Simplified Deep Forest Model Based Just-in-Time Defect Prediction for Android Mobile Apps

Kunsong Zhao, Zhou Xu, Tao Zhang, Yutian Tang, and Meng Yan

***Abstract*—The popularity of mobile devices has led to an explosive growth in the number of mobile apps in which Android mobile apps are the mainstream. Android mobile apps usually undergo frequent update due to new requirements proposed by users. Just-in-time (JIT) defect prediction is appropriate for this scenario for quality assurance because it can provide timely feedback by determining whether a new code commit will introduce defects into the apps. As defect-prediction performance usually relies on the quality of the data representation and the used classification model, in this work, we propose a model, called Simplified Deep Forest (SDF), to conduct JIT defect prediction for Android mobile apps. SDF modifies a state-of-the-art deep forest model by removing the multigrained scanning operation that is designed for data with a high-dimensional feature space. It uses a cascade structure with ensemble forests for representation learning and classification. We conduct experiments on 10 Android mobile apps and experimental results show that SDF performs significantly better than comparative methods in terms of 3 performance indicators.**

***Index Terms*—Deep forest, feature representation learning, just-in-time (JIT) defect prediction, mobile apps, quality assurance.**

I. Introduction

SOFTWARE plays a crucial role in the daily life of people. Due to the increasing scale and complexity of software, almost all of software products we used today exist defects inevitably [1]. The defects in the products may not only negatively impact the user experience, but also result in serious economic losses. It is necessary and challenging to fix defects earlier. Software defect prediction is an important research topic in software engineering, which builds a model to predict whether a software unit is defective or clean before releasing by using machine learning techniques [2]. This process helps with software testing and debugging to save massive manual efforts [3], [4]. Many researchers contribute to this topic for the improvement of software quality.

With the popularity of mobile devices in recent years, mobile apps develop rapidly, especially for the Android mobile apps. As the functional requirements of apps keep increasing, developers need to frequently update the apps [5], [6]. Due to various unexpected factors, the update process may introduce defects into the next version of the mobile apps. It is important to detect defects early in this process for developing high-quality Android mobile apps.

Traditional defect-prediction studies determine the existence of defects in software unit at method or class level [7]–[9]. It cannot detect possible bugs timely. To alleviate this issue, researchers proposed just-in-time (JIT) defect prediction [10]–[14]. The purpose of JIT defect prediction is to determine whether a code commit will introduce defects into the software, which has the potential to provide real-time defect feedback to the developers, helping them take timely actions. From this point of view, JIT defect prediction is especially suitable for the software that needs to be updated frequently involving in many code commits. If a new commit introduces defects into the apps, this commit is deemed as defective, otherwise, clean. JIT defect prediction can help developers find defects early and reduce the effort of software testing [11].

Due to the characteristic of frequent update for Android mobile apps, JIT defect prediction is suitable for mobile apps. Catolino *et al.* [12], [15] proposed JIT defect-prediction models in the context of mobile apps. They employed the feature information derived from commit records for classification. In general, the defect-prediction performance is highly related to the feature representation quality of the defect data and the used classification model.

As the previous studies [12], [15] just used traditional classifiers for classification task and employed the original extracted features as the input of the prediction models without any preprocessing, in this work, we use a novel deep forest model to address the two issues. This model, called gcForest, is a decision tree ensemble method and uses a cascade structure

Manuscript received July 30, 2020; revised December 7, 2020; accepted February 14, 2021. This work was supported in part by the National Natural Science Foundation of China under Grant 62002034, in part by the China Postdoctoral Science Foundation under Grant 2020M673137, in part by the Fundamental Research Funds for the Central Universities under Grant 2020CDCGRJ072 and Grant 2020CDJQY-A021, 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, and in part by the Faculty Research Grant Projects of MUST under Grant FRG-20-008-FI. Associate Editor: C. Budnik. *(Corresponding authors: Zhou Xu; Tao Zhang.)*

Kunsong Zhao is with the School of Big Data and Software Engineering, Chongqing University, Chongqing 400044, China, and also with the School of Computer Science, Wuhan University, Wuhan 430072, China (e-mail: kszhao@whu.edu.cn).

Zhou Xu and Meng Yan are with the School of Big Data and Software Engineering, Chongqing University, Chongqing 400044, China, and also with the Key Laboratory of Dependable Service Computing in Cyber Physical Society, (Chongqing University), Ministry of Education, Chongqing 40004, China (e-mail: zhouxullx@cqu.edu.cn; mengy@cqu.edu.cn).

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

Yutian Tang is with the School of Information Science and Technology, ShanghaiTech University, Shanghai 201210, China (e-mail: csytang@ieee.org).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TR.2021.3060937.

Digital Object Identifier 10.1109/TR.2021.3060937

with ensemble forests for feature representation learning and classification [16]. It consists of a multigrained scanning method for high-dimensional features to enhance representation learning ability. Considering that our defect data for Android mobile apps include low-dimensional features, we use the Simplified Deep Forest (SDF) model that does not use the multigrained scanning to process our defect data for the JIT defect-prediction task on Android mobile apps.

In this work, we conduct experiments on 10 Android mobile apps and employ three indicators to evaluate the performance of our proposed SDF model for JIT defect-prediction task. Across the 10 apps, our proposed SDF model achieves average *F*-measure, Matthew's correlation coefficient (MCC), and Area Under the receiver operating characteristic Curve (AUC) of 0.424, 0.304, and 0.636, respectively. We compare our SDF model with eight commonly used classification models and SDF combining with eight feature extraction methods. The experimental results show that SDF achieves average improvements by 44.7%, 41.7%, and 7.6% in terms of *F*-measure, MCC, and AUC, respectively, toward the comparative classification models, and by 30.3%, 31.7%, and 6.0 in terms of *F*-measure, MCC, and AUC, respectively, toward the comparative feature extraction methods.

The main contributions of this article are summarized as follows.

1) We consider the feature representation learning issue in JIT defect prediction for Android mobile apps, which is neglected in previous studies.
2) We propose a novel method, namely SDF, to conduct JIT defect prediction for Android mobile apps. It achieves both representation learning and classification by employing a cascade structure with ensemble forests.
3) We use defect data from 10 Android mobile apps as benchmark dataset and conduct sufficient statistic test to analyze the experimental results. The results show that our proposed SDF model performs significantly better than 16 comparative methods.

The rest of this article is organized as follows: Section II introduces the related work. The details of our SDF model are described in Section III. In Section IV, we describe the experimental setup, followed by experimental results in Section V. Section VI discusses threats to the validity of our study. Finally, Section VII concludes this article.

II. Related Work

A. JIT Defect Prediction for Traditional Software

In recent years, many studies have focused on JIT defect prediction for traditional software. Kamei *et al.* [10] considered JIT quality assurance for defect-prediction models. They used a wide range of factors based on the characteristics of a software change and found that their method could detect 35% of all defect-inducing changes by using only 20% of effort for inspecting all of them. Fukushima *et al.* [17] empirically evaluated the performance of JIT cross-project models on 11 open-source projects. They found that models having similar correlations between the predictor and dependent variables often got better performance. They stated that JIT cross-project models trained on other projects could provide a viable solution for projects with little historical data. Kamei *et al.* [11] employed an empirical study to evaluate the performance of JIT models in a cross-project context on 11 open-source projects. Their results showed that combining the data of other projects to produce a larger training data tended to improve the performance. McIntosh *et al.* [18] studied JIT models from the rapidly evolving systems with 37 524 changes. They found that fluctuations in the properties of fix-inducing changes could impact the performance of JIT models. Pascarella *et al.* [19] proposed a novel fine-grained JIT defect-prediction model to predict whether the specific commits are defective or not. They conducted experiments on 10 open-source software systems and found that defective commit was a mixture of defective and clean files. Yang *et al.* [3] proposed an approach called Deeper, which leveraged deep belief network and logistic regression classifier to predict defect-prone changes. They used datasets from six large open-source projects containing 137 417 changes. The experimental results showed that their method achieved *F*1-scores between 0.22 and 0.63. Yang *et al.* [20] proposed a two-layer ensemble learning method, called TLEL, which combined decision tree with ensemble learning to improve the performance of JIT defect prediction. They conducted experiments on six large open-source datasets and the experimental results showed that TLEL achieved a significant improvement over the baselines. Chen *et al.* [21] proposed a multiobjective-optimization-based supervised method, called MULTI, which used logistic regression to build JIT models. They evaluated the performance on six open-source projects with 227 417 changes. Their experimental results showed that MULTI could perform significantly better than all of the comparative methods on indicators of AUC and $\mathrm{P_{opt}}$. Cabral *et al.* [22] provided the first investigation of JIT defect prediction with class imbalance evolution on 10 GitHub projects. The experimental results showed that JIT defect prediction with class imbalance issue was worth exploring. In order to explore the influence of context of code for JIT defect prediction, Kondo *et al.* [23] proposed the context metrics that were defined as the number of words or keywords in the context lines. Their results showed that the combination metrics of two extended context metrics significantly outperformed other metrics in all six projects when considering MCC and AUC.

Different from these studies that conducted experiments on traditional software, in this work, we focus on the JIT defect prediction on Android mobile apps.

B. JIT Defect Prediction in Mobile Apps

Due to frequent update in Android mobile apps, JIT defect prediction is particularly appropriate for such scenarios for timely feedback. Catolino *et al.* [15] preliminarily studied the effectiveness of logistic regression model for JIT defect prediction on five mobile apps. Their results showed that the further exploration was needed. Subsequently, Catolino *et al.* [12] employed an empirical study to investigate the useful features, the performance differences of four traditional classifiers, and the effectiveness of ensemble learning techniques for JIT defect prediction on Android mobile apps. Their empirical study were

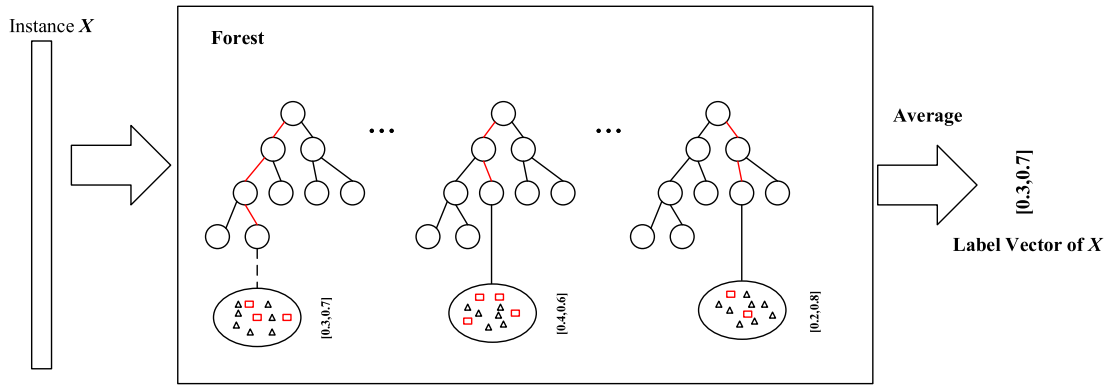

Fig. 1. Decision process of a forest.

conducted on 14 apps with 42 543 commits and the experimental results showed that Naive Bayes achieved the best performance compared with other classifiers and some ensemble learning techniques.

Different from these studies that only used traditional classifications to construct prediction models and did not conduct any conversion on the original feature set, in this work, we propose a new classification model that also performs feature representation learning in the process of the model construction.

C. Deep Forest in Software Defect Prediction

Recently, a novel decision tree ensemble approach was proposed by Zhou *et al.* [16], called gcForest, which consists of a multigrained scanning and a cascade structure. The multigrained scanning is used to enhance representation learning. The cascade structure is inspired by the layer-by-layer processing in deep neural networks. Once the model was proposed, it was used by researchers in the defect-prediction study for traditional software. Zhou *et al.* [24] made the first attempt to use deep forest for defect prediction using the structure with four random forests. Zheng *et al.* [25] proposed an improved deep forest, which used data augmentation technique to replace multigrained scanning for randomly extracting features from defective and clean software units. The experimental results on Eclipse dataset showed that their method could get higher performance than baseline methods.

Different from the two studies, we are the first to introduce deep forest model into defect-prediction task of Android mobile apps and use the different deep forest structure.

III. Our Method

Deep forest is a novel decision tree ensemble approach with a cascade structure, which can take advantage of the useful information of features [16]. It consists of a multigrained scanning and a cascade forest. Each level of cascade structure has many forests, which include many trees. For each tree, the best Gini value is chosen to construct it. To be more specific, for each feature in the data, the Gini value is calculated across all instances. Then, the feature that minimizes the Gini value after the partition is selected to split all the data. Fig. 1 shows the decision process for each forest. The red line illustrates the decision path. Given an instance X, each tree of forest will produce the distribution estimation of the class to which it belongs. Then, the forest calculates the final estimate by averaging the outputs of all trees in this forest.

Meanwhile, the multigrained scanning method is designed to deal with high-dimensional features, such as sequence data and image data. However, as our defect data for Android mobile apps include low-dimensional features, the multigrained scanning process is not necessary for our task. Due to this reason, we employ an SDF model, which only contains a cascade structure for feature representation learning and classification without the multigrained scanning process. More specifically, the original vector of the instance is transformed into the new class vector with this cascade structure. Then, the new class vector is concatenated with the original ones to improve the quality of original features. In addition, in order to make full use of the diversity of features, each level of structure consists of four completely random tree forests and four random forests, in which each forest contains 500 trees at the same time, following the same parameter settings as previous work [16]. Fig. 2 demonstrates the cascade structure of our model. First, we input the feature vector of a commit into the cascade structure and get its processing result by the first level. Second, the next level of cascade receives the input that concatenates the original feature vector with the results processed by its preceding level. Then, we output its processing result to the next level. In the last level of the structure, only the class vector produced by the nearest level is used for classification. The deep forest model has the following advantages: 1) Compared with the complex hyperparameters of deep neural networks, the number of cascade levels can be automatically determined; and 2) its performance is insensitive to parameter settings.

IV. Experimental Setup

A. Datasets

In our work, we employ 10 Android mobile apps provided by Catolino *et al.* [12] as our benchmark dataset to evaluate the performance of our proposed SDF model. In order to make the study corpus be representative and diversity, Catolino *et al.*

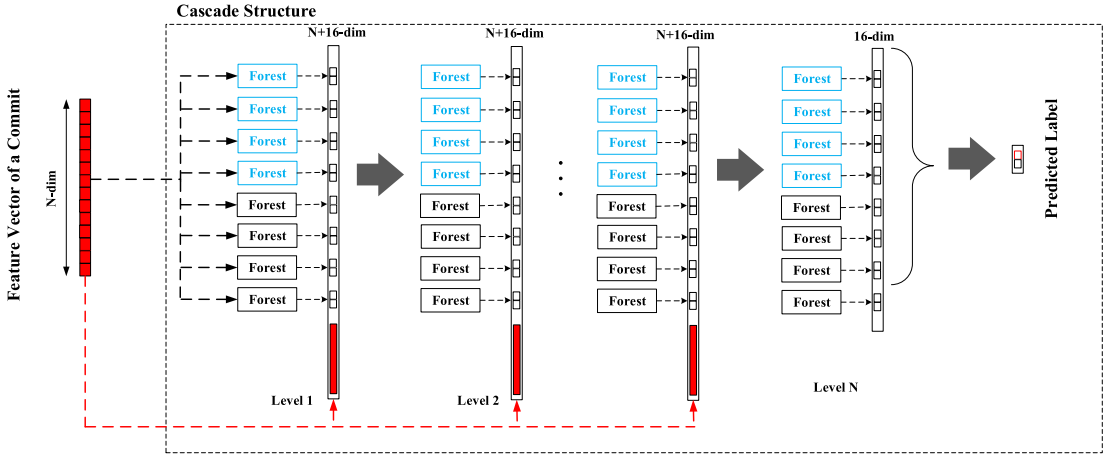

Fig. 2. Cascade structure of our SDF model for JIT defect prediction.

TABLE I
BASIC STATISTIC INFORMATION OF THE 10 APPS

App	# Commits	# Defective	# Clean	% Ratio
Alfresco	1004	214	790	21%
Sync	209	62	147	30%
Keyboard	2971	819	2152	28%
Wallpaper	588	94	494	16%
ChatSecure	2579	853	1726	33%
Facebook	548	180	368	33%
Kiwix	1373	350	1023	25%
Own Cloud	3700	830	2870	22%
Page Turner	164	23	141	14%
Notify Reddit	222	60	162	27%

selected open-source Android mobile apps that are developed for different application domains and have different scales as the context of the study. This is able to reduce the threats related to the generalization of the experimental results. These apps are briefly described as follows: Alfresco is a widely used open-source enterprise knowledge management tool. Android Sync (Sync) is the Android synchronization manager that stores and synchronizes personal information locally. AnySoftKeyboard (Keyboard) provides a screen keyboard for Android devices, which supports multiple languages through external software packages. Android Wallpaper (Wallpaper) provides many beautiful pictures for users, such as dynamic and static wallpapers. ChatSecure Android (ChatSecure) is a communication encryption app, which supports XMPP and OTR encryption. Facebook Android SDK (Facebook) provides a solution that can integrate Facebook in Android apps. Kiwix is a web content reader that allows Wikipedia reading offline. Own Cloud Android (Own Cloud) is a cloud storage software for Android apps with many functions. Page Turner is an ebook reader, which synchronizes the reading process between multiple devices. Notify Reddit allows users to obtain notifications from their Android wearable devices.

Table I describes the basic statistic information of these apps, including the number of commits (# Commit), the number of defective commits (# Defective), the number of nondefective commits (# Clean), and the ratio of defective commits (% Ratio). Each commit in the defect data is represented by six features [12]. The brief description of these features is shown in Table II.

TABLE II
DESCRIPTION OF FEATURES

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

B. Performance Indicators

As our JIT defect-prediction task on Android mobile apps is to determine whether a code commit instance is defective or not, some indicators for the typical binary classification task can be used to evaluate the effectiveness of the proposed SDF method and the methods for comparison. In this work, we use a total of three indicators, including F-measure, MCC, and AUC that are commonly used in previous defect-prediction studies, such as F-measure [26], [27], MCC [28], [29], and AUC [14], [30]–[32]. The details of these indicators are described as follows.

The first indicator is F-measure, which is the weighted harmonic average of Precision and Recall. It is expressed by the following formula:

$$\text{F-measure} = \frac{(1+\theta^2) \times \text{Precision} \times \text{Recall}}{\theta \times \text{Precision} + \text{Recall}} \quad (1)$$

where Precision = TP / (TP + FP) and Recall = TP / (TP + FN). True positive (TP) denotes the number of a commit labeled as defective that is predicted as defective. False positive (FP) denotes the number of a commit labeled as nondefective that is predicted as defective. False negative (FN) denotes the number of a commit labeled as defective that is predicted as nondefective. θ is a tradeoff parameter between Precision and Recall. In this article, we set θ as 2, which are used by previous works [30], [33].

The second indicator is MCC, which is a special case of the Pearson correlation coefficient. Its formula is expressed as follows:

$$\mathrm{MCC} = \frac{\mathrm{TP} \times \mathrm{TN} - \mathrm{FP} \times \mathrm{FN}}{\sqrt{(\mathrm{TP}+\mathrm{FP})(\mathrm{TP}+\mathrm{FN})(\mathrm{TN}+\mathrm{FP})(\mathrm{TN}+\mathrm{FN})}} \tag{2}$$

where true negative (TN) denotes the number of a commit labeled as nondefective that is predicted as nondefective. MCC is a comprehensive indicator considering TP, TN, FP, and FN.

Receiver operating characteristic (ROC) curve is independent of the specific threshold of the classifier decision, which takes the FP rate (FPR = FP / (TN + FP)) and TP rate (TPR = TP /(TP + FN)) as horizontal axis and vertical axis, respectively. To a certain degree, ROC can reflect the effect of different classifiers but is not intuitive enough. Thus, we use the area under the ROC curve, namely AUC, as our third indicator. AUC reflects the ability of classification intuitively.

In terms of the defect-prediction task, as researchers and practitioners aim to identify as many defective code commits as possible, Recall should be more important than Precision. F-measure with parameter $\theta = 2$ used in our work is consistent with this goal as it emphasizes more on the Recall indicator. In addition, the number of defective code commit instances is fewer than that of clean ones. In such case, using MCC and AUC are suitable because they are appropriate indicators to evaluate model performance on imbalanced data [28], [34]. To sum up, using the combination of the three indicators can make our performance evaluation more comprehensive.

F-measure ranges from 0 to 1. The larger F-measure value means the better performance. MCC ranges from -1 to 1. MCC $= -1$ represents the completely inconsistent between the predicted labels by the model and the actual labels. MCC $=$ 1 represents the completely consistent between the predicted labels by the model and the actual labels. MCC $= 0$ means that the performance evaluation is equal to random prediction. AUC ranges from 0 to 1. AUC $= 0.5$ represents that the performance evaluation is equal to random prediction. AUC < 0.5 means that the performance evaluation is worse than random prediction. The closer it gets to 1, the better the performance.

C. Data Partition

To generate the training set and test set, we use the stratified sampling method to partition the data. More specifically, for the defect data of one app, we select half of the commit instances with label defective and half of the commit instances with label clean as the training set, and the remaining ones are treated as the test set. The stratified sampling enables the proportions of commit instances of two labels in the training set and the test set are the same as that in the original data. In order to eliminate the influence of random partition on experimental results, we repeat the data partition process 30 times to reduce the bias of the randomness. In this work, we report the average value and standard deviation of the 30 results for each indicator. For JIT defect-prediction task, the timewise cross validation is the most appropriate way to generate the training set and test set. This cross validation considers the commit time information by ranking the commit changes in a chronological order according to their date. However, the defect data provided by the original authors only consist of the feature sets without the time information. As previous studies [35], [36] about JIT defect prediction on traditional software projects have shown that the results and conclusion based on the timewise cross validation and multifold cross validation are basically the same. Thus, we choose the latter one to conduct our experiments. This is an acceptable choice in the absence of commit time information.

TABLE III
CLIFF'S DELTA WITH RESPECT TO EFFECTIVENESS LEVEL

Cliff's Delta	Effectiveness Level
$\lvert\delta\rvert < 0.147$	Negligible(N)
$0.147 \leq \lvert\delta\rvert < 0.33$	Small(S)
$0.33 \leq \lvert\delta\rvert < 0.474$	Medium(M)
$\lvert\delta\rvert > 0.474$	Large(L)

D. Statistic Test

We first apply Wilcoxon signed-rank test and Cliff's delta (δ) to analyze performance differences between method pairs (i.e., our SDF model and each comparative method) at significance level $\alpha = 0.05$. Wilcoxon signed-rank test is a nonparametric approach, which does not consider the distribution of data [37]. If the p-value of Wilcoxon signed-rank test is lower than 0.05, it means that the performance of two methods is significant, otherwise, not significant. Cliff's delta is a nonparametric measure of effectiveness between two methods [38]. The value of δ ranges from -1 to 1. The greater absolute value of δ means that it has more completely nonoverlapping between two methods. Table III describes different values of δ with respect to level.

In addition, in order to conduct significance analysis among our SDF model and all the comparative methods, we employ a state-of-the-art statistic test method, called Scott–Knott Effect Size Difference (Scott–Knott ESD) [39]. The Scott–Knott ESD uses a clustering method to divide all the methods into many groups with significant difference (significance level $\alpha = 0.05$). Compared with original Scott–Knott, this test corrects the nonnormal distribution of the input data by applying log-transforming for each treatment and merges groups, which have negligible effect size of statistical differences into one group. In our work, we employ a two-phase Scott–Knott ESD test. Fig. 3 illustrates the analysis process of this test. In the first phase, after obtaining the indicator results of 30 rounds for all methods on each mobile app as showed on the top of each blue dotted rectangle, we input these results into the Scott–Knott ESD test to get the ranking results of each method on each app as showed in the bottom-right corner of the blue dotted rectangle. In the second phase, we input the ranking results produced from the previous phase into the Scott–Knott ESD test again to output the overall ranking of each method across all apps and their corresponding groups as showed on the right-hand side of the figure.

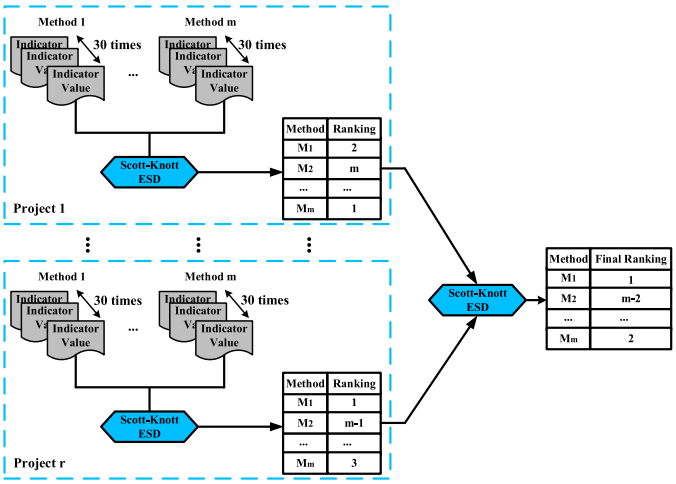

Fig. 3. Analysis process of the Scott–Knott ESD test.

V. Experimental Results

A. RQ1: How Effective is Our SDF Model Compared With Other Classification Models?

Motivation: As previous defect-prediction studies [40], [41] on traditional software stated that different classification models have significant performance difference and suggested that new classification models should be explored for further performance improvement. This question is designed to explore whether our SDF model that can be treated an ensemble classifier can achieve better performance than some typical classification models. As one function of our SDF model is to perform the classification task, this question can be used to investigate the superiority of our SDF method from the aspect of classification.

Methods: To answer this question, we choose eight commonly used classification models as baseline methods, including Naive Bayes (NB), Nearest Neighbors (NN), Logistic Regression (LR), Decision Tree (DT), Random Forest (RF), Support Vector Machine (SVM), Bagging, and AdaBoost. NB is a simple probability-based classifier that applies the Bayesian theorem under the assumption of strong independence between features. NN is an instance-based classifier, which uses the label of the closest training sample instance as the classification output. LR is a function-based classifier and a generalized linear regression analysis model. DT is a tree-based classifier that classifies instances by constructing a tree in which internal nodes represent a partition of features and leaf nodes represent the label to which it belongs. RF is an ensemble technique based on decision tree whose result is determined by all the trees in the forest. SVM is a statistical learning theory-based classifier that makes the original instances become linearly separable by finding a hyperplane in a high-dimensional space. Bagging is a typical ensemble technique based on bootstrap sampling. AdaBoost builds multiple weak classifiers with different weights of instances and integrates them as a strong classifier by using an adaptive approach. Before running these classification models and our SDF model, we first normalize the original data with z-score method [31]. The aim is to eliminate the negative effects of the numerical discrepancy between different features. After applying z-score normalization, the value of each feature is constrained to a mean of 0 and a standard deviation of 1.

Results: Tables IV –VI report the average F-measure, MCC, and AUC values and the corresponding standard deviation of 30 random runnings for SDF and eight comparative classifiers on ten Android mobile apps. The best average indicator values are in bold.

From Table IV, we can observe that, in terms of F-measure, SDF model obtains the best performance on five out of ten Android mobile apps. The average F-measure value by our SDF model across the ten apps achieves improvements by 39.9%, 1.0%, 152.4%, 26.2%, 129.2%, 6.5%, and 2.4% compared with NB, NN, LR, RF, SVM, Bagging, and AdaBoost, respectively. Although the average F-measure by SDF is nearly the same as that by DT, our model outperforms DT on seven apps. From Table V, we can observe that, in terms of MCC, SDF model obtains the best performance on seven out of ten Android mobile apps. The average MCC value by our SDF model across the ten apps achieves improvements by 94.9%, 35.7%, 70.8%, 44.8%, 10.5%, 37.6%, 11.4%, and 27.7% compared with NB, NN, LR, DT, RF, SVM, Bagging, and AdaBoost, respectively. From Table VI, we can observe that, in terms of AUC, SDF model obtains the best performance on seven out of ten Android mobile apps. The average AUC value by our SDF model across the 10 apps achieves improvements by 13.6%, 4.3%, 14.0%, 6.2%, 4.3%, 12.0%, 2.1%, and 4.4% compared with NB, NN, LR, DT, RF, SVM, Bagging, and AdaBoost, respectively.

In addition, from Tables IV to VI, we can find that the performance of the our SDF model and the comparative methods receive the worst performance on Wallpaper app and Page Turner app while achieve better performance on Sync app, ChatSecure app, Facebook app, and Notify Reddit app in most cases in terms of the three indicators. According to Table I, we can observe that the ratios of defective commits for Wallpaper app and Page Turner app are the lowest (nearly 15%) among the studied apps while the latter four apps have relatively higher defective ratios (larger than 25%). The magnitude of the defective ratio corresponds to the degree of class imbalance of the defect data, which is one factor affecting the performance of the defect-prediction model. For the previous analysis, the performance fluctuation of the methods among different Android mobile apps may be related to the degree of class imbalance of the defect data. In other words, the methods tend to achieve worse performance on defect data that are more imbalanced.

Tables VII –IX present the p-value of Wilcoxon signed-rank test and the Cliff delta (δ) for F-measure, MCC, and AUC, respectively. Noting that, in most of the cases, our SDF model is significantly superior to these baseline methods with large and medium effectiveness level in terms of all three indicators.

Fig. 4(a)–(c) visualizes the statistic test results of Scott–Knott ESD for F-measure, MCC, and AUC, respectively. These figures illustrate that our SDF model ranks the first and is significantly superior to all baseline methods in terms of all three indicators.

Besides the performance analysis, we also perform efficiency analysis of our SDF model. For this purpose, we record the execution time of one data partition of SDF and the baseline

TABLE IV
AVERAGE *F*-MEASURE OF SDF AND OTHER CLASSIFICATION MODELS

App	NB	NN	LR	DT	RF	SVM	Bagging	AdaBoost	SDF
Alfresco	0.339(0.156)	**0.418**(0.050)	0.208(0.052)	0.402(0.080)	0.360(0.092)	0.183(0.065)	0.376(0.053)	0.396(0.086)	0.412(0.076)
Sync	0.507(0.096)	0.454(0.077)	0.490(0.077)	0.510(0.092)	0.507(0.105)	0.386(0.116)	0.456(0.068)	0.513(0.103)	**0.526**(0.083)
Keyboard	0.220(0.139)	0.406(0.029)	0.186(0.033)	0.378(0.124)	0.211(0.084)	0.152(0.037)	0.393(0.038)	0.357(0.148)	**0.453**(0.114)
Wallpaper	0.227(0.077)	0.266(0.061)	0.043(0.026)	0.258(0.065)	0.006(0.015)	0.022(0.027)	0.158(0.041)	**0.269**(0.073)	0.072(0.071)
ChatSecure	0.281(0.244)	0.482(0.050)	0.132(0.030)	0.465(0.127)	0.446(0.168)	0.231(0.024)	0.477(0.053)	0.439(0.140)	**0.501**(0.125)
Facebook	0.209(0.244)	0.466(0.062)	0.119(0.064)	0.518(0.093)	0.534(0.149)	0.162(0.085)	0.431(0.071)	0.482(0.103)	**0.592**(0.087)
Kiwix	0.256(0.264)	**0.419**(0.051)	0.033(0.024)	0.413(0.059)	0.131(0.101)	0.051(0.021)	0.387(0.049)	0.394(0.063)	0.408(0.071)
Own Cloud	0.152(0.204)	**0.484**(0.024)	0.032(0.018)	0.443(0.049)	0.332(0.050)	0.14(0.045)	0.457(0.022)	0.421(0.063)	0.467(0.051)
Page Turner	0.270(0.111)	0.246(0.126)	0.036(0.053)	0.316(0.112)	0.175(0.152)	0.017(0.045)	0.246(0.105)	**0.321**(0.129)	0.147(0.143)
Notify Reddit	0.573(0.106)	0.561(0.067)	0.403(0.086)	0.547(0.076)	**0.659**(0.107)	0.507(0.106)	0.604(0.077)	0.544(0.097)	**0.659**(0.069)
Average	0.303(0.128)	0.420(0.092)	0.168(0.153)	**0.425**(0.087)	0.336(0.194)	0.185(0.149)	0.398(0.117)	0.414(0.081)	0.424(0.174)

TABLE V
AVERAGE MCC OF SDF AND OTHER CLASSIFICATION MODELS

App	NB	NN	LR	DT	RF	SVM	Bagging	AdaBoost	SDF
Alfresco	0.196(0.164)	0.223(0.118)	0.281(0.049)	0.241(0.129)	0.378(0.058)	0.264(0.059)	0.307(0.090)	0.301(0.119)	**0.378**(0.053)
Sync	0.410(0.084)	0.25(0.088)	**0.453**(0.076)	0.249(0.156)	0.410(0.097)	0.364(0.083)	0.314(0.062)	0.270(0.142)	0.410(0.059)
Keyboard	0.150(0.048)	0.155(0.067)	0.171(0.019)	0.176(0.087)	0.239(0.044)	0.220(0.023)	0.224(0.062)	0.206(0.089)	**0.287**(0.060)
Wallpaper	0.161(0.079)	**0.123**(0.071)	0.053(0.053)	0.081(0.097)	0.003(0.045)	0.041(0.059)	0.112(0.054)	0.112(0.092)	0.083(0.076)
ChatSecure	0.115(0.072)	0.172(0.066)	0.157(0.021)	0.153(0.178)	0.249(0.119)	0.282(0.025)	0.219(0.067)	0.209(0.146)	**0.288**(0.100)
Facebook	0.022(0.058)	0.197(0.080)	0.116(0.073)	0.188(0.143)	0.257(0.171)	0.179(0.065)	0.230(0.079)	0.221(0.117)	**0.297**(0.131)
Kiwix	0.023(0.074)	0.209(0.059)	0.061(0.052)	0.168(0.113)	0.155(0.085)	0.151(0.031)	0.253(0.064)	0.177(0.097)	**0.295**(0.049)
Own Cloud	0.076(0.062)	0.344(0.035)	0.073(0.035)	0.306(0.073)	0.360(0.024)	0.232(0.036)	0.372(0.034)	0.359(0.053)	**0.406**(0.047)
Page Turner	0.219(0.160)	0.182(0.133)	0.002(0.085)	0.203(0.149)	0.168(0.169)	0.017(0.091)	**0.235**(0.144)	0.207(0.161)	0.123(0.152)
Notify Reddit	0.188(0.225)	0.384(0.097)	0.413(0.064)	0.336(0.148)	0.530(0.085)	0.455(0.079)	0.467(0.105)	0.316(0.192)	**0.471**(0.104)
Average	0.156(0.107)	0.224(0.078)	0.178(0.147)	0.210(0.071)	0.275(0.143)	0.221(0.127)	0.273(0.092)	0.238(0.070)	**0.304**(0.117)

TABLE VI
AVERAGE AUC OF SDF AND OTHER CLASSIFICATION MODELS

App	NB	NN	LR	DT	RF	SVM	Bagging	AdaBoost	SDF
Alfresco	0.573(0.055)	0.611(0.054)	0.578(0.020)	0.611(0.055)	0.641(0.037)	0.569(0.025)	0.629(0.038)	0.628(0.051)	**0.657**(0.033)
Sync	0.686(0.043)	0.623(0.043)	0.690(0.035)	0.623(0.078)	0.685(0.050)	0.642(0.045)	0.643(0.027)	0.635(0.071)	**0.690**(0.033)
Keyboard	0.550(0.026)	0.577(0.031)	0.553(0.008)	0.578(0.044)	0.572(0.028)	0.554(0.012)	0.604(0.027)	0.585(0.047)	**0.633**(0.035)
Wallpaper	**0.566**(0.036)	0.562(0.035)	0.510(0.010)	0.539(0.052)	0.501(0.005)	0.506(0.009)	0.540(0.019)	0.557(0.047)	0.520(0.025)
ChatSecure	0.532(0.027)	0.586(0.031)	0.539(0.008)	0.560(0.071)	0.594(0.048)	0.583(0.010)	0.606(0.027)	0.584(0.057)	**0.623**(0.040)
Facebook	0.507(0.021)	0.596(0.036)	0.531(0.024)	0.591(0.068)	0.613(0.077)	0.549(0.025)	0.606(0.035)	0.604(0.055)	**0.641**(0.063)
Kiwix	0.507(0.025)	0.604(0.028)	0.509(0.008)	0.581(0.054)	0.542(0.031)	0.519(0.008)	0.614(0.026)	0.583(0.043)	**0.630**(0.020)
Own Cloud	0.513(0.017)	0.669(0.014)	0.509(0.006)	0.644(0.033)	0.627(0.019)	0.552(0.016)	0.669(0.013)	0.654(0.025)	**0.678**(0.017)
Page Turner	0.587(0.065)	0.579(0.061)	0.503(0.021)	0.593(0.066)	0.562(0.064)	0.504(0.019)	0.587(0.050)	**0.598**(0.071)	0.545(0.056)
Notify Reddit	0.580(0.097)	0.693(0.047)	0.658(0.037)	0.667(0.068)	**0.766**(0.054)	0.699(0.048)	0.731(0.051)	0.660(0.090)	0.743(0.052)
Average	0.560(0.051)	0.610(0.040)	0.558(0.063)	0.599(0.037)	0.610(0.072)	0.568(0.058)	0.623(0.048)	0.609(0.032)	**0.636**(0.062)

TABLE VII
p-VALUE AND CLIFF'S DELTA (δ) FOR SDF AND OTHER CLASSIFICATION MODELS IN TERMS OF *F*-MEASURE

App	SDF vs. NB		SDF vs. NN		SDF vs. LR		SDF vs. DT		SDF vs. RF		SDF vs. SVM		SDF vs. Bagging		SDF vs. AdaBoost	
	p-value	δ	p-value	δ	p-value	δ	p-value	δ	p-value	δ	p-value	δ	p-value	δ	p-value	δ
Alfresco	1.8e-02	0.35(M)	8.1e-01	-0.07(N)	1.7e-06	1.00(L)	6.0e-01	0.03(N)	5.5e-02	0.29(S)	1.7e-06	1.00(L)	6.0e-02	0.28(S)	4.1e-01	0.06(N)
Sync	4.1e-01	0.07(N)	3.0e-03	0.45(M)	5.5e-02	0.23(S)	6.6e-01	0.11(N)	6.6e-01	0.06(N)	4.1e-05	0.65(L)	4.4e-03	0.47(M)	8.1e-01	0.07(N)
Keyboard	4.3e-06	0.80(L)	7.2e-02	0.21(S)	1.7e-06	1.00(L)	6.9e-02	0.29(S)	1.7e-06	0.92(L)	1.7e-06	1.0(L)	4.1e-02	0.28(S)	1.8e-02	0.36(M)
Wallpaper	1.4e-05	-0.9(L)	2.9e-06	-0.92(L)	2.4e-02	0.30(S)	2.4e-06	-0.91(L)	1.6e-05	0.75(L)	6.3e-05	0.54(L)	6.3e-05	-0.78(L)	2.1e-06	-0.93(L)
ChatSecure	1.3e-03	0.53(L)	5.9e-01	0.01(N)	1.7e-06	1.00(L)	2.6e-01	0.18(S)	1.0e-01	0.28(S)	1.7e-06	1.00(L)	4.5e-01	0.03(N)	7.5e-02	0.29(S)
Facebook	1.2e-05	0.66(L)	1.1e-05	0.72(L)	1.7e-06	1.00(L)	1.7e-02	0.46(M)	1.9e-01	0.11(N)	1.7e-06	1.00(L)	2.4e-06	0.80(L)	3.1e-04	0.59(L)
Kiwix	3.4e-03	0.25(S)	4.7e-01	-0.15(S)	1.7e-06	1.00(L)	5.9e-01	-0.11(N)	2.1e-06	0.96(L)	1.7e-06	1.00(L)	1.5e-01	0.18(S)	3.4e-01	0.12(N)
Own Cloud	1.0e-05	0.62(L)	1.1e-01	-0.26(S)	1.7e-06	1.00(L)	1.5e-01	0.21(S)	1.9e-06	0.97(L)	1.7e-06	1.00(L)	3.2e-01	0.09(N)	2.0e-02	0.41(M)
Page Turner	4.7e-03	-0.49(L)	2.7e-02	-0.41(M)	8.2e-04	0.47(M)	3.9e-04	-0.62(L)	3.7e-01	-0.09(N)	2.4e-04	0.55(L)	9.8e-03	-0.36(M)	5.3e-04	-0.62(L)
Notify Reddit	4.7e-03	0.41(M)	2.4e-04	0.66(L)	1.7e-06	0.99(L)	1.2e-04	0.68(L)	6.4e-01	-0.05(N)	8.5e-06	0.77(L)	4.5e-02	0.35(M)	2.2e-04	0.65(L)

TABLE VIII
p-VALUE AND CLIFF'S DELTA (δ) FOR SDF AND OTHER CLASSIFICATION MODELS IN TERMS OF MCC

App	SDF vs. NB		SDF vs. NN		SDF vs. LR		SDF vs. DT		SDF vs. RF		SDF vs. SVM		SDF vs. Bagging		SDF vs. AdaBoost	
	p-value	δ	p-value	δ	p-value	δ	p-value	δ	p-value	δ	p-value	δ	p-value	δ	p-value	δ
Alfresco	7.0e-06	0.90(L)	5.2e-06	0.79(L)	1.4e-05	0.82(L)	1.1e-05	0.73(L)	8.0e-01	-0.0(N)	3.2e-06	0.86(L)	7.2e-04	0.53(L)	8.7e-03	0.41(M)
Sync	8.8e-01	0.01(N)	3.2e-06	0.86(L)	3.9e-02	-0.31(S)	1.1e-04	0.65(L)	8.6e-01	-0.01(N)	5.5e-02	0.34(M)	1.1e-05	0.74(L)	3.3e-04	0.62(L)
Keyboard	3.5e-06	0.92(L)	4.3e-06	0.84(L)	3.5e-06	0.84(L)	3.1e-05	0.74(L)	6.0e-03	0.49(L)	1.1e-04	0.67(L)	8.3e-04	0.53(L)	1.5e-03	0.58(L)
Wallpaper	4.9e-04	-0.53(L)	7.2e-02	-0.29(S)	6.9e-02	0.21(S)	8.9e-01	-0.04(N)	2.3e-04	0.65(L)	1.0e-02	0.34(M)	2.6e-02	-0.28(S)	1.8e-01	-0.2(S)
ChatSecure	5.8e-06	0.83(L)	6.9e-05	0.64(L)	1.4e-05	0.8(L)	2.1e-03	0.48(L)	2.4e-01	0.23(S)	7.2e-01	0.12(N)	4.7e-03	0.42(M)	3.3e-02	0.30(S)
Facebook	1.9e-06	0.93(L)	6.0e-03	0.48(L)	2.0e-05	0.73(L)	1.0e-02	0.40(M)	6.3e-01	0.13(N)	2.2e-04	0.53(L)	4.1e-02	0.36(M)	1.9e-02	0.35(M)
Kiwix	1.9e-06	0.99(L)	8.2e-05	0.72(L)	1.7e-06	1.00(L)	7.7e-06	0.79(L)	2.6e-06	0.88(L)	1.7e-06	0.97(L)	3.2e-02	0.39(M)	6.9e-05	0.74(L)
Own Cloud	1.7e-06	1.00(L)	6.3e-05	0.71(L)	1.7e-06	1.00(L)	1.0e-05	0.8(L)	6.9e-05	0.65(L)	1.7e-06	0.99(L)	1.7e-03	0.53(L)	1.7e-03	0.50(L)
Page Turner	3.5e-02	-0.34(M)	1.3e-01	-0.24(S)	2.1e-03	0.53(L)	9.0e-02	-0.29(S)	2.3e-01	-0.14(N)	1.0e-02	0.41(M)	6.0e-03	-0.4(M)	8.6e-02	-0.3(S)
Notify Reddit	6.3e-05	0.69(L)	4.1e-03	0.51(L)	1.6e-02	0.48(L)	1.7e-03	0.57(L)	5.2e-02	-0.34(M)	6.4e-01	0.14(N)	8.0e-01	-0.0(N)	1.4e-03	0.56(L)

TABLE IX
p-VALUE AND CLIFF'S DELTA (δ) FOR SDF AND OTHER CLASSIFICATION MODELS IN TERMS OF AUC

App	SDF vs. NB		SDF vs. NN		SDF vs. LR		SDF vs. DT		SDF vs. RF		SDF vs. SVM		SDF vs. Bagging		SDF vs. AdaBoost	
	p-value	δ	p-value	δ	p-value	δ	p-value	δ	p-value	δ	p-value	δ	p-value	δ	p-value	δ
Alfresco	3.2e-06	0.87(L)	1.7e-03	0.49(L)	1.7e-06	1.00(L)	1.6e-04	0.52(L)	1.6e-01	0.24(S)	1.7e-06	0.99(L)	1.2e-02	0.37(M)	2.1e-02	0.35(M)
Sync	8.6e-01	0.02(N)	1.1e-05	0.78(L)	8.8e-01	0.04(N)	1.1e-03	0.55(L)	8.3e-01	0.01(N)	2.6e-04	0.59(L)	5.3e-05	0.68(L)	3.2e-03	0.48(L)
Keyboard	1.7e-06	0.93(L)	9.3e-06	0.74(L)	1.7e-06	1.00(L)	1.4e-04	0.69(L)	1.0e-05	0.80(L)	1.7e-06	1.00(L)	2.3e-03	0.45(M)	8.3e-04	0.57(L)
Wallpaper	2.8e-05	-0.78(L)	5.8e-05	-0.72(L)	3.3e-02	0.24(S)	4.7e-02	-0.38(M)	3.1e-05	0.68(L)	4.9e-04	0.44(M)	2.6e-04	-0.61(L)	7.2e-04	-0.46(M)
ChatSecure	2.6e-06	0.89(L)	4.5e-04	0.57(L)	2.9e-06	0.90(L)	5.7e-04	0.56(L)	5.0e-02	0.35(M)	2.8e-04	0.74(L)	4.7e-02	0.35(M)	1.2e-02	0.41(M)
Facebook	1.9e-06	0.94(L)	7.7e-03	0.46(M)	2.6e-06	0.85(L)	1.0e-02	0.41(M)	2.5e-01	0.23(S)	7.7e-06	0.76(L)	2.4e-02	0.40(M)	1.7e-02	0.36(M)
Kiwix	1.7e-06	0.99(L)	2.6e-03	0.51(L)	1.7e-06	1.00(L)	2.2e-05	0.70(L)	1.7e-06	0.99(L)	1.7e-06	1.00(L)	1.9e-02	0.33(S)	1.4e-05	0.70(L)
Own Cloud	1.7e-06	1.00(L)	4.5e-02	0.18(S)	1.7e-06	1.00(L)	4.5e-05	0.75(L)	1.7e-06	0.99(L)	1.7e-06	1.00(L)	2.6e-02	0.26(S)	7.7e-04	0.56(L)
Page Turner	2.7e-02	-0.39(M)	5.5e-02	-0.36(M)	9.6e-04	0.57(L)	8.7e-03	-0.43(M)	1.9e-01	-0.12(N)	3.4e-03	0.42(M)	7.7e-03	-0.37(M)	9.3e-03	-0.44(M)
Notify Reddit	9.3e-06	0.84(L)	1.5e-03	0.58(L)	2.8e-05	0.82(L)	9.3e-04	0.63(L)	1.5e-01	-0.23(S)	8.7e-03	0.51(L)	4.1e-01	0.14(N)	6.6e-04	0.63(L)

TABLE X
RUNNING TIME OF THE SDF MODEL ON EACH ANDROID MOBILE APP (IN SECONDS)

App	Alfresco	Sync	Keyboard	Wallpaper	ChatSecure	Facebook	Kiwix	Own Cloud	Page Turner	Notify Reddit
Time	101	72	328	52	169	135	142	418	68	85

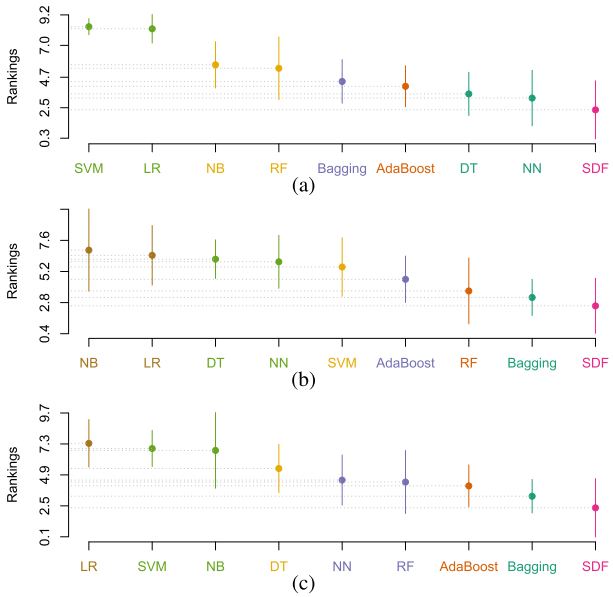

Fig. 4. Scott–Knott ESD test for DF and other classification models. (a) F-measure. (b) MCC. (c) AUC.

methods, including the feature representation learning (for our SDF model), model training, and application time. We found that the eight baseline methods only took a few seconds to complete the JIT defect-prediction task on all Android mobile apps. Thus, we do not report their execution time. Table X reports the running time of our SDF model on each Android mobile app. From the table, we can see that our SDF model needs less than 3 min to run one data partition on most Android mobile apps except on Keyboard app and Own Cloud app. The reason that our SDF model speeds more time than the baseline methods is that SDF consists of an additional feature representation learning process that needs to construct a multiple-layer tree structure. Although the computational cost of our SDF model is larger than the baseline methods, the performance of our proposed SDF model performs much better. In this work, we conduct all experiments on Windows 10 (Memory: 16 GB, Processor: Intel(R) Core(TM) i7-4790 CPU @ 3.60 GHz $\times$ 4). As computer configuration continues to upgrade, the running time of SDF model will be greatly reduced. As a result, the efficiency of our SDF model will be improved and SDF can provide timely feedback with high-end computing devices for JIT defect prediction on Android mobile apps.

To sum up, our SDF model is superior to the compared eight typical classification models, especially on the indicators of MCC and AUC. It means that, as an emerging classification model, our SDF is more suitable for JIT defect prediction on Android mobile apps to achieve the better performance.

B. RQ2: Can the Feature Processing by Typical Feature Extraction and Feature Subset Selection Methods Further Improve the Performance of Our SDF Model?

Motivation: As the initial features may not well represent defect data, previous defect-prediction studies for traditional software applied some feature extraction or feature subset selection methods to the data for the better performance [42]–[44]. This question is designed to investigate whether the defect data for Android mobile apps processed by feature extraction and subset selection methods can further enhance the performance of our SDF model. As another function of our SDF model is to learn better feature representation, this question can be used to investigate the superiority of our SDF method from the aspect of feature representation learning.

Methods: To answer this question, we employ four typical feature extraction methods, i.e., Principal Component Analysis (PCA) method [45], Kernel-based PCA (KPCA) method [46], Locally Linear Embedding (LLE) [47], Isomap Embedding (IE) [48], and two filter-based feature subset selection methods, i.e., Correlation-based feature subset selection (Cor) [49] and Consistency-based feature subset selection (Con) [50] for comparison. PCA is a commonly used linear dimensionality reduction technique, which calculates the eigenvalues of its covariance matrix and the corresponding orthogonal unit eigenvectors to obtain the principal component of samples. KPCA method is the nonlinear variant of the PCA method by using kernel functions to map the features into a new feature space. The effect of KPCA is related to the selected kernel function.

TABLE XI
AVERAGE *F*-MEASURE OF SDF AND SDF COMBINING OTHER FEATURE EXTRACTION METHODS

App	lin_SDF	poly_SDF	rbf_SDF	sig_SDF	cos_SDF	PCA_SDF	LLE_SDF	IE_SDF	SDF
Alfresco	0.319(0.09)	0.332(0.08)	0.326(0.09)	0.366(0.10)	0.344(0.09)	0.319(0.09)	0.259(0.11)	0.318(0.08)	**0.412**(0.08)
Sync	0.398(0.15)	0.460(0.12)	0.402(0.12)	0.452(0.12)	0.493(0.12)	0.398(0.15)	0.322(0.18)	0.393(0.09)	**0.526**(0.08)
Keyboard	0.297(0.06)	0.297(0.07)	0.285(0.06)	0.360(0.06)	0.362(0.05)	0.297(0.06)	0.220(0.09)	0.304(0.07)	**0.453**(0.11)
Wallpaper	0.043(0.05)	**0.085**(0.07)	0.053(0.06)	0.044(0.05)	0.019(0.04)	0.043(0.05)	0.025(0.04)	0.045(0.04)	0.072(0.07)
ChatSecure	0.487(0.07)	0.469(0.08)	0.396(0.06)	**0.533**(0.08)	0.519(0.08)	0.487(0.07)	0.347(0.09)	0.398(0.06)	0.501(0.13)
Facebook	0.519(0.11)	0.459(0.12)	0.446(0.07)	0.523(0.10)	0.510(0.09)	0.519(0.11)	0.312(0.11)	0.377(0.10)	**0.592**(0.09)
Kiwix	0.326(0.09)	0.279(0.08)	0.262(0.08)	0.316(0.09)	0.337(0.11)	0.326(0.09)	0.197(0.08)	0.256(0.08)	**0.408**(0.07)
Own Cloud	0.413(0.06)	0.347(0.06)	0.342(0.07)	0.423(0.06)	0.388(0.07)	0.413(0.06)	0.258(0.07)	0.331(0.08)	**0.467**(0.05)
Page Turner	0.064(0.09)	0.104(0.13)	0.045(0.08)	0.076(0.13)	0.060(0.11)	0.064(0.09)	0.071(0.12)	0.087(0.12)	**0.147**(0.14)
Notify Reddit	0.567(0.12)	0.619(0.08)	0.590(0.13)	0.581(0.10)	0.573(0.09)	0.583(0.14)	0.540(0.10)	0.549(0.12)	**0.659**(0.07)
Average	0.343(0.17)	0.345(0.16)	0.315(0.16)	0.367(0.17)	0.360(0.18)	0.345(0.17)	0.255(0.14)	0.306(0.14)	**0.424**(0.17)

TABLE XII
AVERAGE MCC OF SDF AND SDF COMBINING OTHER FEATURE EXTRACTION METHODS

App	lin_SDF	poly_SDF	rbf_SDF	sig_SDF	cos_SDF	PCA_SDF	LLE_SDF	IE_SDF	SDF
Alfresco	0.302(0.07)	0.326(0.07)	0.312(0.06)	0.333(0.06)	0.319(0.08)	0.302(0.07)	0.201(0.08)	0.273(0.07)	**0.378**(0.05)
Sync	0.281(0.11)	0.338(0.09)	0.283(0.11)	0.345(0.09)	0.351(0.11)	0.281(0.11)	0.228(0.13)	0.291(0.09)	**0.410**(0.06)
Keyboard	0.242(0.03)	0.212(0.06)	0.232(0.03)	0.258(0.04)	0.239(0.06)	0.242(0.03)	0.150(0.06)	0.214(0.04)	**0.287**(0.06)
Wallpaper	0.041(0.07)	**0.097**(0.08)	0.055(0.08)	0.070(0.08)	0.012(0.06)	0.041(0.07)	0.029(0.07)	0.052(0.07)	0.083(0.08)
ChatSecure	0.232(0.06)	0.226(0.07)	0.227(0.04)	0.201(0.09)	0.194(0.12)	0.232(0.06)	0.153(0.04)	0.209(0.04)	**0.288**(0.10)
Facebook	**0.323**(0.07)	0.237(0.08)	0.257(0.05)	0.249(0.12)	0.314(0.09)	**0.323**(0.07)	0.119(0.08)	0.196(0.07)	0.297(0.13)
Kiwix	0.243(0.07)	0.237(0.06)	0.231(0.05)	0.252(0.07)	0.245(0.06)	0.243(0.07)	0.129(0.06)	0.197(0.05)	**0.295**(0.05)
Own Cloud	0.348(0.05)	0.305(0.06)	0.312(0.04)	0.374(0.04)	0.278(0.05)	0.348(0.05)	0.213(0.06)	0.304(0.04)	**0.406**(0.05)
Page Turner	0.042(0.11)	0.090(0.15)	0.052(0.10)	0.043(0.12)	0.054(0.11)	0.042(0.11)	0.065(0.14)	0.091(0.14)	**0.123**(0.15)
Notify Reddit	0.436(0.09)	**0.479**(0.08)	0.442(0.10)	0.376(0.12)	0.440(0.08)	0.430(0.10)	0.384(0.08)	0.411(0.08)	0.471(0.10)
Average	0.249(0.12)	0.255(0.11)	0.240(0.11)	0.250(0.11)	0.245(0.12)	0.248(0.12)	0.167(0.09)	0.224(0.10)	**0.304**(0.12)

TABLE XIII
AVERAGE AUC OF SDF AND SDF COMBINING OTHER FEATURE EXTRACTION METHODS

App	lin_SDF	poly_SDF	rbf_SDF	sig_SDF	cos_SDF	PCA_SDF	LLE_SDF	IE_SDF	SDF
Alfresco	0.614(0.04)	0.623(0.04)	0.619(0.04)	0.635(0.04)	0.622(0.04)	0.614(0.04)	0.577(0.04)	0.608(0.04)	**0.657**(0.03)
Sync	0.622(0.05)	0.651(0.05)	0.623(0.05)	0.653(0.05)	0.664(0.06)	0.622(0.05)	0.596(0.06)	0.623(0.04)	**0.690**(0.03)
Keyboard	0.592(0.02)	0.583(0.03)	0.587(0.02)	0.609(0.02)	0.603(0.03)	0.592(0.02)	0.554(0.03)	0.585(0.02)	**0.633**(0.04)
Wallpaper	0.509(0.02)	**0.524**(0.02)	0.514(0.02)	0.513(0.02)	0.502(0.01)	0.509(0.02)	0.506(0.01)	0.511(0.02)	0.520(0.03)
ChatSecure	0.612(0.03)	0.605(0.03)	0.600(0.01)	0.598(0.04)	0.592(0.06)	0.612(0.03)	0.568(0.02)	0.595(0.02)	**0.623**(0.04)
Facebook	**0.653**(0.04)	0.611(0.04)	0.619(0.02)	0.617(0.06)	0.649(0.04)	**0.653**(0.04)	0.551(0.04)	0.587(0.03)	0.641(0.06)
Kiwix	0.598(0.03)	0.588(0.03)	0.581(0.03)	0.600(0.03)	0.601(0.03)	0.598(0.03)	0.545(0.03)	0.572(0.02)	**0.630**(0.02)
Own Cloud	0.651(0.03)	0.622(0.03)	0.622(0.03)	0.658(0.02)	0.625(0.03)	0.651(0.03)	0.579(0.02)	0.617(0.03)	**0.678**(0.02)
Page Turner	0.514(0.03)	0.534(0.05)	0.514(0.03)	0.521(0.05)	0.519(0.04)	0.514(0.03)	0.523(0.05)	0.526(0.05)	**0.545**(0.06)
Notify Reddit	0.712(0.06)	0.738(0.04)	0.719(0.06)	0.688(0.06)	0.712(0.04)	0.716(0.06)	0.688(0.05)	0.700(0.05)	**0.743**(0.05)
Average	0.608(0.06)	0.608(0.06)	0.600(0.06)	0.609(0.05)	0.609(0.06)	0.608(0.06)	0.569(0.05)	0.592(0.05)	**0.636**(0.06)

Thus, we choose five common kernel functions, including linear kernel, polynomial kernel, Gaussian Radial Basic Function (RBF), sigmoid kernel, and cosine kernel. Linear kernel uses a simple linear function that is $\kappa(x_i, x_j)_{\text{linear}} = x_i^T x_j$, where x_i, x_j are input vectors. It is suitable to solve the linear separability problem. Polynomial kernel can deal with nonlinear problems and is expressed as $\kappa(x_i, x_j)_{\text{poly}} = (\gamma x_i^T x_j)^T, \gamma > 0$, where γ is a kernel parameter. RBF introduces the Euclidean distance and its expression is $\kappa(x_i, x_j)_{\text{rbf}} = \exp(-\frac{||x_i - x_j||^2}{\sigma^2})$, where $||\cdot||$ denotes the L_2 norm and σ is a kernel parameter. Sigmoid kernel employs the sigmoid function and is expressed as $\kappa(x_i, x_j)_{sig} = \tanh(\gamma x_i^T x_j + r)$, where r is a kernel parameter. Cosine kernel applies the cosine function and its expression is $\kappa(x_i, x_j)_{\cos} = \frac{<x_i, x_j>}{||x_i||||x_j||}$. LLE linearly reconstructs the neighboring instances to keep the proximity relation between the instances in the low-dimensional space. IE is a nonlinear dimension reduction method based on spectral theory that aims to preserve the geodesic distances in the low-dimensional space. Cor selects a feature subset in which the features are more relevant to the class labels but have low correlation with each other. Con selects a feature subset that has the same consistency as the original ones. We first use PCA, KPCA with the five kernel functions, LLE, IE, Cor, and Con to process the defect data separately, and then use our SDF model for defect prediction. Thus, we have total ten comparative methods, which are short for PCA_SDF, lin_SDF, poly_SDF, rbf_SDF, sig_SDF, cos_SDF, LLE_SDF, IE_SDF, Cor_SDF, and Con_SDF.

Results: Tables XI–XIII report the average *F*-measure, MCC, and AUC values and corresponding standard deviation of 30 random runnings for SDF and the baseline methods on ten Android mobile apps. The best average indicator values are in bold. Note that the results of Cor and Con are the same as

TABLE XIV
p-VALUE AND CLIFF'S DELTA (δ) FOR SDF AND SDF COMBINING OTHER FEATURE EXTRACTION METHODS IN TERMS OF F-MEASURE

App	SDF vs. lin_SDF		SDF vs. poly_SDF		SDF vs. rbf_SDF		SDF vs. sig_SDF		SDF vs. cos_SDF		SDF vs. PCA_SDF		SDF vs. LLE_SDF		SDF vs. IE_SDF	
	p-value	delta	p-value	delta	p-value	delta	p-value	delta	p-value	delta	p-value	delta	p-value	delta	p-value	delta
Alfresco	1.4e-04	0.53(L)	1.5e-05	0.50(L)	1.5e-04	0.48(L)	3.3e-02	0.21(S)	3.6e-03	0.37(M)	1.4e-04	0.53(L)	2.4e-05	0.74(L)	3.6e-04	0.59(L)
Sync	5.3e-04	0.51(L)	3.9e-02	0.30(S)	5.3e-05	0.57(L)	3.3e-02	0.36(M)	1.9e-01	0.13(N)	5.3e-04	0.51(L)	4.1e-05	0.66(L)	2.0e-05	0.71(L)
Keyboard	7.7e-06	0.76(L)	3.5e-06	0.73(L)	6.3e-06	0.80(L)	1.5e-04	0.47(M)	1.7e-03	0.46(M)	7.7e-06	0.76(L)	1.7e-06	0.90(L)	1.8e-05	0.72(L)
Wallpaper	2.0e-02	0.28(S)	2.6e-01	-0.13(N)	2.8e-01	0.18(S)	4.0e-02	0.28(S)	1.4e-03	0.61(L)	2.0e-02	0.28(S)	5.8e-04	0.51(L)	1.2e-01	0.24(S)
ChatSecure	6.4e-01	0.02(N)	1.8e-01	0.11(N)	8.3e-04	0.47(M)	6.3e-02	-0.2(S)	5.4e-01	-0.11(N)	6.4e-01	0.02(N)	4.5e-05	0.66(L)	3.6e-03	0.44(M)
Facebook	2.6e-03	0.40(M)	3.4e-05	0.69(L)	6.3e-06	0.78(L)	2.1e-04	0.39(M)	1.2e-03	0.50(L)	2.6e-03	0.40(M)	2.6e-06	0.93(L)	2.6e-06	0.89(L)
Kiwix	1.3e-04	0.52(L)	2.9e-06	0.77(L)	1.7e-06	0.80(L)	2.8e-05	0.57(L)	2.6e-04	0.42(M)	1.3e-04	0.52(L)	1.9e-06	0.93(L)	7.0e-06	0.84(L)
Own Cloud	4.9e-04	0.46(M)	1.9e-06	0.86(L)	2.9e-06	0.86(L)	1.4e-03	0.41(M)	1.6e-04	0.60(L)	4.9e-04	0.46(M)	1.7e-06	1.00(L)	6.3e-06	0.84(L)
Page Turner	6.8e-03	0.32(S)	1.0e-01	0.19(S)	1.2e-03	0.41(M)	3.0e-02	0.32(S)	4.3e-03	0.37(M)	6.8e-03	0.32(S)	1.1e-02	0.32(S)	1.2e-01	0.24(S)
Notify Reddit	3.9e-03	0.44(M)	1.8e-02	0.21(S)	3.9e-02	0.30(S)	2.4e-03	0.44(M)	2.7e-04	0.54(L)	1.8e-02	0.32(S)	3.5e-04	0.68(L)	1.1e-03	0.51(L)

TABLE XV
p-VALUE AND CLIFF'S DELTA (δ) FOR SDF AND SDF COMBINING OTHER FEATURE EXTRACTION METHODS IN TERMS OF MCC

App	SDF vs. lin_SDF		SDF vs. poly_SDF		SDF vs. rbf_SDF		SDF vs. sig_SDF		SDF vs. cos_SDF		SDF vs. PCA_SDF		SDF vs. LLE_SDF		SDF vs. IE_SDF	
	p-value	delta	p-value	delta	p-value	delta	p-value	delta	p-value	delta	p-value	delta	p-value	delta	p-value	delta
Alfresco	7.0e-06	0.62(L)	1.9e-04	0.47(M)	1.5e-05	0.59(L)	2.8e-04	0.45(M)	4.9e-04	0.44(M)	7.0e-06	0.62(L)	1.7e-06	0.95(L)	2.2e-05	0.76(L)
Sync	1.6e-05	0.69(L)	5.3e-03	0.47(M)	4.5e-05	0.76(L)	1.2e-02	0.46(M)	1.0e-02	0.36(M)	1.6e-05	0.69(L)	8.5e-06	0.82(L)	1.4e-05	0.75(L)
Keyboard	2.0e-03	0.54(L)	7.5e-05	0.68(L)	1.6e-04	0.60(L)	2.4e-02	0.36(M)	1.1e-03	0.46(M)	2.0e-03	0.54(L)	5.2e-06	0.89(L)	1.2e-04	0.67(L)
Wallpaper	3.5e-02	0.30(S)	3.5e-01	-0.1(N)	1.1e-01	0.24(S)	6.1e-01	0.09(N)	1.8e-03	0.56(L)	3.5e-02	0.30(S)	7.7e-03	0.47(M)	1.5e-01	0.24(S)
ChatSecure	1.1e-02	0.37(M)	3.9e-04	0.37(M)	2.1e-03	0.39(M)	1.4e-03	0.49(L)	2.4e-03	0.45(M)	1.1e-02	0.37(M)	2.0e-05	0.77(L)	1.5e-03	0.47(M)
Facebook	3.8e-01	0.01(N)	9.3e-03	0.32(S)	1.7e-01	0.27(S)	3.2e-02	0.27(S)	8.9e-01	0.02(N)	3.8e-01	0.01(N)	2.8e-05	0.69(L)	3.6e-03	0.46(M)
Kiwix	1.6e-03	0.44(M)	6.3e-05	0.55(L)	1.6e-04	0.66(L)	6.8e-03	0.40(M)	1.7e-03	0.48(L)	1.6e-03	0.44(M)	2.9e-06	0.94(L)	9.3e-06	0.80(L)
Own Cloud	1.2e-05	0.65(L)	1.9e-06	0.83(L)	4.3e-06	0.84(L)	7.2e-04	0.46(M)	1.7e-06	0.91(L)	1.2e-05	0.65(L)	1.9e-06	1.00(L)	3.2e-06	0.86(L)
Page Turner	3.9e-02	0.29(S)	2.7e-01	0.14(N)	6.5e-02	0.23(S)	5.6e-02	0.31(S)	4.6e-02	0.24(S)	3.9e-02	0.29(S)	8.8e-02	0.20(S)	5.4e-01	0.12(N)
Notify Reddit	1.7e-01	0.23(S)	9.8e-01	0.00(N)	3.8e-01	0.20(S)	4.7e-03	0.47(L)	1.3e-01	0.25(S)	9.8e-02	0.26(S)	1.1e-03	0.54(L)	1.5e-02	0.39(M)

TABLE XVI
p-VALUE AND CLIFF'S DELTA (δ) FOR SDF AND SDF COMBINING OTHER FEATURE EXTRACTION METHODS IN TERMS OF AUC

App	SDF vs. lin_SDF		SDF vs. poly_SDF		SDF vs. rbf_SDF		SDF vs. sig_SDF		SDF vs. cos_SDF		SDF vs. PCA_SDF		SDF vs. LLE_SDF		SDF vs. IE_SDF	
	p-value	delta	p-value	delta	p-value	delta	p-value	delta	p-value	delta	p-value	delta	p-value	delta	p-value	delta
Alfresco	2.2e-05	0.58(L)	8.5e-06	0.52(L)	4.5e-05	0.52(L)	3.2e-03	0.33(S)	8.3e-04	0.48(L)	2.2e-05	0.58(L)	3.2e-06	0.84(L)	7.5e-05	0.66(L)
Sync	2.0e-05	0.71(L)	9.3e-03	0.46(M)	3.3e-05	0.71(L)	1.3e-02	0.46(M)	2.7e-02	0.28(S)	2.0e-05	0.71(L)	4.3e-06	0.79(L)	5.2e-06	0.81(L)
Keyboard	4.5e-05	0.64(L)	1.6e-05	0.72(L)	8.5e-06	0.70(L)	2.6e-03	0.44(M)	3.9e-04	0.48(L)	4.5e-05	0.64(L)	2.4e-06	0.93(L)	4.1e-05	0.73(L)
Wallpaper	3.9e-02	0.26(S)	1.9e-01	-0.14(N)	3.7e-01	0.18(S)	1.7e-01	0.19(S)	8.0e-04	0.62(L)	3.9e-02	0.26(S)	1.8e-03	0.49(L)	1.5e-01	0.25(S)
ChatSecure	2.0e-01	0.26(S)	8.2e-03	0.32(S)	8.3e-04	0.49(L)	5.2e-02	0.38(M)	2.2e-02	0.31(S)	2.0e-01	0.26(S)	2.0e-05	0.80(L)	2.8e-03	0.54(L)
Facebook	4.2e-01	-0.03(N)	1.5e-02	0.36(M)	1.0e-01	0.32(S)	2.6e-02	0.24(S)	9.3e-01	-0.02(N)	4.2e-01	-0.03(N)	1.4e-05	0.74(L)	1.6e-03	0.53(L)
Kiwix	4.1e-05	0.6(L)	2.4e-06	0.77(L)	2.1e-06	0.82(L)	3.7e-05	0.54(L)	9.7e-05	0.52(L)	4.1e-05	0.60(L)	1.7e-06	0.99(L)	2.6e-06	0.93(L)
Own Cloud	1.6e-05	0.58(L)	1.7e-06	0.94(L)	1.7e-06	0.94(L)	4.9e-04	0.50(L)	1.7e-06	0.92(L)	1.6e-05	0.58(L)	1.7e-06	1.00(L)	2.6e-06	0.92(L)
Page Turner	1.4e-02	0.29(S)	3.2e-01	0.14(N)	1.7e-02	0.25(S)	8.0e-02	0.29(S)	1.8e-02	0.24(S)	1.4e-02	0.29(S)	4.8e-02	0.22(S)	2.4e-01	0.18(S)
Notify Reddit	3.7e-02	0.32(S)	4.4e-01	0.07(N)	1.3e-01	0.25(S)	1.2e-03	0.56(L)	2.0e-02	0.42(M)	4.4e-02	0.28(S)	6.1e-04	0.60(L)	7.3e-03	0.45(M)

our SDF model; thus, we do not report the results of the two filter-based feature subset selection methods in the tables.

From Table XI, we can observe that, in terms of F-measure, SDF model obtains the best performance on 8 out of 10 Android mobile apps. The average F-measure value by our SDF model across the 10 apps achieves improvements by 23.6%, 22.9%, 34.6%, 15.5%, 17.8%, 22.9%, 66.3%, and 38.6% compared with SDF combining KPCA using linear kernel, polynomial kernel, RBF, sigmoid kernel, cosine kernel, and SDF combining PCA, LLE, and IE, respectively. From Table XII, we can observe that, in terms of MCC, SDF model obtains the best performance on seven out of ten Android mobile apps. The average MCC value by our SDF model across the ten apps achieves improvements by 22.1%, 19.2%, 26.7%, 21.6%, 24.1%, 22.6%, 82.0%, and 35.7% compared with SDF combining KPCA using linear kernel, polynomial kernel, RBF, sigmoid kernel, cosine kernel, and SDF combining PCA, LLE, and IE, respectively. From Table XIII, we can observe that, in terms of AUC, SDF model obtains the best performance on eight out of ten Android mobile apps. The average AUC value by our SDF model across the ten apps achieves improvements by 4.6%, 4.6%, 6.0%, 4.4%, 4.4%, 4.6%, 11.8%, and 7.4% compared with SDF combining KPCA using linear kernel, polynomial kernel, RBF, sigmoid kernel, cosine kernel, and SDF combining PCA, LLE, and IE, respectively.

In addition, the reason that the Cor and Con achieves the same results as our SDF methods is that the two methods both reserve all features as the final choice. In other words, the two filter-based feature subset selection methods have no effect on feature reduction on the JIT defect data for Android mobile apps.

Tables XIV –XVI present the p-value of Wilcoxon signed-rank test and the Cliff delta (δ) for F-measure, MCC, and AUC, respectively. Noting that, in most of the cases, our SDF model is also significantly superior to all baseline methods with large and medium effectiveness level in terms of all three indicators.

Fig. 5(a)–(c) visualizes the statistic test results of Scott–Knott ESD for F-measure, MCC, and AUC, respectively. These figures illustrate that our SDF model ranks the first and is significantly superior to SDF combining PCA and KPCA methods in terms of all three indicators.

These baseline methods are all based on the SDF model. As the time of feature extraction and feature subset selection can be almost negligible compared with the time of the SDF model, the time of the baseline methods is nearly the same as our SDF model. Thus, we do not conduct the efficiency analysis here.

In summary, our SDF model without feature processing performs better than SDF with the feature extraction methods, and achieves the same performance as SDF with the filter-based feature subset selection methods. It means that, as our SDF

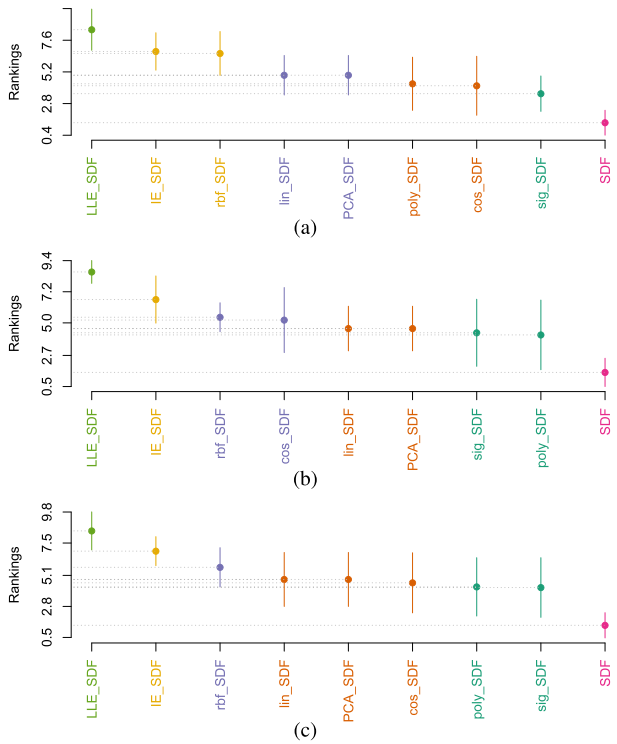

Fig. 5. Scott–Knott ESD test for DF and DF combining PCA or KPCA methods. (a) *F*-measure. (b) MCC. (c) AUC.

inherently contains the ability of feature representation learning, the additional feature extraction operations based on PCA, KPCA, LLE, and IE are not helpful for further performance improvement of our SDF model and the additional feature subset selection operations based on Cor and Con do not work in feature dimension reduction for the studied defect data of Android mobile apps.

VI. Threats to Validity

We conduct experiments on a publicly available dataset that has been released recently. Since our experimental results are derived from defect data of ten Android mobile apps, to further improve the generalization of experimental results, we plan to enrich the benchmark dataset by collecting the defect data from the top-ranked mobile apps in the GitHub repository or collaborating with some companies to collect defect data from the mobile apps in operation as our future work. We implement our SDF model by carefully modifying the source code of deep forest provided by original authors and use the third-party libraries to implement these comparative methods. The aim is to minimize the potential faults due to our own implementation, which threatens the internal validity. In addition, we choose the stratified sampling method to generate the training set and test set without considering the commit time information due to no such information available in the defect data at hand. This setting may not exactly match the unrealistic scenario. We will actively contact the authors for the raw defect data that consist of the time information and use the timewise-cross-validation setting to remove this threat. As the rationality of the selected performance evaluation and statistic test method could potentially threaten the construct validity, in this work, we employ three comprehensive indicators to evaluate the effectiveness of our method and use two types of statistic test methods for both analysis of method pairs and analysis between multiple methods, which makes our evaluation more convincing.

VII. Conclusion

In this article, we proposed a new method, called SDF, to build JIT defect-prediction model on Android mobile apps. This model is an integration of traditional forest models in breadth and depth. It produced a deep forest system with a cascading structure that was used for feature representation learning to improve the effectiveness of the classification task. We conducted experiments on ten Android mobile apps and used three indicators to evaluate the performance of our SDF model. The experimental results illustrated that our SDF model achieved promising performance and obtained significantly better results compared with 16 baseline methods.

In the future, we plan to collect more defect data of mobile apps and use effort-aware indicators to measure the performance of our SDF model. Moreover, as the performance of our SDF method is affected by the class-imbalance issue of the defect data for Android apps, we will improve our SDF model in the model construction to adapt to the imbalanced defect data for further performance improvement for JIT defect-prediction task on Android mobile apps.

References

[1] L. Wang, X. Sun, J. Wang, Y. Duan, and B. Li, "Construct bug knowledge graph for bug resolution," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. Companion*, 2017, pp. 189–191.

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

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

[4] 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.

[5] S. McIlroy, N. Ali, and A. E. Hassan, "Fresh apps: An empirical study of frequently-updated mobile apps in the Google Play store," *Empirical Softw. Eng.*, vol. 21, no. 3, pp. 1346–1370, 2016.

[6] M. Nayebi, B. Adams, and G. Ruhe, "Release practices for mobile apps-what do users and developers think?," in *Proc. 23rd Int. Conf. Softw. Anal., Evol., Reengineering*, vol. 1, 2016, pp. 552–562.

[7] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, pp. 2–13, Jan. 2007.

[8] M. Jureczko and D. Spinellis, "Using object-oriented design metrics to predict software defects," *Models and Methods of System Dependability*. Wrocław, Poland: Oficyna Wydawnicza Politechniki Wrocławskiej, pp. 69–81, 2010.

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

[10] 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.

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

[13] 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.*, to be published.

[14] 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. Found. Softw. Eng.*, 2020, pp. 1308–1319.

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

[16] Z. Zhou and J. Feng, "Deep forest: Towards an alternative to deep neural networks," in *Proc. 26th Int. Joint Conf. Artif. Intell*., 2017, pp. 3553–3559.

[17] 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.

[18] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? A longitudinal case study of just-in-time defect prediction," *IEEE Trans. Softw. Eng.*, vol. 44, no. 5, pp. 412–428, May 2018.

[19] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," *J. Syst. Softw.*, vol. 150, pp. 22–36, 2019.

[20] 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.

[21] 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.

[22] G. G. Cabral, L. L. Minku, E. Shihab, and S. Mujahid, "Class imbalance evolution and verification latency in just-in-time software defect prediction," in *Proc. 41st Int. Conf. Softw. Eng.*, 2019, pp. 666–676.

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

[24] T. Zhou, X. Sun, X. Xia, B. Li, and X. Chen, "Improving defect prediction with deep forest," *Inf. Softw. Technol.*, vol. 114, pp. 204–216, 2019.

[25] W. Zheng, S. Mo, X. Jin, Y. Qu, Z. Xie, and J. Shuai, "Software defect prediction model based on improved deep forest and autoencoder by forest," in *Proc. 31st Int. Conf. Softw. Eng. Knowl. Eng.*, 2019, pp. 419–540.

[26] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, and J. Liu, "Dictionary learning based software defect prediction," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 414–423.

[27] X. Jing, F. Wu, X. Dong, F. Qi, and B. Xu, "Heterogeneous cross-company defect prediction by unified metric representation and CCA-based transfer learning," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, 2015, pp. 496–507.

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

[29] N. Li, M. Shepperd, and Y. Guo, "A systematic review of unsupervised learning techniques for software defect prediction," *Inf. Softw. Technol.*, vol. 122, 2020, Art. no. 106287.

[30] X.-Y. Jing, F. Wu, X. Dong, and B. Xu, "An improved SDA based defect prediction framework for both within-project and cross-project class-imbalance problems," *IEEE Trans. Softw. Eng.*, vol. 43, no. 4, pp. 321–339, Apr. 2017.

[31] F. Zhang, Q. Zheng, Y. Zou, and A. E. Hassan, "Cross-project defect prediction using a connectivity-based unsupervised classifier," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng*., 2016, pp. 309–320.

[32] Y. Zhou *et al.*, "How far we have progressed in the journey? An examination of cross-project defect prediction," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 1, pp. 1–51, 2018.

[33] 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.

[34] S. Wang and X. Yao, "Using class imbalance learning for software defect prediction," *IEEE Trans. Rel.*, vol. 62, no. 2, pp. 434–443, Jun. 2013.

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

[36] 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.

[37] F. Wilcoxon, "Individual comparisons by ranking methods," in Breakthroughs in Statistics. Berlin, Germany: Springer, 1992, pp. 196–202.

[38] N. Cliff, Ordinal Methods for Behavioral Data Analysis. New York, NY, USA: Psychology Press, 2014.

[39] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 1–18, Jan. 2017.

[40] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *Proc. 37th Int. Conf. Softw. Eng*., 2015, pp. 789–800.

[41] Z. Xu, J. Xuan, J. Liu, and X. Cui, "MICHAC: Defect prediction via feature selection based on maximal information coefficient with hierarchical agglomerative clustering," in *Proc. 23rd Int. Conf. Softw. Anal., Evol., Reengineering*, vol. 1, 2016, pp. 370–381.

[42] Z. Xu, J. Liu, Z. Yang, G. An, and X. Jia, "The impact of feature selection on defect prediction performance: An empirical comparison," in *Proc. 27th Int. Symp. Softw. Rel. Eng*., 2016, pp. 309–320.

[43] Z. Xu, J. Liu, X. Luo, and T. Zhang, "Cross-version defect prediction via hybrid active learning with kernel principal component analysis," in *Proc. 25th Int. Conf. Softw. Anal., Evol., Reengineering*., 2018, pp. 209–220.

[44] Z. Xu *et al.*, "Software defect prediction based on kernel PCA and weighted extreme learning machine," *Inf. Softw. Technol.*, vol. 106, pp. 182–200, 2019.

[45] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics Intell. Lab. Syst.*, vol. 2, no. 1–3, pp. 37–52, 1987.

[46] B. Schölkopf, A. Smola, and K.-R. Müller, "Kernel principal component analysis," in *Proc. Int. Conf. Artif. Neural Netw.*, 1997, pp. 583–588.

[47] S. T. Roweis and L. K. Saul, "Nonlinear dimensionality reduction by locally linear embedding," *Science*, vol. 290, no. 5500, pp. 2323–2326, 2000.

[48] J. B. Tenenbaum, V. De Silva, and J. C. Langford, "A global geometric framework for nonlinear dimensionality reduction," *Science*, vol. 290, no. 5500, pp. 2319–2323, 2000.

[49] M. A. Hall, "Correlation-based feature selection of discrete and numeric class machine learning," in *Proc. 17th Int. Conf. Mach. Learn.*, 2000, pp. 359–366.

[50] M. Dash, H. Liu, and H. Motoda, "Consistency based feature selection," in *Proc. 4th Pacific-Asia Conf. Knowl. Discovery Data Mining*. Berlin, Germany: Springer, 2000, pp. 98–109.