Noname manuscript No.
(will be inserted by the editor)

Works for Me! Cannot Reproduce – A Large Scale Empirical Study of Non-reproducible Bugs

Mohammad M. Rahman · Foutse Khomh · Marco Castelluccio

the date of receipt and acceptance should be inserted later

Abstract Software developers attempt to reproduce software bugs to understand their erroneous behaviours and to fix them. Unfortunately, they often fail to reproduce (or fix) them, which leads to faulty, unreliable software systems. However, to date, only a little research has been done to better understand what makes the software bugs non-reproducible. In this article, we conduct a multimodal study to better understand the non-reproducibility of software bugs. First, we perform an empirical study using 576 non-reproducible bug reports from two popular software systems (Firefox, Eclipse) and identify 11 key factors that might lead a reported bug to non-reproducibility. Second, we conduct a user study involving 13 professional developers where we investigate how the developers cope with non-reproducible bugs. We found that they either close these bugs or solicit for further information, which involves long deliberations and counter-productive manual searches. Third, we offer several actionable insights on how to avoid non-reproducibility (e.g., false-positive bug report detector) and improve reproducibility of the reported bugs (e.g., sandbox for bug reproduction) by combining our analyses from multiple studies (e.g., empirical study, developer study). Fourth, we explain the differences between reproducible and non-reproducible bug reports by systematically interpreting multiple machine learning models that classify these reports with high accuracy. We found that links to existing bug reports might help improve the reproducibility of a reported bug. Finally, we detect the connected bug reports to a non-reproducible bug automatically and further demonstrate how 93 bugs connected to 71 non-reproducible bugs from our dataset can offer complementary information (e.g., attachments, screenshots, program flows).

Mohammad M. Rahman
Dalhousie University, Canada
E-mail: masud.rahman@dal.ca

Foutse Khomh
Polytechnique Montreal, Canada
E-mail: foutse.khomh@polymtl.ca

Marco Castelluccio
Mozilla Corporation
E-mail: mcastelluccio@mozilla.com

Keywords Bug reproduction, non-reproducibility, empirical study, grounded theory, developer feedback, reproducibility challenges, key factors, bug report classification, model interpretation.

1 Introduction

Software bugs and failures claim trillions of dollars every year. In 2017 alone, 606 software bugs cost the global economy about \$1.7 trillion with 3.7 billion people affected and 314 companies impacted [5]. Finding such bugs within software code and then fixing them are highly challenging. One of the major challenges is to determine the root cause of a reported bug, which might help gain a better understanding of the bug [44]. Developers often attempt to reproduce a bug from its report to determine its root cause and to explore its erroneous behaviours. Unfortunately, the bug reports often do not contain useful information for reproducing the bugs [21, 22, 16]. This leads to (a) an unexpected delay (e.g., three months [38]) in bug fixing [32, 72] or (b) even worse, the release of a software system without fixing potentially critical bugs, which could be costly in the long run (e.g., Facebook's privacy vulnerability [4]). Thus, (a) studying the key factors (a.k.a., characteristics) that could make the reported bugs non-reproducible and (b) detecting these non-reproducible bugs early in their management cycle – are open research problems that warrant further investigations. Our work in this paper addresses the first research problem.

Existing work from the literature (a) study the characteristics of a good bug report [16] and classify the software bugs from open source systems [56, 42], (b) predict which bugs get fixed [31, 30], re-assigned [32, 66] or re-opened [72] and (c) investigate how bugs are coordinated among various stakeholders (e.g., software testers, users, developers) [15, 60] and how the misclassification of bugs affects the bug prediction task [33]. Unfortunately, little research has been done to better understand what makes the reported bugs non-reproducible. Vyas et al. [60] first analyse the social and human aspects of a bug reproduction process with an ethnographic study. Their study explains the human collaboration aspect, which might not be enough to explain the complex technical aspect of a bug reproduction process. Joorabchi et al. [38] identify several factors (e.g., Inter-bug dependencies, environmental differences) that might contribute to the non-reproducibility of a reported bug. Although their study sheds some light on the factors leading to bug non-reproducibility, it does not go as far as understanding the mitigation strategies currently implemented in the field. Neither does it examine the mechanisms to improve the reproducibility of reported bugs. Our work attempts to fill this gap in the literature.

In this article, we conduct (a) an empirical study to better understand the key factors behind non-reproducibility of software bugs and (b) a developer study to understand how the professional developers cope with non-reproducible bugs and how to improve the bug reports. First, we analyze 576 randomly sampled bug reports marked as *non-reproducible* (250 from Mozilla Firefox + 326 from Eclipse JDT) with a *Grounded Theory* method [29] and identify 11 key factors that might lead a reported bug to non-reproducibility. We also contrast our findings with those from Joorabchi et al. [38]. Second, we validate our empirical and analytical findings with a user study involving 13 professional developers (from

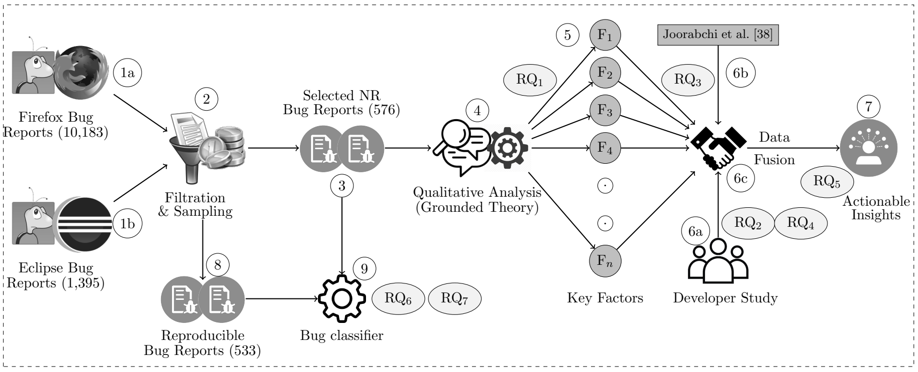


Fig. 1: Schematic diagram of our conducted study

`Mozilla` and `Freelancer`) and gain meaningful insights on how the developers deal with non-reproducible bugs, which were missing in the earlier work [38]. Third, by cross-referencing findings from our qualitative analysis and the responses from our developer study, we report several actionable insights for detecting and then improving the non-reproducible bugs during their submission. Fourth, we show the differences between reproducible and non-reproducible bug reports by interpreting three machine learning models (using the SHAP framework [40]) that classify these reports with high accuracy. Finally, we design a novel technique using Information Retrieval to automatically detect the connected bug reports to a non-reproducible bug. We further manually analyze 93 bug reports connected to 71 non-reproducible bugs from our dataset and demonstrate how these connected reports can deliver useful artifacts (e.g., attachments, screenshots, stack traces, program workflows) to complement the non-reproducible bugs.

Novelty in contribution: This article is a significantly extended version of our earlier work [49]. The earlier work (a) identifies 11 key factors analyzing 576 non-reproducible bug reports to explain the non-reproducibility of a bug report, (b) validates these factors using the feedback from 13 professional developers, and then (c) suggests how to prevent the non-reproducibility of bugs by combining findings from multiple sources (e.g., empirical study, developer survey). In this article, we (d) analyze an additional total of 533 reproducible bug reports, and (e) investigate the differences between reproducible and non-reproducible bug reports by systematically interpreting several machine learning models that classify these reports with high performance (e.g., 83.60% precision, 83.60% recall). We found that links to the past bug reports might help improve the reproducibility of a reported bug. We thus further (f) demonstrate how the past bug reports connected to a non-reproducible bug report can be detected automatically and could also offer useful information (e.g., attachments, screenshots, stack traces). Finally, we also (g) repeat our experiments using an extended collection of 1,990 bug reports (1,009 non-reproducible + 981 reproducible) and report our findings, which increase the confidence and generalizability of our results.

Thus, we answer seven important research questions in this work as follows.

(a) **RQ$_1$: What are the key factors that make a reported software bug non-reproducible?**

To mitigate the issue of bug non-reproducibility, understanding the key causes (or factors) is essential. We find 11 key factors (e.g., bug duplication, intermittency, missing information, ambiguous specifications, third-party defects) that might lead a reported software bug to non-reproducibility.

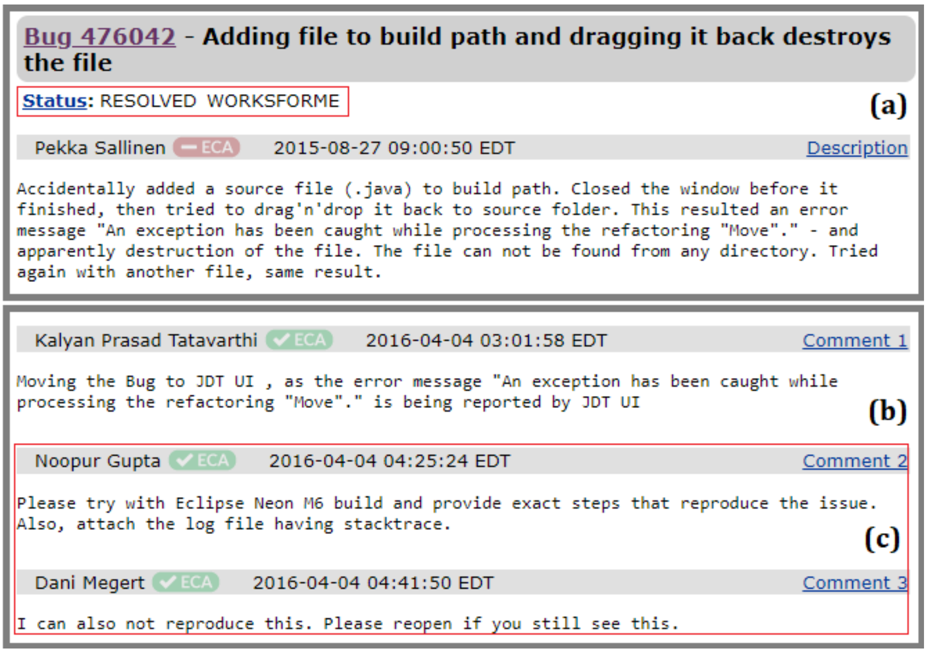
(b) **RQ$_2$: What do professional developers consider to be key factors behind the non-reproducibility of bugs?**
Developers' feedback on empirical findings is important to increase confidence in the findings. Our identified key factors were validated by 13 professional developers with an agreement level between 70% and 90%.

(c) **RQ$_3$: Do the identified factors match with the relevant earlier findings from the literature**?
Generalizability of findings across multiple studies and multiple datasets is important to increase confidence in the findings. Five of our identified factors match with the earlier findings derived from a different dataset [38]. Our study also reveals several novel factors. Thus, our findings not only confirm but also improve the existing understanding of non-reproducible software bugs.

(d) **RQ$_4$: How do the professional developers deal with non-reproducible software bugs?**
Understanding the current practices for dealing with non-reproducible bugs is important to provide efficient solutions. We find that professional developers deliberate over non-reproducible bugs and attempt to collect more information when the bug reports are incomplete or the reported bugs are complex (e.g., intermittent bugs, performance bugs). They also close the duplicate and false-positive bug reports with suitable explanations. Many of these tasks are performed in a counter-productive way due to the lack of appropriate alternatives (e.g., tool supports).

(e) **RQ$_5$: How to prevent the non-reproducibility and/or improve the reproducibility of reported bugs?**
Non-reproducibility of software bugs leads to delays in bug-fixing and potentially buggy software releases, which could be mitigated with appropriate tool supports. For example, intelligent tools (1) for detecting duplicate or false-positive bug reports and (2) for improving the software documentations could help avoid non-reproducible bugs. On the other hand, intelligent tools for complementing the incomplete bug reports could improve their reproducibility. Furthermore, sandbox tools where the developers can repeat their experiments could be useful for reproducing the complex software bugs (e.g., performance bugs, intermittent bugs).

(f) **RQ$_6$: Can we leverage machine learning to characterize non-reproducible software bugs?**
Explaining the differences between non-reproducible and reproducible bug reports using machine learning models and their interpretation frameworks (e.g., SHAP) could provide additional insights. We find that the bug reports submitted by actively contributing members in a project are more likely to be reproducible than those from the others. The presence of attachments and connected bugs in a bug report also improves its chance of being reproducible. Thus, effective tools (1) to find the project members with relevant experience and (2) to detect the linked bugs accurately during the submission of a bug report could help improve its chance of being reproducible.


Bug 476042 - Adding file to build path and dragging it back destroys the file

Status: RESOLVED WORKSFORME

(a)

Pekka Sallinen ECA 2015-08-27 09:00:50 EDT Description

```
Accidentally added a source file (.java) to build path. Closed the window before it
finished, then tried to drag'n'drop it back to source folder. This resulted an error
message "An exception has been caught while processing the refactoring "Move"." - and
apparently destruction of the file. The file can not be found from any directory. Tried
again with another file, same result.
```

Kalyan Prasad Tatavarthi ECA 2016-04-04 03:01:58 EDT Comment 1

```
Moving the Bug to JDT UI , as the error message "An exception has been caught while
processing the refactoring "Move"." is being reported by JDT UI
```

(b)

Noopur Gupta ECA 2016-04-04 04:25:24 EDT Comment 2

```
Please try with Eclipse Neon M6 build and provide exact steps that reproduce the issue.
Also, attach the log file having stacktrace.
```

(c)

Dani Megert ECA 2016-04-04 04:41:50 EDT Comment 3

```
I can also not reproduce this. Please reopen if you still see this.
```

Fig. 2: An example bug reporting thread – (a) title + description, (b) discussions among stakeholders, and (c) non-reproducibility of the bug

(g) **RQ$_7$: Can we automatically detect bug reports connected to a non-reproducible bug and leverage them to support its reproducibility?** About 20% of our non-reproducible bug reports were found to be connected to prior bugs in the bug tracking system. Such connections were manually established by the developers over a long period to extend their understanding of bugs. We design a technique to automatically detect these connected reports from history and to establish such connections during the submission of a non-reproducible bug report. Our designed technique was also able to detect the connected bug reports with 31%–86% accuracy when top 10 results were considered. Missing information is a key factor behind the non-reproducibility of many bug reports. According to our analysis, these connected bug reports contain a wealth of information such as fix patches, attachments, screenshots, test cases, stack traces, and program workflows, which have high potential to complement the non-reproducible bug reports.

Structure of the article: The rest of the article is organized as follows: Section 2 provides an overview of non-reproducible bug reports and Section 3 discusses our study methodology including dataset construction, the design of empirical study and developer study, and the selection of suitable frameworks to interpret the machine learning models. Section 4 presents our study results, Section 5 focuses on the threats to validity, Section 6 discusses the related work and finally, Section 7 concludes our article with future work.

Table 1: Study Dataset

System	Domain	BTS	Comp.	Type	Duration	All	SBR
Initial dataset (1,109 bug reports)							
Firefox	Web browser	Bugzilla	Core	NR	12/29/2017–12/29/2019	1,274	250
Eclipse	IDE	Bugzilla	JDT	NR	12/29/2014–12/29/2019	326	326
Firefox	Web browser	Bugzilla	Core	R	12/29/2017–12/29/2019	8,909	250
Eclipse	IDE	Bugzilla	JDT	R	12/29/2014–12/29/2019	1,069	283
Total	-	-	-	-	-		**1,109**
Extended dataset (1,990 bug reports)							
Firefox	Web browser	Bugzilla	Core	NR	12/29/2017–12/29/2019	1,274	683
Eclipse	IDE	Bugzilla	JDT	NR	12/29/2014–12/29/2019	326	326
Firefox	Web browser	Bugzilla	Core	R	12/29/2017–12/29/2019	8,909	698
Eclipse	IDE	Bugzilla	JDT	R	12/29/2014–12/29/2019	1,069	283
Total	-	-	-	-	-		**1,990**

BTS=Bug Tracking System, **Comp.**=Component, **NR**=Non-reproducible, **R**=Reproducible, **SBR**=Sampled Bug Reports.

2 Non-Reproducible Bugs

Software developers often attempt to reproduce their bugs to better understand them. *Non-reproducible* bugs are the ones that cannot be reproduced by using the information found in their corresponding bug reports. They are annotated with several labels such as *"Works on My Machine"*, *"Works For Me"* and *"Cannot Reproduce"* in the bug-tracking systems [38]. About 17% of the reported bugs could be non-reproducible [38], which is a significant amount. Based on a developer study, Bettenburg et al. [16] suggest that developers expect at least three components within a bug report – Observed Behaviour (OB), Expected Behaviour (EB) and Steps to Reproduce (STR). OB describes the erroneous behaviour of a software system whereas EB outlines the correct behaviour of the system. On the other hand, STR provides the steps to reproduce a bug. Bug reports that miss these components could be difficult to reproduce. According to Chaparro et al. [21], about 65% of the bug reports miss the correct behaviour (EB) and 49% of the reports do not contain any steps to reproduce the bugs. Fig. 2 shows an example bug that erroneously deletes documents from the build path of an Eclipse project. From the discussions, we also see that the bug report fails to provide any concrete steps or complementary information (e.g., logs, stack traces) for reproducing this bug (Fig. 2-(c)). As a result, the bug was marked as *non-reproducible* (a.k.a., WORKSFORME) by the developers and was abandoned without fixing.

3 Study Methodology

Fig. 1 shows the schematic diagram of our performed studies in this paper. We first perform an empirical study on bug non-reproducibility using 576 non-reproducible bugs from two popular software systems. We not only revisit the earlier findings [38] but also deliver novel insights towards the better understanding of non-reproducible bugs. Then we validate our major findings with a developer study and formulate further actionable insights. We also explain the reproducibility of bugs by interpreting the machine learning models that classify them and then further complement our analysis with manual investigation. In this section, we discuss the major steps of our study design as follows.

3.1 Selection of Bug Reports

Non-reproducible bug reports. We use a total of 576 non-reproducible bugs from two popular, mature, open source systems – Mozilla Firefox and Eclipse – for our study. Table 1 shows an overview of our study dataset. Several steps were taken to carefully select these reported bugs. First, we collect all the bug reports from these systems that were marked as WORKSFORME. Both systems use this tag to mark their *non-reproducible* bugs [3]. Then we select the ones that were submitted within the last two years from Firefox and within the last five years from Eclipse. We use the recent bugs that the developers could still remember and that can provide timely, relevant, actionable insights. We also choose the bug reports concerning two major components – *Firefox Core* and *Eclipse JDT* – due to their critical roles. This step provides a total of 1,600 non-reproducible bug reports (1,274 from Firefox + 326 from Eclipse) (Step 1, Fig. 1). Second, since manually analysing hundreds of bug reports would be too expensive, we randomly choose 576 bug reports from them. From Mozilla Firefox, we attempted to choose a sample with a confidence level of 95% and an error margin of 5%, which indicates a total of 250 bug reports. We thus identify top 25 critical subcomponents of Firefox Core (e.g., *WebRender*, *Playback*, *JavaScript Engine*) based on their bug report frequency and choose 10 random bug reports from each of these subcomponents, which provides a total of 250. On the other hand, we choose all 326 bug reports targeting Eclipse JDT from the last five years. Finally, we ended up with 576 (250+326) non-reproducible bugs from two popular software systems (Steps 2-3, Fig. 1).

Reproducible bug reports. We also use a total of 533 (250+283) reproducible bugs from *Firefox Core* and *Eclipse JDT* for our study. These bug reports were marked as FIXED. First, we collect 8,909 fixed (and hence reproducible) bug reports within the same timeline as above from Firefox Core and separate the reports targeting 25 critical components (e.g., *WebRender*, *Playback*). Then we randomly select 10 bug reports from each component, which provides a total of 250 reproducible bug reports. Second, we collect 1,069 reproducible bug reports from Eclipse JDT that were submitted during the last five years. Then we randomly sample 283 bug reports from them, which leads to 95% confidence level and 5% margin of error. Thus, we collect a total of 1,109 (576 non-reproducible+533 reproducible) bug reports from *Firefox Core* and *Eclipse JDT*, which are used to answer RQ_6.

Bug Reports connected to non-reproducible bugs. About 27%–31% of our 576 non-reproducible bugs are duplicates of earlier bugs. That means, they were already solved and thus were not reproducible. However, we also found 350 bugs (187 from Eclipse + 163 from Firefox) that cannot be reproduced due to other factors (e.g., missing information, ambiguous specifications). We further analyze these 350 bugs and select the ones that are connected to earlier bugs from history. In particular, we analyze the metadata of each bug from Bugzilla, check two of their fields – `blocked` and `dependson`, and then select the bugs that have one or more connected bugs. We use a popular Java library namely *Jsoup*[1] to access the metadata. This step leads us to a collection of 71 non-reproducible bugs (12 from Eclipse + 59 from Firefox) that are connected to 93 existing bugs from the bug tracking system. We use these reports to answer our research question RQ_7.

[1] https://jsoup.org/

Extended dataset. We wanted to gain more confidence in our findings by increasing the size of our dataset. Thus, we extend our initial dataset by adding 881 new bug reports from *Firefox Core* system (433 non-reproducible + 448 reproducible). Over the last two years, 1,274 non-reproducible bug reports were submitted targeting this system (Table 1). We extend our initial sample of 250 with 433 non-reproducible bug reports from the 25 critical components of *Firefox Core* (mentioned above). This leads to a sample of 683 non-reproducible bug reports, which has 95% confidence level with 2.5% error margin against the population of 1,274 reports. Similarly, we also randomly select 448 reproducible bug reports from the same 25 components and construct a sample of 698 reproducible bug reports. Such a sample has 95% confidence level with ≈3.5% error margin against the population of 8,909 reproducible bug reports. We did not extend our dataset using *Eclipse JDT* since all of its non-reproducible bug reports from the last five years were already included in the initial dataset. Finally, we thus ended up with an extended dataset of 1,990 bug reports (1,009 non-reproducible and 981 reproducible bug) from two popular software systems for our experiments.

We also analyzed 1,009 non-reproducible bug reports and found that 274 (27%) of them were connected to one or more existing bugs in the bug-tracking system (e.g., Bugzilla). These connections were made by human developers over a long period. We select these reports and their connected bug reports (30 from Eclipse + 222 from Firefox), and construct a corpus of 526 bug reports where 252 connected reports serve as the *ground truth* against 274 non-reproducible bugs during automated detection of their connected bug reports. Then we use these corpus and ground truth to evaluate our technique for automatically detecting the connected bug reports in RQ$_7$ (Section 4.7).

3.2 Identifying Key Factors with Qualitative Analysis

We carefully analyse the information available in the bug reports and attempt to understand the key factors behind the non-reproducibility of their discussed bugs (Steps 4–5, Fig. 1). We first establish the scope of our analysis and then employ a widely used qualitative method – Grounded Theory [29] – for our qualitative analysis as follows.

Determining the Scope of Manual Analysis: In modern bug-tracking systems (e.g., Bugzilla), each bug report captures (1) bug description from a reporter and (2) discussions among various stakeholders (e.g., reporter, developers, testers). Once an encountered bug is reported, the stakeholders engage in a discussion where they attempt to reproduce the bug using the available information at hand. Since we attempt to understand the key issues behind bug non-reproducibility, we analyse both the bug description and the discussion texts from each bug report. While textual contents are mostly prevalent, bug reports might also contain supplementary materials (e.g., stack traces, logs, memory dump, test cases, configuration files) to assist in bug fixing. In this study, we systematically analyse the textual contents and occasionally check the supplementary materials to derive our insights.

Grounded Theory: We analyse bug reports using the Grounded Theory method [29] to determine the key factors behind bug non-reproducibility. Grounded Theory has been widely adopted in the social science researches to derive theories that are firmly grounded in the data under analysis (e.g., interview scripts, questionnaires).

Recently, this method has also found applications in the Software Engineering researches [53, 20]. We systematically analyse the bug description and discussion texts, and look for potential clues (e.g., missing information, technical difficulty) that might explain the non-reproducibility of a reported bug. Grounded theory method involves three stages of coding as follows.

(a) Open Coding consists in breaking the gathered data (e.g., bug reports) into identifiable, interesting chunks. We go through the bug description and discussion texts of each bug report and look for potential clues that might explain the non-reproducibility of a bug. We record our identified clues using a set of key phrases [1]. The core idea was to keep an open mind and to choose as many codes as needed to carefully represent each bug report. In our open coding, 574 unique codes were produced from 576 bug reports. We spent $\approx$100 man-hours in the open coding of 576 bug reports and their discussions.

(b) Axial Coding focuses on finding connections among the open codes. In this stage, we place our open codes into a spreadsheet document and annotate the similar or connected codes with the same colours. We consider not only lexical overlap but also semantic relatedness in establishing the connection. Our goal was to divide the open codes into low-level categories. This step provides a set of 33 tentative categories from our 574 open codes above.

(c) Selective Coding determines the core variables (or categories) and constructs a theory to explain the phenomenon under study, i.e., the non-reproducibility of the reported bugs. While the axial coding provides low-level categories, we carefully merge them into higher level categories based on their common themes and semantic relatedness. This step provides a total of 11 key factors that might explain the non-reproducibility of reported software bugs (Step 5, Fig. 1). Each of these key factors is represented using a set of semantically connected key phrases.

Determining the Prevalence of Key Factors in Bug Reports: During open coding, we represent each bug report using a set of suitable key phrases that explain why the bug could not be reproduced. Similarly, each of our identified factors is represented using a set of semantically connected key phrases. To analyse the prevalence of key factors in our dataset, we determine the presence of one or more key factors in each bug report using their overlapping key phrases.

3.3 Complementing Empirical Study with Developer Study

Although our empirical findings are derived from developer discussions, we further validate and complement these findings with a developer study involving 13 professional developers. We also investigate how these developers cope with non-reproducible bugs in practice and how the research community could help them. We discuss our study setup including questionnaire preparation and participant selection as follows.

Questionnaire Preparation: We first provide a summary of our empirical findings on bug non-reproducibility. Then we ask five different questions on four topics. First, we ask the participants whether they agree or disagree with our identified causes of bug non-reproducibility (Table 2) using dichotomous questions. Second, we ask how they deal with non-reproducible bugs as a part of their job. Third, we ask how the research community could help the developers in dealing with the non-reproducible bugs. Fourth, we ask the developers about their professional experience level, which was used for the demographic analysis.

Participant Selection: We first conduct a pilot study with one professional developer from Mozilla Firefox, which helped us improve our questionnaire. Then we invite the professional developers from Mozilla, Freelancer and Stack Overflow who have relevant bug-fixing experience to answer our questionnaire. We send our invitations to the developers using direct correspondences, organization's mailing lists (e.g., Mozilla Firefox) and public forums (e.g., LinkedIn, Twitter). Thirteen participants responded to our invitation including four developers from Mozilla Firefox. About 23% of these participants have more than 10 years of professional development experience, 15% of them have 5 to 10 years of experience and about 39% have 1 to 5 years of development experience. Our study was non-paid.

3.4 Analyzing Agreement between Independent Findings

Once the key factors behind bug non-reproducibility are identified, we validate them with two independent studies (Steps 6-7, Fig. 1). First, we investigate how these factors are assessed by the professional developers. We not only determine the severity of each factor but also gain further actionable insights from this developer study. Second, we determine how these factors match with the earlier findings of Joorabchi et al. [38]. While Joorabchi et al. adopt an ad hoc method, we employ a systematic method namely *Grounded Theory* [29] for the qualitative analysis. We also use a different dataset in this work. Thus, an agreement between our identified factors and the earlier ones would indicate their generalizability and thus also would strengthen the understanding of bug non-reproducibility.

3.5 Construction of Training Dataset for Machine Learning Models

Our empirical study and developer study provide useful insights on non-reproducible software bugs from a qualitative perspective. Using machine learning-based classification models, we further differentiate between reproducible and non-reproducible bug reports from a quantitative perspective.

We select a total of 17 features (12 structural + 2 textual + 3 semantic) from each bug report to train our classification models. Table 4 provides a brief overview of our selected features. Although our selection has been inspired by relevant existing studies [30, 27], we focus on collecting the features that might be available during the submission of a bug report. Goyal and Sardana [30] first use eight structural features (e.g., component, priority, severity) and one semantic feature (e.g., sentiment) from bug reports to separate the non-reproducible bugs from the reproducible bugs. They suggest that the sentiments expressed in the non-reproducible bug reports could be different from that of the reproducible reports. We thus select six out of their eight features to train our models. We discard the features that might not be available during the submission of a bug report (e.g., number of discussion comments). Fan et al. [27] combine a set of structural (e.g., presence of stack traces) and textual features (e.g., readability) from a bug report and design a machine learning model to separate the invalid bug reports from the valid bug reports. We select one textual feature –readability– from this work.

Thus, we select a total of 17 features capturing three different aspects of a bug report (e.g., structural, textual, semantic, Table 4) to train our models.

Once the features are selected, we collect their values from each of the 1,109 bug reports (576 non-reproducible + 533 reproducible) in our dataset (Section 3.1). We first extract the structural features (e.g., component, priority, severity) using the *Jsoup* library through web scraping. Then we determine the readability of texts from each bug report (i.e., *title* + *description*) using five popular readability metrics –Flesch-Kincaid Grade Level, Gunning-Fog Score, Coleman-Liau Index, SMOG Index and Automated Readability Index [45]. We calculate the readability of *title* and *description* fields separately since they contain different levels of technical details. Then we also collect the semantic feature from each bug report by analyzing its texts with a sentiment analysis tool. We use a popular tool namely Stanford CoreNLP [55] and calculate the number of positive, negative and neutral statements in the *title* of each bug report. One might argue about the use of Stanford CoreNLP library [55]. According to a recent study [39], other tools such as SentiStrength [57] have been found to be more accurate to detect sentiments in software engineering texts (e.g., JIRA issue comments). We thus also apply SentiStrength [57] to our dataset, detect word-level positive and negative sentiments in the *title* of each bug report, and then repeat our experiments.

One might argue about the data imbalance problem in our dataset, i.e., 576 non-reproducible and 533 reproducible bug reports. To address this, we apply down-sampling to the collection of non-reproducible bug reports during training phase, and make our dataset balanced. Then we repeat our experiments using the balanced dataset with RandomForest, the best-performing algorithm with our original dataset. We use WEKA toolkit [2] to train and test our model.

3.6 Selection of Model Interpretation Framework

Using machine learning models and their interpretation frameworks, we further explain the differences between reproducible and non-reproducible bug reports. Over the years, there have been several popular frameworks such as SHAP [40] and LIME [50] to interpret the machine learning models. The SHAP framework considers all possible predictions for an instance by considering all possible combinations of input features, which makes it slow but accurate. On the other hand, the LIME framework builds sparse linear models to explain each prediction for an instance, which makes it fast but less accurate. More importantly, SHAP offers appropriate explanations for tree-based models (e.g., RandomForest, XGBoost) whereas LIME is mainly suitable for simpler models such as KNN[2]. Since we use tree-based models in our classification, we use the SHAP framework [40] to interpret the results of our machine learning models (Section 4.6).

4 Study Results

In this section, we present the results of our study by answering the seven research questions as follows.

[2] https://bit.ly/39Qh2eH

Table 2: Key Factors behind the Non-Reproducibility of Software Bugs

Key Factor	**Overview**
F_1: Bug Duplication	The bug might have been already fixed in the recent releases
F_2: False Positive Bug	The reported issue might not be a bug, but rather indicates a non-existent software feature
F_3: Bug Intermittency	The bug does not occur frequently or consistently
F_4: Missing Information	The required information (e.g., steps to reproduce) is missing in the bug report
F_5: Ambiguous Specifications	The expected behaviour of the software application is misunderstood by the reporter
F_6: Performance Regression	Performance loss that is encountered as a side effect of recent changes
F_7: Lack of Cooperation	The reported bug fails to draw the attention of the stakeholders (e.g., developers)
F_8: Memory Misuse	The bug is triggered by the mismanagement of memory
F_9: Third-Party Defect	The bug has been triggered by defects in a third-party component
F_{10}: Restricted Security Access	Bugs that warrant specialized authentication or authorization for reproduction
F_{11}: Touch & Gestures	The accessibility bugs that warrant touches, gestures and special interactions

4.1 RQ$_1$: What are the key factors that make a reported software bug non-reproducible?

We identify a total of 11 key factors that are likely to explain the non-reproducibility of software bugs. Table 2 shows the identified factors from our qualitative analysis (Section 3.2). We explain each of these factors with illustrative examples collected from the bug reports as follows.

Bug Duplication (F$_1$) is one of the key factors behind the non-reproducibility of software bugs. The duplicate bugs are often known to the developers and thus might have been already fixed in recent releases. As a result, they cannot be reproduced with the up-to-date version of the software system. The following discussion comment from Firefox (Bug #1428773) refers to duplicate bugs and explains why it cannot be reproduced.

D: "This looks a lot like the issues in bug 1420748 and related bugs, so it might be fixed by the WR update in bug 1426116."

Since the duplicate bugs might already be fixed, they are often resolved by closing them as *duplicates* and then pointing their reporters to a recent software version that contains the fixes.

False Positive Bugs (F$_2$) (a.k.a., feature requests) might not be reproducible. Software users often discuss about non-existent features in their bug reports; this cannot be reproduced. Sometimes, they also report configuration issues as bugs, which can be resolved with simple tweaking rather than code level changes. For example, the following comment from Firefox (Bug #1444194) shows how a simple configuration tweak can resolve a bug regarding slow network proxy.

D: "What happens if you set the about:config pref "security.OCSP.enabled" to 0?"
R: "yep disabling the OCSP check fixed it. Then this works fine .."

Bug Intermittency (F$_3$) is another key factor behind bug non-reproducibility. Intermittent bugs have non-deterministic properties and they do not occur frequently or consistently. Thus, they are difficult to reproduce and fix. The following discussion comment from Eclipse (Bug #501488) explains the intermittency of a reported bug.

R: "I forgot to write that the problem does not always appear, sometimes working as it should. I have inserted a screenshot of the problem."

Missing Information ($\mathbf{F}_4$) is a key problem that makes a reported bug non-reproducible. Developers often look for relevant items in a bug report (e.g., steps to reproduce, stack traces, performance profiles, screenshot, test cases, system configurations) that could help them in reproducing the bug. Unfortunately, in practice, these items often are either missing or not reported carefully. The example comments from Eclipse (Bugs #476042, #477898) request for missing information in the bug report.

D: "Please try with Eclipse Neon M6 build and provide exact steps that reproduce the issue. Also, attach the log file having stacktrace."

D: "Please provide the code snippet that reproduces the issue."

Sometimes, the mere presence of the required items might not be sufficient. The reported bugs could also not be reproduced if the provided information is incomplete or inaccurate.

Ambiguous Specifications ($\mathbf{F}_5$) often lead the reported bugs to non-reproducibility. Bug reporters might misunderstand the expected behaviours of a software system if the specifications are not clearly defined. As a consequence, they might characterize a legitimate functionality as a bug, which could introduce confusion or disagreement during bug reproduction. For example, the following discussion comments from Firefox (Bug #1477421) indicate a misunderstanding of the specifications regarding autoplay for muted videos.

R: "Actual results: Video autoplays although the sound is muted. Expected results: Video should not autoplay."

D: "Muted videos are expected to autoplay. This is a design choice."

Performance Regression ($\mathbf{F}_6$) related bugs (a.k.a., performance bugs) are hard to reproduce. They are often subtle and subjective, or dependent on specific characteristics of a machine, which introduces confusion and disagreement among the stakeholders (e.g., reporters, developers) during bug reproduction. For example, the following comment from Firefox (Bug #1485402) indicates the subtle and subjective natures of the performance bug.

*D: "On my 24-core desktop machine, I'm seeing Firefox Nightly 63 being quite a bit *faster* than Chrome DevEdition 70. I wonder why I'm seeing the opposite of Stephen (reporter)..."*

Lack of Cooperation ($\mathbf{F}_7$) among the stakeholders is another key factor that could make a reported bug non-reproducible. An earlier study [60] also suggests that collaboration dynamics could play a major role during bug reproduction. Sometimes the reported bugs fail to draw the attention of human developers. They are closed by either the automated bots (e.g., Eclipse Genie) or the reporters and are marked as WORKSFORME. Bug reproduction might also fail due to the lack of response from the reporters. For example, the following comment from Eclipse (Bug #495568) indicates that the bug cannot be reproduced due to the lack of cooperation (and required information) from the bug reporter.

D: "No further feedback, closing. Please reopen if you can confirm the problem and provide reproducible examples."

Memory Misuse ($\mathbf{F}_8$) such as *memory leaks*, *memory overflows*, and *concurrent modifications* might trigger the complex bugs that are difficult to reproduce. A leak of a small object that is hardly noticeable during the execution of the program might cause the memory usage to grow unbounded. Such issues could also be compounded by legacy hardware. For example, the following comment from Firefox (Bug #1547586) indicates the complex nature of a memory related bug regarding excessive RAM usage.

D: "I am unable to reproduce. I created a new profile, opened ... until the page was finished loading, measured RAM ... and a Firefox about:memory report. Then I disabled accessibility services, and restarted with ..., and re-measured RAM. I did not see any significant change. I tried this on a 9 year old laptop and a 2 year old laptop and saw no significant differences."

Third-Party Defects ($\mathbf{F}_9$) are often responsible for non-reproducible bugs. Modern software systems are routinely developed with third-party dependencies (e.g., libraries, resources) and environmental specifications (e.g., OS, memory, hardware, plug-ins, anti-viruses [13]) that might trigger bugs and failures. However, these bugs might not be reproducible since the developers often do not have enough control over them, or do not have a way to install the same third-party software which is the root cause of the bug. For example, the following comment from Firefox (Bug #1427890) suggests that the bug could be specific to an operating system.

D: "It works for me on Firefox 57 with windows 10. Since reporters use windows 7, maybe it is related to the windows version."

Restricted Security Access ($\mathbf{F}_{10}$) is another important factor behind the non-reproducibility of software bugs. Although bug reports are supposed to provide the required information for reproducing the bugs, many confidential items (e.g., user credentials, security certificates) cannot be shared publicly. Thus, reproducing the end-user's experience accurately could be challenging. For example, if there is a bug with Firefox on Netflix and the developer does not have a NetFlix account, then she might not be able to reproduce it. The following comment from Firefox (Bug #1594272) indicates the non-reproducibility of a bug due to possibly restricted security access.

D: "Hi Mark, I wasn't able to reproduce the bug since I don't have an account but I've chosen a component for this bug ..."

Touch & Gestures ($\mathbf{F}_{11}$) are often hard to imitate precisely, which could make accessibility-related bugs non-reproducible. For example, the following comment from Firefox (Bug #1457726) discusses the challenges in reproducing a touch/gesture related bug.

D: "I have tested this issue on a Surface machine with Windows 10 x64 ... and haven't managed to reproduce the issue. After opening multiple tabs and tapping on the "x" close button, the tab is automatically closed"

Table 3: Prevalence of the Key Factors in Non-Reproducible Bugs

Proposed Study			**Joorabchi et al.**	
Key Factor	**Firefox**	**Eclipse**	**Key Category**	**All**
F_1: Bug Duplication	**26.83%**	**31.33%**	C_1: Inter-bug Dependencies	**45.00%**
F_2: False Positive Bug	**4.57%**	**21.67%**	-	-
F_3: Bug Intermittency	**26.22%**	2.61%	C_5: Non-deterministic Behaviour	3.00%
F_4: Missing Information	1.52%	**13.84%**	C_3: Insufficient Information	**14.00%**
F_5: Ambiguous Specifications	**5.18%**	**9.40%**	C_4: Conflicting Expectations	**12.00%**
F_6: Performance Regression	**8.54%**	1.83%	-	-
F_7: Lack of Cooperation	3.96%	3.66%	-	-
F_8: Memory Misuse	4.88%	1.00%	-	-
F_9: Third-Party Defects	1.83%	2.35%	C_2: Environmental Differences	**24.00%**
F_{10}: Restricted Security Access	4.27%	0.00%	-	-
F_{11}: Touch & Gestures	2.44%	0.00%	-	-
Miscellaneous	9.76%	12.53%	C_6: Others	2.00%

Summary of RQ$_1$-(a): There are at least **11 key factors** (e.g., *bug duplication, intermittency, missing information, ambiguous specifications, third-party defects*) that could lead a reported software bug to non-reproducibility.

To better understand the importance of each of the identified factors, we analyze their prevalence in our dataset containing non-reproducible bugs (Section 3.2). We determine the presence of one or more key factors in each bug report and then summarize our findings (Table 3) as follows.

Table 3 shows the prevalence of 11 key factors in our dataset. We see that bug duplication is a major factor behind bug non-reproducibility. About 29% of bugs from the dataset cannot be reproduced since they are duplicates and were possibly fixed earlier. Both Firefox and Eclipse systems encounter a significant number of duplicate, non-reproducible bugs (e.g., 27%–31%). Bug intermittency is another prevailing factor behind the bug non-reproducibility. On average, 14% of the bugs do not occur frequently and consistently, which makes it hard to reproduce them. Up to 26% of the Firefox bugs are intermittent in nature. Developers also fail to reproduce at least 8% of the bugs due to missing information (e.g., steps to reproduce, stack traces, test cases). This problem is especially severe for Eclipse where $\approx$14% of the bug reports lack the required information for reproduction. During our analysis, we also note that the mere presence of items might not be sufficient rather they should be complete and accurate. Ambiguous specification is another important factor that could lead $\approx$8% of bugs to non-reproducibility. That is, the expected behaviours were either ill-defined or outdated, and the users considered the legitimate software behaviours as bugs. From Table 3, we also see that performance regression and false-positive bugs are also two important factors behind the non-reproducibility of bugs. Minor performance losses as a side effect of recovery from the critical bugs are often acceptable to the developers. Hence, they might be reluctant to reproduce these performance bugs. On the other hand, in false-positive bug reports, the reporters complain about non-existent software features, which are impossible to reproduce. The remaining key factors (e.g., third-party defects, memory misuse, restricted security access) lead $\approx$12% of the bugs to non-reproducibility, which is also a significant amount. Finally, $\approx$11% of the bugs from our dataset are application-specific (e.g., video player autoplay problem, refactoring failure) that cannot be reproduced due to miscellaneous reasons (e.g., novice mistakes).

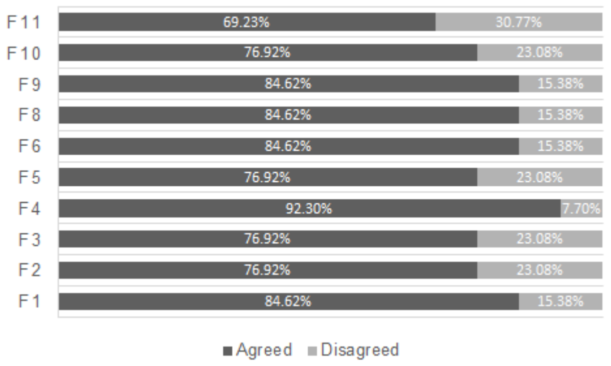


Fig. 3: Developers' responses on the key factors behind bug non-reproducibility

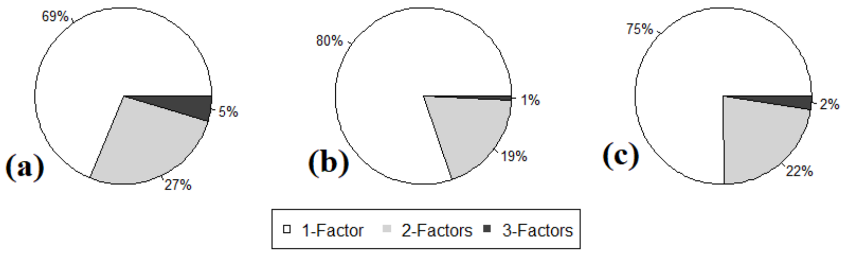


Fig. 4: Prevalence of key factors behind bug non-reproducibility (a) Firefox, (b) Eclipse, and (c) both software systems

We also investigate how one or more key factors might lead the software bugs to non-reproducibility. Fig. 4 summarizes our analysis from this investigation. We found that 75% of the non-reproducible bugs from our dataset cannot be reproduced due to one of the key reasons (e.g., bug duplication, intermittency, missing information). On the other hand, about 22% of the bugs have two key factors and 2% of the bugs have three factors behind their non-reproducibility. Bug non-reproducibility due to multiple factors might be more difficult to resolve than that due to single factor.

Summary of RQ$_1$-(b): About **75%** of the selected bugs are non-reproducible because of single key factors whereas the remaining ones are made non-reproducible by a combination of two or more key factors (e.g., *intermittency + regression*).

4.2 RQ$_2$: What do the professional developers consider to be key factors behind the non-reproducibility of bugs?

In the developer study, we present our identified factors behind bug non-reproducibility (Table 3) to the professional developers. We collect their responses on whether they agree or disagree with these factors. Fig. 3 shows that about 92% of the participants (i.e., 12 of 13) consider missing information (F$_4$) to be a major cause of bug non-reproducibility. About 85% of the participants (i.e., 11 of 13) agree that duplicate bugs (F$_1$), performance bugs (F$_6$), memory misuse related bugs (F$_8$), and third-party defects (F$_9$) are often hard to reproduce. According to 77% of the professional developers (i.e., 10 of 13), false-positive bug reports (F_2), bug intermittency (F_3), ambiguous software specifications (F_5), and restricted security access (F_{10}) could lead the reported bugs to non-reproducibility. Finally, 69% of the developers (i.e., 9 of 13) agree that touch and gesture related bugs (F_{11}) could also be difficult to reproduce due to their subtle, subjective nature. All these findings above suggest that the professional developers agree to a large extent with our identified factors behind the non-reproducibility of bugs.

We also provide free-form text boxes in our questionnaire to allow the developers to mention any factor that was not included in our list. Two more important causes were identified from their responses. First, hardware faults are very hard to reproduce. These bugs might need a specific combination of hardware and software (e.g., device drivers) and a long set of steps to reproduce. They also align with one of our factors– *third-party defects* (F_9). Second, bugs connected to a random function that is initialized with an unknown seed could also be hard to reproduce due to their non-deterministic nature.

Summary of RQ$_2$: About **70%–90%** of the professional developers agree with the factors behind bug non-reproducibility derived from our empirical study. They also point out two additional types of bugs (e.g., hardware faults, random function bugs) that are difficult to reproduce.

4.3 RQ$_3$: Do the identified factors match with the earlier findings from the literature?

Our first research question (RQ$_1$) identifies a list of key factors behind bug non-reproducibility (Table 2) using qualitative analysis, which were cross-examined by a group of professional developers (RQ$_2$). However, the generalizability of bug non-reproducibility could be further strengthened by validating these factors against the previously reported causes [38]. Joorabchi et al. [38] report six major causes of bug non-reproducibility. Table 3 shows their reported causes. We analyse each of our key factors, identify the semantically equivalent causes from their list by consulting their corresponding explanations and examples, and then determine the agreement between these two lists as follows.

From Table 3, we see that five of our key factors can be comfortably mapped to their Top-5 causes as follows. First, both their study and ours suggest that bug duplication (i.e., $F_1 \leftrightarrow C_1$) is the most dominant factor behind bug non-reproducibility. That is, a significant number of non-reproducible bugs are already fixed. Second, while bug intermittency (i.e., $F_3 \leftrightarrow C_5$) is another important factor according to our analysis, Joorabchi et al. found it less important. Third, both studies agree that missing information (i.e., $F_4 \leftrightarrow C_3$) is a major factor that could lead $\approx$14% of the reported bugs to non-reproducibility. Fourth, ambiguous specifications semantically match with conflicting expectations (i.e., $F_5 \leftrightarrow C_4$) due to misunderstanding of the software's correct behaviours. Both their study and ours report this as an important cause behind bug non-reproducibility. Fifth, third-party defects and environmental differences could also be considered as equivalent causes of bug non-reproducibility (i.e., $F_9 \leftrightarrow C_2$). The environmental differences are mostly created by the third-party items (e.g., operating system, network configurations) and the software bugs triggered by them could be hard to reproduce for the developers.

Thus, in essence, our study reproduces all the major causes reported by Joorabchi et al., which strengthens the generalizability of our findings.

Besides the existing equivalent causes, we also identify several novel causes of bug non-reproducibility that were not previously known. For example, we found that false-positive bug reports could be a major source of non-reproducibility (e.g., F_2, 14% bugs). We also found that performance bugs could be difficult to reproduce since they are often subjective in nature. Minor performance loss as a side effect of critical changes is often overlooked by the developers. According to our analysis, memory misuse related bugs (e.g., memory leak, memory overflow) are also hard to reproduce. We also found three other factors – lack of cooperation, restricted security access, touch & gestures – that could be responsible for 7% of non-reproducible bugs. Furthermore, our developer study reveals two more complex bugs (e.g., hardware faults, bugs connected to random function) that could be very hard to reproduce.

We also compare our findings with the earlier ones [38] in terms of assigned cause categories. According to Joorabchi et al., each bug report could be non-reproducible due to only one major cause. However, we found that at least 25% of the reported bugs could be non-reproducible because of a combination of two or more factors.

Summary of RQ$_3$: Five of our key factors match with the previously reported causes of bug non-reproducibility [38]. Our study also reports **eight novel factors** (false-positive bugs, performance regression, lack of cooperation, memory misuse, restricted security access, touch & gestures, hardware faults, bugs from random functions) that could lead the reported bugs to non-reproducibility. Thus, our study strengthens the understanding of bug non-reproducibility both by confirming the earlier findings and by uncovering new factors.

4.4 RQ$_4$: How do the professional developers deal with non-reproducible bugs in practice?

In our developer survey, we ask the developers about how they handle the non-reproducible bugs in practice. We wanted to know what actions they take when the reported bugs cannot be reproduced due to various causes (e.g., Table 2). We carefully analyze their qualitative responses against our questions, detect the general themes, and then summarize their actions in respect to the non-reproducible bugs as follows.

(a) Duplicate, non-reproducible software bugs are generally closed by the developers. That is, if the developers find a duplicate bug to be working in the latest release, they might close it as WORKSFORME. They could also try to find whether the patch of the original bug solves the bug at hand, and then mark this bug as a DUPLICATE of the original bug.

(b) Developers generally close the false-positive bugs as INVALID. They also consult with official functional specifications and occasionally send an explanation to the bug reporters. For example, according to one participant, *"I write a comment explaining why it is a false positive and then close the bug."*

(c) Developers attempt to collect useful information from various sources when they encounter intermittent bugs. For example, if the bug leads to a system crash, they ask for crash dump from the reporter. They also look for debugging logs or system logs associated with the bug, which can provide them rich contexts or insights. They also search for fellow developers and testers who might have experience with similar bugs, and then delegate the reproduction task to them. The intermittent bugs are also marked as low priority bugs by the developers. That is, if they are not encountered again for a certain period (e.g., 12 weeks), they are eventually closed.

(d) Developers request for more information (e.g., steps-to-reproduce, screencast) from the reporters when the bugs cannot be reproduced due to missing information. If the information is not provided in a timely fashion (e.g., two weeks), then the bug is closed as WONTFIX.

(e) Developers ask for further clarifications from the reporters when the bugs cannot be reproduced due to conflicting expectations. In particular, they explain the expected outcome of a software based on official functional requirements, request for the clarifications and then close the bug if no feedback is received within a certain period (e.g., two weeks).

(f) Developers request for performance tracing information (e.g., performance profiles) when they deal with performance bugs. Such information might help them identify the source of performance bottleneck. They also look for the colleagues who might have relevant expertise.

(g) Developers carefully examine the third-party dependencies, their versions and compatibility when they encounter non-reproducible software bugs triggered by third-party defects. They also check the logs for potential clues and use docker containers for more in-depth investigation. While they emphasize on using only authentic, well-tested plug-ins, many of them are in favour of banning such third-party components that have a strong negative impact upon the main applications (e.g., Firefox browser).

(h) Non-reproducibility of software bugs due to restricted security access is a major concern for the developers. They often work with the reporters closely to help her debug and potentially fix the bug. They also request for regression range from the reporter that might have induced a bug. They might also collect appropriate permission and dummy accounts from the testers to reproduce the reported bugs.

(i) Developers attempt to optimize their code when they deal with memory misuse related bugs. They increase the memory size for their application and perform extensive debugging to avoid any potential memory leaks. Although the developers claim that they hardly encounter touch/gesture related bugs, they want to take help from the experts in dealing with these bugs.

Summary of RQ$_4$: Developers manually identify and close the duplicate and false-positive bug reports. They often look for useful, complementary information from multiple sources when they deal with complex bugs such as intermittent bugs, performance bugs or third-party bugs. They also work closely with various stakeholders (e.g., fellow developers, testers, reporters) and often delegate bug reproduction task to them.

4.5 RQ$_5$: How to prevent the non-reproducibility and/or improve the reproducibility of reported bugs?

In our developer study, we ask the developers about how the research community might be able to assist them in dealing with non-reproducible bugs. Given their responses and our empirical analysis, we provide a list of actionable insights both for *preventing* the non-reproducibility and for *improving* the reproducibility of software bugs as follows.

(a) Develop intelligent tools for detecting the duplicate bugs. About 29% of the non-reproducible bugs are duplicate bugs, which are already fixed (e.g., Table 3). Most of these bugs are marked as duplicates by the developers during their failed attempts for reproduction. These reproduction efforts could be saved by carefully detecting the duplicate bugs before their submission. One of our study participants responds, *"Help finding duplicate bugs automatically."* Unfortunately, many existing tools for detecting the duplicate bugs might not be mature enough for practical use. In particular, they simply rely on textual features [23, 59], meta data from bug reports (e.g., products, components) [12] or execution traces [63] for detecting the duplicate bugs, and as a consequence, might fail to detect the complex duplicate bugs that have different symptoms but share the same root causes. Therefore, intelligent tools or techniques are warranted that can accurately detect the duplicate bugs during their report submission and thus can save the wasted efforts in failed reproduction. Furthermore, by putting together multiple similar bugs that share the same root cause, such a tool might equip the developers with enough information for a single bug. One of our developer participants also confirms – *"The more information, the better."*

(b) Develop intelligent tools for detecting the false-positive bug reports. About 5%–22% of the non-reproducible bugs are false-positive bugs. They often discuss the non-existent features of a software system that can be neither exercised nor reproduced by the developers. However, this non-reproducibility is detected by the developers during their failed reproduction attempts and deliberations, which could be costly. Thus, intelligent tools that can detect the false-positive bug reports during the submission could save valuable development time and efforts. A few existing technique [14, 41] attempt to separate the bug reports from the feature requests by analysing their textual features, which might always not be enough. In particular, the underlying semantics could be crucial to separate the software bugs from the features. Thus, further research is warranted to prevent the submission of feature requests as bug reports in the bug-tracking system.

(c) Complement the incomplete bug reports. Bug reports often lack the elements that are crucial to bug reproduction (e.g., steps to reproduce, expected behaviour, stack traces) [21]. A few studies [70, 22] attempt to reproduce the reported bugs by constructing appropriate test cases from the available information in the bug reports (e.g., steps to reproduce). Unfortunately, they are not sufficient since they are likely to fail when the bug reports lack the required information. Thus, more intelligent tools and techniques are warranted that (1) can help the reporters improve their bug reports during submission or (2) can automatically complement the incomplete bug reports by leveraging the historical information. For example, incomplete bug reports could be complemented with partial but valuable information collected from their duplicate or similar bug reports (e.g., stack traces, screen shots).

(d) Improve software specifications and documentations. A significant fraction of the reported bugs (e.g., 8%) cannot be reproduced due to conflicting expectations between the reporters and the developers. Such a conflict is often triggered by an incorrect understanding of the expected behaviours of a software system, which underscores the need for up-to-date, readable software specifications. One of our study participants responds – *"I do see that some companies' documentations are vague or ambiguous, so developers, QA, managers, or users may not have a clear understanding on the requirement."* There have been a few tools for creating software documentations from the code (e.g., Doxygen [7], srcML [10]). Since they provide API-level documentations, they might be useful to the developers but not to the users of a software who need more high-level documentations. Thus, further research is warranted on (1) how to validate the correctness of existing software documentations, (2) how to improve the poor-quality software documentations, and (3) developing tools and techniques that can offer suitable, high-level documentations for software users. Tools that can point the users to the right location within the software documentations could also be very useful.

(e) Develop appropriate sandbox to assist in the bug reproduction. To investigate several complex bugs (e.g., intermittent bugs, concurrency bugs, performance bugs), software systems need to be executed repeatedly. For example, intermittent bugs are non-deterministic and their true characteristics could only be understood from multiple executions. Developers might need to contrast between a normal execution and a crash using their memory dumps when they deal with memory/concurrency bugs. They might also need to compare among the performance profiles from multiple executions to identify the performance bugs. According to the developers, there is a marked lack of such tools and technologies that could help them execute their software applications repeatedly and reproduce these complex bugs. For example, one of our developer participants responds, *"It would be interesting to have a tool that allows the run of a task multiple times and reporting some relevant information as memory usage, dependencies errors, etc."* There have been a few relevant tools (e.g., Firefox Profiler [8], rr [43], Pernosco [9]). *Firefox Profiler* can analyse the performance profiles of Firefox and the Gecko browser engines. On the other hand, *rr* and *Pernosco* can record program executions during testing, which could be useful for debugging. However, many of these tools might be limited in scope (i.e., Firefox-specific) or not well-adapted for reproducing bugs. Thus, further research is warranted to come up with an appropriate sandbox for reproducing the bugs.

(f) Find the people with right expertise automatically. Software developers often look for fellow developers and testers with relevant expertise during reproducing complex bugs (e.g., intermittent bugs, concurrency bugs, performance bugs). Although the search might be trivial for a small development group, it could be a major challenge for a geographically distributed, large group. Besides, the relevant expertise might not be obvious and could be hidden as low-level code changes within the version control history. Thus, an intelligent tool for finding the right people might greatly help the developers. There have been a rich literature on finding experts during bug triaging [52]. Many of these studies simply rely on the texts of a bug report rather than its semantics (e.g., bug types, root causes) to find similar past bug reports and then suggest their assigned developers as experts, which might not be effective enough for practical, widespread adoption. Many existing techniques also rely on naive heuristics (e.g., commit history, code churn)

Table 4: Features Used for the Bug Report Classification

Feature	**Category**	**Description**
component	Structural	Name of the component that a reported bug is related to.
priority		Priority of a reported bug. It has five different levels (e.g., P_1–P_5), where P_1 refers to the highest priority.
severity		Severity of a reported bug. It can be *blocker*, *critical*, *major*, *normal*, *minor*, or *trivial*.
isDependent		Indicates whether a reported bug depends on other bugs from the bug tracking system.
numberOfDepends		Number of bugs on which a reported bug depends on.
doesBlock		Indicates whether a reported bug blocks one or more existing bugs from the bug tracking system.
numberOfBlocked		Number of bugs blocked by a reported bug.
reporterIsAssignee		Indicates whether a bug report has been assigned to its reporter for the fixing.
hasCC		Indicates whether a reported bug has been forwarded to one or more developers.
numberOfCC		Number of developers to whom a reported bug has been forwarded.
hasAttachment		Indicates whether a bug report includes one or more attachments.
numberOfAttachment		Number of attached items to a submitted bug report.
titleReadability	Textual	Readability score of the title from a bug report.
descReadability		Readability score of the description from a bug report.
positiveCount	Semantic	Number of positive statements in the title of a bug report.
negativeCount		Number of negative statements in the title of a bug report.
neutralCount		Number of neutral statements in the title of a bug report.

as a proxy of developer's expertise [47, 58], which might not be enough. Thus, developing more effective tools for finding the experts during bug reproduction could be a scope for future research.

Summary of RQ$_5$: Software developers need intelligent, effective tools for (1) detecting the duplicate or false-positive bug reports, (2) complementing the incomplete bug reports, (3) improving the software documentations, and (4) finding the people with right expertise. They also need a sandbox tool where they can repeat their experiments as a part of reproducing the complex bugs (e.g., performance bugs) and exploring the execution space of their software systems.

4.6 RQ$_6$: Can we leverage machine learning to characterize non-reproducible software bugs?

Our previous research questions explain why the reported bugs might not be reproducible (RQ$_1$-RQ$_4$) and how their reproducibility could be improved (RQ$_5$). In this section, we further contrast between non-reproducible and reproducible bugs, classify them using machine learning models, and then interpret the model predictions using the SHAP framework [40]. Our goals are to (a) explain how the non-reproducible bugs might differ from the reproducible bugs in terms of traditional features (e.g., severity, attachment, readability), and (b) suggest how they could be improved. We thus answer our research question RQ$_6$ as follows.

(a) Model training: We train our classification models using five popular machine learning algorithms – Naive Bayes [37], Logistic Regression [19], J48 [46], RandomForest [18], and XGBoost [24]. Naive Bayes assumes independence among features and predicts a class based on their maximum log-likelihood estimation. Logistic regression assumes a linear relationship between the features and classes, and provides a binary response using a sigmoid function. Unlike the Logistic regression, J48 is a decision-tree based algorithm that assumes a non-linear relationship between the features and classes. On the other hand, RandomForest is an ensemble learning technique that predicts a class based on the responses from multiple decision trees. Finally, XGBoost is the state-of-the-art tree-based ensemble learning technique that adopts an extreme gradient boosting method [24]. We use 10-fold cross-validation to train and test our classification models. To train our first four models, we use WEKA tool whereas the XGBoost model was trained using `xgboost` from the `scikit-learn` library. Each of these models was trained using the default parameters and thresholds that were provided by these libraries.

(b) Classification results and discussions: Table 5 summarizes the results of our five classification models. We see three important findings. First, all five models perform well and achieve high precision, high recall, and high F1-score. While RandomForest and XGBoost models stand out, the other three models can be considered as *comparable* in their performance. Second, with non-reproducible bug reports, recall is relatively higher than corresponding precision for each of our five different models. We conducted Mann-Whitney Wilcoxon and Cliff's delta tests on these precision and recall measures and found the recall measures to be significantly higher than precision measures with a *large* effect size (i.e., *p-value*=0.008, δ=1.00). On the other hand, our models deliver high precision and relatively low recall for the reproducible bug reports. We also conducted Mann-Whitney Wilcoxon and Cliff's delta tests on these precision and recall measures and found the precision measures to be significantly higher than recall measures with a *large* effect size (i.e., *p-value*=0.008, δ=1.00). These findings suggest the inherent differences between the two types of bugs (and their corresponding reports). Third, despite the sophistication (e.g., extreme gradient boosting), the XGBoost model could not always outperform the traditional RandomForest model with our bug reports.

We train our classification models with different combination of features and evaluate their performance using 10-folds cross-validation. Table 5 shows the effectiveness of three different feature types–*structural*, *textual*, and *semantic* (Table 4). When trained with 12 structural features, we see that the XGBoost model performs the best with non-reproducible bugs (e.g., 80% precision, 84% recall). Its performance is comparable to that of RandomForest with the reproducible bugs. However, the XGBoost model outperforms all four other models with a small margin for the whole dataset and achieves 81% precision and 81% recall. When both structural and textual features (e.g., readability) are used in model training, we see that RandomForest outperforms other models in classification. It achieves more than 80% precision for both the non-reproducible and reproducible bugs with 78%–83% recall. Thus, as a whole, it delivers 80.50% precision, 80.40% recall, and 80.40% F1-score for a dataset of 1,109 bug reports. We also repeat our experiment using an extended dataset of 1,990 bug reports and find that RandomForest achieves 83.50% precision and 83.50% recall for the combination of structural and textual features. According to our experiments, the addition of semantic features (e.g., sentiment) did not improve the model performances. In fact, we notice that the performance dropped for many models due to the addition of sentiment features. However, our RandomForest model outperformed the other

Table 5: Classification of Bug Reports

Model	**Non-reproducible**			**Reproducible**			**All**		
	P	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
Using structural features only (1,109 bug reports)									
NB	75.20%	86.10%	80.30%	82.20%	69.20%	75.20%	78.50%	78.00%	77.80%
LR	76.60%	83.00%	79.70%	79.80%	72.60%	76.00%	78.10%	78.00%	77.90%
J48	75.60%	84.90%	80.00%	81.20%	70.40%	75.40%	78.30%	77.90%	77.80%
RF	79.30%	82.50%	80.90%	80.20%	76.70%	78.40%	79.70%	79.70%	79.70%
XGB	80.22%	83.84%	81.90%	81.80%	77.49%	79.49%	**80.98%**	**80.80%**	**80.74%**
Using both structural and textual features (1,109 bug reports)									
NB	74.30%	85.80%	79.60%	81.50%	67.90%	74.10%	77.80%	77.20%	77.00%
LR	76.00%	83.20%	79.40%	79.70%	71.70%	75.50%	77.80%	77.60%	77.50%
J48	75.80%	84.20%	79.80%	80.60%	70.90%	75.40%	78.10%	77.80%	77.70%
RF	80.00%	83.20%	81.50%	81.00%	77.50%	79.20%	**80.50%**	**80.40%**	**80.40%**
XGB	79.15%	82.29%	80.57%	80.48%	76.54%	78.30%	79.79%	79.54%	79.48%
Using both structural and semantic features (1,109 bug reports)									
NB	75.20%	86.80%	80.60%	82.90%	69.00%	75.30%	78.90%	78.30%	78.10%
LR	76.30%	83.30%	79.70%	80.00%	72.00%	75.80%	78.10%	77.90%	77.80%
J48	77.10%	84.20%	80.50%	81.00%	73.00%	76.80%	79.00%	78.80%	78.70%
RF	79.00%	82.10%	80.50%	79.80%	76.40%	78.00%	79.40%	79.40%	79.30%
XGB	79.11%	81.23%	80.08%	79.33%	76.73%	77.91%	79.21%	79.08%	79.04%
Using structural, textual, and semantic features (1,109 bug reports)									
NB	73.90%	85.90%	79.50%	81.50%	67.20%	73.70%	77.60%	76.90%	76.70%
LR	75.60%	83.00%	79.10%	79.50%	71.10%	75.00%	77.50%	77.30%	77.20%
J48	77.70%	83.90%	80.60%	80.90%	73.90%	77.30%	79.20%	79.10%	79.00%
RF	79.60%	83.30%	81.40%	81.00%	76.90%	78.90%	**80.30%**	**80.30%**	**80.20%**
XGB	79.13%	81.94%	80.42%	79.92%	76.55%	78.10%	79.51%	79.35%	79.31%
Using both structural and semantic features (SentiStrength) (1,109 bug reports)									
NB	75.20%	86.30%	80.40%	82.40%	69.20%	75.20%	78.60%	78.10%	77.90%
LR	76.40%	82.60%	79.40%	79.40%	72.40%	75.80%	77.90%	77.70%	77.70%
J48	76.00%	84.20%	79.90%	80.70%	71.30%	75.70%	78.30%	78.00%	77.90%
RF	78.80%	80.90%	79.90%	78.80%	76.50%	77.60%	78.80%	78.80%	78.80%
XGB	77.70%	82.27%	79.86%	79.86%	74.49%	76.99%	78.73%	78.54%	78.48%
Using textual, structural, and semantic features (SentiStrength) (1,109 bug reports)									
NB	74.00%	85.40%	79.30%	81.10%	67.50%	73.70%	77.40%	76.80%	76.60%
LR	76.00%	82.80%	79.20%	79.40%	71.70%	75.30%	77.60%	77.50%	77.40%
J48	76.20%	82.50%	79.20%	79.20%	72.20%	75.60%	77.70%	77.50%	77.50%
RF	78.70%	82.60%	80.60%	80.20%	75.80%	77.90%	79.40%	79.40%	79.30%
XGB	77.70%	82.27%	79.86%	79.86%	74.49%	76.99%	78.73%	78.54%	78.48%
Using structural and textual features (1,990 bug reports)									
NB	74.20%	85.20%	79.30%	82.10%	69.50%	75.30%	78.10%	77.50%	77.30%
LR	78.80%	84.80%	81.70%	83.10%	76.60%	79.70%	80.90%	80.80%	80.70%
J48	81.30%	83.20%	82.20%	82.30%	80.30%	81.30%	81.80%	81.80%	81.80%
RF	82.90%	84.90%	83.90%	84.10%	82.00%	83.00%	**83.50%**	**83.50%**	**83.50%**
XGB	81.85%	83.95%	82.80%	83.22%	80.73%	81.86%	82.53%	82.36%	82.34%
Using structural and semantic features (1,990 bug reports)									
NB	73.50%	85.90%	79.30%	82.50%	68.20%	74.70%	78.00%	77.20%	77.00%
LR	79.30%	83.70%	81.40%	82.30%	77.50%	79.80%	80.70%	80.70%	80.60%
J48	79.70%	85.50%	82.50%	83.90%	77.60%	80.60%	81.80%	81.60%	81.60%
RF	82.50%	83.30%	82.90%	82.70%	81.80%	82.20%	82.60%	82.60%	82.60%
XGB	82.81%	84.64%	83.66%	84.08%	81.96%	82.94%	**83.43%**	**83.32%**	**83.30%**
Using structural, textual, and semantic features (1,990 bug reports)									
NB	74.50%	85.80%	79.70%	82.70%	69.70%	75.70%	78.50%	77.90%	77.70%
LR	80.40%	85.10%	82.70%	83.70%	78.60%	81.10%	82.00%	81.90%	81.90%
J48	80.90%	82.10%	81.50%	81.30%	80.00%	80.60%	81.10%	81.10%	81.10%
RF	82.70%	85.50%	84.10%	84.60%	81.50%	83.00%	**83.60%**	**83.60%**	**83.60%**
XGB	82.90%	84.94%	83.87%	84.24%	81.95%	83.03%	83.56%	83.47%	83.46%

NB = Naive Bayes, **LR** = Logistic Regression, **RF** = RandomForest, **XGB** = Extreme Gradient Boosting, **P** = Precision, **R** = Recall, **F** = F1-score, **Emboldened** = The highest metric among the five techniques

four classification models when all three types of feature (e.g., *structural*, *textual*, *semantic*) were combined (e.g., 80.30% precision, and 80.30% recall).

We repeat our experiments with the sentiment features collected by *SentiStrength* tool [57] (Section 3.5), and Table 5 summarizes our classification results of reproducible and non-reproducible bug reports. We found that our core findings on sentiment features did not change and SentiStrenth was found to be comparable to Stanford CoreNLP with respect to our dataset and trained models.

While our original dataset (576 non-reproducible + 533 reproducible) is imbalanced, we also repeat our experiment with a balanced dataset using RandomForest, the best-performing algorithm, as shown above. Our model achieves 81.40% precision and 81.40% recall with the balanced dataset, which are only 1% higher than those with the original dataset (e.g., 1,109 bug reports). Thus, the data imbalance might have only negligible impacts upon our core findings.

We also repeat our experiments with an extended dataset of 1,990 bug reports (i.e., 1,009 non-reproducible + 981 reproducible). The details on dataset construction can be found in Section 3.1. Table 5 summarizes our experiment details. We found that the precision and recall measures of each model improved consistently with the extended dataset. For example, XGBoost model achieved 83.56% precision and 83.47% recall with this dataset. However, RandomForest model achieved the maximum of 83.60% precision and 83.60% recall when all three types of features – *structural*, *textual*, and *semantic* – were considered. Furthermore, we also note that the semantic features (e.g., sentiment) marginally improved the model performances when they were combined with the structural (e.g., presence of attachment) and textual features (e.g., title readability), which indicates the benefit of semantic features in the bug report classification task and thus also supports an earlier finding [30].

All these findings above clearly suggest that (a) non-reproducible and reproducible bug reports are different in their structural, textual, and semantic features, and (b) they can be separated using machine learning models with a high accuracy.

(c) Interpretation of the classification between non-reproducible and reproducible bugs: Our classification models were able to separate the non-reproducible bugs from the reproducible bugs where they used several structural, textual, and semantic features from the bug reports. In this section, we further investigate how these features might have helped our machine learning models classify the bug reports.

We use SHAP [40], a popular model interpretation framework, to interpret the classification results of our models. The SHAP value is the average marginal contribution of a feature (towards the model's prediction) across all possible combination of features [11]. It indicates whether a feature value can increase a model's prediction over a random baseline or not [40]. In our experiment, we perform a binary classification where bug reproducibility was considered as the positive class and non-reproducibility as the negative class. That is, our models attempt to predict the *bug reproducibility* by default. Thus, a positive SHAP value indicates an increase in our models' prediction towards bug reproducibility (i.e., positive class) whereas a negative SHAP value indicates an increase in the prediction towards bug non-reproducibility (i.e., negative class). Figures 5, 6, 7, 8, 9, 10, and 11 summarize our analysis using SHAP values as follows.

Fig. 5 shows the importance of each feature using a bee swarm plot from our RandomForest model. The bee swarm plot visualizes the SHAP value of a feature from each of the training instances on the x-axis and then sorts all features on the y-axis according to their sum of SHAP values. In our dataset, we encode a boolean response into a number and represent *false* as 0 and *true* as 1. Our plot indicates a low feature value with blue color and a high feature value with red

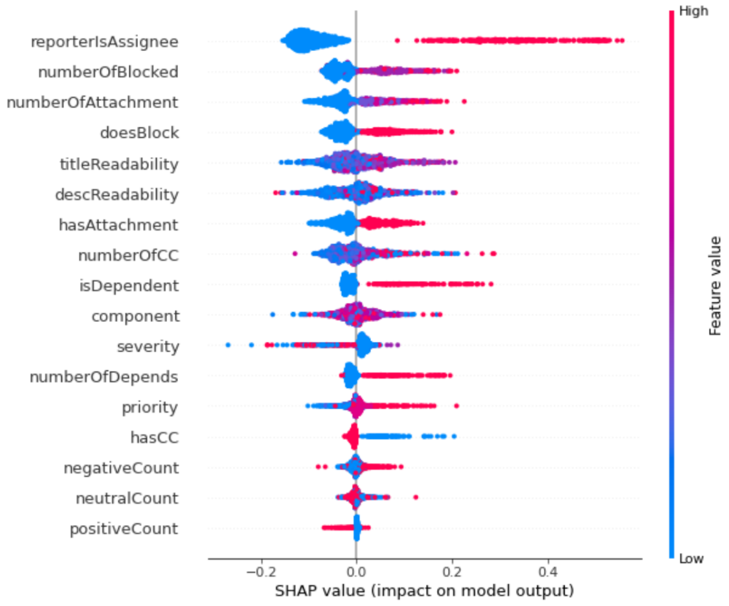


Fig. 5: Feature importance using bee swarm plot (RandomForest model)

color. Thus, the *false* response and small numerical values are represented with blue color whereas the *true* response and large numerical values are represented with red color. We see that *reporterIsAssignee* is the most important feature according to our RandomForest model. Interestingly, the same feature has been reported as the most important by both the XGBoost model (Fig. 6) and Logistic Regression model (Fig. 7). That is, whether a bug has been assigned to its original reporter is an important predictor of its reproducibility. We note that this feature with *true* response leads to positive SHAP values in all three plots, which indicates an increased prediction towards bug reproducibility. We also analyze our dataset for further insights on this. We found that 43.52% of our 533 reproducible bug reports were assigned to their reporters whereas the same statistic is only 3.47% for the non-reproducible bug reports. Thus, the bug reports assigned to their original reporters (or developers with the knowledge of similar bugs) are more likely to be reproducible than those from the others. This observation can be explained in two ways. First, they could reproduce the bugs since they were the ones who observed the manifestation of the bugs and knew the detailed contexts in which the bugs appeared. Second, the working knowledge from the project (e.g., bug fixing) might have equipped them with better insights to reproduce a reported bug.

We also analyze the next most important features – *doesBlock*, *hasAttachment*, and *isDependent*– from each of our three bee swarm plots (Figures 5, 6, 7). We see that their *true* responses are likely to increase our model's prediction towards bug reproducibility. We further investigate these features and found that (a) 51% of the reproducible bug reports have at least one attachment, (b) 46% of them block one or more bugs, and (c) 21% of them depend on other bug reports from the bug-tracking system. On the other hand, 29% of the non-reproducible bug reports have attachments, 16% of them block other bugs, and only 5% of them depend on the other bugs respectively, which are significantly lower. Thus, the

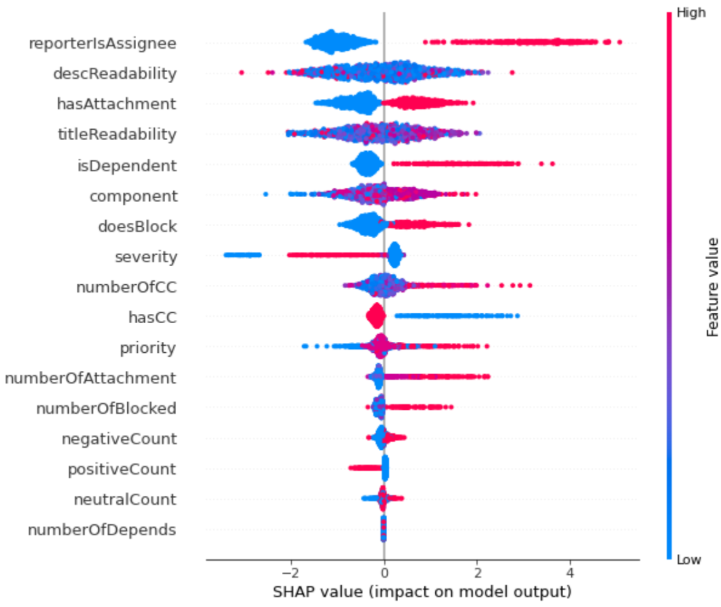


Fig. 6: Feature importance using bee swarm plot (XGBoost model)

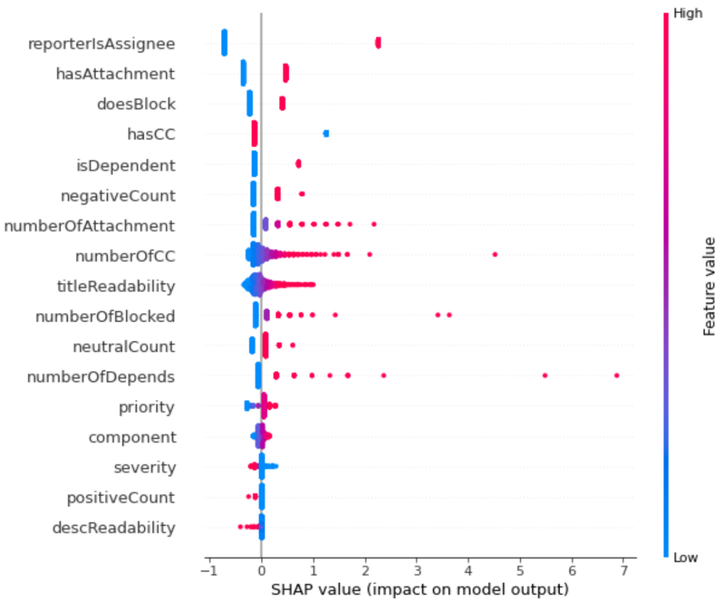


Fig. 7: Feature importance using bee swarm plot (Logistic Regression model)

presence of attachments, blocked bugs, or dependencies might be an indication of bug reproducibility. Our RQ$_7$ (Section 4.7) also explores the connected bugs to determine if they provide rich information that can help improve the non-reproducible bugs. While an attachment can offer complementary information,

Table 6: Feature Comparison between Reproducible & Non-Reproducible Bugs

Feature	**Reproducible**	**Non-Reproducible**
Bug ID	1468131	1574842
component	Layout	DOM: Security
priority	P_2	P_5
severity	normal	normal
IsDependent	false	false
numberOfDepends	0	0
doesBlock	true	false
numberOfBlocked	3	0
reporterIsAssignee	false	false
hasCC	true	false
numberOfCC	2	0
hasAttachment	true	false
numberOfAttachment	2	0
titleReadability	27.16	34.24
descReadability	151.88	130.65
positiveCount	0	0
negativeCount	0	0
neutralCount	1	1

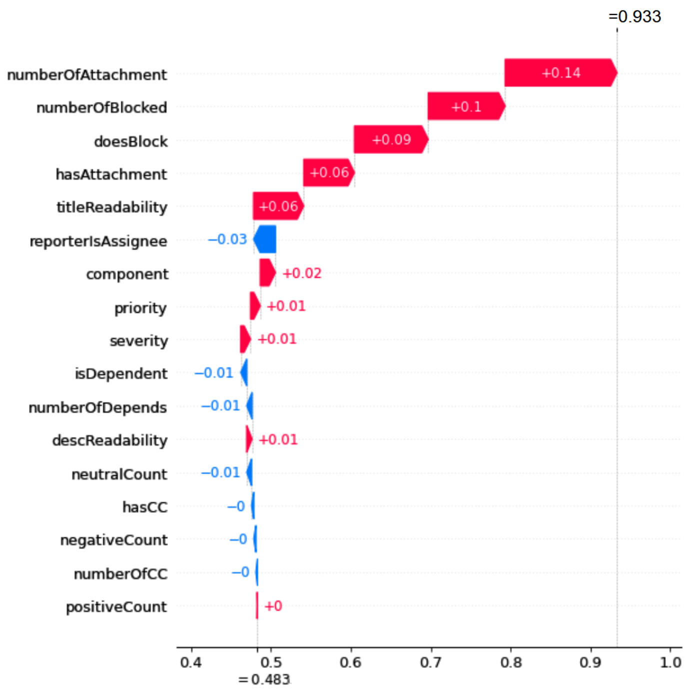


Fig. 8: Features importance using waterfall plot (RandomForest model)

the blocked bugs and dependencies could provide additional, actionable insights to reproduce a bug.

To verify the above findings, we further analyze one *reproducible* bug (ID #1468131) and one *non-reproducible* bug (ID #1574842) from *Firefox Core* component using waterfall (e.g., Figures 8, 10) and force plots (e.g., Figures 9, 11). Waterfall and force plots illustrate how different features contribute to classify an instance into the positive class. In our experiment, we consider *reproducible* as the positive class. From Fig. 8, we see that the presence of attachments and

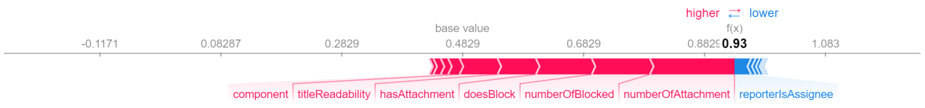


Fig. 9: Feature importance using force plot (RandomForest model)

blocked bugs plays a significant role by pushing our model's prediction for the example bug (ID #1468131) towards *reproducible bug*. According to the Table 6, this bug has two attachments and blocked three other bugs, which confirms our observation from Fig. 8. We also note that the absence of dependencies (i.e., *isDependent=false*, *numberOfDepends*=0) and the assignee not being the reporter (i.e., *reporterIsAssignee=false*) attempt to decrease our model's prediction towards bug reproducibility, as shown by their negative SHAP values. The force plot in Fig. 9 also illustrates how several features (e.g., *numberOfAttachment*, *numberOfBlocked*) increase our model's prediction towards bug reproducibility (i.e., positive class) whereas the others (e.g., *reporterIsAssignee*) decrease the prediction and thus push the prediction towards bug non-reproducibility. All these findings also align with our observations from the three bee swarm plots above (Figures 5, 7, 6). From Fig. 10, we see that the absence of blocked bugs, dependencies, attachments, and the reporter not being the assignee (see details in Table 6) prevent the second example bug (ID #1574842) from being classified as *reproducible* and push the model's prediction towards non-reproducibility. As shown in Table 6, this bug is originally non-reproducible. The force plot in Fig. 11 also reinforces the same idea. All these findings based on empirical evidence suggest a connection between the presence of these items (e.g., attachments, blocked bugs, dependencies) and the reproducibility of a software bug. We also notice that reproducible bugs are likely to have a high priority.

While the above analysis focuses on *structural* features, we also investigate the role of *textual* and *semantic* features in the bug report classification. From the Figures 5, 6, 8, 9, 10, and 11 above, we see that the readability of title or description texts is an important feature for classification. In particular, the high readability of title texts (*i.e., titleReadability*) often leads to increased model prediction towards *reproducible bug*. On the other hand, the readability of description texts has a mixed impact on the classification according to our plots. We also perform further manual analysis on our dataset to investigate this readability aspect. We found that the reproducible bug reports might require marginally higher reading grade-level[3] than non-reproducible bug reports to understand the concepts. That is, they are slightly more difficult to read and comprehend. During our manual analysis, the texts from non-reproducible bug reports were found to be generic, vague, and they did not provide enough technical information to reproduce the bugs. On the other hand, the reproducible bug reports often contained both regular texts and various structured/technical entities (e.g., stack traces, program elements, diffs), which were useful to reproduce a bug. It should be noted that we used traditional tools [45] to measure the readability of contents from bug reports. Since these traditional tools were originally designed for regular texts, they could be less than ideal for determining the readability of bug reports containing structured/technical entities (e.g., stack traces). Thus, the reproducible bug reports might have been considered

[3] https://bit.ly/3rZnUwL

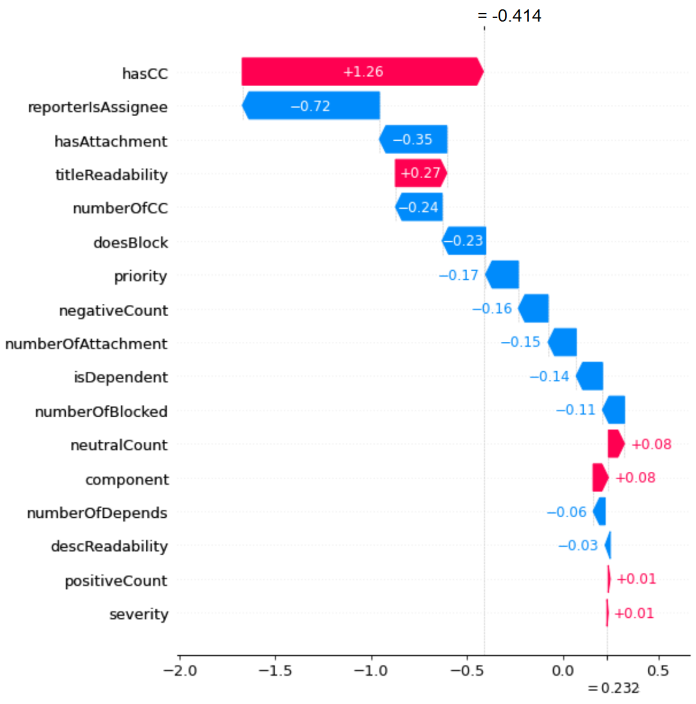


Fig. 10: Feature importance using waterfall plot (Logistic Regression model)

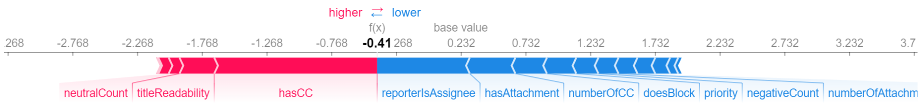


Fig. 11: Feature importance using force plot (Logistic Regression model)

as less readable than the non-reproducible ones. From Figures 10, 11, we also see that semantic features (e.g., positive count, neutral count) have a limited impact on model prediction, which can be confirmed by Figures 5 and 6.

We also repeat our investigations using machine learning models trained with the extended dataset of 1,990 bug reports (1,009 non-reproducible + 981 reproducible). Figures 14, 15, and 16 summarize our investigation details. We see that our findings about structural (e.g., *reporterIsAssignee*, *hasAttachment*) and textual features (e.g., *titleReadability*) hold with the extended dataset, which increases the confidence in our results and also indicates that our core findings might be generalizable. Furthermore, the semantic features (e.g., *positiveCount*) were also found to be less important than the others.

(d) Statistical analysis of the model features. In the above section, we attempt to differentiate between reproducible and non-reproducible bug reports using their structural, textual, and semantic features. Towards this end, we first classify the bug reports using five machine learning models and then identify the important model features using their SHAP values. To gain further insights regarding these features, we perform detailed statistical analysis using significance tests. In particular, we investigate whether these features have significant impact

Table 7: Impact of Model Features on Bug Reproducibility

Model Feature	χ^2	**DF**	**p-value**	**Critical** χ^2
component	58.64	31	0.002***	44.99
priority	44.69	5	1.67e-08***	11.07
severity	66.11	7	8.99e-12***	14.07
reporterIsAssignee	250.66	1	2.2e-16***	3.84
hasAttachment	51.71	1	6.43e-13***	3.84
isDependent	61.62	1	4.16e-15***	3.84
doesBlock	119.16	1	2.2e-16***	3.84
hasCC	38.02	1	7.01e-10***	3.84

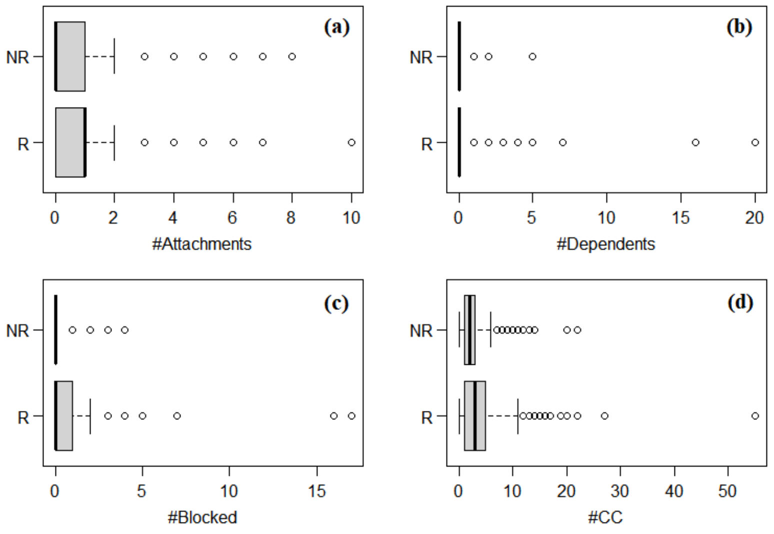


Fig. 12: Box plot of model features – (a) number of attachments, (b) number of dependent bugs, (c) number of blocked bugs, and (d) number of cc members. **R**=Reproducible, **NR**=Non-reproducible

on bug reproducibility and their distributions are significantly different across the two types of bug reports. Tables 7, 8, and Fig. 12 summarize our statistical analysis of the model features.

We select eight categorical features from our models and analyze their impact on bug reproducibility using χ^2-test. Table 7 presents our test details for each of these categorical features where the reproducibility status (e.g., reproducible, non-reproducible) is our dependent variable. We see that each categorical feature including the important ones from our SHAP-based analysis has a significant impact on bug reproducibility. For example, according to χ^2-test, *reporterIsAssignee* feature has a significant impact on the reproducibility of bug reports. The critical χ^2 value for a feature with 1 degree of freedom and a significance level of 0.05 is 3.84. The test involving *reporterIsAssignee* feature returns a χ^2 value of 250.66, which is 65 times higher than the critical value and thus indicates a significant impact on the dependent variable (i.e., reproducibility). Similarly, the other categorical features – *hasAttachment*, *isDependent*, and *doesBlock* – have 13, 16, and 31 times higher χ^2 values than their corresponding critical values, which also indicates their significant impact on bug reproducibility.

Table 8: Difference of Feature Distribution between Reproducible and Non-Reproducible Bug Reports

Model Feature	**t**	**DF**	**p-value**	**Cohen's d**
numberOfAttachment	7.33	907.51	4.93e-13**	0.45 (small)
numberOfDepends	5.06	590.01	5.51e-07*	0.32 (small)
numberOfBlocked	8.19	686.57	1.25e-15**	0.51 (medium)
numberOfCC	4.23	834.09	2.63e-05*	0.26 (small)
titleReadability	2.27	1,091.90	0.02	0.14 (negligible)
descReadability	0.29	1,044	0.77	0.02 (negligible)
positiveCount	-1.25	1,099.40	0.21	0.07 (negligible)

While the categorical features have significant impact, we also collect four relevant numerical features – *numberOfAttachment*, *numberOfDepends*, *numberOfBlocked*, and *numberOfCC*, and analyze their distribution using box plots. From Fig. 12-(a), we see that at least 50% of the reproducible bug reports have more than one attachment whereas such a statistic is only 25% for the non-reproducible bug reports. In Table 8, we also perform statistical significance tests using *t-test* and *Cohen's d*. According to an existing study [26], non-parametric statistical tests (e.g., Mann-Whitney Wilcoxon) might provide false-positive outcome for large sample sizes (e.g., 576 non-reproducible + 533 reproducible) and thus parametric tests should be preferred. Our Shapiro-Wilk test [51] with each of these numeric features indicates that they are normally distributed (i.e., *p-value*=2.2e-16). We thus perform a parametric test namely *t-test* to determine the significance of our features. We find that the number of attachments is significantly higher with a *small* effect size (i.e., *p-value*<0.05, *d*=0.45) in the reproducible bug reports than that in non-reproducible bug reports. The distributions of *numberOfDepends*, *numberOfBlocked*, and *numberOfCC* features in the two types of bug reports are not clearly separable, as shown in Fig. 12-(b), (c), and (d) respectively. However, according to our statistical tests in Table 8, their distributions are significantly different with a *small* to *medium* effect size. That is, the distribution of important features are indeed different between the two types of bug reports. Thus, to a large extent, our statistical analysis supports and complements the findings from the SHAP-based investigation above.

We also perform statistical tests on textual and semantic features such as *titleReadability*, *descReadability*, and *positiveCount*. As shown in Table 8, the readability of title is significantly different between the two types of bug reports, but the effect size is negligible. We also do not find any significant difference in description readability or sentiment between the two types of bug reports. These features were also not found to be important according to our SHAP-based analysis. Thus, our statistical analysis align with the findings from the SHAP-based analysis.

(e) Actionable insights from the model interpretation: The above analysis provides several actionable insights. First, according to our empirical evidence, the bug reports submitted by the individuals involved in a software project are more likely to be reproducible than those from the others. That means, while collecting additional information from a reporter is important, finding the right person having relevant experience might be even essential to reproduce a bug. For example, the submitter of a bug report could be assigned to the bug whenever it is feasible (e.g., contributing member in a project). This finding underscores the necessity of effective bug triage operation, which aligns with our earlier suggestion on finding the experts based on our developer study (RQ$_5$, Section 4.5).

Second, the inclusion of attachment, blocked bugs, and dependencies is likely to improve the chance of a bug report being reproducible since they might offer complementary information. In practice, these linked bugs might be detected and recorded manually by the developers long after the submission of a bug. However, if they can be automatically detected and included in the bug report during its submission, the bug might have a better chance of being reproducible. In our next research question–RQ$_7$, we further analyze the linked bug reports to determine if they actually contain meaningful information (e.g., screenshots, fixed code) that could provide the missing details for bug reproduction.

Third, the readability of the *title* field has an important impact on model performance according to our plots (e.g., Figures 7, 8). The high value of readability might help classify a bug report as *reproducible*. However, this high readability value indicates that one needs high reading grade-levels[4] to understand the reproducible bug reports. That is, the reproducibility of a bug might require more technical specifications in its bug report rather than generic, natural language texts. In other words, the reproducible bug reports could contain such technical texts that are easy to understand for the experts but difficult for both novice developers and non-technical users. Thus, to ensure a better communication between the bug reporters (e.g., non-technical users) and the software developers, we need appropriate tools that can help write such bug reports that are not only easy-to-understand but also contain the necessary technical details for bug reproduction.

Summary of RQ$_6$: Our RandomForest model was able to classify the reproducible and non-reproducible bug reports with **83.60**% precision and **83.60**% recall using a combination of 12 structural, 2 textual, and 3 semantic features. According to our findings, (a) bug reports submitted by *contributing members* of a project are more likely to be reproducible than from the others, and (b) the presence of *attachments*, *blocked bugs*, or *dependencies* in a bug report often improves its chance of being reproducible. All these findings underscore the necessity of novel tools that can help (a) triage the bugs more efficiently, (b) detect the linked bugs more accurately, and (b) improve the readability of contents in a bug report.

4.7 RQ$_7$: Can we automatically detect bug reports connected to a non-reproducible bug and leverage them to support its reproducibility?

Our analysis using the *Grounded Theory* method and a developer survey shows that non-reproducible bug reports often lack critical pieces of information. They need to be complemented with relevant information (RQ$_5$). Our analysis using machine learning models also suggests that the presence of connected bugs in a bug report could increase its chance of being reproducible (RQ$_6$). However, connections between existing bugs and non-reproducible ones are established by human developers over a long period of time. Automatic establishment of these connections during the submission of a bug report can equip the developers with complementary information during bug reproduction. We thus design a technique

[4] https://bit.ly/3rZnUwL

to automatically detect the connected bug reports to a given bug using Information Retrieval algorithms. Furthermore, we manually analyze 93 bug reports connected to 71 non-reproducible bugs to gain further insights.

(a) Automatically detecting bug reports connected to non-reproducible bugs. Connected bug reports are marked by human developers as either *depends on* or *blocked* by the non-reproducible bugs. That is, these bug reports might have some textual or semantic relevance with the non-reproducible bug reports, which can be leveraged using a traditional approach such as Information Retrieval (IR). We thus adapt an existing IR-based technique [68] to detect the connected bug reports from our corpus. Ye et al. [68] first use a combination of textual and semantic similarities to recommend relevant API documentation for a given query. Yang et al. [67] also use a combination of textual and semantic similarities to detect duplicates of a given bug report. Similarly, we apply a combination of textual and semantic similarities to detect the bug reports that are connected to a non-reproducible bug report. First, we calculate textual similarity between any two bug reports using BM25 algorithm [54], a popular, vector-space model-based technique from Apache Lucene library [6]. Textual similarity methods are often limited due to vocabulary mismatch problem [28]. We thus also calculate the semantic similarity between any two bug reports using their word embedding vectors and cosine distance measure where the embeddings were learned by FastText algorithm [17] from our corpus. We use the default parameters from both BM25 and FastText algorithms for our similarity calculation. Finally, we rank each of the candidate bug reports by combining their textual and semantic similarities against a non-reproducible bug report. Then the top K bug reports (e.g., K=10) from the ranked list are recommended as potentially connected bug reports.

To evaluate our adapted technique, we construct a corpus of 526 bug reports (274 non-reproducible + 252 connected). The details on corpus creation are discussed in Section 3.1. We apply standard natural language preprocessing (e.g., stop word removal, token splitting, lower casing) to each corpus document (a.k.a., bug report) and then index them using Apache Lucene indexer [6]. We also use three types of queries from each non-reproducible bug report − *title*, *description*, and *title* + *description*. We execute each of these queries using Apache Lucene, collect the top K documents ($1 \leq K \leq 1000$), re-rank them based on their textual and semantic similarities with a search query (a.k.a., non-reproducible bug report), and then compare the ranked results with ground truth for evaluation. We also use a standard set of performance metrics such as Hit@K, Mean Average Precision (MAP), and Mean Reciprocal Rank (MRR) that are frequently used in the relevant literature [58, 71, 62].

Table 9 summarizes the performance details of our adapted IR-based technique. We see that BM25-based textual similarity was able to achieve 64% Hit@5 and 74% Hit@10 with Eclipse system, which are moderately high according to relevant literature [61, 71, 34]. However, the algorithm achieved a 31% Hit@10 with Firefox system, which is comparatively low. Word Embedding (WE)-based semantic similarity also worked well with Eclipse but not with Firefox system. For example, it achieved 77% Hit@10 with 35% MAP and 0.38 MRR in detecting the bug reports connected to non-reproducible bug reports from Eclipse. On the other hand, these metrics were 26%, 10%, and 0.12 respectively for the Firefox system. When both textual and semantic similarities were combined, we also noticed a significant performance increase with Eclipse but not with Firefox. For example,

Table 9: Detection of Bug Reports Connected to Non-Reproducible Bugs

System	Dataset	Technique	Query	Hit@1	Hit@5	Hit@10	MRR	MAP
Eclipse	22	BM25	T	9.09%	63.64%	72.73%	0.36	32.16%
			D	0.00%	63.64%	63.64%	0.30	27.90%
			T+D	0.00%	72.73%	81.82%	0.34	30.63%
		WE	T	22.73%	59.09%	77.27%	0.38	34.80%
			D	0.00%	54.55%	54.55%	0.22	19.84%
			T+D	0.00%	50.00%	63.64%	0.26	22.47%
		BM25+WE	T	22.73%	63.64%	72.73%	**0.42**	**37.32%**
			D	0.00%	59.09%	77.27%	0.25	22.33%
			T+D	0.00%	63.64%	**86.36%**	0.31	26.73%
Firefox	252	BM25	T	1.19%	21.83%	**31.35%**	**0.12**	10.61%
			D	0.00%	10.71%	13.89%	0.06	5.31%
			T+D	0.00%	13.49%	18.65%	0.08	6.51%
		WE	T	3.57%	19.44%	25.79%	0.12	10.07%
			D	0.00%	6.35%	11.51%	0.04	3.64%
			T+D	0.00%	9.52%	12.70%	0.06	4.88%
		BM25+WE	T	2.38%	20.64%	**31.35%**	**0.12**	**10.97%**
			D	0.00%	9.92%	14.68%	0.05	4.71%
			T+D	0.00%	12.30%	18.25%	0.07	5.84%

WE = Word Embedding-based semantic similarity, **T** = Title, **D** = Description, **Emboldened** = The highest performance metric with each system

the combined algorithm (i.e., BM25+WE) achieved a maximum of 86% Hit@10, 37% MAP and 0.42 MRR with Eclipse system, which are promising. Unfortunately, its performance did not improve with Firefox system. We also analyze the effectiveness of three types of queries to detect the connected bug reports against a non-reproducible bug report. As shown in Table 9, *title* was found to be the most effective as a search query for both Eclipse and Firefox systems. However, a combination of *title* and *description* (as a query) also achieved the highest Hit@10 for both BM25 and BM25+WE algorithms with Eclipse, which was interesting.

One might argue about the size of our dataset from Eclipse and Firefox systems and might use it to explain the performance differences above. To address this concern, we randomly sampled 10 cases from Eclipse and 10 cases from Firefox, and manually analyzed them for further insights. We found that the connected bug reports from Eclipse are more similar to their corresponding non-reproducible bug reports than those from Firefox system. In particular, we noticed the presence of similar keywords or the same program elements in their report titles. On the other hand, Firefox was more likely to attach complementary items (e.g., screenshot, commit diffs) to its bug reports. Our adapted technique did not use these complementary items, which might explain the low performance with Firefox above. Inclusion of these complementary items requires more work and possibly sophisticated approaches, which could be a scope for future work.

(b) Manual analysis of the bug reports connected to non-reproducible bugs. In bug tracking systems (e.g., Bugzilla), non-reproducible bug reports are often connected to existing bugs. While our designed technique automatically detects the connected bug reports to non-reproducible bugs, we further manually analyze the connected reports. Our underlying idea was to determine whether these bug reports had the potential to support the reproducibility of non-reproducible bugs. We thus select 71 non-reproducible bugs from our dataset and their 93 connected bugs (i.e., excluding duplicate ones) from the bug tracking history. Our selection process has been described in Section 3.1.

Table 10: Items of Interest from the Connected Bug Reports

Key	Description
isFixed	Indicates whether the connected bugs are fixed or not.
isSameTopic	Indicates whether a non-reproducible bug report and its connected reports discuss the same or similar topics.
containsS2R	Indicates whether any of the connected bug reports contains step to reproduce or not.
containsProgramElement	Indicates whether any of the connected bug reports contains explicit reference to faulty program elements or not.
containsStackTraces	Indicates whether any of the connected bug reports contains stack traces or not.
containsAttachment	Indicates whether any of the connected bug reports contains any attachment (e.g., screenshot, videos, tests, code) or not.
containsFixes	Contains a fix patch.

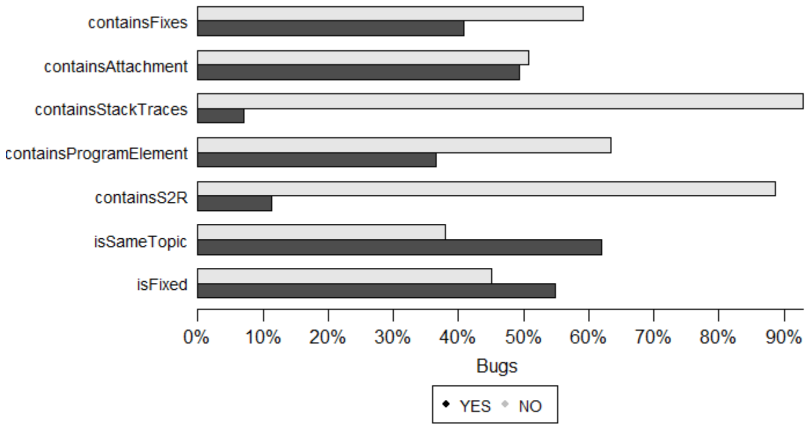


Fig. 13: Presence or absence of structured items in the connected bug reports

We were interested to find out whether the connected bugs could be a source of complementary information for the non-reproducible bugs or not. We thus analyze the connected bug reports and look for meaningful information. Several structured items such as steps to reproduce, stack traces, program elements, and attachments have been reported to contain meaningful, non-trivial information [61, 62, 65]. We thus carefully analyze the 93 bug reports that are connected to 71 non-reproducible bugs, and detect the presence or absence of the structured items in these reports. Table 10 provides a brief overview of all the items of interest from the connected bug reports. We discuss our statistical findings from the analysis as follows.

Fig. 13 shows the presence of various structured items in the bug reports that are connected to our non-reproducible bugs. We see that 44 out of our 71 bugs (59%) are connected to highly similar, existing bugs (i.e., deal with similar problems or topics). More interestingly, 39 of them (55%) are connected to bugs that are fixed. We also further analyze the metadata and found that 33 of these 39 bugs (i.e., 85%) were submitted to the bug tracking system before the submission of non-reproducible bug reports. That means, the discussions from these connected bugs could have been a source of valuable insights for the non-reproducible bugs. We also see that a limited number of connected reports (e.g., 7%–11%) contain stack traces or steps to reproduce their bugs, which aligns with earlier findings from the literature [20, 21]. However, 37% of the connected bug reports contain one or more faulty code elements (e.g., class names, method names) and 49% of them attach various complementary items (e.g., screenshots, tests, videos, memory dumps). Furthermore, we found that the existing bug reports connected to 41% of the non-reproducible bugs contain the fix patches (i.e., solution code) from the bug reporters, which could provide potential clues for reproduction and solution.

We also further analyze the attached items to determine whether they can complement the non-reproducible bug reports. We found that almost 100% of the bug reports that have an attachment contains bug-fix patches (e.g., solution code). We also found code review comments (e.g., Gerrit reviews) for ≈50% of the cases. Furthermore, the attachments contain screenshot, program workflow, and steps to reproduce the bugs. All these items can be a great source of complementary information for improving the non-reproducible bugs.

One can wonder why the non-reproducible bugs above are still non-reproducible despite having their connected bugs. Based on our investigation, we found that 100% of these connected bugs were manually recorded long after the submission or even after the resolution (or labeling) of the non-reproducible bugs (under our investigation). Automatic identification of these connected bugs and their inclusion during the submission of a bug report thus can equip the software developers with more information to reproduce the reported bug. Our IR-based technique is a step towards that direction.

Summary of RQ$_7$: About **20**% (i.e., 71 out of 350) non-reproducible bugs are connected to one or more earlier bugs from the bug tracking system. Our IR-based technique was able to detect the connected bug reports from Eclipse and Firefox with 31%–**86**% accuracy when only top 10 results were considered, which has the potential to encourage more tools and techniques to support bug reproducibility. According to our manual analysis, the connected bugs often contain a wealth of information such as fix patches, screenshots, test cases, stack traces, program elements, and program workflow, which have high potential to complement the non-reproducible bug reports.

5 Threats to Validity

5.1 Threats to Internal Validity

Threats to *internal validity* relate to experimental errors and biases [69]. Our key factors behind bug non-reproducibility were derived from a qualitative study (Section 3.2), which could be a source of subjective bias. However, our identified factors were validated by a group of 13 professional developers with an agreement level of 70%–90% (RQ$_2$). Hence, such a threat might be mitigated. Developers might sometimes use WORKSFORME tag loosely or inconsistently, which might introduce some noise in our dataset [38]. However, since we carefully analyse each of the 576 bug reports and finally summarize our findings, the impacts of such noise might not be significant.

Incorrect classification of bug reports by our trained models could be a threat to our feature analysis (RQ$_6$). However, we selected the best-performing models (e.g., 80.30% precision, 80.30% recall, RandomForest) for our analysis. Furthermore, we observe similar patterns across three trained models (e.g., RandomForest, XGBoost, Logistic Regression), which cross-validates our findings on feature importance. We also repeat our experiments using an extended dataset of 1,990 bug reports and achieve a maximum of 83.60% precision and 83.60% recall in our bug report classification (Table 5). Thus, the impact of model classification errors (e.g., ≈16%) upon our feature analysis might be negligible.

5.2 Threats to External Validity

Threats to *external validity* relate to generalizability of our findings [69]. We analyse 576 bug reports from two open source systems (Firefox and Eclipse), which might not be representatives for the proprietary software systems. However, our findings align with that of an earlier study [38] performed using a different dataset (open source + proprietary) (RQ_3), which possibly indicates the generalizability of our study findings. Furthermore, we used a total of 533 reproducible bug reports (250 from Firefox + 283 from Eclipse) for this study (RQ_7). Thus, we analyzed a total of 1,109 bug reports from two popular software systems to derive our findings reported in this work. Furthermore, we repeated our analysis using a total of 1,990 bug reports and came up with similar findings (Table 5).

5.3 Threats to Conclusion Validity

The observations from our developer study and our conclusions drawn from them could be a source of threat to *conclusion validity* [48]. In particular, there could be a few unseen variables behind the non-reproducibility of bugs (e.g., developer's inexperience, technical infeasibility), which might have been overlooked accidentally. However, we share our dataset [1] publicly for third-party replication and reuse. In the replication package [1], we also include our analysis data of 93 bug reports that are connected to 71 non-reproducible bugs (RQ_7).

6 Related Work

There have been several studies that analyse the characteristics of a good bug report [16] or classify the software bugs from open source systems [56, 42]. Many studies attempt to predict which bugs get fixed [31, 30], re-assigned [32, 66] or re-opened [72]. A few studies also investigate how bugs are coordinated among various stakeholders (e.g., software testers, users, developers) [15, 60] and how the misclassification of bugs affects the bug prediction task [33]. Unfortunately, to date, only little research has been done to better understand what makes the reported bugs non-reproducible or how to improve their reproducibility during the report submission.

Joorabchi et al. [38] first identify six major causes that might explain the non-reproducibility of software bugs (e.g., Inter-bug dependencies, environmental differences). While their work is a source of inspiration, it does not provide actionable insights on how to detect or improve the non-reproducible bugs during their submission. Their findings were also not validated by the developers.

Fan et al. [27] analyse five different dimensions related to software bugs (e.g., bug report texts, reporter's experience, developer-reporter collaborations) and classify *valid* and *invalid* bug reports using machine learning. Although their work is related to ours, all of their adopted features might not be appropriate to characterize the non-reproducible bugs. Furthermore, non-reproducibility of the bugs might always not mean that they are invalid bugs.

Vyas et al. [60] analyse social and human aspects of a bug reproduction process with an ethnographic study. Since their findings focus on human collaboration dynamics, they might also not be enough to properly explain the complex technical aspect of a bug reproduction process.

Unlike many earlier studies above, we conduct an extensive qualitative study with Grounded Theory method [29] using 576 bugs reports from Firefox and Eclipse systems, identify 11 key factors behind bug non-reproducibility, and then validate our major findings with 13 professional developers from the industry (e.g., Mozilla). We not only (1) capture how the professional developers cope with non-reproducible bugs but also (2) offer a list of actionable insights by combining information from multiple analyses (empirical study, developer study), which makes our work novel. Furthermore, we (3) explain the differences between reproducible and non-reproducible bugs using machine learning and a model interpretation framework (e.g., SHAP [40]) (RQ_6), and (4) suggest how to improve the reproducibility of a bug report through an additional manual analysis of linked bugs (RQ_7). Our findings are also generalizable (RQ_3) and the datasets are publicly available for replication and third-party reuse [1].

Recently, several studies [25, 35, 64, 36] have proposed techniques to explain machine learning models, especially designed for software defect prediction, which are relevant to our work. Dam et al. [25] first discuss the necessity of explainable machine learning models in the context of software engineering tasks. They suggest that explainable models are essential to gain the trust of software practitioners. Wattanakriengkrai et al. [64] later propose a line-level defect prediction model that adopts a model-agnostic interpretation framework namely LIME [50]. Jiarpakdee et al. [35] conducted an empirical study where they evaluate three model-agnostic defect prediction techniques. Jiarpakdee et al. [36] recently also conduct a survey where they capture the software practitioners' perceptions about several model interpretation frameworks (e.g., ANOVA, LIME). While these studies are a source of our inspiration, we interpret the models predicting reproducibility of a bug report rather than the defective source code. To the best of our knowledge, no previous studies attempt to interpret the machine learning models predicting bug reproducibility, which indicates the novelty of our work. As a result, our work has the potential to encourage more explainable tools and techniques that can support bug reproducibility.

7 Conclusion and Future Work

Non-reproducibility of software bugs is a major challenge for the developers since it prevents/delays the bug-fixing. Unfortunately, to date, only a little research has been done to understand the non-reproducibility of bugs. In this paper, we conduct an empirical study using 576 non-reproducible bug reports, and identify 11 key factors behind bug non-reproducibility (e.g., bug duplication, bug intermittency, missing information, false-positive bugs). We not only validate our findings using the feedback from 13 professional developers but also investigate how they cope with non-reproducible bugs. Then we provide several actionable insights on how to avoid non-reproducibility and/or improve reproducibility of the reported bugs. We also explain the differences between reproducible and non-reproducible bug reports using machine learning models and their interpretation framework (e.g., SHAP). Using a manual analysis, we demonstrate how the bugs connected to a non-reproducible bug report can offer complementary information (e.g., attachments, screenshots) for the bug reproduction. Finally, we also demonstrate how these connected bug reports can be detected automatically using a traditional approach such as Information Retrieval. By leveraging these insights and findings, future work could focus on developing effective, explainable tools and technologies to assist in the bug reproduction (e.g., sandbox for bug reproduction).

Acknowledgment

This work was supported by Fonds de Recherche du Quebec (FRQ), the Natural Sciences and Engineering Research Council of Canada (NSERC), and Tenure-track startup grant, Faculty of Computer Science, Dalhousie University, Canada. We would also like to thank all the anonymous respondents to the survey.

A Feature Importance from Models Trained with Extended Dataset

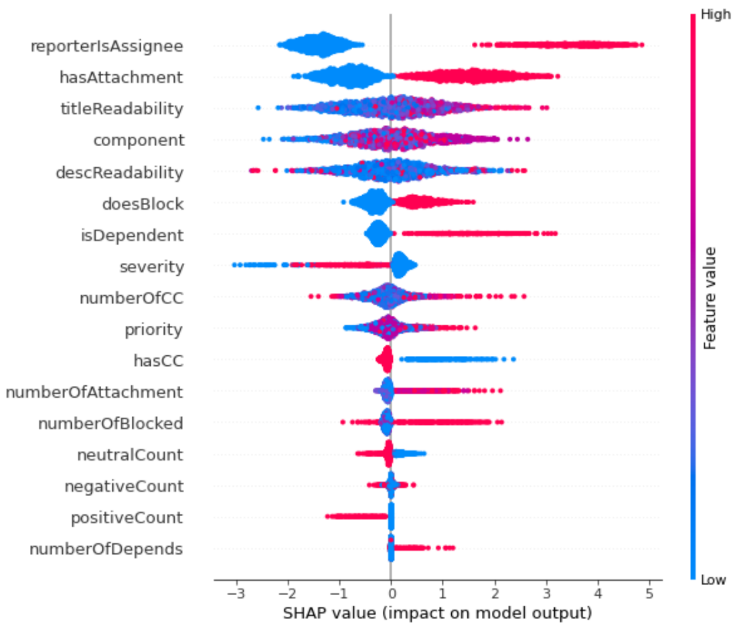


Fig. 14: Feature importance using bee swarm plot (XGBoost model)

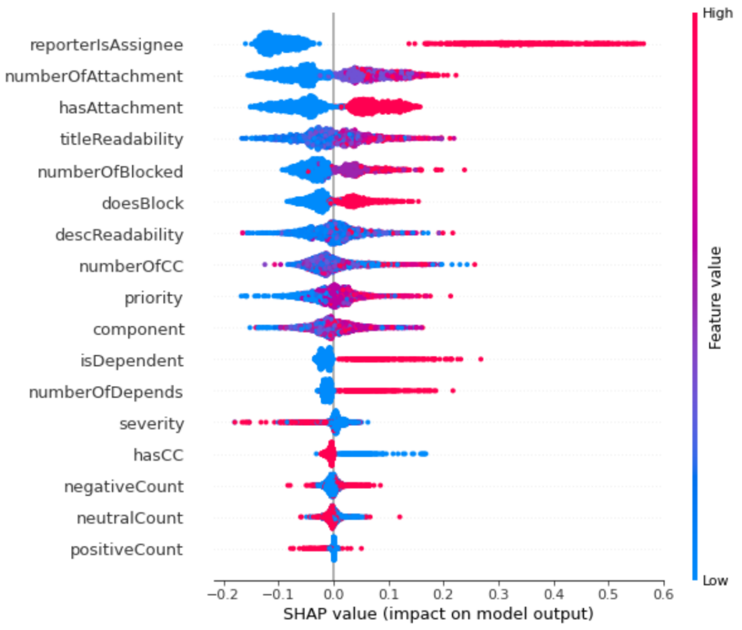


Fig. 15: Feature importance using bee swarm plot (RandomForest model)

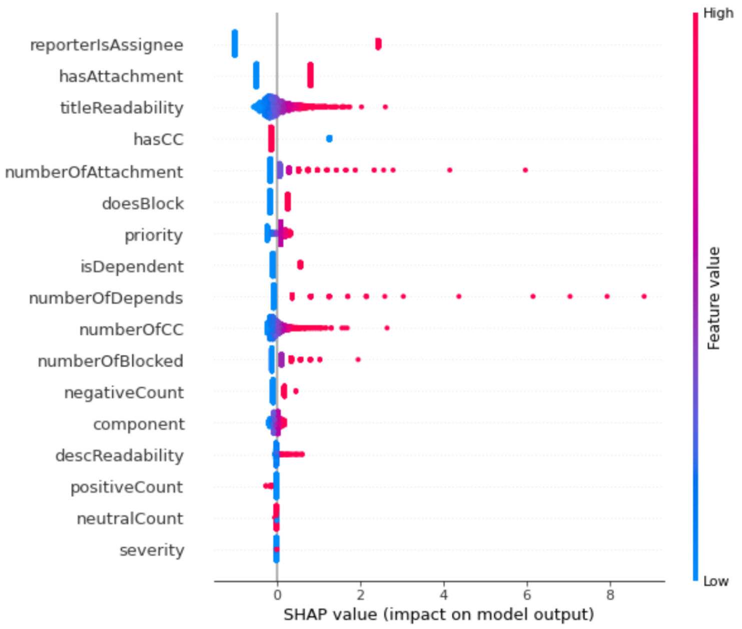


Fig. 16: Feature importance using bee swarm plot (Logistic Regression model)

REFERENCES

1. ICSME 2020 replication package. URL `https://github.com/masud-technope/ICSME2020-Replication-Package`.
2. WEKA Toolkit. URL `http://www.cs.waikato.ac.nz/ml/weka`.
3. Works for me. URL `https://bit.ly/2M94cff`.
4. Researcher posts facebook bug report to mark zuckerberg's wall, 2013. URL `https://cnet.co/2PvIH9O`.
5. Report: Software failure caused $1.7 trillion in financial losses in 2017, 2019. URL `https://tek.io/2FBNl2i`.
6. Apache Lucene Core, 2019. URL `https://lucene.apache.org/core`.
7. Doxygen, 2020. URL `https://www.doxygen.nl/index.html`.
8. Firefox profiler, 2020. URL `https://profiler.firefox.com`.
9. Pernosco, 2020. URL `https://pernos.co/about/overview`.
10. Srcml, 2020. URL `https://www.srcml.org/`.
11. Shapley values, 2021. URL `https://christophm.github.io/interpretable-ml-book/shapley.html`.
12. M. Amoui, N. Kaushik, A. Al-Dabbagh, L. Tahvildari, S. Li, and W. Liu. Search-based duplicate defect detection: An industrial experience. In *Proc. MSR*, pages 173–182, 2013.
13. L. An, M. Castelluccio, and F. Khomh. An empirical study of dll injection bugs in the firefox ecosystem. *EMSE*, 24:1799–1822, 2019.
14. G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y. Guéhéneuc. Is it a bug or an enhancement? a text-based approach to classify change requests. In *Proc. CASCON*, page 15, 2008.
15. J. Aranda and G. Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *Proc. ICSE*, pages 298–308, 2009.
16. N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proc. FSE*, pages 308–318, 2008.
17. P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*, 2016.
18. Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, 2001.

19. S. Le Cessie and J. C. Van Houwelingen. Ridge estimators in logistic regression. *JSTOR*, 41(1):191–201, 1992.
20. O. Chaparro, J. M. Florez, and A Marcus. Using observed behavior to reformulate queries during text retrieval-based bug localization. In *Proc. ICSME*, page to appear, 2017.
21. 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 *Proc. ESEC/FSE*, pages 396–407, 2017.
22. O. Chaparro, C. Bernal-Cárdenas, J. Lu, K. Moran, A. Marcus, M. Di Penta, D. Poshyvanyk, and V. Ng. Assessing the quality of the steps to reproduce in bug reports. In *Proc.ESEC/FSE*, pages 86–96, 2019.
23. O. Chaparro, J. M. Florez, U. Singh, and A. Marcus. Reformulating queries for duplicate bug report detection. In *Proc. SANER*, pages 218–229, 2019.
24. T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *Proc. SIGKDD*, page 785–794, 2016.
25. H. K. Dam, T. Tran, and A. Ghose. Explainable software analytics. In *Proc. ICSE-C*, page 53–56, 2018.
26. M. W. Fagerland. t-tests, non-parametric tests, and large studies–a paradox of statistical practice? *BMC Med Res Methodol*, 12(78), 2012.
27. Y. Fan, X. Xia, D.Lo, and A. E. Hassan. Chaff from the wheat: Characterizing and determining valid bug reports. *TSE*, 2018.
28. G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The Vocabulary Problem in Human-system Communication. *Commun. ACM*, 30(11):964–971, 1987.
29. B. G. Glaser and A. L. Strauss. *The discovery of grounded theory : strategies for qualitative research.* Chicago : Aldine Publishing, 1967.
30. A. Goyal and N. Sardana. Nrfixer: Sentiment based model for predicting the fixability of non-reproducible bugs. *e-Informatica*, 11(1):103–116, 2017.
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 *Proc. ICSE*, pages 495–504, 2010.
32. P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. "not my bug!" and other reasons for software bug report reassignments. In *Proc. CSCW*, pages 395–404, 2011.
33. Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proc. ICSE*, pages 392–401, 2013.
34. A. Hindle and C. Onuczko. Preventing duplicate bug reports by continuously querying bug reports. *Empirical Softw. Engg.*, 24(2):902–936, 2019.
35. J. Jiarpakdee, C. Tantithamthavorn, H. K. Dam, and J. Grundy. An empirical study of model-agnostic techniques for defect prediction models. *TSE*, 2020.
36. J. Jiarpakdee, C. Tantithamthavorn, and J. Grundy. Practitioners' perceptions of the goals and visual explanations of defect prediction models. In *Proc. MSR*, pages 432–443, 2021.
37. G. H. John and P. Langley. Estimating continuous distributions in bayesian classifiers. In *Proc. UAI*, page 338–345, 1995.
38. M. E. Joorabchi, M. Mirzaaghaei, and A. Mesbah. Works for me! characterizing non-reproducible bug reports. In *Proc. MSR*, pages 62–71, 2014.
39. B. Lin, F. Zampetti, G. Bavota, M. Di Penta, M. Lanza, and R. Oliveto. Sentiment analysis for software engineering: How far can we go? In *Proc. ICSE*, pages 94–104, 2018.
40. S. M. Lundberg, G. Erion, H. Chen, A. DeGrave, J. M. Prutkin, B. Nair, R. Katz, J. Himmelfarb, N. Bansal, and S. Lee. From local explanations to global understanding with explainable ai for trees. *Nature machine intelligence*, 2(1):56–67, 2020.
41. W. Maalej and H. Nabil. Bug report, feature request, or simply praise? on automatically classifying app reviews. In *Proc. RE*, pages 116–125, 2015.
42. M. Nayrolles and A. Hamou-Lhadj. Towards a classification of bugs to facilitate software maintainability tasks. In *Proc. SQUADE*, pages 25–32, 2018.
43. R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush. Engineering record and replay for deployability. In *Proc. USENIX*, pages 377–389, 2017.
44. C. Parnin and A. Orso. Are Automated Debugging Techniques Actually Helping Programmers? In *Proc. ISSTA*, pages 199–209, 2011.
45. L. Ponzanelli, A. Mocci, A. Bacchelli, M. Lanza, and D. Fullerton. Improving Low Quality Stack Overflow Post Detection. In *Proc. ICSME*, pages 541–544, 2014.

46. J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers Inc., 1993.
47. M. M. Rahman, C. K. Roy, and J. Collins. CORRECT: Code Reviewer Recommendation Based on Cross-Project and Technology Experience. In *Proc. ICSE*, page to appear, 2016.
48. M. M. Rahman, C. K. Roy, and D. Lo. Automatic query reformulation for code search using crowdsourced knowledge. *EMSE*, 24:1869–1924, 2019.
49. M. M. Rahman, F. Khomh, and M. Castelluccio. Why are some bugs non-reproducible? an empirical investigation using data fusion. In *Proc. ICSME*, page 12, 2020.
50. M. T. Ribeiro, S. Singh, and C. Guestrin. "why should i trust you?": Explaining the predictions of any classifier. In *Proc. KDD*, page 1135–1144, 2016.
51. J. P. Royston. An extension of shapiro and wilk's w test for normality to large samples. *Journal of the Royal Statistical Society*, 31(2):115–124, 1982.
52. A. Sarkar, P. C. Rigby, and B. Bartalos. Improving bug triaging with high confidence predictions at ericsson. In *Proc. ICSME*, pages 81–91, 2019.
53. H. A. Shafiq and Z. Arshad. Automated debugging and bug fixing solutions : A systematic literature review and classification. 2014.
54. Z Shi, J Keung, and Q Song. An Empirical Study of BM25 and BM25F Based Feature Location Techniques. In *Proc. InnoSWDev*, pages 106–114, 2014.
55. R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Ng, and C. Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proc. EMNLP*, pages 1631–1642, 2013.
56. L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai. Bug characteristics in open source software. *EMSE*, 19(6):1665–1705, 2014.
57. M. Thelwall, K. Buckley, G. Paltoglou, D. Cai, and A. Kappas. Sentiment strength detection in short informal text. *JASIST*, 61(12):2544–2558, 2010.
58. P. Thongtanunam, R. G. Kula, N. Yoshida, H. Iida, and K. Matsumoto. Who Should Review my Code ? In *Proc. SANER*, pages 141–150, 2015.
59. Y. Tian, C. Sun, and D. Lo. Improved duplicate bug report identification. In *Proc. CSMR*, page 385–390, 2012.
60. D. Vyas, T. Fritz, and D. Shepherd. Bug reproduction: A collaborative practice within software maintenance activities. In *COOP*, pages 189–207, 2014.
61. S. Wang and D. Lo. Version history, similar report, and structure: Putting them together for improved bug localization. In *Proc. ICPC*, pages 53–63, 2014.
62. S. Wang and D. Lo. Amalgam+: Composing rich information sources for accurate bug localization. *JSEP*, 28(10):921–942, 2016.
63. X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. *Proc. ICSE*, pages 461–470, 2008.
64. S. Wattanakriengkrai, P. Thongtanunam, C. Tantithamthavorn, H. Hata, and K. Matsumoto. Predicting defective lines using a model-agnostic technique. *TSE*, 2020.
65. C. P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *Proc. ICSME*, pages 181–190, 2014.
66. X. Xia, D. Lo, E. Shihab, and X. Wang. Automated bug report field reassignment and refinement prediction. *TSR*, 65(3):1094–1113, 2016.
67. X. Yang, D. Lo, X. Xia, L. Bao, and J. Sun. Combining word embedding with information retrieval to recommend similar bug reports. In *Proc. ISSRE*, pages 127–137, 2016.
68. X Ye, H Shen, X Ma, R Bunescu, and C Liu. From Word Embeddings to Document Similarities for Improved Information Retrieval in Software Engineering. In *Proc. ICSE*, pages 404–415, 2016.
69. T. Yuan, D. Lo, and J. Lawall. Automated Construction of a Software-specific Word Similarity Database. In *Proc. CSMR-WCRE*, pages 44–53, 2014.
70. Y. Zhao, T. Yu, T. Su, Y. Liu, W. Zheng, J. Zhang, and W.G.J. Halfond. Recdroid: Automatically reproducing android application crashes from bug reports. In *Proc. ICSE*, pages 128–139, 2019.
71. J Zhou, H Zhang, and D Lo. Where Should the Bugs Be Fixed? - More Accurate Information Retrieval-based Bug Localization Based on Bug Reports. In *Proc. ICSE*, 2012.
72. T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy. Characterizing and predicting which bugs get reopened. In *Proc. ICSE*, pages 1074–1083, 2012.