Received: 27 February 2021 | Revised: 23 May 2021 | Accepted: 14 June 2021

IET Software

DOI: 10.1049/sfw2.12040

The Institution of Engineering and Technology WILEY

ORIGINAL RESEARCH PAPER

A compositional model for effort-aware Just-In-Time defect prediction on android apps

Kunsong Zhao[1,2] | **Zhou Xu**[1,3] | **Meng Yan**[1,3] | **Lei Xue**[4] | **Wei Li**[5] | **Gemma Catolino**[6]

[1]Key Laboratory of Dependable Service Computing in Cyber Physical Society (Chongqing University), Ministry of Education, Chongqing, China

[2]School of Computer Science, Wuhan University, Wuhan, China

[3]School of Big Data and Software Engineering, Chongqing University, Chongqing, China

[4]Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China

[5]School of Artificial Intelligence and Computer Science, Jiangnan University, Wuxi, China

[6]Jheronimus Academy of Data Science, Tilburg University, Tilburg, The Netherlands

Correspondence

Zhou Xu and Meng Yan, Key Laboratory of Dependable Service Computing in Cyber Physical Society (Chongqing University), Ministry of Education, Chongqing, China.
Email: zhouxullx@cqu.edu.cn (Z.X.) and mengy@cqu.edu.cn (M.Y.)

Funding information

National Key Research and Development Project, Grant/Award Number: 2018YFB2101200; China Postdoctoral Science Foundation, Grant/Award Number: 2020M673137; Special Funds for the Central Government to Guide Local Scientific and Technological Development, Grant/Award Number: YDZX20195000004725; Chongqing Technology Innovation and Application Development Project, Grant/Award Number: cstc2019jscx-mbdxX0064; Natural Science Foundation of Chongqing in China, Grant/Award Number: cstc2020jcyj-bshX0114; National Natural Science Foundation of China, Grant/Award Numbers: 62002034, 62002306; HKPolyU Start-up Fund, Grant/Award Number: ZVU7; European Commission grant, Grant/Award Number: 825040

Abstract

Android apps have played important roles in daily life and work. To meet the new requirements from users, the apps encounter frequent updates, which involves a large quantity of code commits. Previous studies proposed to apply Just-in-Time (JIT) defect prediction for apps to timely identify whether the new code commits can introduce defects into apps, aiming to assure their quality. In general, high-quality features are benefits for improving the classification performance. In addition, the number of defective commit instances is much fewer than that of clean ones, that is the defect data is class imbalanced. In this study, a novel compositional model, called KPIDL, is proposed to conduct the JIT defect prediction task for Android apps. More specifically, KPIDL first exploits a feature learning technique to preprocess original data for obtaining better feature representation, and then introduces a state-of-the-art cost-sensitive cross-entropy loss function into the deep neural network to alleviate the class imbalance issue by considering the prior probability of the two types of classes. The experiments were conducted on a benchmark defect data consisting of 15 Android apps. The experimental results show that the proposed KPIDL model performs significantly better than 25 comparative methods in terms of two effort-aware performance indicators in most cases.

KEYWORDS

fault tolerance, software performance evaluation, software quality

1 | INTRODUCTION

Software has already become an indispensable part of people's daily life and work. However, the existing software products unavoidably contain defects due to the increasing software scale and complexity [1]. The defects can result in a series of negative effects, which are unexpected. Since developing software products without defects is impossible, it is vital to propose suitable techniques to detect defects at the earliest. Software defect prediction emerges to employ different

techniques, such as machine learning methods, to predict defective code regions, which attract many researchers to contribute to it for software quality assurance [2, 3].

Recently, apps, especially Android apps, have become fashionable. It has the advantage that once the developers release the update of apps, the users can download them immediately from App Stores [4]. Apps are usually frequently updated to meet the functional requirements of users. This characteristic is inevitable to introduce defects into the new versions of apps, which hinder the app quality. Thus, the early detection of defects is an urgent issue during the process of development and maintenance for apps.

Many previous studies [5, 6] for defect prediction were based on the class level, which was a lack of immediate feedback for defective-prone codes. To overcome this drawback, researchers proposed Just-In-Time (JIT) defect prediction, which is at the commit level [7, 8]. JIT defect prediction aims at identifying whether a new code commit introduces defects, which can offer timely feedback for developers to detect the defects at the earliest. Considering this advantage, it is appropriate to apply JIT defect prediction to software with the characteristic of frequent updates (such as apps), which involves 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. [9, 10] took the first attempt to build JIT defect prediction models for Android 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 the defect prediction model. One alternative solution for this purpose is to make full use of the feature engineering techniques, such as the typical Principal Component Analysis (PCA) technique. PCA learns the linear combinations of the original features in a new space [11, 12]. However, PCA holds that the data for learning should be linearly separable and with the Gaussian distribution, which is impractical for the real-world data that consist of complicated structures and is difficult to simplify it into a linear subspace [13]. To overcome this shortage, we employ the non-linear enhanced version of PCA, that is, Kernel-based PCA (KPCA) technique [14]. It introduces the non-linear kernel function to map the raw features of the commit instances into a latent high-dimensional space, in which the ability of handling complicated structures is strengthened and the features can be operated linearly. In addition, we can select a part of the mapped features in the new space to mitigate the negative impact of noise data. Previous studies have already indicated that KPCA is superior to PCA method for software engineering tasks [15–17].

Moreover, for the defect data, the number of defective commit instances is much fewer than those of clean ones. In other words, the defect data is class imbalanced. Since the data imbalance characteristic usually deteriorates the performance of the classification model [17], it is crucial to deal with this issue for performance improvement. In this study, we propose a novel Imbalanced Deep Learning method, short for IDL, to address the class imbalanced issue. More specifically, as the deep learning method usually shows excellent performance for classification task, we use the Deep Neural Network (DNN) model to construct an effective classifier on the mapped defect data of Android apps to identify the defective commit instances. However, since the 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 cost-sensitive cross-entropy (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 [18]. In other words, it uses the weighting technique to compensate the class imbalance between the defective and clean commit instances.

In this study, we propose a novel JIT defect prediction model, called KPIDL, which is a compositional framework that incorporates the abovementioned two methods, that is, *KPCA* for feature representation learning and *IDL* for classification model construction. More specifically, KPIDL first exploits the KPCA technique to transform the original data into a new space to learn more representative features towards commit instances. Then, with the transformed features of commit instances, KPIDL employs our proposed IDL method to construct a classification model for handling the inherent class imbalanced problem of the defect data.

As the confusion matrix-based performance indicators used in previous defect prediction studies, such as Precision, Recall, and F-measure, assume that the test efforts of developers are always enough, it is unrealistic in the real-world app quality assurance activities. The reason is that apps have quick update iteration with a short development cycle, this causes the availability of only limited testing resources for code inspection. Thus, the classical performance indicators seem to be out of its depth. Arisholm et al. [19] proposed to take code inspection efforts into account for performance evaluation, termed effort-aware indicators, for defect prediction on traditional software projects. It is more appropriate for practical activities. Due to the above advantages, in this study, we fasten the effort-aware performance evaluation of our proposed JIT defect prediction method for Android apps.

To evaluate the prediction performance of our proposed JIT defect prediction framework KPIDL, in this study, we conduct experiments on a benchmark dataset that includes 15 Android apps and employ Effort-Aware Recall (EARecall) and Effort-Aware F-measure (EAF-measure) as performance indicators. Across the 15 apps, our KPIDL method achieves the average EARecall and EAF-measure values of 0.627 and 0.475, respectively. KPIDL obtains average improvements by 32.1%, 8.7%, 18.3%, 17.3% in terms of EARecall while by 27.6%, 17.1%, 26.9%, and 10.6% in terms of EAF-measure, compared with its five variants, six sampling-based methods, five ensemble-based methods, and nine cost-sensitive based methods, individually. The statistic test results show that our KPIDL method significantly outperforms 24 out of 25 comparative methods in terms of two effort-aware indicators.

This study extends our previous study published as a conference paper [20]. The four main differences between the two studies are highlighted as follows: (1) We take feature learning into account to reduce the impact of noise data and obtain high-quality feature representation to improve the performance of our JIT defect prediction model, which is ignored in the conference version; (2) We replace the original performance measurement with the effort-aware indicators to evaluate our proposed method, which is better to simulate the practical activities in which the code inspection efforts are always limited; (3) We add 5 methods for comparison to verify the effectiveness of our JIT defect prediction framework for Android apps; (4) We extend data by adding 4 real-world apps to enhance the generalisation of our model and make more in-depth analysis towards our experimental results.

The main contributions of this study are summarised as follows:

(1) To the best of our knowledge, we are the first to consider both the feature learning and class imbalance issue simultaneously for JIT defect prediction on Android apps.
(2) We develop a novel compositional model KPIDL for JIT defect prediction towards Android apps. Our proposed model introduces the KPCA method to learn more representative feature representation and then a novel deep learning-based model IDL is developed to relieve the negative impact of class imbalance of the defect data by employing a weighted cross-entropy-based loss function.
(3) We evaluate our KPIDL model on the defect data consisting of 15 Android apps and conduct the statistic test to analyse the experimental results. The results show that our method significantly outperforms 25 comparative methods in most cases.

The remainder of this study is organised as follows. The next section introduces the related work. Section 3 describes our KPIDL model in detail. We introduce the experimental setup and illustrate the performance evaluation in Section 4 and 5, respectively. The threats to the validity of our study are discussed in Section 6. Finally, we conclude this study and present future work in Section 7.

2 | RELATED WORK

2.1 | Feature learning in defect prediction

The process of feature representation learning usually applies some feature engineering methods to preprocess the raw feature set, making the new feature set better reveal the characteristics of the defect data. The widely used feature engineering methods include the feature selection methods and feature extraction methods [21]. The former ones select a part of features to replace the original feature set. The classic methods include filter-based feature ranking methods and wrapper-based feature subset selection methods. The latter ones transform the raw features into a new space in which the new feature form can well represent the defect data. The classic methods include the PCA and its kernel version KPCA.

Liu et al. [22] proposed a clustering and ranking-based feature selection method FECAR that first applied the symmetric uncertainty measure to cluster the original features into multiple groups and then used three relevance measures to obtain more relevant features from each group. The experimental results on Eclipse and NASA datasets demonstrated that FECAR was more effective in choosing the relevant features. Xu et al. [21] empirically assessed the impact of 32 feature selection techniques on defect prediction performance. Their experiments on the three datasets showed that the principal component analysis technique performed the worst among all the methods. Shivaji et al. [23] assessed the impact of filter-based and wrapper-based feature preprocess techniques on the performance of defect prediction. Their experiments on 11 projects showed that only using 10% of the original feature could still improve the defect prediction. Chen et al. [24] proposed a method MOFES that regarded the feature selection as a multi-objective optimisation issue, aiming at minimising the feature number and maximising the prediction performance simultaneously. They conducted experiments on PROMISE dataset and the results indicated that MOFES selected less features but obtained better prediction performance. Ghotra et al. [25] empirically investigated the impact of 30 feature selection methods applied to 21 classification models on the performance of defect prediction. They conducted experiments on NASA and PROMISE datasets and found that the correlation-based feature subset selection method with the best-first search strategy was more suitable for the defect prediction task compared with other baseline methods. Ni et al. [26] developed a novel cluster-based method FeSCH that used a density-based clustering technique and then proposed three different heuristic ranking strategies to select the useful features for cross-project defect prediction. Their experimental results on ReLink and AEEEM datasets showed the effectiveness of FeSCH.

Different from the above studies that took feature preprocess into account for defect prediction in traditional software projects, we simultaneously consider both feature learning and class imbalanced learning for JIT defect prediction on Android apps.

2.2 | Class imbalanced learning in defect prediction

Since most of defect data have an inherent class imbalance property, that is the defective instances are usually far fewer than those of the non-defective ones. The purpose of class imbalanced learning methods is to address the adverse impacts of this issue on the classification models. The widely used class imbalanced learning methods include sampling-based methods, ensemble-based methods, and cost-sensitive-based methods [27]. The former ones make the instance number of two classes the same by increasing or removing some instances. The middle ones combine several weak classifiers to get a better

and more comprehensive classifier with strong ability. The latter ones introduce the concept of misclassification cost to minimise the misclassification errors.

Liu et al. [28] proposed to apply the cost information in both feature selection and classification phases and developed three novel cost-sensitive feature selection methods for software defect prediction. They conducted experiments on NASA dataset and the results showed that the proposed cost-sensitive feature selection algorithms obtained promising prediction performances compared with the traditional methods that only considered cost information in classification. Siers et al. [29] proposed two decision tree-based cost-sensitive methods to minimise the classification cost for software defect prediction. They conducted experiments on six projects and the results illustrated that their techniques performed better than six comparative methods. Bennin et al. [30] empirically explored the statistical and practical effects of six sampling methods on five classification models for the defect prediction task. They conducted experiments on 10 open source projects and the results found that the investigated sampling methods had significant and practical effects in terms of performance indicators Pd, Pf, and G-mean but had no impact on AUC. Bennin et al. [31] empirically assessed the impacts of the percentage of fault-prone modules on seven sampling methods applied to five classification models for defect prediction. They conducted experiments on 10 static metric projects and the results demonstrated that the performance of these classification models could be largely impacted by this parameter except for AUC. Tantithamthavorn et al. [32] empirically investigated the impact of four popularly used sampling-based class rebalancing techniques on the performance of 10 widely used indicators and explored the impact of these sampling methods on the interpretation of defect prediction models. Their large-scale experiments with 101 systems showed that sampling-based class rebalancing methods were not helpful to interpret the defect prediction models. Bennin et al. [33] proposed a novel synthetic oversampling approach MAHAKIL that treated two different sub-classes as parents and produced a new sample that inherited traits from both parents to enhance the diversity of data distribution. They conducted experiments on PROMISE dataset and the results indicated that MAHAKIL was superior to the baseline methods.

Different from the above studies that focussed on relieving the class imbalanced issue with machine learning-based approaches for traditional software defect prediction, we design a deep learning model incorprating a novel cost-sensitive loss function to deal with this issue for JIT defect prediction on Android apps.

2.3 | JIT defect prediction for traditional software

Kamei et al. [7] applied some factors derived from characteristics of the software changes to predict JIT defects. They conducted a large-scale empirical study on 11 projects from different domains and the experimental results showed that their method effectively detected the most risky changes. Fukushima et al. [34] employed a JIT cross-project model to alleviate the issues of demand for a large amount of training data during the development stages. Their results on 11 open source projects showed that ensemble learning methods using historical data achieved better performance. Kamei et al. [35] explored a cross-project model for JIT defect prediction on 11 projects and the results showed that the cross-project model provided an alternative way for projects with limited historical data. McIntosh et al. [36] conducted experiments to address whether the important properties of fix-inducing changes were consistent with system evolution on two software systems with 37,524 changes. Their results showed that fluctuations derived from the system evolution impacted on the consistency. Yang et al. [37] proposed a method, called TLEL, which combined ensemble techniques with random forests to predict JIT defects. They conducted experiments on six open source projects with two indicators and the results showed that TLEL could discover over 70% of defects by reviewing only 20% of the code lines. Pascarella et al. [38] explored to what extent the commits were defective and proposed a fine-grained model for JIT defect prediction. Their experiments on 10 open source projects showed that their method could obtain better prediction performance in terms of the AUC indicator. Cabral et al. [39] conducted the first study to take class imbalance into consideration for JIT defect prediction. The experimental results on 10 projects from GitHub repository showed that their method performed better than baseline methods in terms of the g-mean indicator. Kondo et al. [40] defined the context metrics to perform the JIT defect prediction task. They conducted experiments on six open source projects and the results showed that the composites of two extended context metrics performed significantly better than those of the other metrics in terms of MCC and AUC indicators.

Different from the above studies that only conducted experiments on traditional software, in this study, we focus on JIT defect prediction for Android apps.

2.4 | Defect prediction for android apps

Ortu et al. [41] analysed 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. [42] proposed three metrics to capture the patterns of failure occurrences for defect prediction. They conducted experiments on 18 versions of an enterprise app and the results showed that these metrics predicted defects with a shorter time. Scandariato et al. [43] employed a Support Vector Machine (SVM) model to identify and analyse vulnerable components of apps using source code metrics. They analyzsd a popular application in the Android Market and the results showed that their model achieved higher accuracy and precision. Kaur et al. [44] compared the defect prediction performance using static code-based metrics and process-based

metrics. They conducted experiments on an open source app with seven machine learning methods and the results showed that process metrics-based models achieved better performance for defect prediction on apps. Ricky et al. [45] proposed an SVM method to predict defects on apps. Their experimental results on five datasets showed that their SVM method achieved better performance than that of the decision trees. Malhotra et al. [46] proposed a framework for identifying defective classes using object-oriented metrics. They conducted experiments on seven widely used Android apps and the results showed that there existed performance differences among 18 classification models. Kaur et al. [47] explored process metrics for defect prediction on an open source app. Their experimental results showed that the models with process metrics achieved better performance than those with code metrics.

Since the timely feedback is a characteristic of JIT defect prediction, it is especially suitable for frequently updated apps. However, there are few studies for identifying JIT defects on apps. Catolino et al. [9, 10] took the first attempt to explore the JIT defect prediction for apps and then compared the impacts of multiple machine learning methods and ensemble learning techniques. They extracted the defect data of apps from the COMMIT GURU platform as a benchmark dataset and their experimental results showed that Naive Bayes performed significantly better than other classifiers.

Different from most previous studies that focussed 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 [9, 10] that explored the traditional machine learning classifiers for JIT defect prediction on apps, we propose a novel method to learn effective feature representation for this task.

2.5 | Deep learning in defect prediction

Yang et al. [8] proposed 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. [48] 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. [49] learned semantic features employing 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. [50] proposed a novel method that employed the genetic algorithm and the deep neural network for feature representation learning and classification. Their experimental results on PROMISE dataset showed better accuracy than comparative methods. Xu et al. [51] 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 the deep learning technique into JIT defect prediction on Android apps by considering both feature representation learning and class imbalance learning.

3 | METHODOLOGY

Figure 1 depicts the overview of our proposed KPIDL model, which mainly includes two stages. The first stage is the KPCA-based feature representation learning process and the second stage is the IDL-based classification model construction process. The details of the two stages are described as follows:

3.1 | Feature representation learning with KPCA

In the first stage, we transform the raw features of commit instances with kernel function-based mapping into a latent space to learn more representative features. Here, we present how to transform the original feature set into a new space

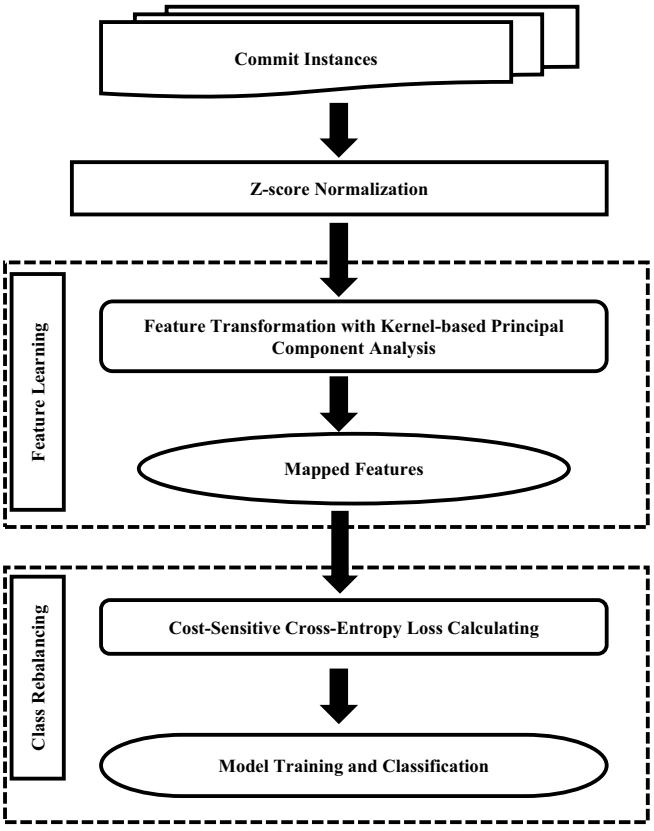

FIGURE 1 The overview of our KPIDL Framework

using the KPCA method to learn more representative features.

Assume that the commit instance set of 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 feature 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 feature learning is to obtain the high quality feature representation that has a more powerful ability to prompt the classification performance. For this purpose, in this study, we first use the Kernel-based Principal Component Analysis (KPCA) technique to acquire more representative features for each commit instance. KPCA employs the non-linear mapping function Φ to transform the original features into a new feature space F [2, 17, 52, 53]. Assume that the centralised projection point of x_i is defined as $\Phi(x_i)$, the covariance matrix C is formulised as follows:

$$C = \frac{1}{n}\sum_{i=1}^{n}\Phi(x_i)\Phi^T(x_i) \tag{1}$$

We execute the linear principal component analysis transformation by diagonalising the covariance matrix C, which can solve the eigenvalue problem in eigenspace F and is formulised as follows:

$$\lambda V = CV \tag{2}$$

where λ denotes the eigenvalues ($\lambda \geq 0$) and V denote the corresponding eigenvectors of covariance matrix C.

Since all eigenvectors V are within the scope of the centralised projection points $\Phi(x_1)$, $\Phi(x_2)$, …, and $\Phi(x_n)$, we multiply both sides of Equation (2) by $\Phi(x_l)^T$ ($l = 1, 2, \ldots, n$), which is formulised as follows:

$$\lambda\Phi(x_l)^T V = \Phi(x_l)^T CV \tag{3}$$

Then, the eigenvector V is formulised as follows:

$$V = \sum_{j=1}^{n}\alpha_j\Phi(x_j) \tag{4}$$

where the coefficient α_j can be treated as the linear expression of $\Phi(x_j)$.

Since it is unrealistic to appoint a specific form of Φ, we introduce the kernel function $\kappa(x_i, x_j)$, which is formulised as follows:

$$\kappa(x_i, x_j) = \Phi(x_i)^T\Phi(x_j) \tag{5}$$

By incorporating Equations (2), (4), and (5), we can obtain the following formula:

$$\mathbf{K}\alpha = n\lambda\alpha \tag{6}$$

where $\mathbf{K}$ is the kernel matrix (with size $n \times n$) corresponding to κ, $\mathbf{K}_{ij} = \kappa(x_i, x_j)$, and $\alpha = [\alpha_1, \alpha_2, \ldots, \alpha_n]^T$.

For a given commit instance x_{ins}, we extract the non-linear projection of k-th kernel component, which is formulised as follows:

$$V^k\Phi(x_{ins}) = \sum_{i=1}^{n}\alpha_i^k\Phi(x_i)\Phi(x_{ins}) = \sum_{i=1}^{n}\alpha_i^k\kappa(x_i, x_{ins}) \tag{7}$$

To perform the non-linear mapping, in this study, we apply the Gaussian Radial Basic Function (RBF) as the basic kernel function, which is formulised as follows:

$$\kappa(x_i, x_j) = exp\left(-\frac{\|x_i - x_j\|^2}{2\delta^2}\right) \tag{8}$$

where $\|\cdot\|$ and $2\delta^2$ signify the L_2 norm and the width of the RBF function, respectively.

After employing the KPCA technique for our defect data, we can obtain the transformed data set $S' = \{(X', Y)|X' \in \mathbb{R}^{n\times p}, Y \in \mathbb{R}^n\}$, where p is the dimension of the new feature space.

3.2 | Classification model construction with IDL

In the second stage, we propose a novel deep learning-based method to build the classification model, which has the ability to deal with the class imbalanced issue. As our KPIDL framework incorporates an improved cross-entropy loss function into a DNN method, we first introduce the original cross-entropy loss function, then detail its improved version, that is, the CSCE loss function, and last illustrate the DNN method.

3.2.1 | Cross-entropy loss function

The goal of 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 generalised 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:

$$\ell(\theta) = \frac{1}{n}\sum_{i=1}^{n}[-y_i\log(\hat{y}_i) - (1 - y_i)\log(1 - \hat{y}_i)] \tag{9}$$

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 [18].

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.

3.2.2 | Cost-sensitive cross-entropy loss function

From the above analysis, it is found that 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 assigning weights to the two kinds of losses, that is, $-y_i \log(\hat{y}_i)$ and $-(1-y_i)\log(1-\hat{y}_i)$.

Since the prior probability ratio, such as the ratio of the number of defective commit instances to the total number of commit instances, is helpful to achieve a balance between different classes, in this work, we introduce this term into the cross-entropy loss function, called CSCE loss function, to compensate the imbalance of the commit defect data, which is formulised as follows:

$$\ell(\theta) = \frac{1}{n}\sum_{i=1}^{n}[-\lambda y_i \log(\hat{y}_i) - (1-\lambda)(1-y_i)\log(1-\hat{y}_i)] \quad (10)$$

where $\lambda = M/N$ 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 [18] 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.

3.2.3 | Deep neural network

Deep Neural Network consists of three kinds of network layers, including the input layer, the hidden layer, and the output layer [54]. 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, weighted coefficient matrix, and bias vectors as inputs, and then outputs the results of the linear operation. In the second step, the back propagation algorithm is applied to optimise 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 apps, we input feature vectors of these mapped instances into the first layer of DNN. After the hidden layers and the output layer processing, the model calculates the total loss between the predicted labels and the true labels for commit instances using the CSCE loss function. Then, 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 Figure 2.

4 | EXPERIMENTAL SETUP

4.1 | Research Questions

To evaluate our proposed KPIDL method, in this study, we design the following four Research Questions (RQs).

RQ 1: *Is our KPIDL method superior to its variants?*

Our proposed KPIDL framework consists of two parts, that is KPCA method for feature learning and IDL method for relieving the class imbalance issue, in which KPCA is the improved version of original linear feature learning method PCA and IDL is the weighted advance of original **D**eep

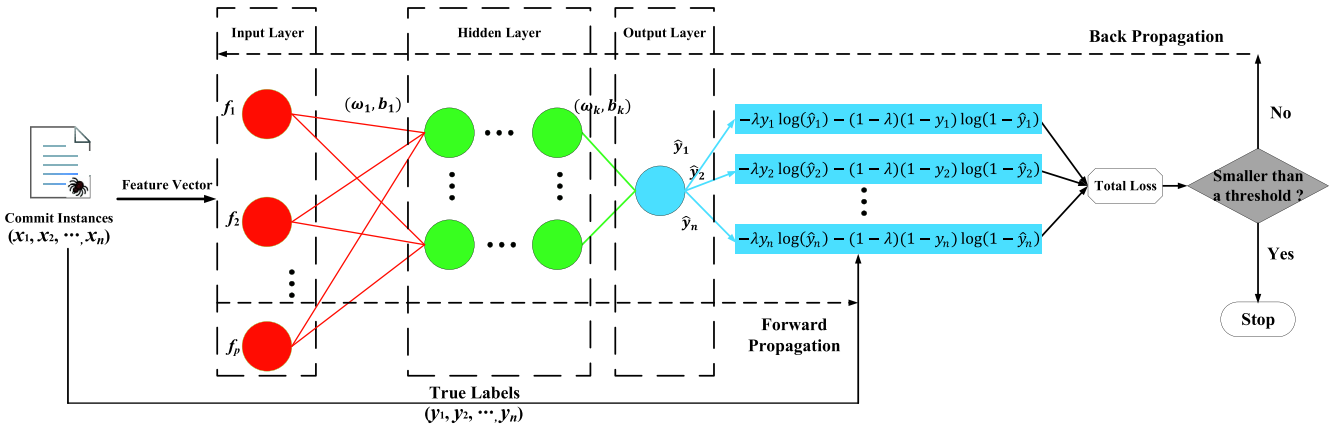

FIGURE 2 The iterative process of Deep Neural Network (DNN)

Learning (**DL**) model. This question is designed to explore whether our KPIDL method is more effective than these technical combinations to enhance JIT defect prediction performance on Android apps.

RQ 2: *Is our KPIDL 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 imbalanced learning method IDL is superior to the sampling-based methods for JIT defect prediction on Android apps.

RQ 3: *Does our KPIDL 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 the overall performance. This question is designed to explore whether our IDL method achieves better performance than that of the ensemble learning methods on imbalanced defect data of Android apps.

RQ 4: *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 minimise the high cost errors. This question is designed to explore whether our IDL method is more effective than the cost-sensitive-based imbalance learning methods to improve the JIT defect prediction performance on Android apps.

4.2 | Benchmark dataset

In order to evaluate the performance of our IDL method, in this study, we employ a benchmark dataset with 15 Android apps denoted by a recent study [10]. Here, we briefly describe these apps. Android *Firewall* is a powerful firewall app based on Linux iptables, which allows users to control which apps can access the networks. *Alfresco* is a business office app, which ensures the corporate documents are accessed securely. Android *Sync* is an Android synchronisation manager, which transmits data between the Android device and PC only using USB. Android *Wallpaper* provides a variety of high-quality wallpapers by using manual checking and sorting. AnySoft*Keyboard* provides the support of multiple languages and privacy protection for screen keyboard in Android mobile devices. *Apg* introduces email encryption into the Android devices for privacy protection. Chat *Secure* provides users with a secure communication app based on open standards, such as XMPP/Jabber and OTR encryption. *Kiwix* is a lightweight piece of an app, which allows users to read and download files (e.g., Wikipedia, Wiktionary, and TED talks) when the internet connection is unusable. Own *Cloud* Android provides a cloud storage platform to synchronise the personal privacy files. Page *Turner* provides an ebook reader, which can maintain the same reading process between multiple devices. Notify *Reddit* allows users to acquire the favourite notifications from their Android wearable. Android Universal *Image* Loader provides synchronous and asynchronous image loading. Observable *Scroll* View provides the listening for scrolling status and can interact with the Toolbar easily. *Applozic* Android SDK is an in-app solution that makes real time chat in apps to be more convenient. *Delta* Chat is an email-based instant messaging tool that relieves the tracking or central control. As we can see from the above descriptions, these apps come from various domains.

Table 1 summarises 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 9506 and 275,637, which means that these apps have different scales. The commit instances of the apps in the benchmark dataset are characterised by a feature set from different scopes, such as *Diffusion*, *Size*, *Purpose*, *History*, and *Experience*. We follow the original work [10] to use the widely used 14 features that have been proved to be the most useful ones to identify defective commit instances in the context of JIT defect prediction for Android apps. Table 2 presents the brief descriptions of the 14 features.

TABLE 1 The basic information of the 15 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	29.7%
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%
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	9506	222	60	162	27.0%
Image	16,530	875	141	734	16.1%
Scroll	27,836	250	54	196	21.6%
Applozic	87,662	946	143	803	15.1%
Delta	96,971	2465	185	2280	7.5%

TABLE 2 The brief descriptions of features

Name	Scope	Definition
NS	Diffusion	Number of modified subsystems involved in the current change
ND		Number of modified directories involved in the current change
NF		Number of modified files involved in the current change
Entropy		Distribution of modified code across files involved in the current change
LA	Size	Lines of code added by the current change
LD		Lines of code deleted by the current change
LT		Lines of code in a file before the current change
FIX	Purpose	Whether or not the current change is a bug fix
NDEV	History	Number of developers changing the files
AGE		Average time interval since the last change
NUC		Number of unique change to modified files
EXP	Experience	Developer experience
REXP		Recent developer experience
SEXP		Developer experience on a subsystem

4.3 | Performance indicators

Traditional confusion matrix-based indicators, such as Precision, Recall, and F-measure, assume that during the testing process, the efforts for reviewing the distinct code snippets are the same and the test sources for code inspection are always enough. Nevertheless, it is impractical to ignore the availability of sufficient test resources and inspecting different code snippets will expend inconsistent efforts. To overcome the aforementioned weaknesses, in this study, we evaluate the performance of our proposed KPIDL method by effort-aware indicators that take code reviews efforts into account for practical simulation. In this study, we regard the sum of features LA and LD as the substitute to code inspection efforts. In addition, the accessible test resources are deemed as 20% of all the efforts, following the previous studies [55, 56]. Below, we briefly depict how to calculate the effort-aware indicators.

Following the previous studies [2, 51], we first train a classification model using our IDL method with the transformed features of commit data by the KPCA technique to predict the commit instances from test set as two groups (i.e., defective and clean). Then, the commit instances from each group are ranked in the ascending order via their code inspection efforts, respectively. Next, the ranked commit instances are merged as candidates in which those predicted to be defective are put in the front. After that, we simulate the practitioners to inspect the candidate instances from high to low. This inspection activity continues until the accumulative effort accounts for 20% of all efforts and we can obtain the following statistical information corresponding to the inspected commit instances to calculate the effort-aware indicators.

- N_d refers to the total number of defective commit instances in candidate set.
- N_i refers to the total number of reviewed commit instances in candidate by inspecting 20% of efforts.
- N_{id} refers to the total number of reviewed defective commit instances by inspecting 20% of efforts.

Based on the above statistics, the first effort-aware indicator is Effort-Aware Recall (EARecall) that is defined as the percent of reviewed defective commit instances to the whole defective commit instances in candidate set. EARecall is denoted as follows:

$$\text{EARecall} = N_{id}/N_d \tag{11}$$

Effort-Aware Precision (EAPrecision) refers to the percent of reviewed defective commit instances to the whole instances in the candidate set, which is denoted as $\text{EAPrecision} = N_{id}/N_i$.

The second effort-aware indicator EAF-measure resembles the traditional F-measure, which is the weighted harmonic average considering EARecall and EAPrecision. EAF-measure is denoted as follows:

$$\text{EAF - measure} = \frac{(1+\theta^2) \times \text{EAPrecision} \times \text{EARecall}}{\theta^2 \times \text{EAPrecision} + \text{EARecall}} \tag{12}$$

where θ is a trade-off parameter and is set as 2 following the previous studies [2, 51, 57].

4.4 | Data partition

In this study, 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 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 KPIDL 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 of the random partition on our experimental results, we repeat this procedure 25 times. Thus, we obtain a total of $25 \times 2 = 50$ indicator values. In this study, we report the average value and the corresponding standard deviation for each performance indicator.

4.5 | Parameter settings

To obtain more representative features with the KPCA technique, in the feature learning stage, we set the kernel parameters of KPCA following the default settings in Scikit-learn library. Also, we specify the transformed data dimension p as 14. In the model construction stage, we set the structure of DNN as one input layer and two hidden layers with 32 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 2000. Moreover, we apply the RMSProp algorithm [58] to optimise our DNN model. In each iteration, we set the learning rate as 0.01 with the decay rate as 0.99. In addition, we employ the exponential moving average model [58] with the decay rate as 0.99 for the learning rate. When calculating the loss, the L2 regularisation 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 study, we apply a state-of-the-art method, namely Scott-Knott Effect Size Difference (SKESD) test [59], to analyse 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 the groups with the negligible effect size of significant differences. To overcome these two limitations, Tantithamthavorn et al. [59] proposed an improved version, called SKESD test that applied log transforming to preprocess the results of the performance indicator and quantified the effect size by applying Cohen's delta. In this study, we perform the SKESD test with two stages to conduct the significant analysis. The process of SKESD test is demonstrated in Figure 3. In the first stage, we take all the performance indicator values of each method on each 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 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 KPIDL method and its variants

Methods: To answer this question, we first treat the IDL combining PCA method (short for PIDL) and IDL without any feature preprocess (i.e., IDL) as baseline methods to investigate how effective is IDL when using non-linear KPCA, linear PCA, and no feature learning. In addition, we compare KPIDL with the baseline methods that combine the traditional DL method with KPCA, PCA, and no feature learning (short for KPDL, PDL, and DL, individually) to investigate the prediction performance when not considering the class imbalance.

Results: Tables 3 and 4 report the average EARecall and EAF-measure values and the corresponding standard deviations of our KPIDL method and the five comparative variants, individually. In these tables, the values in bold denote the best performance for each app or the best average value across all apps. Figure 4 visualises the statistic test results of SKESD for our KPIDL method and the five baseline methods in terms of two effort-aware indicators. Different colours indicate that the methods belong to different groups with significant differences. From these tables and the figure, the following findings can be drawn.

First, in terms of EARecall, our KPIDL method obtains better performance on 6 out of 15 apps compared with the five baseline methods. The average EARecall value by our KPIDL method over all apps achieves improvements by 38.7%, 27.4%, 21.5%, 9.0%, and 63.7% compared with those of DL, PDL, KPDL, IDL, and PIDL, individually. Our KPIDL method obtains the best average EARecall value and achieves an average improvement by 32.1%.

Second, in terms of EAF-measure, our KPIDL method obtains better performance on 9 out of 15 apps compared with the five baseline methods. The average EAF-measure value by our KPIDL method over all apps achieves improvements by 43.1%, 36.9%, 11.8%, 3.7%, and 42.6% compared with those of DL, PDL, KPDL, IDL, and PIDL, individually. Our KPIDL method obtains the best average EAF-measure value and achieves an average improvement by 27.6%.

Third, our KPIDL method ranks the first and has significant differences compared with its five variants in terms of all two effort-aware indicators.

Summary: Different from its variants that only take either the feature learning or the imbalanced learning into consideration, our KPIDL method that combines the KPCA technique and IDL model has the advantages to learn representative features and deal with the class imbalance problem simultaneously. Our KPIDL method is more effective in obtaining significantly better performance than that of its five variants for predicting JIT defects on Android apps.

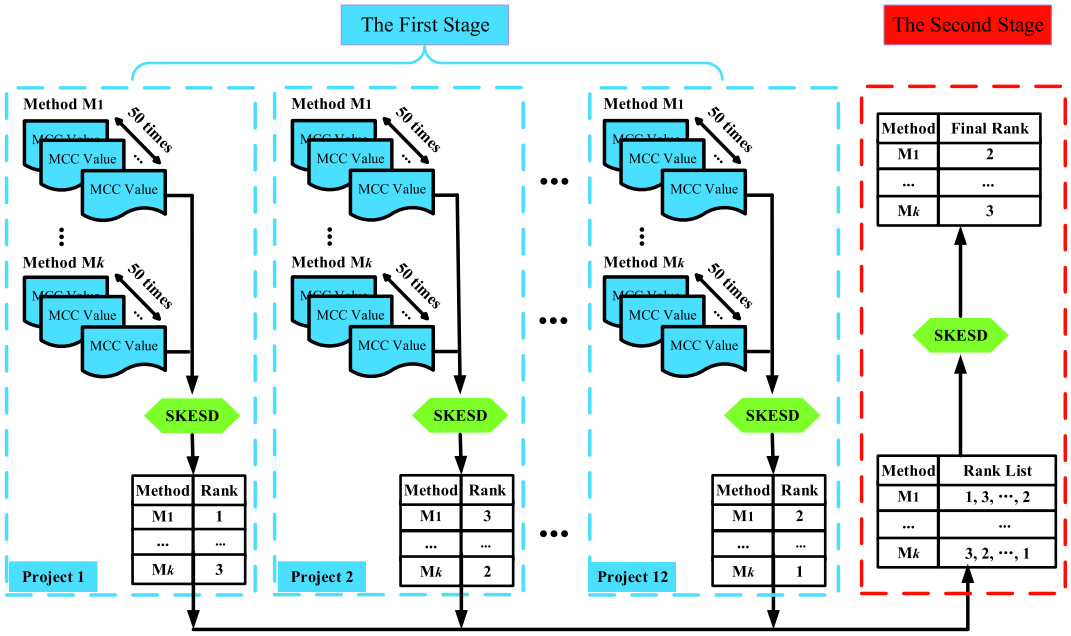

FIGURE 3 The operational process of Scott-Knott Effect Size Difference (SKESD) test

TABLE 3 The average Effort-Aware Recall of our KPIDL method and its variants

Project	DL	PDL	KPDL	IDL	PIDL	KPIDL
Firewall	0.489 ± 0.24	0.377 ± 0.07	0.446 ± 0.14	**0.658** ± 0.21	0.455 ± 0.13	0.530 ± 0.14
Alfresco	0.326 ± 0.20	0.300 ± 0.26	0.573 ± 0.30	0.662 ± 0.28	0.271 ± 0.10	**0.714** ± 0.24
Sync	0.375 ± 0.17	0.356 ± 0.12	0.399 ± 0.11	0.444 ± 0.17	0.288 ± 0.12	**0.466** ± 0.16
Wallpaper	**0.593** ± 0.19	0.578 ± 0.17	0.341 ± 0.15	0.525 ± 0.17	0.297 ± 0.15	0.532 ± 0.12
Keyboard	0.505 ± 0.34	0.681 ± 0.31	0.785 ± 0.30	0.580 ± 0.27	0.289 ± 0.11	**0.822** ± 0.28
Apg	0.428 ± 0.27	0.478 ± 0.28	0.526 ± 0.18	**0.604** ± 0.20	0.468 ± 0.24	0.545 ± 0.24
Secure	0.543 ± 0.25	0.532 ± 0.28	0.656 ± 0.30	0.796 ± 0.16	0.406 ± 0.14	**0.926** ± 0.04
Kiwis	0.587 ± 0.30	0.504 ± 0.32	0.781 ± 0.19	0.607 ± 0.23	0.289 ± 0.10	**0.860** ± 0.05
Cloud	0.260 ± 0.07	0.439 ± 0.22	0.410 ± 0.15	0.589 ± 0.24	0.293 ± 0.12	**0.636** ± 0.23
Turner	0.315 ± 0.23	**0.689** ± 0.12	0.531 ± 0.17	0.198 ± 0.21	0.498 ± 0.26	0.503 ± 0.37
Reddit	0.430 ± 0.18	0.369 ± 0.18	0.425 ± 0.12	**0.492** ± 0.19	0.326 ± 0.15	0.486 ± 0.17
Image	0.288 ± 0.16	0.379 ± 0.25	0.344 ± 0.12	**0.510** ± 0.21	0.240 ± 0.14	0.456 ± 0.15
Scroll	0.582 ± 0.34	**0.587** ± 0.29	0.568 ± 0.25	0.470 ± 0.24	0.233 ± 0.11	0.517 ± 0.24
Applozic	0.379 ± 0.25	0.372 ± 0.19	0.524 ± 0.31	**0.798** ± 0.11	0.467 ± 0.38	0.776 ± 0.11
Delta	0.684 ± 0.35	**0.738** ± 0.32	0.425 ± 0.21	0.686 ± 0.38	0.925 ± 0.23	0.630 ± 0.26
Average	0.452 ± 0.12	0.492 ± 0.13	0.516 ± 0.14	0.575 ± 0.14	0.383 ± 0.17	**0.627** ± 0.15

Note: The best performance values among these methods achieved on each app are shown in bold.

5.2 | Answer to RQ2: the prediction performance of our KPIDL method and the sampling-based imbalanced learning methods

Methods: To answer this question, we choose six sampling methods as the baseline methods, including Random Over-Sampling (ROS), Random Under-Sampling (RUS), The Synthetic Minority Over-sampling Technique (SMOT), SMOT with Tomek links (SMOTT), SMOT with Borderline samples (SMOTB), and over-sampling using ADAptive SYNthetic sampling (ADASYN). We also use random forest as the basic classifier, which is widely used in software defect prediction tasks [60–63].

Results: Tables 5 and 6 report the average EARecall and EAF-measure values and the corresponding standard deviations of our KPIDL method and six comparative sampling-

TABLE 4 The average Effort-Aware F-measure of our KPIDL method and its variants

Project	DL	PDL	KPDL	IDL	PIDL	KPIDL
Firewall	0.449 ± 0.16	0.386 ± 0.05	0.451 ± 0.12	**0.600** ± 0.11	0.461 ± 0.12	0.520 ± 0.09
Alfresco	0.290 ± 0.10	0.253 ± 0.12	0.434 ± 0.12	0.471 ± 0.10	0.278 ± 0.08	**0.507** ± 0.06
Sync	0.336 ± 0.12	0.326 ± 0.09	0.397 ± 0.09	0.424 ± 0.11	0.308 ± 0.13	**0.443** ± 0.10
Wallpaper	0.353 ± 0.08	0.340 ± 0.08	0.293 ± 0.13	0.357 ± 0.06	0.255 ± 0.08	**0.360** ± 0.04
Keyboard	0.388 ± 0.19	0.493 ± 0.16	0.545 ± 0.16	0.503 ± 0.13	0.312 ± 0.12	**0.580** ± 0.13
Apg	0.385 ± 0.16	0.420 ± 0.16	0.494 ± 0.12	**0.540** ± 0.13	0.438 ± 0.19	0.495 ± 0.14
Secure	0.460 ± 0.15	0.446 ± 0.17	0.521 ± 0.17	0.649 ± 0.05	0.409 ± 0.13	**0.674** ± 0.02
Kiwis	0.425 ± 0.16	0.379 ± 0.17	0.544 ± 0.08	0.492 ± 0.09	0.306 ± 0.10	**0.576** ± 0.02
Cloud	0.252 ± 0.04	0.353 ± 0.11	0.384 ± 0.12	0.477 ± 0.08	0.307 ± 0.13	**0.503** ± 0.05
Turner	0.201 ± 0.10	0.340 ± 0.06	**0.393** ± 0.09	0.174 ± 0.14	0.315 ± 0.08	0.252 ± 0.17
Reddit	0.349 ± 0.10	0.323 ± 0.10	0.423 ± 0.07	0.464 ± 0.11	0.350 ± 0.17	**0.461** ± 0.10
Image	0.198 ± 0.07	0.250 ± 0.12	0.309 ± 0.08	**0.394** ± 0.07	0.247 ± 0.14	0.377 ± 0.06
Scroll	0.400 ± 0.16	0.396 ± 0.16	**0.427** ± 0.11	0.399 ± 0.12	0.245 ± 0.11	0.418 ± 0.11
Applozic	0.256 ± 0.12	0.239 ± 0.10	0.525 ± 0.32	**0.675** ± 0.22	0.480 ± 0.39	0.649 ± 0.20
Delta	0.242 ± 0.06	0.256 ± 0.05	0.236 ± 0.08	0.248 ± 0.07	0.277 ± 0.04	**0.310** ± 0.07
Average	0.332 ± 0.09	0.347 ± 0.07	0.425 ± 0.09	0.458 ± 0.13	0.333 ± 0.08	**0.475** ± 0.12

Note: The best performance values among these methods achieved on each app are shown in bold.

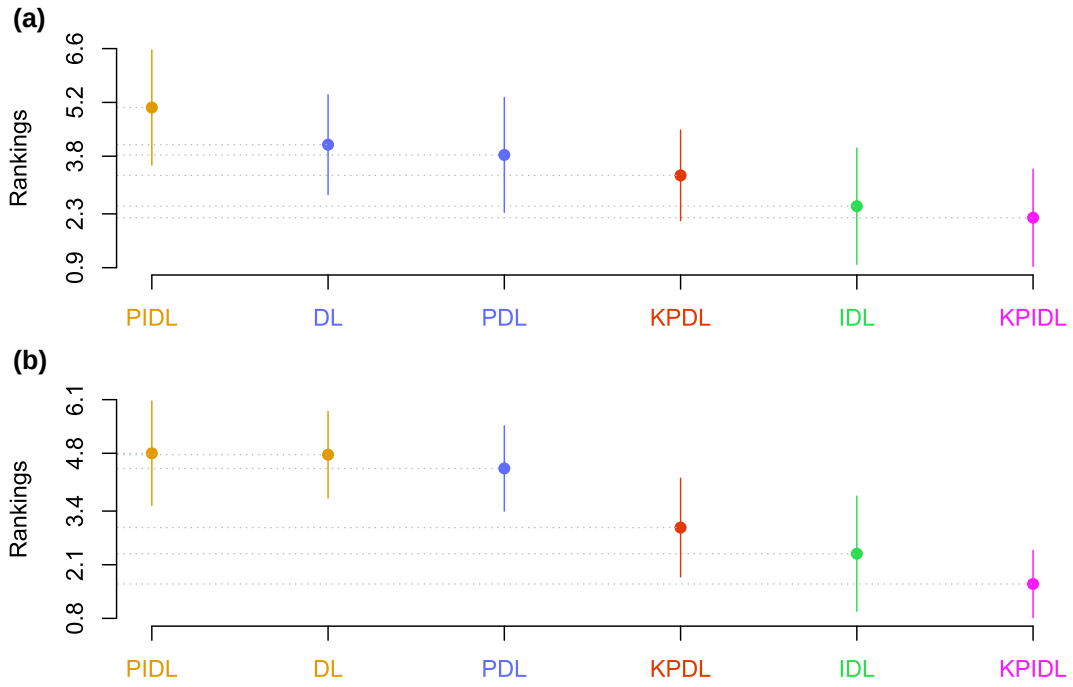

FIGURE 4 Scott-Knott Effect Size Difference test for KPIDL method and its variants. (a) EARecall, (b) EAF-measure

based imbalanced learning methods, individually. Figure 5 visualises the statistic test results of SKESD for our KPIDL method and the six baseline methods in terms of two effort-aware indicators. From these tables and the figure, we can draw the following observations.

First, in terms of EARecall, our KPIDL method obtains better performance on 9 out of 15 apps compared with the six baseline methods. The average EARecall value by our KPIDL method over all apps achieves improvements by 10.0%, 16.1%, 4.3%, 8.5%, 9.4%, and 4.2% compared with those of ROS, RUS, SMOT, SMOTT, SMOTB, and ADASYN, individually. Our KPIDL method obtains the best average EARecall value and achieves an average improvement by 8.7%.

TABLE 5 The Average Effort-Aware Recall of our KPIDL method and sampling-based methods

Project	ROS	RUS	SMOT	SMOTT	SMOTB	ADASYN	KPIDL
Firewall	0.756 ± 0.24	0.804 ± 0.17	0.791 ± 0.20	0.571 ± 0.24	0.762 ± 0.20	**0.939** ± 0.12	0.530 ± 0.14
Alfresco	0.690 ± 0.21	0.671 ± 0.21	0.632 ± 0.25	0.649 ± 0.23	0.668 ± 0.23	0.642 ± 0.15	**0.714** ± 0.24
Sync	0.417 ± 0.20	0.437 ± 0.15	0.429 ± 0.16	0.435 ± 0.14	0.41 ± 0.15	0.461 ± 0.16	**0.466** ± 0.16
Wallpaper	0.490 ± 0.15	0.479 ± 0.13	0.493 ± 0.11	0.485 ± 0.10	0.426 ± 0.15	0.468 ± 0.14	**0.532** ± 0.12
Keyboard	0.769 ± 0.27	0.613 ± 0.27	0.770 ± 0.27	0.817 ± 0.21	0.731 ± 0.17	0.686 ± 0.26	**0.822** ± 0.28
Apg	0.593 ± 0.23	0.571 ± 0.22	0.607 ± 0.32	**0.675** ± 0.30	0.585 ± 0.30	0.614 ± 0.29	0.545 ± 0.24
Secure	0.714 ± 0.12	0.614 ± 0.13	0.808 ± 0.16	0.635 ± 0.11	0.794 ± 0.16	0.650 ± 0.10	**0.926** ± 0.04
Kiwis	0.581 ± 0.24	0.429 ± 0.19	0.703 ± 0.21	0.677 ± 0.24	0.576 ± 0.24	0.737 ± 0.15	**0.860** ± 0.05
Cloud	0.535 ± 0.12	0.455 ± 0.15	0.529 ± 0.19	0.561 ± 0.21	0.524 ± 0.17	0.546 ± 0.21	**0.636** ± 0.23
Turner	**0.542** ± 0.28	0.337 ± 0.12	0.498 ± 0.32	0.498 ± 0.32	0.527 ± 0.28	0.500 ± 0.32	0.503 ± 0.37
Reddit	0.435 ± 0.22	0.467 ± 0.16	0.432 ± 0.17	0.436 ± 0.16	0.421 ± 0.14	0.430 ± 0.16	**0.486** ± 0.17
Image	0.432 ± 0.11	0.414 ± 0.10	**0.510** ± 0.09	0.453 ± 0.11	0.438 ± 0.06	0.506 ± 0.12	0.456 ± 0.15
Scroll	0.390 ± 0.19	0.461 ± 0.14	0.498 ± 0.13	0.456 ± 0.11	0.461 ± 0.13	**0.544** ± 0.16	0.517 ± 0.24
Applozic	0.369 ± 0.10	0.427 ± 0.10	0.391 ± 0.10	0.385 ± 0.14	0.381 ± 0.12	0.381 ± 0.10	**0.776** ± 0.11
Delta	0.830 ± 0.25	0.926 ± 0.17	0.927 ± 0.20	**0.930** ± 0.20	0.898 ± 0.24	0.925 ± 0.21	0.630 ± 0.26
Average	0.570 ± 0.15	0.540 ± 0.16	0.601 ± 0.16	0.578 ± 0.15	0.573 ± 0.16	0.602 ± 0.16	**0.627** ± 0.15

Note: The best performance values among these methods achieved on each app are shown in bold.

TABLE 6 The average Effort-Aware F-measure of our KPIDL method and sampling-based methods

Project	ROS	RUS	SMOT	SMOTT	SMOTB	ADASYN	KPIDL
Firewall	0.629 ± 0.15	0.663 ± 0.10	0.650 ± 0.12	0.514 ± 0.147	0.635 ± 0.13	**0.738** ± 0.09	0.520 ± 0.09
Alfresco	0.464 ± 0.08	0.461 ± 0.09	0.434 ± 0.10	0.438 ± 0.09	0.452 ± 0.10	0.459 ± 0.06	**0.507** ± 0.06
Sync	0.368 ± 0.14	0.381 ± 0.11	0.377 ± 0.11	0.387 ± 0.10	0.368 ± 0.10	0.403 ± 0.11	**0.443** ± 0.10
Wallpaper	0.322 ± 0.06	0.323 ± 0.08	0.325 ± 0.05	0.327 ± 0.04	0.296 ± 0.07	0.311 ± 0.06	**0.360** ± 0.04
Keyboard	0.549 ± 0.13	0.472 ± 0.14	0.551 ± 0.13	0.578 ± 0.09	0.550 ± 0.07	0.512 ± 0.14	**0.580** ± 0.13
Apg	0.503 ± 0.13	0.498 ± 0.12	0.495 ± 0.19	**0.542** ± 0.18	0.487 ± 0.19	0.508 ± 0.17	0.495 ± 0.14
Secure	0.594 ± 0.08	0.548 ± 0.10	0.636 ± 0.07	0.556 ± 0.08	0.634 ± 0.07	0.562 ± 0.06	**0.674** ± 0.02
Kiwis	0.433 ± 0.14	0.359 ± 0.11	0.492 ± 0.11	0.476 ± 0.13	0.431 ± 0.13	0.522 ± 0.07	**0.576** ± 0.02
Cloud	0.401 ± 0.06	0.346 ± 0.10	0.418 ± 0.08	0.426 ± 0.08	0.409 ± 0.08	0.434 ± 0.08	**0.503** ± 0.05
Turner	0.291 ± 0.10	0.207 ± 0.06	0.253 ± 0.14	0.254 ± 0.14	**0.282** ± 0.10	0.254 ± 0.14	0.252 ± 0.17
Reddit	0.359 ± 0.14	0.390 ± 0.11	0.368 ± 0.11	0.372 ± 0.11	0.367 ± 0.11	0.370 ± 0.11	**0.461** ± 0.10
Image	0.267 ± 0.04	0.284 ± 0.06	0.313 ± 0.06	0.283 ± 0.06	0.281 ± 0.04	0.307 ± 0.07	**0.377** ± 0.06
Scroll	0.298 ± 0.13	0.332 ± 0.11	0.365 ± 0.09	0.344 ± 0.08	0.343 ± 0.09	0.386 ± 0.10	**0.418** ± 0.11
Applozic	0.271 ± 0.07	0.302 ± 0.06	0.274 ± 0.06	0.270 ± 0.08	0.272 ± 0.08	0.273 ± 0.07	**0.649** ± 0.20
Delta	0.266 ± 0.05	0.282 ± 0.02	0.279 ± 0.03	0.279 ± 0.03	0.275 ± 0.04	0.278 ± 0.03	**0.310** ± 0.07
Average	0.401 ± 0.12	0.390 ± 0.12	0.415 ± 0.12	0.403 ± 0.11	0.405 ± 0.12	0.421 ± 0.13	**0.475** ± 0.12

Note: The best performance values among these methods achieved on each app are shown in bold.

Second, in terms of EAF-measure, our KPIDL method obtains better performance on 12 out of 15 apps compared with the six baseline methods. The average EAF-measure value by our KPIDL method over all apps achieves improvements by 18.5%, 21.8%, 14.5%, 17.9%, 17.3%, and 12.8% compared with those of ROS, RUS, SMOT, SMOTT, SMOTB, and ADASYN, individually. Our KPIDL method obtains the best average EAF-measure value and achieves an average improvement by 17.1%.

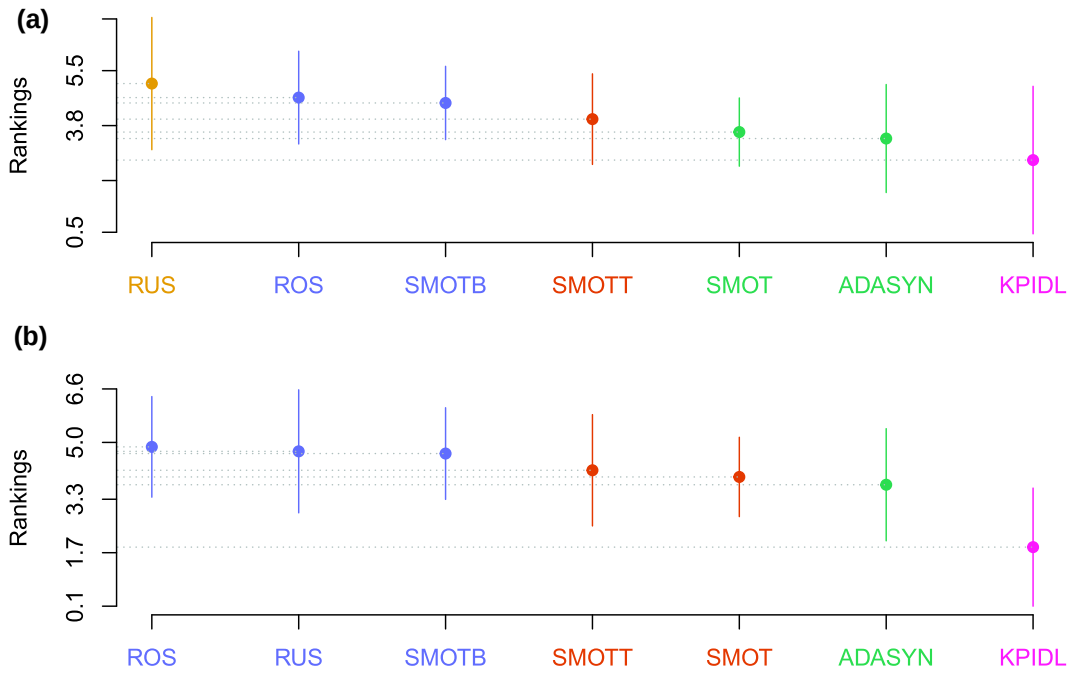

FIGURE 5 Scott-Knott Effect Size Difference test for our KPIDL method and sampling-based methods. (a) EARecall, (b) EAF-measure

Third, our KPIDL method ranks the first and has significant differences compared with the six sampling-based imbalanced learning methods in terms of all two effort-aware indicators.

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

5.3 | Answer to RQ3: the prediction performance of our KPIDL method and the ensemble-based imbalanced learning methods

Methods: To answer this question, we choose five ensemble methods for comparison, including Balanced Random Forest (BRF), EasyEnsemble (EasyEn), Bagging (Bag), Balanced Bagging (BBag), and Adaptive Boost (AdaB).

Results: Tables 7 and 8 report the average EARecall and EAF-measure values and the corresponding standard deviations of our KPIDL method and the five comparative ensemble-based imbalanced learning methods, individually. Figure 6 visualises the statistic test results of SKESD for our KPIDL method and the five baseline methods in terms of two effort-aware indicators. From these tables and the figure, the following findings can be drawn.

First, in terms of EARecall, our KPIDL method obtains better performance on six out of 15 apps compared with the five baseline methods. The average EARecall value by our KPIDL method over all apps achieves improvements by 21.0%, 10.6%, 2.6%, 31.2%, and 25.9% compared with those BRF, EasyEn, Bag, Bbag, and AdaB, individually. Our KPIDL method obtains the best average EARecall value and achieves an average improvement by 18.3%.

Second, in terms of EAF-measure, our KPIDL method obtains better performance on 10 out of 15 apps compared with the five baseline methods. The average EAF-measure value by our KPIDL method over all apps achieves improvements by 25.3%, 18.5%, 23.7%, 31.9%, and 34.9% compared with those of BRF, EasyEn, Bag, Bbag, and AdaB, individually. Our KPIDL method obtains the best average EAF-measure value and achieves an average improvement by 26.9%.

Third, our KPIDL method ranks the first and has significant differences compared with the five ensemble-based imbalanced learning methods in terms of all two effort-aware indicators except for the Bag method with EARecall indicator.

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

5.4 | Answer to RQ4: the prediction performance of our KPIDL 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 Forest of multiple decision trees (SF) [64], Cost-sensitive-

TABLE 7 The average Effort-Aware Recall of our KPIDL method and ensemble-based methods

Project	BRF	EasyEn	Bag	BBag	AdaB	KPIDL
Firewall	**0.752** ± 0.25	0.696 ± 0.20	0.289 ± 0.10	0.387 ± 0.06	0.493 ± 0.30	0.530 ± 0.14
Alfresco	0.693 ± 0.14	0.598 ± 0.08	**0.959** ± 0.01	0.494 ± 0.03	0.680 ± 0.34	0.714 ± 0.24
Sync	0.401 ± 0.13	0.448 ± 0.18	0.405 ± 0.21	0.369 ± 0.10	0.357 ± 0.17	**0.466** ± 0.16
Wallpaper	0.512 ± 0.07	0.523 ± 0.11	**0.631** ± 0.10	0.400 ± 0.06	0.380 ± 0.19	0.532 ± 0.12
Keyboard	0.473 ± 0.12	0.549 ± 0.22	0.516 ± 0.40	0.609 ± 0.24	0.740 ± 0.32	**0.822** ± 0.28
Apg	0.560 ± 0.16	**0.642** ± 0.23	0.526 ± 0.34	0.549 ± 0.19	0.467 ± 0.13	0.545 ± 0.24
Secure	0.582 ± 0.10	0.770 ± 0.20	0.583 ± 0.38	0.406 ± 0.02	0.389 ± 0.03	**0.926** ± 0.04
Kiwis	0.497 ± 0.18	0.534 ± 0.26	0.733 ± 0.28	0.441 ± 0.08	0.528 ± 0.25	**0.860** ± 0.05
Cloud	0.515 ± 0.08	0.589 ± 0.19	**0.690** ± 0.10	0.430 ± 0.03	0.475 ± 0.34	0.636 ± 0.23
Turner	0.186 ± 0.18	0.461 ± 0.36	**0.720** ± 0.13	0.628 ± 0.17	0.562 ± 0.11	0.503 ± 0.37
Reddit	0.413 ± 0.16	0.454 ± 0.13	0.421 ± 0.24	0.426 ± 0.10	0.448 ± 0.22	**0.486** ± 0.17
Image	0.398 ± 0.08	0.486 ± 0.14	**0.582** ± 0.15	0.376 ± 0.06	0.365 ± 0.21	0.456 ± 0.15
Scroll	0.523 ± 0.11	0.507 ± 0.12	**0.551** ± 0.24	0.464 ± 0.25	0.364 ± 0.17	0.517 ± 0.24
Applozic	0.419 ± 0.07	0.443 ± 0.11	0.569 ± 0.21	0.439 ± 0.08	0.362 ± 0.21	**0.776** ± 0.11
Delta	0.843 ± 0.21	0.801 ± 0.23	**0.993** ± 0.05	0.746 ± 0.27	0.865 ± 0.29	0.630 ± 0.26
Average	0.518 ± 0.16	0.567 ± 0.11	0.611 ± 0.18	0.478 ± 0.11	0.498 ± 0.15	**0.627** ± 0.15

Note: The best performance values among these methods achieved on each app are shown in bold.

TABLE 8 The Average Effort-Aware F-measure of our KPIDL method and ensemble-based methods

Project	BRF	EasyEn	Bag	BBag	AdaB	KPIDL
Firewall	**0.622** ± 0.16	0.604 ± 0.13	0.302 ± 0.08	0.388 ± 0.05	0.446 ± 0.20	0.520 ± 0.09
Alfresco	0.472 ± 0.04	0.427 ± 0.03	**0.563** ± 0.01	0.403 ± 0.03	0.444 ± 0.15	0.507 ± 0.06
Sync	0.361 ± 0.10	0.387 ± 0.12	0.346 ± 0.14	0.344 ± 0.09	0.323 ± 0.12	**0.443** ± 0.10
Wallpaper	0.348 ± 0.04	0.343 ± 0.05	**0.362** ± 0.05	0.299 ± 0.05	0.267 ± 0.10	0.360 ± 0.04
Keyboard	0.407 ± 0.09	0.458 ± 0.12	0.385 ± 0.23	0.478 ± 0.11	0.530 ± 0.16	**0.580** ± 0.13
Apg	0.487 ± 0.10	**0.533** ± 0.13	0.437 ± 0.21	0.479 ± 0.10	0.428 ± 0.08	0.495 ± 0.14
Secure	0.506 ± 0.06	0.581 ± 0.13	0.457 ± 0.24	0.38 ± 0.02	0.372 ± 0.02	**0.674** ± 0.02
Kiwis	0.396 ± 0.10	0.408 ± 0.15	0.494 ± 0.16	0.378 ± 0.04	0.408 ± 0.14	**0.576** ± 0.02
Cloud	0.402 ± 0.05	0.405 ± 0.10	0.448 ± 0.06	0.354 ± 0.02	0.309 ± 0.20	**0.503** ± 0.05
Turner	0.111 ± 0.10	0.226 ± 0.17	**0.356** ± 0.05	0.319 ± 0.06	0.354 ± 0.04	0.252 ± 0.17
Reddit	0.351 ± 0.11	0.381 ± 0.09	0.334 ± 0.15	0.372 ± 0.07	0.369 ± 0.13	**0.461** ± 0.10
Image	0.280 ± 0.06	0.323 ± 0.07	0.328 ± 0.07	0.277 ± 0.04	0.248 ± 0.09	**0.377** ± 0.06
Scroll	0.377 ± 0.09	0.37 ± 0.09	0.357 ± 0.13	0.350 ± 0.15	0.280 ± 0.09	**0.418** ± 0.11
Applozic	0.297 ± 0.05	0.295 ± 0.06	0.302 ± 0.11	0.322 ± 0.06	0.242 ± 0.10	**0.649** ± 0.20
Delta	0.269 ± 0.03	0.268 ± 0.03	0.287 ± 0.01	0.261 ± 0.04	0.267 ± 0.05	**0.310** ± 0.07
Average	0.379 ± 0.12	0.401 ± 0.11	0.384 ± 0.08	0.360 ± 0.06	0.352 ± 0.08	**0.475** ± 0.12

Note: The best performance values among these methods achieved on each app are shown in bold.

based decision Forest (CF) [29], and Balanced cost-sensitive decision Forest (BF) [29]. To construct the trees, three voting-based strategies are applied to these methods, including cascading-and-Sharing-based Voting (SV) [65], maximally Diversified multiple decision tree-based Voting (DV) [66], and Cost-sensitive Voting (CV) [29]. After combining each cost-

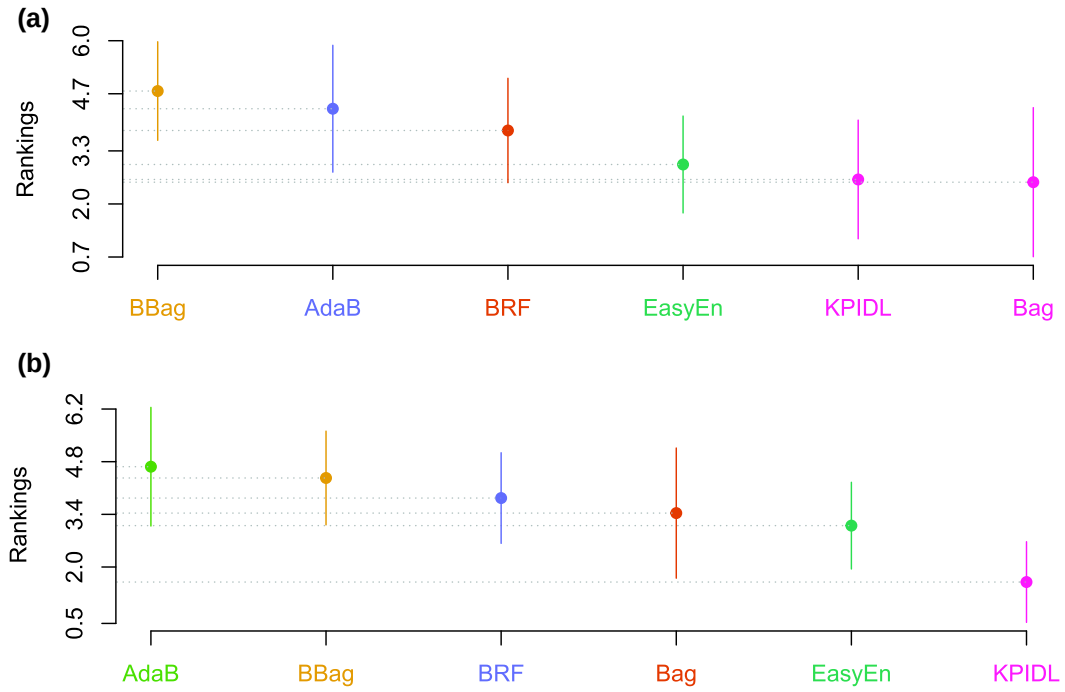

FIGURE 6 Scott-Knott Effect Size Difference test for our KPIDL method and ensemble-based methods. (a) EARecall, (b) EAF-measure

TABLE 9 The average EARecall of our KPIDL method and cost-sensitive-based methods

Project	SFSV	SFDV	SFCV	CFSV	CFDV	CFCV	BFSV	BFDV	BFCV	KPIDL
Firewall	0.519 ± 0.24	0.495 ± 0.23	**0.684** ± 0.25	0.610 ± 0.32	0.629 ± 0.32	0.567 ± 0.29	0.525 ± 0.29	0.497 ± 0.29	0.535 ± 0.29	0.530 ± 0.14
Alfresco	0.389 ± 0.12	0.404 ± 0.13	0.597 ± 0.20	0.461 ± 0.21	0.462 ± 0.23	0.554 ± 0.19	0.558 ± 0.25	0.559 ± 0.24	0.590 ± 0.20	**0.714** ± 0.24
Sync	0.377 ± 0.09	0.381 ± 0.09	0.411 ± 0.09	0.311 ± 0.10	0.297 ± 0.10	0.392 ± 0.11	0.360 ± 0.11	0.363 ± 0.11	0.388 ± 0.10	**0.466** ± 0.16
Wallpaper	0.478 ± 0.23	0.455 ± 0.20	0.482 ± 0.14	0.519 ± 0.20	**0.536** ± 0.19	0.495 ± 0.17	0.498 ± 0.18	0.476 ± 0.18	0.518 ± 0.13	0.532 ± 0.12
Keyboard	0.527 ± 0.25	0.480 ± 0.21	0.681 ± 0.22	0.524 ± 0.29	0.577 ± 0.31	0.597 ± 0.24	0.718 ± 0.30	0.700 ± 0.30	0.585 ± 0.28	**0.822** ± 0.28
Apg	0.560 ± 0.31	0.541 ± 0.28	0.600 ± 0.36	**0.642** ± 0.31	0.639 ± 0.35	0.579 ± 0.33	0.512 ± 0.33	0.511 ± 0.35	0.524 ± 0.32	0.545 ± 0.24
Secure	0.625 ± 0.27	0.576 ± 0.26	0.675 ± 0.28	0.673 ± 0.25	0.682 ± 0.27	0.680 ± 0.26	0.716 ± 0.29	0.707 ± 0.28	0.563 ± 0.28	**0.926** ± 0.04
Kiwis	0.450 ± 0.16	0.461 ± 0.18	0.691 ± 0.17	0.513 ± 0.24	0.452 ± 0.24	0.548 ± 0.20	0.609 ± 0.27	0.605 ± 0.26	0.609 ± 0.24	**0.860** ± 0.05
Cloud	0.431 ± 0.11	0.430 ± 0.11	0.609 ± 0.15	0.414 ± 0.11	0.377 ± 0.11	0.512 ± 0.14	0.505 ± 0.19	0.527 ± 0.19	0.559 ± 0.15	**0.636** ± 0.23
Turner	0.332 ± 0.20	0.332 ± 0.20	0.327 ± 0.19	0.534 ± 0.20	0.530 ± 0.20	0.447 ± 0.22	**0.559** ± 0.20	0.523 ± 0.21	0.500 ± 0.22	0.503 ± 0.37
Reddit	0.464 ± 0.12	0.446 ± 0.12	0.515 ± 0.11	0.447 ± 0.12	0.436 ± 0.12	**0.528** ± 0.11	0.493 ± 0.12	0.499 ± 0.12	0.520 ± 0.14	0.486 ± 0.17
Image	0.302 ± 0.09	0.320 ± 0.09	0.425 ± 0.08	0.303 ± 0.09	0.296 ± 0.09	0.369 ± 0.08	0.344 ± 0.09	0.354 ± 0.09	**0.460** ± 0.10	0.456 ± 0.15
Scroll	0.453 ± 0.13	0.462 ± 0.14	0.536 ± 0.15	0.419 ± 0.16	0.398 ± 0.16	**0.599** ± 0.16	0.440 ± 0.14	0.430 ± 0.13	0.510 ± 0.15	0.517 ± 0.24
Applozic	0.649 ± 0.17	0.649 ± 0.17	0.649 ± 0.17	0.645 ± 0.17	0.622 ± 0.17	0.649 ± 0.17	0.649 ± 0.17	0.649 ± 0.17	0.649 ± 0.17	**0.776** ± 0.11
Delta	0.844 ± 0.29	0.809 ± 0.33	0.661 ± 0.29	0.891 ± 0.20	0.877 ± 0.22	0.812 ± 0.28	**0.922** ± 0.14	0.906 ± 0.17	0.914 ± 0.15	0.630 ± 0.26
Average	0.493 ± 0.13	0.483 ± 0.12	0.57 ± 0.11	0.527 ± 0.15	0.521 ± 0.15	0.555 ± 0.11	0.56 ± 0.14	0.554 ± 0.14	0.562 ± 0.11	**0.627** ± 0.15

Note: The best performance values among these methods achieved on each app are shown in bold.

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: Tables 9 and 10 report the average EARecall and the corresponding standard deviations of our KPIDL method and the nine cost-sensitive-based imbalanced learning methods, individually. Figure 7 visualises the statistic test

TABLE 10 The Average EAF-measure of our KPIDL method and cost-sensitive-based methods

Project	SFSV	SFDV	SFCV	CFSV	CFDV	CFCV	BFSV	BFDV	BFCV	KPIDL
Firewall	0.503 ± 0.18	0.487 ± 0.17	**0.610** ± 0.15	0.532 ± 0.23	0.542 ± 0.22	0.521 ± 0.21	0.480 ± 0.21	0.462 ± 0.21	0.497 ± 0.21	0.520 ± 0.09
Alfresco	0.396 ± 0.09	0.398 ± 0.09	0.497 ± 0.08	0.42 ± 0.09	0.403 ± 0.09	0.491 ± 0.09	0.441 ± 0.10	0.447 ± 0.10	0.496 ± 0.09	**0.507** ± 0.06
Sync	0.378 ± 0.08	0.384 ± 0.08	0.380 ± 0.07	0.311 ± 0.08	0.308 ± 0.09	0.369 ± 0.07	0.352 ± 0.08	0.356 ± 0.09	0.359 ± 0.06	**0.443** ± 0.10
Wallpaper	0.301 ± 0.11	0.309 ± 0.10	0.372 ± 0.08	0.314 ± 0.08	0.322 ± 0.07	0.344 ± 0.06	0.332 ± 0.08	0.326 ± 0.08	**0.378** ± 0.07	0.360 ± 0.04
Keyboard	0.480 ± 0.16	0.461 ± 0.15	**0.594** ± 0.13	0.448 ± 0.17	0.470 ± 0.17	0.524 ± 0.14	0.530 ± 0.15	0.524 ± 0.15	0.501 ± 0.16	0.580 ± 0.13
Apg	0.487 ± 0.22	0.478 ± 0.19	0.497 ± 0.22	**0.519** ± 0.20	0.511 ± 0.23	0.495 ± 0.22	0.437 ± 0.22	0.431 ± 0.22	0.458 ± 0.21	0.495 ± 0.14
Secure	0.543 ± 0.16	0.515 ± 0.15	0.577 ± 0.15	0.569 ± 0.15	0.558 ± 0.16	0.580 ± 0.16	0.557 ± 0.17	0.553 ± 0.17	0.499 ± 0.17	**0.674** ± 0.02
Kiwis	0.444 ± 0.11	0.439 ± 0.11	**0.597** ± 0.09	0.436 ± 0.12	0.391 ± 0.13	0.494 ± 0.12	0.466 ± 0.13	0.464 ± 0.13	0.510 ± 0.14	0.576 ± 0.02
Cloud	0.441 ± 0.07	0.426 ± 0.06	**0.537** ± 0.08	0.423 ± 0.08	0.375 ± 0.07	0.495 ± 0.08	0.434 ± 0.10	0.444 ± 0.08	0.505 ± 0.07	0.503 ± 0.05
Turner	0.234 ± 0.10	0.228 ± 0.10	0.264 ± 0.11	0.294 ± 0.07	0.292 ± 0.07	0.291 ± 0.09	0.299 ± 0.09	0.297 ± 0.08	**0.316** ± 0.09	0.252 ± 0.17
Reddit	0.472 ± 0.09	0.458 ± 0.10	**0.524** ± 0.09	0.439 ± 0.10	0.431 ± 0.09	0.516 ± 0.08	0.490 ± 0.10	0.498 ± 0.10	0.498 ± 0.09	0.461 ± 0.10
Image	0.314 ± 0.07	0.320 ± 0.06	0.386 ± 0.05	0.309 ± 0.07	0.294 ± 0.07	0.375 ± 0.06	0.341 ± 0.07	0.343 ± 0.07	**0.413** ± 0.06	0.377 ± 0.06
Scroll	0.437 ± 0.11	0.425 ± 0.10	0.505 ± 0.11	0.372 ± 0.10	0.361 ± 0.10	**0.531** ± 0.10	0.418 ± 0.11	0.412 ± 0.11	0.481 ± 0.11	0.418 ± 0.11
Applozic	0.546 ± 0.30	0.546 ± 0.30	0.546 ± 0.30	0.545 ± 0.30	0.530 ± 0.30	0.546 ± 0.30	0.546 ± 0.30	0.546 ± 0.30	0.547 ± 0.30	**0.649** ± 0.20
Delta	0.258 ± 0.07	0.251 ± 0.08	0.296 ± 0.06	0.268 ± 0.04	0.266 ± 0.04	0.261 ± 0.06	0.273 ± 0.03	0.271 ± 0.03	0.275 ± 0.03	**0.310** ± 0.07
Average	0.416 ± 0.10	0.408 ± 0.09	**0.479** ± 0.11	0.413 ± 0.10	0.404 ± 0.10	0.456 ± 0.10	0.426 ± 0.10	0.425 ± 0.09	0.449 ± 0.08	0.475 ± 0.12

Note: The best performance values among these methods achieved on each app are shown in bold.

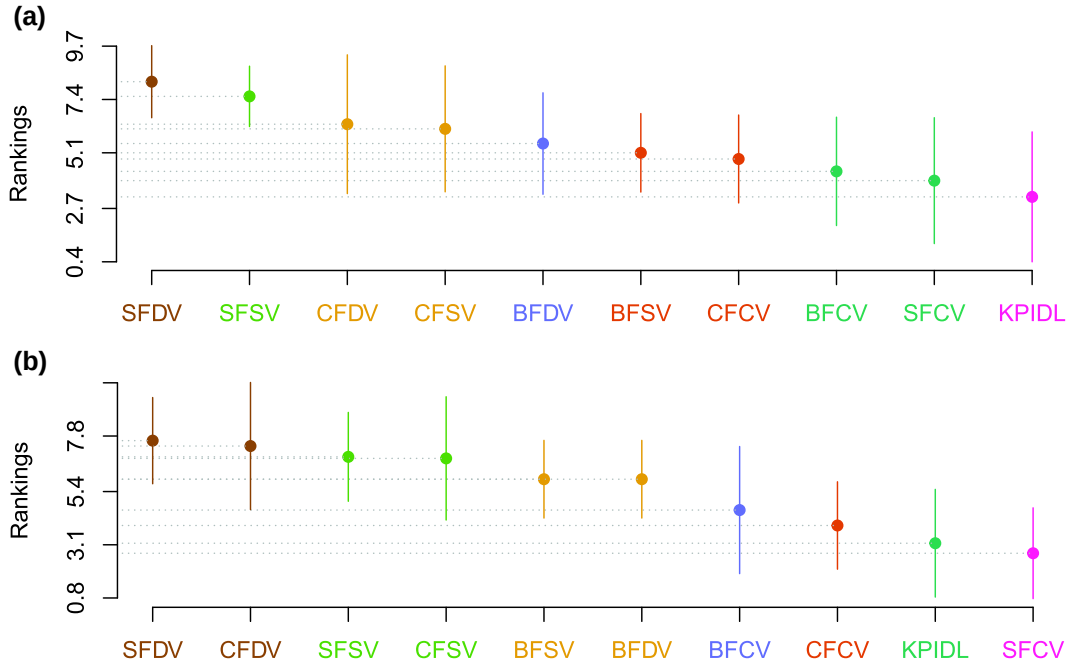

FIGURE 7 Scott-Knott Effect Size Difference (SKESD) test for our KPIDL method and cost-sensitive-based methods. (a) EARecall, (b) EAF-measure

results of SKESD for our KPIDL method and the nine baseline methods in terms of two effort-aware indicators. From these tables and the figure, the following observations can be drawn.

First, in terms of EARecall, our KPIDL method obtains better performance on 7 out of 15 apps compared with the nine baseline methods. The average EARecall value by our KPIDL method over all apps achieves improvements by

27.2%, 29.8%, 10.0%, 19.0%, 20.3%, 13.0%, 12.0%, 13.2%, and 11.6% compared with those of SFSV, SFDV, SFCV, CFSV, CFDV, CFCV, BFSV, BFDV, and BFCV, individually. Our KPIDL method obtains the best average EARecall value and achieves an average improvement by 17.3%.

Second, in terms of EAF-measure, our KPIDL method obtains better performance on 5 out of 15 apps compared with the nine baseline methods. The average EAF-measure value by our KPIDL method over all apps achieves improvements by 14.2%, 16.4%, 15.0%, 17.6%, 4.2%, 11.5%, 11.8%, and 5.8% compared with those of SFSV, SFDV, CFSV, CFDV, CFCV, BFSV, BFDV, and BFCV, individually, while obtains nearly the same EAF-measure value as SFCV. Our KPIDL method obtains the best average EAF-measure value and achieves an average improvement by 10.6%.

Third, our KPIDL method ranks the first and has significant differences compared with those of the nine cost-sensitive-based imbalanced learning methods in terms of all two effort-aware indicators except for the SFCV method in terms of EAF-measure.

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

6 | THREATS TO VALIDITY

6.1 | Threats to external validity

The generalisation of the experimental results threatens the external validity of this study. We conduct experiments on a publicly available benchmark dataset consisting of 15 Android mobile apps developed in the Java programing 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 Android-based mobile apps, it is necessary to investigate IOS-based mobile apps to verify the generalisation of our KPIDL method.

6.2 | Threats to internal validity

The implementation mistakes of the methods in our experiments threaten the internal validity of our study. In this study, we carefully implement the KPCA technique, CSCE loss function, and the DNN structure based on Scikit-learn, TensorFlow, and Python. As we specify multiple parameters empirically, 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 study. In this study, we employ two effort-aware indicators, that is EARecall and EAF-measure, which take the code inspection efforts into consideration when calculating, to evaluate the performance of our KPIDL method for JIT defect prediction on Android apps. In addition, to make our results more convincing, we apply a state-of-the-art statistic test method, that is SKESD, for the significant difference analysis between multiple methods.

7 | CONCLUSION

In this study, we propose a novel JIT defect prediction model, called KPIDL, for Android apps, which incorporates a feature learning stage and a classification model construction stage. More specifically, the KPCA technique used in the first stage is helpful to obtain high-quality feature representation for the defect data. Then, the improved version of DNN is able to alleviate the issue of class imbalance of the defect data by taking 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 KPIDL method, we conduct experiments on 15 Android apps and employ two effort-aware indicators for performance evaluation. The experimental results demonstrate that, in term of each indicator, our KPIDL method performs better than 24 out of 25 comparative methods, including its five variants, six sampling-based, five ensemble-based, and nine cost-sensitive-based methods.

In the future, we plan to collect more data from Android-based apps and IOS-based apps developed in other languages to enhance our experiments. In addition, our method will be adapted to cross-project scenarios for JIT defect prediction on apps.

ACKNOWLEDGEMENTS

This study was supported in part by the National Key Research and Development Project (No.2018YFB2101200), the National Natural Science Foundation of China (Nos.62002034, 62002306), the Fundamental Research Funds for the Central Universities (Nos.2020CDCGRJ072, 2020CDJQY-A021, and JUSRP121073), China Postdoctoral Science Foundation (No.2020M673137), the Special Funds for the Central Government to Guide Local Scientific and Technological Development (No.YDZX20195000004725), the Natural Science Foundation of Chongqing in China (No. cstc2020jcyj-bshX0114), the Key Project of Technology Innovation and Application Development of Chongqing (No.cstc2019jscx-mbdxX0020), HKPolyU Start-up Fund (No.ZVU7), CCF-Tencent Open Research Fund (No. ZDCK), and the European Commission grant (No.825,040) RADON.

CONFLICT OF INTEREST

Authors have no conflict of interest to declare.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in Github at https://github.com/sepine/IET-2021.

ORCID

Zhou Xu https://orcid.org/0000-0003-3307-2994

REFERENCES

1. Zhang, T., et al.: A literature review of research in bug resolution: tasks, challenges and future directions. Comput. J. 59(5), 741–773 (2016)
2. Xu, Z., et al.: Cross version defect prediction with representative data via sparse subset selection. In: Proceedings of the 26th International Conference on Program Comprehension (ICPC), pp. 132–143 (2018)
3. Xu, Z., et al.: Cross project defect prediction via balanced distribution adaptation based transfer learning. J. Comput. Sci. Technol. 34(5), 1039–1062 (2019)
4. McIlroy, S., Ali, N., Hassan, A.E.: Fresh apps: an empirical study of frequently-updated mobile apps in the google play store. Empir. Software Eng. 21(3), 1346–1370 (2016)
5. Menzies, T., Greenwald, J., Frank, A.: Data mining static code attributes to learn defect predictors. IEEE Trans. Software Eng. 33(1), 2–13 (2006)
6. Scanniello, G., et al.: Class level fault prediction using software clustering. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). (IEEE), pp. 640–645 (2013)
7. Kamei, Y., et al.: A large-scale empirical study of just-in-time quality assurance. IEEE Trans. Software Eng. 39(6), 757–773 (2012)
8. Yang, X., et al.: Deep learning for just-in-time defect prediction. In: Proceedings of the 15th IEEE International Conference on Software Quality, Reliability and Security (QRS). (IEEE), pp. 17–26 (2015)
9. Catolino, G.: 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), pp. 201–202 (2017)
10. Catolino, G., Di.Nucci, D., Ferrucci, F.: 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), pp. 99–110 (2019)
11. Liu, F., et al.: Software defect prediction model based on pca-isvm. Computer Simulation. 31(3), 397–401 (2014)
12. Cao, H., Qin, Z., Feng, T.: A novel pca-bp fuzzy neural network model for software defect prediction. Adv. Sci. Lett. 9(1), 423–428 (2012)
13. Shepperd, M., et al.: Data quality: some comments on the nasa software defect datasets. IEEE Trans. Software Eng. 39(9), 1208–1215 (2013)
14. Schölkopf, B., Smola, A., Müller, K.R.: Kernel principal component analysis. In: International conference on artificial neural networks. (Springer), pp. 583–588 (1997)
15. Kim, K.I., Franz, M.O., Scholkopf, B.: Iterative kernel principal component analysis for image modeling. IEEE Trans. Pattern Anal. Mach. Intell. 27(9), 1351–1366 (2005)
16. Schölkopf, B., Smola, A., Müller, K.R.: Nonlinear component analysis as a kernel eigenvalue problem. Neural Comput. 10(5), 1299–1319 (1998)
17. Xu, Z., et al.: Software defect prediction based on kernel pca and weighted extreme learning machine. Inf. Software Technol. 106, 182–200 (2019)
18. Aurelio, Y.S., et al.: Learning from imbalanced data sets with weighted cross-entropy function. Neural Process. Lett. 50(2), 1937–1949 (2019)
19. Arisholm, E., Briand, L.C., Fuglerud, M.: Data mining techniques for building fault-proneness models in telecom java software. In: Proceedings of the 18th IEEE International Symposium on Software Reliability (ISSRE). (IEEE), pp. 215–224 (2007)
20. Zhao, K., et al.: Just-in-time defect prediction for android apps via imbalanced deep learning model. In: Proceedings of the 36th Annual ACM Symposium on Applied Computing. pp. 1447–1454. (2021)
21. Xu, Z., et al.: The impact of feature selection on defect prediction performance: an empirical comparison. In: Proceedings of the 27th IEEE International Symposium on Software Reliability Engineering (ISSRE). (IEEE), pp. 309–320 (2016)
22. Liu, S., et al.: A feature selection framework for software defect prediction. In: Proceedings of the 38th IEEE Annual Computer Software and Applications Conference (COMPSAC). (IEEE), pp. 426–435 (2014)
23. Shivaji, S., et al.: Reducing features to improve code change-based bug prediction. IEEE Trans. Software Eng. 39(4), 552–569 (2012)
24. Chen, X., et al.: Applying feature selection to software defect prediction using multi-objective optimization. In: Proceedings of the 41st IEEE Annual Computer Software and Applications Conference (COMPSAC). (IEEE), 2, pp. 54–59 (2017)
25. Ghotra, B., McIntosh, S., Hassan, A.E.: A large-scale study of the impact of feature selection techniques on defect classification models. In: Proceedings of the 14th IEEE/ACM International Conference on Mining Software Repositories (MSR). (IEEE), pp. 146–157 (2017)
26. Ni, C., et al.: A cluster based feature selection method for cross-project software defect prediction. J. Comput. Sci. Technol. 32(6), 1090–1107 (2017)
27. Song, Q., Guo, Y., Shepperd, M.J.: A comprehensive investigation of the role of imbalanced learning for software defect prediction. IEEE Trans. Software Eng. 45(12), 1253–1269 (2019)
28. Liu, M., Miao, L., Zhang, D.: Two-stage cost-sensitive learning for software defect prediction. IEEE Trans. Reliab. 63(2), 676–686 (2014)
29. Siers, M.J., Islam, M.Z.: Software defect prediction using a cost sensitive decision forest and voting, and a potential solution to the class imbalance problem. Inf. Syst. 51, 62–71 (2015)
30. Bennin, K.E., et al.: The significant effects of data sampling approaches on software defect prioritization and classification. In: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). (IEEE), pp. 364–373 (2017)
31. Bennin, K.E., Keung, J., Monden, A.: Impact of the distribution parameter of data sampling approaches on software defect prediction models. In: 2017 24th Asia-Pacific Software Engineering Conference (APSEC). (IEEE), pp. 630–635 (2017)
32. Tantithamthavorn, C., Hassan, A.E., Matsumoto, K.: The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. IEEE Trans. Software Eng. 46(11), 1200–1219 (2018)
33. Bennin, K.E., et al.: Mahakil: diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction. IEEE Trans. Software Eng. 44(6), 534–550 (2017)
34. Fukushima, T., et al.: An empirical study of just-in-time defect prediction using cross-project models. In: Proceedings of the 11th Working Conference on Mining Software Repositories (MSR). pp. 172–181 (2014)
35. Kamei, Y., et al.: Studying just-in-time defect prediction using cross-project models. Empir. Software Eng. 21(5), 2072–2106 (2016)
36. McIntosh, S., Kamei, Y.: Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. IEEE Trans. Software Eng. 44(5), 412–428 (2017)
37. Yang, X., et al.: Tlel: a two-layer ensemble learning approach for just-in-time defect prediction. Inf. Software Technol. 87, 206–220 (2017)
38. Pascarella, L., Palomba, F., Bacchelli, A.: Fine-grained just-in-time defect prediction. J. Syst. Software. 150, 22–36 (2019)
39. Cabral, G.G., et al.: Class imbalance evolution and verification latency in just-in-time software defect prediction. In: Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE). (IEEE), pp. 666–676 (2019)
40. Kondo, M., et al.: The impact of context metrics on just-in-time defect prediction. Empir. Software Eng. 25(1), 890–939 (2020)
41. Ortu, M., et al.: 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, pp. 1–7 (2017)

42. Khomh, F., et al.: Predicting post-release defects using pre-release field testing results. In: Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM). (IEEE), pp. 253–262 (2017)
43. Scandariato, R., Walden, J.: Predicting vulnerable classes in an android application. In: Proceedings of the 4th International Workshop on Security Measurements and Metrics, pp. 11–16 (2012)
44. Kaur, A., Kaur, K., Kaur, H.: An investigation of the accuracy of code and process metrics for defect prediction of mobile applications. In: Proceedings of the 4th International Conference on Reliability, Infocom Technologies and Optimization. (IEEE), pp. 1–6 (2015)
45. Ricky, M.Y., Purnomo, F., Yulianto, B.: Mobile application software defect prediction. In: 2016 IEEE Symposium on Service-Oriented System Engineering. (IEEE), pp. 307–313 (2016)
46. Malhotra, R.: An empirical framework for defect prediction using machine learning techniques with android software. Appl. Soft Comput. 49, 1034–1050 (2016)
47. Kaur, A., Kaur, K., Kaur, H.: Application of machine learning on process metrics for defect prediction in mobile application. In: Information Systems Design and Intelligent Applications. (Springer), pp. 81–98 (2016)
48. Li, J., et al.: Software defect prediction via convolutional neural network. In: Proceedings of the 17th IEEE International Conference on Software Quality, Reliability and Security (QRS). (IEEE), pp. 318–328 (2017)
49. Phan, A.V., Le.Nguyen, M., Bui, L.T.: 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), pp. 45–52 (2017)
50. Manjula, C., Florence, L.: Deep neural network based hybrid approach for software defect prediction using software metrics. Cluster Comput. 22(4), 9847–9863 (2019)
51. Xu, Z., et al.: Ldfr: learning deep feature representation for software defect prediction. J. Syst. Software. 158, 110402 (2019)
52. Li, S.Z., et al. Kernel machine based learning for multi-view face detection and pose estimation. In: Proceedings of the 8th IEEE International Conference on Computer Vision. ICCV 2001. (IEEE), 2, pp. 674–679 (2001)
53. Huang, J., Yan, X.: Relevant and independent multi-block approach for plant-wide process and quality-related monitoring based on kpca and svdd. ISA (Instrum. Soc. Am.) Trans. 73, 257–267 (2018)
54. Li, W., et al.: On improving the accuracy with auto-encoder on conjunctivitis. Appl. Soft Comput. 81, 105489 (2019)
55. Yan, M., et al.: File-level defect prediction: unsupervised vs. supervised models. In: Proceedings of the 11th International Symposium on Empirical Software Engineering and Measurement (ESEM). (IEEE), pp. 344–353 (2017)
56. Huang, Q., Xia, X., Lo, D.: Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction. Empir. Software Eng. 24(5), 2823–2862 (2019)
57. Xu, Z., et al.: Tstss: a two-stage training subset selection framework for cross version defect prediction. J. Syst. Software. 154, 59–78 (2019)
58. Goodfellow, I., Bengio, Y., Courville, A.: Deep learning. The MIT Press (2016)
59. Tantithamthavorn, C., et al.: An empirical comparison of model validation techniques for defect prediction models. IEEE Trans. Software Eng. 43(1), 1–18 (2016)
60. Guo, L., et al.: Robust prediction of fault-proneness by random forests. In: Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE). (IEEE), pp. 417–428 (2004)
61. Catal, C., Diri, B.: Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem. Inf. Sci. 179(8), 1040–1058 (2009)
62. Xia, X., et al.: Cross-project build co-change prediction. In: Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER). (IEEE), pp. 311–320 (2015)
63. Magal, R.K., Gracia, J.S.: Improved random forest algorithm for software defect prediction through data mining techniques. Int. J. Comput. Appl. 117(23), 18–22 (2015)
64. Islam, M.Z., Giggins, H.: Knowledge discovery through sysfor - a systematically developed forest of multiple decision trees. In: Proceedings of the 9th Australasian Data Mining Conference, AusDM, pp. 195–204 (2011)
65. Li, J., Liu, H.: Ensembles of cascading trees. In: Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM). pp. 585–588 (2003)
66. Hu, H., et al.: A maximally diversified multiple decision tree algorithm for microarray data classification. In: Proceedings of the 2006 Workshop on Intelligent Systems for Bioinformatics. 73, pp. 35–38 (2006)

How to cite this article: Zhao, K., et al.: A compositional model for effort-aware Just-In-Time defect prediction on android apps. IET Soft. 1–20 (2021). https://doi.org/10.1049/sfw2.12040