The Impact of Duplicate Changes on Just-in-Time Defect Prediction

Ruifeng Duan, Haitao Xu, *Member, IEEE*, Yuanrui Fan, *Graduate Student Member, IEEE*, and Meng Yan

***Abstract*—Recently, just-in-time (JIT) defect prediction technique attracted a lot of attention. In JIT defect prediction, all branches and omitting changes outside the main branch should be considered which can significantly affect the performance of JIT defect prediction. However, there are many duplicate changes among all the branches, which are referred to as a pair of changes with identical implementation in different branches. Such changes can influence the calculation of developer experience metrics and are considered as the noisy data for JIT defect prediction. In this article, the impact of duplicate changes on JIT defect prediction is explored. An empirical study on a total of 105 828 changes from eight Apache open-source projects is given. We find that 13% of changes from different branches are duplicate among the studied projects. The duplicate changes have a great influence on the model metrics for JIT defect prediction. For 50% of the changes, removing duplicate changes decreases the experience metrics with an average of 6–55. In addition, the duplicate changes have a significant impact on the evaluation and interpretation of JIT defect prediction models. Removing duplicate changes among the studied projects can significantly improve the performance of JIT defect prediction models ranging from 1 to 125% concerning various performance measures (i.e., area under the curve, Matthews correlation coefficient, and F1). Given the impact of duplicate changes, we suggest that researchers should remove duplicate changes from the original historical changes of software repository when evaluating the performance of JIT defect prediction models in future work.**

***Index Terms*—Branches, just-in-time (JIT) defect prediction, mining software repositories, noisy data.**

I. Introduction

SOFTWARE defect prediction refers to predicting the position of potential defects in software [1], [2]. It has been a long-term research topic in the software quality assurance area [3]. Recently, several studies focus on predicting defects at the change level [4]–[8]. Such a technique is referred to as **just-in-time (JIT) defect prediction** [6]. Different from the traditional defect prediction techniques that perform at a coarse-grained level (i.e., package or file level) [9]–[12], JIT defect prediction aims to identify **bug-introducing changes**, which has many benefits such that developers can only review a pretty smaller amount of potentially defective lines.

JIT defect prediction leverages the historical changes that are stored in version control systems (VCS) of software projects to build prediction models [6]. Developers often leverage branch to manage software versions [13]. Kovalenko *et al.* [14] noted that JIT defect prediction should not only consider changes in the main branch. They proposed that omitting changes outside the main branch can significantly impact the performance of JIT defect prediction.

In our article, we observe that to fix a bug or implement a feature request, developers often need to perform the same changes to the code of different branches [15]. In this article, we define such changes that are submitted to different branches with the same modifications as duplicate changes. Specifically, duplicate changes have two characteristics: 1) they are submitted to different branches; 2) they have the same content (i.e., the same modified files and code). Duplicate changes may impact the data distribution of the training and testing data that are used in the existing JIT defect prediction studies. Consequently, duplicate changes may impact the validity of the existing studies. Fig. 1 provides a pair of duplicate changes to the project Apache/HBase.[1] In this case, changes "4499092" and "$f4860d8$" are created to fix the bug "#953." They are submitted to different branches, but they have the same change files and codes. Such changes may impact the calculation of some features (e.g., the number of changes made by the developer before the current change), which is a threat to the validity of JIT defect prediction models.

In this article, we explore the scale of duplicate changes and the impact of duplicate changes on JIT defect prediction. Following prior studies [6], [16], [17], we leverage 14 change-level features proposed by Kamei *et al.* [6]. For each project, we build a dataset that contains all changes from different branches (*referred to as In-dup change dataset*). Then, we identify duplicate changes by checking whether two changes are from different branches and modify identical lines in identical files. For each pair of duplicate changes, we remove one of the changes and build a dataset that does not contain duplicate

Manuscript received May 30, 2020; revised October 7, 2020 and December 20, 2020; accepted February 18, 2021. This work was supported in part by the National Science Foundation Project of China under Grant 61971032, in part by the National Natural Science Foundation of China under Grant 62002034, and in part by the Fundamental Research Funds for the Central Universities under Grant 2020CDCGRJ072 and Grant 2020CDJQY-A021. Associate Editor: F. Wotawa. *(Corresponding authors: Haitao Xu; Meng Yan.)*

Ruifeng Duan is with the School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing 100083, China, and also with the Yunhe (Henan) Information Technology Company Ltd., Zhengzhou 450000, China (e-mail: duanruifeng0905@163.com).

Haitao Xu is with the School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing 100083, China (e-mail: alex_xuht@hotmail.com).

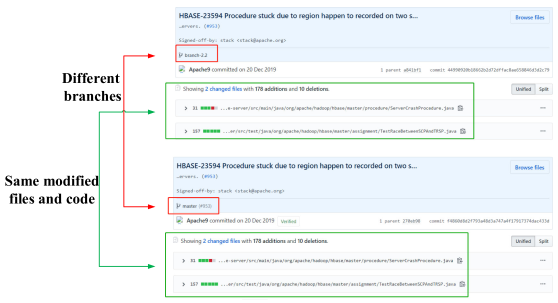
Yuanrui Fan is with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310007, China (e-mail: yrfan@zju.edu.cn).

Meng Yan is with the School of Big Data and Software Engineering, Chongqing University, Chongqing 401331, China (e-mail: mengy@cqu.edu.cn).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TR.2021.3061618.

Digital Object Identifier 10.1109/TR.2021.3061618

[1] [Online]. Available: https://hbase.apache.org/



```
HBASE-23594 Procedure stuck due to region happen to recorded on two s...    Browse files
…ervers. (#953)

Signed-off-by: stack <stack@apache.org>
branch-2.2
Apache9 committed on 20 Dec 2019            1 parent a841bf1   commit 44990920b18662b2d72dffac8ae658846d3d2c79
Showing 2 changed files with 178 additions and 10 deletions.                   Unified  Split
  > 31  ...e-server/src/main/java/org/apache/hadoop/hbase/master/procedure/ServerCrashProcedure.java
  > 157 ...er/src/test/java/org/apache/hadoop/hbase/master/assignment/TestRaceBetweenSCPAndTRSP.java

HBASE-23594 Procedure stuck due to region happen to recorded on two s...    Browse files
…ervers. (#953)

Signed-off-by: stack <stack@apache.org>
master (#953)
Apache9 committed on 20 Dec 2019  Verified  1 parent 270eb98   commit f4860d8d2f793a48d3a747a4f17917374dac433d
Showing 2 changed files with 178 additions and 10 deletions.                   Unified  Split
  > 31  ...e-server/src/main/java/org/apache/hadoop/hbase/master/procedure/ServerCrashProcedure.java
  > 157 ...er/src/test/java/org/apache/hadoop/hbase/master/assignment/TestRaceBetweenSCPAndTRSP.java
```

Fig. 1. Pair of duplicate changes.

changes (*referred to as Un-dup change dataset*). Then, we use the refactoring aware SZZ (**RA-SZZ**) algorithm to label the datasets [17], [18]. After that, we build two models using two datasets, respectively. And we construct two kinds of JIT defect prediction models: 1) classification JIT defect prediction models to identify changes as buggy or clean; 2) effort-aware JIT defect prediction models to prioritize changes considering the inspection effort. For the classification JIT defect prediction models, we leverage three classifiers as underlying classifiers, which are random forest (RF), logistic regression (LR), and Naive Bayes (NB), respectively [19]–[21]. We use area under the curve (AUC) [22], Matthews correlation coefficient (MCC) [23], and F1-score [24] as performance measures. For the effort-aware JIT defect prediction models, two effort-aware models are adopted including classify-before-sorting (CBS) [25] and OneWay [26]. The Recall@20% is used as the performance measure of effort-aware JIT defect prediction models. By doing so, the impact of duplicate changes on evaluating the JIT defect prediction models is investigated. Moreover, through calculating the importance ranking of the metrics of JIT defect prediction models, the impact on the interpretation of JIT models is also explored.

As short-hand notations, the model using the In-dup change dataset is denoted as **In-dup model**, and the model using the Un-dup change dataset is **Un-dup model**.

A scale empirical study on eight projects including 105 828 changes is conducted. It is shown that the average proportion of duplicate changes is 13% among the studied projects. The duplicate changes have a significant impact on the experience metric calculation of JIT defect prediction. For the 50% of the changes, removing duplicate changes among the studied projects decreases the experience metrics with an average of 6–55. For the classification JIT defect prediction models, removing duplicate changes among the studied projects can significantly improve the performance of In-dup models ranging from 1 to 125% concerning various performance measures (i.e., AUC, MCC, and F1). For the effort-aware JIT defect prediction models, in terms of Recall@20%, removing duplicate changes among the studied projects can significantly improve the performance of CBS effort-aware JIT defect prediction models ranging from 4 to 68%, yet reduce the performance of OneWay effort-aware JIT defect prediction models ranging from 2 to 33%. While the duplicate changes lead to a significant impact on the second-ranked and third-ranked metrics, the most important metrics are not to be impacted by the duplicate changes.

The contributions of this article are summarized as follows.

1) To the best of our knowledge, this article is the first to investigate the impact of duplicate changes on JIT defect prediction. We find that the scale of the duplicate changes varies across projects, and, on average, 13% duplicate changes are duplicate among the studied projects.
2) We conduct an empirical study to evaluate the impact of duplicate changes on JIT defect prediction. The results show that the duplicate changes have a significant impact on the JIT defect prediction (i.e., the model metrics, the model evaluation, and the model interpretation).

II. Related Work

A. JIT Defect Prediction

Software defect prediction is one of the important techniques to improve the quality of software. Effective application of software defect prediction technologies can reduce the cost of software maintenance [3], [27]. The traditional defect prediction technology mainly targets coarse-grained software entities, such as files, modules, packages, etc. However, developers have poor traceability for defects from coarse-grained software entities that have large lines of code (LOC) [3]. As a result, researchers should pay more attention to the fine-grained defect prediction techniques [6]. Code change defect prediction is an important kind of fine-grained defect prediction techniques. Mockus and Weiss [5] first proposed a technique to predict defect changes which is referred to as the change-level defect prediction. Sliwerski *et al.* [28] proposed SZZ algorithm, which links each **bug-fixing change** (*changes are treated to fix bugs*) to the source code change introducing the original defect by combining information from the VCS with the issue tracking system (ITS). Kim *et al.* [4] first proposed a scheme to assess risk for each code change. Kamei *et al.* [6] referred to this defect prediction technology as the JIT defect prediction technology for the first time and they performed their case study on 11 different projects (i.e., six open-source and five commercial projects). To conduct experiment datasets, they extracted data from the concurrent versions system repositories of the projects and combined it with bug reports and labeled their data by using SZZ algorithm. The empirical results indicated the practicability of JIT defect prediction technology and also inspired the enthusiasm for studying JIT defect prediction. More research results subsequently appeared. Yang *et al.* [29] first applied the deep learning techniques in JIT defect prediction. They found that using deep learning techniques significantly improved the performance of instant defect prediction models. McIntosh and Kamei [30] found that fluctuations in the properties of fix-inducing changes can impact the performance and interpretation of JIT models.

In addition, due to the limited resources for developers to check, there are many studies on effort-aware JIT defect prediction. Kamei *et al.* [6] used a linear regression method to build an effort-aware change-level defect prediction model effort aware logistic regression (EALR). EALR is to prioritize changes in descending order by predicting the changed defect density. Yang *et al.* [31] applied the unsupervised to effort-aware JIT defect prediction. They found that their unsupervised models could achieve high recall in effort-aware JIT defect prediction.

Huang *et al.* [25] proposed an improved supervised model and is referred to as CBS. They found that CBS significantly improved the precision and F1-score. Fu *et al.* [26] revisited the study of Yang *et al.* [29] and proposed a novel approach to use supervised methods on selecting the best models in effort-aware JIT defect prediction, which is referred to as OneWay.

All of the above studies did not consider the branches outside the main branch during the mining of change histories for constructing their training datasets. We aim to make the first step toward the assessment of this problem.

B. *Noisy Data on Defect Prediction*

Researchers have proposed various data quality issues of defect prediction. It was found that VCS and ITS would be noisy sources of data [30]. Prior studies have shown that noise generally issues with the linkage process and the bug reports. Bachmann *et al.* [32] found that there would occasionally be bias in the links between issue reports and code changes. Bird *et al.* [33] noted that some changes of fixing bug might not be correctly recorded, leading to some initial defect changes that could not be identified. Bird *et al.* [33] further proposed that rich experienced developers could be good at linking bug reports to the corresponding code changes. Noise may also appear in bug reports owing to bug reports that are often mislabeled. Kim *et al.* [34] found that using datasets that have a 20–35% mislabeling rate may significantly influence the evaluation of the defect prediction model. Herzig *et al.* [35] found that 43% of all bug reports are mislabeled, and this mislabeling impacts the ranking of the most defect-prone files.

Tantithamthavorn *et al.* [2] found that labeling errors did not significantly affect the accuracy of the model but significantly affected the recall of the model. Neto *et al.* [18] proposed a framework in their work to evaluate the quality of the data produced by the SZZ algorithm. The noise produced by SZZ may threaten the evaluation of defect prediction models. Fan *et al.* [17] analyzed the impact of noise produced by four SZZ variants on JIT defect prediction models, including B-SZZ, AG-SZZ, MA-SZZ, and RA-SZZ. They noted that AG-SZZ most likely reduces the performance of JIT defect prediction models.

The arising of duplicate changes does not ascribe to the imperfect operations of developers. Duplicate changes are the noisy data from VCS to JIT defect prediction. Our study can be considered as an extension of the above studies.

III. Experiment

In this section, we first take a brief introduction for our studied projects. Second, we describe the details of duplicate changes identification. Third, we describe the studied metrics and the process of data preprocessing. We also depict our experimental procedures. In the end, we present the detail of constructing models and evaluation measures. We attempt to address four questions as follows:

RQ1: How many duplicate changes are there in all the changes?

RQ2: How do duplicate changes impact the model metrics of JIT defect prediction?

TABLE I
Overview of Studied Projects

Projects	**Description**	**#Changes**
ActiveMQ	an open source Java-based messaging server.	8210
Camel	an open source integration framework.	27729
Derby	A relational database management system(RDBMS) written in Java.	9439
Geronimo	an open source application server for Java Enterprise Edition (J2EE).	9146
HBase	a non-relational database for large data.	14593
Hadoop Common	A software for distributed processing of large data sets using clusters of computers.	27240
OpenJPA	Java persistence library using a object-relation mapping (ORM) solution.	6368
Pig	A high-level platform for creating programs using Hadoop to analyze large data sets.	3103
Total		105828

RQ3: How do duplicate changes impact the evaluation of JIT defect prediction models?

RQ4: How do duplicate changes impact the interpretation of JIT defect prediction models?

A. *Studied Projects*

We select projects for our analysis along three criteria as follows.

1) *Hosted in Public Data Repositories:* To ensure the availability of our data and strengthen the replication of our experiments, our studied projects are chosen from Apache open-source system and have been used in many prior JIT defect prediction studies [17], [18], [36].
2) *Including Various Sizes of Data:* Mende [37] noted that the performance of defect models may be impacted by the size of datasets. To deal with the potential impact of the size of used datasets, the eight analyzed projects have various scales of changes.
3) *Easy to Link Code Changes and Bug Reports:* Linking code changes and bug reports is essential for the SZZ algorithm when identifying bug-introducing changes. Prior studies have shown that there may exhibit bias in the link between code changes and bug reports [17], [38], [39]. Bacchelli *et al.* [39] observed that the links between code changes and bug reports that are managed by JIRA ITS are much better than those managed by Bugzilla ITS. Hence, we chose the studied project that leverages JIRA ITS to manage bug reports.

Table I shows the introduction of eight studied projects that satisfy the above three criteria.

To ensure that our data have the least mislabeled changes, we adopt the RA-SZZ algorithm proposed by Neto *et al.* [18]. First, RA-SZZ leverages the git diff command to identify the lines that were changed by bug-fixing changes [36]. Then, RA-SZZ filters away refactoring lines (i.e., blank/comment lines, modification of code indentation, etc.). RA-SZZ further leverages the annotation graph to trace back the other identified lines through the change history [18]. Finally, RA-SZZ identifies the truly bug-introducing changes. Fan *et al.* [17] recommended that RA-SZZ should be used when identifying bug-introducing changes. Hence, in our study, we apply the RA-SZZ algorithm to label our datasets improving the reliability of data labeling.

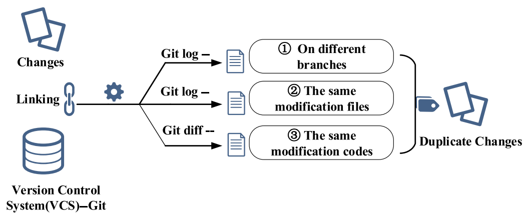

Changes
Linking
Version Control System(VCS)--Git
Git log
Git log
Git diff
① On different branches
② The same modification files
③ The same modification codes
Duplicate Changes

Fig. 2. Process of duplicate changes identification.

B. Duplicate Changes Identification

In this article, we propose an approach to identifying duplicate changes. As shown in Fig. 2, we traverse each change in turn from the VCS of our studied projects. We first traverse changes within a certain time interval from different branches. We set the interval as one week. For each pair of duplicate changes, they must satisfy all the three conditions: 1) whether they belong to different branches; 2) whether they have the same modified files; 3) whether they have the same added and deleted lines in each file. The details of the three conditions are as follows.

1) *Whether They Belong to Different Branches:* All Git repositories start with a master branch by convention [14]. The graph represented by changes can be represented as a directed acyclic graph [38]. Each change is based on one or several prior (parent) versions. We compare the hashID of each change by running the "git log" command [40]. For the two changes, if they are not the ancestors of each other, they belong to different branches.
2) *Whether They Have the Same Modified Files.* We traverse the change histories of each change to get the name and number of modified files by running the "git log" command [40]. For the two changes from different branches, if they modified the same files, they work on similar or related functionality.
3) *Whether They Have the Same Added and Deleted Lines in Each File.* We can view the code change location by running the "git diff" command [40]. For the two changes from different branches, if they have the same added and deleted lines in each file, they modified the same source code.

We also take a manual analysis of the identified duplicated changes. It is difficult to check all the identified duplicate changes since it requires large amount of manual effort to analyze the modified lines of each change. Thus, we randomly sample 100 pairs of identified duplicate changes for each project. We perform a manual analysis of the sampled duplicate changes to investigate the accuracy of our approach.

Table II shows that our identification approach can achieve a high accuracy ranging from 95 to 99%. In terms of the mislabeling cases, we note that the mislabeling cases mainly occurred when developers moved branches. For instance, when developers pulled and merged branches, this change content may be the same as the existing change of the pulled branches. The two changes would be identified by our approach as duplicate changes. However, the two changes should not be considered as duplicate changes since they are implemented for different purposes. The percentage of changes that are incorrectly identified as duplicate changes is 1% in most cases. Such mislabeling cases are not likely to impact our analysis.

TABLE II
ACCURACY OF DUPLICATE CHANGES IDENTIFICATION ON SAMPLED CHANGES

Projects	Random sample	Mislabeling	#acc
ActiveMQ	100	1	99.00%
Camel	100	2	98.00%
Derby	100	3	97.00%
Geronimo	100	1	99.00%
Hbase	100	5	95.00%
Hadoop C.	100	1	99.00%
OpenJPA	100	1	99.00%
Pig	100	1	99.00%
Avg.			98.13%

TABLE III
STUDIED CHANGE METRICS

Dimension	Metric	Description
Diffusion	NS	The number of modified subsystems.
	ND	The number of modified directories.
	NF	The number of modified files.
	Entropy	Distribution of modified code across each file.
Size	LA	Lines of code added.
	LD	Lines of code deleted.
	LT	Lines of code in a file before the change.
Purpose	FIX	Whether or not the change is a defect fix.
History	NDEV	The number of developers that changed the modified files.
	AGE	The average time interval between the last and the current change.
	NUC	The number of unique changes to the modified files.
Experience	EXP	Developer experience.
	REXP	Recent developer experience.
	SEXP	Developer experience on a subsystem.

C. Studied Metrics

In this article, we use the 14 basic features proposed by Kamei *et al.* [6] to construct JIT models. These metrics have proved to be available and useful [16], [17], [29]. Table III outlines such change metrics.

As shown in Table III, they are concerned with five dimensions including diffusion (i.e., NS, ND, NF, and entropy), size (i.e., LA, LD, and LT), purpose (i.e., FIX), history (i.e., NDEV, AGE, and NUC), and experience (i.e., EXP, REXP, and SEXP). Specifically, the diffusion dimension characterizes the distribution of a change. The size dimension represents the scale of a change. The purpose dimension has only one metric, FIX, which indicates whether the change is to fix a bug. The measurement of the historical dimension represents the situation where developers modify files when code history changes. The experience dimension is composed of three metrics: EXP, REXP, and SEXP. The three experience metrics are measured by counting the number of submitted changes by a developer.

D. Data Preprocessing

Some of the above 14 studied metrics are highly skewed and correlated [6], [41], [42]. Such metrics may impact the model evaluation [43]. In this article, we preprocess data in the following two parts.

Logarithmic transformation. Logarithmic transformation is usually used to create monotonic data transformation to alleviate the effect of highly skewed metrics [6], [17]. In our article, the standard logarithmic transformation $\ln(x+1)$ is used for each

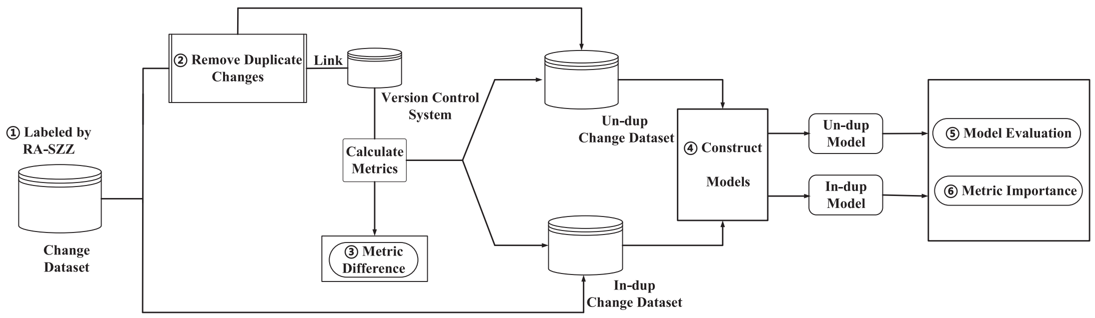


Fig. 3. Overview of the design of our case study.

metric, except the FIX metric since the FIX metric is a binary variable.

Dealing with collinearity. Considering that the degrees of correlation are different for studied metrics, we separately perform the metric selection on each project. After our metrics are performed using the logarithmic transformation, we apply the three policies (i.e., correlation analysis, independence analysis, and redundancy analysis) to stepwise metric selection as follows.

1) *Correlation Analysis:* The Spearman rank correlation test is used to calculate the correlation between each pair of measures on each project [44]. According to the results of the Spearman rank correlation test, we cluster the correlated metrics. We take the correlation threshold as 0.8. As the correlation value is higher than 0.8, we keep the metrics that are easier to understand [6], [17]. For example, we note that the correlation value of NF and ND is larger than 0.8, and we dropped ND and instead used NF.
2) *Independence Analysis:* The Chi-square independence test is used to examine the statistical independence between the FIX metric and other metrics [45]. Eventually, we drop the FIX metric since it is not independent from the other metrics (e.g., ND) among all the studied projects.
3) *Redundancy Analysis:* The $REDUN$ function [46] is used to detect redundant metrics. We note that there are not any redundant metrics among all the studied projects.

E. Case Study Design

In our case study, we construct two JIT defect prediction models: a baseline model that is built using the In-dup change datasets (*referred to as In-dup model*) and a model that is built using the Un-dup change datasets (*referred to as Un-dup model*).

To ensure that our conclusions are statistically robust, we adopt the out-of-sample bootstrap validation technique, which is also used by prior studies [17], [47]. We first generate bootstrap sample of sizes N with replacement from the training data to train models, while we serve the samples that do not appear in the bootstrap samples as testing data. In our article, the out-of-sample bootstrap process iterates 1000 times.

Moreover, the generic metric importance score is used to measure the importance of the metrics, which is proposed by Tantithamthavorn and Hassan [43]. For the testing data in each bootstrap iteration, we first randomly permute the values of the metrics to generate a dataset with only one metric permuted. Then, we compute the difference in the misclassification rate of the models that are applied on the clean dataset and the dataset with randomly permuted metrics. We take the value of the difference as the important scores of each metric. This process also iterates 1000 times.

Fig. 3 depicts the overall framework for the experiment of our article, which is made up of the following steps.

1) *Label the change dataset.* We use the RA-SZZ algorithm to label the change dataset of our studied projects.
2) *Remove duplicate changes.* We remove duplicate changes in order to produce Un-dup change datasets.
3) *Analyze metric difference.* We compare the value of experience metric after removing duplicate changes with that of the original change dataset.
4) *Construct models.* We construct our models using the In-dup change datasets and Un-dup change datasets. In total, we have two JIT models, referred to as In-dup and Un-dup model, respectively. Then, we perform the out-of-sample bootstrap validation technique.
5) *Evaluate model performance.* We repeat the out-of-sample bootstrap process for 1000 times. We further evaluate the models based on the results of various performance measures.
6) *Analyze model interpretation.* We analyze the importance ranking of the metrics by computing the generic metric importance score.

F. Construct JIT Defect Prediction Models

In order for the experiment to be as comprehensive as possible, we build two kinds of JIT defect prediction models: 1) classification JIT defect prediction models; 2) effort-aware JIT defect prediction models.

1) Construct Classification JIT Defect Prediction Models: Prior studies pointed out that the performance of the model would be sensitive to training data settings and different classifiers [17], [24]. We perform two different settings for the training dataset in constructing classification models.

1) The original training data is imbalance data. To ensure that the training and testing corpora share similar characteristics and representative to the original dataset, following prior works [17], [48], we use the original training data to train models.
2) The balance data is also used in many prior JIT studies [6], [7], [25], [26]. Hence, the undersampling technique is used to rebalance the training data. Briefly, we randomly select examples from the majority class (i.e., clean changes) and delete them from the training dataset.

Then, for the test dataset, we keep the original testing data, which is imbalance data. Based on the above settings, we adopt three widely used classifiers [6], [17], [47] as follows.

Random forest. RF is a classifier with multiple decision trees and an ensemble learning method. When performing a classification task and importing the new samples, each decision tree will get its classification result. The RF will treat the result with the most classification as the final result [20].

Logistic regression. LR is a generalized linear regression analysis model and a widely used binary classification classifier. It is used to indicate the possibility of something happening. Its working model is to use one or more variables as independent variables and then use a binary variable as the dependent variable to analyze and estimate the relationship between the independent variable and the dependent variable [19].

Naive Bayes. NB is a classifier that originates from Bayes' theorem and independent assumptions of feature conditions. It assumes that each input variable is independent and uses the knowledge of probability statistics to classify the sample dataset [21].

2) Construct Effort-Aware JIT Defect Prediction Models: Effort-aware models aim to find more defects with limited effort. To ensure that the conclusions of our experiments are robust, we adopt two effort-aware JIT defect prediction techniques including CBS [25] and OneWay [26]. The two technologies are set as follows.

CBS. The working mode of CBS is simply to sort first and then predict. First, a supervised model is used to predict changing defect propensities. Then, for the tendencies which are predicted to be buggy, CBS will prioritize them in ascending order based on their churn size [25].

OneWay. OneWay first evaluates one unsupervised model on supervised training data. Then, the most cost-effective unsupervised model is picked to prioritize changes on testing data [26].

Additionally, for the dataset of the two effort-aware JIT defect prediction models, we balance the training data by the method of undersampling.

G. Evaluation Measures

In our study, we adopt four evaluation measures (i.e., AUC, MCC, F1-score, and Recall@20%) to measure the performance of JIT defect prediction models. There is a brief introduction as follows.

AUC. AUC stands for "area under the receiver operator characteristics (ROC) curve," which measures the entire two-dimensional area underneath the entire ROC curve (think integral calculus) from (0,0) to (1,1). AUC provides an aggregate measure of performance across all possible classification thresholds and is widely used [6], [7], [30].

MCC. MCC is the geometric mean of the regression coefficients of the problem and its dual [23]. Song *et al.* [24] proposed that MCC performs better with performance standards for addressing the imbalance in the data and is potentially useful for software defect prediction, which can be a more balanced measure.

F1. F1 is a weighted harmonic average of recall (*the percentage of changes labeled as buggy that were correctly classified*) and precision (*the percentage of actual buggy changes that were correctly classified*) [24]. Hence, F1-score can be considered as a comprehensive evaluation measure [49].

Recall@20%. Recall@20% measures the percentage of buggy changes that a developer can find in 20% of the code for all bug-introducing changes in the total LOC. Many studies show that the available resources account for only 20% of the workload of checking all changes. It is a commonly used measure to measure cost-effectiveness [6], [25], [26], [31].

IV. Results

In this section, we aim to display experimental results and address the following research questions.

(RQ1) How many duplicate changes are there in all the changes?

Motivation: In software development practice, it is difficult to maintain an overview of various branches. Most of mining software repository activities focus on the master branch [50]. There are many duplicate changes among all the branches, which are referred to as a pair of changes with identical implementation in different branches [15]. However, little is known about the scale of duplicate changes among all the changes. Thus, we would like to explore the distribution of duplicate changes among all the changes.

Approach: We mine eight open-source software projects containing a total of more than 10 k changes from the Apache Git repository. In order to identify duplicate changes, we propose a framework to traverse each change. If the two changes are from different branches and modify identical lines in identical files, we regard the two changes as a pair of duplicate changes. We remove one of the changes and calculate the percentage of duplicate changes among all the projects. In addition, we investigate the scale of duplicate changes that are bug-introducing changes. Furthermore, we investigate the scale of duplicate changes that are bug-fix changes according to the purpose metric.

Result: Table IV provides the distribution of duplicate changes in each project. From left to right, the table shows the number of duplicate changes and the percentage of total changes for each project (i.e., bug-introducing changes, bug-fix changes, etc.). The table also presents the number of branches and releases of each project.

From Table IV, we have the following observations.

1) The average proportion of duplicate changes is 13% in the studied projects. The highest proportion of duplicate changes is our HBase dataset for 36.10%, and the smallest proportion is 6.80% of the OpenJPA dataset. When the results of these two projects are removed, the average proportion of the remaining six projects is 10%.
2) In addition, Table IV shows that the average proportion of duplicate bug-introducing changes is 14% for the bug-introducing changes in our projects. Note that 60.59% of the bug-introducing changes in the HBase dataset are duplicate bug-introducing changes, yet the average proportion of the remaining seven projects is only 7.23%.
3) Furthermore, Table IV also shows that the average percentage of duplicate bug-fix changes in the eight projects is 17%. The HBase dataset also has the highest percentage of duplicate bug-fix changes. The average value of the

TABLE IV
DISTRIBUTION OF DUPLICATE CHANGES IN EACH PROJECT

Projects	**#Changes**	**Duplicate Changes**	**#Bug-introducing Changes**	**Duplicate Bug-introducing Changes**	**#Bug-fix Changes**	**Duplicate Bug-fix Changes**	**#Branches**	**#Releases**
ActiveMQ	8210	644(7.83%)	2919	172(5.89%)	3080	269(8.73%)	15	66
Camel	27729	3050(10.92%)	5255	730(13.89%)	6261	1029(16.44%)	32	155
Derby	9439	791(8.38%)	1136	60 (5.28%)	3363	333(9.90%)	1	0
Geronimo	9146	697(7.62%)	1861	81(4.35%)	2100	313(16.90%)	39	35
HBase	14593	5268(36.10%)	2672	1619(60.59%)	4700	2279(48.49%)	52	736
Hdoop C.	27240	4342(15.94%)	1690	131(7.75%)	4502	602(13.37%)	127	192
OpenJPA	6368	433(6.80%)	692	39(5.64%)	1742	191(10.96%)	30	35
Pig	3103	303(9.76%)	467	37(7.82%)	1012	152(15.02%)	27	59
Avg.	13229	13%	2087	14%	3345	17%	40	160

remaining seven projects is only 13.11%. Also, we find that the project with a high number of branches and releases, the proportion of duplicate changes (i.e., duplicate bug-introducing changes, duplicate bug-fix changes, etc.) is also high. In particular, the HBase dataset not only has a large amount of data but also has a large number of branches and releases compared with other projects.

The scale of duplicate changes varies across projects. On average, 13% of changes from different branches are duplicate.

(RQ2) How do duplicate changes impact the model metrics of JIT defect prediction?

Motivation: All the changes can be characterized by 14 change-level metrics with five dimensions [6]. Duplicate changes from different branches may impact various metrics calculation of change histories, such as develop experience of contributors to a project [50]. As Table III shown, the experience dimension is made up of three metrics: developer experience (EXP), recent experience (REXP), and subsystem experience (SEXP) [6]. The three experience metrics are measured by counting the number of submitted changes by a developer. Generally, each duplicate change may be recorded as the development experience of the contributor who produces this duplicate change. However, we are not clear about the impact of duplicate changes from the branches on the experience metric calculation. Therefore, we would like to quantify the difference resulting from the application of omitting duplicate changes in the history of changes.

Approach: We start from Un-dup datasets that are produced by removing duplicate changes and In-dup datasets (i.e., the original datasets). Next, we do a subtraction operation to compare the three model metrics (i.e., EXP, REXP, and SEXP). If the changes of the two datasets have the same HashID, we will subtract the experience metric of Un-dup datasets from that of In-dup datasets.

To check whether the difference is statistically significant, we perform the Wilcoxon signed-rank test [51] with a Bonferroni correction [52] adjusted to compare the three model metrics (i.e., EXP, REXP, and SEXP) of the Un-dup datasets with that of the In-dup datasets.

TABLE V
ADJUSTED P-VALUES COMPARING THE EXP, REXP, AND SEXP METRICS OF THE UN-DUP CHANGE DATASETS WITH THAT OF THE IN-DUP CHANGE DATASETS

Projects	**Metrics**		
	EXP	REXP	SEXP
ActiveMQ		$<$ 2.2e-16	
Camel		$<$ 2.2e-16	
Derby		$<$ 2.2e-16	
Geronimo		$<$ 2.2e-16	
Hbase		$<$ 2.2e-16	
Hadoop C.		$<$ 2.2e-16	
OpenJPA		$<$ 2.2e-16	
Pig		$<$ 2.2e-16	

Results: In Table V, we show the adjusted p-values comparing the calculated experience metric of the In-dup change datasets with that of the Un-dup change datasets. Fig. 4 shows the difference of the experience metrics between the In-dup change dataset and the Un-dup change dataset for each project.

From Table V and Fig. 4, we observe that duplicate changes significantly impact the calculation of the experience metrics. In terms of 50% of the changes, removing duplicate changes decreases the three model metrics (i.e., EXP, REXP, and SEXP) with an average difference of 10–57, 6–28, and 6–39, respectively. The biggest difference is the Hadoop Common dataset with a large amount of data. The three model metrics (i.e., EXP, REXP, and SEXP) are reduced by 12–83, 5–37, and 7–50, respectively. Although the project Pig has a small amount of data, the three model metrics (i.e., EXP, REXP, and SEXP) are reduced by 6–50, 4–28, and 5–55, respectively. Indeed, the above two datasets have not the low proportion of duplicate changes.

The duplicate changes have a significant impact on the experience metrics calculation of JIT defect prediction. For the 50% of the changes, removing duplicate changes among the studied projects decreases the experience metrics with an average of 6–55.

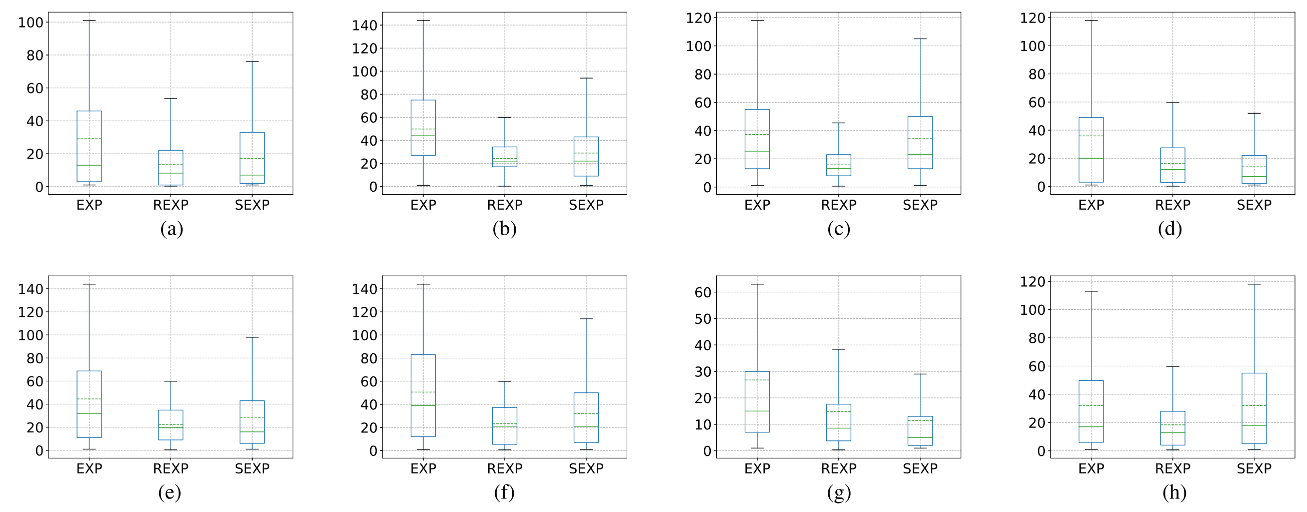


Fig. 4. Difference of the experience metric between the In-dup change dataset and the Un-dup change dataset for each project.

(RQ3) How do duplicate changes impact the evaluation of JIT defect prediction models?

Motivation: JIT defect prediction models not only can help developers find the potentially defective changes in the shortest possible time but also provide early feedback to developers to optimize effort for inspection [3], [53]. In addition, as discussed in RQ2, the duplicate changes significantly impact the calculation of the experience metrics. Hence, we would like to investigate whether the duplicate changes impact the evaluation of JIT defect prediction models, including classification models and effort-aware models.

Approach: To investigate the impact of duplicate changes on the evaluation of JIT defect prediction models, we first construct two datasets: 1) In-dup changes datasets (i.e., datasets where duplicate changes are not removed); 2) Un-dup change datasets (i.e., datasets where duplicate changes are removed). Then, we evaluate the performance of JIT defect prediction models that are built on the In-dup change datasets and Un-dup change datasets, respectively.

For two kinds of JIT defect prediction models: classification JIT defect prediction models and effort-aware JIT defect prediction models. The AUC and MCC measures are used to evaluate the classification JIT defect prediction models trained on imbalance data. The AUC and F1 measures are used to evaluate the classification JIT defect prediction models trained on balance data. The Recall@20% is used to evaluate the effort-aware JIT defect prediction models.

As Fig. 3, we adopt the out-of-sample bootstrap as model validation technique. We train the In-dup model and the Un-dup model in each bootstrap iteration, respectively. Then we apply two models on the testing data and calculate the measure score. After 1000 bootstrap iterations, we have 1000 measure scores for each of the two models. We calculate the average value of the measure scores. Note that the baseline model is In-dup model, which is built using the In-dup change datasets.

To check if our measure scores are statistically significant, we compare the measure scores of the Un-dup model and the In-dup model by applying the Wilcoxon signed-rank test [51] with Bonferroni correction [52]. In addition, the Cliff's delta[2] [54] is also used to measure the magnitude of the difference. Moreover, we analyze the ratios of the absolute performance difference to the performance of the In-dup model (i.e., the percentage of performance improvement).

Results: In order to better describe the experimental results, we divide the experimental results into three parts to illustrate (i.e., the classification JIT defect prediction models trained on imbalance data, the classification JIT defect prediction models trained on balance data, and the effort-aware JIT defect prediction models). As shorthand notations, we denote them as **RQ3-a result**, **RQ3-b result**, and **RQ3-c result**, respectively, as shown below.

RQ3-a result: Table VI shows the average of AUC and MCC scores of the In-dup and Un-dup models. The table also presents the ratios of the absolute performance difference to the performance of the In-dup model with regard to the AUC and MCC scores. Table VII shows the adjusted p-values and Cliff's delta of the Un-dup model compared with the In-dup model in terms of AUC and MCC scores.

From Tables VI and VII, we have the following observations.

1) For the Un-dup model, on across the eight projects, we observe that the AUC score and MCC score are generally higher than that of the In-dup model on all cases. In terms of statistical significance, we observe that the Un-dup model significantly improves performance over the In-dup model with a large effect size on most cases.
2) In terms of AUC, the absolute difference between the In-dup model and Un-dup model is ranging from 0.01 to 0.08. The Un-dup model shows the performance improvement of at least 1% in most cases.
3) In terms of MCC, the absolute difference between the In-dup model and Un-dup model is ranging from 0.01 to 0.05. The Un-dup model shows the performance improvement of at least 3% in most cases. In addition, the value of the percentage of performance improvement of the

[2]A Cliff's delta with 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 VI
AVERAGE AUC AND MCC SCORES OF THE IN-DUP AND UN-DUP MODELS TRAINED ON IMBALANCE DATA

Classifiers	**Projects**	**AUC**			**MCC**		
		In-dup	Un-dup	Ratio	In-dup	Un-dup	Ratio
RF	ActiveMQ	0.81	0.82	1%	0.45	0.47	4%
	Camel	0.84	0.85	1%	0.39	0.40	3%
	Derby	0.80	0.80	0%	0.30	0.30	0%
	Geronimo	0.77	0.78	1%	0.36	0.36	0%
	HBase	0.82	0.86	5%	0.36	0.36	0%
	Hadoop C.	0.86	0.86	0%	0.23	0.25	9%
	OpenJPA	0.79	0.79	0%	0.28	0.29	4%
	Pig	0.80	0.85	6%	0.36	0.41	14%
Avg.		0.81	0.83	2%	0.34	0.36	6%
LR	ActiveMQ	0.81	0.82	1%	0.45	0.46	2%
	Camel	0.83	0.84	1%	0.33	0.34	3%
	Derby	0.76	0.76	0%	0.18	0.21	17%
	Geronimo	0.73	0.73	0%	0.28	0.28	0%
	HBase	0.77	0.85	10%	0.23	0.27	17%
	Hadoop C.	0.79	0.80	1%	0.04	0.09	125%
	OpenJPA	0.73	0.75	3%	0.13	0.15	15%
	Pig	0.78	0.83	6%	0.23	0.32	39%
Avg.		0.78	0.80	3%	0.23	0.27	17%
NB	ActiveMQ	0.77	0.78	1%	0.40	0.42	5%
	Camel	0.80	0.81	1%	0.35	0.37	6%
	Derby	0.75	0.75	0%	0.26	0.27	4%
	Geronimo	0.73	0.73	0%	0.32	0.32	0%
	HBase	0.75	0.83	11%	0.33	0.34	3%
	Hadoop C.	0.74	0.76	3%	0.09	0.18	100%
	OpenJPA	0.73	0.74	1%	0.22	0.23	5%
	Pig	0.76	0.80	5%	0.32	0.33	3%
Avg.		0.75	0.78	4%	0.29	0.31	7%

We also Show the Ratios of the Absolute AUC and MCC Scores Difference to the Performance of the In-dup Model.

TABLE VII
ADJUSTED *P*-VALUES AND CLIFF'S DELTA COMPARING AUC AND MCC SCORES FOR THE UN-DUP MODEL WITH THOSE OF THE IN-DUP MODEL

Classifiers	**Projects**	**Un-dup**			
		AUC		MCC	
RF	ActiveMQ	0.70	(L)***	0.63	(L)***
	Camel	0.96	(L)***	0.67	(L)***
	Derby	0.08	(N)***	0.01	(N)***
	Geronimo	0.38	(M)***	0.01	(N)***
	HBase	1.00	(L)***	0.16	(S)***
	Hadoop C.	0.16	(S)***	0.50	(L)***
	Openjpa	0.13	(N)***	0.10	(N)***
	Pig	0.96	(L)***	0.67	(L)***
LR	ActiveMQ	0.72	(L)***	0.63	(L)***
	Camel	0.97	(L)***	0.55	(L)***
	Derby	0.11	(N)***	0.54	(L)***
	Geronimo	0.11	(N)***	0.15	(N)***
	HBase	1.00	(L)***	0.85	(L)***
	Hadoop C.	0.85	(L)***	0.94	(L)***
	OpenJPA	0.49	(L)***	0.22	(S)***
	Pig	0.95	(L)***	0.92	(L)***
NB	ActiveMQ	0.56	(L)***	0.65	(L)***
	Camel	0.96	(L)***	0.88	(L)***
	Derby	0.22	(S)***	0.29	(S)***
	Geronimo	0.06	(N)***	0.19	(S)***
	HBase	1.00	(L)***	0.16	(S)***
	Hadoop C.	0.86	(L)***	1.00	(L)***
	OpenJPA	0.31	(S)***	0.16	(S)***
	Pig	0.86	(L)***	0.12	(N)***

***$p < 0.001$.

MCC score is greater than the percentage of performance improvement of AUC. However, it does not represent the MCC score tend to be impacted by duplicate changes. Because even if the absolute difference is the same, the percentage change of the smaller number will be greater than that of the larger one.

For the classification JIT defect prediction models trained on imbalance data, after removing duplicate changes, the performance among the studied projects is significantly improved from 1 to 11% and 2 to 125% in terms of AUC and MCC, respectively.

RQ3-b result: Table VIII shows the average AUC and F1 scores of the In-dup and Un-dup models trained on balance data. The table also presents the ratios of the absolute performance difference to the performance of the In-dup model with regard to the AUC and F1 scores. Table IX shows the adjusted *p*-values and Cliff's delta of the Un-dup model compared with the In-dup model in terms of AUC and F1.

From Tables VIII and IX, we have the following observations.

1) For the Un-dup model, on across the eight projects, we observe that the AUC and F1 scores are slightly higher than that of the In-dup model. We find that the Un-dup model shows statistically significant performance improvement for the AUC score in most cases. for the F1 score, we find that only the HBase dataset shows statistically significant reduction with a large effect size. The rest of our projects show statistically significant improvement.
2) In terms of AUC, we note that the absolute difference between the In-dup model and Un-dup model is ranging from 0.01 to 0.06. The Un-dup model shows the performance improvement of at least 1% in most cases. We note that the AUC score of the absolute difference is 0.01 of the two models on random forest classifier.
3) In terms of F1, we note that the absolute difference between the In-dup model and Un-dup model is ranging from 0.01 to 0.07. The Un-dup model shows the performance improvement of at least 2% in most cases. Note that for the HBase dataset, the F1 of the Un-dup model is lower than the In-dup model that may be because the training data is reduced by removing duplicate changes.

For the classification JIT defect prediction models trained on balance data, after removing duplicate changes, the performance among the studied projects and classifiers is significantly improved by 1–9% and 1–21% in terms of AUC and F1 scores, respectively.

RQ3-c result: Table X shows the average Recall@20% scores of the In-dup and Un-dup models. The table also presents the ratios of the absolute performance difference to the performance of the In-dup model with regard to the Recall@20% scores. Table XI presents the adjusted *p*-values and Cliff's delta of the Un-dup compared with the In-dup model in terms of Recall@20%. From Tables X and XI, we have the following observations.

1) For the model built using the CBS technique, on average across eight projects, the Un-dup model shows a significant improvement over the In-dup model with a large effect size on six projects and a medium effect size on two projects. We note that the absolute difference between

TABLE VIII
AUC AND F1 SCORES OF THE IN-DUP AND UN-DUP MODELS TRAINED ON BALANCED DATA

Classifiers	**Projects**	**AUC**			**F1**		
		In-dup	Un-dup	ratio	In-dup	Un-dup	Ratio
RF	ActiveMQ	0.81	0.82	1%	0.66	0.67	2%
	Camel	0.84	0.85	1%	0.54	0.55	2%
	Derby	0.80	0.80	0%	0.42	0.43	2%
	Geronimo	0.89	0.90	1%	0.42	0.43	2%
	HBase	0.85	0.86	1%	0.51	0.48	-6%
	Hadoop C.	0.86	0.86	0%	0.28	0.33	18%
	OpenJPA	0.80	0.80	0%	0.38	0.39	3%
	Pig	0.79	0.84	6%	0.45	0.49	9%
Avg.		0.83	0.84	1%	0.46	0.47	2%
LR	ActiveMQ	0.81	0.82	1%	0.66	0.67	2%
	Camel	0.83	0.84	1%	0.52	0.52	0%
	Derby	0.76	0.76	0%	0.36	0.38	6%
	Geronimo	0.86	0.86	0%	0.34	0.37	9%
	HBase	0.78	0.84	8%	0.45	0.42	-7%
	Hadoop C.	0.76	0.80	5%	0.23	0.25	9%
	OpenJPA	0.73	0.74	1%	0.31	0.32	3%
	Pig	0.77	0.82	6%	0.41	0.45	10%
Avg.		0.79	0.81	3%	0.41	0.42	2%
NB	ActiveMQ	0.77	0.78	1%	0.62	0.64	3%
	Camel	0.80	0.81	1%	0.50	0.51	2%
	Derby	0.75	0.75	0%	0.36	0.37	3%
	Geronimo	0.84	0.84	0%	0.31	0.37	19%
	HBase	0.75	0.82	9%	0.43	0.40	-7%
	Hadoop C.	0.74	0.76	3%	0.16	0.21	31%
	OpenJPA	0.73	0.74	1%	0.30	0.32	7%
	Pig	0.75	0.80	7%	0.34	0.41	21%
Avg.		0.77	0.79	3%	0.38	0.40	5%

We Also Show the Ratios of the Absolute AUC and F1 Scores Difference to the Performance of the In-dup Model.

TABLE IX
ADJUSTED P-VALUES AND CLIFF'S DELTA COMPARING AUC AND F1 SCORES FOR THE UN-DUP MODEL WITH THOSE OF THE IN-DUP MODEL

Classifiers	**Projects**	**Un-dup**			
		AUC		F1	
RF	ActiveMQ	0.70	(L)***	0.70	(L)***
	Camel	0.97	(L)***	0.32	(S)***
	Derby	0.01	(N)***	0.35	(M)***
	Geronimo	0.61	(L)***	0.42	(M)***
	HBase	0.50	(L)***	-1.00	(L)***
	Hadoop C.	-0.32	(S)***	-0.26	(S)***
	OpenJPA	0.20	(S)***	0.33	(S)***
	Pig	0.99	(L)***	0.72	(L)***
LR	ActiveMQ	0.71	(L)***	0.70	(L)***
	Camel	0.97	(L)***	0.20	(S)***
	Derby	0.12	(N)***	0.66	(L)***
	Geronimo	0.15	(N)***	0.89	(L)***
	HBase	1.00	(L)***	-0.94	(L)***
	Hadoop C.	0.99	(L)***	0.88	(L)***
	OpenJPA	0.46	(M)***	0.41	(M)***
	Pig	0.98	(L)***	0.80	(L)***
NB	ActiveMQ	0.55	(L)***	0.64	(L)***
	Camel	0.96	(L)***	0.43	(M)***
	Derby	0.21	(S)***	0.63	(L)***
	Geronimo	0.23	(S)***	1.00	(L)***
	HBase	1.00	(L)***	-0.88	(L)***
	Hadoop C.	0.86	(L)***	0.94	(L)***
	OpenJPA	0.29	(S)***	0.32	(S)***
	Pig	0.96	(L)***	0.92	(L)***

$^{***}p < 0.001$.

the In-dup model and Un-dup model is ranging from 0.02 to 0.32. The In-dup model shows the performance improvement of at least 4% in all cases.

2) For the models built using the OneWay technique, the Un-dup model shows a significant reduction over the In-dup model on four projects. We note that there are only two cases on the absolute difference that are higher than the In-dup model ranging from 0.01 to 0.03. And in the remaining cases, the Un-dup model shows the performance reduction of at least 2% on four cases.

For the effort-aware JIT defect prediction models, in terms of Recall@20%, while removing duplicate changes among the studied projects can significantly improve the performance of CBS effort-aware JIT defect prediction models ranging from 4 to 68%, it can reduce the performance of OneWay effort-aware JIT defect prediction models ranging from 2 to 33%.

TABLE X
RECALL@20% SCORES OF THE IN-DUP AND UN-DUP MODELS

Tech.	**Projects**	**Recall@20%**		
		In-dup	Un-dup	Ratio
CBS	ActiveMQ	0.55	0.71	29%
	Camel	0.52	0.74	42%
	Derby	0.47	0.49	4%
	Geronimo	0.56	0.78	39%
	HBase	0.47	0.79	68%
	Hadoop C.	0.56	0.74	32%
	OpenJPA	0.41	0.43	5%
	Pig	0.46	0.74	61%
Avg.		0.50	0.68	36%
OneWay	ActiveMQ	0.59	0.45	-24%
	Camel	0.49	0.33	-33%
	Derby	0.27	0.26	-4%
	Geronimo	0.54	0.54	0%
	Hadoop C.	0.44	0.45	2%
	HBase	0.43	0.46	7%
	OpenJPA	0.41	0.40	-2%
	Pig	0.38	0.38	0%
Avg.		0.44	0.41	-7%

We Also Show the Ratios of the Absolute Recall@20% Scores Difference to the Performance of the In-dup Model.

(RQ4) How do duplicate changes impact the interpretation of JIT defect prediction models?

Motivation: The interpretation of JIT defect prediction models is closely related to the metrics. By investigating the potential feature interactions, we can understand which features may be important in model decision making that ensures the fairness of the model [3]. According to the importance ranking of the metric, developers can specify the corresponding software quality improvement program to maintain software as easily as possible [4], [6], [17]. Hence, we would like to investigate whether the duplicate changes influence the importance ranking of the metrics.

Approach: We first calculate the generic metric importance scores of the metrics for the In-dup and Un-dup models. Tantithamthavorn *et al.* [43] suggested that the models trained using balance data should be avoided when calculating metric importance. Hence, we calculate metric importance for the models trained on imbalance data. We repeat the bootstrap for 1000 times. Also, the Scott–Knott effect size difference (SK-ESD) is used to test on the metric importance scores [55]. The SK-ESD test is a method of statistical comparison, which has been widely used in prior studies [17], [56]. Then, we count the scale of

TABLE XI
ADJUSTED *P*-VALUES AND CLIFF'S DELTA COMPARING RECALL@20% SCORES FOR THE UN-DUP MODEL WITH THOSE OF THE IN-DUP MODEL

Tech.	Projects	Un-dup (Recall@20%)	
CBS	ActiveMQ	1.00	(L)***
	Camel	1.00	(L)***
	Derby	0.35	(M)***
	Geronimo	1.00	(L)***
	HBase	1.00	(L)***
	Hadoop C.	1.00	(L)***
	OpenJPA	0.37	(M)***
	Pig	1.00	(L)***
OneWay	ActiveMQ	-0.97	(L)
	Camel	-1.00	(L)
	Derby	-0.30	(S)
	Geronimo	-0.08	(N)
	HBase	0.58	(L)***
	Hadoop C.	0.15	(S)***
	OpenJPA	-0.16	(S)
	Pig	-0.04	(N)

***$p < 0.001$.

TABLE XII
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	
	In-dup	Un-dup	In-dup	Un-dup	In-dup	Un-dup
Top-1 important metrics						
NS	0	1	0	0	0	1
NF	4	4	2	2	2	0
LA	4	4	2	2	3	5
ND	0	0	0	1	2	2
LD	0	0	0	0	0	3
LT	0	0	2	2	0	0
Entropy	0	1	0	0	0	0
NDEV	1	0	2	1	1	0
AGE	0	0	0	0	0	0
NUC	0	0	0	0	0	0
EXP	0	1	0	0	0	1
SEXP	0	0	0	1	0	0
Top-3 important metrics						
NS	1	3	1	2	4	5
NF	6	6	4	5	5	2
LA	8	8	5	5	7	8
ND	1	3	2	3	2	2
LD	1	0	0	0	3	4
LT	4	5	5	4	3	2
Entropy	2	1	0	0	0	0
NDEV	1	0	3	2	1	0
AGE	1	1	1	1	0	0
NUC	0	1	5	3	4	3
EXP	2	2	1	3	2	4
SEXP	1	3	2	3	0	1

projects in which a metric as ranked top-1 important metrics and one of the top-3 important metrics. In addition, following Rajbahadur *et al.* [57], we use the rank shifts to denote the important metrics difference between the In-dup model and Un-dup model, which can be calculated as

$$\mathrm{Shifts}(k) = \left(\sum_{v \in V_1(k)} |k - \mathrm{Rank}_2(v)| + \sum_{v \in V_2(k)} |k - \mathrm{Rank}_1(v)| \right) / N_{\mathrm{metric}} \,. \tag{1}$$

In the formula, $V_1(k)$ and $V_1(k)$ are the rank k of the metrics. The N_{metric} is the quantity of the metrics. The $\mathrm{Rank}_1(v)$ and $\mathrm{Rank}_2(v)$ are the rank of the metric v calculated from two models, respectively. To check if the difference is statistically significant, we also use the Wilcoxon signed-rank test to compare the rank shifts with that of no shift case where all shifts are 0.

Result: Fig. 5 depicts the rank difference between the Un-dup model and the In-dup model. Table XII shows the number of projects in which a metric is ranked as the top-1 metric and one of the top-3 important ones, respectively. Table XIII shows the average rank shifts. The table also presents the *p*-values for comparing the rank shifts with the ideal no-shift case.

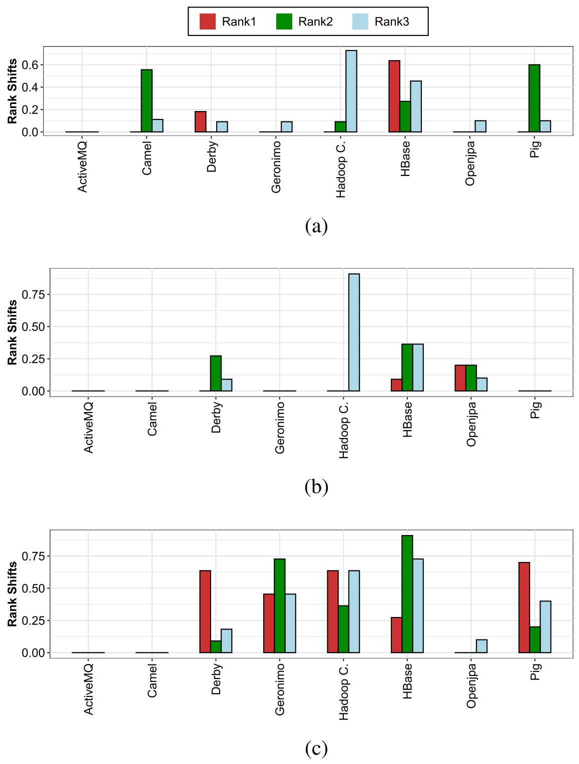


Fig. 5. Rank shifts of the top-3 ranked metrics between the Un-dup model and the In-dup model.

TABLE XIII
AVERAGE RANK SHIFTS COMPARING THE UN-DUP MODEL WITH THE UN-DUP MODEL ACROSS THE EIGHT PROJECTS

Classifiers	Rank	Un-dup	
		Avg.shifts	P-value
RF	1	0.10	0.06
	2	**0.19**	**0.008**
	3	**0.21**	**0.0002**
LR	1	0.03	0.06
	2	**0.10**	**0.02**
	3	**0.18**	**0.008**
NB	1	**0.34**	**0.003**
	2	**0.29**	**0.003**
	3	**0.31**	**0.0009**

The Significant Rank Shifts are in Bold (*p*-Value$<$0.05)

From Tables XII and XIII and Fig. 5, we have the following observations.

1) For the rank difference between the Un-dup model and the In-dup model, the average shifts between the two models range from 0.03 to 0.34, 0.10 to 0.29, and 0.18 to 0.31, respectively. In terms of statistical significant, the top-ranked metrics are obviously robust, only 2–5 projects of the Un-dup model different from the In-dup model. However, the second-ranked and third-ranked metrics are significantly unstable. For the second-ranked and third-ranked metrics, the Un-dup model shows different

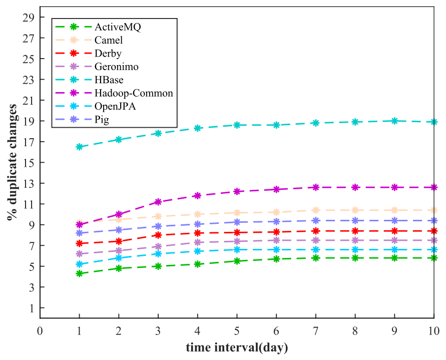


Fig. 6. Proportion of duplicate changes among changes of each studied project when varying the time interval.

from the In-dup model on 3–5 projects and 4–7 projects, respectively.

2) For the most important metrics, we note that LA (i.e., lines of code added) is the most important metric for each of the two models in all cases. In addition, we note that the top-3 metrics of the Un-dup model are consistent with those of the In-dup. For example, LA, NF (i.e., number of files), and LT (i.e., lines of code in a file) are the top-3 metrics for each of the two models on random forest classifier.

The duplicate changes lead to a significant impact on the second-ranked and the third-ranked metrics. However, the most important metrics are not to be impacted by the duplicate changes.

V. Discussion

In this section, we further discuss the underlying experimental subject (i.e., time interval) and data rebalancing techniques of our article.

A. Why Choose One Week as the Time Interval for Identifying Duplicate Changes?

By default, we set one week as the time interval for identifying duplicate changes. The release cycle of modern software can be days or even hours [3]. For instance, an online social entertainment website updates code 50 times a day on average [58]. Given the high frequency of software updates, duplicate changes are more likely to appear within a short time interval. Selecting a different time interval may yield different results. To combat this bias, we conduct an experiment to explore the impact of using different time intervals. We set ten time intervals with one day as a step to repeatedly identify duplicate changes from all branches of eight studied projects. We calculate the proportion of duplicate changes among changes of the studied projects.

Fig. 6 presents the proportion of duplicate changes among changes of each studied project when varying the time interval. We have the following observations: 1) When the time interval is less than seven days, the proportion of duplicate changes increases as the time interval increases; 2) when the time interval is equal or larger than seven days, the proportion of duplicate changes becomes stable with respect to the time interval.

TABLE XIV
Average AUC and F1 Scores of the In-Dup Model for Using Different Data Processing Methods

Classifiers	Projects	Auc			F1		
		Oversampling	Smote	Undersampling	oversampling	Smote	Undersampling
RF	ActiveMQ	**0.82**	0.81	0.81	0.63	0.63	0.66
	Camel	0.84	0.84	0.84	0.52	0.52	0.54
	Derby	**0.81**	0.80	0.80	0.34	0.43	0.42
	Geronimo	**0.90**	**0.90**	0.89	0.43	**0.48**	0.42
	HBase	0.82	0.82	0.85	0.50	**0.54**	0.51
	Hadoop C.	**0.89**	**0.87**	0.86	**0.30**	**0.37**	0.28
	OpenJPA	**0.81**	0.80	0.80	0.33	**0.41**	0.38
	Pig	0.79	0.79	0.79	0.42	**0.48**	0.45
LR	ActiveMQ	0.81	0.80	0.81	0.66	0.65	0.66
	Camel	0.83	0.82	0.83	0.52	0.50	0.52
	Derby	0.76	0.75	0.76	0.36	0.32	0.36
	Geronimo	0.86	0.85	0.86	0.34	0.32	0.34
	HBase	0.78	0.78	0.78	**0.48**	0.45	0.45
	Hadoop C.	**0.78**	0.76	0.76	0.25	0.20	0.23
	OpenJPA	0.73	0.71	0.73	0.31	0.27	0.31
	Pig	0.77	0.77	0.77	0.41	0.38	0.41
NB	ActiveMQ	0.77	0.76	0.77	0.62	0.61	0.62
	Camel	0.80	0.80	0.80	0.50	0.50	0.50
	Derby	0.75	0.74	0.75	0.36	0.35	0.36
	Geronimo	0.84	0.83	0.84	0.31	0.32	0.31
	HBase	0.75	0.75	0.75	0.31	0.33	0.31
	Hadoop C.	0.74	0.70	0.74	0.17	0.17	0.16
	OpenJPA	0.73	0.72	0.73	0.30	0.30	0.30
	Pig	0.75	0.74	0.75	0.33	0.34	0.34
Avg.		**0.80**	0.79	0.79	0.40	0.41	0.41
P-value		>0.05	>0.05		>0.05	>0.05	
Cliff's delta		0.03(N)	-0.07(N)		-0.04(N)	0.01(N)	

The AUC and F1 Scores of the Oversampling and Smote Methods That are Higher Than Those of the Undersampling Method Are in Bold.

In summary, one week is suitable as the time interval for identifying duplicate changes.

B. Impact of the Used Data Rebalancing Techniques

In RQ3-b, we investigate the impact of duplicate changes on the performance of JIT defect prediction models trained on balanced data. To deal with the imbalanced data, we leverage the undersampling technique to balance the training data following prior studies [6], [25]. Undersampling is implemented by reducing the number of majority class instances (i.e., clean changes). In this section, we further evaluate the performance of In-dup and Un-dup models by using two other data-level rebalancing techniques.

Oversampling. Oversampling involves supplementing the training data with multiple copies of some of the minority classes. It increases the samples of the minority class (i.e., buggy changes) by creating samples similar to the existing ones to create an equally proportioned dataset [59]. We randomly copy some minority samples so that the number of majority and minority samples in the training dataset is equal.

Smote. Synthetic Minority Oversampling TEchnique (SMOTE) is a special oversampling method that seeks to avoid overfitting by synthetically creating new minority class instances [60]. It is based on using linear interpolation between minority class data point and one its K-nearest neighbors to generate data. We set $k = 5$.

Tables XIV and XV present the average of AUC and F1 scores for the In-dup and Un-dup models using different data rebalancing techniques, respectively. We also show the adjusted p-values and Cliffs delta of the oversampling and SMOTE techniques compared with the undersampling technique in terms of AUC and F1 scores.

From these two tables, we observe the following. 1) For the In-dup model, in terms of AUC and F1 scores, the undersampling technique achieves a higher score than the other techniques in most cases. The oversampling technique outperforms the other techniques considering the average AUC scores across

TABLE XV
AVERAGE AUC AND F1 SCORES OF THE UN-DUP MODEL FOR USING DIFFERENT DATA PROCESSING METHODS

Classifiers	**Projects**	**Auc**			**F1**		
		Oversampling	Smote	Undersampling	Oversampling	Smote	Undersampling
RF	ActiveMQ	**0.83**	0.82	0.82	0.65	0.65	0.67
	Camel	0.85	0.85	0.85	0.51	0.54	0.55
	Derby	**0.81**	0.80	0.80	**0.35**	**0.44**	0.43
	Geronimo	0.89	0.89	0.90	0.40	**0.48**	0.43
	HBase	0.86	0.86	0.86	0.40	0.48	0.48
	Hadoop C.	**0.88**	0.86	0.86	0.33	**0.40**	0.33
	OpenJPA	**0.81**	0.80	0.80	0.34	**0.41**	0.39
	Pig	**0.85**	**0.85**	0.84	0.49	**0.52**	0.49
LR	ActiveMQ	0.82	0.81	0.82	0.67	0.64	0.67
	Camel	0.84	0.83	0.84	0.52	**0.53**	0.52
	Derby	0.76	0.75	0.76	0.38	**0.39**	0.38
	Geronimo	0.86	0.85	0.86	0.37	**0.39**	0.37
	HBase	**0.85**	0.84	0.84	0.40	0.42	0.42
	Hadoop C.	0.81	0.79	0.80	**0.26**	**0.28**	0.25
	OpenJPA	0.74	0.73	0.74	0.32	**0.33**	0.32
	Pig	0.82	0.81	0.82	0.45	0.41	0.45
NB	ActiveMQ	0.78	0.77	0.78	0.63	0.62	0.64
	Camel	0.81	0.81	0.81	0.51	0.51	0.51
	Derby	0.75	0.74	0.75	0.37	0.37	0.37
	Geronimo	**0.85**	0.84	0.84	0.36	0.36	0.37
	HBase	**0.83**	0.82	0.82	0.37	0.37	0.40
	Hadoop C.	0.75	0.75	0.76	0.20	0.21	0.21
	OpenJPA	0.74	0.73	0.74	0.32	0.31	0.32
	Pig	0.80	0.79	0.80	0.40	0.40	0.41
Avg.		**0.82**	0.81	0.81	0.42	**0.44**	0.43
P-value		>0.01	>0.05		>0.01	>0.01	
Cliff's delta		0.06(N)	-0.06(N)		-0.12(N)	0.02(N)	

The AUC and F1 Scores of the Oversampling and SMOTE Methods That are Higher Than Those of the Undersampling Method Are in Bold.

three classifiers. However, there is no statistically significant difference in the AUC and F1 scores of oversampling, SMOTE, and undersampling techniques. 2) For the Un-dup model, in terms of AUC and F1 scores, the undersampling technique does not perform worse than the other techniques in most cases. The oversampling technique outperforms the other techniques considering the average AUC scores across three classifiers. The SMOTE technique outperforms the other techniques on RF and LR classifiers with respect to F1 scores. However, the undersampling technique does not show a statistically significant difference in the effectiveness scores compared with oversampling and SMOTE techniques.

In summary, our analysis shows that different data rebalancing techniques have no substantial impact on the evaluation of In-dup and Un-dup models.

VI. Threats to Validity

A. Internal Validity

The potential errors of our experimental implementation are threats to internal validity. The first potential threat has to do with our code. We adopted an amount of technical and analytical work. Our code has been carefully tested and inspected, but there may still be errors that we did not notice. Our code and datasets are shared at our accompanying GitHub repository.[3]

Additionally, another threat is related to data labeling; we adopt the default diff algorithm *Myers* in RA-SZZ to calculate the location of the deleted lines of bug-fixing changes (i.e., bug-related lines). Many existing JIT defect prediction studies used the default *Myers* diff algorithm [14], [18], [36]. Note that our focus is to investigate if duplicate changes impact the validity of existing studies. Hence, our experimental setup is consistent with prior studies. Nurgroho *et al.* [61] observed that the *Histogram* algorithm performs better in identifying the bug-related lines than the default algorithm. Our datasets and code are made publicly available. Future research can explore if using different diff algorithms have an impact on our analysis.

Furthermore, among the eight studied projects, our manual analysis of the duplicate changes shows that our approach achieves nearly perfect accuracy (98.13%). The scale of mislabeling duplicate changes is 1% in most cases. Soares *et al.* [62] noted that less than 1% of refactors attempts introduce bugs. Hence, we believe that mislabeling duplicate changes are hardly affecting the distribution (i.e., the scale of buggy changes are smaller than clean duplicate changes) of the duplicate changes and are not likely to impact the conclusion of our article.

B. External Validity

The generality of our conclusions is a threat to external validity. We have analyzed eight open-source Java software projects containing more than 100 k changes. These projects comprise different application domains and are also widely used by many other researchers [17], [18], [36]. This guarantees data availability and openness of JIT software defect prediction community. Nevertheless, to extend our findings, it would be valuable to investigate software projects from other software systems written in different programming languages.

C. Construct Validity

The suitability of our evaluation measures is a threat to construct validity. We use AUC, MCC, F1-score, and Recall@20% in our case study, which are widely used in past studies [17], [24], [30], [31]. Additionally, to ensure that our experimental results are robust, we apply the Wilcoxon signed-rank test and Cliff's delta as statistical measures.

VII. Conclusion

In this article, we conducted a large-scale empirical study for investigating the impact of duplicate changes on JIT defect prediction.

Based on our experimental results, we observed the following. 1) The number of the duplicate changes varies across projects. The average proportion of duplicate changes is 13% among the studied projects. 2) The duplicate changes have a significant impact on the experience metric calculation of JIT defect prediction. For the 50% of the changes, removing duplicate changes among the studied projects decreases the experience metrics with an average of 6–55. 3) The duplicate changes obviously impact the evaluation of JIT defect prediction models. Removing duplicate changes among the studied projects can significantly improve the performance the classification JIT defect prediction models ranging from 1 to 125% concerning various performance measures (AUC, MCC, and F1-score). Additionally, in terms of Recall@20%, while removing duplicate changes among the studied projects can significantly improve the performance of CBS effort-aware JIT defect prediction models ranging from 4 to 68%, it can reduce the performance of OneWay effort-aware JIT defect prediction models ranging from 2 to 33%. 4) For the JIT defect prediction model interpretation, the duplicate changes lead to a significant impact on the second-ranked and third-ranked metrics. However, the most important metrics are not to be impacted by the duplicate changes.

[3][Online]. Available: https://github.com/deref007/Duplicate-change-TR

Our article is an important extension in studying the noisy data of JIT defect prediction. Considering the impact of duplicate changes, we recommend that future work should remove duplicate changes from the original historical changes of software repository. We hope that our study can inspire more researchers to conduct various case studies in the practice of the problem. This will further improve the practicality of JIT defect prediction technology.

References

[1] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1276–1304, Nov. 2012.

[2] 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 *Proc. 2015 IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, Florence, Italy, 2015, pp. 812–823.

[3] Y. Kamei and E. Shihab, "Defect prediction: Accomplishments and future challenges," in *Proc. Leaders Tomorrow Symp.: Future Softw. Eng.*, 2016, pp. 33–45.

[4] S. Kim, E. J. Whitehead, Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?," *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 181–196, Mar. 2008.

[5] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Tech. J.*, vol. 5, no. 2, pp. 169–180, Apr. 2000.

[6] Y. Kamei *et al.*, "A large-scale empirical study of just-in-time quality assurance," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 757–773, Jun. 2013.

[7] 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 Softw. Eng.*, vol. 21, no. 5, pp. 2072–2106, Oct. 2016

[8] M. Yan, X. Xia, Y. Fan, A. E. Hassan, D. Lo, and S. Li, "Just-in-time defect identification and localization: A two-phase framework," *IEEE Trans. Softw. Eng.*, early access, doi: 10.1109/TSE.2020.2978819.

[9] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proc. IEEE 31st Int. Conf. Softw. Eng.*, May 2009, pp. 78–88.

[10] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 485–496, Jul. 2008.

[11] T. Menzies *et al.*, "Local versus global lessons for defect prediction and effort estimation," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 822–834, Jun. 2013.

[12] X. Xia, D. Lo, S. J. Pan, N. Nagappan, and X. Wang, "HYDRA: Massively compositional model for cross-project defect prediction," *IEEE Trans. Softw. Eng.*, vol. 42, no. 10, pp. 977–998, Oct. 2016.

[13] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. Germán, and P. T. Devanbu, "The promises and perils of mining git," in *Proc. 6th IEEE Int. Work. Conf. Mining Softw. Repositories*, 2009, pp. 1–10.

[14] V. Kovalenko, F. Palomba, and A. Bacchelli, "Mining file histories: Should we consider branches?" in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2018, pp. 202–213.

[15] T. Dhaliwal, F. Khomh, Y. Zou, and A. E. Hassan, "Recovering commit dependencies for selective code integration in software product lines," in *Proc. 28th IEEE Int. Conf. Softw. Maintenance*, 2012, pp. 202–211.

[16] X. Yang, D. Lo, X. Xia, and J. Sun, "TLEL: A two-layer ensemble learning approach for just-in-time defect prediction," *Inf. Softw. Technol.*, vol. 87, pp. 206–220, 2017.

[17] Y. Fan, X. Xia, D. Alencar da Costa, D. Lo, A. E. Hassan, and S. Li, "The impact of changes mislabeled by SZZ on just-in-time defect prediction," *IEEE Trans. Softw. Eng.*, early access, doi: 10.1109/TSE.2019.2929761.

[18] E. C. Neto, D. A. da Costa, and U. Kulesza, "The impact of refactoring changes on the SZZ algorithm: An empirical study," in *Proc. IEEE 25th Int. Conf. Softw. Analysis, Evol. Reeng.*, 2018, pp. 380–390.

[19] D. W. Hosmer, Jr., S. Lemeshow, and R. X. Sturdivant, *Applied Logistic Regression*, 3rd ed. Hoboken, NJ, USA: Wiley, 2013. [Online]. Available: https://www.wiley.com/en-us/Applied Logistic Regression

[20] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.

[21] P. Domingos and M. Pazzani, "On the optimality of the simple Bayesian classifier under zero-one loss," *Mach. Learn.*, vol. 29, no. 2, pp. 103–130, Nov. 1997.

[22] J. Huang and C. X. Ling, "Using AUC and accuracy in evaluating learning algorithms," *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 3, pp. 299–310, Mar. 2005.

[23] P. Baldi, S. Brunak, Y. Chauvin, C. A. F. Andersen, and H. Nielsen, "Assessing the accuracy of prediction algorithms for classification: An overview," *Bioinformatics*, vol. 16, no. 5, pp. 412–424, 2000.

[24] Q. Song, Y. Guo, and M. Shepperd, "A comprehensive investigation of the role of imbalanced learning for software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 45, no. 12, pp. 1253–1269, Dec. 2019.

[25] Q. Huang, X. Xia, and D. Lo, "Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, Sep. 2017, pp. 159–170.

[26] W. Fu and T. Menzies, "Revisiting unsupervised learning for defect prediction," 2017, *arXiv:1703.00132*.

[27] H. Li, H. Xu, C. Zhou, X. Lu, and Z. Han, "Joint optimization strategy of computation offloading and resource allocation in multi-access edge computing environment," *IEEE Trans. Veh. Technol.*, vol. 69, no. 9, pp. 10214–10226, Sep. 2020.

[28] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, May 2005.

[29] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur.*, Aug. 2015, pp. 17–26.

[30] S. McIntosh and Y. Kamei, "[Journal first] are fix-inducing changes a moving target?: A longitudinal case study of just-in-time defect prediction," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng.*, May 2018, pp. 560–560.

[31] Y. Yang *et al.*, "Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 13–18, 2016, pp. 157–168.

[32] A. Bachmann, C. Bird, F. Rahman, P. T. Devanbu, and A. Bernstein, "The missing links: Bugs and bug-fix commits," in *Proc. 18th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2010, pp. 97–106.

[33] C. Bird *et al.*, "Fair and balanced?: Bias in bug-fix datasets," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf.*, 2009, pp. 121–130.

[34] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 481–490.

[35] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: How misclassification impacts bug prediction," in *Software-Engineering and Management 2015, Multikonferenz Der GI-Fachbereiche Softwaretechnik (SWT) Und Wirtschaftsinformatik (WI), FA WI-MAW*, Dresden, Germany, März 17–20, 2015, pp. 103–104.

[36] 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 Trans. Softw. Eng.*, vol. 43, no. 7, pp. 641–657, Jul. 2017.

[37] T. Mende, "Replication of defect prediction studies: Problems, pitfalls and recommendations," in *Proc. 6th Int. Conf. Predictive Models Softw. Eng.*, T. Menzies and G. Koru, Eds. Timisoara, Romania: ACM, Sep. 12–13, 2010, p. 5.

[38] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git," in *Proc 6th IEEE Int. Work. Conf. Mining Softw. Repositories*, May 2009, pp. 1–10.

[39] A. Bacchelli, M. Lanza, and R. Robbes, "Linking e-mails and source code artifacts," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng.*, 2010, pp. 375–384.

[40] M. Sasaki, S. Matsumoto, and S. Kusumoto, "Integrating source code search into git client for effective retrieving of change history," in *Proc. IEEE Workshop Mining Analyzing Interact. Hist.*, Mar. 2018, pp. 1–5.

[41] F. Qiu *et al.*, "JITO: A tool for just-in-time defect identification and localization," in *Proc. 28th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, P. Devanbu, M. Cohen, and T. Zimmermann, Eds. 2020, pp. 1586–1590.

[42] M. Yan, X. Xia, Y. Fan, D. Lo, A. E. Hassan, and X. Zhang, "Effort-aware just-in-time defect identification in practice: A case study at Alibaba," in *Proc. 28th ACM Joint Euro. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds. 2020, pp. 1308–1319.

[43] C. Tantithamthavorn and A. E. Hassan, "An experience report on defect modelling in practice: Pitfalls and challenges," in *Proc. 40th Int. Conf. Softw. Eng.: Softw. Eng. Pract.* 2018, pp. 286–295.

[44] J. H. Zar, "Spearman rank correlation," in *Encyclopedia of Biostatistics*, vol. 7. New Jersey, NY, USA: Wiley, 2005.

[45] R. L. Plackett, "Karl Pearson and the chi-squared test," *Int. Stat. Rev./Revue Int. Statistique*, vol. 51, no. 1, pp. 59–72, 1983.

[46] F. E. H. Jr, “RMS: Regression modeling strategies,” R Package Version 5.1-3, 2019.
[47] G. K. Rajbahadur, S. Wang, Y. Kamei, and A. E. Hassan, “The impact of using regression models to build defect classifiers,” in *Proc. 14th Int. Conf. Mining Softw. Repositories*, 2017, pp. 135–145.
[48] J. Jiarpakdee, C. Tantithamthavorn, and A. E. Hassan, “The impact of correlated metrics on the interpretation of defect models,” *IEEE Trans. Softw. Eng.*, vol. 47, no. 2, pp. 320–331, Feb. 2021.
[49] C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto, “The impact of class rebalancing techniques on the performance and interpretation of defect prediction models,”, 2018, arXiv:1801.10269.
[50] V. Kovalenko, F. Palomba, and A. Bacchelli, “Mining file histories: Should we consider branches?” in *Proc. 33rd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2018, pp. 202–213.
[51] F. Wilcoxon, “Individual comparisons by ranking methods,” *Biometrics Bull.*, vol. 1, no. 6, pp. 80–83, 1945.
[52] H. Abdi, “The Bonferonni and Sid´ak corrections for multiple comparisons,” *Encyclopedia Meas. Statist.*, vol. 3, pp. 103–107, 2007.
[53] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi, “Deepjit: An end-to-end deep learning framework for just-in-time defect prediction,” in *Proc. 16th Int. Conf. Mining Softw. Repositories,*, 2019, pp. 34–45.
[54] N. Cliff, *Ordinal Methods for Behavioral Data Analysis*. Hove, U.K.: Psychology Press, 2014. [Online]. Available: https://doi.org/10.4324/9781315806730
[55] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, “An empirical comparison of model validation techniques for defect prediction models,” *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 1–18, Jan. 2017.
[56] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, “The impact of automated parameter optimization on defect prediction models,” *IEEE Trans. Softw. Eng.*, vol. 45, no. 7, pp. 683–711, Jul. 2019.
[57] G. K. Rajbahadur, S. Wang, Y. Kamei, and A. E. Hassan, “The impact of using regression models to build defect classifiers,” in *Proc. 2017 IEEE/ACM 14th Int. Conf. Mining Softw. Repositories*, 2017, pp. 135–145.
[58] B. Adams, “On software release engineering,” *ICSE Technical Briefing*, 2012.
[59] M. Galar, A. Fernández, E. B. Tartas, H. B. Sola, and F. Herrera, “A review on ensembles for the class imbalance problem: Bagging-, boosting-, and hybrid-based approaches,” *IEEE Trans. Syst. Man Cybern. Part C*, vol. 42, no. 4, pp. 463–484, Jul. 2012.
[60] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, “SMOTE: Synthetic minority over-sampling technique,” *J. Artif. Intell. Res.*, vol. 16, no. 1, pp. 321–357, 2002.
[61] Y. S. Nugroho, H. Hata, and K. Matsumoto, “How different are different diff algorithms in git?,” *Empir. Softw. Eng.*, vol. 25, no. 1, pp. 790–823, 2020.
[62] G. Soares, R. Gheyi, and T. Massoni, “Automated behavioral testing of refactoring engines,” *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 147–162, Feb. 2013.

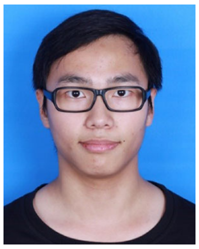
Ruifeng Duan received the B.S. degree in communication engineering from Henan University, Kaifeng, China, in 2018, and the M.S. degree in electronic and communication engineering from the University of Science and Technology Beijing (USTB), Beijing, China, in 2021.

He is currently with Yunhe (Henan) Information Technology Company, Ltd, Zhengzhou, China. His research interests include empirical software engineering, mining software repositories, communication network security, and machine learning.

Haitao Xu (Member, IEEE) received the B.S. degree in communication engineering from Sun Yat-Sen University, Guangzhou, China, in 2007, the M.S. degree in communication system and signal processing from the University of Bristol, Bristol, U.K., in 2009, and the Ph.D. degree in communication and information system from the University of Science and Technology Beijing, Beijing, China, in 2014.

He was with the Department of Communication Engineering, University of Science and Technology Beijing as a Postdoc, from 2014 to 2016. He is currently an Associate Professor with the Department of Communication Engineering, University of Science and Technology Beijing. His research interests include wireless resource allocation and management, wireless communications and networking, dynamic game and mean field game theory, big data analysis, and security.

Dr. Xu has co-edited a book titled *Security in Cyberspace* and coauthored more than 50 technical papers. He has been serving in the organization teams of some international conferences, e.g., CCT2014 and CCT2015.



Yuanrui Fan is currently working toward the Ph.D. degree in software engineering with the College of Computer Science and Technology, Zhejiang University, Hangzhou, China.

His research interests include mining software repositories and empirical software engineering.

Meng Yan received the B.S., M.S., and Ph.D. degrees in software engineering from Chongqing University, Chongqing, China, in 2011, 2013, and 2017, respectively.

He is currently a tenure-track Assistant Professor with the School of Big Data and Software Engineering, Chongqing University. Proir to joining Chongqing University, he was a Postdoctoral Research Fellow with Zhejiang University, Hangzhou, China. He was a Visiting Scholar with Monash University, Melbourne, Australia, in 2019. His current research interests include mining software repositories, empirical software engineering, and software analytics.