Front. Comput. Sci., 2022, 16(6): 166207
https://doi.org/10.1007/s11704-021-1013-5

RESEARCH ARTICLE

Effort-aware cross-project just-in-time defect prediction framework for mobile apps

Tian CHENG[1], Kunsong ZHAO[2], Song SUN[1], Muhammad MATEEN[3], Junhao WEN (✉)[1]

1 School of Big Data and Software Engineering, Chongqing University, Chongqing 401331, China
2 School of Computer Science, Wuhan University, Wuhan 430072, China
3 Department of Computer Science, Air University Multan Campus, Multan 60000, Pakistan

© Higher Education Press 2022

Abstract As the boom of mobile devices, Android mobile apps play an irreplaceable roles in people's daily life, which have the characteristics of frequent updates involving in many code commits to meet new requirements. Just-in-Time (JIT) defect prediction aims to identify whether the commit instances will bring defects into the new release of apps and provides immediate feedback to developers, which is more suitable to mobile apps. As the within-app defect prediction needs sufficient historical data to label the commit instances, which is inadequate in practice, one alternative method is to use the cross-project model. In this work, we propose a novel method, called KAL, for cross-project JIT defect prediction task in the context of Android mobile apps. More specifically, KAL first transforms the commit instances into a high-dimensional feature space using kernel-based principal component analysis technique to obtain the representative features. Then, the adversarial learning technique is used to extract the common feature embedding for the model building. We conduct experiments on 14 Android mobile apps and employ four effort-aware indicators for performance evaluation. The results on 182 cross-project pairs demonstrate that our proposed KAL method obtains better performance than 20 comparative methods.

Keywords kernel-based principal component analysis, adversarial learning, just-in-time defect prediction, cross-project model

1 Introduction

Software, an irreplaceable product in people's everyday life, necessarily exists defects because of its increased scale and complexity. These defects may give rise to a large quantity of unexpectedly negative effects on user experience. Fixing the defects earlier is a challenging task for developers in maintenance process. Software defect prediction aims to detect whether a code snippet (such as a file, a method, or a class) is defective or clean by constructing models with machine learning techniques, which is a hot research topic for software quality assurance [1,2].

Many previous studies [3,4] focused on traditional software products (such as the desktop software) that cannot satisfy people's need to use software whenever and wherever possible. Mobile products, especially mobile apps, narrow the gap by allowing user download and update software products immediately and directly from app markets at any time. Since mobile apps need continuously update to meet new features or requirements, this process unavoidably brings defects into the new release of mobile apps on account of many uncontrollable factors. It is crucial to early recognize these defects and report them to the app developers for repairing timely before releasing to the public, aiming to improve the quality of mobile apps.

Most of the existing studies for software defect prediction mainly focus on code snippets at method or class level [5], which has a certain time delay in detecting defects. To discover possible defects timely, researchers came up with Just-in-Time (JIT) defect prediction [6,7] that aims to identify whether a code commit will introduce defects into the software products based on change level (or commit level). JIT defect prediction provides developers immediate feedback about the defects, which helps them recheck the code snippets that are still in their recollection. For this reason, JIT defect prediction is highly applicable for the software products relating to a great many of code updates (such as mobile apps). Catolino et al. [8] were the first to introduce JIT defect prediction into mobile apps and they built the classification model based on the features originated from the commit instances. If a new code commit instance brings defects into the app, this instance is deemed as defective, otherwise, clean. JIT defect prediction has the potential to accelerate the maintenance process and save developers' test efforts.

As new mobile apps are constantly being developed to satify the requirements from the public, they usually lack historical data to conduct the traditional within-project defect prediction task that usually needs sufficient historical data to collect labels information for the model building. To overcome this drawback, one optional solution is to develop a cross-project defect prediction model that applies the labeled data from

Received January 11, 2021; accepted June 25, 2021

E-mail: jhwen@cqu.edu.cn

other mobile apps (an app is regarded as a project) to aid the label identification for the new developed mobile apps. In this work, we propose a novel method combining the Kernel-based Principal Component Analysis (KPCA) with the Adversarial Learning (AL) technique (short for KAL) for cross-project JIT defect prediction. More specifically, as the prediction performance highly relies on the feature quality, we first apply the KPCA technique to convert the original commit instances into a high-dimensional feature space to obtain more representative features. In addition, as the transfer learning has the potential to find a common feature space in which features from the source app are most similar as those from the target app, we employ the AL technique to produce the common feature representation for the cross-project pair (i.e., the source app and the target app). The AL technique consists of a common feature generator that produces the commom embedding for each cross-project pair and an app discriminator that identifies whether the commit instances come from the source app (or the target app). After obtaining the common feature embedding, a classification model is built for JIT defect prediction task.

Previous defect prediction studies [9,10] always use the traditional confusion matrix based performance indicators, such as Precision, Recall, and F-measure, to evaluate the performance of the defect prediction model. It holds that the test resources are always available, which is unreasonable in the practical activitites. Recently, researchers proposed the effort-aware indicators [11] for traditional defect prediction task. It holds that only limited test resources are available for developers to inspect code snippets, which is more suitable for the practical cases. In this work, we pay attention to the effort-aware performance of our proposed KAL method for JIT defect prediction toward mobile apps.

In this work, we conduct experiments on a publicly available benchmark dataset including 14 Android mobile apps and employ four effort-aware indicators, that is Effort-Aware Recall (EARecall), Effort-Aware F-measure (EAF-measure), P_{opt}, and Proportion of Changes Inspected (PCI) to evaluate the performance of our proposed KAL method on 182 cross-project pairs. The results show that KAL obtains an average EARecall value of 0.704, an average EAF-measure value of 0.468, an average P_{opt} value of 0.825, and an average PCI value of 0.762 across all cross-project pairs. In addition, KAL obtains better performance than five instance selection based methods, seven transfer learning based methods, four classifier combination based methods, its three variants, and the state-of-the-art method.

The main contributions of this paper are highlighted as follows:

- To the best of our knowledge, we are the first to introduce the adversarial learning technique into the JIT defect prediction on mobile apps.
- We propose a novel method, called KAL, which employs the kernel-based principal component analysis method to convert original commit instances into a high-dimensional feature space, and then applies the adversarial learning technique to obtain the common feature representation.
- We employ the effort-aware indicators to evaluate the performance of our method for cross-project JIT defect prediction task. The experimental results show that our KAL method obtains satisfactory performance than 20 comparative methods.

The remainder of this paper is organized as follows. Section 2 presents the related work. Section 3 details our proposed KAL method. Section 4 and Section 5 describe the experimental setup and experimental results, individually. Section 6 discusses the threats to validity. Finally, we conclude our work and present the future work in Section 7.

2 Related work

2.1 Transfer learning based cross-project defect prediction

Transfer Learning aims to transform the information obtained from the source domain to improve the performance in the target domain. In the cross-project defect prediction scenario, transfer learning transforms the data from the source domain and the target domain into a new feature space in which the transformed data has the most similar distribution. The aim is to make that the transformed data is more effective than the original one for classification.

Ma et al. [12] were the first to introduce transfer learning based method into cross-project defect prediction. They proposed a method, called TNB, which employed a weighted Naive Bayes model for information transformation from the source project to the target project. More specifically, TNB first calculated the feature information from the target project and then compared them with each sample in the source project to calculate the degree of similarity. Then, each sample was given a weight based on the obtained similarity and the model was built on the weighted samples. They conducted experiments on NASA and SOFTLAB datasets and the results showed that TNB obtained the better prediction performance than two baseline methods. Nam et al. [13] proposed a TCA+ method that extended the Transfer Component Analysis (TCA) technique into the cross-project defect prediction scenario. TCA+ first sought the appropriate data normalization strategy by using the predefined decision rules. Then, the TCA technique was used to narrow the data distribution of the source and target projects. They conducted experiments on AEEEM and RELINK datasets and the results showed the superiority of TCA+. Chen et al. [14] proposed a transfer learning based method DTB that applied data gravitation to re-weight the samples in source project for distribution similarity inprovement and then relieved the negative samples in the source project based on a transfer boosting technique and a small quantity of labeled samples. Their experimental results on 15 projects showed that DTB performed better than 4 baseline methods for cross-project defect prediction. Ryu et al. [15] proposed a cost-sensetive based transfer learning method TCSBoost that introduced class imbalance issues into information transformation process for cross-project defect prediction. More specifically, the similarity weights were calculated based on the distribution characteristics between two projects and the data distribution of the two kinds of

classes in source project were rebalanced. After that, the cost-sensetive boost technique was employed to address the distribution diversity between the source and target projects. Their experimantal results on 15 projects showed the effectiveness of this method compared with five baseline methods. Liu et al. [16] proposed a two-phase method, namely TPTL, for cross-project defect prediction. More specifically, in the first phase, TPTL automatically selected two source projects as candidates in which these projects had the highest distribution similarity with the target project, and then two source projects with the best F1-score and cost-effectiveness were selected from the condidate set. In the second phase, the TCA+ method was applied to build the transfer learning models based on the two selected projects. They conducted experiments on 42 versions from 14 projects and the results showed that TPTL outperformed the five comparative methods. Xu et al. [17] employed a novel Balanced Distribution Adaptation (BDA) based method for cross-project defect prediction. BDA simultaneously took the marginal distribution and conditional distribution of data into account and automatically assigned the weights for the two kinds of distributions, aiming at obtaining the optimum information transformation. They conducted experiments on AEEEM, NASA, SOFTLAB, and RELINK datasets and the results demonstrated that BDA achieved better performance than 12 comparative methods in terms of six indicators.

Different from the abovementioned studies that mainly focused on the cross-project defect prediction on the file-level or method-level, in this work, we pay attention to the JIT defect prediction which is on the change-level.

2.2 JIT defect prediction

As traditional software defect prediction studies cannot detect the possible defects timely, researchers focus on JIT defect prediction. Kamei et al. [6] proposed to use 14 matrics to characterize software changes for JIT defect prediction task. They conducted experiments on 11 projects and the results illustrated that JIT quality assurance reduced the costs of high-quality software development. Kamei et al. [7] then extended their work [6] to the cross-project scenario for JIT defect prediction models. Their experimental results on 11 projects demonstrated that JIT models performed better in the cross-project scenario when the training data were selected carefully. McIntosh et al. [18] conducted a longitudinal case study of 37,524 changes to explore JIT models as system evlove and found that the important factors of fix-inducing changes fluctuated as system evolve. Pascarella et al. [19] proposed a novel fine-grained JIT defect prediction model that used 24 basic features to characterize each file involved in a commit. Their experimental results on 10 projects showed that nearly half of the defective commits included defect-inducing and clean changes. In addition, researchers also considered the effort-aware issues [20] and class imbalanced issues [21] for JIT defect prediction task.

Different the above studies that conduct JIT defect prediction on the traditional software, Catolino et al. [8] took the first attempt to introduce the JIT defect prediction model into the mobile app scenario. They empirically investigated which Kamei et al.'s matrics [6] were more suitable for JIT models in the context of Andriod mobile apps and also explored the effectiveness of four classic classification models and four ensemble learning techniques for JIT defect prediction. They conducted experiments on 14 mobile apps and the results illustrated that Naive Bayes obtained the best prediction performance compared with the eight baseline methods.

Different from the previous studies that separately focused on the traditional software for JIT defect prediction or with the classic indicators holding that the test resource was sufficient, which was unrealistic in practice, in this work, we concentrate on the JIT defect prediction on mobile apps and with effort-aware indicators simultaneously.

3 Method

3.1 Framework

As the original features cannot reveal the potential structure information of the commit data, we utilize a kernel function based feature extraction method to learn the high-quality feature representation. To narrow the data distribution difference and capture the common feature from each cross-project pair, we employ the adversarial learning technique to obtain the common feature embedding. Figure 1 depicts an overview of our proposed KAL method, which consists of a Kernel-based Principal Component Analysis (KPCA) based mapping stage and a common feature extraction stage. More specifically, for each commit instance derived from the source

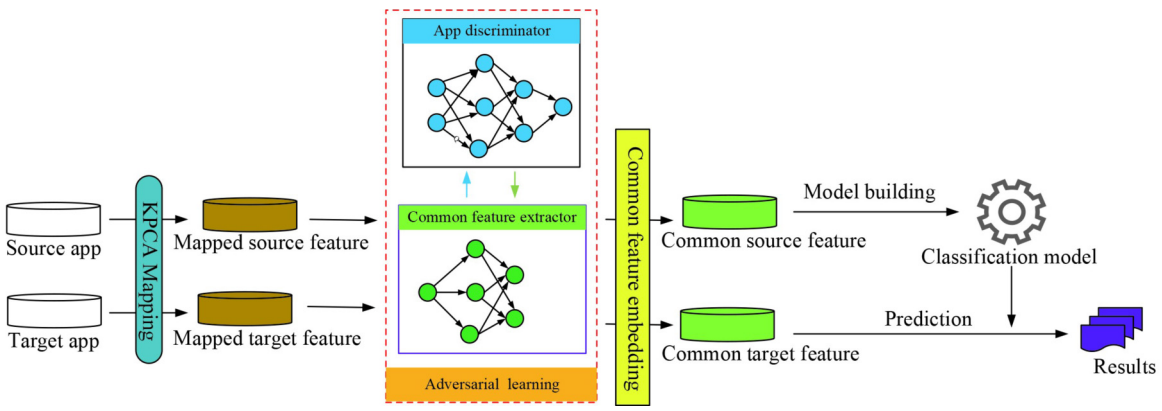

Fig. 1 An overview of our proposed KAL method

app and the target app, we first map the original features into a high-dimensional space to obtain the representative features. After that, the mapped features from each cross-project pair are used to train the adversarial learning model to obtain the common embedding. The adversarial learning techinque consists of a common feature generator and an app discriminator, in which the generator produces the same feature representation as the commit instance from the source (or target) app, and the discriminator estimates which kind of apps (source or target) the commit instance belongs to. After obtaining the common embedding for each cross-project pair, a classification model is built to identify whether the new commit instance is defective or clean. Before running our method, the z-score method is used to standardize the original features. Below, we detail the used KPCA method and the adversarial learning method, respectively.

3.2 Kernel-based principal component analysis (KPCA)

KPCA [22−25] is a non-linear feature mapping technique, which converts the low-dimensional feature into a high-dimensional feature space F using a non-linear mapping function Φ. In the first stage, we use the KPCA technique to obtain the representative features of the commit data.

For a given feature set of the commit instance $X = \{x_1, x_2, ..., x_m\} \in \mathbb{R}^{n\times m}$, where m denotes the number of features and n presents the number of commit instances, we assume that $\Phi(x_i)$ is the centralized projection point of $x_i (i = 1, 2, ..., m)$. Then, the covariance matrix C is defined as:

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

To solve the eigenvalue problem in the eigenspace, the covariance matrix C is diagonalized by performing a linear PCA transformation in feature space F

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

where $\lambda_i \in \lambda$ denotes the eigenvalues ($\lambda_i \geqslant 0$), and V denotes the eigenvectors of matrix C.

Because of all eigenvectors corresponding to $\lambda \geqslant 0$ lie in the span of $\Phi(x_1), \Phi(x_2), ..., \Phi(x_n)$, so there exists coefficients α_j that

$$V = \sum_{j=1}^{n}\alpha_j\Phi(x_j). \tag{3}$$

However, it is unpractical that we specify a concrete form of Φ. Thus, we define the kernel function $\kappa(x_i, x_j)$ as follows:

$$\kappa(x_i, x_j) = \Phi(x_i)^{\mathrm{T}}\Phi(x_j). \tag{4}$$

We can get the following formula by substituting Eq. (3) and Eq. (4) into Eq. (2) as

$$K\alpha = n\lambda\alpha, \tag{5}$$

where K is kernel matrix correspond to κ, $K_{ij} = \kappa(x_i, x_j)$, and $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)^{\mathrm{T}}$.

Given a specific commit instance x, the lth kernel component based on the non-linear projection is formulized as

$$V^l\Phi(x) = \sum_{i=1}^{n}\alpha_i^l\Phi(x_i)\Phi(x) = \sum_{i=1}^{n}\alpha_i^l\kappa(x_i, x). \tag{6}$$

To obtain the representative features, in this work, we apply the Gaussian radial basic function as kernel function, which is formulized as

$$\kappa(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{\delta^2}\right), \tag{7}$$

where $\|\cdot\|$ denotes the L_2 norm and δ is a kernel parameter.

For each commit instance from each cross-project pair, we apply the KPCA method to convert the original features into the high-dimensional space to obtain the representative features, and then we take the converted features as the candidate for the adversarial learning.

3.3 Adversarial learning

After obtaining the representative features, the goal for the second stage is to obtain the common feature representation that simultaneously narrows the discrimination in each cross-project pair and characterizes each cross-project pair in an unified manner. For this purpose, the adversarial learning technique [26,27] is employed to extract the common feature embedding from the cross-project pair.

Generally, the adversarial learning includes a generator and a discriminator. The generator is used to learn the distribution from each cross-project pair whereas the discriminator takes the advantage to measure which kind of apps a commit instance comes from. In this work, we take the common feature extractor as the generator.

For the common feature extractor, we apply the **D**eep **N**eural **N**etwork (**DNN**) which consists of three types of layers: the input layer, the hidden layer, and the output layer, to learn the feature embedding. Assume that G represents the structure of common feature extractor, the common feature is formulized as

$$g_{com}^{\gamma} = G(X^{\gamma}), \tag{8}$$

where X denotes the initial embedding processed by KPCA technique and $\gamma \in \{s, t\}$ denotes the commit instances from the source app or the target app.

In addition, we employ the Binary Cross Entropy (BCE) loss function to optimize the parameters in the generator, which is formulized as

$$l(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \cdot \log\hat{y}_i + (1 - y_i)\cdot\log(1 - \hat{y}_i)\right], \tag{9}$$

where $\hat{y}_i$ denotes the ouput of the DNN model corresponding to the i-th commit instance.

For the common feature extractor, we calculate the total BCE loss l_G for the cross-project pair, which is formulized as

$$l_G(\theta_G) = l_s(\theta_G) + l_t(\theta_G), \tag{10}$$

where θ_G denotes the parameter estimated by the generator, $l_s(\theta_G)$ ($l_t(\theta_G)$) denotes the loss produced by generator, which comes from the source (target) app.

To narrow the difference of commit instances in each cross-project pair during the adversarial learning process, we also employ the DNN structure as the app discriminator to evaluate which kind of apps the commit instance comes from. Similarly, for the loss from the discriminator (short for l_D), we can also define the following formula

$$l_D(\theta_D) = l_s(\theta_D) + l_t(\theta_D), \tag{11}$$

where θ_D denotes the parameter estimated by the app discriminator.

To obtain the common feature embedding in the adversarial learning, the generator G and the discriminator D play the minimax two-player game, which is formulized as follows:

$$l_{total} = \min_{\theta_G}(\max_{\theta_D} \sum_{\gamma} \sum_{t=1}^{T^{\gamma}} \log D(G(x_i^{\gamma})), \tag{12}$$

where T^{γ} denotes the number of commit instances from γ, and x_i^{γ} denotes the ith commit instance from γ.

In this work, we take all the commit instances after the preprocessing of the KPCA technique from each cross-project pair as candidate to excute the adversarial learning. After that, the converted commit instances from the source app and the target app are seperately input into the common feature extactor to obtain the common feature embedding.

4 Experimental setup

4.1 Dataset

To evaluate the effectiveness of our proposed KAL method, in this work, we conduct experiment on a benchmark dataset provided by previous work [8], which contains 14 Android mobile apps: Android **Firewall**, **Alfresco**, Android **Sync**, Android **Wallpaper**, AnySoft**Keyboard**, **Apg**, **Atmosphere**, Chat **Secure** Android, **Facebook** Android SDK, **Flutter**, **Kiwix**, Own **Cloud** Android, Page **Turner**, and Notify **Reddit**. Table 1 presents the statistic information of these apps, including lines of code (# LOC), the number of commit instances (# Instances), the number of defective commit instances (# Defective) and clean commit instances (# Clean), and the defective ratio (% Ratio). These apps originate from different fields and the inequable LOC means that these apps have different scales.

Each commit instance from each app is characterized by six features including Number of Unique Change to modified files (NUC), Number of DEVelopers working on the files (NDEV), Lines of code Deleted (LD), Lines of code Added (LA), Number of modifiled Files (NF), and Number of modified Directories (ND) [8].

Table 1 Statistic information of these apps

Project	# LOC	# Instances	# Defective	# Clean	Ratio/%
Firewall	77,243	1025	414	611	40.39
Alfresco	152,047	1004	214	790	21.31
Sync	275,637	209	62	147	29.67
Wallpaper	35,917	588	94	494	15.99
Keyboard	114,784	2971	819	2152	27.57
Apg	151,204	3780	1304	2476	34.50
Atmosphere	56,686	5474	2174	3300	39.72
Secure	98,768	2579	853	1726	33.07
Facebook	103,802	548	180	368	32.85
Flutter	639,350	10405	130	10275	1.25
Kiwix	32,598	1373	350	1023	25.49
Cloud	115,169	3700	830	2870	22.43
Turner	30,943	164	23	141	14.02
Reddit	9,506	222	60	162	27.03

4.2 Performance indicators

In this work, we use the effort-aware indicators to evaluate the performance of our proposed KAL method. Compared with the classical classification indicators, such as Precision, Recall, and F-measure, which are based on the adequate test sources for code reviews and treat efforts for inspecting different code snippets as the same, effort-aware indicators take the inspection efforts into account to meet the practical need that only restricted test sources are available. We treat the sum of features LA and LD as the proxy to evaluate the efforts inspecting a commit instance and the restricted test source is considered as 20% of all the efforts. Following the previous studies [23,2], the brief description of calculating the effort-aware indicators is introduced in the following.

Firstly, when obtaining the common feature embedding based on the commit instances from the cross-project pair by using our KAL method, the classification model is built on the embedded commit instances from the source app and predicts the commit instances from the target app as two groups, that is defective and clean. Secondly, the commit instances for each group are sorted ascendingly by their inspection efforts individually. Thirdly, the sorted commit instances in two groups are merged in which the defective commit instances are put in the front. Next, we simulate the developers to inspect merged commit instances in order from high to low until the accumulated effort percent of the inspected commit instances reaches to 20%. Then, we apply the statistic information of the inspected commit instances to calculate the effort-aware indicators. Table 2 presents the description of three basic terms for the indicator calculation. Below, we briefly introduce the effort-aware indicators used in this work.

Effort-Aware Recall (EARecall) is the first indicator used in our work, which is defined as the percentage of inspected defective commit instances with 20% of efforts among all defective commit instances in the target app. It is formulized as:

$$\text{EARecall} = n_{id}/n_d. \tag{13}$$

Effort-Aware Precision (EAPrecision) is defined as the percentage of inspected defective commit instances among all inspected commit instances with 20% of efforts, which is formulized as EAPrecision = n_{id}/n_i.

Effort-Aware F-measure (EAF-measure) is the second indicator used in our work. It is the weighted harmonic average of EARecall and EAPrecision like classical F-measure, which is formulized as:

$$\text{EAF-measure} = \frac{(1+\theta^2)\times\text{EAPrecision}\times\text{EARecall}}{\theta\times\text{EAPrecision}+\text{EARecall}}, \tag{14}$$

where θ is a weight parameter. In this work, we set it as 2 following the previous studies [23,28,2].

P_{opt} is the third indicator used in our work, which is derived

Table 2 The description of three basic terms

Name	Description
n_d	the number of defective commit instances in the target app
n_i	the number of all the inspected commit instances with 20% of efforts
n_{id}	the number of inspected defective commit instances with 20% of efforts

from the area under the effort curve in an Alberg diagram [29,30]. It relies on three types of curves that include an optimal model (opt), our model (m), and a worst model (worst). The optimal model indicates that the defective and clean commit instances are sorted in an ascending order by their LOC individually and these two sorted groups are merged in which the defective one is put in the front. The m indicates that all commit instances are sorted by the above ranking strategy. The worst model contains the opposite results of the optimal model. This indicator is formulized as:

$$\mathrm{P_{opt}} = \frac{Area(\mathrm{m}) - Area(\mathrm{worst})}{Area(\mathrm{opt}) - Area(\mathrm{worst})}, \tag{15}$$

where $Area()$ denotes the area under the corresponding curve. The large $\mathrm{P_{opt}}$ value means the smaller difference between our model and the optimal model.

The last indicator used in our work is called Proportion of Changes Inspected (PCI) that is the defined as the percentage of the inspected commit instances among all commit instances with 20% of efforts [31]. It is formulized as:

$$\mathrm{PCI} = n_i/n. \tag{16}$$

4.3 Parameter settings

In this work, we use DNN as basic structure to build the common feature extractor and the app discriminator. For both models, the DNN structure consists of two hidden layers with 16 units. We set the batch size as 32 and the adversarial learning process repeats 100 times. In addition, the adaptive moment estimation algorithm is used to optimize the parameters with the learning rate as 0.001 for both the common feature extractor and the app discriminator during the adversarial training process.

4.4 Classification model

When acquiring the common embedding of commit instances for the cross-project pair, the source app data is used to build a classification model to discern whether the commit instances from target app data will introduce defects into the apps. For this purpose, we apply the Random Forest (RF) [32] as the basic classifier to construct the predictor.

Taking the decision tree as basic learner, RF further introduces the random feature selection processing into the model building procedure. Different from the classical decision tree that selects the optimal feature from the feature set of each node as candidate to split the features, RF selects the global optimal feature as candidate to split the features. More specifically, for each node of the base decision tree, a subset containing multiple features is first randomly selected from the feature set of this node. Then, an optimal feature is selected from this feature subset for features partition. RF is commonly used in the defect prediction studies [33,2,34].

4.5 Statistic test

In this work, we analyze the significant differences between our proposed KAL method and other baseline methods using the Friedman test with the Nemenyi post-hoc test [35] at significance level $\alpha = 0.05$. The non-parametric hypothesis Friedman test has the ability to detect whether there exist significant differences among the comparative methods. In addition, the Nemenyi post-hoc test can determine whether the difference between two methods exceeds a critical distance to divide the methods into different ranking groups [35].

4.6 Research questions

To evaluate the effectiveness of our proposed KAL method, in this work, we design the following four research questions (RQs).

RQ1: *Is our proposed KAL method superior to instance selection based cross-project model for mobile apps JIT defect prediction performance?*

Instance selection based cross-project defect prediction methods select parts of commit instances from the source app which are representative to the target app. This question is designed to explore whether our proposed KAL method performs better than instance selection based methods in terms of JIT defect prediction performance on mobile apps.

RQ2: *Does our proposed KAL method achieve better mobile apps JIT defect prediction performance than transfer learning based cross-project model?*

Transfer learning based cross-project defect prediction methods learn a shared feature space to narrow the differences of data distribution in different apps by applying the feature transformation techniques. This question is designed to investigate whether our proposed KAL method achieves better JIT defect prediction performance on mobile apps than transfer learning based methods.

RQ3: *Does our proposed KAL method outperform classifier combination based cross-project model for mobile apps JIT defect prediction performance?*

Classifier combination based cross-project defect prediction methods apply the ensemble strategies to enhance the cross-project defect prediction performance by integrating predicted results of multiple classifiers. This question is designed to explore whether our proposed KAL method performs better than classifier combination based methods for JIT defect prediction performance on mobile apps.

RQ4: *Is our proposed KAL method superior to its variants and the state-of-the-art method for cross-project JIT defect prediction on mobile apps?*

As our KAL includes the KPCA method for obtaining representitive features and the AL technique for common feature extraction, this question is designed to investigate whether our KAL method is superior to the model combinations and the state-of-the-art method to improve the JIT defect prediction performance.

5 Experimental results

5.1 Answer to RQ1: the prediction performance of our proposed KAL method and the instance selection based methods

Methods: To answer this question, we choose four instance selection based methods as baselines, including Nearest Neighbor Filtering (NNF) [36], Peters Filter (PF) [37], Density-based Spatial Clustering Filter (DSCF) [38], and Agglomerative Clustering based Filter (ACF) [39]. Below, we briefly describe these baseline methods.

- NNF selects the 10 nearest commit instances from the source app for each commit instance in the target app. Then, the redundant commit instances are filtered out and the remains are as candidate for model training.
- PF first applies k-means algorithm to partition commit instances combining the source app and the target app into multiple clusters. Then the clusters that include at least one commit instance from the target app are retained. In each retained cluster, for each commit instance from the source app, its nearest commit instance from the target app is called the popularity instance. For each popularity instance, PF selects its nearest commit instance from the source app in the retained cluster as the candidate for model training.
- DSCF applies the density-based spatial clustering algorithm to partition the commit instances combining the source app and the target app into multiple clusters. Then the clusters that include at least one commit instance from the target app are retained. Finally, the commit instances from the source app in the retained clusters are collected as the candidate for model training.
- ACF first uses the the agglomerative clustering algorithm to partition the commit instances from the source app and the target app into mutiple clusters. Then the clusters that include at least one commit instance from the target app are retained. Finally, the commt instances from the source app in the retained clusters are collected as the candidate for model training.

In addition, we also employ the method that only applies the RF classifier (short for NONE) as the most basic baseline for comparison.

Results: Tables 3–6 present the results of the indicator values and the corresponding standard deviations (in the brackets) for our proposed KAL method and the five instance selection based methods in terms of EARecall, EAF-measure, P_{opt}, and PCI, individually. Note that each row denotes the 14 average results taking the corresponding project as the target app on each method. Figure 2 illustrates the corresponding statistical test results of our KAL method and other comparative methods across 182 cross-project pairs. Different colors mean that these methods belong to distinct rank groups (top, middle, and bottom). From these tables and the figure, we can draw the following findings.

Table 3 EARecall values of our KAL method and the instance selection based methods

Project	NONE	NNF	PF	DSCF	ACF	KAL
Firewall	0.363	0.431	0.537	0.412	0.543	**0.924**
Alfresco	0.491	0.503	0.602	0.548	0.453	**0.802**
Sync	0.291	0.296	0.314	0.310	0.297	**0.445**
Wallpaper	0.295	0.298	0.358	0.399	0.345	**0.566**
Keyboard	0.421	0.365	0.600	0.469	0.551	**0.939**
Apg	0.617	0.665	**0.754**	0.684	0.684	0.712
Atmosphere	0.630	0.640	0.657	0.616	0.694	**0.839**
Secure	0.441	0.418	0.580	0.474	0.527	**0.724**
Facebook	0.434	0.369	0.422	0.504	0.311	**0.673**
Flutter	0.421	0.354	0.456	0.449	0.446	**0.777**
Kiwix	0.475	0.409	0.456	0.456	0.441	**0.717**
Cloud	0.375	0.416	0.412	0.487	0.514	**0.559**
Turner	0.364	0.361	0.373	0.344	0.367	**0.470**
Average	0.432	0.425	0.502	0.473	0.475	**0.704**

Table 4 EAF-measure values of our KAL method and the instance selection based methods

Project	NONE	NNF	PF	DSCF	ACF	KAL
Firewall	0.330	0.389	0.446	0.372	0.480	**0.717**
Alfresco	0.373	0.370	0.439	0.405	0.381	**0.497**
Sync	0.269	0.265	0.282	0.290	0.280	**0.352**
Wallpaper	0.239	0.241	0.277	0.308	0.269	**0.367**
Keyboard	0.357	0.322	0.464	0.395	0.437	**0.646**
Apg	0.512	0.553	**0.602**	0.563	0.547	0.550
Atmosphere	0.531	0.537	0.544	0.516	0.565	**0.637**
Secure	0.393	0.374	0.469	0.411	0.449	**0.548**
Facebook	0.186	0.189	0.190	0.232	0.134	**0.243**
Flutter	0.289	0.249	0.270	0.297	0.287	**0.373**
Kiwix	0.391	0.349	0.367	0.384	0.361	**0.467**
Cloud	0.295	0.324	0.277	0.314	**0.352**	0.316
Turner	0.321	0.323	0.349	0.315	0.334	**0.375**
Average	0.345	0.345	0.383	0.369	0.375	**0.468**

Table 5 P_{opt} values of our KAL method and the instance selection based methods

Project	NONE	NNF	PF	DSCF	ACF	KAL
Firewall	0.686	0.717	0.704	0.690	0.729	**0.947**
Alfresco	0.725	0.705	0.711	0.722	0.680	**0.863**
Sync	0.603	0.588	0.635	0.622	0.590	**0.731**
Wallpaper	0.633	0.610	0.625	0.613	0.614	**0.720**
Keyboard	0.656	0.671	0.743	0.710	0.741	**0.932**
Apg	0.797	0.801	0.801	**0.827**	0.716	0.806
Atmosphere	0.773	0.788	0.760	0.738	0.738	**0.852**
Secure	0.663	0.662	0.722	0.653	0.683	**0.788**
Facebook	0.699	0.683	0.694	0.697	0.698	**0.796**
Flutter	0.706	0.684	0.700	0.697	0.641	**0.851**
Kiwix	0.666	0.648	0.696	0.696	0.643	**0.886**
Cloud	0.578	0.627	0.652	0.677	0.677	**0.785**
Turner	0.581	0.595	0.589	0.570	0.547	**0.762**
Average	0.674	0.675	0.695	0.686	0.669	**0.825**

Table 6 PCI values of our KAL method and the instance selection based methods

Project	NONE	NNF	PF	DSCF	ACF	KAL
Firewall	0.324	0.407	0.527	0.400	0.490	**0.943**
Alfresco	0.473	0.517	0.559	0.531	0.358	**0.841**
Sync	0.345	0.400	0.380	0.348	0.306	**0.611**
Wallpaper	0.318	0.314	0.355	0.403	0.327	**0.614**
Keyboard	0.381	0.311	0.575	0.428	0.519	**0.942**
Apg	0.593	0.650	**0.754**	0.682	0.662	0.706
Atmosphere	0.601	0.610	0.620	0.605	0.670	**0.844**
Secure	0.416	0.397	0.579	0.495	0.509	**0.753**
Facebook	0.416	0.324	0.389	0.502	0.316	**0.687**
Flutter	0.401	0.305	0.460	0.425	0.408	**0.811**
Kiwix	0.427	0.360	0.417	0.388	0.416	**0.825**
Cloud	0.370	0.367	0.449	0.588	0.528	**0.744**
Turner	0.434	0.398	0.305	0.357	0.336	**0.582**
Average	0.423	0.412	0.490	0.473	0.450	**0.762**

First, in terms of Table 3, our proposed KAL method obtains better performance on 13 out of 14 apps compared with the five baseline methods. The average EARecall value by KAL over all apps achieves improvements by 62.9%,

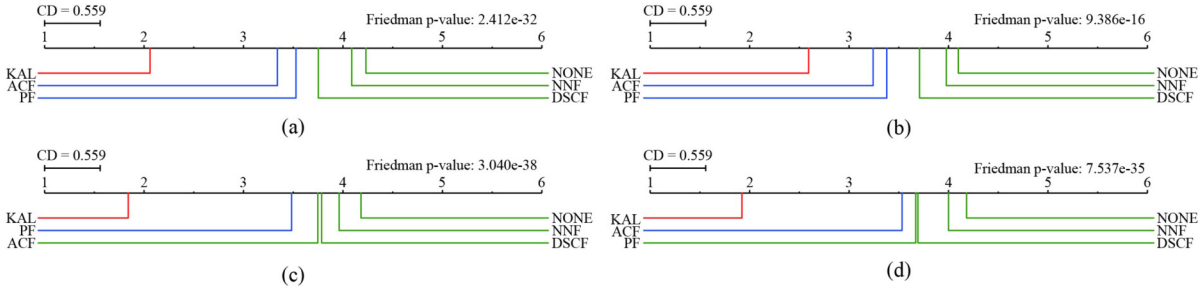

Fig. 2 Statistical test for our KAL method and the five instance selection based methods. (a) EARecall; (b) EAF-measure; (c) P_{opt}; (d) PCI

65.6%, 40.3%, 48.7%, and 48.2% compared with NONE, NNF, PF, DSCF, and ACF, individually. Our proposed KAL method obtains the best average value and achieves an average improvement by 53.1% with the EARecall indicator.

Second, in terms of Table 4, our proposed KAL method obtains better performance on 12 out of 14 apps compared with the five baseline methods. The average EAF-measure value by KAL over all apps achieves improvements by 35.7%, 35.7%, 22.2%, 26.8%, and 24.8% compared with NONE, NNF, PF, DSCF, and ACF, individually. Our proposed KAL method obtains the best average value and achieves an average improvement by 29.0% with the EAF-measure indicator.

Third, in terms of Table 5, our proposed KAL method obtains better performance on 13 out of 14 apps compared with the five baseline methods. The average P_{opt} value by KAL over all apps achieves improvements by 22.4%, 22.2%, 18.7%, 20.3%, and 23.3% compared with NONE, NNF, PF, DSCF, and ACF, individually. Our proposed KAL method obtains the best average value and achieves an average improvement by 21.4% with the P_{opt} indicator.

Fourth, in terms of Table 6, our proposed KAL method obtains better performance on 13 out of 14 apps compared with the five baseline methods. The average PCI value by KAL over all apps achieves improvements by 80.1%, 85.0%, 55.5%, 61.1%, and 69.3% compared with NONE, NNF, PF, DSCF, and ACF, individually. Our proposed KAL method obtains the best average value and achieves an average improvement by 70.2% with the PCI indicator.

Fifth, in terms of Fig. 2, our KAL method ranks first and belongs to the top rank group in terms of all four indicators. In addition, KAL shows the significant differences compared with the five baseline methods.

Different from the instance selection based methods that select the more representitive commit instances from the source app for model building, our KAL method use the KPCA technique to learn more representitive features from the original commit data, which can obtain high-quality feature representation.

Answer: Overall, our proposed KAL method outperforms the comparative instance selection based methods for JIT defect prediction on Andoid mobile apps in terms of all four effort-aware indicators.

5.2 Answer to RQ2: the prediction performance of our proposed KAL method and the transfer learning based methods

Methods: To answer this question, we choose seven transfer learning methods as baselines for comparison, including IFS_5 [40], IFS_16 [41], Transfer Component Analysis (TCA) [42], Conditional Distribution based Transfer learning (CDT) [17], Joint Distribution based Transfer learning (JDT) [43], Transfer Naive Bayes (TNB) [12], and Balanced Distribution Adaptation based transfer learning (BDA) [17]. Below, we briefly describe these baseline methods.

- IFS_5 converts the original commit instances into a new feature space, in which five distribution traits are used to characterize each commit instance, including median, mean, minimum, maximum, and variance values [40].
- IFS_16 is an advanced version of IFS_5, which characterizes each commit instances with 16 distribution traits: mode, median, mean, harmonic mean, minimum, maximum, range, variation ratio, first quartile, third quartile, interquartile range, variance, standard deviation, coefficient of variation, skewness, and kurtosis values [41].
- TCA only takes the margin distribution of commit instances into account for model building.
- CDT only takes the conditional disrtibution of commit instances into account for model building.
- JDT simultaneously takes both margin distribution and conditional distribution of commit instances into account and gives them the same weight.
- TNB gives the different weights for commit instances in the source app by using the data gravitation technique to evaluate the information derived from the target app.
- BDA simultaneously takes both margin distribution and conditional distribution of commit instances into account but the weights are adaptive.

Results: Tables 7–10 present the results of the indicator values and the corresponding standard deviations for our proposed KAL method and the seven transfer learning based methods in terms of EARecall, EAF-measure, P_{opt}, and PCI, individually. Figure 3 illustrates the corresponding statistical test results. From these tables and the figure, we can draw the following findings.

First, in terms of Table 7, our proposed KAL method obtains better performance on 13 out of 14 apps compared with the seven baseline methods. The average EARecall value by KAL over all apps achieves improvements by 64.0%, 56.2%, 63.0%, 65.9%, 66.9%, 54.4%, and 25.6% compared

Table 7 EARecall values of our KAL method and the transfer learning based methods

Project	IFS_5	IFS_16	TCA	CDT	JDT	TNB	BDA	KAL
Firewall	0.391	0.413	0.385	0.492	0.385	0.392	0.570	**0.924**
Alfresco	0.547	0.473	0.480	0.473	0.467	0.545	0.651	**0.802**
Sync	0.346	0.339	0.345	0.314	0.374	0.397	0.432	**0.445**
Wallpaper	0.399	0.417	0.460	0.410	0.463	0.543	**0.587**	0.566
Keyboard	0.470	0.677	0.378	0.415	0.396	0.567	0.579	**0.939**
Apg	0.613	0.540	0.512	0.471	0.499	0.460	0.638	**0.712**
Atmosphere	0.590	0.676	0.437	0.433	0.437	0.369	0.720	**0.839**
Secure	0.471	0.428	0.491	0.460	0.458	0.521	0.562	**0.724**
Facebook	0.328	0.332	0.372	0.349	0.371	0.396	0.444	**0.673**
Flutter	0.372	0.423	0.475	0.417	0.429	0.377	0.565	**0.777**
Kiwix	0.320	0.375	0.461	0.438	0.420	0.454	0.578	**0.717**
Cloud	0.360	0.397	0.460	0.471	0.427	0.455	0.549	**0.559**
Turner	0.369	0.367	0.357	0.371	0.355	0.450	0.409	**0.470**
Average	0.429	0.451	0.432	0.424	0.422	0.456	0.560	**0.704**

Table 8 EAF-measure values of our KAL method and the transfer learning based methods

Project	IFS_5	IFS_16	TCA	CDT	JDT	TNB	BDA	KAL
Firewall	0.361	0.367	0.364	0.447	0.366	0.411	0.512	**0.717**
Alfresco	0.417	0.373	0.383	0.387	0.366	0.502	0.475	**0.497**
Sync	0.320	0.309	0.301	0.289	0.322	**0.376**	0.371	0.352
Wallpaper	0.309	0.317	0.334	0.292	0.332	**0.431**	0.401	0.367
Keyboard	0.391	0.508	0.341	0.361	0.348	0.546	0.477	**0.646**
Apg	0.508	0.476	0.455	0.424	0.439	0.461	0.542	**0.550**
Atmosphere	0.507	0.565	0.405	0.393	0.399	0.391	0.592	**0.637**
Secure	0.407	0.376	0.434	0.411	0.409	0.513	0.485	**0.548**
Facebook	0.174	0.152	0.197	0.182	0.205	0.222	0.228	**0.243**
Flutter	0.238	0.265	0.320	0.284	0.317	0.306	0.368	**0.373**
Kiwix	0.297	0.321	0.371	0.355	0.348	0.441	0.442	**0.467**
Cloud	0.283	0.307	0.325	0.354	0.313	0.395	**0.399**	0.316
Turner	0.342	0.354	0.342	0.340	0.331	**0.458**	0.394	0.375
Average	0.350	0.361	0.352	0.348	0.346	0.420	0.438	**0.468**

Table 9 P_{opt} values of our KAL method and the transfer learning based methods

Project	IFS_5	IFS_16	TCA	CDT	JDT	TNB	BDA	KAL
Firewall	0.704	0.736	0.643	0.682	0.673	0.551	0.708	**0.947**
Alfresco	0.696	0.649	0.681	0.676	0.706	0.610	0.751	**0.863**
Sync	0.671	0.659	0.626	0.603	0.630	0.640	0.650	**0.731**
Wallpaper	0.686	0.715	0.681	0.673	0.683	0.656	**0.748**	0.720
Keyboard	0.750	0.791	0.666	0.683	0.649	0.632	0.714	**0.932**
Apg	0.757	0.727	0.744	0.728	0.703	0.515	0.781	**0.806**
Atmosphere	0.740	0.762	0.707	0.657	0.678	0.491	0.775	**0.852**
Secure	0.644	0.629	0.664	0.644	0.654	0.592	0.667	**0.788**
Facebook	0.641	0.639	0.624	0.615	0.629	0.504	0.693	**0.796**
Flutter	0.712	0.732	0.695	0.689	0.714	0.510	0.733	**0.851**
Kiwix	0.665	0.679	0.693	0.654	0.655	0.557	0.716	**0.886**
Cloud	0.596	0.663	0.643	0.640	0.641	0.600	0.720	**0.785**
Turner	0.606	0.627	0.602	0.609	0.612	0.620	0.616	**0.762**
Average	0.682	0.693	0.667	0.658	0.664	0.575	0.713	**0.825**

with IFS_5, IFS_16, TCA, CDT, JDT, TNB, and BDA, individually. Our proposed KAL method obtains the best average value and achieves an average improvement by 56.6% with the EARecall indicator.

Second, in terms of Table 8, our proposed KAL method obtains better performance on 10 out of 14 apps compared with the seven baseline methods. The average EAF-measure value by KAL over all apps achieves improvements by 33.7%, 29.6%, 33.0%, 34.5%, 35.3%, 11.4%, and 6.8% compared with IFS_5, IFS_16, TCA, CDT, JDT, TNB, and BDA, individually. Our proposed KAL method obtains the best average value and achieves an average improvement by 26.3% with the EAF-measure indicator.

Third, in terms of Table 9, our proposed KAL method obtains better performance on 13 out of 14 apps compared with the seven baseline methods. The average P_{opt} value by KAL over all apps achieves improvements by 21.0%, 19.0%, 23.7%, 25.4%, 24.2%, 43.5%, and 15.7% compared with IFS_5, IFS_16, TCA, CDT, JDT, TNB, and BDA, individually. Our proposed KAL method obtains the best

Table 10 PCI values of our KAL method and the transfer learning based methods

Project	IFS_5	IFS_16	TCA	CDT	JDT	TNB	BDA	KAL
Firewall	0.361	0.387	0.362	0.472	0.359	0.250	0.528	**0.943**
Alfresco	0.478	0.409	0.472	0.405	0.427	0.301	0.589	**0.841**
Sync	0.359	0.392	0.409	0.346	0.422	0.280	0.465	**0.611**
Wallpaper	0.394	0.424	0.500	0.460	0.515	0.408	0.611	**0.614**
Keyboard	0.423	0.650	0.355	0.390	0.354	0.342	0.534	**0.942**
Apg	0.614	0.516	0.489	0.452	0.468	0.302	0.621	**0.706**
Atmosphere	0.566	0.658	0.406	0.423	0.418	0.226	0.712	**0.844**
Secure	0.449	0.419	0.488	0.434	0.459	0.319	0.520	**0.753**
Facebook	0.320	0.311	0.368	0.357	0.367	0.290	0.402	**0.687**
Flutter	0.352	0.404	0.451	0.395	0.406	0.241	0.535	**0.811**
Kiwix	0.256	0.342	0.443	0.452	0.416	0.249	0.557	**0.825**
Cloud	0.337	0.359	0.466	0.413	0.431	0.257	0.457	**0.744**
Turner	0.346	0.289	0.303	0.359	0.341	0.228	0.296	**0.582**
Average	0.404	0.428	0.424	0.412	0.414	0.284	0.525	**0.762**

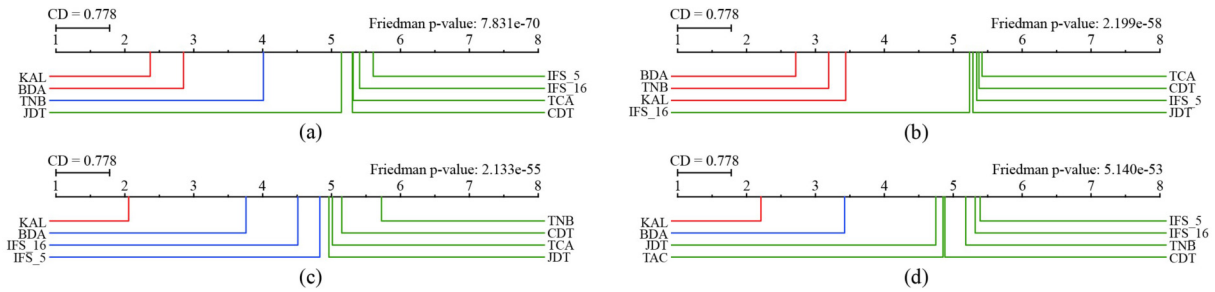

Fig. 3 Statistical test for our KAL method and the seven transfer learning based methods. (a) EARecall; (b) EAF-measure; (c) P_{opt}; (d) PCI

average value and achieves an average improvement by 24.6% with the P_{opt} indicator.

Fourth, in terms of Table 10, our proposed KAL method obtains better performance on all 14 apps compared with the seven baseline methods. The average PCI value by KAL over all apps achieves improvements by 88.6%, 78.0%, 79.7%, 85.0%, 84.1%, 168.3%, and 45.1% compared with IFS_5, IFS_16, TCA, CDT, JDT, TNB, and BDA, individually. Our proposed KAL method obtains the best average value and achieves an average improvement by 89.8% with the PCI indicator.

Fifth, in terms of Fig. 3, our KAL method ranks first and belongs to the top rank group in terms of three indicators except for EAF-measure. In addition, KAL shows the significant differences compared with the seven baseline methods.

Different from the transfer learning based methods that learn a shared feature space to narrow the data distribution difference, our KAL method employ the adversarial learning strategy that treats the learning process as a minimax two-player game to attain the common feature representation for each cross-project pair.

Answer: In summary, our proposed KAL method performs better than the comparative transfer learning based methods for JIT defect prediction on Andoid mobile apps in terms of all four effort-aware indicators.

5.3 Answer to RQ3: the prediction performance of our proposed KAL method and the classifier combination based methods

Methods: To answer this question, we choose four classifier combination methods as baselines for comparation, including COmbined DEfect Predictor (CODEP) [44], Max Diversity (MD) [45], Bagging_J48 (Bag_J48) [46], and Adaptive Selection of Classifiers in bug predIction (ASCI) [47]. Below, we briefly describe these baseline methods.

- CODEP first employs multiple classifiers to the app data and takes the probability results of classifiers as feature representation of commit instances for model building.
- MD trains multiple classifiers with multiple different parameters tuning and the diversity is calculated on each classifier pair. Then, the classifier pair that has the max dirversity is chosen as basic classifier to build the model.
- Bag_J48 first trains the Bagging and J48 models, and then combines these two models to improve the generalization.
- ASCI dynamically selects a best classifier from mutiple basic classifiers, which has better prediction results.

Results: Tables 11–14 present the results of the indicator values and the corresponding standard deviations for our proposed KAL method and the four classifier combination based methods in terms of EARecall, EAF-measure, P_{opt}, and PCI, individually. Figure 4 illustrates the corresponding statistical test results. From these tables and the figure, we can draw the following findings.

First, in terms of Table 11, our proposed KAL method obtains better performance on 13 out of 14 apps compared with the four baseline methods. The average EARecall value by KAL over all apps achieves improvements by 80.5%,

Table 11 EARecall values of our KAL method and the classifier combination based methods

Project	CODEP	MD	Bag_J48	ASCI	KAL
Firewall	0.370	0.364	0.345	0.384	**0.924**
Alfresco	0.479	0.471	0.536	0.519	**0.802**
Sync	0.280	0.337	0.313	0.300	**0.445**
Wallpaper	0.315	0.390	0.368	0.262	**0.566**
Keyboard	0.459	0.466	0.447	0.393	**0.939**
Apg	0.478	0.505	**0.771**	0.547	0.712
Atmosphere	0.565	0.510	0.505	0.448	**0.839**
Secure	0.309	0.483	0.450	0.370	**0.724**
Facebook	0.330	0.347	0.372	0.313	**0.673**
Flutter	0.390	0.350	0.425	0.299	**0.777**
Kiwix	0.417	0.369	0.426	0.350	**0.717**
Cloud	0.361	0.373	0.406	0.331	**0.559**
Turner	0.318	0.387	0.415	0.391	**0.470**
Average	0.390	0.412	0.444	0.377	**0.704**

Table 12 EAF-measure values of our KAL method and the classifier combination based methods

Project	CODEP	MD	Bag_J48	ASCI	KAL
Firewall	0.346	0.362	0.324	0.348	**0.717**
Alfresco	0.373	0.379	0.404	0.386	**0.497**
Sync	0.274	0.315	0.295	0.289	**0.352**
Wallpaper	0.259	0.312	0.280	0.211	**0.367**
Keyboard	0.376	0.424	0.383	0.330	**0.646**
Apg	0.397	0.448	**0.606**	0.461	0.550
Atmosphere	0.464	0.466	0.438	0.394	**0.637**
Secure	0.303	0.434	0.400	0.341	**0.548**
Facebook	0.202	0.186	0.185	0.158	**0.243**
Flutter	0.242	0.247	0.275	0.209	**0.373**
Kiwix	0.358	0.330	0.365	0.309	**0.467**
Cloud	**0.318**	0.295	0.310	0.266	0.316
Turner	0.326	0.364	0.374	0.358	**0.375**
Average	0.326	0.351	0.357	0.312	**0.468**

Table 13 P_{opt} values of our KAL method and the classifier combination based methods

Project	CODEP	MD	Bag_J48	ASCI	KAL
Firewall	0.656	0.673	0.614	0.676	**0.947**
Alfresco	0.681	0.684	0.705	0.677	**0.863**
Sync	0.598	0.631	0.631	0.605	**0.731**
Wallpaper	0.535	0.660	0.586	0.598	**0.720**
Keyboard	0.597	0.675	0.714	0.640	**0.932**
Apg	0.700	0.787	**0.838**	0.704	0.806
Atmosphere	0.742	0.733	0.691	0.683	**0.852**
Secure	0.611	0.727	0.537	0.579	**0.788**
Facebook	0.607	0.647	0.674	0.636	**0.796**
Flutter	0.640	0.656	0.661	0.640	**0.851**
Kiwix	0.647	0.668	0.669	0.666	**0.886**
Cloud	0.585	0.620	0.630	0.586	**0.785**
Turner	0.522	0.635	0.677	0.667	**0.762**
Average	0.625	0.677	0.664	0.643	**0.825**

Table 14 PCI values of our KAL method and the classifier combination based methods

Project	CODEP	MD	Bag_J48	ASCI	KAL
Firewall	0.341	0.315	0.317	0.337	**0.943**
Alfresco	0.393	0.447	0.484	0.450	**0.841**
Sync	0.284	0.328	0.321	0.271	**0.611**
Wallpaper	0.282	0.396	0.369	0.280	**0.614**
Keyboard	0.426	0.394	0.378	0.352	**0.942**
Apg	0.476	0.503	**0.774**	0.534	0.706
Atmosphere	0.553	0.490	0.470	0.427	**0.844**
Secure	0.243	0.450	0.404	0.355	**0.753**
Facebook	0.306	0.326	0.363	0.308	**0.687**
Flutter	0.322	0.352	0.398	0.234	**0.811**
Kiwix	0.333	0.334	0.348	0.293	**0.825**
Cloud	0.257	0.337	0.395	0.295	**0.744**
Turner	0.189	0.311	0.359	0.328	**0.582**
Average	0.339	0.383	0.414	0.343	**0.762**

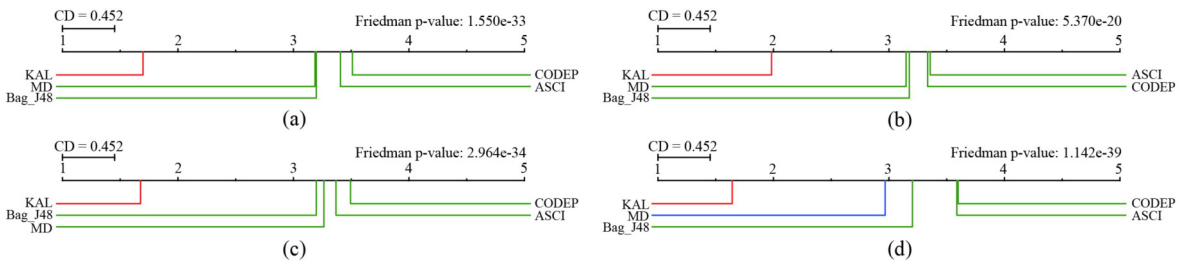

Fig. 4 Statistical test for our KAL method and the four classifier combination based methods. (a) EARecall; (b) EAF-measure; (c) P_{opt}; (d) PCI

70.9%, 58.6%, and 86.7% compared with CODEP, MD, Bag_J48 and ASCI, individually. Our proposed KAL method obtains the best average value and achieves an average improvement by 74.2% with the EARecall indicator.

Second, in terms of Table 12, our proposed KAL method obtains better performance on 12 out of 14 apps compared with the four baseline methods. The average EAF-measure value by KAL over all apps achieves improvements by 43.7%, 33.4%, 31.2%, and 50.0% compared with CODEP, MD, Bag_J48 and ASCI, individually. Our proposed KAL method obtains the best average value and achieves an average improvement by 39.6% with the EAF-measure indicator.

Third, in terms of Table 13, our proposed KAL method obtains better performance on 13 out of 14 apps compared with the four baseline methods. The average P_{opt} value by KAL over all apps achieves improvements by 32.0%, 21.9%, 24.2%, and 28.3% compared with CODEP, MD, Bag_J48 and ASCI, individually. Our proposed KAL method obtains the best average value and achieves an average improvement by 26.6% with the P_{opt} indicator.

Fourth, in terms of Table 14, our proposed KAL method obtains better performance on 13 out of 14 apps compared with the four baseline methods. The average PCI value by KAL over all apps achieves improvements by 124.8%, 99.0%, 84.1%, and 122.2% compared with CODEP, MD, Bag_J48 and ASCI, individually. Our proposed KAL method obtains the best average value and achieves an average improvement by 107.5% with the PCI indicator.

Fifth, in terms of Fig. 4, our KAL method ranks first and belongs to the top rank group in terms of all four indicators. In addition, KAL shows the significant differences compared with the four baseline methods.

Different from the classifier combination based methods that utilize the ensemble strategy to enhance the prediction performance, our KAL method combines the KPCA technique and AL strategy for both high-quality feature representation and common feature embedding to improve the performance of cross-project JIT defect prediction task.

Answer: In a word, our proposed KAL method is more effective to obtain significantly better JIT defect prediction performance on Android mobile apps than the comparative classifier combination based methods.

5.4 Answer to RQ4: the prediction performance of our proposed KAL method, its variants, and the state-of-the-art method

Methods: To answer this question, we combine the used techniques in different ways and design three variants (i.e., ALRF, KRF, and KNB) as baseline methods. In addition, we also treat the state-of-the-art (SOTA) method [8] for comparison. Below, we briefly describe these baseline methods.

- ALRF only uses the original feature for common feature extraction and then employs the RF model for classification.
- KRF only uses the KPCA method to obtain the representitive features and then employs the RF model for classification.
- KNB employs the Naive Bayes classfier to build the classification model after extracting the common feature embedding.
- SOTA is the state-of-the-art method proposed by [8] that only uses the Naive Bayes model without feature learning process for cross-project JIT defect prediction task.

Results: Tables 15–18 present the results of the indicator values and the corresponding standard deviations for our proposed KAL method, its variants, and the SOTA method in terms of EARecall, EAF-measure, P_{opt}, and PCI, individually. Figure 5 illustrates the corresponding statistical test results. From these tables and the figure, we can draw the following findings.

Table 15 EARecall values of our KAL method, its variants, and the SOTA method

Project	ALRF	KRF	KNB	SOTA	KAL
Firewall	0.576	0.776	0.751	0.228	**0.924**
Alfresco	0.680	0.782	0.728	0.321	**0.802**
Sync	0.390	0.391	0.393	0.272	**0.445**
Wallpaper	0.548	0.480	**0.675**	0.273	0.566
Keyboard	0.703	0.892	0.781	0.275	**0.939**
Apg	0.581	**0.877**	0.651	0.238	0.712
Atmosphere	0.733	0.735	0.774	0.384	**0.839**
Secure	0.555	0.669	0.653	0.205	**0.724**
Facebook	0.528	0.660	0.550	0.233	**0.673**
Flutter	0.481	0.664	0.750	0.206	**0.777**
Kiwix	0.578	0.663	0.597	0.235	**0.717**
Cloud	0.455	0.540	0.514	0.310	**0.559**
Turner	0.431	0.422	0.365	0.251	**0.470**
Average	0.557	0.658	0.629	0.264	**0.704**

Table 16 EAF-measure values of our KAL method, its variants, and the SOTA method

Project	ALRF	KRF	KNB	SOTA	KAL
Firewall	0.478	0.622	0.618	0.215	**0.717**
Alfresco	0.430	0.486	0.456	0.251	**0.497**
Sync	0.324	0.319	0.326	0.247	**0.352**
Wallpaper	0.356	0.316	**0.442**	0.209	0.367
Keyboard	0.525	0.627	0.546	0.218	**0.646**
Apg	0.445	**0.659**	0.533	0.187	0.550
Atmosphere	0.582	0.577	0.611	0.320	**0.637**
Secure	0.430	0.510	0.508	0.171	**0.548**
Facebook	0.213	0.241	0.211	0.168	**0.243**
Flutter	0.246	0.339	0.366	0.091	**0.373**
Kiwix	0.377	0.439	0.421	0.175	**0.467**
Cloud	0.282	0.314	**0.323**	0.267	0.316
Turner	0.342	0.338	0.349	0.231	**0.375**
Average	0.387	0.445	0.439	0.211	**0.468**

Table 17 P_{opt} values of our KAL method, its variants, and the SOTA method

Project	ALRF	KRF	KNB	SOTA	KAL
Firewall	0.793	0.812	0.820	0.537	**0.947**
Alfresco	0.795	0.850	0.797	0.546	**0.863**
Sync	0.714	0.716	0.641	0.598	**0.731**
Wallpaper	0.800	0.689	**0.809**	0.543	0.720
Keyboard	0.820	0.883	0.798	0.450	**0.932**
Apg	0.740	0.887	0.744	0.546	0.806
Atmosphere	0.831	0.782	0.785	0.557	**0.852**
Secure	0.712	0.736	0.760	0.452	**0.788**
Facebook	0.740	**0.822**	0.741	0.577	0.796
Flutter	0.760	0.754	0.818	0.521	**0.851**
Kiwix	0.760	0.787	0.764	0.536	**0.886**
Cloud	0.713	0.758	0.694	0.518	**0.785**
Turner	**0.779**	0.744	0.654	0.513	0.762
Average	0.766	0.786	0.756	0.530	**0.825**

Table 18 PCI values of our KAL method, its variants, and the SOTA method

Project	ALRF	KRF	KNB	SOTA	KAL
Firewall	0.573	0.780	0.743	0.189	**0.943**
Alfresco	0.719	0.823	0.788	0.294	**0.841**
Sync	0.523	0.549	0.476	0.273	**0.611**
Wallpaper	0.599	0.529	**0.699**	0.268	0.614
Keyboard	0.698	0.881	0.785	0.252	**0.942**
Apg	0.606	**0.899**	0.631	0.245	0.706
Atmosphere	0.714	0.739	0.772	0.375	**0.844**
Secure	0.569	0.680	0.689	0.224	**0.753**
Facebook	0.551	**0.694**	0.554	0.235	0.687
Flutter	0.506	0.690	0.780	0.203	**0.811**
Kiwix	0.663	0.750	0.675	0.261	**0.825**
Cloud	0.551	0.680	0.595	0.245	**0.744**
Turner	0.561	0.569	0.299	0.239	**0.582**
Average	0.602	0.713	0.653	0.254	**0.762**

First, in terms of Table 15, our proposed KAL method obtains better performance on 12 out of 14 apps compared with the four baseline methods. The average EARecall value by KAL over all apps achieves improvements by 26.4%,

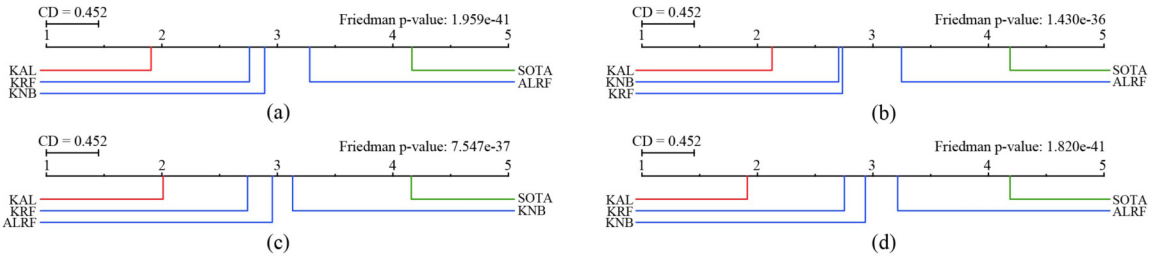

Fig. 5 Statistical test for our KAL method, its three variants, and the SOTA method. (a) EARecall; (b) EAF-measure; (c) P_{opt}; (d) PCI

7.0%, 11.8%, and 166.9% compared with ALRF, KRF, KNB, and SOTA, individually. Our proposed KAL method obtains the best average value and achieves an average improvement by 53.0% with the EARecall indicator.

Second, in terms of Table 16, our proposed KAL method obtains better performance on 11 out of 14 apps compared with the four baseline methods. The average EAF-measure value by KAL over all apps achieves improvements by 21.1%, 5.2%, 6.6%, and 121.6% compared with ALRF, KRF, KNB, and SOTA, individually. Our proposed KAL method obtains the best average value and achieves an average improvement by 38.6% with the EAF-measure indicator.

Third, in terms of Table 17, our proposed KAL method obtains better performance on 10 out of 14 apps compared with the four baseline methods. The average P_{opt} value by KAL over all apps achieves improvements by 7.7%, 4.9%, 9.1%, and 55.4% compared with ALRF, KRF, KNB, and SOTA, individually. Our proposed KAL method obtains the best average value and achieves an average improvement by 19.3% with the P_{opt} indicator.

Fourth, in terms of Table 18, our proposed KAL method obtains better performance on 11 out of 14 apps compared with the four baseline methods. The average PCI value by KAL over all apps achieves improvements by 26.4%, 6.9%, 16.7%, and 199.9% compared with ALRF, KRF, KNB, and SOTA, individually. Our proposed KAL method obtains the best average value and achieves an average improvement by 62.5% with the PCI indicator.

Fifth, in terms of Fig. 5, our KAL method ranks first and belongs to the top rank group in terms of all four indicators. In addition, KAL shows the significant differences compared with the four baseline methods.

We analyze the reasons of why our KAL method is superior to its three variants from the following three parts. (1) *in terms of the KPCA technique*: as the KPCA technique has the potential to obtain high quality features by transforming the original features into a high-dimensional feature space, KAL can attain more representative features that can represent the commit instances better compared with only using the raw data features; (2) *in terms of the AL technique*: as the AL technique can narrow the discrepancy between each cross-project pair, we can obtain the common feature representation that learns the inherent characteristics from both the source app and the target app. This common feature representation can make the model more robust; (3) *in terms of the classification model*: we employ the RF model as the basic classifier. Compared with the Naive Bayes assuming the conditional independence between features, RF has more powerful ability to adapt to features of the commit instances.

Answer: In a word, our proposed KAL method is more effective to obtain significantly better JIT defect prediction performance on Android mobile apps than its three variants and the state-of-the-art method.

6 Threats to validity

Threats to construct validity mainly lie in the used performance indicators and the statistical test method. In this work, we use four effort-aware indicators including EARecall, EAF-measure, P_{opt}, and PCI, to assess the performance of our proposed KAL method. In addition, we use the Friedman test with the Nemenyi post-hoc test to conduct the significance analysis. Threats to internal validity focus on the possible faults during the implementation of our KAL method and other baseline methods. In this work, we rely on the third-part libraries, including scikit-learn and PyTorch, to implement our KAL method. As for the comparative methods, we carefully incorporate the source code provided by authors into our experiment. Threats to external validity concentrate on the generalization of our experimental results. Due to the complexity of adversarial learning, we empirically specify the parameter settings used in our KAL method. However, whether the better parameter combinations exist will be explored in the future. In this work, we conduct experiments on 14 open-source mobile apps that are from distinct domains and with different sizes. Since this work mainly focus on the JIT defect prediction on Android mobile apps, whether our KAL method is suitable for the traditional software will be explored in the future.

7 Conclusion

In this work, we propose a novel method, called KAL, for cross-project JIT defect prediction task toward mobile apps. More specifically, KAL first applies the kernel-based principal component analysis technique to transform the original features into a high-dimensional feature space to obtain the representative features. Then, the adversarial learning strategy is employed to narrow the difference in each cross-project pair and extract the common feature embedding of the commit intances of each cross-project pair. We evaluate the performance of our proposed KAL method on 14 Andorid mobile apps with four effort-aware indicators. The experimental results on 182 cross-project pairs demonstrate that KAL achieves better prediction performance than five instance selection based methods, seven transfer learning based methods, four classifier combination based methods, its three variants, and the state-of-the-art method for JIT defect

prediction.

In the future, we plan to verify the generalization of our proposed KAL method on traditional software or on more defect data of mobile apps collected from other platform, such as IOS.

Acknowledgements This work was supported by the National Natural Science Foundation of China (Grant No. 62072060).

References

1. Ghotra B, McIntosh S, Hassan A E. Revisiting the impact of classification techniques on the performance of defect prediction models. In: Proceedings of the 37th IEEE International Conference on Software Engineering. 2015, 789–800
2. Xu Z, Li S, Xu J, Luo X, Zhang T, Keung J, Tang Y. LDFR: learning deep feature representation for software defect prediction. Journal of Systems and Software, 2019, 158: 110402
3. Xu Z, Xuan J, Liu J, Cui X. MICHAC: defect prediction via feature selection based on maximal information coefficient with hierarchical agglomerative clustering. In: Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering. 2016, 370–381
4. Chen X, Mu Y, Qu Y, Ni C, Liu M, He T, Liu S. Do different crossproject defect prediction methods identify the same defective modules? Journal of Software: Evolution and Process, 2020, 32(5): e2234
5. Menzies T, Greenwald J, Frank A. Data mining static code attributes to learn defect predictors. IEEE Transactions on Software Engineering, 2006, 33(1): 2–13
6. Kamei Y, Shihab E, Adams B, Hassan, A E, Mockus A, Sinha A, Ubayashi N. A large-scale empirical study of just-in-time quality assurance. IEEE Transactions on Software Engineering, 2012, 39(6): 757–773
7. Kamei Y, Fukushima T, McIntosh S, Yamashita K, Ubayashi N, Hassan A E. Studying just-in-time defect prediction using cross-project models. Empirical Software Engineering, 2016, 21(5): 2072–2106
8. 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. 2019, 99–110
9. Jing X Y, Ying S, Zhang Z W, Wu S S, Liu J. Dictionary learning based software defect prediction. In: Proceedings of the 36th International Conference on Software Engineering. 2014, 414–423
10. Xia X, Lo D, Pan S J, Nagappan N, Wang X. Hydra: massively compositional model for cross-project defect prediction. IEEE Transactions on Software Engineering, 2016, 42(10): 977–998
11. 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. 2007, 215–224
12. Ma Y, Luo G, Zeng X, Chen A. Transfer learning for cross-company software defect prediction. Information and Software Technology, 2012, 54(3): 248–256
13. Nam J, Pan S J, Kim S. Transfer defect learning. In: Proceedings of the 35th International Conference on Software Engineering. 2013, 382−391
14. Chen L, Fang B, Shang Z, Tang Y. Negative samples reduction in crosscompany software defects prediction. Information and Software Technology, 2015, 62: 67–77
15. Ryu D, Jang J I, Baik J. A transfer cost-sensitive boosting approach for cross-project defect prediction. Software Quality Journal, 2017, 25(1): 235–272
16. Liu C, Yang D, Xia X, Yan M, Zhang X. A two-phase transfer learning model for cross-project defect prediction. Information and Software Technology, 2019, 107: 125–136
17. Xu Z, Pang S, Zhang T, Luo X P, Liu J, Tang Y T, Xue L. Cross project defect prediction via balanced distribution adaptation based transfer learning. Journal of Computer Science and Technology, 2019, 34(5): 1039–1062
18. McIntosh S, Kamei Y. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. IEEE Transactions on Software Engineering, 2017, 44(5): 412–428
19. Pascarella L, Palomba F, Bacchelli A. Fine-grained just-in-time defect prediction. Journal of Systems and Software, 2019, 150: 22–36
20. Chen X, Zhao Y, Wang Q, Yuan Z. MULTI: multi-objective effortaware just-in-time software defect prediction. Information and Software Technology, 2018, 93: 1–13
21. Cabral G G, Minku L L, Shihab E, Mujahid S. 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. 2019, 666–676
22. Li S Z, Fu Q, Gu L, Scholkopf B, Cheng Y, Zhang H. Kernel machine based learning for multi-view face detection and pose estimation. In: Proceedings of the 8th IEEE International Conference on Computer Vision. 2001, 674–679
23. Xu Z, Liu J, Luo X, Zhang T. Cross-version defect prediction via hybrid active learning with kernel principal component analysis. In: Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering. 2018, 209–220
24. Huang J, Yan X. Relevant and independent multi-block approach for plant-wide process and quality-related monitoring based on KPCA and SVDD. ISA Transactions, 2018, 73: 257–267
25. Xu Z, Liu J, Luo X, Yang Z, Zhang Y, Yuan P, Zhang T. Software defect prediction based on kernel PCA and weighted extreme learning machine. Information and Software Technology, 2019, 106: 182–200
26. Goodfellow I, Pouget-Abadie J, Mirza M, Xu B, Warde-Farley D, Ozair S, Bengio Y. Generative adversarial nets. In: Proceedings of the 27th International Conference on Neural Information Processing systems, 2014, 2672−2680
27. Li W, Ding W, Sadasivam R, Cui X, Chen P. His-GAN: a histogrambased GAN model to improve data generation quality. Neural Networks, 2019, 119: 31–45
28. Xu Z, Li S, Luo X, Liu J, Zhang T, Tang Y, Xu J, Yuan P, Keung, J. TSTSS: a two-stage training subset selection framework for cross version defect prediction. Journal of Systems and Software, 2019, 154: 59–78
29. Arisholm E, Briand L C, Johannessen E B. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. Journal of Systems and Software, 2010, 83(1): 2–17
30. Xu Z, Li L, Yan M, Liu J, Luo X, Grundy J, Zhang Y, Zhang X. A comprehensive comparative study of clustering-based unsupervised defect prediction models. Journal of Systems and Software, 2021, 172: 110862
31. Huang Q, Xia X, Lo D. Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction. Empirical Software Engineering, 2019, 24(5): 2823–2862
32. Breiman L. Random forests. Machine Learning, 2001, 45(1): 5–32
33. Tantithamthavorn C, Hassan A E, Matsumoto K. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. IEEE Transactions on Software Engineering, 2018,

46(11): 1200–1219
34. Yang X, Lo D, Xia X, Sun J. TLEL: a two-layer ensemble learning approach for just-in-time defect prediction. Information and Software Technology, 2017, 87: 206–220
35. Demšar J. Statistical comparisons of classifiers over multiple data sets. The Journal of Machine Learning Research, 2006, 7: 1–30
36. Turhan B, Menzies T, Bener A B, Di Stefano J. On the relative value of cross-company and within-company data for defect prediction. Empirical Software Engineering, 2009, 14(5): 540–578
37. Peters F, Menzies T, Marcus A. Better cross company defect predictio. In: Proceedings of the 10th Working Conference on Mining Software Repositories. 2013, 409–418
38. Kawata K, Amasaki S, Yokogawa T. Improving relevancy filter methods for cross-project defect prediction. In: Proceedings of the 3rd International Conference on Applied Computing and Information Technology/2nd International Conference on Computational Science and Intelligence. 2015, 2–7
39. Yu X, Zhou P, Zhang J, Liu J. A data filtering method based on agglomerative clustering. In: Proceedings of the 29th International Conference on Software Engineering and Knowledge Engineering. 2017, 392–397
40. He P, Li B, Ma Y. Towards cross-project defect prediction with imbalanced feature sets. 2014, arXiv preprint arXiv: 1411.4228
41. He Z, Shu F, Yang Y, Li M, Wang Q. An investigation on the feasibility of cross-project defect prediction. Automated Software Engineering, 2012, 19(2): 167–199
42. Pan S J, Tsang I W, Kwok J T, Yang Q. Domain adaptation via transfer component analysis. IEEE Transactions on Neural Networks, 2010, 22(2): 199–210
43. Long M, Wang J, Ding G, Sun J, Yu P S. Transfer feature learning with joint distribution adaptation. In: Proceedings of the IEEE International Conference on Computer Vision. 2013, 2200−2207
44. Panichella A, Oliveto R, De Lucia A. Cross-project defect prediction models: L'union fait la force. In: Proceedings of the 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering. 2014, 164–173
45. Petrić J, Bowes D, Hall T, Christianson B, Baddoo N. Building an ensemble for software defect prediction based on diversity selection. In: Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. 2016, 1–10
46. Zhang Y, Lo D, Xia X, Sun J. An empirical study of classifier combination for cross-project defect prediction. In: Proceedings of the 39th IEEE Annual Computer Software and Applications Conference. 2015, 264–269
47. Di Nucci D, Palomba F, De Lucia A. Evaluating the adaptive selection of classifiers for cross-project bug prediction. In: Proceedings of the 6th IEEE/ACM International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering. 2018, 48–54

Tian Cheng is a PhD student at the School of Data and Software Engineering, Chongqing University, China. He received the BS and MS degree in Macao Polytechnic Institute and Chongqing University, China in 2012 and 2014, respectively. His research interest includes Web data extraction, data mining, and software engineering.

Kunsong Zhao is a master student at the School of Computer Science, Wuhan University, China. He received the BS degree at School of Computer Science and Information Engineering, Hubei University, China. His research interest includes software engineering, deep learning, and natural language processing.

Song Sun received the BS and MS degrees in software engineering from Chongqing University, China in 2011 and 2014, respectively. He is currently pursuing the PhD degree with the School of Big Data and Software Engineering, Chongqing University, China. His research interest includes recommendation system, computer vision, and machine learning.

Muhammad Mateen received his master's degree in computer science from Air University, Islamabad, Pakistan in 2015 and PhD in Software Engineering from Chongqing University, China in 2020. Currently, he is working as an Assistant Professor at Air University Multan Campus, Pakistan. His research interest includes software engineering, image processing, and deep learning. He is a member of China Computer Federation (CCF).

Junhao Wen received the PhD degree from the Chongqing University, China in 2008. He is a vice head and professor of the School of Big Data and Software Engineering, Chongqing University, China. His research interest includes service computing, cloud computing, and software dependable engineering. He has published more than 80 refereed journal and conference papers in these areas. He has more than 30 research and industrial grants and developed many commercial systems and software tools.