This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2929761, IEEE Transactions on Software Engineering

The Impact of Mislabeled Changes by SZZ on Just-in-Time Defect Prediction

Yuanrui Fan, Xin Xia, Daniel Alencar da Costa, David Lo, Ahmed E. Hassan, Shanping Li

Abstract—Just-in-Time (JIT) defect prediction—a technique which aims to predict bugs at change level—has been paid more attention. JIT defect prediction leverages the SZZ approach to identify bug-introducing changes. Recently, researchers found that the performance of SZZ (including its variants) is impacted by a large amount of noise. SZZ may considerably mislabel changes that are used to train a JIT defect prediction model, and thus impact the prediction accuracy.

In this paper, we investigate the impact of the mislabeled changes by different SZZ variants on the performance and interpretation of JIT defect prediction models. We analyze four SZZ variants (i.e., B-SZZ, AG-SZZ, MA-SZZ, and RA-SZZ) that are proposed by prior studies. We build the prediction models using the labeled data by these four SZZ variants. Among the four SZZ variants, RA-SZZ is least likely to generate mislabeled changes, and we construct the testing set by using RA-SZZ. All of the four prediction models are then evaluated on the same testing set. We choose the prediction model built on the labeled data by RA-SZZ as the baseline model, and we compare the performance and metric importance of the models trained using the labeled data by the other three SZZ variants with the baseline model. Through a large-scale empirical study on a total of 126,526 changes from ten Apache open source projects, we find that in terms of various performance measures (AUC, F1-score, G-mean and Recall@20%), the mislabeled changes by B-SZZ and MA-SZZ are not likely to cause a considerable performance reduction, while the mislabeled changes by AG-SZZ cause a statistically significant performance reduction with an average difference of 1%–5%. When considering developers' inspection effort (measured by LOC) in practice, the changes mislabeled B-SZZ and AG-SZZ lead to 9%–10% and 1%–15% more wasted inspection effort, respectively. And the mislabeled changes by B-SZZ lead to significantly more wasted effort. The mislabeled changes by MA-SZZ do not cause considerably more wasted effort. We also find that the top-most important metric for identifying bug-introducing changes (i.e., number of files modified in a change) is robust to the mislabeling noise generated by SZZ. But the second- and third-most important metrics are more likely to be impacted by the mislabeling noise, unless random forest is used as the underlying classifier.

Index Terms—Just-in-Time Defect Prediction, SZZ, Noisy Data, Mining Software Repositories

1 Introduction

To reduce the costs consumed by software defects, a plethora of research has focused on **defect prediction** techniques [25]. In recent years, change-level defect prediction techniques have been paid more attention [22], [35]–[38], [54], [63]. Kamei et al. referred to these techniques as **Just-in-Time (JIT) defect prediction** [37]. In comparison to traditional defect prediction techniques that predict the defect-proneness of files or modules [29], [44], [51], [80], JIT defect prediction has the following advantages: 1) developers can inspect a much smaller number of potential risky lines; 2) defective changes can be inspected when the modification contexts are still fresh in developers' mind [37].

In JIT defect prediction, a key task is to identify the changes that initially introduce bugs, i.e., **bug-introducing changes**, from the historical changes of a project. Modern software projects contain large amounts of historical changes, and it is impractical to manually pinpoint the bug-introducing changes. Hence, researchers proposed the SZZ approach to automatically identify bug-introducing changes [15], [40], [56], [65].

- *Yuanrui Fan and Shanping Li are with the College of Computer Science and Technology, Zhejiang University, Hangzhou, China. E-mail: {yrfan, shan}@zju.edu.cn*
- *Xin Xia is with Faculty of Information Technology, Monash University, Melbourne, Australia. E-mail: xin.xia@monash.edu*
- *Daniel Alencar da Costa is with the Information Science Department, University of Otago, Dunedin, New Zealand. Email: danielcalencar@otago.ac.nz*
- *David Lo is with the School of Information Systems, Singapore Management University, Singapore. E-mail: davidlo@smu.edu.sg*
- *Ahmed E. Hassan is with School of Computing, Queen's University. Email: ahmed@queensu.ca*
- *Xin Xia is the corresponding author.*

The SZZ approach was initially proposed and implemented by Śliwerski, Zimmermann, and Zeller—hence the acronym [65]. To identify bug-introducing changes of a bug, SZZ first looks for the changes whose change log contains the bug identifier. These changes are deemed to fix bugs, which are referred to as **bug-fixing changes**. In the bug-fixing changes, SZZ identifies the lines that are modified so that the bug is removed. These lines are referred to as **buggy lines**. Next, SZZ traces back through the code change history to search for the changes where buggy lines were introduced. Such changes are deemed as potential bug-introducing changes. After obtaining the initial set of potential bug-introducing changes, SZZ filters away the incorrect ones. Finally, the remaining changes are finally deemed as bug-introducing.

Many JIT defect prediction studies employ SZZ to identify bug-introducing changes and label their datasets [22], [35]–[38], [63]. Due to the foundational role of SZZ, researchers have raised concerns about the quality of SZZ-generated data. Prior studies observed that SZZ is affected by a large amount of noise (e.g., changes that only modify code comments or blank lines), which result in mislabeled changes [15], [40], [56]. The mislabeled changes include false positives and false negatives. Notice that in this context, **false positives** refer to changes that do not introduce any bugs but are labeled as bug-introducing, and **false negatives** refer to changes that truly introduce bugs but are labeled as clean.

Prior studies proposed improvements for SZZ to avoid the effect of noise [15], [40], [56]. However, the existing JIT defect prediction studies have used the original SZZ approach as proposed by Śliwerski et al. [65], which may considerably mislabel changes [21], [34], [35], [37], [38], [63]. In our study, we find that compared to the labeled data by the latest SZZ variant, up to 44% of the bug-introducing changes labeled by the original SZZ algorithm are false positives. Training models using such mislabeled data may significantly impact the models [4], [6], [39]. Hence, the mislabeled changes by SZZ may threaten the validity of the observations of prior JIT defect prediction models.

In this study, we set out to investigate the impact of the mislabeled changes by SZZ on JIT defect prediction models. We evaluate JIT defect prediction models that use change metrics proposed by Kamei et al. [37]. Kamei et al. showed that the models built using their metrics are effective in identifying bug-introducing changes [37]. Moreover, their metrics are widely used in prior studies [21], [34], [36], [37], [50], [81]–[83]. We totally analyze four SZZ variants, which are proposed by previous studies and briefly described as follows.

- Śliwerski et al.'s SZZ: the original SZZ [65]. In this paper, we refer to it as **B**asic **SZZ** (**B-SZZ**). B-SZZ considers all lines modified by bug-fixing changes as buggy lines. It uses the built-in `annotate` command in version control systems to trace back through the change history.
- Kim et al.'s SZZ: an SZZ variant that is built on top of B-SZZ [40]. It discards non-semantic lines including blank/comment lines and those involving format modifications (e.g. modifications to code indentation) [40]. This implementation uses the annotation graph—a tool for tracing the evolution of lines of code along the code history as proposed by Zimmermann et al. [85]—to trace back through the change history. We refer to Kim et al.'s SZZ variant as **A**nnotation **G**raph **SZZ** (**AG-SZZ**).
- Da Costa et al.'s SZZ: an SZZ variant that is built on top of AG-SZZ [15]. This variant mitigates the noise caused by branch/merge changes and property changes for AG-SZZ. Da Costa et al. referred to such changes as *meta-changes*. The authors referred to their SZZ variant as **M**eta-change **A**ware **SZZ** (**MA-SZZ**).
- Neto et al.'s SZZ: an SZZ variant that is built on top of MA-SZZ [56]. It detects refactoring modifications in changes and mitigates the noise induced by such modifications. Neto et al. referred to their SZZ variant as **R**efactoring **A**ware **SZZ** (**RA-SZZ**).

We first label our datasets leveraging the four SZZ variants. Then, we leverage the labeled data by each SZZ variant machine learning techniques including random forest [8], logistic regression [32] and naive Bayes [17] to learn models. RA-SZZ has the most filters to deal with noise compared to the other SZZ variants. Da Costa et al.'s analysis on the evaluation of different SZZ variants [15] and our manual analysis on the detected refactorings of RA-SZZ indicate that RA-SZZ generates the cleanest data compared to the other SZZ variants. We construct the testing set using the labeled data by RA-SZZ. Next, we evaluate the models learned from the labeled data by studied SZZ variants on the testing set. We use AUC (Area Under the Curve) [33], F1-score and G-mean [68] as performance measures. We use the model trained using the labeled data by RA-SZZ as a baseline model, since it is built on the cleanest data. We compare the models trained using the labeled data by B-SZZ, AG-SZZ and MA-SZZ with the baseline model. By doing so, we analyze the impact of the mislabeled changes by B-SZZ, AG-SZZ, MA-SZZ on the performance of JIT models. And we also investigate the impact of the mislabeled changes by SZZ when JIT models are applied to predict bug-introducing changes. We compare the number of incorrectly predicted changes and wasted inspection effort (measured by LOC) made by the models trained using the labeled data by each SZZ variant. Furthermore, we calculate and compare the most important metrics in identifying bug-introducing changes for the models.

In addition to predicting bug-introducing changes, prior studies have proposed JIT defect prediction techniques to *prioritize* changes for developers by considering the limited resources and human effort to inspect changes [21], [34], [83]. These techniques are referred to as **effort-aware JIT defect prediction techniques**. We also investigate the impact of mislabeled changes by SZZ on performance of effort-aware JIT defect prediction models. We leverage two techniques to build the effort-aware models: Huang et al.'s CBS [34] and Fu et al.'s OneWay [21]. We use Recall@20% to evaluate effort-aware models and compare the performance of the models.

Notice that in this study, as shorthand notations, we denote the four models as B, AG, MA and RA models, respectively

We conduct a large-scale empirical study on ten projects containing a total of 126,526 code changes. Experimental results show that the mislabeled changes by B-SZZ and MA-SZZ do not cause a significant performance reduction for JIT models in identifying bug-introducing changes. The mislabeled changes by AG-SZZ cause a significant performance reduction with an average difference of 1%–5% in terms of AUC, F1-score and G-mean. When considering developers' effort to inspect the bug-introducing changes that are predicted by JIT models, the mislabeled changes by B-SZZ and AG-SZZ lead to 9%–10% and 1%–15% more wasted effort on false positives. The mislabeled changes by B-SZZ significantly increase developers' wasted effort. In terms of model interpretation, the mislabeled changes by SZZ do not impact the top-most important metric for identifying bug-introducing changes (i.e., number of files in a change). But the mislabeled changes can impact the second- and third-most important metrics for identifying bug-introducing changes, unless random forest is used as the underlying classifier. For the effort-aware JIT models, experimental results show that the mislabeled changes by B-SZZ and MA-SZZ do not cause performance reduction, while the mislabeled changes by AG-SZZ cause a significant performance reduction in terms of Recall@20% with an average difference of 2% when the CBS technique [34] is used.

We analyze the reason as to why the B model wastes considerably more effort than the RA model even if the prediction performance of the B model is not lower than the RA model (See Section 4.3). We find that B-SZZ tends to mislabel larger changes as bug-introducing and smaller changes as clean compared to RA-SZZ, and thus, the B model is more likely to predict large changes as bug-introducing. In such a case, developers may waste more effort on the changes that are incorrectly predicted as bug-introducing by the B model. Moreover, we discuss the reasons why the B and MA models achieve a similar performance compared to the RA model and why the AG model shows a significant performance reduction (see Section 5.1). We find that the labeled data by B-SZZ and MA-SZZ has a relatively low false positive rate and false negative rate, which may not considerably impact the prediction of JIT models. On the other hand, the labeled data by AG-SZZ contains a considerable number of false negatives (the average false negative rate is 30%), which are likely to induce bias to the prediction of JIT models.

To the best of our knowledge, this paper is the first to systematically investigate the impact of mislabeled changes by different SZZ variants on Just-in-Time defect prediction models. We conducted a large-scale empirical study on a total of 126,526 code changes that are collected from ten Apache open source projects. We note the following findings along with their implications:

- Although mislabeled changes by the original SZZ algorithm (i.e., B-SZZ) do not cause a considerable performance reduction, the mislabeled changes lead to significantly more wasted inspection effort for the practical usage of JIT models. Furthermore, the mislabeled changes may impact the interpretation of JIT models. Hence, prior JIT defect prediction studies that applied B-SZZ may need to be revisited to determine whether considerable amount of inspection effort was wasted in practice and whether the interpretation of their JIT models is impacted (e.g., Kamei et al. [37]).
- The mislabeled changes by AG-SZZ lead to a significant performance reduction, and they cause more wasted inspection effort. Also, the mislabeled changes impact the interpretation of JIT models. Hence, AG-SZZ should be avoided in future studies. The mislabeled changes by MA-SZZ do not cause a significant performance reduction nor do they considerably waste more effort. RA-SZZ may be difficult to use when refactoring detection tools are not available (e.g., projects are written in programming languages other than Java). In such a case, MA-SZZ may be an alternative for RA-SZZ to label data. To combat the impact of the mislabeled changes by MA-SZZ on the interpretation of JIT models, practitioners can use a random forest classifier as the underlying classifier for their JIT models.

The datasets and code for reproducing our study are available from our accompanying GitHub repository[1].

1. https://github.com/YuanruiZJU/SZZ-TSE

Paper organization. The remainder of this paper is organized as follows. Section 2 provides background concepts and briefly reviews the related work. Section 3 describes our experimental setup (e.g., the studied projects). Section 4 presents the results with respect to five research questions, while Section 5 further discusses our experimental results and discloses the threats to the validity of our study. Finally, Section 6 concludes our paper.

2 Background and Related Work

In this section, we describe the concepts that are necessary to understand our study. We also highlight the studies that are most closely related to our work.

2.1 Just-in-Time Defect Prediction

Traditional defect prediction techniques focus on predicting defect-prone software entities at a coarse-grained level, e.g., predicting defect-proneness of files or modules. Such granularity challenges the practical usage of these techniques. For example, a prediction model is likely to predict large files containing thousands of lines as defect-prone [42]. However, it is difficult for developers to inspect thousands of lines of code to find and fix potential bugs. Furthermore, a predicted defect-prone file may be modified by hundreds of developers. Hence, it is also difficult to find an expert who should be assigned to fix potential bugs in such a file [23], [38].

To tackle the above-mentioned challenges, prior studies proposed the change-level defect prediction, i.e., predicting defect-proneness for software changes, which are more fine-grained than files or modules [54]. Mockus and Miss proposed the change-level defect prediction [54]. They leveraged a prediction model to predict defects for initial maintenance requests (IMRs). An IMR consists of multiple changes. Failures of IMRs are recorded in a tracking system, thus labeling training data is easy.

Śliwerski et al. proposed the SZZ approach, which can identify bug-introducing changes based on data stored in an issue tracking system (ITS, e.g., JIRA) and version control system (VCS, e.g., Git) of a software project [65]. Due to the advent of SZZ, practitioners have the opportunity to identify the individual changes that introduce bugs. Using SZZ, they can label each historical change of a project as bug-introducing or clean. Hence, in recent studies, more researchers have been focusing on techniques that predict the defect-proneness of individual changes [35]–[38], [50]. Kamei et al. referred to these techniques as Just-in-Time defect prediction techniques since they can predict the defect-proneness of a change when it is initially submitted [37]. Figure 1 presents the general framework of Just-in-Time (JIT) defect prediction which consists of four steps. First, JIT defect prediction leverages the SZZ approach to label historical changes as stored in VCS as bug-introducing or clean. Then, it extracts change metrics to characterize the code changes. Next, it leverages machine learning techniques (such as random forest) to learn a model based on the labels and change metrics. Finally, JIT defect prediction leverages the learned model to predict the defect-proneness of new code changes.

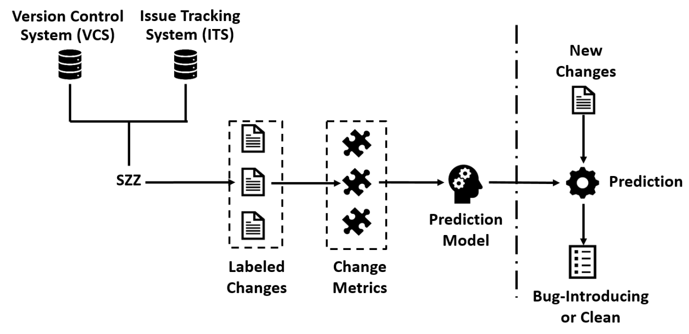


Fig. 1: General framework of Just-in-Time (JIT) defect prediction

Many prior studies have focused on JIT defect prediction techniques. And in these studies, SZZ plays a foundational role. As shown in Figure 1, it generates the labels of the studied datasets for training models. Kim et al. are the first to predict the defect-proneness for individual changes [38]. Kamei et al. conducted a large-scale empirical study on JIT defect prediction using both open-source and commercial projects [37]. Jiang et al. proposed to predict defect-proneness of changes in a personalized way [35]. Kamei et al. investigated the performance of JIT defect prediction models in several cross-project settings [36]. All of the above studies used B-SZZ (i.e., the original SZZ approach) to label changes. Prior studies noted that this variant is affected by a large amount of noise [15], [40], [56]. The noise results in mislabeled changes, which may threaten the validity of these studies [35]–[38].

Recently, McIntosh et al. conducted a longitudinal case study of JIT defect prediction [50]. To prepare their studied datasets, they first used B-SZZ to generate labels. Then, they removed false positives, e.g., the potential bug-introducing changes that only modify code comments or whitespace and those that contain too much churn and too many files. However, the method used by McIntosh et al. does not address the false negatives generated by SZZ. Note that Kim et al. observed that B-SZZ may generate false negatives since changes of code indentation can hinder B-SZZ from identifying the real bug-introducing changes [40]. Thus, mislabeled changes by SZZ may still threaten the validity of McIntosh et al.'s study. Notice that although McIntosh et al. made an effort to deal with the noise generated by SZZ (e.g., removing false positives to clean their datasets), the authors did not investigate whether the removed noise impact JIT models.

In practice, due to limited resources and human effort, developers can only inspect a limited number of lines of code. Prior studies proposed effort-aware JIT defect prediction techniques to prioritize changes for developers to find more bugs while exerting the same effort [21], [34], [37], [83].

Kamei et al. proposed a supervised effort-aware JIT defect prediction technique EALR [37]. EALR predicts the defect density of changes and prioritizes changes using the predicted defect density in descending order. Yang et al. proposed an unsupervised effort-ware JIT defect prediction technique [83]. Their models first sort changes using one of Kamei et al.'s metrics [37], and then, prioritize changes for developers in that order. Yang et al. showed that their unsupervised models achieve better cost-effectiveness than EALR. Huang et al. proposed a supervised effort-aware JIT defect prediction technique CBS [34]. CBS first predicts defect-proneness of changes. Then, for changes that are predicted as bug-introducing, CBS prioritizes them for developers by their chunk size. Fu et al. proposed a supervised effort-aware JIT defect prediction technique OneWay [21]. OneWay first runs Yang et al.'s unsupervised models on the training data and chooses the model that achieves the best cost-effectiveness. Then it uses the model to prioritize changes in the testing data. CBS and OneWay can achieve better cost-effectiveness than Yang et al.'s unsupervised models. These studies have applied the Kamei et al.'s data [37] labeled by B-SZZ.

Mislabeled changes by SZZ cannot impact Yang et al.'s unsupervised models, since these models do not require the labels of changes. However, for studies based on supervised models which require change label(e.g., Huang et al. [34]), mislabeled changes by SZZ may threaten the validity of these studies.

To the best of our knowledge, apart from using the original SZZ (i.e., B-SZZ), no prior studies have investigated the impact of adopting the newly proposed SZZ variants (i.e., AG-SZZ, MA-SZZ and RA-SZZ) to build JIT defect prediction models. Due to the foundational role of SZZ in JIT defect prediction, it is necessary to analyze the impact of noise that is introduced by SZZ on JIT defect prediction models. We are the first to investigate this problem systematically.

2.2 The SZZ Approach

2.2.1 The Original SZZ Approach

The original SZZ approach (B-SZZ) was proposed in Śliwerski et al.'s study [65]. For a given bug, Śliwerski et al. proposed four steps to identify the bug-introducing changes [65]. In Figure 2, we show a bug from ActiveMQ (bug AMQ-1381 [1]) to illustrate the four proposed steps:

(1) Identify bug-fixing changes. SZZ begins with searching for bug-fixing changes. Many projects adopt an issue tracking system (ITS) to store bug reports. Each bug has an identifier. In JIRA, the bug identifier has the format PROJECT-ID (AMQ-1381 in our example). Developers typically record the bug identifier in the commit message of a change to indicate that the change fixes the bug [7]. To identify changes that are likely to be bug fixes, Śliwerski et al.'s SZZ checks whether the change log contains the bug identifier as recorded in ITS [65]. As a result, SZZ identifies that Change #645599 is the change fixing the bug AMQ-1381.

(2) Identify buggy lines. SZZ leverages the `diff` command provided by the VCS to identify the lines of code modified by the bug-fixing changes. These lines of code are considered as buggy lines. In the example shown in Figure 2, SZZ identifies a buggy line that involves a function declaration, where the type of the `command` parameter is incorrect, i.e., its type should be `Object` rather than `Command`. The description of the bug AMQ-1381 also indicates that the function cannot be called because of the incorrect parameter type [1]. Hence, this line is truly buggy.

(3) Identify potential bug-introducing changes. SZZ traces back through the code history to identify the changes that

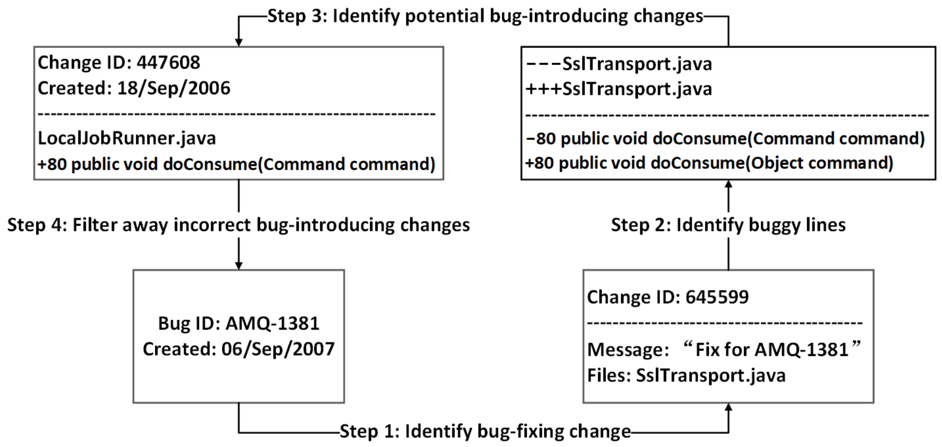

Step 3: Identify potential bug-introducing changes
Change ID: 447608
Created: 18/Sep/2006
LocalJobRunner.java
+80 public void doConsume(Command command)
---SslTransport.java
+++SslTransport.java
−80 public void doConsume(Command command)
+80 public void doConsume(Object command)
Step 4: Filter away incorrect bug-introducing changes
Step 2: Identify buggy lines
Bug ID: AMQ-1381
Created: 06/Sep/2007
Change ID: 645599
Message: "Fix for AMQ-1381"
Files: SslTransport.java
Step 1: Identify bug-fixing change

Fig. 2: Illustrative example for the SZZ approach.

introduce one or more of the buggy lines. We refer to these changes as **potential bug-introducing changes**. Śliwerski et al. proposed to use the built-in `annotate` command of the VCS to trace back through the code history [65]. As shown in Figure 2, SZZ identifies that Change #447608 introduces the incorrect function declaration as found in Step 2.

(4) Filter away incorrect bug-introducing changes. SZZ filters away incorrect ones from the potential bug-introducing changes. Śliwerski et al. proposed that potential bug-introducing changes which were created after the bug report date are incorrect and should be filtered away [65]. And the remaining potential bug-introducing changes are eventually identified as changes that introduce the bug. Since Change #447608 is created before the date AMQ-1381 was reported, it is identified as a bug-introducing change.

2.2.2 Limitations and Improvements of the Original SZZ

Researchers noticed that SZZ is affected by a large amount of noise when SZZ performs Step 2 (identify buggy lines) and Step 3 (identify potential bug-introducing changes) [15], [40], [56]. As a result, considerable changes may be mislabeled. To deal with the noise, prior studies proposed improvements for SZZ to perform the two steps more accurately.

Kim et al. observed that B-SZZ may consider non-semantic lines including blank/comment lines and those involving format modifications (e.g., modifications to the code indentation) to be buggy lines [40]. These lines of code should not be considered to introduce bugs since they do not affect the behavior of the code, i.e., these lines should be ignored in Step 2 of SZZ. Moreover, format modifications pose another challenge: in Step 3 of SZZ, format modifications to the buggy lines (as identified in Step 2) may appear when SZZ traces back through the code history. In such a case, SZZ stops at the format modification changes, and the format modification changes are identified to introduce the bug. As a result, real bug-introducing changes cannot be identified. Hence, format modifications may cause false positives and false negatives. We use three code changes as shown in Figure 3 to illustrate the false positives and false negatives caused by format modifications. In Figure 3, Change #170 fixes a bug by

Change #110 (the real bug-introducing change)
```
if (x == null) return x;
```

Change #152 (format modification)
```
if (x == null)
    return x;
```

Change #170 (bug-fix)
```
if (x != null)
    return x;
```

Fig. 3: False positive and false negative example caused by a format modification.

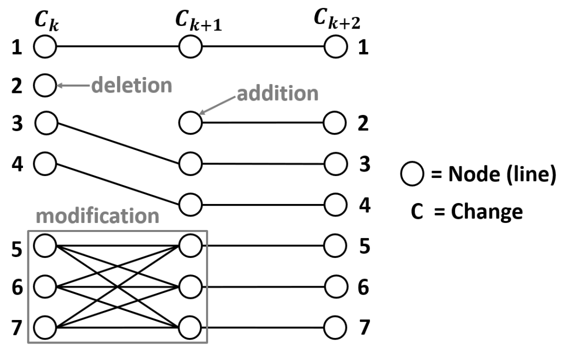


Fig. 4: Illustrative example of an annotation graph

changing the buggy line `if(x == null)` to `if(x != null)`. When searching for the potential bug-introducing changes, B-SZZ incorrectly stops at Change #152, which only modifies the code indentation. As a result, Change #152 is mislabeled as bug-introducing (i.e., false positive) and Change #110 is mislabeled as clean (i.e., false negative). To deal with the challenge introduced by format modifications, Kim et al. proposed AG-SZZ, which uses the *annotation graph* [85] rather than the `annotate` command in Step 3. The annotation graph provides more comprehensive information about line moves and modifications within a file than the `annotate` command.

Figure 4 presents an illustrative example of an annotation graph which traces the evolution of the lines across three consecutive changes: C_k, C_{k+1} and C_{k+2}. In the graph, each node denotes a line. The annotation graph uses edges to map the relationships between the lines of two changes. For example, from C_k to C_{k+1}, one line of code is deleted, and one line of code is added. If several lines are modified, the VCS understands this case as deleted and added, and the annotation graph will use edges to connect each added line of C_{k+1} with each deleted line of C_k. When tracing back the code history, annotation graph performs a *depth-first search* method to find the bug-introducing changes. Furthermore, Williams and Spacco proposed an enhancement to the annotation graph [77], [78]. In their work, they used weights to map the evolution of a line. For example, if a change modifies a `while` statement to a different `while` statement and two new lines of code, the annotation graph would indicate that the previous `while` statement is changed into these three lines. On the other hand, if the change updates the `while` statement, the enhanced algorithm would put a heavier weight on the edge between the previous and updated `while` statements. Such a heavier weight indicates that the previous `while` statement is more likely to be updated rather than to be replaced with a new line of code.

Da Costa et al. noticed that AG-SZZ flags meta-changes as potential bug-introducing changes due to the limitation of the annotation graph [15]. Meta-changes refer to changes that are not related to code modification, and hence they are considered as noise. Da Costa et al. observed three types of meta-changes: branch changes (i.e., changes copying code from one branch to a new branch), merge changes (i.e., changes applying code modifications from one branch to another) and property changes (i.e., changes impacting file properties) [15]. Da Costa et al. found that meta-changes are incorrectly flagged because the annotation graph cannot map the lines of code between meta-changes and their prior changes [15]. In such a case, AG-SZZ may stop at meta-changes in Step 3. Note that since B-SZZ uses a different strategy (i.e., the `annotate` command) to perform Step 3, B-SZZ is not affected by branch/merge changes. Hence, in comparison with B-SZZ, AG-SZZ may generate more false negatives due to the limitation of the annotation graph. To mitigate the impact of meta-changes for AG-SZZ, Da Costa et al. proposed MA-SZZ (Meta-change Aware SZZ) on top of AG-SZZ, in which they enhanced Kim et al.'s annotation graph by connecting all of the nodes in a given meta-change with its prior change. This improvement avoids identifying meta-changes as potential bug-introducing changes.

Neto et al. observed that prior SZZ variants (including B-SZZ, AG-SZZ and MA-SZZ) may identify incorrect bug-introducing changes due to the impact of refactoring lines [56]. Refactoring lines refer to those involving refactoring modifications (e.g., modification to a function name). These lines impact Step 2 and Step 3 of SZZ. In Step 2, since refactoring modifications are not likely to fix bugs, refactoring lines should not be considered as buggy. In Step 3, when tracing back through the code history to identify the changes that introduce buggy lines, SZZ may stop at a change in which buggy lines are refactored. In fact, the bug had been introduced before the refactoring modifications were performed. Hence, SZZ is hindered from identifying the real bug-introducing changes. To deal with the impact of refactoring lines, Neto et al. proposed Refactoring Aware SZZ (RA-SZZ), where they incorporated RefDiff into MA-SZZ. RefDiff is a refactoring-detection tool for Java code [64]. In Step 2 and Step 3, RA-SZZ uses the tool to detect refactoring lines. In Step 2, refactoring lines are not considered as buggy lines. In Step 3, RA-SZZ does not stop at changes involving refactoring lines and it will further trace back through the code history to identify the real bug-introducing changes.

Note that in this study, we use the same datasets that were studied by Da Costa et al. [15] and Neto et al. [56]. From the datasets, Da Costa et al and Neto et al. have provided many concrete examples of mislabeled changes by different SZZ variants, which are helpful for understanding the mislabeling of the studied SZZ variants.

Prior studies evaluated the generated data by different SZZ variants [15], [16], [40], [56].

To evaluate B-SZZ, Davies et al. manually identified bug-introducing changes of 174 bugs [16]. Then they compared the manually identified changes with the bug-introducing changes identified by B-SZZ. They found that the labeled data by B-SZZ contains many false positives and false negatives.

To evaluate AG-SZZ, Kim et al. compared the set of bug-introducing changes identified by AG-SZZ (S_1) with the set of bug-introducing changes identified by B-SZZ (S_2). They assumed that AG-SZZ is more accurate than B-SZZ. Then they calculated the false positives ($\frac{|S_2-S_1|}{|S_2|}$) and false negatives ($\frac{|S_1-S_2|}{|S_1|}$). However, Da Costa et al. analyzed the bug-introducing changes identified by AG-SZZ and observed that AG-SZZ flags meta-changes (including branch/merge changes and property changes) as bug-introducing for 90%–98% of studied bugs due to limitations of the annotation graph [15]. They also analyzed the generated data by B-SZZ and found that B-SZZ does not flag branch/merge changes as bug-introducing, but it flags property changes as bug-introducing for 0%–48% of the studied bugs. Hence, AG-SZZ may not be more accurate than B-SZZ.

MA-SZZ deals with the meta-changes on top of AG-SZZ. Meta-changes can be correctly recognized using information of the version control system. Thus, MA-SZZ is not likely to introduce new noise when dealing with meta-changes, i.e., MA-SZZ is more accurate than AG-SZZ. Furthermore, Da Costa et al. proposed an evaluation framework for the generated data by SZZ [15]. The framework uses three metrics: 1) count of future bugs introduced by a change; 2) time span of future bugs introduced by a change; 3) time span of bug-introducing changes of a bug. Da Costa et al. noted that the SZZ-generated data with lower values of the three metrics is more likely to be reliable. They leveraged the framework to evaluate the generated data by B-SZZ and MA-SZZ. They found that the generated data by MA-SZZ is more reliable than B-SZZ.

RA-SZZ is built on top of MA-SZZ and further deals with the refactoring modifications detected by RefDiff. Neto et al. assumed that RefDiff can achieve high precision in detecting refactoring modifications [56]. Based on the assumption, they found that RA-SZZ reduces 20.8% of lines that involved refactoring modifications but were identified as buggy lines by MA-SZZ, and they concluded that RA-SZZ can create cleaner data than MA-SZZ. However, they did not analyze the reliability of this assumption. In this study, we perform a manual evaluation on 1,000 randomly sampled refactoring changes as detected by RefDiff (See Section 3.2). Our manual analysis verifies Neto et al.'s assumption that RefDiff achieves high precision—RefDiff achieves a precision of 98.1% on the sampled data. Therefore, RA-SZZ generates cleaner data than MA-SZZ, i.e., RA-SZZ generates the cleanest data compared to the other SZZ variants. In this paper, we use the labels generated by RA-SZZ to create the testing set.

2.3 Impact of Data Quality on Defect Prediction Models

In recent years, researchers have raised concerns about the quality of data employed by defect prediction models including file-level and change-level defect prediction models.

Several studies investigated the impact of the missing link issue between bugs and bug-fixing changes on defect prediction models [4], [6]. Bird et al. noted that developers may not explicitly record which commits correspond to bug-fixes, and the linked bug repository exhibits bias with respect to the severity of bug and reporter experience [6]. Bachmann et al. further found that the missing link issue causes bias in the recorded bug-fix datasets [4]. Bird et al. and Bachmann et al. evaluated the bias caused by the missing link issue on file-level defect prediction models and concluded that the bias can severely impact the models. Note that Bird et al.'s study and Bachmann et al.'s study are both based on the data in Bugzilla ITS. In our study, all our studied projects use the JIRA issue tracking system. Bissyande et al. analyzed the data in the JIRA ITS and observed that the quality of links in the JIRA ITS is much better than that of Bugzilla ITS [7].

Kim et al. conducted an empirical study on the impact of noise on both file-level and change-level defect prediction models [39]. In their study, they intentionally and randomly injected false positives and false negatives into their datasets. And they found that performance of the defect prediction models significantly decreases when 25%–35% changes of their dataset are mislabeled.

Prior studies also noticed that bug reports may be mislabeled, and performed analysis on the impact of mislabeled bug reports on defect prediction models [3], [30], [72]. Antoniol et al. and Kochhar et al. noted issue report mislabeling—reports labeled as bugs but actually refer to non-bug issues [3], [41]. Herzig et al. manually inspected more than 7,000 bug reports [30]. They considered that bug reports that do not result in bug fixes are incorrectly labeled. And they found that a large proportion of bug reports are mislabeled. Tantithamthavorn et al. leveraged Herzig et al.'s manually curated dataset to investigate the impact of the mislabeled bug reports on file-level defect prediction models [72]. They found that mislabeling does not significantly impact precision of the models, but significantly impacts recall of the models.

Herzig et al. noted that developers may perform multiple tasks (such as bug fix and refactoring) in a single change, and they referred to these changes as *tangled code changes* [31]. They found that tangled changes only affect 1.5%–7.1% of source files which were originally labeled as defective—these files should be labeled as clean. Since only a small fraction of the files that were originally labeled as defective are affected, the noise induced by tangled changes does not significantly impact the models which predict whether source files have bugs or not. On the other hand, Herzig et al. observed that tangled changes impact the *number* of associated bugs for 16.6% of all source files. Since such a considerable number of files are affected with respect to the *number* of associated bugs, the noise induced by tangled changes significantly impacts the models which predict the *number* of bugs in source files.

Our study is orthogonal to the above studies. First, we focus our analysis on the impact of noise on Just-in-Time (i.e., change-level) defect prediction models. Second, we focus on noisy labels generated by SZZ. The noisy labels are not randomly generated. Also, they are not caused by developers' imperfect operations. Instead, they are generated due to the inherent limitations of different SZZ variants.

3 Experiment Settings

In this section, we first introduce our studied projects. Then, we elaborate the four SZZ variants and describe the labeled data by the SZZ variants. Next, we describe the studied changes and our data preprocessing methods. After that, we elaborate the details of our experimental setup, e.g., model building and metric calculation. Finally, we introduce the performance measures used in this study. We would like to answer five research questions in this paper:

RQ1: How do mislabeled changes by SZZ impact the performance of JIT defect prediction models trained using the original data?

RQ2: How do mislabeled changes by SZZ impact the performance of JIT defect prediction models trained using re-balanced data?

RQ3: How do mislabeled changes by SZZ impact the practical usage of JIT defect prediction models?

RQ4: How do mislabeled changes by SZZ impact on the performance of effort-aware JIT defect prediction models?

RQ5: How do mislabeled changes by SZZ impact the interpretation of JIT defect prediction models?

3.1 Studied Projects

In this study, we conduct our analysis on the ten Apache projects that were used in Da Costa et al.'s study [15] and Neto et al.'s study [56]. Table 1 shows the description of our studied projects. We collect historical data of the ten projects from August 2003 to September 2013. All the data is retrieved from the Apache Git repository[2].

All of the projects use JIRA ITS to manage bug reports. JIRA provides add-ons for connecting issues to version control systems, and thus it is easier for developers using this ITS to link bug reports to changes [7]. Bissyande et al. observed that the quality of links in JIRA is much better than that of Bugzilla ITS [7]. Hence, our experimental results are not likely to be impacted by the bias induced by the

2. http://git.apache.org/

TABLE 1: Overview of Studied Projects

Project	Description	#Changes
ActiveMQ	A message broker with a Java message service client.	7,753
Camel	A versatile integration framework for a variety of domain-specific languages.	17,374
Derby	A relational database management system (RDBMS) written in Java	9,775
Geronimo	An application server fully certified by Java Enterprise Edition 6 (Java EE 6).	16,152
Hadoop Common	A software for distributed processing of large data sets using clusters of computers.	27,077
HBase	A distributed and scalable database adopted by Hadoop for big-data store.	14,581
Mahout	A framework for implementing distributed or scalable machine learning algorithms.	2,762
OpenJPA	Java persistence library using a object-relation mapping (ORM) solution.	6,368
Pig	A high-level platform for creating programs using Hadoop to analyze large data sets.	3,089
Tuscany	A software that provides an infrastructure for developing and managing service-oriented architecture.	21,595
Total		126,526

missing link problem between bug reports and code changes as noted by Bird et al. [6] and Bachmann et al. [4] in the Bugzilla ITS.

As noted by Wang et al. [75] and McIntosh et al. [50], large changes are likely to be caused by routine maintenance (e.g., copyright updates). Hence, such changes are unlikely to introduce bugs. Moreover, in practice, developers are unlikely to take great efforts to review large changes. Hence, in this study, we discard large changes from our analysis. We consider changes that modify more than 10,000 lines or 100 files as large changes following McIntosh et al.'s study [50]. Table 1 shows the number of changes of each studied project.

3.2 Data Labeling

In this study, we analyze four SZZ variants, namely B-SZZ, AG-SZZ, MA-SZZ and RA-SZZ. Then we use the four SZZ variants to label our data. Notice that in our study, we use the implementation of B-SZZ, AG-SZZ and MA-SZZ provided by Da Costa et al. [15] and the implementation of RA-SZZ provided by Neto et al. [56]. The implementation of the four SZZ variants is available on GitHub [13], [14].

To identify bug-introducing changes, all of the four SZZ variants need to perform the four steps as shown in Figure 2. All of them use the same method to perform Step 1 (i.e., identify bug-fixing changes) and Step 4 (i.e., filter away incorrect bug-introducing changes). But they apply different strategies when performing Step 2 (i.e., identify buggy lines) and Step 3 (i.e., identify potential bug-introducing changes).

When identifying bug-fixing changes, all of the four SZZ variants take the following two steps: (1) identify occurrences of bug identifiers with the format PROJECT-ID (e.g., HBASE-2975) in the commit log; (2) check if the corresponding bug report with the given identifier is defined as a defect in ITS. When filtering incorrect bug-introducing changes for each bug, all of four SZZ variants apply the strategy proposed by Śliwerski et al. [65], which is: filter away the potential bug-introducing changes that are created after the bug report date.

Table 2 presents the mapping and searching strategies that are applied in Step 2 and 3 of the variants, respectively. The mapping strategies are applied in Step 2 of SZZ to identify the real buggy lines from the lines that are modified by bug-fixing changes. In Step 3 of SZZ, the searching strategies are applied when tracing back through the change history to search for potential bug-introducing changes. For instance, when identifying buggy lines (Step 2), RA-SZZ filters away non-semantic lines and those involving refactoring modifications from the lines modified by bug-fixing changes and only use the remaining lines.

For each bug, we separately use the four SZZ variants to identify and label bug-introducing changes. Then, we get the labeled data by the four SZZ variants. Table 3 shows the number and percentage of bug-introducing changes in the labeled data by the four SZZ variants in each project.

Neto et al. concluded that RA-SZZ is more accurate than MA-SZZ based on the assumption that RefDiff achieves high precision in detecting refactoring modifications [56]. To verify whether the assumption holds for our study, we perform a manual analysis of RefDiff as follows. We first run RefDiff on each of the ten studied projects. RefDiff takes as input the commit id of each change. It outputs the refactoring changes with refactoring modifications in the changes (e.g., rename method). Table 3 presents the number of refactoring changes as detected by RefDiff for each project. Then, for each project, we randomly sample 100 refactoring changes. In total, we sample 1,000 changes. Next, the first and third author separately determined whether the sampled changes have performed the refactoring modifications as detected by RefDiff. Both authors have multiple years of programming experience in Java. Finally, both authors compared their determination results to uncover any disagreements, i.e., refactoring changes on which the two authors have different decisions. We find that for 4% changes, the two authors have different decisions—the two authors share the same decision on 96% of the 1,000 changes. For each change with decision conflict, the two authors further discuss whether the change contains the refactoring modifications as detected by RefDiff.

In Table 3, we show the precision of RefDiff on the sampled changes in each project. From the table, we notice that RefDiff achieves a precision of 97%–99% across the ten projects. And it achieves a precision of 98.1% on the 1,000 sampled changes. Our manual analysis verifies the assumption that RefDiff can achieve high precision. Hence, RA-SZZ generates cleaner data than MA-SZZ. Furthermore, as described in Section 2.2.2, prior studies have evaluated the generated data by the different SZZ variants [15], [16], [40], [56]. According to Da Costa et al.'s analysis [15], MA-SZZ is more accurate than B-SZZ and AG-SZZ. Hence, RA-SZZ can generate the cleanest data compared to the other SZZ variants.

To be fair to the models analyzed in our study, we need to evaluate the models on the same testing set which contains as few mislabeled changes as possible. RA-SZZ generates the cleanest data compared to the other SZZ variants. Hence, in this study, we create the testing set using the labels output by RA-SZZ—all the models trained using the labeled data by the four SZZ variants will be evaluated on the same testing set.

TABLE 2: The four studied SZZ variants.

SZZ	Description	Step 2: Mapping Strategy	Step 3: Searching Strategy
B-SZZ	The original SZZ proposed by Śliwerski et al. [65]	All lines modified by bug-fixing changes are identified as buggy lines.	The built-in `annotate` command of VCS is used to identify the last change that modified each line of code in a file for a given change. Using such information, all last changes that modify lines involved in bug-fixing changes are flagged as potential bug-introducing changes.
AG-SZZ	Improved SZZ variant proposed by Kim et al. [40] which is built on top of B-SZZ.	Non-semantic lines (including blank lines, comment lines and those involving code format modifications) are discarded. Other lines involved in bug-fixing changes are identified as buggy lines.	The annotation graph is used to record evolution for lines of code in source files. A depth-first search of the annotation graph is performed to identify potential bug-introducing changes. The searching process does not stop at format modifications to buggy lines.
MA-SZZ	Improved SZZ variant proposed by Da Costa et al. [15] which is built on top of AG-SZZ.	Non-semantic lines and those involving refactoring modifications are discarded. Other lines involved in bug-fixing changes are identified as buggy lines.	In addition to steps done by AG-SZZ, the searching process does not stop at meta-changes.
RA-SZZ	Improved SZZ variant proposed by Neto et al. [56] which is built on top of MA-SZZ.		In addition to steps done by AG-SZZ and MA-SZZ, the searching process does not stop at refactoring modifications to buggy lines.

TABLE 3: Number and percentage of bug-introducing changes in the labeled data by the four SZZ variants. The last two columns show the number of refactoring changes detected by RefDiff and RefDiff's precision on sampled changes.

Project	B-SZZ	AG-SZZ	MA-SZZ	RA-SZZ	#Ref. Changes	Prec. of RefDiff
ActiveMQ	1,697 (22%)	1,157 (15%)	1,336 (17%)	1,245 (16%)	649	98%
Camel	2,957 (17%)	2,032 (12%)	2,186 (13%)	2,018 (12%)	1,832	97%
Derby	1,772 (18%)	1,799 (18%)	1,559 (16%)	1,153 (12%)	1,025	99%
Geronimo	2,309 (14%)	1,245 (08%)	1,999 (12%)	1856 (11%)	1,200	97%
Hadoop C.	1,951 (07%)	1,469 (05%)	1,907 (07%)	1,681 (06%)	3,899	98%
Hbase	3,323 (23%)	2,449 (17%)	3,072 (21%)	2,670 (18%)	1,902	98%
Mahout	539 (20%)	354 (13%)	456 (17%)	420 (15%)	357	99%
OpenJPA	810 (13%)	659 (10%)	853 (13%)	692 (11%)	509	99%
Pig	549 (18%)	441 (14%)	528 (17%)	467 (15%)	263	97%
Tuscany	2,157 (10%)	1,021 (5%)	1,594 (07%)	1,506 (07%)	2,007	99%

TABLE 4: Number of mislabeled changes (misl.), false positives (FP) and false negatives (FN) in the labeled data by B-SZZ, AG-SZZ and MA-SZZ compared to the labeled data by RA-SZZ. False positive rate (FPR) and false negative rate (FNR) of the labeled data by B-SZZ, AG-SZZ and MA-SZZ are also shown in the table.

Project	B-SZZ					AG-SZZ					MA-SZZ				
	#misl.	FP	FPR	FN	FNR	#misl.	FP	FPR	FN	FNR	#misl.	FP	FPR	FN	FNR
ActiveMQ	892	672	10%	220	18%	552	232	4%	320	26%	149	120	2%	29	2%
Camel	1,503	1,221	8%	282	14%	1,186	600	4%	586	29%	268	218	1%	50	1%
Derby	939	779	9%	160	14%	1,016	831	10%	185	16%	572	489	6%	83	6%
Geronimo	931	692	5%	239	13%	1,047	218	2%	829	45%	259	201	1%	58	1%
Hadoop C.	876	573	2%	303	18%	772	280	1%	492	29%	356	291	1%	65	1%
Hbase	1,381	1017	9%	364	14%	1,087	433	4%	654	24%	598	500	4%	98	4%
Mahout	225	172	7%	53	13%	168	51	2%	117	28%	84	60	3%	24	3%
OpenJPA	398	258	5%	140	20%	431	199	4%	232	34%	271	216	4%	55	4%
Pig	196	139	5%	57	12%	166	70	3%	96	21%	91	76	3%	15	3%
Tuscany	1,121	886	4%	235	16%	813	164	1%	649	43%	250	169	1%	81	1%

Moreover, to estimate the amount of the changes that are mislabeled by the different SZZ variants, we need data that contains no false positives nor false negatives. However, in practice, retrieving such clean data is very challenging as it requires a considerable amount of manual effort (e.g., manually analyzing thousands of lines of code) and in-depth domain knowledge of the project (which is only feasible by contacting the core developers of a project). RA-SZZ generates the cleanest data compared to the other three SZZ variants. We compare the generated data by B-SZZ, AG-SZZ and MA-SZZ with the generated data by RA-SZZ. By doing so, we calculate the number of mislabeled changes in the labeled data by B-SZZ, AG-SZZ and MA-SZZ. Also, we calculate the number of false positives and false negatives generated by B-SZZ, AG-SZZ and MA-SZZ. Furthermore, we calculate the false positive rate and false negative rate of the labeled data by the three SZZ variants compared to the labeled data by RA-SZZ. Notice that the false positive rate is calculated as the proportion of false positives over the total changes that are truly clean as labeled by RA-SZZ, and the false negative rate is calculated as the proportion of false negatives over the total changes that are truly bug-introducing as labeled by RA-SZZ. Table 4 shows the number of mislabeled changes, false positives and false negatives in the labeled data by B-SZZ, AG-SZZ and MA-SZZ as compared to the labeled data by RA-SZZ. We also present the false positive rate and false negative rate of the labeled data by B-SZZ, AG-SZZ and MA-SZZ in the table.

We calculate the percentage of the false positives in the bug-introducing changes labeled by B-SZZ, AG-SZZ and MA-SZZ as shown in Table 3. We find that 25%–44%, 14%–46% and 9%–31% of the bug introducing changes labeled by B-SZZ, AG-SZZ and MA-SZZ are false positives (i.e., clean changes labeled by RA-SZZ), respectively. Hence, B-SZZ, AG-SZZ and MA-SZZ generate a considerable number of

TABLE 5: Studied Change Metrics

Dimension	Metric	Description
Diffusion	NS	Number of subsystems modified by this change
	ND	Number of directories modified by this change
	NF	Number of files modified by this change
	Entropy	Distribution of code across the modified files
Size	LA	Lines of code added in this change
	LD	Lines of code deleted in this change
	LT	Lines of code in the files before this change
Purpose	FIX	Whether this change fixes a bug
History	NDEV	Number of developers who modified the files
	AGE	Average time interval between this change and last changes that modify the files
	NUC	Number of changes that modified the files
Experience	EXP	Developer's experience (number of changes)
	REXP	Developer's recent experience
	SEXP	Developer's experience on a subsystem

mislabeled changes.

In Table 4, we notice that AG-SZZ generates more mislabeled changes than B-SZZ in several projects (e.g., Derby)—indicating that AG-SZZ may not be more accurate than B-SZZ. Also, we notice that AG-SZZ generates more false negatives than B-SZZ on the ten studied projects. As mentioned in Section 2.2.2, AG-SZZ is impacted by branch/merge changes (two types of meta-changes) due to the use of annotation graph, while B-SZZ is not impacted by such changes. Due to the impact of branch/merge changes, AG-SZZ generates more false negatives than B-SZZ.

3.3 Studied Metrics

In this section, we introduce the studied change metrics. We use the 14 change metrics that are proposed by Kamei et al. [37]. Table 5 presents the overview of the 14 change metrics, which are grouped along five dimensions including diffusion, size, purpose, history and experience.

The diffusion dimension quantifies distribution of code within a change. Prior work shows that scattered changes are more likely to be defective [29]. The size dimension measures size of a change, and a larger change is more defect-prone since such a change introduces or modifies more code [55]. The purpose dimension only includes the FIX metric, which indicates whether the change fixes a bug. Prior studies show that bug-fixing changes are more likely to be defective and introduce new bugs [24], [59]. The metrics in history dimension characterize how developers modify the files within the change in the code history. These metrics have been shown to be good indicators of bugs, e.g., Matsumoto et al. noted that files previously modified by many developers contain more bugs [49]. The metrics in the experience dimension quantify the change experience of the developer who makes the change. These metrics are based on the number of changes that are previously submitted by the developer. Mockus et al. noted that developers with more experience are less likely to introduce bugs in their changes [54].

3.4 Data Preprocessing

We characterize each change in our datasets using the 14 studied metrics. Directly using these metrics to build models may lead to reduced model performance or incorrect model interpretation. For example, some metrics may be correlated. Tantithamthavorn and Hassan noted that correlated metrics impact interpretation (i.e., metric importance) of defect prediction models [70]. Hence, we need to preprocess our datasets before we build JIT models following prior studies [37], [45], [62]. Our data preprocessing includes the following two parts:

Dealing with Skew. Most of our change metrics are highly skewed, and to alleviate the effect of highly skewed values of the metrics, we apply a logarithmic transformation following prior studies [37], [62]. And we use the standard logarithmic transformation $ln(x + 1)$. We apply the logarithmic transformation for each metric except the FIX metric since the FIX metric is a binary variable.

Dealing with Correlated Metrics. Tantithamthavorn et al. demonstrated that correlated metrics can impact defect prediction models [70]. Hence, we perform metric selection to remove correlated metrics before we use the metrics to learn a model. Change metrics in different projects may show different degrees of correlation, and thus, we separately perform the metric selection on each project. Notice that our metric selection are performed after we apply the logarithmic transformation for our metrics since our models are trained using the transformed metrics. Following the guidelines proposed by Harrel for building models [27], we perform the metric selection using the following steps:

(1) Correlation Analysis. In this step, we deal with the correlation between each pair of metrics. For each project, we first calculate the correlation between each pair of metrics leveraging the Spearman rank correlation test [84]. Furthermore, we use the variable clustering analysis implemented in the **Hmisc** R package to cluster the correlated metrics based on the results of Spearman rank correlation test. Following Li et al. [45], we set the correlation threshold for removing collinearity as 0.8. If the absolute correlation value between a pair of metrics is larger than 0.8, we keep the one that is easier to understand to ease model interpretation following prior studies [37], [45]. For example, we notice that EXP and REXP are highly correlated in each project. We keep the EXP metric and drop the REXP metric.

(2) Independence Analysis. We notice that FIX is a binary variable and the other metrics are continuous. The Spearman rank correlation test may not work for the FIX metric since this metric does not have an order between its two values. Hence, in this step, we use the Chi-squared test of independence [58] to analyze the statistical independence of the FIX metric from each of the other continuous metrics. We apply the *chisq.test* function in the **vcd** R package [53]. We notice that the FIX metric is not independent from the other metrics (e.g., LA) on our studied projects, and thus the FIX metric is dropped.

(3) Redundancy Analysis. In this step, we further remove redundant metrics that can be predicted by the combination of other metrics. These redundant metrics should be removed since they do not contribute to the model in the appearance of the other metrics. To detect such redundant metrics, we apply the *redun* function implemented in the **rms** R package [28]. For all the studied projects, we did

not detect any redundant metric. Hence, we do not remove metrics in this step.

3.5 Case Study Setup

We adopt the out-of-sample bootstrap technique [18] to evaluate the performance and metric importance for our JIT models. Tantithamthavorn et al. suggested that out-of-sample bootstrap tends to produce performance estimates with the least bias and variance [73]. Using the out-of-sample bootstrap can ensure that our conclusions about JIT models are robust.

The out-of-sample bootstrap on an original change dataset of size N involves two steps. First, N changes are bootstrap sampled with replacement from the original change dataset. These changes serve as the training data. Second, the changes that do not appear in the N sampled ones are also retrieved, which serve as the testing data. On average, the testing data pertains to 36.8% of the original change data [73].

Figure 5 depicts the overall framework for evaluating the performance and metric importance for the JIT models trained using the labeled data by the studied four SZZ variants in an out-of-sample bootstrap. We take seven steps to do that:

1. *Perform out-of-sample bootstrap.* Changes in a project are split into training dataset and testing dataset using out-of-sample bootstrap process described earlier.
2. *Label the training dataset.* We leverage B-SZZ, AG-SZZ, MA-SZZ and RA-SZZ to label the training dataset.
3. *Build model.* We build a prediction model using the training dataset labeled by each SZZ variant. In total, we have four JIT models trained using the training data that is labeled by B-SZZ, AG-SZZ, MA-SZZ and RA-SZZ, respectively. Our methods for building models are described in Section 3.6.
4. *Apply models.* We apply the four JIT models on the testing dataset. For each change in the testing dataset, each of the four models predicts whether the change is likely to introduce bugs and outputs a label.
5. *Label the testing dataset.* As mentioned in Section 3.2, all the models are evaluated on the data that is labeled by RA-SZZ. Hence, we apply RA-SZZ to label changes in the testing dataset.
6. *Analyze model performance.* We calculate performance measures (such as AUC) based on the prediction results output by the four models and the labels of the testing dataset as generated by RA-SZZ.
7. *Calculate metric importance.* We calculate importance score of each change metric for each of the four models. To calculate metric importance for a model, we apply the generic metric importance score proposed by Tantithamthavorn et al. [74]. This importance score is elaborated in Section 3.7.

The out-of-sample bootstrap process is repeated 1,000 times. After the 1,000 runs, we further conduct our analysis on the performance measures and the metric importance calculated in the 1,000 runs.

3.6 Model Building

In this study, we evaluate two groups of models: JIT models which predict a change as bug-introducing or clean, and effort-aware JIT models which prioritize changes for developers to inspect. We describe our method for building the two groups of models as follows.

3.6.1 Building Prediction Models

We build JIT models using two settings:

1. We use the original training data to build JIT models. Prior JIT defect prediction studies have built models using the original training data [36], [50].
2. Also, we notice that the labeled data by SZZ is imbalanced. To deal with the imbalanced data, Kamei et al. re-balanced training data before building their JIT models [37]. Hence, we also build models on re-balanced training data and investigate the impact of mislabeled changes by SZZ on such models. To re-balance our training data, we leverage the undersampling method following Kamei et al. [37]. From training data, we randomly remove the changes in the majority class (i.e., clean changes) until the majority class drops to the same level as the minority class (i.e., bug-introducing changes). Notice that we do not re-balance the testing data.

For both settings, we leverage three types of classifiers to build models. These classifiers include machine learning-based (i.e., random forest), regression-based (i.e., logistic regression) and probability-based (i.e., naive Bayes) classifiers. We choose the three types of classifiers following Tantithamthavorn et al.'s study [73]. Moreover, the studied classifiers are also widely used in prior JIT defect prediction studies [34]–[37], [63]. We briefly describe the classifiers as follows.

Random Forest. Random forest is proposed by Breiman [8]. It is an ensemble approach which is specially designed for decision trees. A random forest model contains multiple decision trees, each of which is built using a random subset of metrics. When deciding the class of a sample, the decision trees may report different outcomes. Random forest aggregates votes of the decision trees to decide a final class for the sample. In this work, we use the random forest implementation provided by the **randomForest** R package [46].

Logistic Regression. Logistic regression is a regression-based technique which is usually applied to estimate the relationship between a binary dependent variable (i.e., bug-introducing change or clean change in our case) and one or more independent variables (i.e., our studied metrics) [32]. In this work, we implement logistic regression using the **glm** R function.

Naive Bayes. Naive Bayes is a probabilistic classifier which is based on Bayes' theorem and assumes that all predictors are independent of each other [17]. We implement naive Bayes using the **naivebayes** R package [47].

As shorthand notations, we denote random forest, logistic regression and naive Bayes as **RF**, **LR** and **NB**, respectively.

3.6.2 Building Effort-Aware Models

We build effort-aware JIT defect prediction models using two state-of-art effort-aware JIT defect prediction techniques, i.e., CBS [34] and OneWay [21]. We choose the

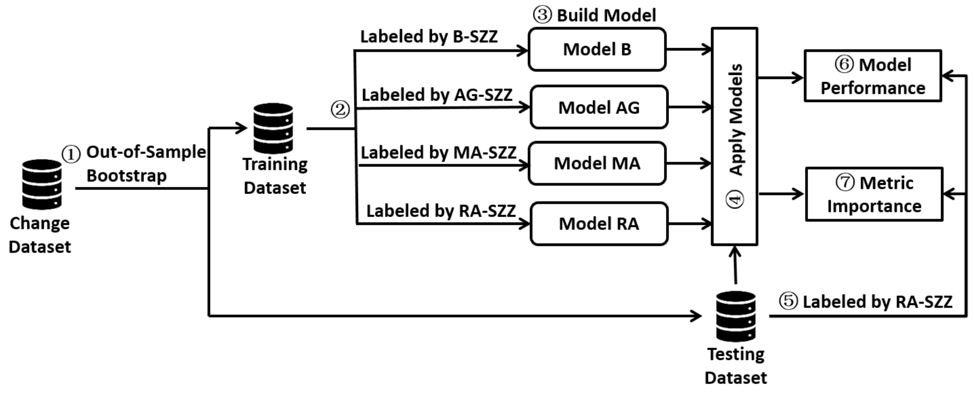


Fig. 5: Evaluating performance and metric importance for the JIT models trained using the labeled data by different SZZ variants in an out-of-sample bootstrap. The described process is repeated 1,000 times.

two techniques since they show better cost-effectiveness than other state-of-art techniques [21], [34]. The two techniques are elaborated below.

CBS is a supervised effort-aware JIT defect prediction technique proposed by Huang et al. [34]. CBS first uses a supervised model to predict the defect-proneness of changes. Then, for changes that are predicted as buggy, CBS prioritizes them using their churn size in ascending order. Following Huang et al.'s study [34], when building the supervised model that is used by CBS, we use logistic regression as the underlying classifier. Moreover, following Huang et al.'s study [34], we use the undersampling method to re-balance the class distribution of training data before building models.

OneWay is a supervised effort-aware JIT defect prediction technique proposed by Fu et al. [21]. OneWay is inspired by Yang et al.'s unsupervised technique [83]. Yang et al.'s models first sort changes using one of Kamei et al.'s metrics (e.g., LT) [37], then prioritize changes for developers in that order. Fu et al. observed that most of Yang et al.'s unsupervised models cannot achieve better cost-effectiveness than supervised models in the within-project setting [21]. They noted the necessity of using supervised data to prune weaker models away. OneWay first evaluates Yang et al.'s unsupervised models on labeled training data, then it selects the unsupervised model with the best cost-effectiveness (in our study, we use Recall@20% to measure the cost-effectiveness) to prioritize changes in the testing data.

3.7 Metric Importance Calculation

Tantithamthavorn et al. proposed a generic metric importance score based on Breiman's metric importance score [8], [74]. Tantithamthavorn et al.'s metric importance score can be applied for any classifier [74]. In comparison, Breiman's metric importance score can only be applied for random forest. In this study, we take three types of classifiers (including random forest, logistic regression and naive Bayes) as the underlying classifier for building models. Hence, Tantithamthavorn et al.'s metric importance score is more appropriate for our case. Moreover, using the same importance measurement gives us a way of conducting metric importance estimation in an unbiased setting.

Following Tantithamthavorn et al. [74], we calculate metric importance scores for each of the four models by taking two steps in each bootstrap iteration:

1. For each metric, we first randomly permutate the values of the metric on the testing dataset, producing a testing dataset with the metric permutated and all other metrics unchanged.
2. For each metric, we calculate the difference of the misclassification rate when applying the model on the original testing dataset and when applying it on the testing dataset with the metric randomly permutated. The larger the difference, the greater is the importance of the metric. Hence, the difference is considered as the importance score of the metric.

Tantithamthavorn and Hassan note that class re-balancing techniques should be avoided when calculating metric importance for classifiers [70]. Hence, in this study, we calculate metric importance in identifying bug-introducing changes using the models that are trained on the original data.

3.8 Performance Measures

We use the AUC, F1-score and G-mean to evaluate JIT models with respect to the performance in identifying bug-introducing changes. And we use the Recall@20% as the performance measure to evaluate the effort-aware JIT models with respect to their performance in prioritizing changes. The performance measures are described below.

AUC: AUC is defined as the area under the Receiver Operator Characteristics Curve, which plots true positive rate against false positive rate across all thresholds [33]. Notice that in this context, a false positive refers to a truly clean change (as labeled by RA-SZZ) that is predicted as bug-introducing. AUC ranges from 0 to 1. A larger AUC score indicates better performance. A classifier with an AUC of 0.5 is considered to be no better than random guessing. AUC has been used in the evaluation of many prior Just-in-Time defect prediction studies [36], [37], [50].

AUC is a threshold-independent measure. When a classifier determines the class of a change, it calculates a probability score to be bug-introducing for the change, and then, a threshold is needed to be set up to classify the change as bug-introducing if the score is higher than the threshold and clean otherwise. AUC evaluates the prediction performance at all thresholds (from 0 to 1). Hence, AUC is independent of the threshold.

Furthermore, the calculation of AUC automatically considers the inherent class imbalance of a dataset [60]. Thus, AUC is not sensitive towards class distributions and the level of data imbalance.

F1-score: F1-score is calculated based on Precision and Recall. For bug-introducing changes, Precision is the proportion of changes that are correctly predicted as bug-introducing over the total of changes that are predicted as bug-introducing. And Recall is the proportion of changes that are correctly predicted as bug-introducing over the total of changes that are truly bug-introducing. There is a tradeoff between Precision and Recall. Thus, prior studies use F1-score—a summary measure combining both precision and recall—to evaluate the performance of JIT models [35], [37], [63]. Let us denote Precision and Recall for bug-introducing changes as $P(b)$ and $R(b)$, respectively. F1-score for bug-introducing changes is calculated as the harmonic mean of $P(b)$ and $R(b)$ (See Eq. 1).

$$F1\text{-}score = \frac{2 \times P(b) \times R(b)}{P(b) + R(b)} \tag{1}$$

Tantithamthavorn et al. show that different levels of data imbalance can impact the F1-score of defect prediction models [71]. Thus, for the models that are trained using the original data, we avoid using F1-score as the performance measure. On the other hand, we use F1-score to evaluate the models built using the re-balanced data, since the class imbalance problem has been mitigated in this case.

G-mean: G-mean is a summary measure of the Recall scores for the two classes (bug-introducing and clean changes in our case) [43]. G-mean assumes that the Precision of the two classes has equal weight. And this measure tries to maximize the Recall for each class while keeping the Recall scores balanced. Let us denote the Recall for clean changes as $R(c)$. $R(c)$ is calculated as the proportion of changes that are correctly predicted as clean over the total number of changes that are truly clean. G-mean is calculated as the geometric mean of $R(b)$ and $R(c)$ (See Eq. 2). This measure is also used by many prior defect prediction studies [11], [68].

$$G\text{-}mean = \sqrt{R(b) \times R(c)} \tag{2}$$

Kubat et al.'s study showed that G-mean is also impacted by the level of data imbalance [43]. Hence, similarly to F1-score, we do not use the G-mean to evaluate the models trained using the original imbalanced data. And we use the G-mean to evaluate the models that are trained using the re-balanced data.

Recall@20%: Recall@20% is defined as the Recall of the bug-introducing changes when inspecting 20% of the total lines of code (LOC). Defect prediction studies have used LOC as a measure of the needed effort to inspect code [9], [10], [52]. Besides, empirical studies showed that around 80% of defects are in 20% of the files [26], [57]. Hence, many JIT defect prediction studies assume nowadays that available resources only pertain to 20% of the effort to inspect all changes and commonly use Recall@20% as their performance measure to evaluate the cost-effectiveness of JIT models [21], [34], [35], [37], [83].

4 Results

In this section, we present the results of our analysis with respect to the five research questions.

4.1 RQ1: How do mislabeled changes by SZZ impact the performance of JIT defect prediction models trained using the original data?

Motivation: In practice, JIT models can be used to *predict* which changes are likely to be defective when they are initially submitted. Leveraging the prediction results, software developers can fix their changes before committing a bug-introducing change into the code base, which would save the effort of fixing bugs in the future. In this research question, we would like to investigate whether mislabeled changes by SZZ impact the prediction performance of JIT models. Prior studies have used the original training data to build JIT defect prediction models [36], [50]. In this research question, we investigate the impact of mislabeled changes by SZZ on the JIT models trained using the original data.

Approach: As mentioned in Section 3.2, changes labeled by RA-SZZ are less likely to be mislabeled than those labeled by the other three SZZ variants. We take the model trained using the labeled data by RA-SZZ as the baseline model. By comparing the models trained using the labeled data by B-SZZ, AG-SZZ and MA-SZZ with the baseline model, we investigate the impact of the mislabeled changes by the three SZZ variants on the performance of the JIT models. We perform the comparison for each of the studied three classifiers (i.e., random forest, logistic regression and naive Bayes). As described in Section 3.8, we use AUC as the performance measure to combat the bias induced by data imbalance following prior studies [36], [50].

As illustrated in Figure 5, we train four models in each bootstrap iteration using training data that is labeled by B-SZZ, AG-SZZ, MA-SZZ and RA-SZZ, respectively. As shorthand notations, we denote the four models as B, AG, MA and RA models, respectively. Then, we apply the four models on the testing data that is labeled by RA-SZZ and calculate the AUC score assuming that the labels produced by RA-SZZ are correct. After 1,000 bootstrap iterations, we have 1,000 AUC scores for each of the four models. We calculate the average value of the 1,000 AUC scores. Then we compare the average AUC scores of the B, AG and MA models with those of the RA model. To do so, we calculate the ratios of the average AUC scores of the B, AG and MA models to those of the RA model.

Furthermore, we investigate whether B, AG and MA models show a statistically significant performance reduction in terms of AUC. To do this, we apply the Wilcoxon signed-rank test [76] with a Bonferroni correction [2] to compare the AUC scores for the B, AG and MA models with the RA model. We also calculate the Cliff's delta, which can evaluate "how frequently the values of one distribution is higher than the values of another distribution" [12]. The Cliff's delta is a widely used effect size measure[3]. A negative Cliff's delta indicates performance reduction of the B, AG or MA model as compared to the RA model.

Notice that the Cliff's delta does not reflect the distance in magnitude between the values of two distributions. For example, if all the values of the first distribution D_1 are 0.91 and all the values of the second distribution D_2 are 0.90, the Cliff's delta is 1 (i.e., a large effect size) since the frequency of the event *"values in D_1 are larger than values in D_2"* is 1. However, the absolute difference between the values of the two distributions is actually small (only 0.01). Hence, for the cases where the B, AG and MA models show a significant performance reduction with a non-negligible effect size, we need to verify whether the absolute difference in performance is substantially large. To do so, we look into each case and calculate the ratio of the absolute difference in performance to the performance of the RA model, i.e., the percentage of the performance reduction.

TABLE 6: AUC scores of the B, AG, MA and RA models. We also show the ratios of the AUC scores of the B, AG and MA models to those of the RA model. The AUC scores of the B, AG and MA models that are lower than those of the RA model are in bold. *Cls* refers to the underlying classifier used by the four models.

Cls	**Project**	**B**		**AG**		**MA**		**RA**
RF	ActiveMQ	0.80	100%	**0.78**	**98%**	0.81	101%	0.80
	Camel	**0.84**	**99%**	**0.81**	**95%**	0.85	100%	0.85
	Derby	**0.79**	**99%**	**0.77**	**96%**	**0.79**	**99%**	0.80
	Geronimo	0.84	101%	**0.81**	**98%**	0.83	100%	0.83
	Hadoop C.	**0.87**	**99%**	**0.86**	**98%**	0.88	100%	0.88
	HBase	**0.83**	**97%**	**0.82**	**95%**	**0.85**	**99%**	0.86
	Mahout	0.85	102%	**0.82**	**99%**	0.84	101%	0.83
	OpenJPA	**0.79**	**99%**	**0.75**	**94%**	**0.79**	**99%**	0.80
	Pig	**0.80**	**99%**	**0.80**	**99%**	0.81	100%	0.81
	Tuscany	0.82	100%	**0.79**	**96%**	0.82	100%	0.82
Avg.		0.82	99%	0.80	96%	0.83	100%	0.83
LR	ActiveMQ	0.79	100%	**0.78**	**99%**	0.79	100%	0.79
	Camel	0.79	100%	**0.78**	**99%**	0.79	100%	0.79
	Derby	0.76	100%	0.76	100%	0.76	100%	0.76
	Geronimo	0.79	100%	**0.78**	**99%**	0.79	100%	0.79
	Hadoop C.	0.79	100%	**0.78**	**99%**	0.79	100%	0.79
	HBase	0.78	100%	**0.77**	**99%**	0.78	100%	0.78
	Mahout	0.83	101%	**0.81**	**99%**	0.82	100%	0.82
	OpenJPA	0.74	100%	**0.73**	**99%**	0.74	100%	0.74
	Pig	0.78	100%	**0.77**	**99%**	0.78	100%	0.78
	Tuscany	0.81	100%	0.81	100%	0.81	100%	0.81
Avg.		0.79	100%	0.78	99%	0.79	100%	0.79
NB	ActiveMQ	0.74	100%	**0.73**	**99%**	0.74	100%	0.74
	Camel	0.77	101%	**0.73**	**96%**	0.75	99%	0.76
	Derby	**0.74**	**99%**	**0.74**	**99%**	**0.73**	**97%**	0.75
	Geronimo	0.76	101%	**0.73**	**97%**	0.75	100%	0.75
	Hadoop C.	0.74	100%	0.74	100%	0.74	100%	0.74
	HBase	0.76	100%	**0.75**	**99%**	**0.75**	**99%**	0.76
	Mahout	0.80	100%	**0.79**	**99%**	0.80	100%	0.80
	OpenJPA	0.74	101%	**0.71**	**97%**	0.73	100%	0.73
	Pig	0.75	100%	**0.74**	**99%**	0.75	100%	0.75
	Tuscany	**0.76**	**99%**	0.77	100%	0.77	100%	0.77
Avg.		0.76	100%	0.74	97%	0.75	99%	0.76

Results: Table 6 presents the average AUC scores of the B, AG, MA and RA models and the ratios of the average AUC scores of the B, AG and MA models to those of the RA model. And Table 7 shows the adjusted p-values and Cliff's delta comparing AUC scores for the B, AG and MA models with those of the RA model.

From Tables 6 and 7, we have the following findings:

- For the B model, the AUC score is 97%–102% of that of the RA model. On average across the ten projects, the AUC score of the B model is 99%–100% of that of the RA model. When random forest is used as the underlying classifier of the models, statistical tests show that the AUC score of the B model is statistically significantly lower than that of the RA model with a non-negligible effect size on seven projects. When logistic regression and naive Bayes are used as the underlying classifier of the models, statistical tests show that in most cases, the B model is not performing statistically significantly worse than the RA model.
- For the AG model, the AUC score is 94%–100% of that of the RA model. On average across the ten projects, the AUC score of the AG model is 96%–99% of that of the RA model. Statistical tests show that in most cases, the AUC score of the AG model is statistically significantly lower than that of the RA model. Moreover, in most cases, the effect sizes are non-negligible.
- For the MA model, the AUC score is 97%–101% of that of the RA model. On average across the ten projects, the AUC score of the MA model is 99%–100% of that of the RA model. Statistical tests show that in most cases, the MA model is not performing statistically significantly worse than the RA model.

In most of the cases where the B and MA models show a statistically significant performance reduction as compared to the RA model, we find that the B and MA models only show 1% performance reduction in terms of AUC. Such difference may not be deemed as substantially large for practitioners. On the other hand, we notice that the AG model shows more performance reduction in terms of AUC. In several cases, the AG model shows at least 5% performance reduction.

In summary, the mislabeled changes by B-SZZ and MA-SZZ are not likely to cause a considerable performance reduction in terms of AUC. Finally, the mislabeled changes by AG-SZZ can cause a statistically significant performance reduction with an average difference ranging from 1% to 4% in terms of AUC.

4.2 RQ2: How do mislabeled changes by SZZ impact the performance of JIT defect prediction models trained using re-balanced data?

Motivation: Apart from using measures that are insensitive towards class imbalance (e.g., AUC), prior studies also leveraged class re-balancing techniques when building JIT defect prediction models to combat the class imbalance problem [37], [69]. In this research question, we investigate the impact of mislabeled changes by SZZ on the performance of JIT models trained using re-balanced data.

Approach: Following Kamei et al. [37], we re-balance the training data that is labeled by each SZZ variant using the undersampling method. In each bootstrap iteration, we train B, AG, MA and RA models using the re-balanced training data. As mentioned in Section 3.8, in this setting, we use AUC, F1-score and G-mean to evaluate performance of the models.

3. A Cliff's delta less than 0.147, between 0.147 and 0.33, between 0.33 and 474, and larger than 0.474 is considered as a negligible, small, medium and large effect size, respectively.

TABLE 7: Adjusted P-values and Cliff's delta comparing AUC scores for the B, AG and MA models with those of the RA model. P-values and Cliff's delta that show significant performance reduction are in bold.

Cls	Project	B		AG		MA	
RF	ActiveMQ	**-0.15**	**(S)*****	**-0.91**	**(L)*****	0.08	(N)
	Camel	**-0.32**	**(S)*****	**-1.00**	**(L)*****	0.02	(N)
	Derby	**-0.30**	**(S)*****	**-0.93**	**(L)*****	**-0.42**	**(M)*****
	Geronimo	0.22	(S)	**-0.99**	**(L)*****	0.11	(N)
	Hadoop C.	**-0.81**	**(L)*****	**-0.96**	**(L)*****	0.14	(N)
	HBase	**-1.00**	**(L)*****	**-1.00**	**(L)*****	**-0.65**	**(L)*****
	Mahout	0.48	(L)	**-0.52**	**(L)*****	0.14	(N)
	OpenJPA	**-0.20**	**(S)*****	**-0.97**	**(L)*****	**-0.17**	**(S)*****
	Pig	**-0.31**	**(S)*****	**-0.40**	**(M)*****	0.14	(N)
	Tuscany	0.44	(M)	**-0.99**	**(L)*****	-0.04	(N)***
LR	ActiveMQ	0.05	(N)	**-0.48**	**(L)*****	0.00	(N)
	Camel	-0.12	(N)***	**-0.89**	**(L)*****	-0.06	(N)***
	Derby	**-0.19**	**(S)*****	**-0.34**	**(M)*****	**-0.26**	**(S)*****
	Geronimo	-0.01	(N)	**-0.47**	**(M)*****	0.03	(N)
	Hadoop C.	0.41	(M)	**-0.63**	**(L)*****	-0.02	(N)***
	HBase	0.01	(N)	**-0.37**	**(M)*****	-0.03	(N)***
	Mahout	0.14	(N)	**-0.29**	**(S)*****	0.03	(N)
	OpenJPA	-0.08	(N)***	**-0.58**	**(L)*****	0.03	(N)
	Pig	0.07	(N)	**-0.27**	**(S)*****	-0.06	(N)***
	Tuscany	-0.06	(N)***	**-0.27**	**(S)*****	-0.03	(N)***
NB	ActiveMQ	0.18	(S)	**-0.47**	**(M)*****	0.10	(N)
	Camel	0.70	(L)	**-0.97**	**(L)*****	**-0.18**	**(S)*****
	Derby	-0.12	(N)***	**-0.25**	**(S)*****	**-0.49**	**(L)*****
	Geronimo	0.29	(S)	**-0.76**	**(L)*****	0.06	(N)
	Hadoop C.	0.00	(N)	**-0.05**	**(N)*****	-0.05	(N)***
	HBase	0.11	(N)	**-0.45**	**(M)*****	**-0.19**	**(S)*****
	Mahout	0.13	(N)	**-0.33**	**(M)*****	0.05	(N)
	OpenJPA	0.29	(S)	**-0.69**	**(L)*****	-0.02	(N)**
	Pig	0.02	(N)	**-0.43**	**(M)*****	-0.06	(N)***
	Tuscany	**-0.63**	**(L)*****	0.07	(N)	-0.14	(N)***

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

TABLE 8: AUC scores of the B, AG, MA and RA models. We also show the ratios of the AUC scores of the B, AG and MA models to those of the RA model. The AUC scores of the B, AG and MA models that are lower than those of the RA model are in bold.

Cls	Project	B		AG		MA		RA
RF	ActiveMQ	**0.80**	**99%**	**0.79**	**98%**	0.81	100%	0.81
	Camel	**0.84**	**99%**	**0.82**	**96%**	0.85	100%	0.85
	Derby	**0.79**	**99%**	**0.78**	**98%**	**0.79**	**99%**	0.80
	Geronimo	0.84	100%	**0.81**	**96%**	0.84	100%	0.84
	Hadoop C.	0.87	99%	**0.86**	**98%**	0.88	100%	0.88
	HBase	0.83	98%	**0.82**	**96%**	**0.84**	**99%**	0.85
	Mahout	0.85	101%	**0.83**	**99%**	0.84	100%	0.84
	OpenJPA	0.80	100%	**0.76**	**95%**	0.80	100%	0.80
	Pig	0.80	100%	**0.79**	**99%**	0.81	101%	0.80
	Tuscany	0.83	100%	**0.82**	**99%**	0.83	100%	0.83
Avg.		0.83	100%	0.81	98%	0.83	100%	0.83
LR	ActiveMQ	0.79	100%	**0.78**	**99%**	0.79	100%	0.79
	Camel	0.79	100%	**0.78**	**99%**	0.79	100%	0.79
	Derby	0.76	100%	0.76	100%	0.76	100%	0.76
	Geronimo	0.79	100%	**0.77**	**97%**	0.79	100%	0.79
	Hadoop C.	0.80	103%	**0.76**	**97%**	0.78	100%	0.78
	HBase	0.78	100%	0.78	100%	0.78	100%	0.78
	Mahout	0.83	101%	0.82	100%	0.82	100%	0.82
	OpenJPA	0.74	100%	**0.72**	**97%**	0.74	100%	0.74
	Pig	0.78	100%	**0.77**	**99%**	**0.77**	**99%**	0.78
	Tuscany	0.81	100%	**0.80**	**99%**	0.81	100%	0.81
Avg.		0.79	101%	0.77	99%	0.78	100%	0.78
NB	ActiveMQ	0.74	100%	**0.73**	**99%**	0.74	100%	0.74
	Camel	0.77	101%	**0.73**	**96%**	**0.75**	**99%**	0.76
	Derby	0.74	100%	0.74	100%	**0.73**	**99%**	0.74
	Geronimo	0.76	101%	**0.73**	**97%**	0.75	100%	0.75
	Hadoop C.	0.74	100%	0.74	100%	0.74	100%	0.74
	HBase	0.76	100%	**0.75**	**99%**	**0.75**	**99%**	0.76
	Mahout	0.80	101%	**0.78**	**99%**	0.80	101%	0.79
	OpenJPA	0.74	101%	**0.71**	**97%**	0.73	100%	0.73
	Pig	0.75	100%	**0.74**	**99%**	0.75	100%	0.75
	Tuscany	**0.76**	**99%**	0.77	100%	0.77	100%	0.77
Avg.		0.76	101%	0.74	99%	0.75	100%	0.75

We calculate the average AUC, F1-score and G-mean across the 1,000 bootstrap iterations for the B, AG, MA and RA models. For the average AUC scores, F1-scores and G-mean scores, we calculate the ratios of the scores of the B, AG and MA models to those of the RA model. Moreover, we apply the Wilcoxon signed-rank test with a Bonferroni correction to investigate whether the B, AG and MA models show statistically significant performance reduction in terms of AUC, F1-score and G-mean. We also calculate the Cliff's delta.

Results: Table 8, 9 and 10 present the average AUC, F1-score and G-mean score of the B, AG, MA and RA models that are trained using re-balanced data, respectively. The tables also show the ratios of the performance of the B, AG and MA models to that of the RA model in terms of AUC, F1-score and G-mean score, respectively. Table 11, 12 and 13 present the adjusted p-values and Cliff's delta comparing the AUC, F1-score and G-mean score for the B, AG and MA models with the RA model, respectively.

From the two tables, we have the following findings:

- For the B model, the AUC score is 98%–103% of that of the RA model, the F1-score is 94%–107% of that of the RA model, and the G-mean score is 96%–103% of that of the RA model. In terms of AUC, the B model performs statistically significantly worse than the RA model on seven projects when using random forest as the underlying classifier; but the B model does not perform worse than the RA model in most cases when using logistic regression or naive Bayes as the underlying classifier. In terms of F1-score, the B model performs significantly worse than the RA model on ten and seven projects when using random forest and logistic regression as the underlying classifier, respectively; but the B model does not perform worse than the RA model in most cases when using naive Bayes as the underlying classifier. In terms of G-mean, the B model does not perform significantly worse than the RA model in most cases.
- For the AG model, the AUC score is 95%–100% of that of the RA model, the F1-score is 88%–113% of that of the RA model, and the G-mean score is 89%–100% of that of the RA model. Statistical tests show that in most cases, the AUC score, F1-score and G-mean score of the AG model are statistically significantly lower than the three scores of the RA model with a non-negligible effect size.
- For the MA model, the AUC score is 99%–101% of that of the RA model, the F1-score is 86%–103% of that of the RA model, and the G-mean score is 96%–101% of that of the RA model. Statistical tests show that in most cases, the MA model is not performing worse than the RA model in terms of AUC and F1-score. In terms of G-mean, the MA model performs significantly worse than the RA model on five projects when using naive Bayes as the underlying classifier, whereas it does not perform significantly worse than the RA model when using the other classifiers.

In terms of AUC, we notice that the B and MA models show less than a 1% of performance reduction in most

TABLE 9: F1-scores of the B, AG, MA and RA models. We also show the ratios of the F1-scores of the B, AG and MA models to those of the RA model. The F1-scores of the B, AG and MA models that are lower than those of the RA model are in bold.

Cls	Project	B		AG		MA		RA
RF	ActiveMQ	**0.47**	**96%**	**0.46**	**94%**	0.49	100%	0.49
	Camel	**0.44**	**96%**	**0.42**	**91%**	**0.45**	**98%**	0.46
	Derby	**0.39**	**95%**	**0.38**	**93%**	**0.40**	**98%**	0.41
	Geronimo	0.44	100%	**0.42**	**95%**	0.44	100%	0.44
	Hadoop C.	**0.35**	**97%**	**0.33**	**92%**	0.36	100%	0.36
	HBase	**0.54**	**95%**	**0.53**	**93%**	**0.55**	**96%**	0.57
	Mahout	0.51	100%	**0.50**	**98%**	0.51	100%	0.51
	OpenJPA	0.39	100%	**0.35**	**90%**	**0.38**	**97%**	0.39
	Pig	**0.46**	**98%**	**0.45**	**96%**	0.47	100%	0.47
	Tuscany	**0.31**	**97%**	**0.31**	**97%**	**0.31**	**97%**	0.32
Avg.		0.43	98%	0.42	95%	0.44	100%	0.44
LR	ActiveMQ	**0.44**	**98%**	**0.44**	**98%**	0.45	100%	0.45
	Camel	**0.38**	**97%**	**0.37**	**95%**	0.39	100%	0.39
	Derby	**0.35**	**95%**	**0.35**	**95%**	**0.36**	**97%**	0.37
	Geronimo	**0.36**	**97%**	**0.36**	**97%**	**0.36**	**97%**	0.37
	Hadoop C.	0.26	104%	**0.23**	**92%**	**0.24**	**96%**	0.25
	HBase	**0.48**	**98%**	**0.48**	**98%**	**0.48**	**98%**	0.49
	Mahout	**0.48**	**98%**	**0.48**	**98%**	0.49	100%	0.49
	OpenJPA	0.31	100%	0.31	100%	0.31	100%	0.31
	Pig	0.42	100%	**0.41**	**98%**	**0.41**	**98%**	0.42
	Tuscany	0.27	100%	0.27	100%	0.27	100%	0.27
Avg.		0.38	100%	0.37	97%	0.38	100%	0.38
NB	ActiveMQ	**0.33**	**97%**	**0.33**	**97%**	0.34	100%	0.34
	Camel	0.36	106%	**0.30**	**88%**	**0.33**	**97%**	0.34
	Derby	0.36	100%	**0.33**	**92%**	**0.31**	**86%**	0.36
	Geronimo	0.30	100%	0.35	117%	0.30	100%	0.30
	Hadoop C.	**0.17**	**94%**	**0.16**	**89%**	**0.17**	**94%**	0.18
	HBase	0.44	100%	**0.43**	**98%**	**0.43**	**98%**	0.44
	Mahout	0.50	100%	**0.48**	**96%**	0.50	100%	0.50
	OpenJPA	0.32	107%	**0.29**	**97%**	0.31	103%	0.30
	Pig	**0.35**	**97%**	**0.32**	**89%**	**0.35**	**97%**	0.36
	Tuscany	0.27	100%	0.27	100%	0.27	100%	0.27
Avg.		0.34	100%	0.33	97%	0.33	97%	0.34

TABLE 10: G-mean scores of the B, AG, MA and RA models. We also show the ratios of the G-mean scores of the B, AG and MA models to those of the RA model. The G-mean scores of the B, AG and MA models that are lower than those of the RA model are in bold.

Cls	Project	B		AG		MA		RA
RF	ActiveMQ	0.73	100%	**0.70**	**96%**	0.73	100%	0.73
	Camel	**0.76**	**99%**	**0.74**	**96%**	0.77	100%	0.77
	Derby	**0.72**	**99%**	**0.71**	**97%**	**0.72**	**99%**	0.73
	Geronimo	0.76	101%	**0.73**	**97%**	0.75	100%	0.75
	Hadoop C.	0.80	100%	**0.78**	**98%**	0.81	101%	0.80
	HBase	**0.76**	**99%**	**0.74**	**96%**	0.77	100%	0.77
	Mahout	0.76	100%	**0.75**	**99%**	0.76	100%	0.76
	OpenJPA	0.72	100%	**0.68**	**94%**	0.72	100%	0.72
	Pig	0.73	100%	**0.71**	**97%**	0.73	100%	0.73
	Tuscany	0.75	100%	**0.73**	**97%**	0.75	100%	0.75
Avg.		0.75	100%	0.73	97%	0.75	100%	0.75
LR	ActiveMQ	0.72	100%	**0.71**	**99%**	0.72	100%	0.72
	Camel	0.72	100%	**0.71**	**99%**	0.72	100%	0.72
	Derby	0.70	100%	**0.69**	**99%**	0.70	100%	0.70
	Geronimo	0.72	101%	**0.70**	**99%**	0.71	100%	0.71
	Hadoop C.	0.74	103%	**0.70**	**97%**	0.72	100%	0.72
	HBase	0.71	100%	0.71	100%	0.71	100%	0.71
	Mahout	0.75	100%	**0.74**	**99%**	0.75	100%	0.75
	OpenJPA	0.67	100%	**0.66**	**99%**	0.68	101%	0.67
	Pig	0.70	100%	**0.69**	**99%**	0.70	100%	0.70
	Tuscany	0.73	100%	**0.72**	**99%**	0.73	100%	0.73
Avg.		0.72	101%	0.70	99%	0.71	100%	0.71
NB	ActiveMQ	**0.55**	**96%**	0.57	100%	**0.56**	**98%**	0.57
	Camel	0.71	103%	**0.65**	**94%**	0.69	100%	0.69
	Derby	**0.68**	**99%**	**0.68**	**99%**	**0.66**	**96%**	0.69
	Geronimo	0.66	100%	0.68	103%	0.66	100%	0.66
	Hadoop C.	0.64	102%	**0.58**	**92%**	**0.61**	**97%**	0.63
	HBase	0.69	100%	**0.68**	**99%**	**0.68**	**99%**	0.69
	Mahout	0.73	101%	0.72	100%	0.73	101%	0.72
	OpenJPA	0.68	101%	**0.65**	**97%**	0.67	100%	0.67
	Pig	0.63	100%	**0.56**	**89%**	**0.62**	**98%**	0.63
	Tuscany	**0.69**	**97%**	**0.70**	**99%**	0.71	100%	0.71
Avg.		0.67	100%	0.65	97%	0.66	99%	0.67

cases. On average across the ten projects, the AUC of the two models is not lower than that of the RA model. On the other hand, the AG model shows at least a 3% of performance reduction in several cases. On average across the ten projects, the AUC score of the AG model is 98%–99% of that of the RA model.

In terms of F1-score, we notice that the B and MA models show around a 5% performance reduction in most cases. For the B or MA model, the average F1-score across the ten projects is 97%–100% of that of the RA model. On the other hand, the AG model shows at least a 8% performance reduction in several cases. The average F1-score of the AG model is 95%–97% of that of the RA model. Tables 8 and 9 show that for the B, AG and MA models, the percentage of performance reduction in terms of F1-score is larger than that in term of AUC. However, it does not mean that F1-score of the JIT models is more likely to be impacted by the mislabeled changes than AUC. By observing Tables 8 and 9, we find that the F1-scores of the RA model are much smaller than its AUC scores. Hence, the percentage of performance reduction (in terms of F1-score) is likely to be inflated, because the percentage change for smaller numbers tend to be higher than the percentage change for greater numbers even when the absolute difference is the same. For instance, when using naive Bayes as the underlying classifier, the B and RA models achieve an F1-score of 0.17 and 0.18 on our Hadoop Common dataset. The absolute difference between the two scores is 0.01 but the percentage of performance reduction is 6%. For each case where the B, AG or MA model

TABLE 11: Adjusted P-values and Cliff's delta comparing AUC scores for the B, AG and MA models with those of the RA model. P-values and Cliff's delta that show significant performance reduction are in bold.

Cls	Project	B		AG		MA	
RF	ActiveMQ	**-0.38**	**(M)***	**-0.91**	**(L)***	-0.02	(N)
	Camel	**-0.61**	**(L)***	**-1.00**	**(L)***	-0.08	(N)***
	Derby	**-0.43**	**(M)***	**-0.93**	**(L)***	**-0.55**	**(L)***
	Geronimo	0.05	(N)	**-0.98**	**(L)***	0.05	(N)
	Hadoop C.	**-0.87**	**(L)***	**-0.97**	**(L)***	0.03	(N)
	HBase	**-0.96**	**(L)***	**-1.00**	**(L)***	**-0.50**	**(L)***
	Mahout	0.26	(S)	**-0.38**	**(M)***	0.10	(N)
	OpenJPA	-0.03	(N)	**-0.96**	**(L)***	**-0.17**	**(S)***
	Pig	**-0.15**	**(S)***	**-0.49**	**(L)***	0.10	(N)
	Tuscany	**-0.18**	**(S)***	**-0.88**	**(L)***	0.01	(N)
LR	ActiveMQ	0.05	(N)	**-0.45**	**(M)***	0.02	(N)
	Camel	**-0.17**	**(S)***	**-0.82**	**(L)***	0.00	(N)
	Derby	**-0.20**	**(S)***	**-0.25**	**(S)***	**-0.20**	**(S)***
	Geronimo	0.07	(N)	**-0.84**	**(L)***	-0.03	(N)***
	Hadoop C.	0.89	(L)	**-0.88**	**(L)***	-0.02	(N)*
	HBase	0.00	(N)	**-0.37**	**(M)***	-0.08	(N)***
	Mahout	0.08	(N)	**-0.29**	**(S)***	0.02	(N)
	OpenJPA	0.01	(N)	**-0.64**	**(L)***	0.05	(N)
	Pig	0.13	(N)	**-0.32**	**(S)***	-0.06	(N)***
	Tuscany	-0.12	(N)***	**-0.68**	**(L)***	-0.02	(N)***
NB	ActiveMQ	0.17	(S)	**-0.46**	**(M)***	0.10	(N)
	Camel	0.67	(L)	**-0.96**	**(L)***	**-0.18**	**(S)***
	Derby	-0.11	(N)***	**-0.22**	**(S)***	-0.44	(M)***
	Geronimo	0.29	(S)	**-0.72**	**(L)***	0.06	(N)
	Hadoop C.	0.01	(N)	**-0.10**	**(N)***	-0.04	(N)***
	HBase	0.13	(N)	**-0.44**	**(M)***	**-0.15**	**(S)***
	Mahout	0.15	(N)	**-0.29**	**(S)***	0.05	(N)
	OpenJPA	0.27	(S)	**-0.65**	**(L)***	0.00	(N)
	Pig	0.02	(N)	**-0.43**	**(M)***	-0.07	(N)***
	Tuscany	**-0.55**	**(L)***	0.01	(N)	-0.11	(N)***

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

TABLE 12: Adjusted P-values and Cliff's delta comparing F1-scores for the B, AG and MA models with those of the RA model. P-values and Cliff's delta that show significant performance reduction are in bold.

Cls	Project	B		AG		MA	
RF	ActiveMQ	**-0.52**	**(L)*****	**-0.72**	**(L)*****	-0.02	(N)
	Camel	**-0.81**	**(L)*****	**-0.97**	**(L)*****	-0.14	(N)***
	Derby	**-0.75**	**(L)*****	**-0.95**	**(L)*****	**-0.60**	**(L)*****
	Geronimo	**-0.28**	**(S)*****	**-0.82**	**(L)*****	-0.14	(N)***
	Hadoop C.	**-0.43**	**(M)*****	**-0.92**	**(L)*****	-0.09	(N)***
	HBase	**-0.96**	**(L)*****	**-0.99**	**(L)*****	**-0.69**	**(L)*****
	Mahout	**-0.16**	**(S)*****	**-0.19**	**(S)*****	0.00	(N)
	OpenJPA	**-0.16**	**(S)*****	**-0.89**	**(L)*****	**-0.28**	**(S)*****
	Pig	**-0.21**	**(S)*****	**-0.44**	**(M)*****	-0.11	(N)***
	Tuscany	**-0.29**	**(S)*****	**-0.47**	**(M)*****	-0.11	(N)***
LR	ActiveMQ	**-0.26**	**(S)*****	**-0.46**	**(M)*****	-0.06	(N)***
	Camel	**-0.58**	**(L)*****	**-0.73**	**(L)*****	-0.10	(N)***
	Derby	**-0.55**	**(L)*****	**-0.69**	**(L)*****	**-0.42**	**(M)*****
	Geronimo	**-0.19**	**(S)*****	**-0.17**	**(S)*****	-0.13	(N)***
	Hadoop C.	0.69	(L)	**-0.87**	**(L)*****	-0.07	(N)***
	HBase	**-0.66**	**(L)*****	**-0.32**	**(S)*****	**-0.46**	**(M)*****
	Mahout	**-0.22**	**(S)*****	**-0.24**	**(S)*****	-0.02	(N)
	OpenJPA	-0.06	(N)***	**-0.26**	**(S)*****	0.06	(N)
	Pig	-0.03	(N)	**-0.29**	**(S)*****	-0.11	(N)***
	Tuscany	**-0.19**	**(S)*****	**-0.26**	**(S)*****	-0.05	(N)***
NB	ActiveMQ	**-0.29**	**(S)*****	**-0.15**	**(S)*****	-0.04	(N)***
	Camel	0.68	(L)	**-0.92**	**(L)*****	**-0.33**	**(M)*****
	Derby	-0.01	(N)	**-0.69**	**(L)*****	**-0.91**	**(L)*****
	Geronimo	**-0.21**	**(S)*****	0.99	(L)	-0.11	(N)***
	Hadoop C.	0.00	(N)	**-0.41**	**(M)*****	**-0.36**	**(M)*****
	HBase	0.08	(N)	**-0.32**	**(S)*****	**-0.26**	**(S)*****
	Mahout	-0.03	(N)	**-0.24**	**(S)*****	0.00	(N)
	OpenJPA	0.44	(M)	**-0.32**	**(S)*****	0.12	(N)
	Pig	-0.05	(N)**	**-0.62**	**(L)*****	-0.09	(N)***
	Tuscany	**-0.19**	**(S)*****	0.06	(N)	-0.08	(N)***

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

TABLE 13: Adjusted P-values and Cliff's delta comparing Gmean scores for the B, AG and MA models with those of the RA model. P-values and Cliff's delta that show significant performance reduction are in bold.

Cls	Project	B		AG		MA	
RF	ActiveMQ	0.04	(N)	**-0.78**	**(L)*****	0.11	(N)
	Camel	**-0.22**	**(S)*****	**-0.99**	**(L)*****	0.02	(N)
	Derby	-0.07	(N)***	**-0.68**	**(L)*****	**-0.21**	**(S)*****
	Geronimo	0.40	(M)	**-0.93**	**(L)*****	0.16	(S)
	Hadoop C.	**-0.29**	**(S)*****	**-0.86**	**(L)*****	0.24	(S)
	HBase	**-0.79**	**(L)*****	**-1.00**	**(L)*****	**-0.36**	**(M)*****
	Mahout	0.25	(S)	**-0.27**	**(S)*****	0.13	(N)
	OpenJPA	0.04	(N)	**-0.87**	**(L)*****	-0.06	(N)*
	Pig	-0.03	(N)	**-0.42**	**(M)*****	0.08	(N)
	Tuscany	0.19	(S)	**-0.78**	**(L)*****	0.03	(N)
LR	ActiveMQ	0.06	(N)	**-0.42**	**(M)*****	0.02	(N)
	Camel	-0.07	(N)***	**-0.70**	**(L)*****	0.05	(N)
	Derby	-0.13	(N)***	**-0.37**	**(M)*****	**-0.21**	**(S)*****
	Geronimo	0.14	(N)	**-0.83**	**(L)*****	0.01	(N)
	Hadoop C.	0.95	(L)	**-0.86**	**(L)*****	0.26	(S)
	HBase	0.04	(N)	**-0.26**	**(S)*****	-0.08	(N)***
	Mahout	0.02	(N)	**-0.41**	**(M)*****	0.00	(N)
	OpenJPA	0.00	(N)	**-0.40**	**(M)*****	0.11	(N)
	Pig	0.05	(N)	**-0.33**	**(M)*****	-0.08	(N)***
	Tuscany	**-0.38**	**(M)*****	**-0.66**	**(L)*****	-0.02	(N)
NB	ActiveMQ	**-0.65**	**(L)*****	-0.08	(N)***	**-0.15**	**(S)*****
	Camel	0.70	(L)	**-0.96**	**(L)*****	**-0.33**	**(M)*****
	Derby	**-0.29**	**(S)*****	**-0.26**	**(S)*****	**-0.62**	**(L)*****
	Geronimo	**-0.25**	**(S)*****	0.54	(L)	-0.13	(N)***
	Hadoop C.	0.02	(N)	**-0.43**	**(M)*****	**-0.37**	**(M)*****
	HBase	0.11	(N)	**-0.39**	**(M)*****	**-0.30**	**(S)*****
	Mahout	0.07	(N)	-0.11	(N)***	0.10	(N)
	OpenJPA	0.12	(N)	**-0.49**	**(L)*****	0.07	(N)
	Pig	-0.03	(N)	**-0.73**	**(L)*****	-0.10	(N)***
	Tuscany	**-0.66**	**(L)*****	**-0.43**	**(M)*****	-0.06	(N)**

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

shows a statistically significant performance reduction, we also calculate the absolute difference between the F1-scores of the model and the RA model. For the B and MA models, the absolute difference is 0.01 in most of the cases. Such differences are unlikely to be considered as substantially large for practitioners. On the other hand, for the AG model, the absolute difference ranges from 0.01 to 0.04.

In terms of G-mean, we notice that the B model shows less than a 1% of performance reduction in most cases. On average across the ten projects, the G-mean score of the B model is not lower than that of the RA model. The MA model also shows less than a 1% of performance reduction in most cases. For most of the cases where the MA model shows a significant performance reduction, the absolute difference of the G-mean scores between the MA and RA models is 0.01. The average G-mean score of the MA model is 99%–100% of that of the RA model. On the other hand, the AG model shows at least a 3% of performance reduction in several cases. For the cases where the AG model shows a significant performance reduction, the absolute difference of G-mean scores between the AG and RA models ranges from 0.01 to 0.05. On average across the ten projects, the G-mean score of the AG model is 97%–99% of that of the RA model.

In summary, for JIT models that are built using re-balanced data, the mislabeled changes by B-SZZ and MA-SZZ are unlikely to cause a considerable performance reduction in terms of AUC, F1-score and G-mean score. The mislabeled changes by AG-SZZ cause a statistically significant performance reduction with an average difference of 1%–2%, 3%–5% and 1%–3% in terms of AUC, F1-score and G-mean score, respectively.

4.3 RQ3: How do mislabeled changes by SZZ impact the practical usage of JIT defect prediction models?

Motivation: The difference of performance measures (e.g., AUC) as shown in RQ1 and RQ2 may not be able to reflect how the mislabeled changes by SZZ impact the practical usage of JIT models. In this research question, we investigate the impact of the mislabeled changes on the practical usage of JIT models.

Approach: We apply the JIT models trained using the re-balanced data to conduct our analysis. These models can be used in practice since the class imbalance problem has been mitigated, and the models do not favor the major class (i.e., clean changes).

We calculate the number of incorrectly predicted changes (including false positives and false negatives) in the prediction of the B, AG, MA and RA models in each bootstrap iteration. Notice that in this context, a false positive refers to a change that is truly clean but predicted as bug-introducing, while a false negative refers to a change that is truly bug-introducing but predicted as clean. Moreover, considering that developers can inspect all changes that are predicted as bug-introducing, their inspection effort on false positives is wasted. We investigate whether the mislabeled changes by SZZ cause more wasted effort. In addition, we also consider developers' overall inspection effort on the changes that are predicted as bug-introducing. We assume that developers are more likely to favor a model that requires less overall inspection effort.

TABLE 14: Number of false positives and false negatives in the prediction of the B, AG, MA and RA models

Cls.	Project	#False Positive				#False Negative			
		B	AG	MA	RA	B	AG	MA	RA
RF	ActiveMQ	574	495	501	487	139	171	150	154
	Camel	1,264	1,154	1,123	1,097	183	238	196	200
	Derby	797	819	736	655	126	136	136	142
	Geronimo	1,084	986	1,029	987	190	241	202	210
	Hadoop C.	1,591	1,700	1,584	1,536	144	154	134	142
	HBase	991	871	924	837	252	316	251	260
	Mahout	185	159	166	160	39	49	44	46
	OpenJPA	452	468	458	428	85	101	86	88
	Pig	220	213	218	202	54	60	54	57
	Tuscany	1,612	1,474	1,543	1,516	157	188	165	167
Avg.		877	834	828	791	137	165	142	147
LR	ActiveMQ	723	690	667	650	121	136	132	136
	Camel	1,594	1,518	1,465	1,430	202	227	217	222
	Derby	981	994	921	853	123	126	133	138
	Geronimo	1,567	1,315	1,524	1,482	185	241	193	199
	Hadoop C.	2,592	2,644	2,628	2,515	144	201	167	181
	HBase	1,260	1,060	1,146	1,028	282	331	309	331
	Mahout	204	181	187	185	40	48	43	43
	OpenJPA	667	646	653	650	85	92	85	86
	Pig	289	287	287	279	51	55	52	53
	Tuscany	2,032	1,969	2,056	2,044	148	160	140	141
Avg.		1,191	1,130	1,153	1,112	138	162	147	153
NB	ActiveMQ	1,585	1,492	1,519	1,497	53	68	59	63
	Camel	1,750	2,434	2,185	2,025	201	184	171	184
	Derby	748	1,109	1,342	825	169	121	100	153
	Geronimo	2,388	1,326	2,331	2,270	141	265	147	153
	Hadoop C.	4,707	5,575	5,247	4,778	113	87	95	107
	HBase	1,684	1,715	1,797	1,701	225	239	212	225
	Mahout	140	147	140	134	57	58	57	59
	OpenJPA	604	778	739	785	90	81	74	69
	Pig	486	613	493	472	33	24	32	35
	Tuscany	1,707	1,718	1,873	1,847	208	201	182	182
Avg.		1,580	1,691	1,767	1,633	129	133	113	123

Hence, we also investigate whether the mislabeled changes by SZZ cause more overall effort. We use number of lines of code (LOC) as a proxy to measure developers' inspection effort following prior studies [34], [37]. In each bootstrap iteration, we calculate the wasted effort and overall effort for the B, AG, MA and RA models. Afterwards, we calculate the average number of false positives, false negatives, wasted effort and overall effort (measured by LOC) across the 1,000 bootstrap iterations, and we compare the results of the B, AG and MA models with the RA model. Furthermore, we apply the Wilcoxon signed-rank test with a Bonferroni correction to investigate whether the wasted effort and overall effort in the prediction of the B, AG and MA models is significantly larger than that of the RA model. We also calculate the Cliff's delta.

Results: Table 14 presents the number of false positives, false negatives in the prediction of the B, AG, MA and RA models. Table 15 presents developers' wasted effort and overall effort (measured by LOC) in the prediction of the B, AG, MA and RA models. And Table 16 presents the adjusted p-values and Cliff's delta for comparing the wasted effort and overall effort of the B, AG and MA models with those of the RA model.

From Table 14, we observe that on average across the ten projects, the B model leads to more false positives and less false negatives than the RA model when using random forest or logistic regression as the underlying classifier. And the B model leads to less false positives and more false negatives than the RA model when naive Bayes is used. Furthermore, we find that the MA model leads to more false positives and less false negatives than the RA model. The RA model cannot be deemed as better than the B or MA models when it comes to the number of both false positives and false negatives. On the other hand, the AG model leads to 18–58 more false positives and 9–18 more false negatives than the RA model. Hence, the RA model is better than the AG model for the number of both false positives and false negatives. Thus, when considering incorrectly predicted changes, the RA model may not be better than the B and MA model, but it is better than the AG model.

Furthermore, from Tables 15 and 16, we have the following findings:

- When using the B model, on average, developers inspect 31,822–41,582 more false positive lines (i.e., the wasted effort) than when using the RA model. The additional wasted effort is 9%–10% of the wasted effort of the RA model. And on average, the B model requires developers to inspect 36,499–46,712 more lines on the changes that are predicted as bug-introducing than the RA model. The additional effort is 7% of the overall effort required by the RA model. Statistical tests show that the wasted effort of the B model is significantly larger than the RA model on 7–10 projects and the overall effort of the B model is significantly larger than the RA model on 8–10 projects.
- When using the AG model, on average, developers inspect 2,425–58,151 more false positive lines (i.e., the wasted effort) than when using the RA model. The additional wasted effort is 1%–15% of the wasted effort of the RA model. The overall effort of the AG model is not larger than that of the RA model when using a random forest classifier. When using logistic regression or naive Bayes as the underlying classifier, on average, the AG model requires developers to inspect 11,867–60,702 more lines on the changes that are predicted as bug-introducing than the RA model. The additional effort is 2%–10% of the overall effort required by the RA model. Statistical tests show that the wasted effort of the AG model is significantly larger than the RA model on 4–5 projects, and the overall effort of the AG model is significantly larger than the RA model on 3–4 projects.
- When using the MA model, developers' wasted effort is not larger than when using the RA model if a random forest or logistic regression classifier is used. When using naive Bayes, on average, the MA model leads to 4,153 more inspected lines that are wasted compared to using the RA model. The additional wasted effort is 1% of the wasted effort of the RA model. And the overall effort of the MA model is not larger than that of the RA model when using a random forest or logistic regression classifier. When using naive Bayes, on average, the MA model requires developers to inspect 3,785 more lines on the changes that are predicted as bug-introducing than the RA model, and the additional effort is less than 1% of the overall effort of the RA model. Statistical tests show that on 7–10 of the projects, the wasted effort and overall effort of the MA model are not significantly larger than the RA model.

We find that the prediction performance of the B model is not lower than the RA model but using the B model wastes considerably more effort than the RA model. In addition, the B model requires more overall inspection effort on the

TABLE 15: Developers' wasted effort and overall effort (measured by LOC) in the prediction of the B, AG, MA and RA models

Cls.	Project	Wasted Effort (#LOC)				Overall Effort (#LOC)			
		B	AG	MA	RA	B	AG	MA	RA
RF	ActiveMQ	118,141	91,240	111,859	110,221	228,584	186,914	221,310	219,279
	Camel	296,535	276,205	282,136	282,275	541,864	504,564	526,059	526,241
	Derby	384,098	347,884	287,713	305,590	619,422	566,718	503,230	530,573
	Geronimo	407,898	458,425	395,386	382,093	708,572	745,592	691,214	677,544
	Hadoop C.	759,906	585,724	660,650	704,120	1,014,051	812,657	901,905	949,769
	HBase	335,923	294,304	316,208	310,688	776,534	693,529	753,317	747,503
	Mahout	75,960	74,630	74,764	71,651	187,409	181,593	185,468	181,523
	OpenJPA	155,668	147,824	144,915	140,401	261,515	247,752	249,389	244,778
	Pig	90,009	82,352	87,083	86,614	194,662	181,688	189,871	188,663
	Tuscany	959,855	931,429	861,554	872,118	1,295,166	1,245,632	1,183,909	1,196,918
Avg.		358,399	329,002	322,227	326,577	582,778	536,664	540,567	546,279
LR	ActiveMQ	176,539	152,056	161,043	160,001	298,646	267,988	280,380	279,085
	Camel	369,020	354,584	353,733	354,233	621,230	598,878	602,750	603,373
	Derby	479,157	448,953	363,385	377,776	727,269	687,399	591,797	609,799
	Geronimo	688,260	719,702	651,383	655,038	1,002,095	1,029,585	961,745	965,848
	Hadoop C.	1,146,744	1,039,812	1,036,366	1,061,585	1,415,719	1,289,032	1,292,703	1,318,654
	HBase	418,393	394,750	397,958	395,382	868,931	837,288	841,587	838,366
	Mahout	96,387	86,344	91,457	91,969	210,529	198,107	204,806	205,360
	OpenJPA	225,030	204,642	198,338	196,704	336,258	315,107	310,134	307,635
	Pig	111,549	96,072	101,536	103,085	219,029	200,959	208,282	209,887
	Tuscany	1,349,123	1,286,330	1,253,440	1,248,603	1,703,008	1,629,915	1,602,233	1,597,581
Avg.		506,020	478,325	460,864	464,438	740,271	705,426	689,642	693,559
NB	ActiveMQ	106,579	98,801	102,089	100,175	201,555	189,052	196,053	193,600
	Camel	330,336	325,546	318,628	315,623	569,571	550,500	549,230	546,891
	Derby	426,231	365,743	321,087	297,190	653,863	585,255	529,322	510,407
	Geronimo	383,510	566,876	375,927	366,847	661,893	859,558	653,323	642,412
	Hadoop C.	1,238,411	1,305,400	1,266,688	1,245,824	1,493,424	1,563,709	1,520,122	1,499,343
	HBase	337,962	332,973	336,829	333,343	763,718	750,355	755,276	750,747
	Mahout	80,573	75,794	76,658	73,583	189,552	182,062	184,815	180,399
	OpenJPA	200,902	182,163	179,022	177,072	310,103	287,600	286,487	282,173
	Pig	108,912	108,498	109,855	109,127	211,988	208,510	212,011	211,463
	Tuscany	1,029,105	1,117,401	852,437	878,907	1,355,586	1,442,735	1,163,526	1,194,881
Avg.		424,252	447,920	393,922	389,769	641,125	661,934	605,017	601,232

changes that are predicted as bug-introducing. To explain it, we combine the data of the ten projects and perform analysis on the combined data. Using Wilcoxon rank sum test [48], we find that the churn size of the changes mislabeled as bug-introducing by B-SZZ (i.e., RA-SZZ labels them as clean) is statistically larger than that of the changes mislabeled as clean (i.e., RA-SZZ labels them as bug-introducing) with p-value < 0.001. Hence, B-SZZ mislabels larger changes as bug-introducing. The B model trained using such data will be more likely to predict large changes as bug-introducing compared to the RA model. Hence, the B model requires more inspection effort and developers waste more effort on the false positives when using the B model.

In summary, when considering incorrectly predicted changes, the mislabeled changes by B-SZZ and MA-SZZ may not make it worse for the use of JIT models, while the mislabeled changes by AG-SZZ make it worse for the use of JIT models. On the other hand, when considering developers' inspection effort (measured by LOC), the mislabeled changes by B-SZZ lead to significantly more wasted effort with an average difference of 9%–10%, and they also lead to significantly more overall inspection effort. The mislabeled changes by AG-SZZ lead to 1%–15% more wasted effort, and they may lead to more overall inspection effort. For several projects, the mislabeled changes by AG-SZZ lead to significantly more wasted effort and overall effort. The mislabeled changes by MA-SZZ are unlikely to cause significantly more wasted effort and overall effort.

4.4 RQ4: How do mislabeled changes by SZZ impact on the performance of effort-aware JIT defect prediction models?

Motivation: Considering the limited budget in practice, developers can only inspect a limited number of lines of code, and they would expect to find as many bug-introducing changes as possible while exerting the same effort [9], [10]. Prior studies have leveraged effort-aware JIT defect prediction techniques to *prioritize* changes for developers to inspect aiming to find more bugs while exerting the same effort [21], [34]. In this research question, we investigate the impact of mislabeled changes by SZZ on the performance of effort-aware JIT models.

Approach: We build the B, AG, MA and RA models using Huang et al.'s CBS technique [34] and Fu et al.'s OneWay technique [21]. As described in Section 3.8, we use Recall@20% as the performance measure to evaluate the models.

Similarly to RQ1, we calculate the average Recall@20% score across the 1,000 bootstrap iterations for the B, AG, MA and RA models, and the ratios of the Recall@20% scores of the B, AG and MA models to those of the RA model. Furthermore, we apply the Wilcoxon signed-rank test to investigate whether the B, AG and MA models show statistically significant performance reduction in terms of Recall@20%. We also calculate the Cliff's delta.

Results: Table 17 presents the average Recall@20% score of

TABLE 16: Adjusted P-values and Cliff's delta comparing developers' wasted effort and overall effort (measured by LOC) in the prediction of the B, AG and MA models with those of the RA model. Developers' wasted effort values and overall effort values of the B, AG and MA models which are significantly larger than those of the RA model are in bold.

Cls.	Project	Wasted Effort						Overall Effort					
		B		AG		MA		B		AG		MA	
RF	ActiveMQ	**0.31**	**(S)*****	-0.67	(L)	0.05	(N)***	**0.27**	**(S)*****	-0.78	(L)	0.05	(N)***
	Camel	**0.44**	**(M)*****	-0.19	(S)	0.00	(N)	**0.37**	**(M)*****	-0.48	(L)	0.00	(N)
	Derby	**0.94**	**(L)*****	**0.70**	**(L)*****	-0.34	(M)	**0.91**	**(L)*****	**0.51**	**(L)*****	-0.40	(M)
	Geronimo	**0.37**	**(M)*****	**0.82**	**(L)*****	**0.18**	**(S)*****	**0.37**	**(M)*****	**0.69**	**(L)*****	**0.15**	**(S)*****
	Hadoop C.	**0.45**	**(M)*****	-0.81	(L)	-0.36	(M)	**0.47**	**(M)*****	-0.83	(L)	-0.36	(M)
	HBase	**0.50**	**(L)*****	-0.34	(M)	0.12	(N)***	**0.41**	**(M)*****	-0.69	(L)	0.09	(N)***
	Mahout	**0.26**	**(S)*****	**0.15**	**(S)*****	**0.18**	**(S)*****	**0.21**	**(S)*****	0.00	(N)	0.14	(N)***
	OpenJPA	**0.45**	**(M)*****	**0.22**	**(S)*****	0.14	(N)***	**0.39**	**(M)*****	0.07	(N)***	0.11	(N)***
	Pig	**0.15**	**(S)*****	-0.16	(S)	0.03	(N)	**0.17**	**(S)*****	-0.18	(S)	0.04	(N)
	Tuscany	**0.68**	**(L)*****	**0.45**	**(M)*****	-0.08	(N)	**0.68**	**(L)*****	**0.34**	**(M)*****	-0.09	(N)
LR	ActiveMQ	**0.45**	**(M)*****	-0.23	(S)	0.02	(N)***	**0.47**	**(M)*****	-0.26	(S)	0.03	(N)***
	Camel	**0.45**	**(M)*****	0.01	(N)	-0.02	(N)	**0.42**	**(M)*****	-0.11	(N)	-0.01	(N)
	Derby	**0.96**	**(L)*****	**0.84**	**(L)*****	-0.22	(S)	**0.95**	**(L)*****	**0.80**	**(L)*****	-0.23	(S)
	Geronimo	**0.40**	**(M)*****	**0.67**	**(L)*****	-0.04	(N)	**0.40**	**(M)*****	**0.62**	**(L)*****	-0.04	(N)
	Hadoop C.	**0.64**	**(L)*****	-0.19	(S)	-0.23	(S)	**0.67**	**(L)*****	-0.23	(S)	-0.22	(S)
	HBase	**0.45**	**(M)*****	-0.01	(N)	0.05	(N)***	**0.42**	**(M)*****	-0.02	(N)	0.05	(N)***
	Mahout	**0.21**	**(S)*****	-0.27	(S)	-0.02	(N)	**0.17**	**(S)*****	-0.23	(S)	-0.02	(N)
	OpenJPA	**0.69**	**(L)*****	**0.21**	**(S)*****	0.05	(N)***	**0.60**	**(L)*****	**0.16**	**(S)*****	0.05	(N)***
	Pig	**0.30**	**(S)*****	-0.26	(S)	-0.05	(N)	**0.24**	**(S)*****	-0.24	(S)	-0.04	(N)
	Tuscany	**0.72**	**(L)*****	**0.29**	**(S)*****	0.03	(N)*	**0.71**	**(L)*****	**0.23**	**(S)*****	0.03	(N)*
NB	ActiveMQ	**0.46**	**(M)*****	-0.10	(N)	0.14	(N)***	**0.37**	**(M)*****	-0.23	(S)	0.11	(N)***
	Camel	**0.51**	**(L)*****	**0.35**	**(M)*****	0.13	(N)***	**0.53**	**(L)*****	0.08	(N)***	0.06	(N)***
	Derby	**0.96**	**(L)*****	**0.83**	**(L)*****	**0.54**	**(L)*****	**0.95**	**(L)*****	**0.74**	**(L)*****	**0.31**	**(S)*****
	Geronimo	**0.40**	**(M)*****	**1.00**	**(L)*****	**0.23**	**(S)*****	**0.36**	**(M)*****	**1.00**	**(L)*****	**0.20**	**(S)*****
	Hadoop C.	-0.09	(N)	**0.41**	**(M)*****	**0.17**	**(S)*****	-0.07	(N)	**0.41**	**(M)*****	**0.15**	**(S)*****
	HBase	0.12	(N)***	-0.01	(N)	0.10	(N)***	**0.20**	**(S)*****	0.00	(N)	0.07	(N)***
	Mahout	**0.29**	**(S)*****	0.09	(N)***	0.13	(N)***	**0.26**	**(S)*****	0.05	(N)**	0.13	(N)***
	OpenJPA	**0.53**	**(L)*****	0.09	(N)***	0.07	(N)***	**0.54**	**(L)*****	0.09	(N)***	0.12	(N)***
	Pig	-0.01	(N)	-0.02	(N)	0.04	(N)***	0.02	(N)	-0.08	(N)	0.02	(N)***
	Tuscany	**0.80**	**(L)*****	**0.82**	**(L)*****	-0.15	(S)	**0.79**	**(L)*****	**0.80**	**(L)*****	-0.16	(S)

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

TABLE 17: Recall@20% scores of the B, AG, MA and RA models. We also show the ratios of the Recall@20% scores of the B, AG and MA models to those of the RA model.

Tech.	Project	B		AG		MA		RA
CBS	ActiveMQ	0.58	104%	0.57	102%	0.57	102%	0.56
	Camel	0.38	100%	**0.36**	**95%**	0.38	100%	0.38
	Derby	0.47	100%	0.47	100%	0.47	100%	0.47
	Geronimo	0.57	104%	**0.48**	**87%**	0.56	102%	0.55
	Hadoop C.	0.64	108%	**0.56**	**95%**	0.62	105%	0.59
	HBase	0.51	109%	0.47	100%	0.49	104%	0.47
	Mahout	0.37	100%	**0.36**	**97%**	0.37	100%	0.37
	OpenJPA	0.43	102%	**0.40**	**95%**	0.42	100%	0.42
	Pig	0.47	102%	0.46	100%	0.46	100%	0.46
	Tuscany	**0.52**	**95%**	**0.52**	**95%**	0.55	100%	0.55
Avg.		0.49	102%	0.47	98%	0.49	102%	0.48
OneWay	ActiveMQ	0.44	100%	0.44	100%	0.44	100%	0.44
	Camel	**0.32**	**97%**	0.33	100%	0.33	100%	0.33
	Derby	0.29	107%	0.28	104%	0.28	104%	0.27
	Geronimo	0.49	100%	0.49	100%	0.49	100%	0.49
	Hadoop C.	0.44	100%	0.44	100%	0.44	100%	0.44
	HBase	0.44	100%	0.44	100%	0.44	100%	0.44
	Mahout	0.35	100%	0.35	100%	0.35	100%	0.35
	OpenJPA	0.41	100%	0.41	100%	0.41	100%	0.41
	Pig	0.38	100%	0.38	100%	0.38	100%	0.38
	Tuscany	0.51	100%	0.51	100%	0.51	100%	0.51
Avg.		0.41	100%	0.41	100%	0.41	100%	0.41

TABLE 18: Adjusted P-values and Cliff's delta comparing Recall@20% scores for the B, AG and MA models with those of the RA model. P-values and Cliff's delta that show significant performance reduction are in bold.

Tech.	Project	B		AG		MA	
CBS	ActiveMQ	0.28	(S)	0.16	(S)	0.09	(N)
	Camel	0.04	(N)	**-0.53**	**(L)*****	0.19	(S)
	Derby	0.05	(N)	0.01	(N)	0.25	(S)
	Geronimo	0.45	(M)	**-0.99**	**(L)*****	0.27	(S)
	Hadoop C.	0.92	(L)	**-0.66**	**(L)*****	0.53	(L)
	HBase	0.93	(L)	-0.14	(N)***	0.61	(L)
	Mahout	0.10	(N)	**-0.22**	**(S)*****	0.07	(N)
	OpenJPA	0.17	(S)	**-0.33**	**(S)*****	0.01	(N)
	Pig	0.06	(N)	-0.05	(N)***	0.02	(N)
	Tuscany	**-0.73**	**(L)*****	**-0.67**	**(L)*****	0.08	(N)
OneWay	ActiveMQ	0.00	(N)	0.00	(N)	0.00	(N)
	Camel	-0.03	(N)***	0.03	(N)	0.01	(N)
	Derby	0.27	(S)	0.25	(S)	0.15	(N)
	Geronimo	0.00	(N)	-0.02	(N)**	0.00	(N)
	Hadoop C.	-0.01	(N)*	0.01	(N)	-0.01	(N)
	HBase	0.00	(N)	0.00	(N)	0.00	(N)
	Mahout	0.02	(N)	-0.04	(N)***	-0.01	(N)
	OpenJPA	0.00	(N)	0.01	(N)	0.01	(N)
	Pig	0.00	(N)	0.00	(N)	0.00	(N)
	Tuscany	0.00	(N)	-0.01	(N)	0.00	(N)

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

the B, AG, MA and RA models that are built using the CBS and OneWay techniques. Table 18 presents the adjusted p-values and Cliff's delta comparing Recall@20% scores for the B, AG and MA models with the RA model.

From the two tables, we have the following findings:

- For the models built using the CBS technique, the B model and MA model are not performing worse than the RA model in terms of Recall@20%. On the other hand, the Recall@20% score of the AG model is 87%–102% of that of the RA model. On average across the ten projects, the AG model shows

2% performance reduction. The AG model shows a statistically significant performance reduction with a non-negligible effect size on six projects.

- For the models built using the OneWay technique, the B, AG and MA models are not performing worse than the RA model in terms of Recall@20%.

In summary, for the effort-aware JIT models using the CBS technique, the mislabeled changes by B-SZZ and MA-SZZ are not likely to cause a considerable performance reduction in terms of Recall@20%. And the mislabeled changes by AG-SZZ cause a statistically significant performance reduction with an average difference of 2%. For the effort-aware JIT models using the OneWay technique, mislabeled changes by SZZ are unlikely to impact the performance of these models in terms of Recall@20%.

4.5 RQ5: How do mislabeled changes by SZZ impact the interpretation of JIT defect prediction models?

Motivation: In addition to predicting bug-introducing changes, prior studies also used JIT models to *understand* the impact of various metrics on the defect-proneness of changes [37], [38]. By understanding the most important metrics for identifying bug-introducing changes, developers can avoid the pitfalls that show high association with the appearance of bugs. In this research question, we investigate whether mislabeled changes by SZZ lead to conflicting conclusions about the most important metrics for identifying bug-introducing changes.

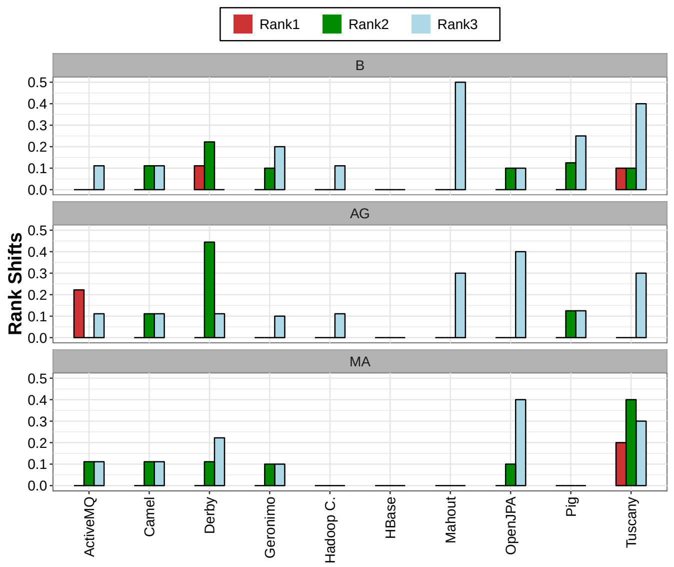
Approach: For each project, we calculate the importance ranks of the change metrics for the B, AG, MA and RA models. Then, we compare the top-3 ranked important metrics of the B, AG and MA models with those of the RA model (i.e., the baseline model). By doing so, we investigate the impact of the mislabeled changes by B-SZZ, AG-SZZ and MA-SZZ on the most important metrics for identifying bug-introducing changes.

First, in each bootstrap iteration, we take the seven steps described in Figure 5 to calculate metric importance scores for the B, AG, MA and RA models. The bootstrap iterates 1,000 times.

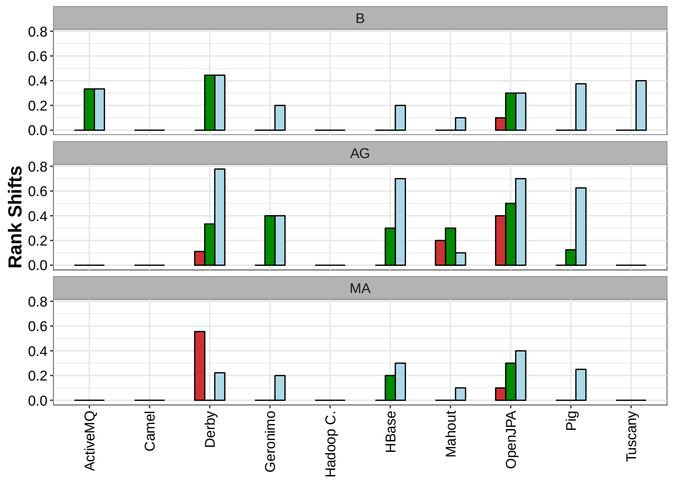
Next, following prior studies [19], [73], [79], we apply the Scott-Knott Effect Size Difference (SK-ESD) test [73] on the metric importance scores from the 1,000 iterations of bootstrap. The SK-ESD test is an enhancement of the Scott-Knott test [61]. Differently from the Scott-Knott test, the SK-ESD test mitigates the skewness of input data to relax the assumption of normally distributed data (which is required by Scott-Knott test) [73]. Moreover, the SK-ESD test considers the effect size of input data and merges any two statistically distinct groups with a negligible effect size into one group.

Finally, we compare the top-3 ranked important metrics of the B, AG and MA models with those the RA model. To do so, we apply Rajbahadur et al.'s rank shift [60].

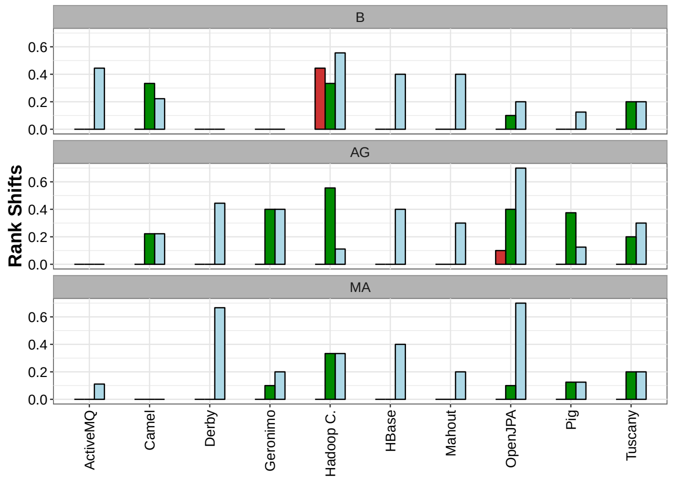
Rajbahadur et al. define a rank shift as the amount that a metric shifts its importance rank between two models regarding the total number of metrics in the dataset [60]. Suppose $V_1(k) = \{v_1, v_2, ..., v_m\}$ and $V_2(k) = \{v_1, v_2, ..., v_n\}$ are the metrics that appear at rank k for the first and second model, respectively. Let us denote the



(a) Random Forest



(b) Logistic Regression



(c) Naive Bayes

Fig. 6: Rank shifts of the top-3 ranked metrics between each of the B, AG and MA models and the RA model

TABLE 19: Average rank shifts in top-3 ranked metrics comparing the B, AG and MA models with the RA model across the ten projects. We also show the P-values for comparing the rank shifts with the ideal no shift case in which all shifts are 0. The significant rank shifts are in bold.

Cls	**Rank**	**Model B**		**Model AG**		**Model MA**	
		Avg. Shifts	P-value	Avg. Shifts	P-value	Avg. Shifts	P-value
RF	1	0.02	>0.05	0.02	>0.05	0.02	>0.05
	2	**0.08**	**<0.01**	**0.07**	**<0.05**	**0.09**	**<0.01**
	3	**0.18**	**<0.001**	**0.17**	**<0.001**	**0.12**	**<0.01**
LR	1	0.01	>0.05	**0.07**	**<0.05**	0.07	>0.05
	2	**0.11**	**<0.05**	**0.20**	**<0.01**	0.05	>0.05
	3	**0.24**	**<0.001**	**0.33**	**<0.01**	**0.15**	**<0.01**
NB	1	0.04	>0.05	0.01	>0.05	0.00	>0.05
	2	**0.10**	**<0.05**	**0.22**	**<0.01**	**0.09**	**<0.01**
	3	**0.25**	**<0.001**	**0.30**	**<0.001**	**0.29**	**<0.001**

TABLE 20: Number of projects where a metric is ranked as the top-1 and one of the top-3 important metrics.

Features	**Random Forest**				**Logistic Regression**				**Naive Bayes**			
	B	AG	MA	RA	B	AG	MA	RA	B	AG	MA	RA
Top-1 important metrics												
NS	0	0	0	0	0	0	0	0	0	0	0	0
NF	8	9	9	9	5	4	5	5	8	8	7	7
LA	1	0	1	0	0	1	0	0	1	1	1	1
LD	0	0	0	0	0	0	0	0	1	1	1	1
LT	1	0	1	0	3	2	3	3	0	0	0	0
NDEV	1	1	1	1	1	2	1	1	0	1	1	1
AGE	0	0	0	0	0	0	0	0	0	0	0	0
NUC	0	0	0	0	0	1	1	0	0	0	0	0
EXP	0	0	0	0	1	1	1	1	0	0	0	0
SEXP	0	1	0	0	1	0	1	0	0	0	0	0
Top-3 important metrics												
NS	1	2	1	1	1	2	2	2	3	3	3	2
NF	10	10	10	10	8	7	9	7	10	9	10	10
LA	9	8	10	9	3	6	3	2	8	6	6	7
LD	0	1	2	2	0	0	0	0	2	9	9	7
LT	8	7	8	7	6	6	6	7	4	4	2	5
NDEV	2	2	2	2	5	4	4	5	1	2	1	1
AGE	2	2	4	1	1	1	0	0	0	3	3	2
NUC	0	1	2	1	1	3	3	4	1	3	1	2
EXP	1	2	3	1	3	2	4	5	6	5	6	5
SEXP	1	2	3	2	4	4	5	5	1	0	2	0

number of metrics in the given dataset as N_{metric}. The rank shifts of the metrics that appear at rank k between the two models are calculated as:

$$\begin{aligned} Shifts(k) =& (\sum_{v \in V_1(k)} |k - Rank_2(v)| + \\ & \sum_{v \in V_2(k)} |k - Rank_1(v)|)/N_{metric}. \end{aligned} \tag{3}$$

In the formula, $Rank_1(v)$ and $Rank_2(v)$ refer to the rank of the metric v calculated from the first and second model, respectively. The rank shifts can quantify the difference in the important metrics between the two models for identifying bug-introducing changes.

For each of the B, AG and MA models, we calculate the rank shifts of the top-3 ranked metrics of the model and the RA model, respectively. Furthermore, to investigate whether the importance ranks of two given models are statistically significantly different, we apply the Wilcoxon signed-rank test to compare the rank shifts between the two models with the ideal no shift case in which all shifts are 0.

Using the rank shifts, we can compare the difference of metric importance between two models on each studied project. In addition, we compare the most important metrics of the four models across all studied projects. We calculate number of projects where a metric is ranked as top-1 and one of the top-3 important metrics. By comparing the numbers for each of the B, AG and MA models with the RA model, we investigate whether the mislabeled changes by B-SZZ, AG-SZZ and MA-SZZ affect the conclusions on the top-1 and top-3 important metrics for identifying bug-introducing changes.

Results: Figure 6 presents the rank difference computed by comparing top-3 important metrics of the B, AG and MA models with those of the RA model. Table 19 shows the average rank shifts for each rank and the p-values for comparing the rank shifts with the ideal no shift case. And Table 20 shows the number of projects where each of the metrics is ranked as the top-most and one of the top-3 important ones, respectively. The REXP, ND, Entropy and FIX metrics are correlated with the other ones in each project, and thus, they are dropped.

From Figure 6 and Table 19, we have the following findings:

- For top-ranked metrics, the B, AG and MA models show difference from the RA models on only 0–3 projects. The average shifts between the B, AG or MA model and the RA model range from 0 to 0.07. Moreover, statistical tests show that the top-ranked metrics are not significantly different between the B, AG or MA model and the RA model in most cases.
- For second-ranked metrics, the B, AG and MA models

show difference from the RA models on 2–6 projects. The average shifts between the B, AG or MA model and the RA model range from 0.07 to 0.22. Statistical tests show that the second-ranked metrics are significantly different between the B, AG or MA model and the RA model in most cases.

- For third-ranked metrics, the B, AG and MA models show difference from the RA models on 6–9 projects. The average rank shifts between the B, AG or MA model and the RA model range from 0.12 to 0.33. Moreover, statistical tests show that the third-ranked metrics are significantly different between the B, AG or MA model and the RA model in all cases.

From Table 20, we find that considering the top-1 most important metrics, NF (i.e., number of files modified by a change) is the dominant one. Hence, NF is the most important metric in identifying bug-introducing changes for all of the four models. Also, we notice that the top-3 important metrics for the B, AG, MA and RA models are consistent when JIT models use random forest as the underlying classifier (i.e., NF, LA and LT). However, when JIT models use logistic regression or naive Bayes as the underlying classifier, we notice that the B, AG and MA models are impacted by different important metrics as compared to the RA model. For example, when using naive Bayes as the underlying classifier, the number of projects where LD is ranked in top-3 important ones is drastically different between the B and RA models (2 vs 7).

In summary, the mislabeled changes by B-SZZ, AG-SZZ and MA-SZZ do not impact the most important metric for identifying bug-introducing changes (i.e., NF). However, metrics in the second and third ranks are more likely to be impacted by the mislabeled changes, unless random forest is used as the underlying classifier.

5 Discussion

In this section, we further discuss our experimental results and threats to the validity of our study.

5.1 Why is the performance of the B and MA models similar to that of the RA model? And why does the AG model show a significant performance reduction?

In RQ1 and RQ2 we notice that in terms of AUC, F1-score and G-mean, the performance of the B or MA model is not lower that of the RA model, whereas the AG model shows a significant performance reduction compared to the RA model. In RQ3, we have the same findings in terms of Recall@20% when the B, AG, MA and AG models use the CBS technique. In this section, we discuss the reasons as to why the two phenomena occur: 1) why is the performance of the B and MA models similar to that of the RA model? 2) why does the AG model show a significant performance reduction?

We answer these two questions by comparing the characteristics of the mislabeled changes by B-SZZ, AG-SZZ and MA-SZZ as shown in Table 4.

First, we find that the labeled data by MA-SZZ has a smaller false positive rate and false negative rate compared to the labeled data by B-SZZ and AG-SZZ. On average across the ten projects, the labeled data by MA-SZZ has a false positive rate and false negative rate of 3% and 4%, i.e., only 3% of truly clean changes and 4% of truly bug-introducing changes are mislabeled by MA-SZZ. The mislabeled changes introduced by MA-SZZ are not likely to impact the performance of the MA model.

Secondly, we find that the labeled data by AG-SZZ shows a visible difference from the labeled data by B-SZZ, which is: AG-SZZ generates much more false negatives than B-SZZ. On average across the ten studied projects, the labeled data by AG-SZZ has a false negative rate of 30%—indicating that the AG model is learned from only 70% of truly bug-introducing changes. In contrast, the labeled data by B-SZZ has a false negative rate of 15%, i.e., the B model is learned from 85% of truly bug-introducing changes. In addition, we notice that the labeled data by B-SZZ and AG-SZZ has a small false positive rate on average across the projects (6% and 4%, respectively). Hence, false negatives have more impact on the B and AG models than false positives. And the AG model is more likely to be impacted by false negatives in comparison with the B model. For the B model, the mislabeled 15% bug-introducing changes may not have a considerable impact on the performance of the model. On the other hand, for the AG model, the mislabeled 30% bug-introducing changes are likely to introduce a bias on the prediction of the AG model, which results in a significant performance reduction.

In summary, the labeled data by B-SZZ and MA-SZZ has a relatively low false positive rate and false negative rate, which may not considerably impact the prediction of the B and MA models. On the other hand, the labeled data by AG-SZZ contains a much larger number of false negatives which pertain to 30% of truly bug-introducing changes. These false negatives may have a considerably negative impact on the performance of the AG model.

5.2 Threats to validity

Internal Validity. Threats to internal validity are concerned with potential errors in our code implementation and study settings. Our code has been double-checked, but there may still exist errors that we did not note.

It is very challenging to retrieve the truly clean data containing no false positives and false negatives. Among the four studied SZZ variants, RA-SZZ is less likely to generate wrong labels compared to the other SZZ variants. Hence, in this study, we evaluate all JIT models on the labeled data by RA-SZZ. We use the model trained using the labeled data by RA-SZZ as the baseline model and compare it with models trained using the labeled data by B-SZZ, AG-SZZ and MA-SZZ. Nevertheless, RA-SZZ may still mislabel changes. For example, Neto et al. noted that RA-SZZ can only detect 13 types of refactorings due to the limitations of RefDiff tool [56], [64]. The undetected refactorings may lead to false positives and false negatives. Neto et al. studied the same datasets. They manually analyzed 365 buggy lines detected by RA-SZZ and found that 8.49% of the analyzed lines are related to undetected refactorings of RefDiff. Nevertheless, our manual analysis of the refactorings detected by RefDiff shows that RefDiff achieves nearly perfect precision (98.1%). Therefore, although the undetected refactorings may limit the results of RA-SZZ, RA-SZZ still shows an advantage over the other three SZZ variants by dealing with the detected refactorings.

In addition, Soares et al. observed that in specific situations, refactoring attempts may be defective [66], [67]. Bavota et al. also observed that refactorings such as those involving hierarchies (e.g., pull up method) can induce faults [5]. RA-SZZ cannot handle such defective refactorings, since it assumes that refactoring changes should not introduce bugs (as defined by Fowler et al. [20]). Still, Soares et al. observed that less than 1% of refactoring attempts introduce bugs [66]. Hence, defective refactorings are not likely to heavily impact the labels generated by RA-SZZ.

Furthermore, in this study, we use the data in the JIRA ITS to deal with the missing link problem that is reported in the Bugzilla ITS by Bird et al. and Bachmann et al. [4], [6]. We may still find missing links in the JIRA ITS, e.g., due to developers' occasional typo errors of entering bug identifiers in the commit messages. The missing links in our data may still induce a bias. However, Bissyande et al. have shown that the quality of links in the JIRA ITS is much better than that of Bugzilla ITS [7]. Thus, the bias induced by the missing links is unlikely to have a considerable impact on our models compared to the bias induced by the missing links in the Bugzilla ITS.

External Validity. Threats to external validity are concerned with the generality of our conclusions. In our study, we conduct our experiments on ten Apache open source projects which contains a total of 126,526 changes. The ten projects are from different application domains. Also, the projects have different sizes. We have investigated the impact of mislabeled changes by SZZ when JIT models are trained in different settings including models trained using the original data and re-balanced data. We also investigate the impact of the mislabeled changes on effort-aware JIT models. We have consistent conclusions about the impact of the mislabeled changes on the performance of JIT models in different settings.

Nevertheless, we may still find that the impact of mislabeled changes by SZZ can be higher or lower in different projects with different programming languages. Therefore, additional replication studies are needed to verify our results on more projects.

Construct Validity. Threats to construct validity relate to the suitability of our performance measures for evaluating prediction performance of JIT defect prediction models. We use AUC, F1-score and G-mean to evaluate the prediction performance of JIT models. And we use Recall@20% to evaluate the cost-effectiveness of effort-aware JIT models. These performance measures are widely used in prior studies [21], [34]–[37], [50], [68], [83]. Furthermore, we leverage statistical tests to ensure that our conclusions are robust.

6 Conclusion

In this paper, we investigate the impact of mislabeled changes by SZZ variants on Just-in-Time defect prediction models in terms of model performance and model interpretation. We analyze four SZZ variants that are proposed by prior studies, namely B-SZZ, AG-SZZ, MA-SZZ and RA-SZZ. The evaluation of SZZ variants in prior studies and our manual analysis of RefDiff indicate that RA-SZZ generates the cleanest data compared the other SZZ variants. To conduct our analysis, we evaluate all JIT models on the labeled data by RA-SZZ. Furthermore, we consider the model trained using the labeled data by RA-SZZ as the baseline model. By comparing models trained using the labeled data by B-SZZ, AG-SZZ and MA-SZZ with the baseline model, we investigate the impact of the mislabeled changes by B-SZZ, AG-SZZ and MA-SZZ on JIT models. In terms of various performance measures (AUC, F1-score, G-mean and Recall@20%), the mislabeled changes by B-SZZ and MA-SZZ are unlikely to cause a considerable performance reduction, while the mislabeled changes by AG-SZZ cause a significant performance reduction with an average difference ranging from 1%-5%. When considering developers' inspection effort (measured by LOC) in practice, the mislabeled changes by B-SZZ and AG-SZZ lead to 9%–10% and 1%–15% more wasted effort compared to using RA-SZZ, respectively. The mislabeled changes by B-SZZ significantly increase developers' wasted effort. In terms of model interpretation, the top-most important metric for identifying bug-introducing changes (i.e., NF) is robust towards the mislabeling noise generated by SZZ, but the second- and third-most important metrics are likely to be impacted by the mislabeling noise, unless random forest is used as the underlying classifier.

Many existing JIT defect prediction studies have applied the original SZZ (B-SZZ) to label data. In this study, we show that the mislabeled changes by B-SZZ lead to additional wasted inspection effort, and that they may impact the interpretation of JIT models. Hence, prior studies should be revisited to determine whether their JIT models have wasted considerably more inspection effort in practice and whether the interpretation of their models is impacted. Future studies should avoid using AG-SZZ since the mislabeled changes by AG-SZZ can cause a significant performance reduction, and they may considerably increase wasted inspection effort. In addition, using the labeled data by MA-SZZ does not cause a significant performance reduction or more wasted inspection effort. Hence, MA-SZZ may be an alternative for RA-SZZ when refactoring detection tools are not available (e.g., projects that are written in programming languages other than Java). In this case, practitioners can use a random forest classifier as the underlying JIT classifier to deal with the impact of the mislabeled changes by MA-SZZ on the interpretation of the JIT models.

References

[1] Activemq/amq-1381. https://issues.apache.org/jira/browse/AMQ-1381. Accessed: 2018-09-18.

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

[3] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc. Is it a bug or an enhancement?: a text-based approach to classify change requests. In *Proceedings of the 18th conference of the center for advanced studies on collaborative research: meeting of minds*, page 23. ACM, 2008.

[4] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein. The missing links: bugs and bug-fix commits. In *Proceedings of the 18th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 97–106. ACM, 2010.

[5] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo. When does a refactoring induce bugs? an empirical study. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pages 104–113. IEEE, 2012.

[6] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 121–130. ACM, 2009.

[7] T. F. Bissyande, F. Thung, S. Wang, D. Lo, L. Jiang, and L. Reveillere. Empirical evaluation of bug linking. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering*, pages 89–98. IEEE, 2013.

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

[9] G. Canfora, A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella. Multi-objective cross-project defect prediction. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 252–261. IEEE, 2013.

[10] G. Canfora, A. D. Lucia, M. D. Penta, R. Oliveto, A. Panichella, and S. Panichella. Defect prediction as a multiobjective optimization problem. *Software Testing, Verification and Reliability*, 25(4):426–459, 2015.

[11] C. Catal. Software fault prediction: A literature review and current trends. *Expert systems with applications*, 38(4):4626–4636, 2011.

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

[13] D. A. da Costa. Ma-szz implementation for svn repositories. https://github.com/danielcalencar/ma-szz, 2019.

[14] D. A. da Costa. Refactoring-aware szz (ra-szz) implementation. https://github.com/danielcalencar/ra-szz, 2019.

[15] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering*, 43(7):641–657, 2017.

[16] S. Davies, M. Roper, and M. Wood. Comparing text-based and dependence-based approaches for determining the origins of bugs. *Journal of Software: Evolution and Process*, 26(1):107–139, 2014.

[17] P. Domingos and M. Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine learning*, 29(2-3):103–130, 1997.

[18] B. Efron and R. J. Tibshirani. *An introduction to the bootstrap*. CRC press, 1994.

[19] Y. Fan, X. Xia, D. Lo, and A. E. Hassan. Chaff from the wheat: Characterizing and determining valid bug reports. *IEEE Transactions on Software Engineering*, 2018.

[20] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[21] W. Fu and T. Menzies. Revisiting unsupervised learning for defect prediction. In *Proceedings of the 25th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 72–83. ACM, 2017.

[22] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi. An empirical study of just-in-time defect prediction using cross-project models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 172–181. ACM, 2014.

[23] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall. Method-level bug prediction. In *Proceedings of the 6th international symposium on Empirical software engineering and measurement*, pages 171–180. ACM, 2012.

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

[25] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2012.

[26] M. Hamill and K. Goseva-Popstojanova. Common trends in software fault and failure data. *IEEE Transactions on Software Engineering*, 35(4):484–496, 2009.

[27] F. E. Harrell. *Regression modeling strategies: with applications to linear models, logistic regression, and survival analysis*. Springer, 2001.

[28] F. E. Harrell Jr. *rms: Regression Modeling Strategies*, 2019. R package version 5.1-3.

[29] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE, 2009.

[30] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *Proceedings of the 35th international conference on software engineering*, pages 392–401. IEEE, 2013.

[31] K. Herzig, S. Just, and A. Zeller. The impact of tangled code changes on defect prediction models. *Empirical Software Engineering*, 21(2):303–336, 2016.

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

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

[34] Q. Huang, X. Xia, and D. Lo. Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction. In *Proceedings of the 33rd International Conference on Software Maintenance and Evolution*, pages 159–170. IEEE, 2017.

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

[36] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering*, 21(5):2072–2106, 2016.

[37] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013.

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

[39] S. Kim, H. Zhang, R. Wu, and L. Gong. Dealing with noise in defect prediction. In *Proceeding of the 33rd international conference on Software engineering*, pages 481–490. IEEE, 2011.

[40] S. Kim, T. Zimmermann, K. Pan, E. James Jr, et al. Automatic identification of bug-introducing changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 81–90. IEEE, 2006.

[41] P. S. Kochhar, F. Thung, and D. Lo. Automatic fine-grained issue report reclassification. In *Proceedings of the 19th International Conference on Engineering of Complex Computer Systems*, pages 126–135. IEEE, 2014.

[42] A. G. Koru, D. Zhang, K. El Emam, and H. Liu. An investigation into the functional form of the size-defect relationship for software modules. *IEEE Transactions on Software Engineering*, 35(2):293–304, 2009.

[43] M. Kubat, S. Matwin, et al. Addressing the curse of imbalanced training sets: one-sided selection. In *Icml*, volume 97, pages 179–186. Nashville, USA, 1997.

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

[45] H. Li, W. Shang, Y. Zou, and A. E. Hassan. Towards just-in-time suggestions for log changes. *Empirical Software Engineering*, 22(4):1831–1865, 2017.

[46] A. Liaw and M. Wiener. Classification and regression by randomforest. *R News*, 2(3):18–22, 2002.

[47] M. Majka. *naivebayes: High Performance Implementation of the Naive Bayes Algorithm*, 2019. R package version 0.9.3.

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

[49] S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, and M. Nakamura. An analysis of developer metrics for fault prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, page 18. ACM, 2010.

[50] S. McIntosh and Y. Kamei. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. *IEEE Transactions on Software Engineering*, 44(5):412–428, 2018.

[51] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann. Local versus global lessons for defect prediction and effort estimation. *IEEE Transactions on software engineering*, 39(6):822–834, 2013.

[52] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, 2010.

[53] D. Meyer, A. Zeileis, and K. Hornik. *vcd: Visualizing Categorical Data*, 2017. R package version 1.4-4.

[54] A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[55] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*, pages 181–190. ACM, 2008.

[56] E. C. Neto, D. A. da Costa, and U. Kulesza. The impact of refactoring changes on the szz algorithm: An empirical study. In *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering*, pages 380–390. IEEE, 2018.

[57] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Where the bugs are. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 86–96. ACM, 2004.

[58] R. L. Plackett. Karl pearson and the chi-squared test. *International Statistical Review/Revue Internationale de Statistique*, pages 59–72, 1983.

[59] R. Purushothaman and D. E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, 2005.

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

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

[62] E. Shihab, Z. M. Jiang, W. M. Ibrahim, B. Adams, and A. E. Hassan. Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project. In *Proceedings of the 4th International Symposium on Empirical Software Engineering and Measurement*, page 4. ACM, 2010.

[63] S. Shivaji, E. J. Whitehead, R. Akella, and S. Kim. Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering*, 39(4):552–569, 2013.

[64] D. Silva and M. T. Valente. Refdiff: detecting refactorings in version histories. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 269–279. IEEE, 2017.

[65] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the 2nd international workshop on Mining software repositories*, volume 30, pages 1–5. ACM, 2005.

[66] G. Soares, R. Gheyi, and T. Massoni. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering*, 39(2):147–162, 2013.

[67] G. Soares, R. Gheyi, D. Serey, and T. Massoni. Making program refactoring safer. *IEEE software*, 27(4):52–57, 2010.

[68] Q. Song, Y. Guo, and M. Shepperd. A comprehensive investigation of the role of imbalanced learning for software defect prediction. *IEEE Transactions on Software Engineering*, 2018.

[69] M. Tan, L. Tan, S. Dara, and C. Mayeux. Online defect prediction for imbalanced data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 99–108. IEEE, 2015.

[70] C. Tantithamthavorn and A. E. Hassan. An experience report on defect modelling in practice: Pitfalls and challenges. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 286–295. ACM, 2018.

[71] C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *arXiv preprint arXiv:1801.10269*, 2018.

[72] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto. The impact of mislabelling on the performance and interpretation of defect prediction models. In *Proceedings of the 37th International Conference on Software Engineering*, pages 812–823. IEEE, 2015.

[73] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering*, 43(1):1–18, 2017.

[74] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering*, 2018.

[75] S. Wang, D. Lo, and X. Jiang. Understanding widespread changes: A taxonomic study. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering*, pages 5–14. IEEE, 2013.

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

[77] C. Williams and J. Spacco. Szz revisited: verifying when changes induce fixes. In *Proceedings of the 2008 workshop on Defects in large software systems*, pages 32–36. ACM, 2008.

[78] C. C. Williams and J. W. Spacco. Branching and merging in the repository. In *Proceedings of the 5th international working conference on Mining software repositories*, pages 19–22. ACM, 2008.

[79] X. Xia, L. Bao, D. Lo, P. S. Kochhar, A. E. Hassan, and Z. Xing. What do developers search for on the web? *Empirical Software Engineering*, 22(6):3149–3185, 2017.

[80] X. Xia, D. Lo, S. J. Pan, N. Nagappan, and X. Wang. Hydra: Massively compositional model for cross-project defect prediction. *IEEE Transactions on software Engineering*, 42(10):977–998, 2016.

[81] X. Yang, D. Lo, X. Xia, and J. Sun. Tlel: A two-layer ensemble learning approach for just-in-time defect prediction. *Information and Software Technology*, 87:206–220, 2017.

[82] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun. Deep learning for just-in-time defect prediction. In *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 17–26. IEEE Computer Society, 2015.

[83] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 157–168. ACM, 2016.

[84] J. H. Zar. Spearman rank correlation. *Encyclopedia of Biostatistics*, 7, 2005.

[85] T. Zimmermann, S. Kim, A. Zeller, and E. J. Whitehead Jr. Mining version archives for co-changed lines. In *Proceedings of the 3rd international workshop on Mining software repositories*, pages 72–75. ACM, 2006.