Effort-Aware Just-in-Time Bug Prediction for Mobile Apps Via Cross-Triplet Deep Feature Embedding

Zhou Xu, Kunsong Zhao, Tao Zhang, Chunlei Fu, Meng Yan, Zhiwen Xie, Xiaohong Zhang, and Gemma Catolino

***Abstract*—Just-in-time (JIT) bug prediction is an effective quality assurance activity that identifies whether a code commit will introduce bugs into the mobile app, aiming to provide prompt feedback to practitioners for priority review. Since collecting sufficient labeled bug data is not always feasible for some mobile apps, one possible approach is to leverage cross-app models. In this work, we propose a new cross-triplet deep feature embedding method, called CDFE, for cross-app JIT bug prediction task. The CDFE method incorporates a state-of-the-art cross-triplet loss function into a deep neural network to learn high-level feature representation for the cross-app data. This loss function adapts to the cross-app feature learning task and aims to learn a new feature space to shorten the distance of commit instances with the same label and enlarge the distance of commit instances with different labels. In addition, this loss function assigns higher weights to losses caused by cross-app instance pairs than that by intra-app instance pairs, aiming to narrow the discrepancy of cross-app bug data. We evaluate our CDFE method on a benchmark bug dataset from 19 mobile apps with two effort-aware indicators. The experimental results on 342 cross-app pairs show that our proposed CDFE method performs better than 14 baseline methods.**

***Index Terms*—Cross-triplet feature embedding, effort-aware performance, just-in-time bug prediction, metric learning, mobile app bug prediction.**

Manuscript received June 19, 2020; revised September 13, 2020 and January 25, 2021; accepted February 28, 2021. This work was supported in part by the National Key Research and Development Project under Grant 2018YFB2101200, in part by the National Natural Science Foundation of China under Grant 62002034, in part by the Fundamental Research Funds for the Central Universities under Grants 2020CDCGRJ072 and 2020CDJQY-A021, in part by China Postdoctoral Science Foundation under Grant 2020M673137, in part by the Natural Science Foundation of Chongqing in China under Grant cstc2020jcyj-bshX0114, in part by the Science and Technology Development Fund of Macau under Grant 0047/2020/A1, in part by Faculty Research Grant Projects of MUST under Grant FRG-20-008-FI, and in part by the European Commission under Grant 825040 RADON. Associate Editor: Z. Jin. *(Corresponding authors: Chunlei Fu; Meng Yan.)*

Zhou Xu, Chunlei Fu, Meng Yan, and Xiaohong Zhang are with the Key Laboratory of Dependable Service Computing in Cyber Physical Society (Chongqing University), Ministry of Education, China, and School of Big Data and Software Engineering, Chongqing University, Chongqing 401331, China (e-mail: zhouxullx@cqu.edu.cn; clfu@cqu.edu.cn; mengy@cqu.edu.cn; xhongz@cqu.edu.cn).

Kunsong Zhao and Zhiwen Xie are with the School of Computer Science, Wuhan University, Wuhan 430072, China (e-mail: kszhao@whu.edu.cn; xiezhiwen@mails.ccnu.edu.cn).

Tao Zhang is with the Faculty of Information Technology, Macau University of Science and Technology, Macau 999078, China (e-mail: tazhang@must.edu.mo).

Gemma Catolino is with the Jheronimus Academy of Data Science, Tilburg University, Tilburg 90153, The Netherlands (e-mail: g.catolino@tilburguniversity.edu).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TR.2021.3066170.

Digital Object Identifier 10.1109/TR.2021.3066170

I. Introduction

MOBILE Internet has become an indispensable environment for information communication in human society. As it continues to evolve, smart devices (such as the smart phones and terminals) have been popularized rapidly. Mobile apps have largely facilitated the flourishment of smart devices. Thus, the quality of mobile apps has a direct impact on smart devices development. To satisfy new features or requirements, mobile apps are needed to continuously update. Due to a variety of uncontrollable factors, the process of fast iterative updates inevitably introduces bugs into the next release of the mobile apps [1]. It is a trending topic to early detect the bugs and recommend them to the app practitioners for fixing before releasing to the market, which is called bug prediction task. This process is expected to speed up program repair that can save massive manual efforts in software debugging [2].

Researchers have proposed many bug prediction methods for identifying the buggy code snippets (such as classes) in which the supervised bug prediction methods have been widely studied. The general process of the supervised bug prediction method usually contains two steps described as follows: first, a set of features and the bug labels are collected from the code and historical development data to form the labeled bug data. Then, a machine learning model is built on the labeled data to predict the bug labels of the unlabeled instances. Most studies investigated the instances at the file or class level. For mobile apps, they continuously release newer versions, instead of being constrained under a clearly defined road map [1]. The frequent updates usually involve a large number of code changes or commits, such as adding new code snippets, deleting old code snippets, and changing existing code snippets. Developers prefer a method that can alert them whether one code change will introduce bugs into the apps when they submit a code commit. This can speed up the bug solving process because the change details are still fresh in developers' mind what they have done at the moment. But this goal is beyond the scope of bug prediction at file or class level. For this purpose, Catolino [3] proposed just-in-time (JIT) bug prediction for mobile apps. JIT bug prediction uses features and labels of commit instances from the code change logs or comments to build a classification

model for identifying bug-introducing code commits, which can provide the developers with immediate feedback [4], [5]. JIT bug prediction is particularly well suited for software products, such as the mobile apps, that are characterized with frequent version updates involving a large amount of code commits. In this work, we focus on JIT bug prediction on mobile apps.

The typical intra-app bug prediction scenario needs sufficient labeled commit data for model training. However, new developed mobile apps are usually lack of historical development data to collect the label information, which hinders the application of intra-app bug prediction. One alternative solution is to utilize the labeled commit data of other mobile apps to assist the label identification of the mobile app at hand with the cross-app model.

As deep learning has presented the power feature learning strength for bug prediction task, in this work, we focus on proposing a deep learning-based metric learning method to learn the effective feature representation for cross-app bug prediction. A recent study [6] was the first to introduce the triplet loss-based deep feature embedding for the bug prediction task on traditional software projects and achieved promising performance. Triplet loss is used for metric learning aiming to learn a feature embedding space to make the instances with the same label as close as possible while the instances with different labels as far as possible. However, the triplet loss-based deep feature embedding method in [6] could only be applied to intra-app bug prediction scenario since it does not consider the cross-domain (an app denotes a domain in our work) discrepancy. Motivated by the work in [7], we propose a cross-triplet deep feature embedding (CDFE) method for our cross-app bug prediction task. Our CDFE method extends the original triplet loss to cross-app scenario by considering the domain information. More specifically, we apply a deep neural network (DNN) model with the improved cross-triplet loss to learn the high-level feature representation. The novel cross-triplet loss function not only simultaneously considers the intra-app and cross-app similarity during the process of feature representation learning, but also assigns higher weight to the cross-app triplet loss than the intra-app triplet loss, aiming to narrow the large discrepancy across app bug data.

As bug prediction is a binary classification task, previous studies mainly use some typical confusion matrix-based indicators, such as F-measure, to measure the performance of bug prediction on mobile apps. As app quality assurance activities have their own restrictions, such as limited testing resources, thus, general classification indicators in machine learning may not be well adapted to the bug prediction scenario. Arisholm *et al.* [8] proposed the effort-aware indicators designed for bug prediction on traditional software system. This type of indicators assumes that only limited test efforts are available for code inspection and is deemed as the appropriate performance measure for bug prediction in practical applications. In this work, we focus on the effort-aware performance of our proposed bug prediction model on mobile apps.

To evaluate the effectiveness of our CDFE method for cross-app bug prediction, we use a recently-released bug data that consists of 19 mobile apps as our benchmark dataset. We employ two widely used effort-aware indicators, i.e., Effort-Aware Recall (EARecall) and Effort-Aware F-measure (EAF-measure), to measure the performance of our CDFE method on a total of 342 (19×18) cross-app pairs. Our proposed CDFE method achieves an average EARecall value of 0.589 and an average EAF-measure value of 0.405 across these cross-app pairs. Compared with 14 baseline methods, including one within-app method, four instance selection methods, six transfer learning methods, and three classifier combination methods, the experimental results show that our CDFE method achieves the best performance in terms of both effort-aware indicators.

The contributions of our work are highlighted as follows.

1) To the best of our knowledge, we are the first to introduce the deep learning-based feature embedding into JIT bug prediction on mobile apps.
2) We propose a novel metric learning method, called CDFE, to learn high-level feature representation for cross-app bug data. CDFE adapts the tripet loss function to a cross-app scenario that takes the intra-app loss, cross-app loss, and their different important degrees into consideration.
3) We make the first attempt to apply the effort-aware indicators to evaluate the performance of our cross-app bug prediction method. The experimental results of 342 cross-app pairs from 19 mobile apps show the superiority of our proposed method.

II. Related Work

A. Just-in-Time Bug Prediction

To our best knowledge, Kamei *et al.* [9] were the first to propose the notion of JIT bug prediction for software quality assurance. They collected a benchmark dataset consisting of 11 defect data from six open source and five commercial projects. The experimental results on logistic regression classifier showed that they achieved an average recall of 0.64 and identified 35% of buggy code commits by inspecting 20% of efforts. Since then, many researches followed this work to conduct a series of studies. Fukushima *et al.* [10], [11] used a cross-project model to cope with the data scarcity issue for JIT bug prediction. They conducted experiments on 11 open-source projects and the results showed that the models that achieved better performance under the within-project scenario did not necessarily perform well under cross-project scenarios and the ensemble learning models tended to yield more accurate results. Yang *et al.* [12] found that simple feature ranking-based unsupervised models can outperform a series of supervised models under three validation settings for effort-aware JIT bug prediction on six open-source projects. Tourani and Adams [13] investigated the impact of issue and review discussions based features on JIT defect prediction. Their experiments on 15 open-source projects showed that these human discussions could complement the conventional change-based features and had a positive effect on the identification of buggy commits on half of projects. Tourani and Adams [13] investigated an unsupervised model based on a feature that was neglected in Yang *et al.*'s work [12]. They conducted experiments with the same benchmark dataset and validation settings as Yang *et al.*'s work [12], and found that this

neglected unsupervised model performed best compared with the methods in. Also, following Yang *et al.*'s work [12], Huang *et al.* [14], [15] proposed a simple supervised model called CBS and its improved version CBS+ for JIT defect prediction. The experiments on six open-source projects showed that the two supervised models achieved similar effort-aware Recall and better effort-aware Precision and effort-aware F-measure than the best unsupervised model in [12]. Chen *et al.* [16] proposed a multiobjective optimization based supervised method and Yang *et al.* [17] proposed a hybrid model combining decision tree and ensemble learning to improve performance of JIT bug prediction. In addition, some studies investigated the impacts of the context information [18] and SZZ-based labeling methods [19] on the performance of JIT bug prediction.

All these above studies focused on the JIT bug prediction on traditional standard software projects. Recently, two studies emerged to apply this topic on mobile apps. Catolino [3] conducted experiments on five mobile apps and suggested that future work was needed to improve the performance of JIT bug prediction on mobile apps. Later, they investigated the cross-app bug prediction using leave-one-out cross-validation [1]. Their experimental results on 14 mobile apps showed that ensemble methods performed the best. However, they did not consider the cross-app scenario based on the pair among two apps, which is a more commonly-used setting, but mixed the bug data of all apps except one as the training set. In addition, they just used standard classification model for prediction without transforming features by taking the distribution difference into considering. In view of the above shortcomings, in this work, we propose a deep feature embedding based metric learning to learn high-level feature representation which helps to alleviate the data distribution difference across apps, and conduct the cross-app experiment on each pair of apps.

B. Effort-Aware Bug Prediction

To our best knowledge, Arisholm *et al.* [8] were the first to introduce the concept of effort into a bug prediction model. They compared the performance of nine classifiers on a legacy Telecom software and pointed out that the effort-aware indicator (called cost-effectiveness in the original paper) was more appropriate than the usual confusion matrix based classification indicators. Mende *et al.* [20] compared a trivial model (called lines-of-code-based module order) with five classifiers. Their experimental results on 13 projects showed that the trivial model achieved the best AUC values but the worst effort-aware performance. Mende *et al.* [21] designed two strategies to incorporate effort into the prediction process. Their results on 15 projects showed that both strategies significantly improved the effort-aware performance of bug prediction models in a practical sense. They also found that the classifier achieving better values on confusion matrix-based indicators achieved relatively lower performance in terms of the effort-aware indicator. Some studies investigated the impacts of the lines of code (LOC) feature [22], slice-based cohesion features [23], the dependence clusters of program elements [24], the sampling-based balance strategies [25], dependency relationships-based network features [26], code churn-based features [27], and the learning to rank methods [28] on the performance of effort-aware bug prediction models. In addition, some studies proposed the deep ensemble learning method [29], neural network method [30], and sample-based semi-supervised method [31] to improve the effort-aware performance of bug prediction.

All the above studies investigated the effort-aware performance of bug prediction models on traditional software systems. However, no work has devoted to this topic on mobile apps. In this work, we bridge this research gap to explore the effort-aware cross-app bug prediction performance of our proposed deep feature embedding model.

C. Bug Prediction on Mobile Apps

To our best knowledge, Scandariato and Walden [32] were the first to study the mobile app bug prediction. They conducted experiments on five versions of a mobile app and used the SVM classifier to predict which classes of the app are buggy. Kaur *et al.* [33], [34] suggested that the bug prediction models based on process features could achieve better performance than that based on code complexity features for mobile apps. Malhotra [35] investigated the bug prediction performance of 18 classification models on seven mobile apps. The results showed that there existed significant performance differences among these models and the SVM-based models performed the worst. Ricky *et al.* [36] pointed out that SVM performed better than decision tree classifier for bug prediction on mobile apps. In addition, some studies used log files [37] and high-frequency keywords extracted from abstract syntax trees (AST) [38] to identify the buggy code snippets of mobile apps.

Considering that the above studies focused on the bug prediction on mobile apps at class level, Catolino *et al.* [1], [3] explored the JIT bug prediction at commit level which was more suitable for apps. We follow their work to conduct our research by proposing a cross-domain metric learning method for cross-app bug prediction task.

D. Deep Learning in Bug Prediction

As the successful application of deep learning with its powerful representation learning ability in other fields, some researchers utilized it to improve bug prediction performance. The present studies can be roughly divided into two categories: one is to use deep learning models to learn new feature representation from the handcrafted features of the bug data or as a classification model for bug prediction, and the other is to use deep learning models to extract representation of the software units from the source code instead of handcrafted features. For the former ones, Yang *et al.* [39] were the first to use deep brief network (DBN) model to integrate handcrafted features for JIT bug prediction. Albahli [29] proposed a deep ensemble learning method that combined DNN model, random forest, and XGBoost for JIT bug prediction. Manjula and Florence [40] proposed a hybrid method that used genetic algorithm-based feature optimization and DNN-based classification model for bug prediction. Qiao *et al.* [41] employed the DNN model to predict the number of

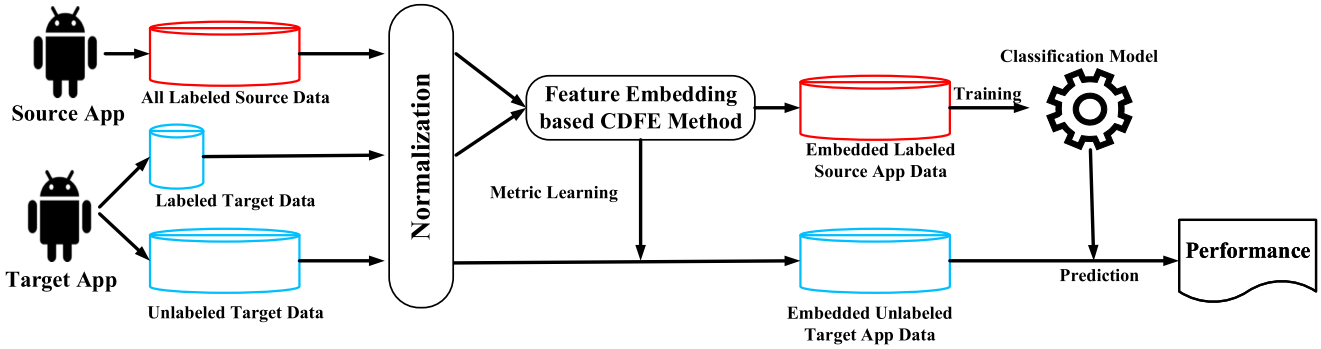

Fig. 1. Overview of our method.

bugs. Hasanpour *et al.* [42] compared the performance of two deep learning models, i.e., stack sparse auto-encoder and DBN model for bug prediction, and found that the former one achieved overall better performance. Xu *et al.* [6] proposed a hybrid loss (combining the triplet loss and cross-entropy loss) based DNN model for performance improvement of bug prediction.

For the later ones, Wang *et al.* [43], [44] were the first to use DBN model to extract semantic features of the program from its AST. Li *et al.* [45] used the convolutional neural network (CNN) model to learn semantic and structural features from AST. Dam *et al.* [46] used the long short-term memory (LSTM) network to extract both syntactic and structural information of the program from its AST. Phan *et al.* [47] utilized the directed graph-based CNN model to extract semantic features from the control flow graphs. Chen *et al.* [48] employed a bidirectional LSTM network to learn semantic features from AST. Fan *et al.* [49] applied the recurrent neural network model to learn semantic features from AST.

The later ones usually extract feature vectors for the instances with very high dimensions which will increase model training time, and the feature of each dimension has no specific meaning toward the code. We refer to the former ones to propose a new deep learning method to learn high-level feature representation from the handcrafted features. Different from the studies in the former ones that are applied to the traditional software projects and within-project scenario, our work focuses on the mobile apps and cross-app scenario.

III. Method

A. Overview

Fig. 1 presents an overview of our proposed method. As our method needs some labeled commit instances from the target app to assist the data of source app to learn the cross-app feature embedding, we randomly select a small number of commit instances as having labels for participation. Then, the data of the source app, the labeled data of the target app, and the unlabeled data of the target app are normalized with the z-score method. The data of the source app and the labeled data of the target app are input into our method to learn an optimal embedding mode. Then, the data of the source app and the unlabeled data of the target app are transfered into a new embedding space with this mode. At last, the embedding data of the source app are used to train a classification model which is applied to predict the labels of the unlabeled commit instances of the target app.

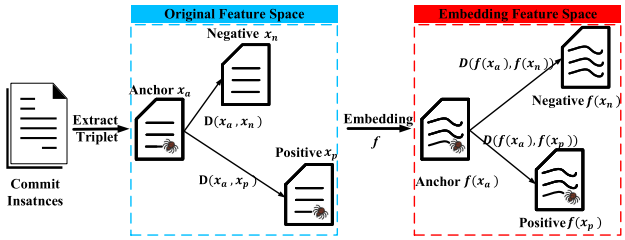

Fig. 2. An example of the learning process of the triplet loss function.

B. Triplet Loss

Triplet loss function is used to learn a better representation of the input features. Assume a triplet data as (x_a, x_p, x_n), where x_a denotes an anchor of the triplet, x_p denotes the positive instance that has the same label as x_a (called matched pairs), and x_n denotes the negative instance that has different labels with x_a (called unmatched pairs). The goal of the triplet loss function is to learn an embedding representation space in which the distance between the commit instances with the same class label is as close as possible while the commit instances with different class labels are as far as possible. The embedding process is demonstrated in Fig. 2.

For the above purpose, each commit instance is chosen as the anchor and the triplet loss function is formalized as follows:

$$\ell = \frac{3}{2\,m}\sum_{i}^{m/3}[D^2_{x_{(i)a,p}} - D^2_{x_{(i)a,n}} + m_d]^2_+ \qquad (1)$$

where $D_{x_{(i)a,p}} = ||f(x_{(i)a}) - f(x_{(i)p})||_2$ denotes the distance between matched pairs, $D_{a,n} = ||f(x_{(i)a}) - f(x_{(i)n})||_2$ denotes the distance between unmatched pairs, $f(\cdot)$ denotes the embedding function, and m_d is a margin parameter. The objective of the loss function is to make that $D_{x_{(i)a,n}}$ is larger than $D_{x_{(i)a,p}}$ plus m_d.

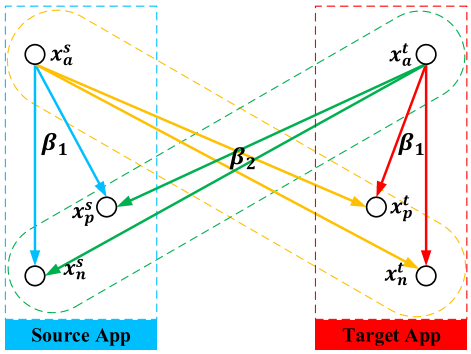

Fig. 3. An example of cross-triplet embedding.

C. Cross-Triplet Embedding Loss

Original triplet embedding method is mainly applied to a single domain; in other words, all the elements in the triplet (x_i^a, x_i^p, x_i^n) are from the same data. This limits its application for the feature representation learning tasks across different data. Jiang *et al.* [7], [50] made the first attempt to adapt the triplet loss function to cross-domain scenario. Their method was used to process image data from different domains with a deep convolutional neural network, called AlexNet model [51] which is not suitable for the commit instances of our bug data. Motivated by their work, we modify their method to adapt it to our bug prediction task. More specifically, the simplest way to consider all the triplet combinations whose three elements are randomly selected from either the source app or the target app. In this case, we have a total of eight triplet combinations, i.e., (x_a^s, x_p^s, x_n^s), (x_a^t, x_p^t, x_n^t), (x_a^s, x_p^t, x_n^t), (x_a^t, x_p^s, x_n^s), (x_a^s, x_p^t, x_n^s), (x_a^s, x_p^s, x_n^t), (x_a^t, x_p^s, x_n^t), and (x_a^t, x_p^t, x_n^s), where the superscript s (or t) denotes the corresponding commit instance comes from the source app (or target app). This simple way may cause some problems. For example, it treats the triplets from the same app and across apps the same, i.e., assigning them the equal weights when calculating the corresponding triplet loss function. However, the triplet loss for the triplets whose three elements are from cross-app data should be assigned higher weights than those whose three elements are from the same app data. The reason is that cross-app discrepancy is larger and should be treated more carefully [50]. In addition, for the last four triplets of the eight combinations, i.e., (x_a^s, x_p^t, x_n^s), (x_a^s, x_p^s, x_n^t), (x_a^t, x_p^s, x_n^t), and (x_a^t, x_p^t, x_n^s), whose positive instances and negative instances come from different apps, it is difficult to decide the margins m_d and the weights. For example, in terms of the matched pair (x_a^t, x_p^s) from cross apps and the unmatched pair (x_a^t, x_n^t) from intra app, we could not judge whether the loss of the former should be larger than the loss of the latter one.

To overcome the above issues, we employ an improved version of triplet embedding loss, called cross-triplet embedding loss, which is applied to the labeled data of source app and a small amount of labeled data of target app. The cross-triplet embedding loss only chooses the first four triplets of the combinations as shown in Fig. 3. The triplets in the blue and red rectangles [i.e., (x_a^s, x_p^s, x_n^s) and (x_a^t, x_p^t, x_n^t)] are from intra-app, while the triplets in the green and orange rounded rectangles [i.e., (x_a^s, x_p^t, x_n^t) and (x_a^t, x_p^s, x_n^s)] are from cross apps. In addition, the cross-triplet embedding loss calculates the total loss by assigning different weights to the losses from cross-app triplets and the losses from intra-app triplets, aiming to consider the domain information. The cross-triplet embedding loss function is formalized as follows:

$$\ell_{cross} = \underbrace{\beta_1(\ell^{s,s} + \ell^{t,t})}_{\text{intra app}} + \underbrace{\beta_2(\ell^{s,t} + \ell^{t,s})}_{\text{cross app}} \tag{2}$$

where $\ell^{s,s}$ and $\ell^{t,t}$ denote the intra-app loss of the triplets from source app (x_a^s, x_p^s, x_n^s) and target app (x_a^t, x_p^t, x_n^t), individually. $\ell^{s,t}$ and $\ell^{t,s}$ denote the cross-app loss of the triplets from (x_a^s, x_p^t, x_n^t) and (x_a^t, x_p^s, x_n^s), individually. β_1 and β_2 denote the weights for intra-app loss and cross-app loss, individually. For each triplet, the loss is calculated using (1). To highlight the cross-app triplet loss, we set $\beta_2 > \beta_1$. From the formula, we can see that the positive instance and the negative instance in a triplet must select from the same app, no matter the anchor instance comes from the source app or target app.

D. Deep Neural Network

In our work, we use the DNN with the cross-app triplet loss to learn the deep feature embedding. In general, the DNN contains three kinds of layers (e.g., the input layer, hidden layer, and output layer). The first layer receives the input feature vectors of commit instances, which is called input layer. The hidden layer transforms the feature vectors, aiming to learn the high-level feature representation. The last layer generates the prediction results (i.e., the commit instance label), which is called output layer. As our aim is mainly for feature representation learning using DNN without involving in the label prediction, in this work, we employ the DNN structure that only contains the first two types of layers. We use the fully connected strategy to construct the network structure, that is, the units among layers are fully connected while no connections exist between the units in the same layer.

The training process of DNN consists of two types of propagation algorithms, i.e., the forward propagation and the back propagation. The former one carries out a series of linear operations and activation operations with feature vectors of the triplet, the weight vector, and the bias vector. The latter one calculates the cross-triplet embedding loss to optimize the network parameters, aiming to learning a high-level feature representation for the cross-app data by considering the domain information. We apply adaptive moment estimation (Adam) algorithm [52] to optimize the model parameters during the training process. The code is available at https://figshare.com/search?q=10.6084/m9.figshare.13 635 347.

IV. Experimental Setup

A. Benchmark Dataset

In this work, we conduct experiment on a publicly available benchmark dataset recently released by a previous study [1]. This dataset consists of 19 Android mobile apps, including Android Firewall, Alfresco, Android Sync, Android Wallpaper,

TABLE I
BASIC STATISTIC INFORMATION OF THE 19 APPS

Project	Description	# LOC	# Total	# Buggy	# Clean	% Ratio
Firewall	A Linux iptables based firewall app	77,243	1,025	414	611	40.39%
Alfresco	A business office app for reviewing documents securely	152,047	1,004	214	790	21.31%
Sync	A synchronization app using USB for data transmission	275,637	209	62	147	29.67%
Wallpaper	An app with multiple high-quality wallpapers	35,917	588	94	494	15.99%
Keyboard	An app with multilanguage support for users	114,784	2,971	819	2,152	27.57%
Apg	A library for providing email encryption to Android platform	151,204	3,780	1,304	2,476	34.50%
Applozic	A library for integrating real time chat and messaging in mobile apps	87,662	946	143	803	15.12%
Atmosphere	An event driven app supporting WebSocket and HTTP	56,686	5,474	2,174	3,300	39.72%
Secure	An open standard based app for secure communication	98,768	2,579	853	1,726	33.07%
Delta	An instant messaging tools based on email	96,971	2,465	185	2,280	7.51%
Facebook	A bridge for Android devices and facebook	103,802	548	180	368	32.85%
Image	A library for managing and controlling the image loading and caching process	16,530	875	141	734	16.11%
Kiwix	An offline app for reading files of users	32,598	1,373	350	1,023	25.49%
Lottie	A library for parsing and rendering Adobe After Effects natively	57,291	235	36	199	15.32%
Scroll	A library for observing scroll events in mobile apps	27,836	250	54	196	21.60%
Cloud	An app with cloud storage for privacy protection	115,169	3,700	830	2,870	22.43%
Turner	A process synchronization based app for ebook reader	30,943	164	23	141	14.02%
Reddit	A receiver to obtain the information from user's Android wearable	9,506	222	60	162	27.03%
Telegram	A cloud-based free messaging app	4,158,369	289	145	144	50.17%

TABLE II
BRIEF DESCRIPTIONS OF THE 14 FEATURES

Scope	Name	Definition
Diffusion	NS	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
Size	LA	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
Purpose	FIX	Whether or not the current change is a bug fix
History	NDEV	Number of developers changing the files
	AGE	Average time interval since the last change
	NUC	Number of unique change to modified files
Experience	EXP	Developer experience
	REXP	Recent developer experience
	SEXP	Developer experience on a subsystem

AnySoft Keyboard, Apg, Applozic Android SDK, Atmosphere, Chat Secure Android, Delta Chat, Facebook Android SDK, Android Universal Image Loader, Kiwix, Lottie, Observable Scroll View, Own Cloud Android, Page Turner, Notify Reddit, and Telegram. The basic statistic information of these apps is presented in Table I, including the briefly functional description (Description), lines of code (# LOC), the total number of commit instances (# Total), the number of defective instances (# Buggy) and clean instances (# Clean), and the buggy ratio (% Ratio). From the table, we can observe that these apps are developed for various types of functions. This means that these apps belong to different domains, which improve the diversity of the mobile apps used. Meanwhile, the LOCs range from 9506 to 4 158 369, which means that these apps have different scales. Fourteen commonly used features from five families are collected to characterize the code commit instances and these features are used to conduct the JIT defect prediction on mobile apps. The brief descriptions of the 14 features are presented in Table II.

B. Performance Indicators

Typical classification indicators based on confusion matrix, such as Precision, Recall, and F-measure, hold that the test sources are sufficient for code reviews and treat the efforts for the test activities, such as inspecting different code snippets, as the same. However, the test resources are always limited in practical cases, and the inspection efforts for distinct snippets are also different. Therefore, typical classification indicators are not suitable to measure the bug prediction performance.

As some previous studies [12], [14]–[16], [27] focusing on tradition software recommended to evaluate the JIT bug prediction performance using effort-aware indicators which are more appropriate performance measurement for bug prediction in practical applications, we follow these studies to employ effort-aware indicators which take the inspection efforts into consideration for performance evaluation of our cross-app bug prediction method. Thus, we do not use the typical noneffort-aware performance indicators, such as accuracy, the area under the receiver operating characteristic curve (AUC), and Matthews correlation coefficient (MCC) in our work. Below, we briefly describe the calculation process of our used two effort-aware indicators. Note that the sum of the features LA and LD is treated as the proxy measure of efforts (i.e., cost) involved in inspecting a file, while the limited test resource is defined as 20% of all efforts [14], [15], [53].

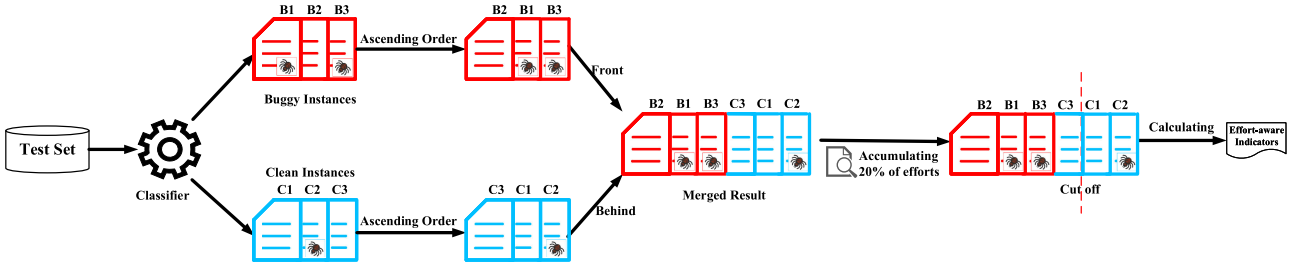

Fig. 4. Calculating process of the effort-aware indicators.

In this work, we follow previous studies [6], [54] to calculate the effort-aware indicators. More specifically, when the commit data of two mobile apps are embedding into a new feature space by using our proposed deep feature embedding method, a classifier is trained on the embedded source app and predicts the embedded target app as two groups, i.e., buggy and clean. Next, the commit instances in each group are sorted with ascending order based on their inspection efforts, respectively. Third, the sorted results of the two groups are merged in which the sorted results of the group with label buggy are placed in the front. Then, we imitate the practitioners to inspect these commit instances one by one from the highest to the lowest ranked instances according to the sorted results. The inspection process stops when the cumulative effort percentage of the inspected commit instances reaches 20%. Some statistics of the inspected commit instances are used to calculate the effort-aware indicators. More specifically, we first introduce three basic terms as follows.

1) t_d denotes the total number of commit instances that are buggy in the data of the target app.
2) t_n denotes the total number of checked commit instances, including both buggy and clean ones after inspecting 20% of efforts.
3) t_{nd} denotes the number of checked commit instances that are buggy after inspecting 20% of efforts.

The first effort-aware indicator that we used is called Effort-Aware Recall (EARecall). It denotes the proportion of detected buggy commit instances during the inspection process among all buggy commit instances in the target app. EARecall is formulated as follows:

$$\text{EARecall} = t_{nd}/t_d. \tag{3}$$

Effort-Aware Precision (EAPrecision) denotes the proportion of detected buggy commit instances during the inspection process among all checked instances, which is formulated as $\text{EAPrecision} = t_{nd}/t_n$.

The second effort-aware indicator that we used is called Effort-Aware F-measure (EAF-measure). Like the definition of typical F-measure in machine learning, EAF-measure is defined as the weighted harmonic average between EARecall and EA-Precision, which is formulated as follows:

$$\text{EAF-measure} = \frac{(1+\theta^2) \times \text{EAPrecision} \times \text{EARecall}}{\theta^2 \times \text{EAPrecision} + \text{EARecall}} \tag{4}$$

where θ is a trade-off parameter between EARecall and EAPrecision. In this work, we set θ as 2 following the same parameter setting in previous studies [6], [54], [55]. Fig. 4 gives an example to show the process of calculating the effort-aware indicators.

C. Data Partition

In this work, we combine all the commit instances from source app and 10% of the commit instances from target app as candidate set to train the cross-triplet embedding model. The 10% of the commit instances are selected with stratified sampling strategy. Compared with using random sampling to obtain the 10% of data, the used stratified sampling is more suitable since it can ensure that the bug ratio of the sampled data is the same as the target app data. To generate the training set and test set, we take all the commit instances from source app as the training set and the remainder (90%) of the commit instances from the target app as the test set to build our classification model. For a fair comparison, we also use 90% data of the target app as the test set for all baseline methods. In addition, we repeat this partition procedure 50 times to reduce the randomness of the bias and record the average values and the corresponding standard deviations on each indicator individually.

D. Parameter Settings

In this work, we set the DNN structure as two hidden layers with 16 units to build the cross-triplet embedding model. For the hyperparameters, we set the batch size as 32 and the iteration epochs as 30. In each iteration, the learning rate is set as 0.1 with the L_2 regularization to relieve the overfitting. In addition, for the cross-triplet embedding loss function, we set m_d as 0.5, β_1 as 1, and β_2 as 2 (i.e., $\beta_2/\beta_1 = 2$), following the same parameter setting in the previous work [7].

As our deep feature embedding method CDFE requires to combine the data from the source app and part of commit instances from the target app to learn the feature representation for the cross-app data, we use stratified strategy to randomly select 10% commit instances from the target app to participate the feature embedding learning. The stratified sampling denotes that we randomly select 10% buggy and 10% clean commit instances from the data of the target app, which avoids to select only the clean commit instances due to the class imbalance of bug data and ensures that the selected commit instances have the same bug ratio as the data of target app. After the feature embedding task being completed, the embedding data of source

app are used as the training set and the remaining 90% data of target app are used as the test set for the classification model. We repeat the selection operation of the 10% data of target app 50 times to reduce the random bias and report the average indicator values of the 50 rounds.

E. Classification Model

After obtaining the feature embedding of the commit instances for the cross-app data, a classifier is needed to build on the training data which is used to determine whether the commit instances in the test data will introduce the bug into the apps, i.e., buggy or clean. In this work, we employ the logistic regression (LR) as the basic classifier to build the prediction model. LR is a generalized linear model, which extends the linear regression model by incorporating the logistic function. The reasons of the choice of LR are that it is a simple classifier which solves the relationship between the features and labels of the commit instances, and is proved to be effective in previous bug prediction studies [23], [24], [56]–[59].

Let define the label as y_s and feature vector as $x_s = x_{s_1}, x_{s_2}, \ldots, x_{s_{ds}}$ of a commit instance in the source app data, where ds is the feature dimension. In addition, $w = w_1, \ldots, w_{ds}$ and b denote the weight vector associated with the features in x_s and a bias parameter, respectively, where w_i is the weight of ith feature of x_s. Then, the confidence scores (or probabilities) for x_s to be buggy and clean are calculated as follows:

$$\mathrm{P}(y_s = \mathrm{buggy}|x_s) = \frac{\exp(w \cdot x_s + b)}{1 + \exp(w \cdot x_s + b)} \tag{5}$$

$$\mathrm{P}(y_s = \mathrm{clean}|x_s) = \frac{1}{1 + \exp(w \cdot x_s + b)}. \tag{6}$$

The core of LR is to evaluate the weight vector w and bias b from the training set, i.e., the source app data. Then, the built model based on these parameters is used to predict the labels of the commit instances in the target app data.

F. Statistic Test

To statistically analyze the performance differences between our method and the comparative methods, we employ the Friedman test with the Nemenyi *post hoc* test [60] for significance test at significance level α with 0.05. This test takes the advantage that it does not require the performance results being analyzed to satisfy a particular distribution and is less sensitive to outliers [61]. The Friedman is a nonparametric hypothesis test which examines whether the significant differences exist among the average ranks of different methods. This test returns a p-value to decide whether the null hypothesis is true. The null hypothesis will be rejected when the p-value is less than 0.05.

When the null hypothesis is rejected, that is, the performance among multiple methods are significantly different, the Nemenyi *post hoc* test is employed to determine which method group is significantly different compared with the others [62]. Nemenyi *post hoc* test detects whether the average rank difference of two methods surpasses a critical distance (CD). If does, it denotes that the two methods have significant performance difference.

However, the original Nemenyi *post hoc* test will produce overlapping groups, i.e., a method may appear in multiple groups with significant differences. To overcome this drawback, in this work, we refer to the same strategy in [61] to produce the nonoverlapping groups with significant differences. More specifically, assume that the best and the worst average rank among the comparative methods as r_b and r_w individually, and the distance between r_b and r_w as d, we have the following three rules: 1) If $d > 2$ CD, the methods will be divided into three non-overlapping groups: the first group, called the top rank group, includes the methods whose absolute value of the rank difference toward r_b is less than CD. The second group, called the bottom rank group, includes the methods whose absolute value of the rank difference toward r_w is less than CD. The third group, called the middle rank group, includes the remaining methods. 2) If $\mathrm{CD} < d < 2$ CD, the methods will be divided into two non-overlapping groups: the top rank group contains the methods whose average rank is closer to r_b than r_w and the bottom rank group contains the method whose average rank is closer to r_w than r_b. 3) If $d <$ CD, all the methods belong to one group.

V. Performance Evaluation

A. RQ1: Does Our Proposed CDFE Method Achieve Better Cross-App Bug Prediction Performance Than Instance Selection-Based Cross-App Model?

Motivation: One way to alleviate the data distribution between two apps is to use instance selection methods to select some representative commit instances towards the target app data from the source app data. This research question is designed to investigate whether our proposed cross-app method CDFE can achieve better JIT bug prediction performance on mobile apps than instance selection based methods.

Methods: To answer this research question, we select three instance selection methods for comparison, including NF [63], PF [64], and YF [65], which are briefly described as follows.

1) *NF*: For each commit instance in the target app, NF selects its closest 10 commit instances from the source app. The nonredundant commit instances are used as the training set.
2) *PF*: This method applies k-means clustering to divide the data combining the source app and target app into multiple groups and reserves the groups that contain at least one commit instance for the target app. For each instance from the source app in the reserved groups, its closest instance in the target app is called the popular one. For each popular instance, PF selects its closest one from the source app as the candidate of the training set.
3) *YF*: This method applies agglomerative clustering to divide the data combining the source app and target app into multiple groups and reserves the groups that contain at least one commit instance for the target app. The commit instances from the source app in the reserved groups are used as the training set.

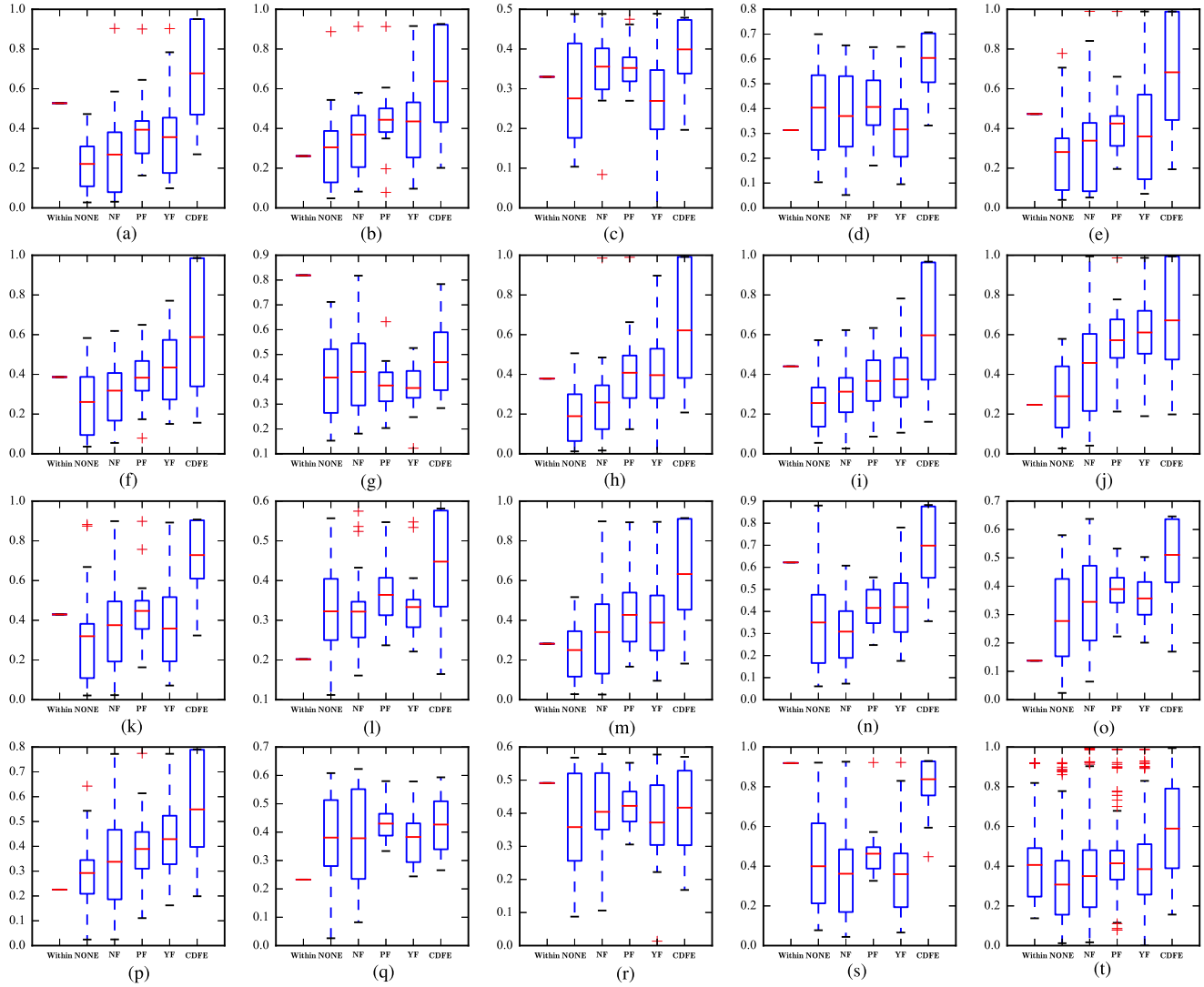

Fig. 5. Box-plots for EARecall across cross-app pairs on each app and all apps for CDFE, four instance selection methods, and one within-app method. (a) Firewall. (b) Alfresco. (c) Sync. (d) Wallpaper. (e) Keyboard. (f) Apg. (g) Applozic. (h) Atmosphere. (i) Secure. (j) Delta. (k) Facebook. (l) Image. (m) Kiwix. (n) Lottie. (o) Scroll. (p) Cloud. (q) Turner. (r) Reddit. (s) Telegram. (t) All.

TABLE III
AVERAGE INDICATOR VALUES OF OUR CDFE METHOD, FOUR INSTANCE SELECTION-BASED METHODS, AND ONE WITHIN-APP METHOD ACROSS ALL CROSS-APP PAIRS

Indicator	Within	NONE	NF	PF	YF	CDFE
EARecall	0.406	0.307	0.350	0.414	0.385	**0.589**
EAF-measure	0.388	0.277	0.309	0.352	0.341	**0.405**

These instance selection methods reserve part of commit instances in the source apps, which are representative toward the instances in the target app.

We also design the *NONE* method that only uses the classification model without instance selection and feature transformation on the source and target apps as the most basic instance selection based cross-app method. In addition, we design a within-app bug prediction method *Within* that only uses the LR classifier on the training and test data, both from the target app, for comparison.

Results: Table III reports the average EARecall and EAF-measure values of our proposed CDFE method, the four instance selection-based cross-app methods, and the within-app prediction method. Figs. 5 and 6 depict the box-plots of the six methods in which the first 19 subfigures show the box-plots on the cross-app pairs for each app and the last subfigure shows the box-plots on the cross-app pairs for all apps. Fig. 7 visualizes the corresponding statistical test results across all 342 cross-app pairs in which the methods with red, blue, and green color denote that they belong to the top, middle, and bottom rank groups, respectively.

Table III shows that our CDFE method achieves the best average EARecall and EAF-measure values compared with the within-app method and four instance selection methods. Compared with the best average indicator values among the five baseline methods, CDFE achieves improvements of 60.1% and 23.1% in terms of EARecall and EAF-measure, respectively.

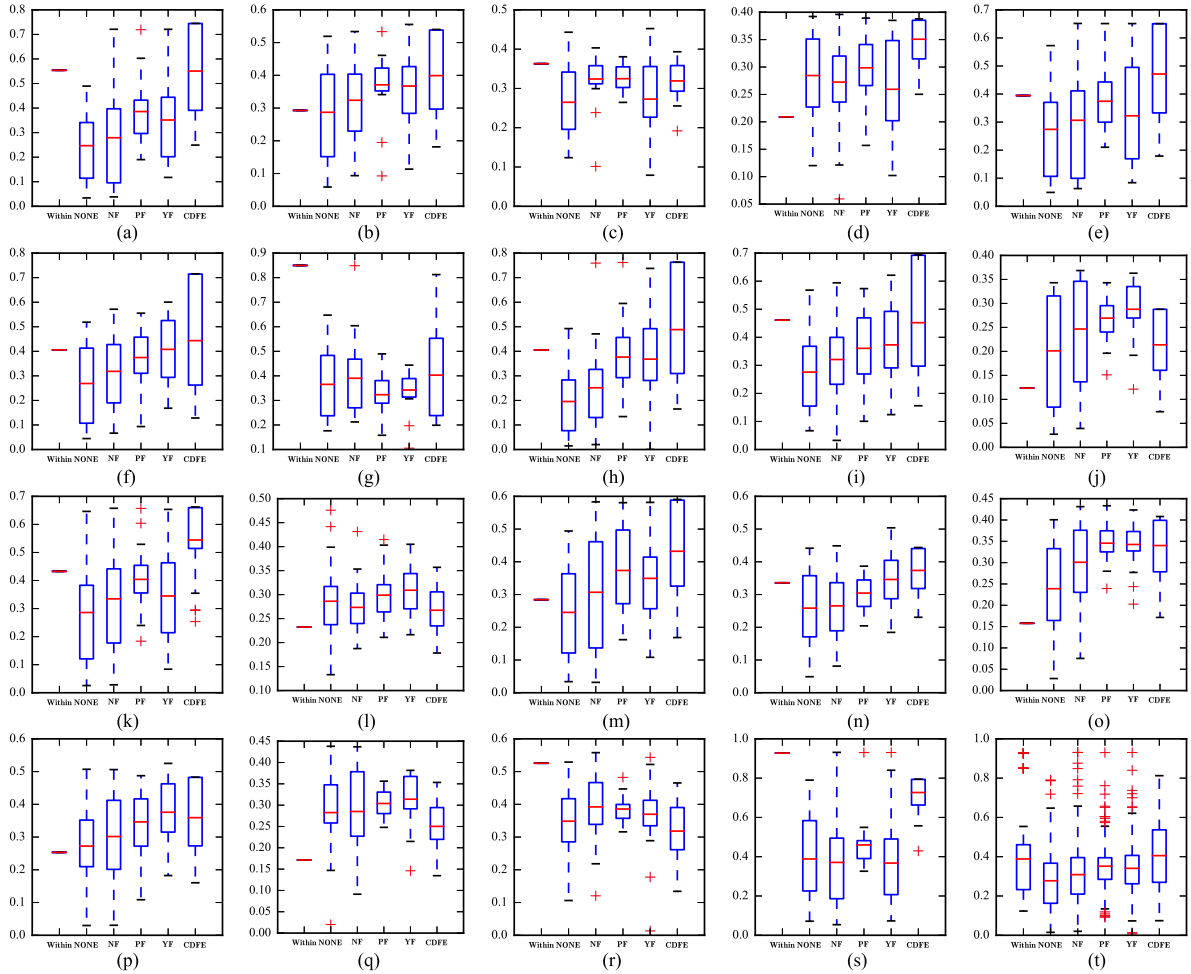

Fig. 6. Box-plots for EAF-measure across cross-app pairs on each app and all apps for CDFE, four instance selection methods, and one within-app method. (a) Firewall. (b) Alfresco. (c) Sync. (d) Wallpaper. (e) Keyboard. (f) Apg. (g) Applozic. (h) Atmosphere. (i) Secure. (j) Delta. (k) Facebook. (l) Image. (m) Kiwix. (n) Lottie. (o) Scroll. (p) Cloud. (q) Turner. (r) Reddit. (s) Telegram. (t) All.

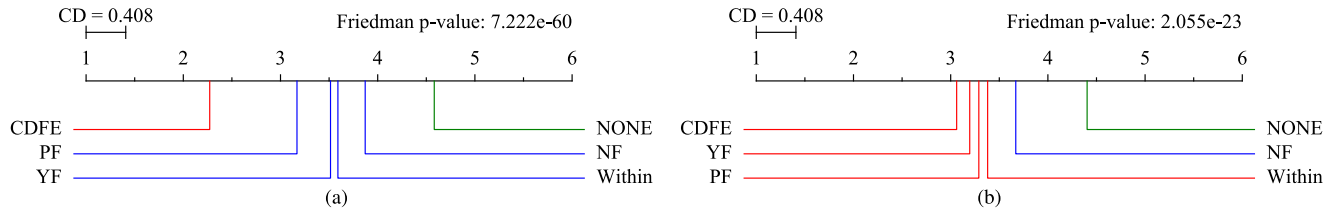

Fig. 7. Statistic results with the Nemenyi test among CDFE, four instance selection methods, and one within-app method for two effort-aware indicators. (a) EARecall. (b) EAF-measure.

Fig. 5 shows that the EARecall values of our CDFE method are apparently higher than that of the four instance selection-based methods on each app and across all apps. Compared with the Within method, the EARecall values of our CDFE method are apparently higher on 16 apps (except for Applozic, Reddit, and Telegram) and among all apps. Fig. 6 shows the EAF-measure values of our CDFE method are apparently higher than that of the four instance selection-based methods on eight apps (i.e., Firewall, Wallpaper, Keyboard, Atmosphere, Secure, Facebook, Kiwix, and Telegram), a little higher on four apps (i.e., Alfresco, Apg, Applozic, and Lottie), and across all apps, while a little lower on other apps. Compared with the Within method, the EAF-measure values of our CDFE method are apparently higher on 13 apps (except for Firewall, Sync, Applozic, Secure, Reddit, and Telegram) and among all apps.

Fig. 7 demonstrates that our CDFE method ranks first and belongs to the top rank group in terms of EARecall and EAF-measure. In addition, CDFE performs significantly better than

all baseline methods in terms of EARecall and than two baseline methods in terms of EAF-measure.

Answer to RQ1: In sum, our proposed CDFE method achieves better cross-app bug prediction performance than the instance selection methods and the Within method on the cross-app pairs of nearly all mobile apps in terms of the two effort-aware indicators.

B. RQ2: Does Our Proposed CDFE Method Perform Better Than Transfer Learning-Based Cross-App Model for Bug Prediction Performance?

Motivation: Transfer learning, especially for the feature based transfer learning, is the most commonly used cross-project model. Such kind of methods employs some feature transformation strategies, such as matrix transformation, to learn a common feature space to narrow the data distribution differences across domains. This research question is designed to investigate whether our proposed feature embedding-based metric learning method CDFE performs better than transfer learning methods in terms of JIT bug prediction performance on mobile apps.

Methods: To answer this research question, we select six transfer learning methods for comparison, including IFS_5 [66], IFS_16 [67], transfer component analysis (TCA) [68], conditional distribution-based transfer learning (CDT) [69], joint distribution-based transfer learning (JDT) [70], and extended TCA (TCA+) [58], which are briefly described as follows:

1) *IFS_5*: This method transforms the original features into a new space including five distribution characteristics of the commit instances, i.e., median, mean, minimum, maximum, and variance values.
2) *IFS_16*: This method is a variant version of IFS_5, which includes 16 distribution characteristics, i.e., 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.
3) *TCA*: TCA only considers the margin distribution of the cross-app data.
4) *CDT*: CDT only considers the conditional distribution of the cross-app data.
5) *JDT*: JDT simultaneously considers the margin distribution and conditional distribution of the cross-app data with the same weight.
6) *TCA+*: TCA+ first employs some predefined rules to preprocess the data of the two apps with a specific normalization strategy and then applies the TCA method.

These transfer learning methods map the data of the source and target apps into a new feature space in which their distributions are more similar.

Results: Table IV reports the average EARecall and EAF-measure values of our proposed CDFE method and the six transfer learning-based cross-app methods. Figs. 8 and 9 depict the box-plots of the six methods on the cross-app pairs for each app as well as all apps. Fig. 10 visualizes the corresponding statistical test results across all 342 cross-app pairs.

TABLE IV
AVERAGE INDICATOR VALUES OF OUR CDFE METHOD AND SIX TRANSFER LEARNING METHODS ACROSS ALL CROSS-APP PAIRS

Indicator	IFS_5	IFS_16	TCA	CDT	JDT	TCA+	CDFE
EARecall	0.348	0.345	0.354	0.350	0.354	0.363	**0.589**
EAF-measure	0.281	0.287	0.311	0.308	0.311	0.312	**0.405**

Table IV shows that our CDFE method achieves the best average EARecall and EAF-measure values compared with the six transfer learning methods. Compared with the best average indicator values among the six baseline methods, CDFE achieves improvements of 67.2% and 34.5% in terms of EARecall and EAF-measure, respectively.

Fig. 8 shows that the EARecall values of our CDFE method are apparently higher than that of the six baseline methods on 14 apps (i.e., Firewall, Wallpaper, Keyboard, Apg, Applozic, Atmosphere, Secure, Facebook, Image, Kiwix, Lottie, Scroll, Cloud, and Telegram) and across all apps, a little higher on four apps (i.e., Alfresco, Sync, Delta, and Turner), while a little lower on one app (i.e., Reddit) on one method (i.e., TCA+). Fig. 9 shows the EAF-measure values of our CDFE method are apparently higher than that of the six baseline methods on nine apps (i.e., Firewall, Keyboard, Applozic, Atmosphere, Secure, Facebook, Kiwix, Cloud, and Telegram), a little higher on five apps (i.e., Alfresco, Wallpaper, Apg, Lottie, and Scroll) and across all apps, while a little lower than on other five apps.

Fig. 10 demonstrates that our CDFE method ranks first in terms of EARecall and EAF-measure, respectively, and belongs to the top rank group in terms of both two effort-aware indicators. In addition, CDFE performs significantly better than all baseline methods in terms of both two effort-aware indicators.

Answer to RQ2: To sum up, our proposed CDFE method outperforms the transfer learning methods on the cross-app pairs of most mobile apps in terms of the two effort-aware indicators.

C. RQ3: Is Our CDFE Method Superior to Classifier Combination-Based Cross-App Model for Bug Prediction Performance?

Motivation: Classifier combination methods refer to the idea of ensemble learning to improve the cross-domain prediction performance with the help of multiple classification results. This research question is designed to investigate whether our proposed feature embedding method CDFE performs better than classifier combination methods in terms of JIT bug prediction performance on mobile apps.

Methods: To answer this research question, we select three classifier combination methods for comparison, including Bagging_ J48 (B_J48) [71], COmbined DEfect Predictor (CODEP) method [72], and Adaptive Selection of Classifiers in bug predIction (ASCI) method [73], which are briefly described as follows.

1) *B_J48*: This method trains multiple learning models and then combines them to reduce the generalization errors.
2) *CODEP*: This method first applies multiple classifiers to the cross-app data independently, and then uses the probability outputs of these classifiers as the new features of the the commit instances.

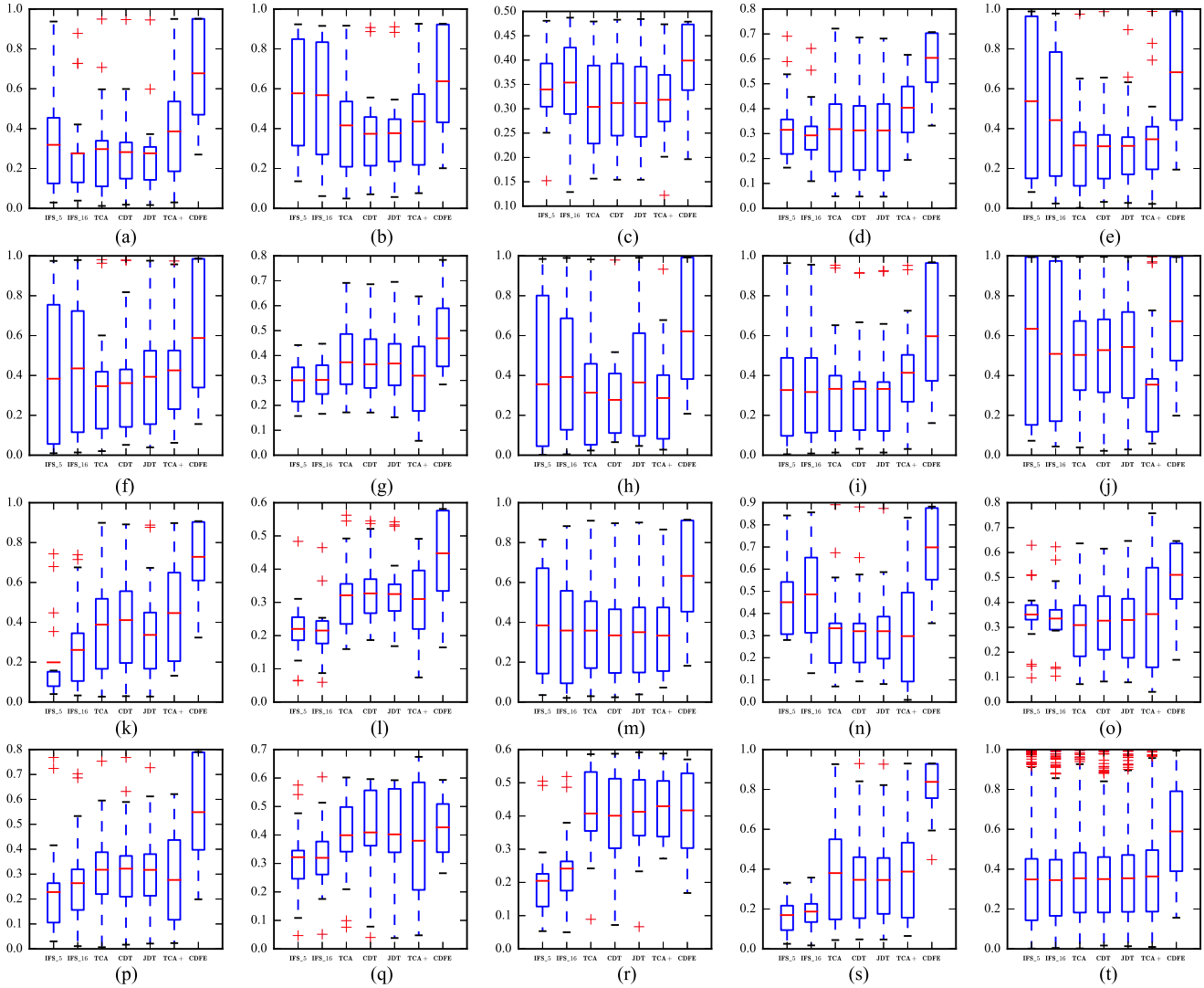

Fig. 8. Box-plots for EARecall across cross-app pairs on each app and all apps for CDFE and six transfer learning methods. (a) Firewall. (b) Alfresco. (c) Sync. (d) Wallpaper. (e) Keyboard. (f) Apg. (g) Applozic. (h) Atmosphere. (i) Secure. (j) Delta. (k) Facebook. (l) Image. (m) Kiwix. (n) Lottie. (o) Scroll. (p) Cloud. (q) Turner. (r) Reddit. (s) Telegram. (t) All.

TABLE V
AVERAGE INDICATOR VALUES OF OUR CDFE METHOD AND THREE CLASSIFIER COMBINATION METHODS ACROSS ALL CROSS-APP PAIRS

Indicator	B_J48	CODEP	ASCI	CDFE
EARecall	0.453	0.440	0.388	**0.589**
EAF-measure	0.368	0.366	0.339	**0.405**

3) *ASCI*: This method dynamically selects the right one from multiple classifiers based on the characteristics of the class that the classifier can better predict.

These classifier combination methods aim to improve the cross-app bug prediction performance from the perspective of classifier selection and combination.

Results: Table V reports the average EARecall and EAF-measure values of our CDFE method and the three classifier combination based cross-app methods. Figs. 11 and 12 depict the box-plots of the four methods on cross-app pairs for each app as well as all apps. Fig. 13 visualizes the corresponding statistical test results across all 342 cross-app pairs.

Table V shows that our CDFE method achieves the best average EARecall and EAF-measure values compared with the three classifier combination methods. Compared with the best average indicator values among the three baseline methods, CDFE achieves improvements of 38.6% and 13.4% in terms of EARecall and EAF-measure, respectively.

Fig. 11 shows that the EARecall values of CDFE are apparently higher than that of the three baseline methods on 12 apps (i.e., Firewall, Sync, Wallpaper, Keyboard, Applozic, Secure, Facebook, Image, Kiwix, Lottie, Scoll, and Telegram) and across all apps, a little higher on five apps (i.e., Alfresco, Apg, Atmosphere, Cloud, and Reddit), while a little lower on two apps (i.e., Delta and Turner). Fig. 12 shows the EAF-measure values of CDFE are apparently higher than that of the three baseline methods on five apps (i.e., Firewall, Wallpaper, Keyboard, Facebook, and Telegram), a little higher on five apps (Applozic,

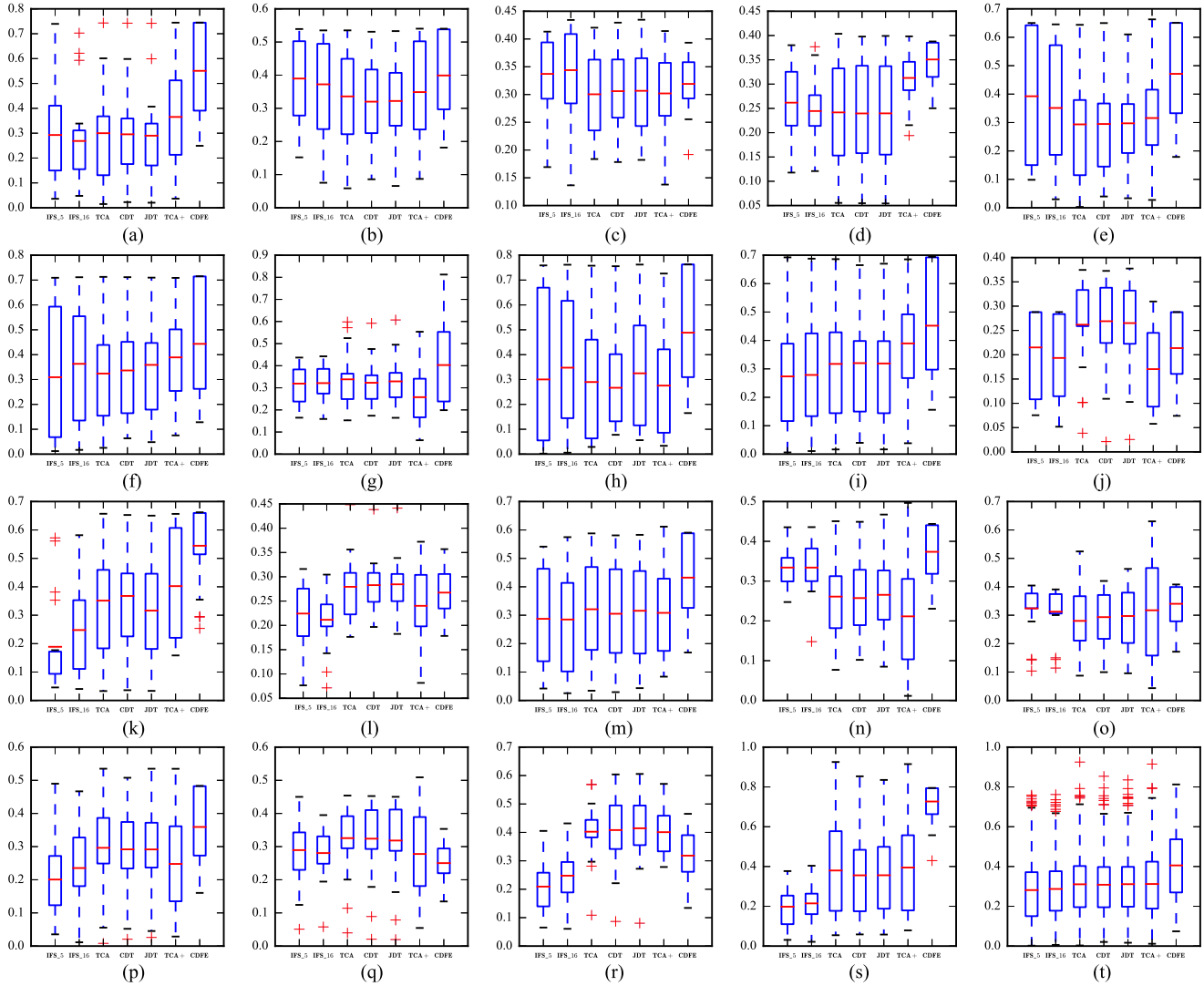

Fig. 9. Box-plots for EAF-measure across cross-app pairs on each app and all apps for CDFE and six transfer learning methods. (a) Firewall. (b) Alfresco. (c) Sync. (d) Wallpaper. (e) Keyboard. (f) Apg. (g) Applozic. (h) Atmosphere. (i) Secure. (j) Delta. (k) Facebook. (l) Image. (m) Kiwix. (n) Lottie. (o) Scroll. (p) Cloud. (q) Turner. (r) Reddit. (s) Telegram. (t) All.

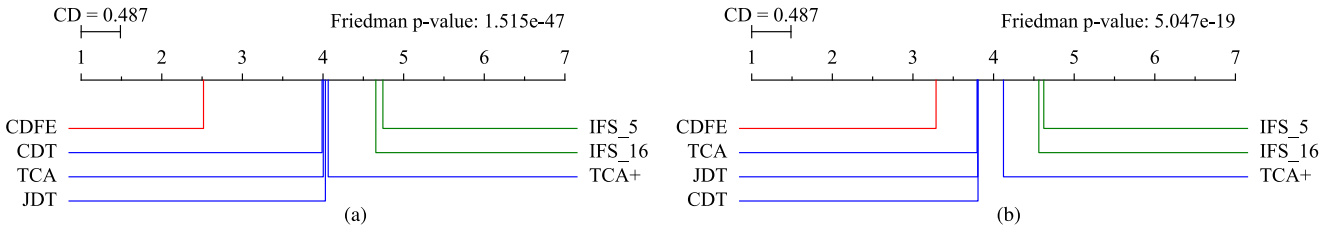

Fig. 10. Statistic results with the Nemenyi test among CDFE and six transfer learning methods for two effort-aware indicators. (a) EARecall. (b) EAF-measure.

Secure, Kiwix, Lottie, and Scroll) and across all apps, while a little lower on other apps.

Fig. 13 demonstrates that our CDFE method ranks first and belongs to the top rank group in terms of EARecall and EAF-measure. In addition, CDFE performs significantly better than all baseline methods in terms of EARecall and than two baseline methods in terms of EAF-measure expect for B_J48.

Answer to RQ3: In short, our proposed CDFE method performs significantly better than the classifier combination methods across all cross-app pairs in terms of the two effort-aware indicators.

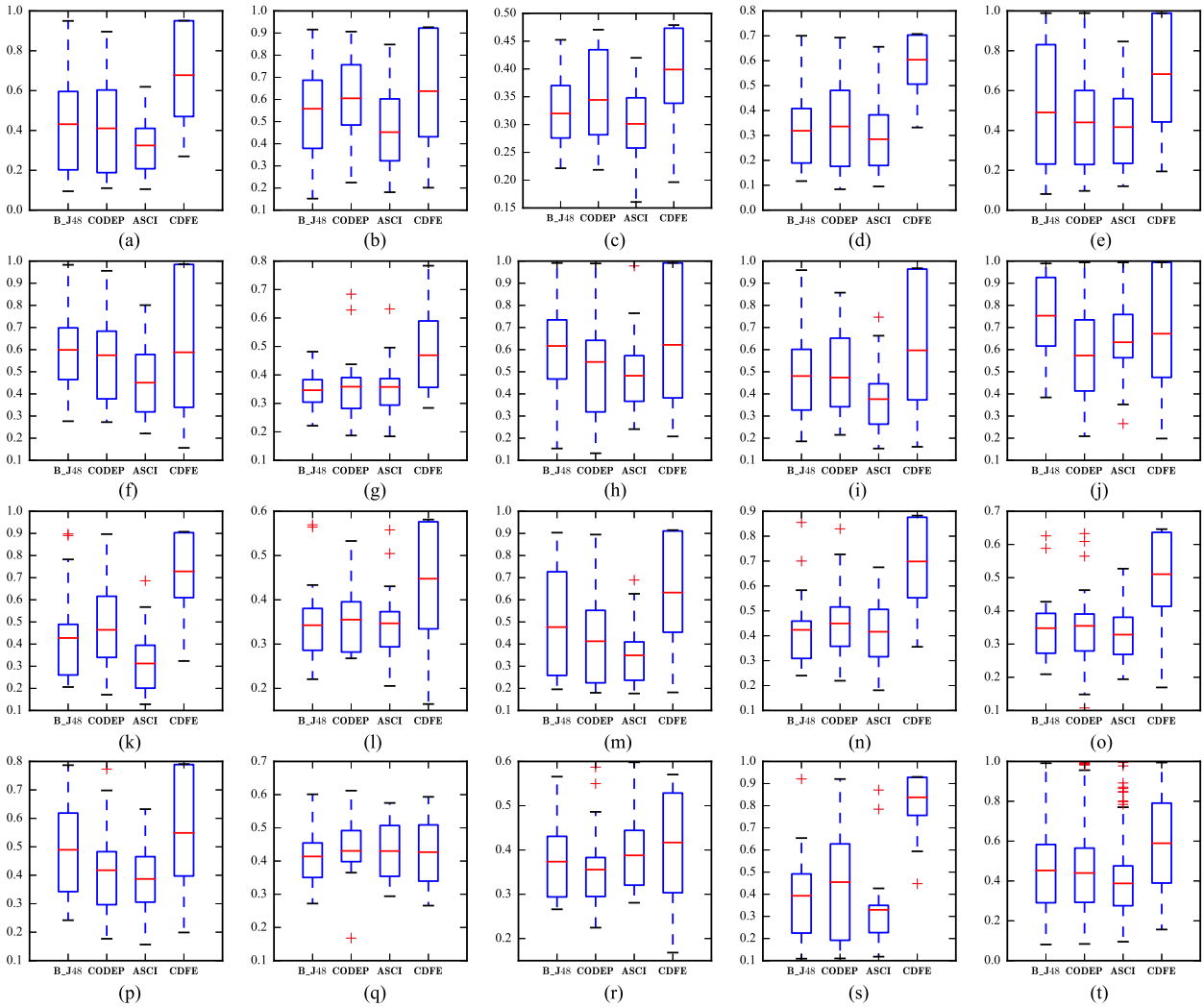

Fig. 11. Box-plots for EARecall across cross-app pairs on each app and all apps for CDFE and three classifier combination methods. (a) Firewall. (b) Alfresco. (c) Sync. (d) Wallpaper. (e) Keyboard. (f) Apg. (g) Applozic. (h) Atmosphere. (i) Secure. (j) Delta. (k) Facebook. (l) Image. (m) Kiwix. (n) Lottie. (o) Scroll. (p) Cloud. (q) Turner. (r) Reddit. (s) Telegram. (t) All.

VI. Threats to Validity

A. Threats to Construct Validity

Threats to construct validity depend on the reasonability of the used performance measurement and statistic test method. EARecall and EAF-measure are used as our main performance indicators which have been suggested as the appropriate performance evaluation for the bug prediction task. The used improved statistic test method combining Friedman test with a Nemenyi *post hoc* test has the advantage to generate nonoverlapped groups with significant differences, which has been successfully applied to many previous bug prediction studies [55], [57], [69], [74]–[76].

B. Threats to Internal Validity

Threats to internal validity lie in experimental mistakes when implementing our method and replicating the comparative methods. We carefully implement CDFE method with tensorflow and the first two authors have double-checked the implementation details to reduce the threats to the internal validity. Since the original implementations of most baseline methods (except for TCA, CDT, and JDT) were not available, we reimplement them by strictly following the corresponding descriptions in the original papers. Nonetheless, we still could not claim that our versions of the baseline methods can fully restore all of their original details.

C. Threats to External Validity

Threats to external validity come from the generalization of our experimental results to other datasets. Currently, the benchmark dataset consists of 19 apps which have different size and are from different application domains. This is helpful to generalize the experimental results to a certain extent. Selecting more apps to be included in the benchmark dataset can further improve the generalizability of the proposed method, which is

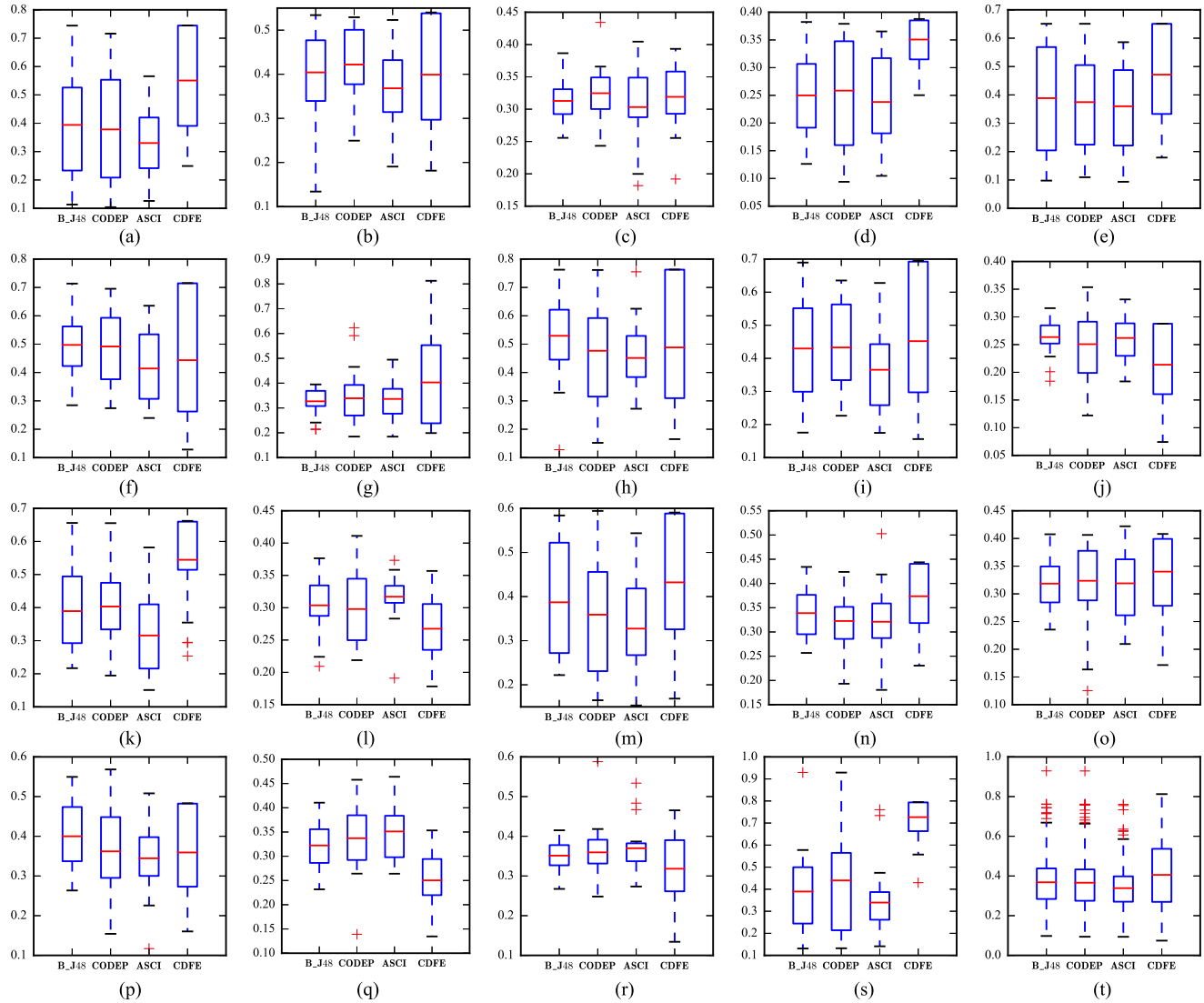

Fig. 12. Box-plots for EAF-measure across cross-app pairs on each app and all apps for CDFE and three classifier combination methods. (a) Firewall. (b) Alfresco. (c) Sync. (d) Wallpaper. (e) Keyboard. (f) Apg. (g) Applozic. (h) Atmosphere. (i) Secure. (j) Delta. (k) Facebook. (l) Image. (m) Kiwix. (n) Lottie. (o) Scroll. (p) Cloud. (q) Turner. (r) Reddit. (s) Telegram. (t) All.

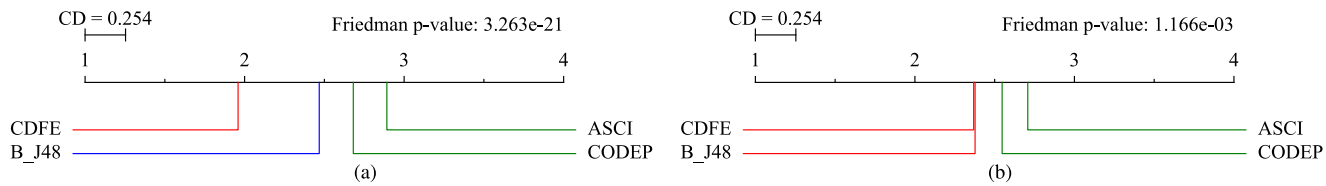

Fig. 13. Statistic results with the Nemenyi test among CDFE and three classifier combination methods for two effort-aware indicators. (a) EARecall. (b) EAF-measure.

our future work. Since the 19 apps are developed on the Android platform, considering additional apps developed on other platform, such as IOS, could also strengthen the generalizability of our method. As the programming constructs and structures can vary among different programming languages and using different constructs and structures may cause the calculated feature values different [77], selecting mobile apps written in different languages (such as Kotlin), not only Java as our apps, would further increase the external validity of this work.

VII. Conclusion

In this work, we developed a deep learning-based cross-domain metric learning method, called CDFE, that learns deep

feature embedding for cross-app JIT bug prediction task. This method incorporated a state-of-the-art cross-triplet loss function into a DNN model to learn high-level feature representation for the cross-app data. Cross-triplet loss function is a variant of original triplet loss function which extends the later one from within-domain to cross-domain scenario. This loss function aims to learn a feature space in which the commit instances with the same label are close together, whereas the commit instances with different labels are far apart. We evaluated our proposed CDFE method on a publicly available benchmark dataset consisting of 19 mobile apps and compared it with 14 baseline methods. The experimental results showed that our method achieved the best average performance values in terms of two effort-aware indicators.

As potential future work, we are planning to apply our method to more bug data of mobile apps and adapt our method to imbalance data since class imbalance is inherent in bug prediction on mobile apps.

REFERENCES

[1] G. Catolino, D. Di Nucci, and F. Ferrucci, "Cross-project just-in-time bug prediction for mobile apps: An empirical assessment," in *Proc. 6th Int. Conf. Mobile Softw. Eng. Syst.*, 2019, pp. 99–110.

[2] X. Kong, L. Zhang, W. E. Wong, and B. Li, "Experience report: How do techniques, programs, and tests impact automated program repair?" in *Proc. 26th Int. Symp. Softw. Rel. Eng.*, 2015, pp. 194–204.

[3] G. Catolino, "Just-in-time bug prediction in mobile applications: The domain matters!" in *Proc. 4th Int. Conf. Mobile Softw. Eng. Syst.*, 2017, pp. 201–202.

[4] M. Yan, X. Xia, Y. Fan, A. E. Hassan, D. Lo, and S. Li, "Just-in-time defect identification and localization: A two-phase framework," *IEEE Trans. Softw. Eng.*, 2020, doi: 10.1109/TSE.2020.2978819.

[5] M. Yan, X. Xia, Y. Fan, D. Lo, A. E. Hassan, and X. Zhang, "Effort-aware just-in-time defect identification in practice: A case study at Alibaba," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Foundations Softw. Eng.*, 2020, pp. 1308–1319.

[6] Z. Xu *et al.*, "LDFR: Learning deep feature representation for software defect prediction," *J. Syst. and Softw.*, vol. 158, 2019, Art. no. 110402.

[7] S. Jiang, Y. Wu, and Y. Fu, "Deep bi-directional cross-triplet embedding for cross-domain clothing retrieval," in *Proc. 24th ACM Int. Conf. Multimedia*, 2016, pp. 52–56.

[8] E. Arisholm, L. C. Briand, and M. Fuglerud, "Data mining techniques for building fault-proneness models in telecom java software," in *Proc. 18th IEEE Int. Symp. Softw. Rel. Eng.*, 2007, pp. 215–224.

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

[10] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi, "An empirical study of just-in-time defect prediction using cross-project models," in *Proc. 11th Work. Conf. Mining Softw. Repositories*, 2014, pp. 172–181.

[11] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models," *Empirical Softw. Eng.*, vol. 21, no. 5, pp. 2072–2106, 2016.

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

[13] P. Tourani and B. Adams, "The impact of human discussions on just-in-time quality assurance: An empirical study on openstack and eclipse," in *Proc. 23rd Int. Conf. Softw. Anal. Evol. Reengineering*, 2016, vol. 1, pp. 189–200.

[14] Q. Huang, X. Xia, and D. Lo, "Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction," in *Proc. 33 rd Int. Conf. Softw. Maintenance Evol.*, 2017, pp. 159–170.

[15] Q. Huang, X. Xia, and D. Lo, "Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction," *Empirical Softw. Eng.*, vol. 24, no. 5, pp. 2823–2862, 2019.

[16] X. Chen, Y. Zhao, Q. Wang, and Z. Yuan, "Multi: Multi-objective effort-aware just-in-time software defect prediction," *Inf. Softw. Technol.*, vol. 93, pp. 1–13, 2018.

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

[18] M. Kondo, D. M. German, O. Mizuno, and E.-H. Choi, "The impact of context metrics on just-in-time defect prediction," *Empirical Softw. Eng.*, vol. 25, no. 1, pp. 890–939, 2020.

[19] Y. Fan, X. Xia, D. A. da Costa, D. Lo, A. E. Hassan, and S. Li, "The impact of changes mislabeled by SZZ on just-in-time defect prediction," *IEEE Trans. Softw. Eng.*, 2019, doi: 10.1109/TSE.2019.2929761.

[20] T. Mende and R. Koschke, "Revisiting the evaluation of defect prediction models," in *Proc. 5th Int. Conf. Predictor Models Softw. Eng.*, 2009, pp. 1–10.

[21] T. Mende and R. Koschke, "Effort-aware defect prediction models," in *Proc. 14th Eur. Conf. Softw. Maintenance Reengineering*, 2010, pp. 107–116.

[22] E. Shihab, Y. Kamei, B. Adams, and A. E. Hassan, "Is lines of code a good measure of effort in effort-aware models?" *Inf. Softw. Technol.*, vol. 55, no. 11, pp. 1981–1993, 2013.

[23] Y. Yang *et al.*, "Are slice-based cohesion metrics actually useful in effort-aware post-release fault-proneness prediction? an empirical study," *IEEE Trans. Softw. Eng.*, vol. 41, no. 4, pp. 331–357, Apr. 2015.

[24] Y. Yang *et al.*, "An empirical study on dependence clusters for effort-aware fault-proneness prediction," in *Proc. 31st Int. Conf. Automated Softw. Eng.*, 2016, pp. 296–307.

[25] K. E. Bennin, J. Keung, A. Monden, Y. Kamei, and N. Ubayashi, "Investigating the effects of balanced training and testing datasets on effort-aware fault prediction models," in *Proc. 40th Annu. Comput. Softw. Appl. Conf.*, 2016, vol. 1, pp. 154–163.

[26] W. Ma, L. Chen, Y. Yang, Y. Zhou, and B. Xu, "Empirical analysis of network measures for effort-aware fault-proneness prediction," *Inf. Softw. Technol.*, vol. 69, pp. 50–70, 2016.

[27] J. Liu, Y. Zhou, Y. Yang, H. Lu, and B. Xu, "Code churn: A neglected metric in effort-aware just-in-time defect prediction," in *Proc. 11th Int. Symp. Empirical Softw. Eng. Meas.*, 2017, pp. 11–19.

[28] X. Yu, K. E. Bennin, J. Liu, J. W. Keung, X. Yin, and Z. Xu, "An empirical study of learning to rank techniques for effort-aware defect prediction," in *Proc. 26th Int. Conf. Softw. Anal., Evol. Reengineering*, 2019, pp. 298–309.

[29] S. Albahli, "A deep ensemble learning method for effort-aware just-in-time defect prediction," *Future Internet*, vol. 11, no. 12, pp. 1–13, 2019.

[30] L. Qiao and Y. Wang, "Effort-aware and just-in-time defect prediction with neural network," *PLoS One*, vol. 14, no. 2, pp. 1–19. 2019.

[31] W. Li, W. Zhang, X. Jia, and Z. Huang, "Effort-aware semi-supervised just-in-time defect prediction," *Inf. Softw. Technol.*, vol. 126, 2020, Art. no. 106364.

[32] R. Scandariato and J. Walden, "Predicting vulnerable classes in an android application," in *Proc. 4th Int. Workshop Secur. Meas. Metrics*, 2012, pp. 11–16.

[33] A. Kaur, K. Kaur, and H. Kaur, "An investigation of the accuracy of code and process metrics for defect prediction of mobile applications," in *Proc. 4th Int. Conf. Reliability Infocom Technol. Optim.*, 2015, pp. 1–6.

[34] A. Kaur, K. Kaur, and H. Kaur, "Application of machine learning on process metrics for defect prediction in mobile application," in *Inf. Syst. Design Intell. Appl.*. Springer, 2016, pp. 81–98.

[35] R. Malhotra, "An empirical framework for defect prediction using machine learning techniques with android software," *Appl. Soft Comput.*, vol. 49, pp. 1034–1050, 2016.

[36] M. Y. Ricky, F. Purnomo, and B. Yulianto, "Mobile application software defect prediction," in *Proc. 10th IEEE Symp. Service-Oriented Syst. Eng.*, 2016, pp. 307–313.

[37] N. Gayatri, S. Nickolas, and A. Reddy, "A frame work for business defect predictions in mobiles," *Int. J. Comput. Appl.*, vol. 975, pp. 39–44, 2013.

[38] Y. Fan, X. Cao, J. Xu, S. Xu, and H. Yang, "High-frequency keywords to predict defects for android applications," in *Proc. 42nd Annu. Comput. Softw. Appl. Conf.*, 2018, vol. 2, pp. 442–447.

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

[40] C. Manjula and L. Florence, "Deep neural network based hybrid approach for software defect prediction using software metrics," *Cluster Comput.*, vol. 22, no. 4, pp. 9847–9863, 2019.

[41] L. Qiao, X. Li, Q. Umer, and P. Guo, "Deep learning based software defect prediction," *Neurocomputing*, vol. 385, pp. 100–110, 2020.

[42] A. Hasanpour, P. Farzi, A. Tehrani, and R. Akbari, "Software defect prediction based on deep learning models: Performance study," 2020, arXiv:2004.02589.>

[43] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 297–308.

[44] S. Wang, T. Liu, J. Nam, and L. Tan, "Deep semantic feature learning for software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 46, no. 12, pp. 1267–1293, Dec. 2020.

[45] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *Proc. IEEE Int. Conf. Softw. Quality Rel. Secur.*, 2017, pp. 318–328.

[46] H. K. Dam *et al.*, "A deep tree-based model for software defect prediction," 2018, arXiv:1802.00921.

[47] A. V. Phan, M. Le Nguyen, and L. T. Bui, "Convolutional neural networks over control flow graphs for software defect prediction," in *Proc. 29th Int. Conf. Tools Artif. Intell.*, 2017, pp. 45–52.

[48] D. Chen, X. Chen, H. Li, J. Xie, and Y. Mu, "DeepCPDP: Deep learning based cross-project defect prediction," *IEEE Access*, vol. 7, pp. 184 832–184848, 2019.

[49] G. Fan, X. Diao, H. Yu, K. Yang, and L. Chen, "Deep semantic feature learning with embedded static metrics for software defect prediction," in *Proc. 26th Asia-Pacific Softw. Eng. Conf.*, 2019, pp. 244–251.

[50] S. Jiang, Y. Wu, and Y. Fu, "Deep bidirectional cross-triplet embedding for online clothing shopping," *ACM Trans. Multimedia Comput.,Commun. Appl.*, vol. 14, no. 1, pp. 1–22, 2018.

[51] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.

[52] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. 3rd Int. Conf. Learn. Representations (ICLR)*, 2015, pp. 1–15.

[53] M. Yan, Y. Fang, D. Lo, X. Xia, and X. Zhang, "File-level defect prediction: Unsupervised vs. supervised models," in *Proc. 11th Int. Symp. Empirical Softw. Eng. Meas.*, 2017, pp. 344–353.

[54] Z. Xu *et al.*, "Cross version defect prediction with representative data via sparse subset selection," in *Proc. 26th Int. Conf. Prog. Comprehension*, 2018, pp. 132–143.

[55] Z. Xu *et al.*, "TSTSS: A two-stage training subset selection framework for cross version defect prediction," *J. Syst. Softw.*, vol. 154, pp. 59–78, 2019.

[56] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan, "Heterogeneous defect prediction," *IEEE Trans. Softw. Eng.*, vol. 44, no. 9, pp. 874–896, Sep. 2018.

[57] J. Nam and S. Kim, "CLAMI: Defect prediction on unlabeled datasets (t)," in *Proc. 30th Int. Conf. Automated Softw. Eng.*, 2015, pp. 452–463.

[58] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 382–391.

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

[60] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *J. Mach. Learn. Res.*, vol. 7, no. Jan, pp. 1–30, 2006.

[61] S. Herbold, A. Trautsch, and J. Grabowski, "A comparative study to benchmark cross-project defect prediction approaches," *IEEE Trans. Softw. Eng.*, vol. 44, no. 9, pp. 811–833, Sep. 2018.

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

[63] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Softw. Eng.*, vol. 14, no. 5, pp. 540–578, 2009.

[64] F. Peters, T. Menzies, and A. Marcus, "Better cross company defect prediction," in *Proc. 10th Work. Conf. Mining Softw. Repositories*, 2013, pp. 409–418.

[65] X. Yu, P. Zhou, J. Zhang, and J. Liu, "A data filtering method based on agglomerative clustering," in *Proc. 29th Int. Conf. Softw. Eng. Knowl. Eng.*, 2017, pp. 392–397.

[66] P. He, B. Li, and Y. Ma, "Towards cross-project defect prediction with imbalanced feature sets," 2014, arXiv:1411.4228.

[67] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang, "An investigation on the feasibility of cross-project defect prediction," *Automated Softw. Eng.*, vol. 19, no. 2, pp. 167–199, 2012.

[68] S. J. Pan, I. W. Tsang, J. T. Kwok, and Q. Yang, "Domain adaptation via transfer component analysis," *Trans. Neural Netw.*, vol. 22, no. 2, pp. 199–210, 2011.

[69] Z. Xu *et al.*, "Cross project defect prediction via balanced distribution adaptation based transfer learning," *J. Comput. Sci. Technol.*, vol. 34, no. 5, pp. 1039–1062, 2019.

[70] M. Long, J. Wang, G. Ding, J. Sun, and P. S. Yu, "Transfer feature learning with joint distribution adaptation," in *Proc. 14th Int. Conf. Comput. Vis.*, 2013, pp. 2200–2207.

[71] Y. Zhang, D. Lo, X. Xia, and J. Sun, "An empirical study of classifier combination for cross-project defect prediction," in *Proc. 39th Annu. Comput. Softw. Appl. Conf.*, 2015, vol. 2, pp. 264–269.

[72] A. Panichella, R. Oliveto, and A. De Lucia, "Cross-project defect prediction models: L'union fait la force," in *Proc. 21st Softw. Evol. Week Conf. Softw. Maintenance, Reengineering Reverse Eng.*, 2014, pp. 164–173.

[73] D. Di Nucci, F. Palomba, and A. De Lucia, "Evaluating the adaptive selection of classifiers for cross-project bug prediction," in *Proc. 6th Int. Workshop Realizing Artif. Intell. Synergies Softw. Eng.*, 2018, pp. 48–54.

[74] Z. Li, X.-Y. Jing, F. Wu, X. Zhu, B. Xu, and S. Ying, "Cost-sensitive transfer kernel canonical correlation analysis for heterogeneous defect prediction," *Automated Softw. Eng.*, vol. 25, no. 2, pp. 201–245, 2018.

[75] Z. Xu *et al.*, "Identifying crashing fault residence based on cross project model," in *Proc. 30th Int. Symp. Softw. Rel. Eng.*, 2019, pp. 183–194.

[76] H. Chen, X.-Y. Jing, Z. Li, D. Wu, Y. Peng, and Z. Huang, "An empirical study on heterogeneous defect prediction approaches," *IEEE Trans. Softw. Eng.*, 2020, doi: 10.1109/TSE.2020.2968520.

[77] R. Ferenc, P. Gyimesi, G. Gyimesi, Z. Tóth, and T. Gyimóthy, "An automatically created novel bug dataset and its validation in bug prediction," *J. Syst. and Softw.*, vol. 169, 2020, Art. no. 110691.