

TABLE 15
Importance of the 13 features in the Firefox dataset as ranked according to the Scott-Knott ESD test. The second and third columns show P-values, Cliff's Delta for the features. The features with non-negligible effect sizes are in bold.

Groups	Features	P-value	Cliff's delta	
1	**desc-dmnb-score**	**<0.001**	**0.75**	**(Large)**
2	**valid-rate**	**<0.001**	**0.67**	**(Large)**
3	**summary-dmnb-score**	**<0.001**	**0.63**	**(Large)**
4	**summary-nb-score**	**<0.001**	**0.44**	**(Med)**
5	**lix**	**<0.001**	**-0.20**	**(Small)**
6	coleman-liau	<0.001	-0.06	(Negligible)
7	**has-step**	**<0.001**	**-0.48**	**(Large)**
8	**desc-nb-score**	**<0.001**	**0.40**	**Med**
9	has-patch	<0.001	0.09	(Negligible)
10	has-screenshot	<0.001	0.01	(Negligible)
11	has-testcase	>0.05	0.00	(Negligible)
12	has-code	<0.001	0.00	(Negligible)
13	has-stack	>0.05	0.00	(Negligible)

TABLE 16
Importance of the 13 features in the Thunderbird dataset as ranked according to the Scott-Knott ESD test. The second and third columns show P-values, Cliff's Delta for the features. The features with non-negligible effect sizes are in bold.

Groups	Features	P-value	Cliff's delta	
1	**desc-dmnb-score**	**<0.001**	**0.68**	**Large**
2	**valid-rate**	**<0.001**	**0.62**	**(Large)**
3	**summary-dmnb-score**	**<0.001**	**0.52**	**(Large)**
4	**clustering-coefficient**	**<0.001**	**0.50**	**(Large)**
5	coleman-liau	>0.05	0.01	(Negligible)
6	lix	<0.001	-0.10	(Negligible)
7	**desc-nb-score**	**<0.001**	**0.41**	**(Small)**
8	**has-step**	**<0.001**	**-0.43**	**(Med)**
9	**has-patch**	**<0.001**	**0.16**	**(Small)**
10	has-screenshot	<0.001	0.02	(Negligible)
11	has-code	<0.05	0.00	(Negligible)
12	has-stack	>0.05	0.00	(Negligible)
13	has-testcase	>0.05	0.00	(Negligible)

ESD test results in the Eclipse, Netbeans, Mozilla, Firefox and Thunderbird, respectively. As shown in the tables, the features *desc-dmnb-score*, *valid-rate* and *summary-dmnb-score* are consistently ranked in the top three important features that affect the random forest models to differentiate *valid* bug reports and *invalid* ones across the five datasets.

Step4: Effect of Important Features. To understand the impact of each feature, we compare the values of the remaining features between *valid* and *invalid* bug reports in the five datasets. Similar to Section 3.2, we apply the Wilcoxon rank-sum test [52] to analyze the statistical significance of the difference between *valid* and *invalid* bug reports. We use Cliff's delta to measure the effect size of difference between the two groups of bug reports. Tables 12, 13, 14, 15 and 16 present the p-values and Cliff's delta for the remaining features of the Eclipse, Netbeans, Mozilla, Firefox and Thunderbird datasets, respectively.

Based on the results shown in Tables 12–16, the features *desc-dmnb-score*, *valid-rate* and *summary-dmnb-score* are the most important features to distinguish *valid* and *invalid* bug reports. Across the five datasets, these features are ranked in the top three most important features, and they have statistically significant and non-negligible positive effect. This indicates that for a bug report, its textual features and its reporter's experience are the most important factors that positively influence the likelihood of the bug report being *valid*.

4.4 RQ4: How effective is our model when built on a subset of our features?

In this research question, we investigate the effectiveness of various subsets of our features. We do this in two parts. We first focus on the five feature dimensions and investigate the contributions of each dimension. Then, we focus on subsets of individual features coming across the five dimensions. Similarly to Section 4.2, in this section, we denote our model learned using all the 33 features as All Features.

4.4.1 Feature Dimensions

In this part, we investigate effectiveness of the five dimensions. To do this, we evaluate contribution of each dimension to our approach in presence of the other four dimensions.

To investigate contribution of a dimension to our approach, we exclude the dimension and compare performance of the All Features model and the random forest model built using features of remaining four dimensions. If the dimension has contributions to our approach, the All Features model will achieve better performance than the latter model.

We exclude a dimension at a time and learn a random forest model using the remaining dimensions. In total, we build five random forest models for each dataset. We denote them as *reporter experience*$^{\complement}$, *collaboration network*$^{\complement}$, *completeness*$^{\complement}$, *readability*$^{\complement}$ and *text*$^{\complement}$, respectively. The symbol $\complement$ denotes that the model uses the complement set of the features (e.g., *text*$^{\complement}$ denotes the model using features excluding the text dimension). We then experiment the five random forest models on each dataset, and compute the average AUC, precision, recall and F1-scores for *valid* and *invalid* bug reports across the 10 folds.

Table 17 presents the AUC, precision, recall and F1-scores for *valid* and *invalid* bug reports of All Features models in comparison with the random forest models built using features of four dimensions. In Table 17, we find that the All Features models achieve better AUC, F1(v) and F1(i) scores than *reporter experience*$^{\complement}$ and *text*$^{\complement}$ models. However, the All Features models do not show improvements over the *collaboration network*$^{\complement}$, *completeness*$^{\complement}$ and *readability*$^{\complement}$ models. We use Wilcoxon signed-rank test with Bonferroni correction to investigate whether the All Features models statistically significantly improve the models learned using features from four dimensions. And we use Cliff's delta to measure effect size. Table 18 presents the adjusted p-values and Cliff's delta comparing AUC, F1(v) and F1(i) scores for All Features models with the models learned using four dimensions.

From Table 18, we have the following findings:

- The All Features models statistically significantly improve the *reporter experience*$^{\complement}$ models in terms of AUC, F1(v) and F1(i) across the five datasets. All the effect sizes are positive and most of the effect sizes are non-negligible.

TABLE 17
AUC, precision, recall and F1-scores for *valid* and *invalid* bug reports of All Features models in comparison with models built using features of four dimensions. The symbol C denotes the model uses the complement set of features. The best results are in bold.

Project	Approach	AUC	Valid			Invalid		
			F1(v)	P(v)	R(v)	F1(i)	P(i)	R(i)
Eclipse	All Features	**0.76**	**0.88**	**0.82**	**0.96**	**0.35**	**0.63**	0.24
	Reporter Experience^C	0.73	**0.88**	0.81	**0.96**	0.31	0.60	0.21
	Collaboration Network^C	**0.76**	**0.88**	**0.82**	**0.96**	**0.35**	0.62	0.24
	Completeness^C	**0.76**	**0.88**	**0.82**	**0.96**	0.34	**0.63**	0.24
	Readability^C	**0.76**	**0.88**	**0.82**	**0.96**	**0.35**	0.61	**0.25**
	Text^C	0.73	**0.88**	0.81	**0.96**	0.29	0.55	0.20
Netbeans	All Features	**0.73**	**0.76**	**0.72**	**0.80**	**0.55**	**0.61**	0.49
	Reporter Experience^C	0.72	0.75	0.71	**0.80**	0.53	0.60	0.48
	Collaboration Network^C	**0.73**	**0.76**	**0.72**	**0.80**	**0.55**	**0.61**	**0.50**
	Completeness^C	**0.73**	**0.76**	**0.72**	**0.80**	**0.55**	**0.61**	0.49
	Readability^C	**0.73**	0.75	0.71	**0.80**	0.54	**0.61**	0.49
	Text^C	0.69	0.73	0.69	0.79	0.49	0.57	0.44
Mozilla	All Features	**0.83**	**0.86**	**0.82**	**0.90**	**0.62**	**0.71**	0.55
	Reporter Experience^C	0.82	**0.86**	**0.82**	**0.90**	0.61	0.70	0.54
	Collaboration Network^C	0.82	**0.86**	**0.82**	**0.90**	**0.62**	**0.71**	**0.56**
	Completeness^C	0.82	**0.86**	**0.82**	**0.90**	**0.62**	**0.71**	0.55
	Readability^C	0.82	**0.86**	**0.82**	**0.90**	**0.62**	**0.71**	0.55
	Text^C	0.79	0.85	0.80	**0.90**	0.57	0.68	0.49
Firefox	All Features	**0.87**	0.62	**0.70**	0.56	**0.90**	0.88	**0.92**
	Reporter Experience^C	**0.87**	0.61	0.69	0.56	**0.90**	0.88	**0.92**
	Collaboration Network^C	**0.87**	**0.63**	**0.70**	**0.57**	**0.90**	**0.89**	**0.92**
	Completeness^C	**0.87**	0.62	**0.70**	**0.57**	**0.90**	**0.89**	**0.92**
	Readability^C	**0.87**	0.62	**0.70**	0.56	**0.90**	0.88	**0.92**
	Text^C	0.84	0.57	0.67	0.51	0.89	0.87	**0.92**
Thunderbird	All Features	**0.84**	0.59	0.74	**0.51**	**0.92**	**0.89**	0.95
	Reporter Experience^C	0.83	0.58	0.73	0.49	**0.92**	0.88	0.95
	Collaboration Network^C	**0.84**	0.59	**0.75**	0.50	**0.92**	**0.89**	**0.96**
	Completeness^C	**0.84**	0.59	0.73	0.50	**0.92**	**0.89**	0.95
	Readability^C	**0.84**	**0.60**	0.74	**0.51**	**0.92**	**0.89**	0.95
	Text^C	0.82	0.57	0.71	0.48	0.91	0.88	0.95

TABLE 18
Adjusted P-values and Cliff's Delta comparing AUC, F1(v) and F1(i) for All Features models with models built using features of four dimensions.

Project	Approach	AUC		F1(v)		F1(i)	
Eclipse	Reporter Experience^C	0.74	(Large)**	0.18	(Small)**	0.70	(Large)**
	Collaboration Network^C	0.14	(Negligible)	0.02	(Negligible)	-0.04	(Negligible)
	Completeness^C	0.28	(Small)**	0.00	(Negligible)	0.08	(Negligible)
	Readability^C	0.02	(Negligible)	0.12	(Negligible)**	-0.02	(Negligible)
	Text^C	0.74	(Large)**	0.46	(Med)**	0.82	(Large)**
Netbeans	Reporter Experience^C	0.40	(Med)**	0.16	(Small)**	0.34	(Med)*
	Collaboration Network^C	0.08	(Negligible)	0.00	(Negligible)	-0.02	(Negligible)
	Completeness^C	0.02	(Negligible)	0.04	(Negligible)	0.02	(Negligible)
	Readability^C	0.14	(Negligible)	0.10	(Negligible)	0.04	(Negligible)
	Text^C	0.80	(Large)**	0.40	(Med)**	0.68	(Large)**
Mozilla	Reporter Experience^C	0.26	(Small)**	0.16	(Small)**	0.52	(Large)**
	Collaboration Network^C	0.06	(Negligible)	0.06	(Negligible)	0.00	(Negligible)
	Completeness^C	0.06	(Negligible)	-0.04	(Negligible)	0.02	(Negligible)
	Readability^C	0.10	(Negligible)	0.08	(Negligible)	0.06	(Negligible)
	Text^C	0.56	(Large)**	0.40	(Med)**	0.98	(Large)**
Firefox	Reporter Experience^C	0.18	(Small)**	0.10	(Negligible)	0.10	(Negligible)*
	Collaboration Network^C	0.02	(Negligible)	-0.14	(Negligible)	-0.06	(Negligible)
	Completeness^C	0.12	(Negligible)*	-0.02	(Negligible)	0.00	(Negligible)
	Readability^C	0.04	(Negligible)	0.06	(Negligible)	0.00	(Negligible)
	Text^C	0.58	(Large)**	0.30	(Small)**	0.14	(Negligible)**
Thunderbird	Reporter Experience^C	0.16	(Small)**	0.10	(Negligible)	0.14	(Negligible)
	Collaboration Network^C	0.06	(Negligible)	0.00	(Negligible)	-0.04	(Negligible)
	Completeness^C	0.06	(Negligible)	0.04	(Negligible)	0.06	(Negligible)
	Readability^C	0.02	(Negligible)	-0.02	(Negligible)	-0.02	(Negligible)
	Text^C	0.30	(Small)**	0.16	(Small)**	0.22	(Small)*

***p<0.001, **p<0.01, *p<0.05

- The All Features models do not achieve statistically significant improvements over the *collaboration network*^{C} and *readability*^{C} models in terms of AUC, F1(v) and F1(i) across the five datasets.
- The All Features models achieve only one statistically significant improvement with positive and non-negligible effect size over the *completeness*^{C} models. In other cases, the improvements are not significant.
- The All Features models statistically significantly improve the *text*^{C} models in terms of AUC, F1(v) and F1(i) across the five datasets. All the effect sizes are positive. And only one effect size is negligible.

In summary, in the presence of the other four dimensions, the collaboration network, completeness or readability dimension has little contribution to the performance of our approach. On the other hand, the reporter experience and text dimensions have statistically significant contributions—indicating that the two dimensions play a notably important role in our approach. Our findings from the above analysis are consistent with our conclusion in Section 4.3: reporter experience and textual features of bug reports are the most important features in identifying *valid* bug reports.

4.4.2 Individual Features

In this part, we investigate effectiveness of feature subsets in which features are across the five dimensions. We would like to find out a feature subset achieving similar performance of the full features for each dataset. We leverage the results of Section 4.3 to achieve this goal.

In Section 4.3, correlated and redundant features of each dataset are removed. For each dataset, the remaining features are ranked by their importance as shown in Tables 12–16. We focus on two subsets of features for each dataset: top 5 and top 10 most important features. Based on each subset of features, we learn a random forest model. In total, we build two random forest models for each dataset. We denote the random forest models using top 5 and top 10 most important features as TOP5 and TOP10, respectively. We then experiment the two random forest models on each dataset, and compute the average AUC, precision, recall and F1-scores for *valid* and *invalid* bug reports across the 10 folds. Table 19 presents the AUC, precision, recall and F1-scores for *valid* and *invalid* bug reports of the All Features models in comparison with TOP5 and TOP10 models. From the table, we find that TOP5 and TOP10 achieve similar performance of the All Features models in terms of AUC, F1(v) and F1(i) scores. In addition, by comparing the results shown in Tables 19 and 8, we find that TOP5 and TOP10 models outperform the two baselines using Zanetti et al.'s features (i.e., SVMZ and RFZ) in terms of AUC and F1-scores for *valid* and *invalid* bug reports (Zanetti et al.'s models use 9 features).

We use Wilcoxon signed-rank test with Bonferroni correction to investigate whether the All Features models statistically significantly improve TOP5 and TOP10 models in terms of AUC, F1(v) and F1(i). And we use Cliff's delta to measure effect size. Table 20 presents the adjusted p-values and Cliff's delta comparing AUC, F1(v) and F1(i) scores for All Features models with TOP5 and TOP10 models. From the table, we find that the All Features models cannot achieve statistically significant improvements over TOP5 and TOP10 across the five datasets in terms of F1(v) and F1(i).

In summary, models built using top 5 or top 10 important features can effectively determine the validity of a bug report and achieve similar performance of our models that use all features. And our top 5 or top 10 important features are more effective than Zanetti et al.'s 9 features.

5 Discussion

In this section, we further analyze and discuss the performance of our approach in comparison with the baselines in various settings.

5.1 Impact of Tuning SVM Parameters on Our Baseline

By default, we use the default SVM parameters in Weka for SVMZ. Fu et al. found that parameter selection can impact the classification performance of SVM models [28]. Thus, in this section, we would like to investigate whether parameter tuning can improve the performance of SVMZ. We also investigate whether our approach can still outperform SVMZ with parameter tuning in terms of AUC and F1-scores for *valid* and *invalid* bug reports. We use the parameter tuning technique proposed by Fu et al. [28]. In the following paragraphs, we describe the details of SVM parameters, our parameter tuning approach, and evaluation results of SVMZ with parameter tuning.

SVM Parameters: SVM has several parameters which we can use to control its learning process. Following Fu et al.'s study, we focus on four parameters: *C*, *Kernel*, *Gamma* and *Coef0*. Also, we use the same searching range of these parameters in Fu et al.'s study. The parameter *C* is the regularization term in SVM. It controls the tradeoff between training errors and model complexity. A smaller *C* will generate a simple SVM model with more training errors, while a larger *C* leads to a more complex SVM model with fewer training errors. In our study, we tune *C* ranging from 1 to 50. As for the parameter *Kernel*, we use it to introduce different nonlinearities into the SVM model by applying the kernel function to training data. In our study, we tune *Kernel* using four candidate kernel functions: linear kernel, polynomial kernel, radial-based function (rbf) kernel and sigmoid kernel. The parameter *Gamma* controls how far the influence of a single training example reaches—low values mean "far" and high values mean "close". *Gamma* is used in polynomial, rbf and sigmoid kernel functions. We tune *Gamma* ranging from 0 and 1. The parameter *Coef0* is an independent parameter which is used in polynomial and sigmoid kernel functions. In our study, we tune *Coef0* ranging from 0 and 1.

The LibSVM library uses rbf kernel as the default kernel function. By default, it sets C and $Coef0$ as 1 and 0, respectively. The default *Gamma* of LibSVM is $1/k$, where k denotes the number of input features. We use this parameter setting as the default parameter setting of SVM in our study.

Parameter Tuning Approach: We use the parameter tuning approach that is proposed in Fu et al.'s study [28]. We present this approach as follows.

TABLE 19
AUC, precision, recall and F1-scores for *valid* and *invalid* bug reports of All Features models in comparison with models built using top 5 and top 10 most important features. The best results are in bold.

Project	Approach	AUC	Valid			Invalid		
			F1(v)	P(v)	R(v)	F1(i)	P(i)	R(i)
Eclipse	All Features	**0.76**	**0.88**	**0.82**	**0.96**	0.35	**0.63**	0.24
	TOP5	0.73	**0.88**	**0.82**	0.95	**0.37**	0.59	**0.27**
	TOP10	**0.76**	**0.88**	**0.82**	**0.96**	0.36	0.61	0.26
Netbeans	All Features	**0.73**	**0.76**	**0.72**	**0.80**	**0.55**	**0.61**	0.49
	TOP5	0.71	0.74	0.71	0.77	0.53	0.58	**0.50**
	TOP10	0.72	0.75	**0.72**	0.79	**0.55**	0.60	**0.50**
Mozilla	All Features	**0.83**	**0.86**	**0.82**	**0.90**	**0.62**	**0.71**	**0.55**
	TOP5	0.81	0.85	**0.82**	0.89	0.61	0.69	**0.55**
	TOP10	0.82	**0.86**	**0.82**	**0.90**	**0.62**	0.70	**0.55**
Firefox	All Features	**0.87**	**0.62**	**0.70**	**0.56**	**0.90**	**0.88**	**0.92**
	TOP5	0.86	0.61	0.68	0.55	**0.90**	**0.88**	**0.92**
	TOP10	0.86	0.61	0.69	**0.56**	**0.90**	**0.88**	**0.92**
Thunderbird	All Features	**0.84**	**0.59**	**0.74**	**0.51**	**0.92**	**0.89**	0.95
	TOP5	0.82	0.58	0.71	0.50	**0.92**	**0.89**	0.95
	TOP10	0.83	**0.59**	**0.74**	0.50	**0.92**	**0.89**	**0.96**

TABLE 20
Adjusted P-values and Cliff's Delta comparing AUC, F1(v) and F1(i) for All Features models with models built using top 5 and top 10 most important features.

Project	Approach	AUC		F1(v)		F1(i)	
Eclipse	TOP5	0.62	(Large)**	0.28	(Small)	-0.20	(Small)
	TOP10	0.28	(Small)	0.16	(Small)	-0.14	(Negligible)
Netbeans	TOP5	0.48	(Large)**	0.36	(Med)**	0.30	(Small)*
	TOP10	0.28	(Small)**	0.16	(Small)**	0.08	(Negligible)
Mozilla	TOP5	0.38	(Med)**	0.30	(Small)**	0.36	(Med)**
	TOP10	0.24	(Small)*	0.16	(Small)**	0.18	(Small)
Firefox	TOP5	0.36	(Med)**	0.14	(Negligible)**	0.08	(Negligible)**
	TOP10	0.24	(Small)**	0.10	(Negligible)	0.00	(Negligible)
Thunderbird	TOP5	0.28	(Small)**	0.08	(Negligible)**	0.12	(Negligible)*
	TOP10	0.18	(Small)**	0.02	(Negligible)	0.02	(Negligible)

***p<0.001, **p<0.01, *p<0.05

In each fold, we first use stratified random sampling technique to split the training dataset into two subsets. The first subset contains 80% bug reports of the training dataset, which is used for training when tuning parameters. To distinguish the training dataset and the first subset, we call the first subset as *tuning-training dataset*. The second subset contains the remaining bug reports, which is used for validation when tuning parameters. We call the second subset as *validation dataset*.

When tuning parameters, we train SVM models on the tuning-training dataset using different combinations of four parameters and evaluate the performance of these models on the validation dataset. Our aim is to find a combination of parameters to optimize an objective function on validation dataset. As shown in Table 1, class distributions of our datasets are imbalanced. In the Eclipse, Netbeans and Mozilla datasets, bug reports of the minority class are *invalid*. In the Firefox and Thunderbird datasets, bug reports of the minority class are *valid*. In Section 4.1, we find that using default parameter setting in Weka, SVMZ suffers from the data imbalance problem. It achieves a much lower F1-score for the minority class than our approach on the five datasets. Thus, we use the F1-score for the minority class as our objective function. When tuning parameters, we aim to maximize the F1-score for the minority class on validation dataset.

When searching the optimal combination of parameters, we follow Fu et al.' study [28] and use the differential evolution technique [20], [73]. Differential evolution (DE) is a heuristic approach that optimizes a problem by iteratively trying to improve a candidate solution with regard to a give objective function [73]. The technique has been used in many previous software engineering studies [2], [28], [29]. When tuning parameters, we take the same DE steps presented in Fu et al.'s study [28] to find a combination of parameters which maximize the objective function. Then we use the combination of parameters to train an SVM model on the tuning-training dataset, and evaluate its performance on the testing dataset of the fold.

Evaluation Results: In this section and the following text, we use SVMZ denote SVMZ using the default parameter setting. And we denote SVMZ with the parameter tuning approach as T-SVMZ. For each dataset, we calculate the average AUC and F1-scores for *valid* and *invalid* bug reports of T-SVMZ across the 10 folds. Table 21 presents the average AUC and F1-scores for *valid* and *invalid* bug reports of our approach in comparison with SVMZ and T-SVMZ. From the table, we find that T-SVMZ improves SVMZ in terms of F1-scores for the minority class on each dataset. For example, on Eclipse dataset, SVMZ cannot identify *invalid* bug reports—its F1(i) is 0. With parameter tuning approach, T-SVMZ achieves an F1(i) of 0.12. We also find

TABLE 21
AUC and F1-scores for *valid* and *invalid* bug reports of our approach in comparison with SVMZ using the default parameter setting and SVMZ using the parameter tuning approach. The best results are in bold.

Project	Approach	AUC	F1(v)	F1(i)
Eclipse	Ours	**0.76**	**0.88**	**0.35**
	SVMZ	0.50	**0.88**	0.00
	T-SVMZ	0.46	0.78	0.12
Netbeans	Ours	**0.73**	**0.76**	**0.55**
	SVMZ	0.58	0.69	0.39
	T-SVMZ	0.59	0.65	0.51
Mozilla	Ours	**0.83**	**0.86**	**0.62**
	SVMZ	0.58	0.81	0.33
	T-SVMZ	0.58	0.75	0.40
Firefox	Ours	**0.87**	**0.62**	**0.90**
	SVMZ	0.70	0.55	0.88
	T-SVMZ	0.74	0.60	0.88
Thunderbird	Ours	**0.84**	**0.59**	**0.92**
	SVMZ	0.68	0.49	0.90
	T-SVMZ	0.72	0.56	0.90

that T-SVMZ outperforms SVMZ in terms of AUC, F1(v) and F1(i). However, as shown in Table 21, our approach achieves the best AUC, F1(v) and F1(i) scores across the five datasets. In terms of AUC, F1(v) and F1(i), our approach on average outperforms T-SVMZ by 31%, 10% and 20%, respectively. Thus, our approach still outperforms SVMZ with the parameter tuning approach.

5.2 Impact of Different Underlying Classifiers

In this paper, we use random forest as the default underlying classifier to evaluate the performance of our approach. Here, we would like to investigate the impact of different underlying classifiers on the classification performance. In addition to random forest, we also use other classifiers namely SVM, naive Bayes and decision tree [35]. These classifiers are widely used in past software engineering studies [41], [44], [81]. For naive Bayes and decision tree, we apply the default implementation in Weka [34]. And we use the **LibSVM** package in Weka as our SVM implementation. In addition to evaluate the performance of SVM models with the default parameter setting, we also evaluate the performance of SVM models using the parameter tuning approach introduced in Section 5.1. We denote SVM models using the parameter tuning approach as T-SVM.

Table 22 presents the AUC and F1-scores for *valid* and *invalid* bug reports of models built on our features using different underlying classifiers. In Table 22, we notice that the random forest models achieve the best performance in terms of AUC across the five datasets. And the random forest and naive Bayes models achieve better performance in terms of F1-scores for *valid* and *invalid* bug reports than the other three models. In most cases, the random forest models show better performance as compared to models using the other classifiers in terms of AUC and F1-scores.

We notice that the T-SVM models achieve better classification performance than the SVM models, especially on the Firefox and Thunderbird datasets. The SVM models suffer from imbalanced data and achieve low F1-scores for *valid* bug reports on the two datasets. With the parameter tuning approach, T-SVM models achieves better AUC, F1(v) and F1(i) scores than SVM models. However, the random forest models still achieve better classification performance than T-SVM in terms of AUC, F1(v) and F1(i) scores.

TABLE 22
AUC and F1-scores for *valid* and *invalid* bug reports of models built on our features using different underlying classifiers. The best results are in bold.

Project	Approach	AUC	F1(v)	F1(i)
Eclipse	Random Forest	**0.76**	**0.88**	0.35
	SVM	0.53	**0.88**	0.14
	T-SVM	0.55	0.82	0.29
	Naive Bayes	0.72	0.85	**0.47**
	Decision Tree	0.64	0.87	0.37
Netbeans	Random Forest	**0.73**	**0.76**	0.55
	SVM	0.57	**0.76**	0.32
	T-SVM	0.61	0.70	0.53
	Naive Bayes	0.70	0.70	**0.58**
	Decision Tree	0.61	0.71	0.52
Mozilla	Random Forest	**0.83**	**0.86**	**0.62**
	SVM	0.59	0.84	0.34
	T-SVM	0.66	0.85	0.51
	Naive Bayes	0.80	0.84	0.60
	Decision Tree	0.73	0.84	0.59
Firefox	Random Forest	**0.87**	0.62	**0.90**
	SVM	0.55	0.15	0.86
	T-SVM	0.73	0.59	0.89
	Naive Bayes	0.85	**0.65**	0.88
	Decision Tree	0.75	0.60	0.89
Thunderbird	Random Forest	**0.84**	0.59	**0.92**
	SVM	0.53	0.07	0.88
	T-SVM	0.71	0.54	0.90
	Naive Bayes	0.83	**0.62**	0.90
	Decision Tree	0.70	0.56	0.90

In practice, to achieve the best performance for our approach, we recommend developers to use random forest as the underlying classifier to build models.

Here, we would like to investigate why random forest outperforms T-SVM. We analyze the classification errors made by random forest and T-SVM. This may also shed lights on an avenue to further improve the performance of our approach. In Table 22, we notice that on the Eclipse dataset, random forest substantially outperforms T-SVM in terms of AUC, F1(v) and F1(i). Thus, we investigate the difference of the incorrectly classified reports between random forest and T-SVM on the Eclipse dataset.

For the Eclipse dataset, we record the bug reports which are incorrectly labeled by the classifier (random forest or T-SVM) in each fold. And we combine these incorrectly labeled bug reports across the 10 folds. In total, across the 10 folds, the random forest model incorrectly labels 21,503 bug reports, in which 3,295 (15%) ones are truly *valid* and 18,208 (85%) ones are truly *invalid*. And the T-SVM model in total incorrectly labels 31,661 bug reports, in which 14,611 (46%) ones are truly *valid* and 17,050 (54%) ones are truly *invalid*. The T-SVM model incorrectly labels many more truly *valid* bug reports than the random forest model. Hence, T-SVM achieves much worse classification performance than random forest.

We use a box-plot [88] to observe the distributions of our features in the truly *valid* and *invalid* bug reports which are incorrectly labeled by the random forest model and T-SVM model. A box-plot is a graphical method for depicting the distribution of numerical data based on minimum (0%), first quartile (25%), median (50%), third quartile (75%), and

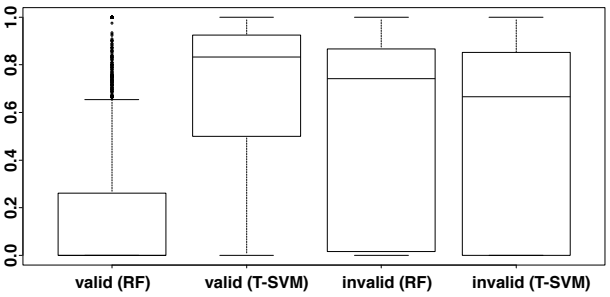

Fig. 2. Distributions of *valid-rate* in the truly *valid* and *invalid* bug reports that are incorrectly labeled by random forest and T-SVM on the Eclipse dataset.

maximum (100%). We find that the box plots, which show the distributions of *valid-rate* of the incorrectly labeled bug reports, present visible differences between the two models. Notice that *valid-rate* of a bug report refers to the valid rate of the prior reports with known labels as submitted by the reporter of this bug report. A larger *valid-rate* indicates that the reporter is more likely to have more experience. In the following paragraphs, we focus on using *valid-rate* to analyze the classification errors made by the two models.

Figure 2 presents the box plots showing the distributions of *valid-rate* in the truly *valid* and *invalid* bug reports that are incorrectly labeled by random forest and T-SVM on the Eclipse dataset. In the figure, valid (RF) and valid (T-SVM) denote the truly *valid* bug reports that are incorrectly labeled as *invalid* by the random forest and T-SVM models, respectively. And invalid (RF) and invalid (T-SVM) denote the truly *invalid* bug reports that are incorrectly labeled as *valid* by the random forest and T-SVM models, respectively.

From Figure 2, we find that box plots showing the distributions of *valid-rate* for valid (RF) and valid (T-SVM) are visibly different. For random forest, most of the incorrectly labeled truly *valid* reports by random forest have a *valid-rate* less than 0.3, while for T-SVM, most of those reports have a *valid-rate* larger than 0.5. In Section 4.3, we find that *valid-rate* is an important feature in identifying *valid* reports from *invalid* ones and it has a positive effect size. It indicates that a bug report with a larger *valid-rate* is more likely to be *valid*. T-SVM classifies a large number of bug reports with high *valid-rate* as *invalid*, while random forest does not have such a problem. This observation helps explain why T-SVM incorrectly labels many more truly *valid* bug reports than random forest. Here, we present more empirical evidence to verify this claim.

In the incorrectly labeled bug reports of random forest and T-SVM, we find that only 4% of the reports have a *valid-rate* in the range (0.1, 0.5). Thus, we could ignore these bug reports and focus more on the bug reports having a *valid-rate* in the range [0, 0.1] or [0.5, 1]. We count that bug reports with a *valid-rate* $\leq$ 0.1 are submitted by inexperienced reporters. And we count that bug reports with a *valid-rate* $\geq$ 0.5 are submitted by experienced reporters. Then, for random forest and T-SVM, we categorize incorrectly labeled bug reports having a *valid-rate* in the range [0, 0.1] or [0.5, 1] as follows.

The following are the categories of errors made by random forest:

1. Inexperienced bug reporters who submit *valid* bug reports (*valid-rate* $\leq$ 0.1). This account for 11% of the errors (including 2403 bug reports).
2. Experienced bug reporters who submit *invalid* bug reports (*valid-rate* $\geq$ 0.5). This account for 55% of the errors (including 11868 bug reports).
3. Inexperienced bug reporters who submit *invalid* bug reports (*valid-rate* $\leq$ 0.1). This account for 26% of the errors (including 5644 bug reports).
4. Experienced bug reporters who submit *valid* bug reports (*valid-rate* $\geq$ 0.5). This account for 3% of the errors (including 624 bug reports).

The following are the categories of errors made by T-SVM:

1. Inexperienced bug reporters who submit *valid* bug reports (*valid-rate* $\leq$ 0.1). This account for 11% of the errors (including 3335 bug reports).
2. Experienced bug reporters who submit *invalid* bug reports (*valid-rate* $\geq$ 0.5). This account for 31% of the errors (including 9838 bug reports).
3. Inexperienced bug reporters who submit *invalid* bug reports (*valid-rate* $\leq$ 0.1).This account for 20% of the errors (including 6331 bug reports).
4. Experienced bug reporters who submit *valid* bug reports (*valid-rate* $\geq$ 0.5). This account for 35% of the errors (including 11024 bug reports).

Based on the above analysis, we can notice that T-SVM incorrectly labels many more *valid* bug reports which are submitted by experienced reporters, which is the most different error category in comparison with random forest. This results in that T-SVM achieves much worse performance than random forest.

5.3 Impact of Imbalanced Data

From Table 3.1, we notice that all of our datasets are imbalanced. In the Eclipse, Netbeans and Mozilla datasets, *valid* bug reports are the majority class and *invalid* ones are the minority class. In the Firefox and Thunderbird datasets, *invalid* bug reports are the majority class and *valid* ones are the minority class. By default, our approach uses the original imbalanced training dataset to learn a model. In this discussion, we would like to investigate whether we can further improve the performance of our approach by dealing with the data imbalance problem.

To deal with the imbalanced data, we re-balance the training dataset in each fold and use the re-balanced dataset to train a model. We re-balance the data using a re-sampling technique that removes the bug reports from the majority class (under-sampling) and repeats the bug reports from the minority class (over-sampling). Notice that in each fold, we only re-balance the training dataset and the original testing dataset is not re-balanced.

We denote our model learned from the original imbalanced training dataset as Default. And we denote our model learned from the re-balanced training dataset as Rebalanced. Table 23 presents AUC and F1-scores for *valid* and *invalid* bug reports of Rebalanced models in comparison with Default models. In the table, we find that across the five datasets, the Rebalanced models achieve better F1-scores for

TABLE 23
AUC and F1-scores for *valid* and *invalid* bug reports of our models that use the re-balanced training dataset (i.e., Rebalanced) in comparison with our models that use the original imbalanced training dataset (i.e., Default).

Project	Approach	AUC	F1(v)	F1(i)
Eclipse	Default	**0.76**	**0.88**	0.35
	Rebalanced	**0.76**	0.85	**0.48**
Netbeans	Default	**0.73**	**0.76**	0.55
	Rebalanced	0.72	0.72	**0.60**
Mozilla	Default	**0.83**	**0.86**	0.62
	Rebalanced	**0.83**	0.84	**0.65**
Firefox	Default	**0.87**	0.62	**0.90**
	Rebalanced	**0.86**	**0.66**	0.89
Thunderbird	Default	**0.84**	0.59	**0.92**
	Rebalanced	**0.84**	**0.62**	0.90

the minority class. But in terms of AUC, the Rebalanced and Default models achieve similar performance.

In summary, dealing with imbalanced data can improve the F1-score for the minority class of our approach. But it does not improve the AUC of our approach.

5.4 Impact of Feature Selection

In Section 4.3, we apply a feature selection method when we investigate the most important features. Feature selection is also widely used to improve performance of classifiers in previous studies [32]. In this section, we would like to investigate whether automated feature selection methods can further improve the performance of our approach. Feature selection can be performed in two phases of our approach. First, feature selection can be applied from our evaluation experiment. Second, feature selection can be applied when we calculate textual features for bug reports. Here, we discuss the impact of automated feature selection methods when they are applied in both two cases with regard to the classification performance of our approach.

For convenience, we denote our model without using automatic feature selection methods as Default. And we denote the feature selection method that is used in Section 4.3 as CRA, with "C", "R" and "A" denotes "correlation", "redundancy" and "analysis", respectively.

5.4.1 Feature Selection in Evaluation

In this part, we apply feature selection to our features from our evaluation experiment. And we use the longitudinal data setup to evaluate the models. In each fold, we first use CRA to remove correlated and redundant features for the training dataset. Then, we build a model using the preprocessed training dataset and evaluate the model on the testing dataset. Finally, we calculate the average AUC and F1-scores for *valid* and *invalid* bug reports across the 10 folds. We denote our model built with CRA applied as Default+CRA. Table 24 presents the AUC and F1-scores for *valid* and *invalid* bug reports of Default+CRA models in comparison with Default models. In the table, we find that across the five datasets, the Default+CRA models achieve a slightly lower performance (not statistically significant difference) in terms of AUC, F1(v) and F1(i) in comparison with the Default models.

TABLE 24
AUC and F1-scores for *valid* and *invalid* bug reports of our approach with feature selection applied when building models in comparison with our models without applying feature selection methods.

Project	Approach	AUC	F1(v)	F1(i)
Eclipse	Default	**0.76**	**0.88**	**0.35**
	Default+CRA	**0.76**	**0.88**	0.34
Netbeans	Default	**0.73**	**0.76**	**0.55**
	Default+CRA	0.72	0.75	0.54
Mozilla	Default	**0.83**	**0.86**	**0.62**
	Default+CRA	0.82	**0.86**	0.61
Firefox	Default	**0.87**	**0.62**	**0.90**
	Default+CRA	0.86	0.61	**0.90**
Thunderbird	Default	**0.84**	**0.59**	**0.92**
	Default+CRA	0.83	0.57	0.91

5.4.2 Feature Selection in Textual Feature Extraction

When extracting the textual features, we learn classifiers using token features of bug reports. Then, we use the likelihood scores to be valid of the bug reports output by the classifiers as textual features. In this part, we apply feature selection to the token features when we learn the classifiers. We investigate whether applying feature selection to the token features impacts the performance of our approach. We investigate the impact of two feature selection methods: CRA and Information Gain (IG). IG is a widely used feature selection method to select useful features for text classification in previous studies [68], [98]. Recently, Huang et al. [39] found that IG can improve the performance of their text classification classifiers. Thus, in this part, we investigate impact of applying IG to the token features of bug reports. When applying IG, we choose top 5% of the total number of token features.

In each fold, to extract textual features for the training dataset, we first split the training dataset into two subsets of equal sizes using stratified random sampling. Next, we apply a feature selection method (CRA or IG) to token features of each subset. Then, we train classifiers based on the selected token features of one subset and use the classifiers to calculate textual features for the other subset. To extract textual features for the testing dataset, we apply the same feature selection method to token features of the whole training dataset. Then we train classifiers based on the selected token features of the training dataset and use the classifiers to calculate textual features for the testing dataset. Since we separately extract textual features for summary and description of bug reports, we need to perform six runs of feature selection in each fold (three runs for summary of bug reports and three runs for description of bug reports).

We name the models with CRA and IG applied when extracting textual features as Default+TEXT-CRA and Default+TEXT-IG, respectively. Table 25 presents AUC and F1-scores for *valid* and *invalid* bug reports of Default+TEXT-CRA and Default+TEXT-IG models in comparison with the Default models. In the table, we find that applying CRA or IG when extracting textual features does not improve the performance of our approach in terms of AUC, F1(v) and F1(i) scores.

In summary, applying automated feature selection method does not improve the performance of our approach.

TABLE 25
AUC and F1-scores for *valid* and *invalid* bug reports of our approach with feature selection methods applied when extracting textual features in comparison with our models without applying feature selection methods.

Project	Approach	AUC	F1(v)	F1(i)
Eclipse	Default	**0.76**	**0.88**	**0.35**
	Default+TEXT-CRA	**0.76**	**0.88**	**0.35**
	Default+TEXT-IG	**0.76**	**0.88**	**0.35**
Netbeans	Default	**0.73**	**0.76**	**0.55**
	Default+TEXT-CRA	**0.73**	0.75	**0.55**
	Default+TEXT-IG	0.72	0.75	0.53
Mozilla	Default	**0.83**	**0.86**	**0.62**
	Default+TEXT-CRA	**0.83**	**0.86**	**0.62**
	Default+TEXT-IG	0.82	**0.86**	0.61
Firefox	Default	**0.87**	**0.62**	**0.90**
	Default+TEXT-CRA	**0.87**	**0.62**	**0.90**
	Default+TEXT-IG	**0.87**	0.61	**0.90**
Thunderbird	Default	**0.84**	**0.59**	**0.92**
	Default+TEXT-CRA	**0.84**	**0.59**	**0.92**
	Default+TEXT-IG	**0.84**	**0.59**	**0.92**

5.5 Impact of Using N-gram Features

In this study, we use eight textual features to characterize the textual contents (i.e., summary and description) of bug reports. To calculate these textual features, we leverage four classifiers (i.e., naive Bayes classifier and its variants) to convert the token features that are extracted from summary and description of bug reports into numerical scores. We denote our approach that uses this default setting as Ours-Default.

Many prior studies used N-gram features to perform text classification tasks, where an N-gram refers to a consecutive sequence of N tokens from a text [17]. In this discussion, we would like to investigate the impact of using N-gram features to determine the validity of a bug report. We use three types of N-gram features together, namely 1-gram features (i.e., token features), 2-gram features and 3-gram features. We investigate the impact of using the three types of N-gram features together to determine the validity of a bug report. We do this in two ways: 1) investigating the effectiveness of the N-gram features of bug reports in identifying *valid* bug reports; 2) investigating the performance of our approach when we use the N-gram features of summary and description of bug reports to calculate our eight textual features.

To achieve 1), we compare the performance of our approach that uses the default setting and text classification models that are built using the N-gram features of bug reports. Since we leverage four classifiers including naive Bayes classifier and its variants to calculate our textual features, we also use these four classifiers as underlying classifiers to build the text classification models. Our approach leverages both summary and description of bug reports. To leverage these two textual contents for the text classification models, we combine the summary and description for each bug report, and extract 1-gram, 2-gram and 3-gram features from the combined text. Then, we use the three types of N-gram features together to build text classification models. We name the text classification models using naive Bayes classifier, multinomial naive Bayes classifier, discriminative multinomial naive Bayes classifier and complement naive Bayes classifier as NB-NGRAM, MNB-NGRAM, DMNB-NGRAM and CNB-NGRAM, respectively. We use the longitudinal data setup presented in Section 3.4 to evaluate these models. And we calculate the average AUC and F1-scores for *valid* and *invalid* bug reports across the 10 folds.

TABLE 26
AUC and F1-scores for *valid* and *invalid* bug reports of our approach that uses default setting (i.e., Ours-Default) in comparison with models trained using N-gram features (i.e., NB-NGRAM, MNB-NGRAM, DMNB-NGRAM and CNB-NGRAM) and our approach that uses N-gram features of bug reports to calculate the eight textual features (i.e., Ours-NGRAM).

Project	Approach	AUC	F1(v)	F1(i)
Eclipse	Ours-Default	**0.76**	**0.88**	0.35
	NB-NGRAM	0.65	0.83	**0.36**
	MNB-NGRAM	0.61	0.87	0.27
	DMNB-NGRAM	0.66	0.87	0.33
	CNB-NGRAM	0.58	0.87	0.29
	Ours-NGRAM	**0.76**	**0.88**	0.34
Netbeans	Ours-Default	**0.73**	**0.76**	**0.55**
	NB-NGRAM	0.61	0.69	0.43
	MNB-NGRAM	0.66	0.74	0.47
	DMNB-NGRAM	0.67	0.73	0.52
	CNB-NGRAM	0.61	0.74	0.48
	Ours-NGRAM	**0.73**	0.75	0.54
Mozilla	Ours-Default	**0.83**	**0.86**	**0.62**
	NB-NGRAM	0.73	0.79	0.52
	MNB-NGRAM	0.76	0.85	0.58
	DMNB-NGRAM	0.79	0.84	0.60
	CNB-NGRAM	0.71	0.84	0.58
	Ours-NGRAM	**0.83**	**0.86**	**0.62**
Firefox	Ours-Default	**0.87**	**0.62**	**0.90**
	NB-NGRAM	0.74	0.53	0.82
	MNB-NGRAM	0.83	0.61	0.85
	DMNB-NGRAM	0.84	0.58	0.89
	CNB-NGRAM	0.79	0.61	0.85
	Ours-NGRAM	**0.87**	0.61	**0.90**
Thunderbird	Ours-Default	**0.84**	**0.59**	**0.92**
	NB-NGRAM	0.71	0.48	0.83
	MNB-NGRAM	0.81	0.56	0.85
	DMNB-NGRAM	0.81	0.55	0.90
	CNB-NGRAM	0.75	0.56	0.84
	Ours-NGRAM	**0.84**	**0.59**	**0.92**

To achieve 2), we compare the performance of our approach that uses the default setting and our approach that uses N-gram features of bug reports to calculate the eight textual features. We first separately extract N-gram features (including 1-gram, 2-gram and 3-gram features) from summary and description of bug reports. Then, we apply the method presented in Section 3.2 to convert the N-gram features into the eight textual features for the reports. We denote our approach using the three types of N-gram features to calculate the textual features as Ours-NGRAM. To evaluate the performance of Ours-NGRAM, we use the longitudinal data setup introduced in Section 3.4 and calculate the average AUC and F1-scores for *valid* and *invalid* bug reports of Ours-NGRAM across the 10 folds.

Table 26 presents the AUC and F1-scores for *valid* and *invalid* bug reports of our approach that uses default setting in comparison with models using N-gram features (including 1-gram, 2-gram and 3-gram features) and our approach that uses N-gram features to calculate the eight textual features.

From Table 26, we find that in most cases, our approach

that uses default setting achieves better performance than the NB-NGRAM, MNB-NGRAM, DMNB-NGRAM and CNB-NGRAM models in terms of AUC, F1(v) and F1(i)—indicating that the 33 features that we use in our approach to determine the validity of a bug report are more effective than the N-gram features of bug reports.

From the table, we also find that across the five datasets, Ours-Default and Ours-NGRAM achieve similar performance in terms of AUC, F1(v) and F1(i). Ours-Default only uses token features (i.e., 1-gram features) to calculate the eight textual features, while Ours-NGRAM uses more N-gram features (i.e., 2-gram and 3-gram features) in addition to features used by Ours-Default to do that. However, our experimental results show that the additional 2-gram and 3-gram features that are used by Ours-NGRAM cannot further improve the performance of our approach. Moreover, the additional 2-gram and 3-gram features cause more computation costs. Hence, our default setting is preferred over Ours-NGRAM.

In summary, our features are more effective than N-gram features of bug reports in determining the validity of a bug report. Moreover, we recommend to use token features to calculate the eight textual features when using our approach in practice.

5.6 Cross-validation Setup

By default, we use longitudinal data setup to evaluate the effectiveness of our approach and the baselines in this paper. Still, in many software engineering studies, cross-validation is used as the evaluation setting [65], [83], [97]. Thus, for completeness sake, here, we would like to compare the effectiveness of our approach with SVMZ and RFZ when we use 10-times stratified 10-fold cross-validation data setup.

We perform 10 times stratified 10-fold cross-validation. In each run of stratified 10-fold cross-validation, the dataset for each project are first randomly divided into 10 folds using stratified random sampling. Stratified random sampling technique can keep the class distribution of each fold the same as the original dataset. Of the 10 folds, a single fold is retained as the testing dataset, and the remaining nine folds are used as training dataset. The process is then repeated 10 times, with each of the 10 folds used exactly once as the testing dataset. In each fold, we use our approach present in Section 3.2 to convert token features of training and testing datasets into textual scores. Then we calculate the average AUC scores and F1-scores for *valid* and *invalid* bug reports across the 10 runs of stratified 10-fold cross-validation. We empirically find that time efficiency of tuning SVM parameters for SVMZ is unacceptable in this setting. Thus, in this section, we do not tune SVM parameters for SVMZ.

Table 27 presents the AUC and F1-scores for *valid* and *invalid* bug reports of our approach in comparison with the baselines when we use 10-times stratified 10-fold cross-validation data setup. On average, across the five datasets, our approach achieves an AUC, F1(v) and F1(i) of 0.83, 0.77 and 0.71, respectively. In comparison with the baselines, our approach achieves better performance in terms of AUC, F1(v) and F1(i) across the five datasets. Similar to RQ1 and RQ2, we apply Wilcoxon signed-rank test with Bonferroni correction to investigate whether the improvements of our approach over the baselines are statistically significant. We also use Cliff's delta to measure effect size. Table 28 presents the adjusted p-values and Cliff's delta comparing AUC scores for our approach with the baselines in 10-times stratified 10-fold cross validation.

TABLE 27
AUC and F1-scores for *valid* and *invalid* bug reports of our approach in comparison with the baselines when we use 10-times stratified 10-fold cross-validation data setup. The best results are in bold.

Project	Approach	AUC	F1(v)	F1(i)
Eclipse	Ours	**0.81**	**0.90**	**0.49**
	SVMZ	0.52	0.88	0.09
	RFZ	0.68	0.88	0.13
Netbeans	Ours	**0.75**	**0.77**	**0.57**
	SVMZ	0.51	0.76	0.07
	RFZ	0.65	0.75	0.17
Mozilla	Ours	**0.85**	**0.87**	**0.65**
	SVMZ	0.63	0.81	0.48
	RFZ	0.75	0.80	0.56
Firefox	Ours	**0.89**	**0.67**	**0.91**
	SVMZ	0.75	0.62	0.90
	RFZ	0.80	0.61	0.90
Thunderbird	Ours	**0.86**	**0.64**	**0.92**
	SVMZ	0.72	0.58	0.91
	RFZ	0.76	0.58	0.91

TABLE 28
Adjusted P-values and Cliff's Delta comparing AUC, F1(v) and F1(i) for our approach with the baselines in 10-times stratified 10-fold cross-validation.

Project	Approach	AUC	F1(v)	F1(i)
Eclipse	SVMZ	1.00 (Large)**	1.00 (Large)**	1.00 (Large)**
	RFZ	1.00 (Large)**	1.00 (Large)**	1.00 (Large)**
Netbeans	SVMZ	1.00 (Large)**	1.00 (Large)**	1.00 (Large)**
	RFZ	1.00 (Large)**	1.00 (Large)**	1.00 (Large)**
Mozilla	SVMZ	1.00 (Large)**	1.00 (Large)**	1.00 (Large)**
	RFZ	1.00 (Large)**	1.00 (Large)**	1.00 (Large)**
Firefox	SVMZ	1.00 (Large)**	1.00 (Large)**	1.00 (Large)**
	RFZ	1.00 (Large)**	1.00 (Large)**	1.00 (Large)**
Thunderbird	SVMZ	1.00 (Large)**	0.98 (Large)**	0.98 (Large)**
	RFZ	1.00 (Large)**	1.00 (Large)**	1.00 (Large)**

***p<0.001, **p<0.01, *p<0.05

Based on the results shown in Tables 27 and 28, we have the following findings:

- In terms of AUC, our approach on average outperforms SVMZ and RFZ by 32% and 14%, respectively. Statistical tests show that the improvements are significant and all the effect sizes are large.
- In terms of F1-scores for *valid* bug reports, i.e., F1(v), our approach on average outperforms SVMZ and RFZ by 5% and 7%, respectively. Statistical tests show that the improvements are significant and all the effect sizes are large.
- In terms of F1-scores for *invalid* bug reports, i.e., F1(i), our approach on average outperforms SVMZ and RFZ by 45% and 34%, respectively. Statistical tests show that the improvements are significant and all the effect sizes are large.

5.7 Cross-project Setting

In the experiment setting of RQ1 and RQ2, for each project, we train a model using the historical bug reports with

known labels within the project. However, a newly built project may not possess enough bug report data to build a model. In this section, we would like to investigate the effectiveness of our approach for determining *valid* and *invalid* bug reports in the cross-project setting. For each project, we first build a model using data from it. We denote the project as *source project*. Then, we use the model to determine the validity of bug reports in the other projects. We denote these projects as *target projects*.

In the cross-project setting, we use bug report data in *source project* as training dataset and use data in *target project* as testing dataset. We use the approach presented in Section 3.2 to convert token features into textual scores for training and testing datasets. Similar to Section 5.6, we empirically find that time efficiency of tuning SVM parameters for SVMZ is unacceptable in the cross-project setting. Thus, we also do not tune SVM parameters for SVMZ.

Tables 29–33 present the AUC and F1-scores for *valid* and *invalid* bug reports of our approach in comparison with the baselines when we use one of the five projects (i.e., Eclipse, Netbeans, Mozilla, Firefox and Thunderbird) as a *source project* and another as a *target project*. As shown in the tables, our approach achieves the best AUC scores in all cases in comparison with the baselines. In terms of F1(v) and F1(i), our approach also achieves better performance than SVMZ and RFZ in most cases.

TABLE 29
AUC and F1-scores for *valid* and *invalid* bug reports of our approach in comparison with the baselines in cross-project setting when using Eclipse as source project. The best results are in bold.

Source & Target Project	Approach	AUC	F1(v)	F1(i)
Eclipse ⇒ Netbeans	Ours	**0.70**	0.74	**0.51**
	SVMZ	0.50	**0.76**	0.00
	RFZ	0.60	**0.76**	0.04
Eclipse ⇒ Mozilla	Ours	**0.74**	**0.83**	**0.22**
	SVMZ	0.50	0.82	0.00
	RFZ	0.57	0.79	0.02
Eclipse ⇒ Firefox	Ours	**0.80**	**0.48**	**0.62**
	SVMZ	0.50	0.37	0.00
	RFZ	0.77	0.37	0.01
Eclipse ⇒ Thunderbird	Ours	**0.79**	**0.43**	**0.59**
	SVMZ	0.50	0.35	0.00
	RFZ	0.74	0.35	0.01

TABLE 30
AUC and F1-scores for *valid* and *invalid* bug reports of our approach in comparison with the baselines in cross-project setting when using Netbeans as source project. The best results are in bold.

Source & Target Project	Approach	AUC	F1(v)	F1(i)
Netbeans ⇒ Eclipse	Ours	**0.74**	0.87	**0.38**
	SVMZ	0.50	**0.88**	0.00
	RFZ	0.62	**0.88**	0.01
Netbeans ⇒ Mozilla	Ours	**0.73**	**0.83**	**0.33**
	SVMZ	0.50	0.82	0.01
	RFZ	0.58	0.80	0.03
Netbeans ⇒ Firefox	Ours	**0.82**	**0.50**	**0.65**
	SVMZ	0.50	0.37	0.00
	RFZ	0.78	0.37	0.01
Netbeans ⇒ Thunderbird	Ours	**0.79**	**0.45**	**0.65**
	SVMZ	0.50	0.34	0.00
	RFZ	0.75	0.35	0.01

TABLE 31
AUC and F1-scores for *valid* and *invalid* bug reports of our approach in comparison with the baselines in cross-project setting when using Mozilla as source project. The best results are in bold.

Source & Target Project	Approach	AUC	F1(v)	F1(i)
Mozilla ⇒ Eclipse	Ours	**0.74**	**0.88**	0.35
	SVMZ	0.62	0.73	**0.41**
	RFZ	0.61	0.72	**0.41**
Mozilla ⇒ Netbeans	Ours	**0.67**	**0.75**	0.44
	SVMZ	0.61	0.66	**0.55**
	RFZ	0.59	0.62	0.56
Mozilla ⇒ Firefox	Ours	**0.90**	**0.71**	**0.92**
	SVMZ	0.78	0.63	0.87
	RFZ	0.77	0.63	0.88
Mozilla ⇒ Thunderbird	Ours	**0.85**	**0.64**	**0.91**
	SVMZ	0.75	0.59	0.88
	RFZ	0.74	0.58	0.88

TABLE 32
AUC and F1-scores for *valid* and *invalid* bug reports of our approach in comparison with the baselines in cross-project setting when using Firefox as source project. The best results are in bold.

Source & Target Project	Approach	AUC	F1(v)	F1(i)
Firefox ⇒ Eclipse	Ours	**0.73**	**0.76**	**0.47**
	SVMZ	0.57	0.54	0.38
	RFZ	0.62	0.46	0.37
Firefox ⇒ Netbeans	Ours	**0.69**	**0.54**	**0.60**
	SVMZ	0.61	0.53	0.58
	RFZ	0.62	**0.54**	0.58
Firefox ⇒ Mozilla	Ours	**0.78**	**0.73**	**0.59**
	SVMZ	0.59	0.71	0.45
	RFZ	0.63	0.64	0.48
Firefox ⇒ Thunderbird	Ours	**0.85**	**0.62**	**0.92**
	SVMZ	0.71	0.56	0.90
	RFZ	0.76	0.49	0.90

TABLE 33
AUC and F1-scores for *valid* and *invalid* bug reports of our approach in comparison with the baselines in cross-project setting when using Thunderbird as source project. The best results are in bold.

Source & Target Project	Approach	AUC	F1(v)	F1(i)
Thunderbird ⇒ Eclipse	Ours	**0.73**	**0.78**	**0.48**
	SVMZ	0.56	0.53	0.37
	RFZ	0.62	0.50	0.37
Thunderbird ⇒ Netbeans	Ours	**0.70**	**0.53**	**0.61**
	SVMZ	0.61	0.52	0.59
	RFZ	0.62	0.50	0.59
Thunderbird ⇒ Mozilla	Ours	**0.73**	**0.73**	**0.53**
	SVMZ	0.58	0.70	0.44
	RFZ	0.60	0.67	0.48
Thunderbird ⇒ Firefox	Ours	**0.87**	**0.65**	**0.90**
	SVMZ	0.74	0.61	0.89
	RFZ	0.78	0.56	0.89

We calculate the average AUC scores for each target project. On average, our approach achieves an AUC score of 0.74, 0.69, 0.75, 0.85 and 0.82 for Eclipse, Netbeans, Mozilla, Firefox and Thunderbird, respectively. Also, we compare the AUC scores of our approach shown in Tables 29–33 and the average AUC scores of our approach in the within-project setting shown in Table 8. Table 34 presents the ratio values between the AUC scores of our approach in this setting and the average AUC scores of our approach in the within-project setting for our five datasets. As shown in the table, in terms of AUC, our approach can achieve at least 88% of

TABLE 34
The ratio values between the AUC scores of our approach in the cross-project setting and the average AUC scores of our approach in the within-project setting.

Source Project \ Target Project	Eclipse	Netbeans	Mozilla	Firefox	Thunderbird
Eclipse	-	96%	89%	92%	94%
Netbeans	97%	-	88%	94%	94%
Mozilla	97%	92%	-	103%	101%
Firefox	96%	95%	94%	-	101%
Thunderbird	96%	96%	88%	100%	-

the performance that our approach achieves in the within-project setting.

From Table 34, we find that in most cases, AUC scores of our approach in the cross-project setting are slightly lower than AUC scores of our approach in within-project setting. In Section 4.3, we find that textual features of bug reports are very important for our approach when identifying *valid* bug reports. Textual contents of bug reports in different projects are likely to be different. Words frequently appearing in bug reports of one project may not frequently appear in bug reports of other projects. For example, we find that the word "launcher" appears in 5.8% of the bug reports in our Eclipse dataset while in the other datasets, only up to 0.2% of the bug reports contain this word. As a result, the textual features may have a bias in the cross-project setting, which causes the performance of our approach worse than the performance in within-project setting. Moreover, in Table 34, we also notice that when we use Mozilla as *target project*, the ratio values are lower as compared to other projects. This may due to that training data we use is not sufficient—compared with the Mozilla dataset, the other four datasets contain much less bug reports.

5.8 Threats to Validity

Threats to internal validity refer to errors in our datasets and implementation code. We have double-checked the code. However, there may still exist some errors which we did not find.

Developers can reopen bug reports and modify their status and resolution, which may introduce a threat to labels of our datasets. Status and resolution of a bug report may be changed many times in its history. In this study, we focused on the final status and resolution of bug reports until the time we collected our datasets (i.e., May 2017). And we analyzed the severity of the threat that is introduced by reopening bug reports on our datasets. We found that for at least 96% of the bug reports, their status and resolution remained unchanged until the time that we collected our datasets—c.f., Section 3.1 for details. Thus, developers are unlikely to reopen most of the bug reports of our datasets. We believe that labels of at least 96% of the bug reports in our datasets are reliable.

In this paper, we consider non-reproducible bug reports whose final resolution is *WORKSFORME* as *invalid*. In practice, developers may have to fix a non-reproducible bug. In this case, we cannot simply consider a non-reproducible bug report as *invalid*. However, since we focused on the final status and resolution of bug reports, the bug reports that were once considered as non-reproducible (with a resolution of *WORKSFORME*) but resolved later with a final resolution of *FIXED* or *WONTFIX* are actually considered as *valid* in our study. Hence, we have considered the *valid* non-reproducible bug reports. Nevertheless, the bug reports with a final resolution of *WORKSFORME* may still include *valid* ones that report real bugs, which may introduce a bias into our study.

Threats to external validity refer to the generality of our approach. In this paper, we analyzed five bug report datasets from Eclipse, Netbeans, Mozilla, Firefox and Thunderbird. The datasets contain a total of 560,697 bug reports. The datasets have different class distributions. And across the five datasets, our approach achieves better performance than the baselines. Thus, we believe that our approach can achieve good performance for datasets of different class distributions. Moreover, in Section 5.7, we presented that our approach can achieve better performance than the two baselines in the cross-project setting.

The five studied projects are GUI-based. Bug report for projects without GUIs may be different. Future studies should re-examine our finding on projects that are not GUI-based.

Threats to construct validity refer to the suitability of the evaluation measure. In this paper, we mainly used AUC and F1-scores for *valid* and *invalid* bug reports as our evaluation measure. The AUC and F1-scores are widely used evaluation measures in past software engineering studies [28], [47], [59], [61], [97]. Therefore, we believe that there exists little threat to construct validity of our study.

6 Related Work

In this section, we describe the related studies on valid bug determination, bug report quality and bug report management.

6.1 Study on Valid Bug Determination

To our best knowledge, Zanetti et al.'s study [99] is the most related work to our paper on *valid* bug determination. Zanetti et al. proposed the usage of collaboration network to determine *valid* bugs in open source projects. They constructed collaboration networks based on *ASSIGN* and *CC* relations of bug reports. Then, they extracted nine features for bug reporters from the networks, e.g., closeness centrality and clustering coefficient. After that, they built models based on their extracted features to determine the validity of a bug report. Our approach considers more features that are not considered by Zanetti et al., We not only extract features from collaboration

network, but also from reporter experience and bug report contents. We combine features which are grouped along five dimensions (i.e., reporter experience, collaboration network, completeness, readability and text), and our approach statistically significantly improves two baselines based on Zanetti et al.'s features (i.e., SVMZ and RFZ) in terms of AUC and F1-scores for *valid* and *invalid* bug reports.

6.2 Studies on Bug Report Quality

There have been number of studies on bug report quality [19], [36], [43], [50], [51], [66], [85], [101], [104].

Hooimeijer et al. identified high-quality bug reports as those with short resolution time and proposed quantitative and qualitative features extracted from bug reports to model bug report quality. [36]. They mainly focused on bug-fixing time prediction. Linstead et al. proposed an unsupervised technique to estimate bug report quality using Latent Dirichlet Allocation [50]. They identified bug reports with coherence topic as high-quality ones. In this paper, we identify *valid* bug reports according to their resolution field and we focus on determining the validity of a bug report, which is different from their studies.

Zimmermann et al. studied characteristics of good bug reports by surveying developers [104]. In the survey, they asked developers which characteristics of bug reports they most needed when they triaged and fixed bugs. They also proposed an automated technique called CUEZILLA which could analyze bug report quality and categorize bug reports into five levels (from very bad to very good). However, they trained CUEZILLA based on a very small dataset—only 289 bug reports. Unlike Zimmermann et al., we study the most important characteristics of *valid* bug reports by means of analyzing large-scale bug report datasets collected from five open source projects. Moreover, we not only consider extracting features from bug report contents, but also from submission history of reporters and their activities in the collaboration process of the project.

Recently, Chaparro et al. proposed an automated technique to help improve the quality of bug reports [19]. Their technique can detect missing information including observed behavior, expected behavior and steps to reproduce in description of the bug reports. By doing this, the technique can further alert the reporters about the detected missing information. They found 154 discourse patterns which are frequently used by reporters to describe the three types of information. Based on these patterns, they used regular expressions, heuristics and machine learning-based methods to automatically detect missing information of bug reports. In this paper, we detect six types of technical information in description of bug reports including stack traces, steps to reproduce, code samples, patches, test cases and screenshots. We use several regular expressions, words and phrases to detect the information. Our method is simple and straightforward. However, since reporters may not explicitly describe technical information in bug reports (e.g., steps to reproduce) [19], our method may lead to more false negatives as compared to Chaparro et al.'s method.

In addition to Chaparro et al.'s study, many studies are also conducted on improving bug report quality. Weimer et al. proposed an algorithm to automatically construct patches and they found that bug reports accompanied by patches were three times as likely to be fixed as standard bug reports [85]. Just et al. conducted a survey on information needs and commonly faced problems with bug reporting among several hundred developers [43]. Then, they suggested improvements to make bug tracking systems easier to use and facilitate submission of high-quality bug reports. Schroter et al. provided empirical evidence of the usefulness of stack traces in bug reports by statistically analyzing bug reports from Eclipse [66]. Lotufo et al. investigated whether Stack Overflow's game mechanisms are effective in increasing bug report quality [51]. They found that the game mechanisms are effective in such capabilities, and they suggested that bug tracking systems should be improved with game mechanisms. Zhang et al. found that many bug reports have limited information and proposed a sentence ranking algorithm to select proper contents for bug report enrichment [101]. Our study is different from these studies. These studies focus on improving bug report quality, while in our paper, we focus on analyzing the validity of a bug report and determining whether a bug report is *valid*.

6.3 Studies on Bug Report Management

There are other studies that have been proposed to help developers deal with a large number of bug reports. We highlight past studies on bug assignment, duplicated bug detection, severity prediction, and reopened bug prediction.

1) Bug Assignment: There have been a number of studies on bug assignment [4], [40], [57], [72], [77], [93]. Murphy et al. and Anvik et al. leveraged machine techniques such as Naive Bayes and SVM to recommend bug fixers [4], [57]. Tamrawi et al. proposed a fuzzy-based approach to recommend fixers [77]. Jeong et al. investigated bug reassignment phenomenon in Eclipse and Mozilla [40]. And they proposed the usage of tossing graph to recommend bug fixers. Shokripour et al. proposed a two-phase local-based approach which used simple term filtering and weighting to recommend developers [72]. Xia et al. extended Latent Dirichlet Allocation (LDA) and proposed a specialized topic modeling algorithm for bug assignment [93].

2) Duplicate Bug Detection: There have been a number of studies on duplicate bug report detection [58], [75], [76], [84]. Wang et al. identified duplicate bug reports using natural language and execution information present in bug reports [84]. Sun et al. proposed a discriminative model based on features for duplicate bug report detection [76]. The model was based on features they extracted from duplicate and non-duplicate bug reports. Then they use the likelihood scores output by the model to rank existing bug reports given a new report. In their other work, Sun et al. proposed a retrieval function called REP to measure the similarity between two bug reports [75]. They considered a number of special features of duplicate bug reports and extended BM25F—an effective retrieval function proposed in the information retrieval (IR) community. Nguyen et al. proposed a duplicate bug report detection approach taking advantage of both IR-based and topic based features [58].

3) Severity Prediction: There have been a number of studies on bug severity prediction [47], [55], [80]. Menzies

et al. proposed an automated method which could assign severity levels to bug reports using multi-class classification techniques [55]. Lamkanfi et al. extended Menzies et al.'s work by predicting the severity of bug reports into two severity labels, i.e., severe and not severe [47]. In a later work, Tian et al. proposed an approach to predict fine-grained severity labels of bug reports using information retrieval techniques and nearest neighbor classification [80].

4) Reopened Bug Prediction: There have been a number of studies on reopened bug prediction [70], [71], [95], [103]. Shihab et al. proposed the problem of identifying reopened bugs and they used machine learning techniques to predict reopened bugs in Eclipse, Apache and OpenOffice [70], [71]. Zimmermann et al. performed an empirical study on reopened bugs and analyzed their root causes in the Microsoft Windows operating system [103]. Xia et al. proposed a composite approach which combines various features to improve the performance of reopened bug prediction [95].

7 Conclusion

In this paper, we propose an approach to determine whether a newly submitted bug report is *valid* or *invalid*. We extract 33 features to characterize a bug report; these features are grouped along five dimensions: reporter experience, collaboration network, completeness, readability and text. Then, we use random forest to build models. To investigate the effectiveness of our approach, we conduct an experiment on large-scale bug report datasets from five open source projects containing a total of 560,697 bug reports. Our experimental results show that across the five datasets, our approach achieves an average F1-score for *valid* bug reports and F1-score for *invalid* ones of 0.74 and 0.67, respectively. In terms of F1-score for *valid* bug reports, our approach outperforms SVMZ and RFZ by 9% and 12%, respectively. And in terms of F1-score for *invalid* bug reports, our approach outperforms SVMZ and RFZ by 34% and 29%, respectively. Furthermore, our approach achieves an average AUC of 0.81, which outperforms SVMZ and RFZ by 33% and 19%, respectively. Our experimental results show that our approach statistically significantly outperforms the two baselines by a substantial margin. We also find that among the 33 features, *desc-dmnb-score*, *valid-rate* and *summary-dmnb-score* are the most important ones that distinguish *valid* bug reports from *invalid* ones.

In the future, we plan to evaluate our approach with more bug reports from more software projects. And we also plan to study more features that can impact validity of bug reports, and design a better approach to improve the performance further.

References

[1] H. Abdi. Bonferroni and šidák corrections for multiple comparisons. *Encyclopedia of measurement and statistics*, 3:103–107, 2007.

[2] A. Agrawal, W. Fu, and T. Menzies. What is wrong with topic modeling?(and how to fix it using search-based se). *arXiv preprint arXiv:1608.08176*, 2016.

[3] J. Anderson. Lix and rix: Variations on a little-known readability index. *Journal of Reading*, 26(6):490–496, 1983.

[4] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, pages 361–370. ACM, 2006.

[5] J. Aranda and G. Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *Proceedings of the 31st International Conference on Software Engineering*, pages 298–308. IEEE Computer Society, 2009.

[6] D. Bertram, A. Voida, S. Greenberg, and R. Walker. Communication, collaboration, and bugs: the social nature of issue tracking in small, collocated teams. In *Proceedings of the 2010 ACM conference on Computer supported cooperative work*, pages 291–300. ACM, 2010.

[7] N. Bettenburg and A. E. Hassan. Studying the impact of social structures on software quality. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pages 124–133. IEEE, 2010.

[8] N. Bettenburg and A. E. Hassan. Studying the impact of social interactions on software quality. *Empirical Software Engineering*, 18(2):375–431, 2013.

[9] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Extracting structural information from bug reports. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 27–30. ACM, 2008.

[10] P. Bhattacharya and I. Neamtiu. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[11] P. Bhattacharya and I. Neamtiu. Bug-fix time prediction models: can we do better? In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 207–210. ACM, 2011.

[12] P. Bonacich. Power and centrality: A family of measures. *American journal of sociology*, 92(5):1170–1182, 1987.

[13] A. P. Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern recognition*, 30(7):1145–1159, 1997.

[14] U. Brandes and C. Pich. Centrality estimation in large networks. *International Journal of Bifurcation and Chaos*, 17(07):2303–2318, 2007.

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

[16] S. Carmi, S. Havlin, S. Kirkpatrick, Y. Shavitt, and E. Shir. A model of internet topology using k-shell decomposition. *Proceedings of the National Academy of Sciences*, 104(27):11150–11154, 2007.

[17] W. B. Cavnar, J. M. Trenkle, et al. N-gram-based text categorization. *Ann Arbor MI*, 48113(2):161–175, 1994.

[18] C.-C. Chang and C.-J. Lin. Libsvm: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):27, 2011.

[19] O. Chaparro, J. Lu, F. Zampetti, L. Moreno, M. Di Penta, A. Marcus, G. Bavota, and V. Ng. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 396–407. ACM, 2017.

[20] J. M. Chaves-González and M. A. Pérez-Toledano. Differential evolution with pareto tournament for the multi-objective next release problem. *Applied Mathematics and Computation*, 252:1–13, 2015.

[21] N. Cliff. *Ordinal methods for behavioral data analysis*. Psychology Press, 2014.

[22] M. Coleman and T. L. Liau. A computer readability formula designed for machine scoring. *Journal of Applied Psychology*, 60(2):283, 1975.

[23] D. A. da Costa, S. L. Abebe, S. McIntosh, U. Kulesza, and A. E. Hassan. An empirical study of delays in the integration of addressed issues. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 281–290. IEEE, 2014.

[24] K. Ehrlich and M. Cataldo. All-for-one and one-for-all?: a multi-level analysis of communication patterns and individual performance in geographically distributed software development. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pages 945–954. ACM, 2012.

[25] R. Flesch. A new readability yardstick. *Journal of applied psychology*, 32(3):221, 1948.

[26] R. F. Flesch. *How to write plain English: A book for lawyers and consumers*. Harpercollins, 1979.

[27] L. C. Freeman. Centrality in social networks conceptual clarification. *Social networks*, 1(3):215–239, 1978.

[28] W. Fu and T. Menzies. Easy over hard: A case study on deep learning. *arXiv preprint arXiv:1703.00133*, 2017.

[29] W. Fu, V. Nair, and T. Menzies. Why is differential evolution better than grid search for tuning defect predictors? *arXiv preprint arXiv:1609.02613*, 2016.

[30] R. Gunning. The technique of clear writing. 1952.

[31] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 495–504. IEEE, 2010.

[32] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003.

[33] A. Hagberg, P. Swart, and D. S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Laboratory (LANL), 2008.

[34] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[35] J. Han, J. Pei, and M. Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.

[36] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 34–43. ACM, 2007.

[37] D. Hosmer, S. Lemeshow, and R. X. Sturdivant. *Applied logistic regression*, volume 398. John Wiley & Sons, 2013.

[38] J. Huang and C. X. Ling. Using auc and accuracy in evaluating learning algorithms. *IEEE Transactions on knowledge and Data Engineering*, 17(3):299–310, 2005.

[39] Q. Huang, E. Shihab, X. Xia, D. Lo, and S. Li. Identifying self-admitted technical debt in open source projects using text mining. *Empirical Software Engineering*, pages 1–34, 2017.

[40] G. Jeong, S. Kim, and T. Zimmermann. Improving bug triage with bug tossing graphs. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 111–120. ACM, 2009.

[41] T. Jiang, L. Tan, and S. Kim. Personalized defect prediction. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 279–289. IEEE Press, 2013.

[42] M. Joblin, S. Apel, and W. Mauerer. Evolutionary trends of developer coordination: A network approach. *Empirical Software Engineering*, 22(4):2050–2094, 2017.

[43] S. Just, R. Premraj, and T. Zimmermann. Towards the next generation of bug tracking systems. In *Visual languages and Human-Centric computing, 2008. VL/HCC 2008. IEEE symposium on*, pages 82–85. IEEE, 2008.

[44] S. Kim, E. J. Whitehead Jr, and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.

[45] J. P. Kincaid, R. P. Fishburne Jr, R. L. Rogers, and B. S. Chissom. Derivation of new readability formulas (automated readability index, fog count and flesch reading ease formula) for navy enlisted personnel. Technical report, Naval Technical Training Command Millington TN Research Branch, 1975.

[46] O. Kononenko, O. Baysal, and M. W. Godfrey. Code review quality: how developers see it. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 1028–1038. IEEE, 2016.

[47] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals. Predicting the severity of a reported bug. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 1–10. IEEE, 2010.

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

[49] H. Li, W. Shang, Y. Zou, and A. E. Hassan. Towards just-in-time suggestions for log changes. *Empirical Software Engineering*, pages 1–35, 2016.

[50] E. Linstead and P. Baldi. Mining the coherence of gnome bug reports with statistical topic models. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pages 99–102. IEEE, 2009.

[51] R. Lotufo, L. Passos, and K. Czarnecki. Towards improving bug tracking systems with game mechanisms. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 2–11. IEEE Press, 2012.

[52] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[53] G. H. Mc Laughlin. Smog grading-a new readability formula. *Journal of reading*, 12(8):639–646, 1969.

[54] A. McCallum, K. Nigam, et al. A comparison of event models for naive bayes text classification. In *AAAI-98 workshop on learning for text categorization*, volume 752, pages 41–48. Madison, WI, 1998.

[55] T. Menzies and A. Marcus. Automated severity assessment of software defect reports. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 346–355. IEEE, 2008.

[56] A. Mockus, M. Nagappan, and A. Hassan. Best practices and pitfalls for statistical analysis of se data. In *Proc. ICSE*, 2014.

[57] G. Murphy and D. Cubranic. Automatic bug triage using text categorization. In *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*, 2004.

[58] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 70–79. ACM, 2012.

[59] M. H. Osman, M. R. Chaudron, and P. Van Der Putten. An analysis of machine learning algorithms for condensing reverse engineered class diagrams. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 140–149. IEEE, 2013.

[60] J. Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. 1998.

[61] G. K. Rajbahadur, S. Wang, Y. Kamei, and A. E. Hassan. The impact of using regression models to build defect classifiers. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 135–145. IEEE Press, 2017.

[62] C. R. Reis and R. P. de Mattos Fortes. An overview of the software engineering process and tools in the mozilla project. In *Proceedings of the Open Source Software Development Workshop*, pages 155–175, 2002.

[63] J. D. Rennie, L. Shih, J. Teevan, and D. R. Karger. Tackling the poor assumptions of naive bayes text classifiers. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 616–623, 2003.

[64] J. Saramäki, M. Kivelä, J.-P. Onnela, K. Kaski, and J. Kertesz. Generalizations of the clustering coefficient to weighted complex networks. *Physical Review E*, 75(2):027105, 2007.

[65] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, 40(10):993–1006, 2014.

[66] A. Schroter, A. Schröter, N. Bettenburg, and R. Premraj. Do stack traces help developers fix bugs? In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 118–121. IEEE, 2010.

[67] A. J. Scott and M. Knott. A cluster analysis method for grouping means in the analysis of variance. *Biometrics*, pages 507–512, 1974.

[68] F. Sebastiani. Machine learning in automated text categorization. *ACM computing surveys (CSUR)*, 34(1):1–47, 2002.

[69] R. Senter and E. A. Smith. Automated readability index. Technical report, CINCINNATI UNIV OH, 1967.

[70] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto. Predicting re-opened bugs: A case study on the eclipse project. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 249–258. IEEE, 2010.

[71] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto. Studying re-opened bugs in open source software. *Empirical Software Engineering*, 18(5):1005–1042, 2013.

[72] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani. Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 2–11. IEEE Press, 2013.

[73] R. Storn and K. Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997.

[74] J. Su, H. Zhang, C. X. Ling, and S. Matwin. Discriminative parameter learning for bayesian networks. In *Proceedings of the 25th international conference on Machine learning*, pages 1016–1023. ACM, 2008.

[75] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang. Towards more accurate retrieval of duplicate bug reports. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 253–262. IEEE, 2011.

[76] C. Sun, D. Lo, X. Wang, J. Jiang, and S. Khoo. A discriminative model approach for accurate duplicate bug report retrieval. 1:45–54, 2010.

[77] A. Tamrawi, T. T. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Fuzzy set-based automatic bug triaging: Nier track. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 884–887. IEEE, 2011.

[78] C. Tantithamthavorn and A. E. Hassan. An experience report on defect modelling in practice: Pitfalls and challenges. *forest*, 15(17):71–73, 2018.

[79] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. An empirical comparison of model validation techniques for defect prediction models. (1), 2017.

[80] Y. Tian, D. Lo, and C. Sun. Information retrieval based nearest neighbor classification for fine-grained bug severity prediction. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 215–224. IEEE, 2012.

[81] Y. Tian, D. Lo, X. Xia, and C. Sun. Automated prediction of bug report priority using multi-factor analysis. *Empirical Software Engineering*, 20(5):1354–1383, 2015.

[82] Y. Tian, M. Nagappan, D. Lo, and A. E. Hassan. What are the characteristics of high-rated apps? a case study on free android applications. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 301–310. IEEE, 2015.

[83] H. Valdivia Garcia and E. Shihab. Characterizing and predicting blocking bugs in open source projects. In *Proceedings of the 11th working conference on mining software repositories*, pages 72–81. ACM, 2014.

[84] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 461–470. IEEE, 2008.

[85] W. Weimer. Patches as better bug reports. In *Proceedings of the 5th international conference on Generative programming and component engineering*, pages 181–190. ACM, 2006.

[86] G. Widmer and M. Kubat. Learning in the presence of concept drift and hidden contexts. *Machine learning*, 23(1):69–101, 1996.

[87] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics bulletin*, 1(6):80–83, 1945.

[88] D. F. Williamson, R. A. Parker, and J. S. Kendrick. The box plot: a simple visual method to interpret data. *Annals of internal medicine*, 110(11):916–921, 1989.

[89] D. H. Wolpert and W. G. Macready. An efficient method to estimate bagging's generalization error. *Machine Learning*, 35(1):41–55, 1999.

[90] X. Wu, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, S. Y. Philip, et al. Top 10 algorithms in data mining. *Knowledge and information systems*, 14(1):1–37, 2008.

[91] X. Xia, L. Bao, D. Lo, P. S. Kochhar, A. E. Hassan, and Z. Xing. What do developers search for on the web? *Empirical Software Engineering*, pages 1–37, 2017.

[92] X. Xia and D. Lo. Feature generation and engineering for software analytics. *Feature Engineering for Machine Learning and Data Analytics*, page 335, 2018.

[93] X. Xia, D. Lo, Y. Ding, J. M. Al-Kofahi, T. N. Nguyen, and X. Wang. Improving automated bug triaging with specialized topic model. *IEEE Transactions on Software Engineering*, 43(3):272–297, 2017.

[94] X. Xia, D. Lo, E. Shihab, X. Wang, and X. Yang. Elblocker: Predicting blocking bugs with ensemble imbalance learning. *Information and Software Technology*, 61:93–106, 2015.

[95] X. Xia, D. Lo, E. Shihab, X. Wang, and B. Zhou. Automatic, high accuracy prediction of reopened bugs. *Automated Software Engineering*, 22(1):75–109, 2015.

[96] X. Xia, D. Lo, X. Wang, and B. Zhou. Accurate developer recommendation for bug resolution. In *Reverse engineering (WCRE), 2013 20th working conference on*, pages 72–81. IEEE, 2013.

[97] X. Xia, E. Shihab, Y. Kamei, D. Lo, and X. Wang. Predicting crashing releases of mobile applications. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 29. ACM, 2016.

[98] Y. Yang and J. O. Pedersen. A comparative study on feature selection in text categorization. In *Icml*, volume 97, pages 412–420, 1997.

[99] M. S. Zanetti, I. Scholtes, C. J. Tessone, and F. Schweitzer. Categorizing bugs with social networks: a case study on four open source software communities. In *Software Engineering (ICSE), 2013 35th international conference on*, pages 1032–1041. IEEE, 2013.

[100] H. Zhang, L. Gong, and S. Versteeg. Predicting bug-fixing time: an empirical study of commercial software projects. In *Proceedings of the 2013 international conference on software engineering*, pages 1042–1051. IEEE Press, 2013.

[101] T. Zhang, J. Chen, H. Jiang, X. Luo, and X. Xia. Bug report enrichment with application of automated fixer recommendation. In *Proceedings of the 25th International Conference on Program Comprehension*, pages 230–240. IEEE Press, 2017.

[102] Y. Zhou, Y. Tong, R. Gu, and H. C. Gall. Combining text mining and data mining for bug report classification. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 311–320. IEEE, 2014.

[103] T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy. Characterizing and predicting which bugs get reopened. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1074–1083. IEEE Press, 2012.

[104] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss. What makes a good bug report? *IEEE Transactions on Software Engineering*, 36(5):618–643, 2010.