Just-in-Time Defect Prediction for Android Apps via Imbalanced Deep Learning Model

Kunsong Zhao
Wuhan University
Wuhan, China
kszhao@whu.edu.cn

Zhou Xu*
Chongqing university
Chongqing, China
zhouxullx@cqu.edu.cn

Meng Yan
Chongqing University
Chongqing , China
mengy@cqu.edu.cn

Yutian Tang
ShanghaiTech University
Shanghai, China
tangyt1@shanghaitech.edu.cn

Ming Fan
Xi'an Jiaotong University
Xi'an, China
mingfan@mail.xjtu.edu.cn

Gemma Catolino
Tilburg University
Tilburg, The Netherlands
G.Catolino@tilburguniversity.edu

ABSTRACT

Android mobile apps have played important roles in our daily life and work. To meet new requirements from users, the mobile apps encounter frequent updates, which involves in a large quantity of code commits. Previous studies proposed to apply Just-in-Time (JIT) defect prediction for mobile apps to timely identify whether new code commits can introduce defects into apps, aiming to assure the quality of mobile apps. In general, the number of defective commit instances is much fewer than that of clean ones, in other words, the defect data is class imbalanced. In this work, we propose a novel Imbalanced Deep Learning model, called IDL, to conduct JIT defect prediction task for Android mobile apps. More specifically, we introduce a state-of-the-art cost-sensitive cross-entropy loss function into the deep neural network to learn the high-level feature representation, in which the loss function alleviates the class imbalance issue by taking the prior probability of the two types of classes into account. We conduct experiments on a benchmark defect data consisting of 12 Android mobile apps. The results of rigorous experiments show that our proposed IDL model performs significantly better than 23 comparative imbalanced learning methods in terms of Matthews correlation coefficient performance indicator.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

JIT defect prediction, mobile apps, imbalanced learning

*Corresponding author.

SAC '21, March 22–26, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8104-8/21/03…$15.00
https://doi.org/10.1145/3412841.3442019

ACM Reference Format:
Kunsong Zhao, Zhou Xu, Meng Yan, Yutian Tang, Ming Fan, and Gemma Catolino. 2021. Just-in-Time Defect Prediction for Android Apps via Imbalanced Deep Learning Model. In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC '21), March 22–26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3412841.3442019

1 INTRODUCTION

Software has already became an indispensable part of people's daily life and work. However, existing software products unavoidably contain defects due to the increasing software scale and complexity [37]. The defects can result in a series of negative effects which are unexpected. Since the impossibility of developing software products without defects, it is vital to propose suitable techniques to detect defects earlier. Software defect prediction emerges to employ different techniques, such as machine learning methods, to predict defective code regions, which attracts many researchers to contribute to it for software quality assurance [30][32].

Recently, mobile apps, especially Android apps, become fashionable. It has the advantage that once developers release the update of apps, users can download them immediately from App Stores [18]. Mobile apps are usually frequently updated to meet the functional requirements of users. This characteristic is inevitable to introduce defects into new versions of mobile apps, which hinders the app quality. Thus, early detection of defects is an urgent issue during the process of development and maintenance for mobile apps.

Many previous studies [19][24][29] for defect prediction were based on class level, which was lack of immediate feedback for defective-prone codes. To overcome this drawback, researchers proposed Just-in-Time (JIT) defect prediction, which is at commit level [10][33][34]. JIT defect prediction aims at identifying whether a new code commit introduces defects, which can offer timely feedback for developers to detect defects early. Considering this advantage, it is appropriate to apply JIT defect prediction to software with the characteristic of frequent updates (such as mobile apps) which involve in a large number of code commits. If a new commit instance introduces defects into the app, this instance is regarded as defective, otherwise, clean. Catolino et al. [3][4] took the first attempt to build JIT defect prediction models for Android mobile apps based on features derived from the commit information.

In general, the feature representation quality, to a large extent, impacts the classification performance. Thus, learning the high-level feature representation has the potential to promote the performance of defect prediction model. Moreover, for the defect data, the number of defective commit instances is much fewer than those of clean, in other words, the defect data is class imbalanced. Since the data imbalance characteristic usually deteriorates the performance of the classification model, it is crucial to deal with this issue for performance improvement. In this work, we propose a novel **I**mbalanced **D**eep **L**earning method, short for IDL, to address the above two issues. More specifically, as the deep learning method has the powerful ability for feature extraction, we use the **D**eep **N**eural **N**etwork (DNN) model to learn the high-quality feature representation for the defect data of Android mobile apps. However, since traditional cross-entropy loss function in the DNN model holds that the losses of two classes have the same impact on the total loss, it is not suitable for the imbalanced defect data. To overcome this drawback, we introduce a novel **C**ost-**S**ensitive **C**ross-**E**ntropy (CSCE) loss function into DNN. This loss function takes the prior probability of the two kinds of classes into consideration when calculating the cross-entropy loss [1]. In other words, it uses the weighted technique to compensate the class imbalance between defective and clean commit instances.

To evaluate the prediction performance of our proposed JIT defect prediction method IDL, in this work, we conduct experiments on a benchmark dataset that includes 12 Android mobile apps and employ **M**atthews **C**orrelation **C**oefficient (MCC) as the performance indicator. Across the 12 apps, our IDL method achieves the average MCC value of 0.307 and obtains average improvements by 13.0%, 20.5%, and 29.5% compared with the eight sampling-based, six ensemble-based, and nine cost-sensitive based imbalanced learning methods, individually. The statistic test results show that our IDL method significantly outperforms 23 comparative methods in terms of MCC indicator.

The main contributions of this paper are summarized as follows:

(1) We propose a novel deep learning model IDL to learn effective feature representation for the defect data and relieve the negative impact of class imbalance of the defect data by employing a weighted cross-entropy based loss function.

(2) We evaluate our IDL model on the defect data consisting of 12 Android mobile apps and conduct the statistic test to analyze the experimental results. The results show that our method significantly outperforms 23 comparative methods.

2 RELATED WORK

2.1 Defect Prediction for Android Mobile Apps

Ortu et al. [20] analyzed defect characteristics from the logs of traditional software and mobile apps using natural language text classification techniques. Their experimental results showed that the High-Priority and Low-Priority defects in the domains of traditional and mobile software were different. Khomh et al. [12] proposed three metrics to capture the patterns of failure occurrences for defect prediction. They conducted experiments on 18 versions of an enterprise mobile app and the results showed that these metrics predicted defects with a shorter time. Scandariato et al. [23] employed a SVM model to identify and analyze vulnerable components of apps using source code metrics. They analyzed a popular application in the Android Market and the results showed that their model achieved higher accuracy and precision. Ricky et al. [22] proposed a SVM method to predict defects on mobile apps. Their experimental results on five datasets showed that their SVM method achieved better performance than decision trees. Malhotra et al. [16] proposed a framework for identifying defective classes using object-oriented metrics. They conducted experiments on seven mobile apps and the results showed that there existed performance differences among 18 classifiers. Kaur et al. [11] explored process metrics for defect prediction on an open source mobile app. Their experimental results showed that models with process metrics achieved better performance than that with code metrics.

Since the timely feedback characteristic of JIT defect prediction, it is especially suitable for frequently updated mobile apps. However, there are few studies for identifying JIT defects on mobile apps. Catolino et al. [3][4] took the first attempt to explore the JIT defect prediction for mobile apps and then compared the impacts of multiple machine learning methods and ensemble learning techniques. They extracted the defect data of mobile apps from the COMMIT GURU platform and experimental results showed that Naive Bayes performed significantly better than other classifiers.

Different from most previous studies that focused on the traditional mobile software at the class level, in this work, we study JIT defect prediction at the code change or commit level. Different from [3][4] that explored the traditional machine learning classifiers for JIT defect prediction on mobile apps, we propose a novel method to learn effective feature representation for this task.

2.2 Deep Learning in Defect Prediction

Yang et al. [35] proposed the Deeper, which employed a deep belief network for defect prediction. Their experimental results on six projects with 137,417 changes showed significantly better performance than those of Kamei et al.'s approach on most projects. Li et al. [13] proposed a method that extracted features from ASTs and then employed Convolutional Neural Networks (CNNs) for feature representation learning. They conducted experiments on seven open source projects and the results showed significant performance improvement of their method. Phan et al. [21] learned semantic features employing a directed graph-based CNNs for defect prediction. They conducted experiments on four projects and the results showed the significant superiority compared with the six baseline methods. Manjula et al. [17] proposed a novel methods that employed the genetic algorithm and deep neural network for feature representation learning and classification. Their experimental results on PROMISE dataset showed the better accuracy than comparative methods. Xu et al. [31] proposed a method, called LDFR, which employed the deep neural network with a cross-entropy loss function for predicting defective modules. They conducted experiments on 27 project versions and the results showed that LDFR presented significant superiority.

Different from above studies that used deep learning techniques for defect prediction on traditional software projects, we take the first attempt to introduce deep learning technique into JIT defect prediction on Android mobile apps by considering both feature representation learning and class imbalance learning.

3 OUR PROPOSED IDL MODEL

As our proposed IDL method incorporates an improved cross-entropy loss function into a DNN method, in this section, we first introduce the original cross-entropy loss function, then detail its improved version, i.e., the CSCE loss function, and last illustrate the DNN method.

3.1 Cross-Entropy Loss Function

Assume that the commit instance set of mobile apps is defined as $S = \{(X, Y)|X \in \mathbb{R}^{n\times m}, Y \in \mathbb{R}^n\}$, where $X = [x_1, x_2, \ldots, x_n]$ is the commit instance set, $x_i = [x_{i1}, x_{i2}, \ldots, x_{im}]$ is the feature set of the i-th commit instance, and $Y = \{y_i|i = 1, 2, \ldots, n\}$ is the corresponding label set. The goal of the label classification is to make the output labels of the commit instances generated by an unknown learning function $f(x)$ as close as possible to the real labels. Here, we define a generalized model $f(x|\theta)$ to obtain the output, where θ is a parameter set of the model. The parameter θ can be estimated by the cross-entropy loss function which is a convex function. The formula is defined as follows:

$$\ell(\theta) = \frac{1}{n}\sum_{i=1}^{n}[-y_i \log(\hat{y}_i) - (1 - y_i)\log(1 - \hat{y}_i)] \quad (1)$$

where $\hat{y}_i = f(x_i|\theta)$ is the output of model corresponding to the i-th commit instance. When $y_i = 0$, $\ell(\theta)$ is equal to $-\log(1 - \hat{y}_i)$, and when $y_i = 1$, $\ell(\theta)$ is equal to $-\log(\hat{y}_i)$. The $\hat{y}_i$ tends to y_i with the loss decreasing logarithmically [1].

On the balanced data, the losses from $-\log(\hat{y}_i)$ and $-\log(1 - \hat{y}_i)$ account for half of the total loss for a specific model output $\hat{y}_i$, individually. However, for the imbalanced data, the loss from the instances in the majority class has larger impacts on the total loss $\ell(\theta)$. The reason is that it ignores which class the instances causing the loss belong to when calculating the total loss.

3.2 Cost-Sensitive Cross-Entropy Loss Function

From the above analysis, the traditional cross-entropy loss function could not work well on the imbalanced data. To alleviate the class imbalanced issue, the key point lies in how to assign weights to the two kinds of losses, i.e., $-y_i \log(\hat{y}_i)$ and $-(1 - y_i)\log(1 - \hat{y}_i)$. For this purpose, in this work, we introduce an improved version of the cross-entropy loss function [1].

Since the prior probability ratio, such as the ratio of the number of defective commit instances to the total number of commits instances, is helpful to achieve a balance between different classes, in this work, we introduce this term into the cross-entropy loss function, called **C**ost-**S**ensitive **C**ross-**E**ntropy (CSCE) loss function, to compensate the imbalance of the commit defect data, which is formalized as follows:

$$\ell(\theta) = \frac{1}{n}\sum_{i=1}^{n}[-\lambda y_i \log(\hat{y}_i) - (1 - \lambda)(1 - y_i)\log(1 - \hat{y}_i)] \quad (2)$$

where λ = M/N is the percentage of defective instances, M is the number of commit instances with the defective label ($y_i = 1$), and N is the total number of the commit instances. The previous study [1] has proved that the CSCE loss rate was almost constant when the prior probability was taken into account. This would lead to a balance between the two different classes.

3.3 Deep Neural Network

DNN consists of three kinds of network layers, including the input layer, the hidden layer, and the output layer. In general, the first layer is the input layer with many units, which receives the input feature vectors. The last layer is the output layer, which outputs the results generated by the DNN model. The hidden layer consists of one or more layers. Different from the basic multi-layer perceptron that only has one unit in the output layer, DNN extends the network structure with many hidden layers and the output layer with one or more units, which improves the ability of representation learning. In DNN structure, the network units between layers are fully connected and the network units in the same layer are not connected. There are two main steps in the training process of DNN. The first step is the forward propagation, in which each layer takes the original vectors, the weighted coefficient matrix, and the bias vectors as inputs, and then outputs the results of linear operation. In the second step, the back propagation algorithm is applied to optimize the model parameters in each layer. The aim is to make the model output values as close as possible to the real labels.

Given a set of commit instances in mobile apps, we input feature vectors of these instances into the first layer of DNN. After the process of hidden layers and the output layer, the model calculates the total loss between the predicted labels and the true labels for commit instances using the CSCE loss function. Then the back propagation is employed to obtain the optimal parameters. These two processes terminate until the total loss attains a certain threshold. The training procedure of DNN is illustrated in Fig. 1.

4 EXPERIMENTAL SETUP

4.1 Research Questions

As many class imbalance learning methods are based on sampling-based, ensemble-based, cost-sensitive based ones [7], to evaluate our proposed IDL method that aims to alleviate the class imbalance issue of the defect data of mobile apps, in this work, we design the following three research questions (RQs).

RQ1: Is our IDL method superior to sampling-based imbalanced learning methods?

Sampling-based methods relieve the class imbalance issue by adjusting the number of positive and negative samples. This question is designed to explore whether our IDL method is superior to sampling-based methods for JIT defect prediction on Android apps.

RQ2: Does our IDL method perform better than ensemble-based imbalanced learning methods?

Ensemble learning methods deal with the imbalanced data by creating multiple base models and then integrating the predictions of these base models to improve overall performance. This question is designed to explore whether our IDL method achieves better JIT defect prediction performance than ensemble-based methods.

RQ3: How effective is our IDL method compared with cost-sensitive based imbalanced learning methods?

Cost-sensitive based methods alleviate the class imbalance issue by assigning higher misclassification costs with instances in the minority class and seeking to minimize the high cost errors. This question is designed to explore whether our IDL method is more effective than cost-sensitive based methods to improve the JIT defect prediction performance on Android apps.

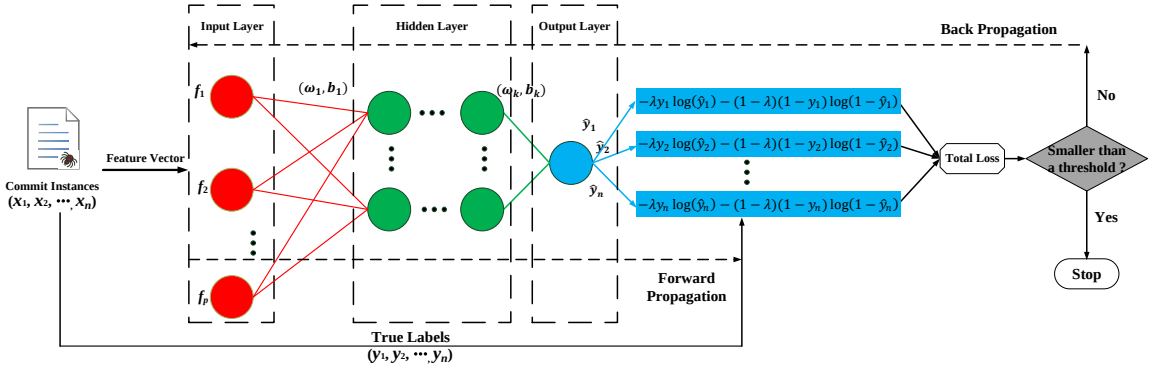

Figure 1: The iterative process of DNN.

4.2 Dataset

In order to evaluate the performance of our IDL method, in this work, we employ a benchmark dataset with 12 Android mobile apps denoted by a recent study [4]. Readers can refer to [4] for the description of these apps. Table 1 summarized the basic information of these apps, including lines of the code (# LOC), the total number of commit instances (# TC), the number of defective instances (# DC), the number of clean instances (# CC), and the ratio of defective instances (% DR). If a new commit instance introduces the defects, this instance is deemed as defective, otherwise, clean. The code lines of these apps are between 9,506 and 275,637, which means that these apps have different scales. The commit instances of the mobile apps in the benchmark dataset are characterized by a feature set from different scopes, such as *History*, *Size*, and *Diffusion*. We follow the original work [4] to use the same six features which have been proved to be the most useful ones to identify defective commit instances in the context of JIT defect prediction for mobile apps. Table 2 presents the brief descriptions of the six features.

Table 1: The basic information of the 12 apps

Project	# LOC	# TC	# DC	# CC	% DR
Firewall	77,243	1025	414	611	40.4%
Alfresco	152,047	1004	214	790	21.3%
Sync	275,637	209	62	147	30.0%
Wallpaper	35,917	588	94	494	16.0%
Keyboard	114,784	2971	819	2152	27.6%
Apg	151,204	3780	1304	2476	34.5%
Secure	98,768	2579	853	1726	33.1%
Facebook	103,802	548	180	368	32.8%
Kiwix	32,598	1373	350	1023	25.5%
Cloud	115,169	3700	830	2870	22.4%
Turner	30,943	164	23	141	14.0%
Reddit	9,506	222	60	162	27.0%

4.3 Performance Indicator

Considering that our defect data is class imbalanced, we employ the MCC as the performance indicator to evaluate the effectiveness of our IDL method. MCC is a comprehensive performance measure derived from the Pearson correlation coefficient which takes all four basic terms TP, FP, FN, and TN into consideration. The formula of MCC is expressed as

$$\text{MCC} = \frac{\text{TP} \times \text{TN} - \text{FP} \times \text{FN}}{\sqrt{(\text{TP}+\text{FP})(\text{TP}+\text{FN})(\text{TN}+\text{FP})(\text{TN}+\text{FN})}} \quad (3)$$

where TP, TN, FP, and FN denotes true positive, true negative, false positive, and false negative, respectively.

MCC ranges from -1 to 1. The lager indicator value means that it achieves better prediction performance. MCC = -1 denotes that the predicted value is the complete opposite of the true value. MCC = 1 denotes that the predicted value is exactly the same as the true value. MCC = 0 denotes that the prediction performance of model is equal to random guesses. Previous studies [26][36] have suggested that MCC is the most appropriate performance indicator for defect prediction task compared with F-measure and AUC on the imbalanced dataset.

Table 2: The brief descriptions of features

Scope	Feature	Description
History	NUC	Number of unique change to modified files
	NDEV	Number of developers working on the files
Size	LD	Lines of code deleted
	LA	Lines of code added
Diffusion	NF	Number of modified files
	ND	Number of modified directories

4.4 Data Partition

In this work, we employ the stratified sampling method to generate the training set and test set to ensure that the two sets have the same instance ratio of the two kinds of labels. More specifically, for each mobile app, we take the data that merges half of the defective commit instances and half of the clean commit instances as the training set and the remainder as the test set to run our IDL method and the comparative methods. After that, we exchange the training set and test set and then run these methods again. For each data partition, we can obtain two results. To reduce the negative impacts

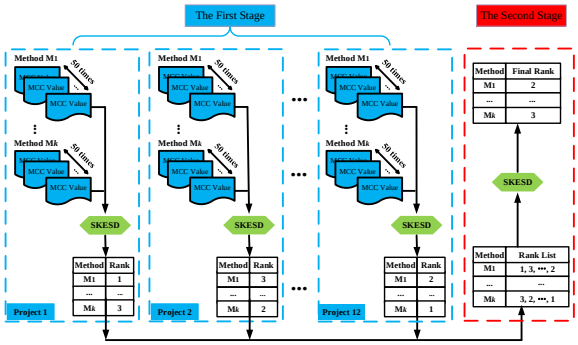

Figure 2: The operational process of SKESD test.

of the random partition on our experimental results, we repeat this procedure 25 times. Thus, we obtain total $25 \times 2 = 50$ MCC values. In this work, we report the average MCC value and the corresponding standard deviation.

4.5 Parameter Settings

In this work, we follow the previous study [31] to set the structure of DNN as one input layer and two hidden layers with 10 hidden units, following with one output layer with one unit. For the hyper-parameters, we set the batch size as 16, and the iterations as 10,000. Moreover, we apply the RMSProp algorithm [5] to optimize our DNN model. In each iteration, we set the learning rate as 0.1 with the decay rate as 0.99. In addition, we employ the exponential moving average model [5] with the decay rate as 0.99 for the learning rate. When calculating the loss, the L2 regularization is applied to reduce the overfitting. The training process is automatically terminated until the total loss is less than 0.05.

4.6 Statistic Test

In this work, we apply a state-of-the-art method, namely **S**cott-**K**nott **E**ffect **S**ize **D**ifference (short for SKESD) test [27], to analyze the significant differences between our IDL method and the comparative methods. The original Scott-Knott test uses a cluster analysis algorithm to divide all the methods with significant differences into different groups. However, this test method requires the data with normal distribution and cannot well handle with the groups with the negligible effect size of significant differences. To overcome these two limitations, Tantithamthavorn et al. [27] proposed an improved version, called SKESD test that applied the log-transforming to preprocess the results of the performance indicator and quantified the effect size by applying Cohen's delta. In this work, we perform the SKESD test with two stages to conduct the significant analysis. The process of SKESD test is demonstrated in Fig. 2. In the first stage, we take all the performance indicator values of each method on each mobile app as inputs to the SKESD test and obtain the output of the corresponding rank list of each method on each app. In the second stage, we take the output results from the previous processing as inputs and then get the final rank of each method across all mobile apps. The lower ranking value of a method means that it obtains better performance.

5 PERFORMANCE EVALUATION

5.1 Answer to RQ1: the prediction performance of our IDL method and the sampling-based imbalanced learning methods

Methods: To answer this question, we choose eight sampling methods as baseline methods, including **R**andom **O**ver-**S**ampling (ROS), **R**andom **U**nder-**S**ampling (RUS), The **S**ynthetic **M**inority **O**ver-sampling **T**echnique (SMOT), **SMOT** with **E**dited **N**earest neighbors (SMOTEN), **SMOT** with **T**omek links (SMOTT), **SVM** algorithm with **SMOT** (SVMSMOT), **SMOT** with **B**orderline samples (SMOTB), and over-sampling using **ADA**ptive **SYN**thetic sampling (ADASYN). We also use random forest as the basic classifier, which is widely used in software defect prediction task [6][2][28][15].

Results: Table 3 reports the average MCC value and the corresponding standard deviation of our IDL method and the eight comparative sampling-based imbalanced learning methods. In the table, the values in bold denote the best performance for each app or the best average value across all apps. Fig. 3 visualizes the statistic test result of SKESD. From the table and the figure, we can draw the following observations.

First, our IDL method obtains better performance on 8 out of 12 apps compared with the eight baseline methods. The average MCC value by our IDL method over all apps achieves improvements by 8.9%, 10.8%, 12.5%, 12.9%, 12.9%, 12.0%, 16.3%, and 17.6% compared with ROS, RUS, SMOT, SMOTEN, SVMSMOT, SMOTT, SMOTB, and ADASYN, individually. Our IDL method obtains the best average indicator value and achieves an average improvement by 13.0%.

Second, our IDL method ranks the first and has significant differences compared with the eight sampling-based imbalanced learning methods in terms of MCC indicator.

Different from the sampling-based methods which need change the distribution of commit instances to balance the defect data, our IDL method uses weights strategy to deal with the imbalanced issue. To sum up, our IDL method performs significantly better than the comparative sampling-based methods for predicting JIT defects on Android mobile apps.

5.2 Answer to RQ2: the prediction performance of our IDL method and the ensemble-based imbalanced learning methods

Methods: To answer this question, we choose six ensemble methods for comparison, including **Bag**ging (Bag), **B**alanced **Bag**ging (BBag), **B**alanced **R**andom **F**orest (BRF), **EasyEn**semble (EasyEn), **Ada**ptive **B**oost (AdaB), and **R**andom **U**nder-**S**ampling **B**oost (RUSB).

Results: Table 4 reports the average MCC value and the corresponding standard deviation of our IDL method and the six comparative ensemble-based imbalanced learning methods. Fig. 4 visualizes the statistic test result of SKESD. From the table and the figure, we can draw the following findings.

First, our IDL method obtains better performance on 9 out of 12 apps compared with the six baseline methods. The average MCC value by our IDL method over all apps achieves improvements by 6.6%, 24.3%, 17.2%, 8.1%, 37.7%, and 29.0% compared with BRF, EasyEn, Bag, Bbag, AdaB, and RUSB, individually. Our IDL method

Table 3: The Average MCC of our IDL method and sampling-based methods

Project	ROS	RUS	SMOT	SMOTEN	SVMSMOT	SMOTT	SMOTB	ADASYN	IDL
Firewall	0.162±(0.153)	0.146±(0.154)	0.166±(0.149)	0.154±(0.160)	0.157±(0.156)	0.160±(0.154)	0.156±(0.146)	0.159±(0.146)	**0.249**±(0.057)
Alfresco	0.347±(0.126)	0.327±(0.151)	0.340±(0.147)	0.313±(0.141)	0.334±(0.122)	0.345±(0.141)	0.326±(0.145)	0.315±(0.165)	**0.383**±(0.053)
Sync	0.402±(0.089)	0.390±(0.090)	0.357±(0.125)	0.353±(0.083)	0.383±(0.106)	0.352±(0.107)	0.339±(0.137)	0.339±(0.142)	**0.410**±(0.058)
Wallpaper	0.221±(0.055)	0.210±(0.067)	**0.232**±(0.050)	0.218±(0.051)	0.221±(0.059)	0.226±(0.065)	0.223±(0.066)	0.223±(0.067)	0.157±(0.057)
Keyboard	0.212±(0.127)	0.213±(0.123)	0.199±(0.120)	0.269±(0.070)	0.233±(0.123)	0.220±(0.115)	0.181±(0.122)	0.173±(0.123)	**0.288**±(0.025)
Apg	0.215±(0.123)	0.210±(0.122)	0.234±(0.121)	**0.235**±(0.128)	0.171±(0.146)	0.223±(0.120)	0.206±(0.119)	0.214±(0.114)	0.205±(0.092)
Secure	0.229±(0.162)	0.221±(0.157)	0.230±(0.155)	0.207±(0.154)	0.163±(0.160)	0.226±(0.154)	0.224±(0.155)	0.221±(0.154)	**0.275**±(0.073)
Facebook	0.220±(0.199)	0.226±(0.203)	0.218±(0.203)	0.210±(0.193)	0.219±(0.204)	0.223±(0.196)	0.217±(0.198)	0.220±(0.201)	**0.282**±(0.098)
Kiwix	0.258±(0.119)	0.258±(0.116)	0.231±(0.129)	0.213±(0.139)	0.234±(0.120)	0.220±(0.132)	0.232±(0.123)	0.233±(0.125)	**0.311**±(0.021)
Cloud	0.336±(0.073)	0.337±(0.082)	0.299±(0.125)	0.292±(0.129)	0.278±(0.128)	0.294±(0.135)	0.251±(0.189)	0.265±(0.157)	**0.385**±(0.018)
Turner	0.257±(0.141)	0.259±(0.152)	0.254±(0.165)	0.274±(0.128)	**0.329**±(0.111)	0.280±(0.124)	0.307±(0.114)	0.268±(0.150)	0.303±(0.143)
Reddit	0.523±(0.118)	0.531±(0.097)	0.521±(0.116)	0.525±(0.090)	**0.542**±(0.080)	0.521±(0.099)	0.511±(0.118)	0.505±(0.121)	0.438±(0.053)
Average	0.282±(0.098)	0.277±(0.100)	0.273±(0.092)	0.272±(0.092)	0.272±(0.106)	0.274±(0.092)	0.264±(0.092)	0.261±(0.089)	**0.307**±(0.081)

Table 4: The Average MCC of our IDL method and ensemble-based methods

Project	BRF	EasyEn	Bag	BBag	AdaB	RUSB	IDL
Firewall	0.194±(0.127)	0.167±(0.114)	0.216±(0.091)	0.222±(0.096)	0.175±(0.101)	0.170±(0.098)	**0.249**±(0.057)
Alfresco	0.327±(0.149)	0.279±(0.190)	0.327±(0.092)	0.331±(0.067)	0.339±(0.075)	0.272±(0.117)	**0.383**±(0.053)
Sync	0.396±(0.081)	0.361±(0.120)	0.335±(0.081)	0.341±(0.070)	0.233±(0.136)	0.328±(0.107)	**0.410**±(0.058)
Wallpaper	**0.203**±(0.079)	0.141±(0.078)	0.092±(0.058)	0.170±(0.047)	0.099±(0.089)	0.094±(0.086)	0.157±(0.057)
Keyboard	0.252±(0.089)	0.190±(0.129)	0.211±(0.075)	0.242±(0.079)	0.196±(0.077)	0.201±(0.069)	**0.288**±(0.025)
Apg	0.142±(0.139)	0.102±(0.161)	0.215±(0.042)	**0.226**±(0.045)	0.169±(0.113)	0.181±(0.094)	0.205±(0.092)
Secure	0.244±(0.135)	0.189±(0.158)	0.213±(0.068)	0.206±(0.068)	0.199±(0.121)	0.213±(0.100)	**0.275**±(0.073)
Facebook	0.269±(0.188)	0.242±(0.153)	0.274±(0.081)	0.275±(0.079)	0.227±(0.149)	0.259±(0.119)	**0.282**±(0.098)
Kiwix	0.310±(0.088)	0.255±(0.108)	0.242±(0.053)	0.241±(0.049)	0.207±(0.086)	0.238±(0.060)	**0.311**±(0.021)
Cloud	0.371±(0.085)	0.369±(0.094)	0.361±(0.030)	0.37±(0.031)	0.316±(0.077)	0.298±(0.075)	**0.385**±(0.018)
Turner	0.234±(0.146)	0.218±(0.173)	0.205±(0.154)	0.263±(0.117)	0.159±(0.167)	0.199±(0.160)	**0.303**±(0.143)
Reddit	0.510±(0.103)	0.455±(0.134)	0.453±(0.068)	**0.521**±(0.064)	0.353±(0.168)	0.408±(0.108)	0.438±(0.053)
Average	0.288±(0.097)	0.247±(0.099)	0.262±(0.090)	0.284±(0.091)	0.223±(0.074)	0.238±(0.079)	**0.307**±(0.081)

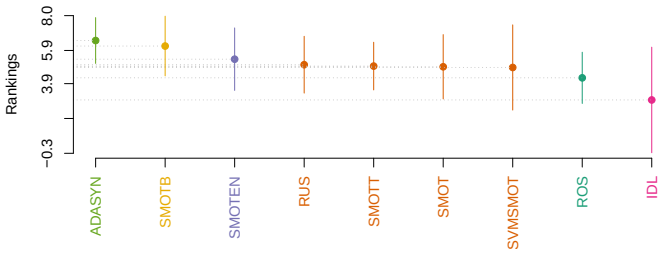

Figure 3: SKESD test for our IDL method and sampling-based methods.

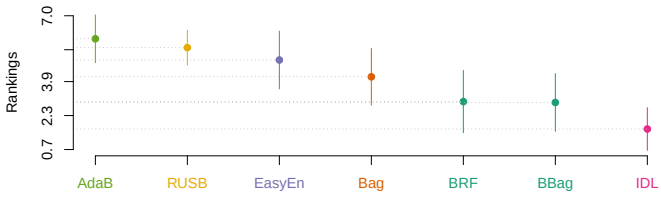

Figure 4: SKESD test for our IDL method and ensemble-based methods.

obtains the best average indicator value and achieves an average improvement by 20.5%.

Second, our IDL method ranks the first and has significant differences compared with the six ensemble-based imbalanced learning methods in terms of MCC indicator.

Different from the ensemble-based methods which combine the outputs of multiple classification models, our IDL method uses feature representation learning for performance improvement. In summary, our IDL method is more effective to obtain significantly better performance than ensemble-based methods for predicting JIT defects on Android mobile apps.

5.3 Answer to RQ3: the prediction performance of our IDL method and the cost-sensitive based imbalanced learning methods

Methods: To answer this question, we employ three cost-sensitive based methods, including the **S**ystematically developed **F**orest of multiple decision trees (SF) [9], **C**ost-sensitive based decision **F**orest

Table 5: The Average MCC of our IDL method and cost-sensitive based methods

Project	SFSV	SFDV	SFCV	CFSV	CFDV	CFCV	BFSV	BFDV	BFCV	IDL
Firewall	0.206±(0.128)	0.181±(0.142)	0.150±(0.176)	0.172±(0.122)	0.175±(0.120)	0.171±(0.142)	0.112±(0.143)	0.107±(0.144)	0.113±(0.142)	**0.249**±(0.057)
Alfresco	0.359±(0.089)	0.345±(0.117)	0.270±(0.195)	0.329±(0.115)	0.322±(0.115)	0.315±(0.161)	0.277±(0.157)	0.261±(0.170)	0.284±(0.169)	**0.383**±(0.053)
Sync	0.364±(0.118)	0.375±(0.118)	0.288±(0.164)	0.349±(0.189)	0.360±(0.183)	0.319±(0.189)	0.249±(0.226)	0.212±(0.225)	0.283±(0.172)	**0.410**±(0.058)
Wallpaper	0.028±(0.061)	0.036±(0.072)	**0.175**±(0.097)	0.101±(0.074)	0.089±(0.071)	0.128±(0.076)	0.140±(0.074)	0.136±(0.067)	0.150±(0.064)	0.157±(0.057)
Keyboard	0.246±(0.074)	0.240±(0.063)	0.192±(0.126)	0.206±(0.084)	0.198±(0.080)	0.224±(0.098)	0.167±(0.122)	0.152±(0.127)	0.188±(0.114)	**0.288**±(0.025)
Apg	0.119±(0.169)	0.100±(0.176)	0.086±(0.155)	0.099±(0.183)	0.110±(0.184)	0.084±(0.166)	0.093±(0.168)	0.111±(0.149)	0.095±(0.153)	**0.205**±(0.092)
Secure	0.243±(0.121)	0.244±(0.112)	0.214±(0.158)	0.278±(0.092)	**0.279**±(0.087)	0.252±(0.132)	0.235±(0.113)	0.235±(0.118)	0.228±(0.124)	0.275±(0.073)
Facebook	0.318±(0.125)	**0.328**±(0.095)	0.272±(0.178)	0.311±(0.116)	0.298±(0.132)	0.307±(0.140)	0.262±(0.139)	0.259±(0.139)	0.284±(0.160)	0.282±(0.098)
Kiwix	0.260±(0.066)	0.274±(0.057)	0.272±(0.092)	0.263±(0.065)	0.262±(0.061)	0.300±(0.078)	0.253±(0.079)	0.242±(0.089)	0.265±(0.089)	**0.311**±(0.021)
Cloud	0.376±(0.077)	0.367±(0.083)	0.331±(0.121)	0.325±(0.080)	0.324±(0.078)	0.350±(0.088)	0.303±(0.132)	0.301±(0.135)	0.301±(0.142)	**0.385**±(0.018)
Turner	0.067±(0.133)	0.073±(0.137)	0.144±(0.182)	0.070±(0.151)	0.071±(0.153)	0.199±(0.166)	0.167±(0.150)	0.149±(0.168)	0.161±(0.164)	**0.303**±(0.143)
Reddit	0.421±(0.096)	0.419±(0.095)	**0.447**±(0.174)	0.440±(0.105)	0.442±(0.122)	0.408±(0.183)	0.410±(0.159)	0.395±(0.178)	0.352±(0.250)	0.438±(0.053)
Average	0.251±(0.121)	0.249±(0.122)	0.237±(0.093)	0.245±(0.111)	0.244±(0.111)	0.255±(0.092)	0.222±(0.087)	0.213±(0.083)	0.225±(0.079)	**0.307**±(0.081)

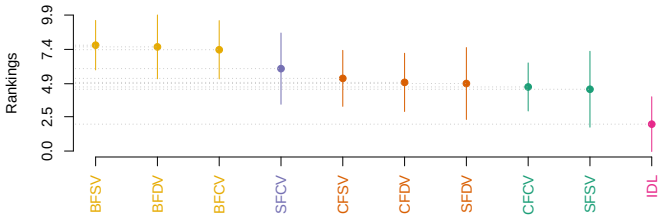

Figure 5: SKESD test for our IDL method and cost-sensitive based methods.

(CF) [25], and **B**alanced cost-sensitive decision **F**orest (BF) [25]. To construct the trees, three voting based strategies are applied to these methods, including cascading-and-**S**haring based **V**oting (SV) [14], maximally **D**iversified multiple decision tree based **V**oting (DV) [8], and **C**ost-sensitive **V**oting (CV) [25]. After combining each cost-sensitive based method and each voting-based strategy, we totally have nine comparative methods, short for SFSV, SFDV, SFCV, CFSV, CFDV, CFCV, BFSV, BFDV, and BFCV, respectively.

Results: Table 5 reports the average MCC value and the corresponding standard deviation of our IDL method and the nine cost-sensitive based imbalanced learning methods. Fig. 5 visualizes the statistic test result of SKESD. From the table and the figure, we can draw the following observations.

First, our IDL method obtains better performance on 8 out of 12 apps compared with the nine baseline methods. The average MCC value by our IDL method over all apps achieves improvements by 22.3%, 23.3%, 29.5%, 25.3%, 25.8%, 20.4%, 38.3%, 44.1%, and 36.4% compared with SFSV, SFDV, SFCV, CFSV, CFDV, CFCV, BFSV, BFDV, and BFCV, individually. Our IDL method obtains the best average indicator value and achieves an average improvement by 29.5%.

Second, our IDL method ranks the first and has significant differences compared with the nine cost-sensitive based imbalanced learning methods in terms of MCC indicator.

Different from the above methods which introduce the cost-sensitive strategy into the construction of trees without performing feature transformation, our IDL method integrates the weight strategy into feature representation learning. In sum, our IDL method significantly outperforms the comparative cost-sensitive based methods for predicting JIT defects on Android mobile apps.

6 THREATS TO VALIDITY

6.1 Threats to External Validity

The generalization of the experimental results threatens the external validity of this work. We conduct experiments on a publicly available benchmark dataset consisting of 12 Android mobile apps developed in Java language. We need to further explore whether our method is suitable for the mobile apps developed in other languages, such as Kotlin. In addition, since we only investigate the Android based mobile apps, it is necessary to investigate the IOS based mobile apps to verify the generalization of our IDL method.

6.2 Threats to Internal Validity

The implementation mistakes of the methods in our experiments threaten the internal validity of our work. In this work, we carefully implement the cost-sensitive cross-entropy loss function and the DNN structure based on TensorFlow and Python. As we specify multiple parameters according to the previous studies, the selection of the more optimal parameter settings needs to be explored in the future. As the code of cost-sensitive based baseline methods were released by authors, we carefully integrate it into our experiments. In addition, for other comparative methods, we implement them based on third-part libraries with the default parameter settings.

6.3 Threats to Construct Validity

The rationality of the used performance evaluation indicators and statistical test methods threatens the construct validity of our work. In this work, we only employ MCC, an indicator that is appropriate for imbalanced data, to evaluate the performance of our IDL method for JIT defect prediction on mobile apps, and will explore the use of effort-aware indicators in the future. In addition, to make our results more convincing, we apply a state-of-the-art statistic test method, i.e., SKESD, for the significant difference analysis.

7 CONCLUSION

In this work, we propose a novel JIT defect prediction method, called IDL, for Android mobile apps, which improves a deep learning model. More specifically, we introduce a novel cost-sensitive cross-entropy loss function into DNN to alleviate the issue of class imbalance in the process of feature representation learning, which takes

the prior probability of classes into account to compensate the imbalance between defective and clean commit instances when calculating the total loss. To evaluate the effectiveness of our IDL method, we conduct experiments on 12 Android mobile apps and employ the MCC indicator for performance evaluation. The experimental results demonstrate that our IDL method significantly outperforms 23 imbalanced learning methods, including eight sampling-based, six ensemble-based, and nine cost-sensitive based methods.

In the future, we plan to adapt our method to cross-project scenarios for JIT defect prediction on mobile apps.

ACKNOWLEDGMENTS

This work is supported by the National Key Research and Development Project (No.2018YFB2101200), the National Natural Science Foundation of China (Nos.62002034, 61902306), the Fundamental Research Funds for the Central Universities (2020CDCGRJ072), China Postdoctoral Science Foundation (No.2020M673137, 2019TQ0251, 2020M673439), the Natural Science Foundation of Chongqing in China (No.cstc2020jcyj-bshX0114).

REFERENCES

[1] Yuri Sousa Aurelio, Gustavo Matheus de Almeida, Cristiano Leite de Castro, and Antonio Padua Braga. 2019. Learning from imbalanced data sets with weighted cross-entropy function. *Neural Processing Letters* 50, 2 (2019), 1937–1949.

[2] Cagatay Catal and Banu Diri. 2009. Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem. *Information Sciences* 179, 8 (2009), 1040–1058.

[3] Gemma Catolino. 2017. Just-in-time bug prediction in mobile applications: the domain matters!. In *Proceedings of the 4th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 201–202.

[4] Gemma Catolino, Dario Di Nucci, and Filomena Ferrucci. 2019. Cross-project just-in-time bug prediction for mobile apps: an empirical assessment. In *Proceedings of the 6th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 99–110.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. The MIT Press.

[6] Lan Guo, Yan Ma, Bojan Cukic, and Harshinder Singh. 2004. Robust prediction of fault-proneness by random forests. In *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 417–428.

[7] Guo Haixiang, Li Yijing, Jennifer Shang, Gu Mingyun, Huang Yuanyue, and Gong Bing. 2017. Learning from class-imbalanced data: Review of methods and applications. *Expert Systems with Applications* 73 (2017), 220–239.

[8] Hong Hu, Jiuyong Li, Hua Wang, Grant Daggard, and Mingren Shi. 2006. A maximally diversified multiple decision tree algorithm for microarray data classification. In *Proceedings of the 2006 Workshop on Intelligent Systems for Bioinformatics*, Vol. 73. 35–38.

[9] Md Zahidul Islam and Helen Giggins. 2011. Knowledge Discovery through SysFor - a Systematically Developed Forest of Multiple Decision Trees. In *Proceedings of the 9th Australasian Data Mining Conference, AusDM*. 195–204.

[10] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2012. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering (TSE)* 39, 6 (2012), 757–773.

[11] Arvinder Kaur, Kamaldeep Kaur, and Harguneet Kaur. 2016. Application of machine learning on process metrics for defect prediction in mobile application. In *Information Systems Design and Intelligent Applications*. Springer, 81–98.

[12] Foutse Khomh, Brian Chan, Ying Zou, Anand Sinha, and Dave Dietz. 2011. Predicting post-release defects using pre-release field testing results. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 253–262.

[13] Jian Li, Pinjia He, Jieming Zhu, and Michael R Lyu. 2017. Software defect prediction via convolutional neural network. In *Proceedings of the 17th IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 318–328.

[14] Jinyan Li and Huiqing Liu. 2003. Ensembles of Cascading Trees. In *Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM)*. 585–588.

[15] Kalai Magal R and Shomona Gracia Jacob. 2015. Improved Random Forest Algorithm for Software Defect Prediction through Data Mining Techniques. *International Journal of Computer Applications* 117, 23 (2015), 18–22.

[16] Ruchika Malhotra. 2016. An empirical framework for defect prediction using machine learning techniques with Android software. *Applied Soft Computing* 49 (2016), 1034–1050.

[17] C Manjula and Lilly Florence. 2019. Deep neural network based hybrid approach for software defect prediction using software metrics. *Cluster Computing* 22, 4 (2019), 9847–9863.

[18] Stuart McIlroy, Nasir Ali, and Ahmed E Hassan. 2016. Fresh apps: an empirical study of frequently-updated mobile apps in the Google play store. *Empirical Software Engineering (EMSE)* 21, 3 (2016), 1346–1370.

[19] Tim Menzies, Jeremy Greenwald, and Art Frank. 2006. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering (TSE)* 33, 1 (2006), 2–13.

[20] Marco Ortu, Giuseppe Destefanis, Stephen Swift, and Michele Marchesi. 2016. Measuring high and low priority defects on traditional and mobile open source software. In *Proceedings of the 7th International Workshop on Emerging Trends in Software Metrics*. 1–7.

[21] Anh Viet Phan, Minh Le Nguyen, and Lam Thu Bui. 2017. Convolutional neural networks over control flow graphs for software defect prediction. In *Proceedings of the 29th IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 45–52.

[22] Michael Yoseph Ricky, Fredy Purnomo, and Budi Yulianto. 2016. Mobile application software defect prediction. In *2016 IEEE Symposium on Service-Oriented System Engineering*. IEEE, 307–313.

[23] Riccardo Scandariato and James Walden. 2012. Predicting vulnerable classes in an android application. In *Proceedings of the 4th International Workshop on Security Measurements and Metrics*. 11–16.

[24] Giuseppe Scanniello, Carmine Gravino, Andrian Marcus, and Tim Menzies. 2013. Class level fault prediction using software clustering. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 640–645.

[25] Michael J. Siers and Md Zahidul Islam. 2015. Software defect prediction using a cost sensitive decision forest and voting, and a potential solution to the class imbalance problem. *Information Systems* 51 (2015), 62–71.

[26] Qinbao Song, Yuchen Guo, and Martin J. Shepperd. 2019. A Comprehensive Investigation of the Role of Imbalanced Learning for Software Defect Prediction. *IEEE Transactions on Software Engineering (TSE)* 45, 12 (2019), 1253–1269.

[27] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering (TSE)* 43, 1 (2016), 1–18.

[28] Xin Xia, David Lo, Shane McIntosh, Emad Shihab, and Ahmed E Hassan. 2015. Cross-project build co-change prediction. In *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 311–320.

[29] Zhou Xu, Li Li, Meng Yan, Jin Liu, Xiapu Luo, John Grundy, Yifeng Zhang, and Xiaohong Zhang. 2020. A comprehensive comparative study of clustering-based unsupervised defect prediction models. *Journal of Systems and Software (JSS)* (2020), 110862.

[30] Zhou Xu, Shuai Li, Yutian Tang, Xiapu Luo, Tao Zhang, Jin Liu, and Jun Xu. 2018. Cross version defect prediction with representative data via sparse subset selection. In *Proceedings of the 26th International Conference on Program Comprehension (ICPC)*. 132–143.

[31] Zhou Xu, Shuai Li, Jun Xu, Jin Liu, Xiapu Luo, Yifeng Zhang, Tao Zhang, Jacky Keung, and Yutian Tang. 2019. LDFR: Learning deep feature representation for software defect prediction. *Journal of Systems and Software (JSS)* 158 (2019), 110402.

[32] Zhou Xu, Jin Liu, Xiapu Luo, Zijiang Yang, Yifeng Zhang, Peipei Yuan, Yutian Tang, and Tao Zhang. 2019. Software defect prediction based on kernel PCA and weighted extreme learning machine. *Information and Software Technology (IST)* 106 (2019), 182–200.

[33] Meng Yan, Xin Xia, Yuanrui Fan, Ahmed E Hassan, David Lo, and Shanping Li. 2020. Just-In-Time Defect Identification and Localization: A Two-Phase Framework. (2020).

[34] Meng Yan, Xin Xia, Yuanrui Fan, David Lo, Ahmed E Hassan, and Xindong Zhang. 2020. Effort-aware just-in-time defect identification in practice: a case study at Alibaba. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*. 1308–1319.

[35] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. 2015. Deep learning for just-in-time defect prediction. In *Proceedings of the 15th IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 17–26.

[36] Jingxiu Yao and Martin Shepperd. 2020. Assessing software defection prediction performance: why using the Matthews correlation coefficient matters. In *Proceedings of the 24th Evaluation and Assessment in Software Engineering (EASE)*. 120–129.

[37] Tao Zhang, He Jiang, Xiapu Luo, and Alvin TS Chan. 2016. A literature review of research in bug resolution: Tasks, challenges and future directions. *Comput. J.* 59, 5 (2016), 741–773.