The Journal of Systems and Software 158 (2019) 110402

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

In Practice

LDFR: Learning deep feature representation for software defect prediction☆

Zhou Xu[a,b], Shuai Li[b], Jun Xu[c], Jin Liu[a,d,*], Xiapu Luo[b], Yifeng Zhang[a], Tao Zhang[e], Jacky Keung[f], Yutian Tang[b]

[a] School of Computer Science, Wuhan University, China
[b] Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China
[c] College of Computer Science, Nankai University, China
[d] Key Laboratory of Network Assessment Technology, Institute of Information Engineering, Chinese Academy of Sciences, China
[e] Faculty of Information and Technology, Macau University of Science and Technology, Macao, China
[f] Department of Computer Science, City University of Hong Kong, Hong Kong, China

ARTICLE INFO

Article history:
Received 7 October 2018
Revised 8 August 2019
Accepted 22 August 2019
Available online 2 September 2019

MSC:
00-01
99-00

Keywords:
Software defect prediction
Deep feature representation
Triplet loss
Weighted cross-entropy loss
Deep neural network

ABSTRACT

Software **D**efect **P**rediction (**SDP**) aims to detect defective modules to enable the reasonable allocation of testing resources, which is an economically critical activity in software quality assurance. Learning effective feature representation and addressing class imbalance are two main challenges in SDP. Ideally, the more discriminative the features learned from the modules and the better the rescue performed on the imbalance issue, the more effective it should be in detecting defective modules. In this study, to solve these two challenges, we propose a novel framework named **LDFR** by **L**earning **D**eep **F**eature **R**epresentation from the defect data for SDP. Specifically, we use a deep neural network with a new hybrid loss function that consists of a triplet loss to learn a more discriminative feature representation of the defect data and a weighted cross-entropy loss to remedy the imbalance issue. To evaluate the effectiveness of the proposed LDFR framework, we conduct extensive experiments on a benchmark dataset with 27 defect data (each with three types of features), using three traditional and three effort-aware indicators. Overall, the experimental results demonstrate the superiority of our LDFR framework in detecting defective modules when compared with 27 baseline methods, except in terms of the indicator of Precision.

1. Introduction

Most exceptions and failures during the process of software execution are rooted in defects in software modules (such as classes and files). Although tremendous efforts have been made to secure programming, software defects are still inevitable. Therefore, detecting defective software modules before releasing software artifacts is a critical issue that should not be overlooked in the software development lifecycle (Fenton and Ohlsson, 2000).

To identify highly risky software modules that are potentially defective (Song et al., 2011), **S**oftware **D**efect **P**rediction (**SDP**) can effectively guide the direction of software testing by allocating the limited resources available for testing and verification to highly risky modules. It is therefore beneficial in improving software quality and reliability, which leads to safer and more robust software artifacts.

Many SDP methods have been proposed, and most use machine learning approaches (Lessmann et al., 2008; Shepperd et al., 2014), which can be categorized from two perspectives. From the perspective of whether they use historical defect information for SDP, existing studies can be roughly divided into supervised SDP and unsupervised SDP. The supervised SDP uses historical defect data with labels to train a classification or regression model, and the model is then used to determine the defect information of the new software modules. In this study, the defect data consist of a set of module features that characterize the modules, and the defect information that denotes the number of defects or just binary labels. The binary labels indicate whether the corresponding modules contain defects. Unlike the supervised SDP, the unsupervised SDP relies only upon the module features to partition the modules into different groups (usually two groups: the defective group and the non-defective group) using clustering methods, such as k-means clustering (Zhong et al., 2004), affinity propagation cluster-

☆ Fully documented templates are available in the elsarticle package on CTAN

* The corresponding authors.
E-mail addresses: jinliu@whu.edu.cn (J. Liu), csxluo@comp.polyu.edu.hk (X. Luo).

https://doi.org/10.1016/j.jss.2019.110402
0164-1212/

ing (Yang et al., 2008), and spectral clustering (Zhang et al., 2016). From the perspective of whether the defect information is binary, the supervised SDP can be further divided into defect-prone prediction and defect-number prediction. The former scenario usually trains a classification model on the labeled defect data and then predicts whether the new modules will be defective. Widely used classification models include Bayesian networks (Okutan and Yıldız, 2014), logistic regression (Tantithamthavorn et al., 2017), decision tree (Wang and Yao, 2013), nearest neighbor (Jing et al., 2015), random forest (Yang et al., 2017), and support vector machine (Arora and Saha, 2018). The latter scenario generally uses the module features and defect number to fit a regression model to describe their relationship, and then estimates the number of defects in new modules. In this study, we focus on the former scenario, defect-prone prediction, because it is a more general form of SDP.

The performance of SDP is greatly affected by the feature representation of the original defect data because the initial collected module features may not well represent the intrinsic structure hidden behind the original defect data (Wang et al., 2016b; Yang et al., 2015). Motivated by the success of the application of deep learning techniques to learn features in various areas (such as computer vision (Cheng et al., 2016) and natural language processing (Collobert and Weston, 2008)), in this paper, we propose a new **D**eep **N**eural **N**etwork (**DNN**)-based framework, named **L**earning **D**eep **F**eature **R**epresentation (**LDFR**), to learn a high-level feature representation for the defect data. In our LDFR framework, we introduce a novel hybrid loss function to the DNN to learn more discriminative deep features while alleviating the imbalance issue. More specifically, the proposed hybrid loss function consists of an advanced triplet loss function (Schroff et al., 2015) and a weighted cross-entropy loss function. The triplet loss function has the advantage of maximizing the interclass variations and minimizing the intraclass variations among the learned features (Parchami et al., 2017). In other words, the learned features favor smaller distances between module pairs with the same label and larger distances for module pairs with different labels. The commonly used cross-entropy loss function (De Boer et al., 2005) in DNNs aims to maximize the discriminative property among the learned deep features of the training modules. In addition, because SDP emphases the detection of defective software modules, the misclassification cost of identifying a defective module as non-defective should be higher than that of identifying a non-defective module as defective. However, the cross-entropy loss function treats the two cases equally. In this study, we broaden its application by extending the cross-entropy loss to a weighted form that can punish more on the cost when the defective modules are predicted to be non-defective. With the joint supervision of triplet loss and weighted cross-entropy loss, we can train the DNN to attain more effective feature representation. To the best of our knowledge, this study is among the first to introduce the triplet loss into SDP and combine it with a weighted cross-entropy loss to address both the feature representation learning and the class imbalance issue for SDP.

To evaluate the effectiveness of our proposed LDFR framework, we conduct extensive experiments on a benchmark defect dataset that includes 14 projects with 27 total versions. Each project contains three types of metrics (a.k.a. features): **C**hidamber and **K**emerer (**CK**) metrics, process metrics, and network metrics. We use six indicators as our performance measurement, including three traditional indicators (i.e., Precision, Recall, and F-measure) and three effort-aware indicators (i.e., EAPrecision, EARecall, and EAF-measure (Xu et al., 2018)). The main performance improvements in our two research questions are summarized as follows:

- RQ1: Compared with the best average indicator values among four variant methods of LDFR, LDFR achieves average improvements of 7.6%, 13.5%, 15.3%, 7.1%, and 11.6% under defect data with CK features, of 5.1%, 11.5%, 16.0%, 9.8%, and 12.6% under defect data with process features, and of 6.4%, 10.1%, 14.7%, 7.9%, and 11.8% under defect data with network features in terms of Recall, F-measure, EAPrecision, EARecall, and EAF-measure, respectively. The average Precision obtained by LDFR is only lower than that obtained by one variant method under three types of defect data.
- RQ2: Compared with the best average indicator values among 23 existing imbalanced learning methods, LDFR achieves average improvements of 5.5%, 16.5%, and 23.2% under defect data with CK features, of 13.2%, 29.5%, and 22.4% under defect data with process features, and of 7.0%, 25.3%, and 12.4% under defect data with network features in terms of F-measure, EARecall, and EAF-measure, respectively. The average Precision, Recall, and EAPrecision obtained by LDFR are inferior to those obtained by several baseline methods.

In summary, we highlight the major contributions as follows:

1. We propose a novel LDFR framework to learn deep feature representation for the defect data. LDFR combines a triplet loss and a weighted cross-entropy loss for DNN training. The features induced by the triplet loss simultaneously favor compactness for the intraclass modules and dispersion for the interclass modules. The feature induced by the weighted cross-entropy loss can remedy the model bias to identify more defective modules.
2. We conduct large-scale empirical experiments on 27 project versions. As each project contains three types of metrics, we could conduct a thorough exploration of the effectiveness of our proposed LDFR framework on defect data with various feature types.
3. We perform an extensive performance comparison of our LDFR framework and 27 baseline methods with both traditional and effort-aware indicators. Compared with the baseline methods, the results demonstrate that LDFR usually achieves encouraging performance in terms of five indicators except Precision.

The rest of this article is structured as follows: In Section 2, we briefly introduce the background of SDP and triplet loss based feature representation learning. In Section 3, we present the technical details of our LDFR framework. In Section 4, we introduce the experimental design of our study, such as the benchmark dataset, the evaluation indicators, and the parameter setting. In Section 5, we report and analyze the experimental results in detail. In Section 6, we discuss the impacts of different classifiers and feature types on the performance of LDFR. In Section 7, we list the potential threats to validity. In Section 8, we describe the related work. In Section 9, we conclude our study.

2. Background

2.1. Software defect prediction

In recent decades, numerous studies have examined the realm of SDP. In general, most studies focus mainly on defect-prone prediction, that is, predicting whether modules are defective or non-defective.

The most popular way to identify whether new modules are defective is to train a supervised classification model on the historical labeled defect data and then use the learned model to predict the labels of upcoming software modules. Many studies devoted to this research domain have proposed various SDP methods. These studies can be roughly divided into three groups. The first group focuses on the use of feature selection methods to identify a subset of initial features. This selected feature subset replaces the original set to train the classification models. Most feature selection methods belong to filter-based feature-ranking meth-

ods or wrapper-based feature subset selection methods. Filter-based feature-ranking methods individually calculate an importance score for each feature towards the class label and then rank the features based on their scores, and finally the top-ranked features are reserved. Classical filter-based feature-ranking methods include chi-square, information gain, and ReliefF (Xu et al., 2016; Ghotra et al., 2017). Wrapper-based feature subset selection methods evaluate the performance of various feature subsets for a given indicator under a predetermined classifier and select the feature subset that can achieve the best indicator value (Xu et al., 2016; Ghotra et al., 2017). The second group concentrates on the use of various imbalance learning methods to alleviate the class imbalance issue of the defect data. The imbalance trait will lead general classification models' bias to predict the modules as the majority class. Most imbalance learning methods can be classified as sampling-based, ensemble-based, or cost-sensitive-based methods (Huang et al., 2016). The third group concerns the effectiveness of the classification models for SDP. Previous researchers (Lessmann et al., 2008; Challagulla et al., 2008; Mende and Koschke, 2009; Ghotra et al., 2015) have conducted empirical studies to compare the performance differences of various classification models for SDP.

As deep learning appears, some recent studies have come closer to our study by applying deep leaning techniques to a defect prediction task. Yang et al. (2015) employed a probabilistic generation model, called **D**eep **B**rief **N**etwork (**DBN**) (Salakhutdinov, 2015), for just-in-time defect prediction. They mainly used the original DBN as an unsupervised feature learning method to preserve as many characteristics of the original feature as possible while reducing the feature dimensions. In contrast, our study mainly proposes a novel deep learning method with a well-designed hybrid loss function to learn the feature representation of the defect data. This is a discriminative model that not only retains the original characteristics but also automatically adjusts the distances among the modules with the same or different labels. Manjula and Florence (2018) proposed a hybrid approach based on DNN for defect classification, but the innovation of their work is that an improved genetic algorithm is introduced to select a feature subset that was later used as the inputs of a general DNN method later. Other studies Wang et al. (2016a); Li et al. (2017a); Phan et al. (2018); Dam et al. (2017, 2018) have used existing deep learning models, such as DBN, convolutional neural network, and long short-term memory, to extract features directly from the source code of the projects. The main difference between our study and the others is that we learn features with a novel proposed deep learning method from various feature representations (i.e., object-oriented design features, process features, and network features) of the source code, not directly from the code.

2.2. Triplet loss based feature representation learning

Since its introduction, the triplet loss paradigm has been successfully applied to learn discriminate feature representation in many applications, including fingerprint matching Zhang et al. (2017), speaker verification Zhang and Koishida (2017), video retrieval Dong and Li (2017), person re-identification Cheng et al. (2016), and face recognition Parkhi et al. (2015). Triplet loss-based feature representation learning aims to find an embedded feature space for the training samples because the initial features may not well encode the data space. In the embedded space, the distances between the samples that share the same labels are fixed or decreased, whereas the distances between samples with different labels are increased to some extent. Schroff et al. (2015) were the first to propose the triplet loss function to map images of faces into a compact feature space in which the Euclidean distances directly represent the similarity of the faces. They used a deep convolutional network to optimize feature embedding and achieved the best face recognition performance at that time. To scale triplet loss-based feature learning to a large-scale dataset, Ming et al. (2017) combined this method with interclass/intraclass distance feature learning to reduce the number of triplets for training. The experimental results showed that their method could achieve performance comparable to that of Schroff et al. with fewer trained triplets. Liao et al. (2017) proposed a weighted triplet loss function by adding two weights to further reduce the interclass distances while increasing the intraclass distances for person re-identification. Su et al. (2017) proposed a face recognition framework via an adaptive triplet loss function with a softmax function. The softmax function was used to initiate the parameters of the neural network, and the adaptive triplet loss was used to generate high-quality triplets.

Unlike the studies above which mainly used triplet loss to learn effective feature representation for images, we focus on learning discriminative features for software modules from the defect data with collected features. As this loss remains uninvestigated in the software engineering domain, we are among the first to introduce the triplet loss-based feature representation learning to defect prediction. In addition, we design a new DNN framework that combines the triplet loss function with a weighted cross-entropy loss function to learn more discriminative feature representation from the collected module features.

3. Our LDFR framework

3.1. Triplet loss-based feature learning

One challenge of SDP is to increase the separability of modules that belong to different classes. This issue can be addressed by using triplet loss to learn highly discriminative feature representation for the modules with different labels. Here, we describe the details of the triplet loss function.

Assume that the feature set of defect data as $\mathbf{X} = \{\boldsymbol{x}_i\}_{i=1}^{n} \in \mathbb{R}^{n\times m}$ and the label set as $\mathbf{Y} = \{\boldsymbol{y}_i\}_{i=1}^{n} \in \mathbb{R}^{n\times 1}$, where $\boldsymbol{x}_i = [\boldsymbol{x}_{i1}, \boldsymbol{x}_{i2}, \ldots, \boldsymbol{x}_{im}] \in \mathbb{R}^m$ denotes the ith module and $y_i \in \{0, 1\}$ denotes the label of $\boldsymbol{x}_i$. $y_i = 0$ indicates that the corresponding module $\boldsymbol{x}_i$ is non-defective, whereas $y_i = 1$ indicates that the corresponding module $\boldsymbol{x}_i$ is defective. The input variables of the feature representation learning are the feature set and the label set. The goal of feature representation is to learn a mapping representation $f(\boldsymbol{x}_i)$ for the module $\boldsymbol{x}_i$ from the original feature space $\mathbb{R}^m$ into a d-dimensional embedding space $\mathbb{R}^d$. In this new feature space, the distances of the modules within the same class decrease while the distances of the modules between different classes increase. The embedding is constrained on the d-dimensional hypersphere (Schroff et al., 2015), i.e., $\|f(\boldsymbol{x})\|_2 = 1$. Each triplet consists of three elements $(\boldsymbol{x}_i, \boldsymbol{x}_i^+, \boldsymbol{x}_i^-)$ that are defined as follows:

- $\boldsymbol{x}_i$: an *anchor* example, i.e., a module,
- $\boldsymbol{x}_i^+$: a *positive* example, i.e., the module with the same label as $\boldsymbol{x}_i$,
- $\boldsymbol{x}_i^-$: a *negative* example, i.e., the module with different label as $\boldsymbol{x}_i$.

Triplet loss is motivated by nearest-neighbor classification and strives to ensure that the anchor is closer to the positive module than the negative module in the embedded feature space. This process is visualized in Fig. 1. Assume that the set of all possible triplets in the training set is τ with the cardinality N, the following relationship then holds true in the embedded feature space:

$$d(f(\boldsymbol{x}_i), f(\boldsymbol{x}_i^+)) + \alpha < d(f(\boldsymbol{x}_i), f(\boldsymbol{x}_i^-)), \forall(\boldsymbol{x}_i, \boldsymbol{x}_i^+, \boldsymbol{x}_i^-) \in \tau, \quad (1)$$

where $d(\cdot) = \|\cdot\|_2^2$ is the Euclidean distance, α is a margin between the positive and negative pairs. The triplet loss (Loss1) is

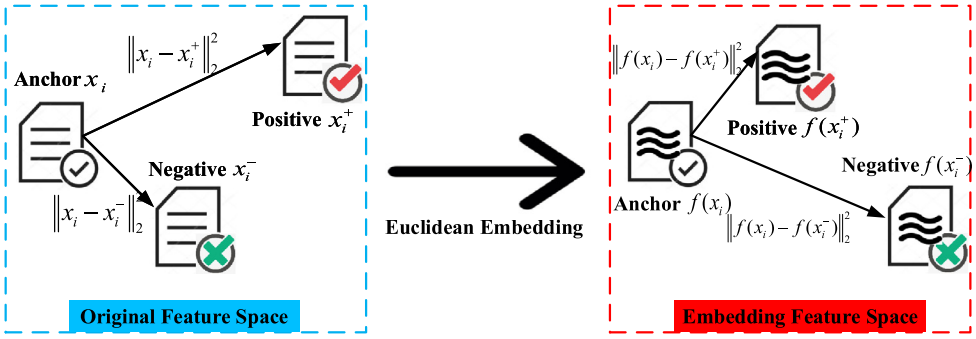

Fig. 1. An illustrate of the merit of triplet loss.

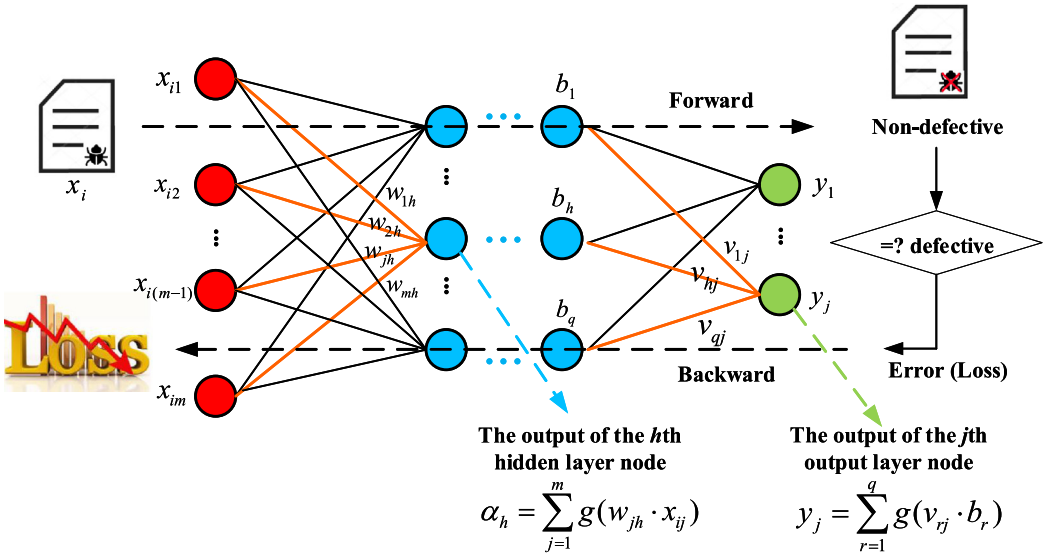

Fig. 2. An neural structure of the DNN.

then defined as follows

$$Loss1 = \sum_{i=1}^{N} [d(f(\boldsymbol{x}_i), f(\boldsymbol{x}_i^+)) - d(f(\boldsymbol{x}_i), f(\boldsymbol{x}_i^-)) + \alpha]_+, \tag{2}$$

where $[\cdot]_+$ indicates that the value of $\cdot$ is used as the loss if it is positive, otherwise, the value of $\cdot$ equals to 0. More specifically, when the margin between the distance $d(f(\boldsymbol{x}_i), f(\boldsymbol{x}_i^-))$ and the distance $d(f(\boldsymbol{x}_i), f(\boldsymbol{x}_i^+))$ is less than α (i.e., after subtracting a positive margin α, the negative module $\boldsymbol{x}_i^-$ is closer to the anchor than the positive module $\boldsymbol{x}_i^+$), it will cause loss. Otherwise, the loss is set as 0. The optimized objective function is to minimize the triplet loss. When the loss of the triplet $(\boldsymbol{x}_i, \boldsymbol{x}_i^+, \boldsymbol{x}_i^-)$ is larger than 0, Eq. 2 is a convex function that can be solved by gradient derivation. Its gradients with respect to $f(\boldsymbol{x}_i)$, $f(\boldsymbol{x}_i^+)$, and $f(\boldsymbol{x}_i^-)$ are derived as follows:

$$\begin{aligned} \frac{\partial Loss1}{\partial f(\boldsymbol{x}_i)} &= 2(f(\boldsymbol{x}_i) - f(\boldsymbol{x}_i^+)) - 2(f(\boldsymbol{x}_i) - f(\boldsymbol{x}_i^-)) \\ &= 2(f(\boldsymbol{x}_i^-) - f(\boldsymbol{x}_i^+)). \end{aligned} \tag{3}$$

$$\frac{\partial Loss1}{\partial f(\boldsymbol{x}_i^+)} = 2(f(\boldsymbol{x}_i) - f(\boldsymbol{x}_i^+)) \cdot (-1) = 2(f(\boldsymbol{x}_i^+) - f(\boldsymbol{x}_i)). \tag{4}$$

$$\frac{\partial Loss1}{\partial f(\boldsymbol{x}_i^-)} = -2(f(\boldsymbol{x}_i) - f(\boldsymbol{x}_i^-)) \cdot (-1) = 2(f(\boldsymbol{x}_i) - f(\boldsymbol{x}_i^-)). \tag{5}$$

3.2. Deep neural network (DNN)

In general, DNN consists of three types of network layer. The first type of layer is called the input layer, which corresponds to the input units, that is, the module features, in this study. The second type of layer is called the hidden layer, which is used to transform the features from the previous nearest layer. The last type of layer is called the output layer, which gives the specified outcomes, such as the labels of the input modules for SDP. After obtaining the outcomes, the loss value can be calculated for parameter optimization with the back-propagation algorithm (Rumelhart et al., 1986). In general, the nodes among various layers are fully connected, whereas the nodes within the same layer have no direct connections. In addition, the number of the nodes in the input and output layers are assigned for specific applications, whereas the number of hidden layers and the number of nodes for each hidden layer are designed empirically.

Fig. 2 depicts a basic structure of DNN with fully connected networks. For example, the output of the first hidden layer is $\alpha_h = \sum_{j=1}^{m} g(\omega_{jh} \cdot x_{ij})$, where ω_{jh} denotes the input weight vector connecting the jth input node and the hth hidden node, x_{ij} denotes the jth input vector of the module $\boldsymbol{x}_i$, and $g(\cdot)$ is a nonlinear activation function. In addition, the output of the output layer is $y_j = \sum_{r=1}^{q} g(v_{rj} \cdot b_r)$, where v_{rj} denotes the output weight vector connecting the jth output node and the rth hidden node, and b_r is the output value of the rth hidden node. y_j is the probabil-

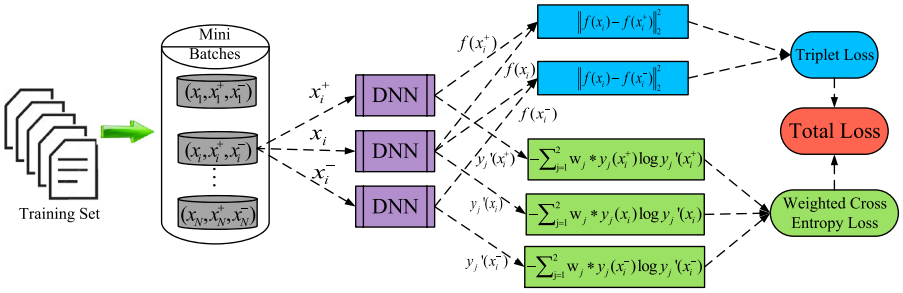

Fig. 3. Our feature representation learning architecture with the hybrid loss function.

ity that a given module $\boldsymbol{x}_i$ belongs to the jth class. Note that each hidden node can have a bias term; we omit the bias terms from Fig. 2 for simplicity. The output of DNN is a vector of probability values for which each element corresponds to a class label. The class label with the highest probability indicates that the module belongs to this class. The classification loss is calculated as the gap between the actual probability and the output probability of the module label by the DNN. Note that, in this study, the actual probabilities for the defective and non-defective modules are 1.0 and 0.0, respectively. This loss is used to train the DNN to optimize the network parameters to maximize the probability of the correct class label and minimize the probability of the incorrect class label, that is, to minimize the classification loss over the given training set.

The training process of the DNN consists of two alternative updating steps: the forward transmission of the information and the back-propagation of the loss. As showed in Fig. 2, given a set of input vectors of a defective modules in the forward transmission process, the module features and parameter information spread the neural network architecture to predict the label of the input module. If the network identifies the module as non-defective, the loss will be generated. In the back-propagation process, the loss is used to update the network parameters with a hill-climbing optimization process called gradient descent (Sze et al., 2017).

Traditional DNNs normally use cross-entropy loss as the loss function to train the neural network parameters. For a module $\boldsymbol{x}_i$, the corresponding cross-entropy loss ($Loss2$) is defined as follows:

$$Loss2 = -\sum_{j=1}^{C} y'_j(\boldsymbol{x}_i) \log y_j(\boldsymbol{x}_i), \tag{6}$$

where $y'_j(\boldsymbol{x}_i)$ denotes the ground-truth label (a.k.a. actual label) probability of module $\boldsymbol{x}_i$, $y_j(\boldsymbol{x}_i)$ denotes the output probability of module $\boldsymbol{x}_i$ by DNN, and C denotes the number of classes. There are only two classes in SDP (the defective class and the non-defective class), that is, $C = 2$ in this study. The optimized objective function is to minimize the cross-entropy loss, that is, to minimize the classification error.

Here, we give an example of how to calculate the cross-entropy loss. In the multiple classification scenario, given a sample with ground-truth class probability as $y' = (0.0, 1.0, 0.0)$, we feed the sample into a DNN model and the model outputs the probability of the sample as $y = (0.3, 0.6, 0.1)$. The cross-entropy loss of the sample is then $-0.0 \times \log 0.3 - 1.0 \times \log 0.6 - 0.0 \times \log 0.1$. The loss is used for a back-propagation algorithm to nudge the network parameters so that y gradually approaches to y'.

3.3. Weighted cross-entropy loss

Software defect data usually exhibit class imbalance that leads general classification models to predict that the defective modules are non-defective. However, in the context of SDP, the cost (i.e., loss) of classifying an actual defective module as non-defective should be higher than the cost of classifying an actual non-defective module as defective. The reason is that misclassifying a defective module may result in failure of the software artifacts, whereas misclassifying a defective module only consumes additional test resources. Unfortunately, cross-entropy loss treats the two losses equally. To remedy this deficiency, in this study, we use a weighted cross-entropy loss function designed specifically for SDP as follows

$$Loss2' = -\sum_{j=1}^{C} w_j * y'_j(\boldsymbol{x}_i) \log y_j(\boldsymbol{x}_i). \tag{7}$$

3.4. Our proposed hybrid loss function

This study is among the first to use a combination of the two loss functions to optimize the network parameters to learn better feature representation toward the software modules. Our **H**ybrid **Loss** function (**HLoss**) is defined as

$$HLoss = Loss1 + Loss2'. \tag{8}$$

The intuition here is that the triplet loss function is used to ensure that the learned features are more distinguishable, whereas the weighted cross-entropy loss is used to alleviate the class imbalance issue. The goal of combining the two loss functions is to learn a more effective feature representation for classification. Fig. 3 depicts our feature representation learning network architecture. Given a training set, we sample mini batches of the triplet set because consideration of all triplets is very time consuming and usually infeasible due to limitations in memory size. Each triplet member is then fed independently into three identical DNNs with shared parameters, and the generated feature embeddings are used to calculate the triplet loss and weighted cross-entropy loss. Finally, the hybrid loss that combines the above two losses is used to update the DNN parameters with the back-propagation algorithm to further reinforce the learned feature representation.

3.5. Overall framework

Fig. 4 depicts a flow-chart of our proposed LDFR framework. We first process the defect data according to various types of features. As a result, for each project, we obtain three sets of defect data: those with CK features, those with process features, and those with network features. Next, for each project with each feature set, we use a 50/50 stratified sampling strategy to generate the training set and test set. We then use our LDFR framework to learn the deep feature representation for the training set and use the same mapping rule to extract the deep feature representation for the test set. With the mapped training set, we train a classification model and apply it to the mapped test set. Finally, we report the experimental results and use statistical tests to evaluate the effectiveness of our proposed LDFR framework.

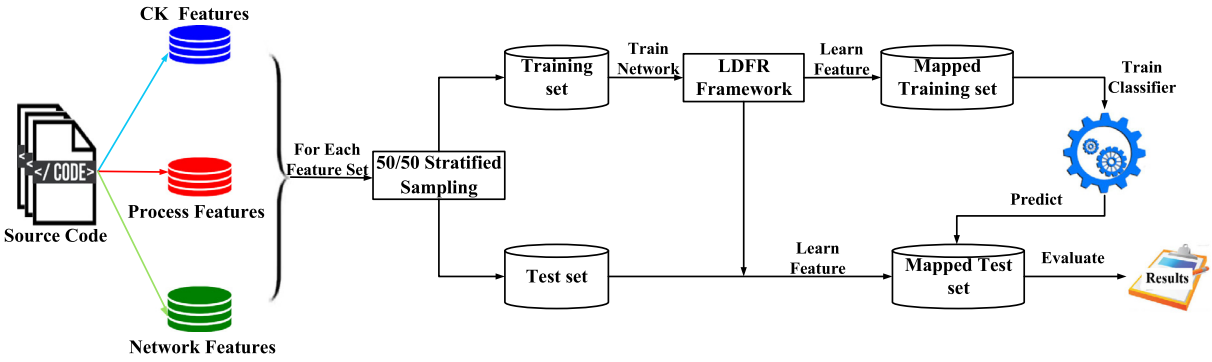

Fig. 4. The flow chart of our overall framework.

Table 1
Description of the benchmark dataset.

Project	Version	# M	% D	Project	Version	# M	% D
*ant	1.3	125	16.00%	jedit	3.2	272	33.09%
	1.4	178	22.47%		4.0	306	24.51%
	1.5	293	10.92%		4.1	312	25.32%
	1.6	351	26.21%		4.2	367	13.08%
*camel	1.0	339	3.83%		4.3	492	2.24%
	1.2	608	35.53%	velocity	1.6	229	34.06%
	1.4	872	16.63%	xerces	1.2	440	16.14%
	1.6	965	19.48%		1.3	453	15.23%
ivy	2.0	352	11.36%	Equinox	3.4	324	39.81%
log4j	1.0	135	25.19%	JDT Core	3.4	997	20.66%
poi	2.0	314	11.78%	Lucene	2.4.0	691	9.26%
synapse	1.0	157	10.19%	Mylyn	3.1	1862	13.16%
	1.1	222	27.03%	PDE UI	3.4.1	1497	13.96%
	1.2	256	33.59%	Average		497	19.66%

4. Experiment setup

4.1. Benchmark dataset

We evaluate the proposed LDFR framework on the publicly available software defect data provided by Song et al. (2018) as a benchmark dataset. This dataset includes 22 project versions from the PROMISE repository (Jureczko and Spinellis, 2010) and five projects from the AEEEM dataset (DAmbros et al., 2012) for a total of 27 project versions. For each project version, the benchmark dataset provides three types of module features: CK features, process features, and network features. Table 1 tabulates the statistical information of these 27 project versions, where # M and % D denote the number of total modules and the percentage of defective modules, respectively.

4.2. Three types of features

Here, we give brief descriptions of the features and the process by which they are collected.

(1) CK features: CK features, also known as object-oriented design features, are static measurements of the source code. Many previous studies Ghotra et al. (2015); Jing et al. (2017); Chen et al. (2015); Xia et al. (2016) have used such features to investigate the performance of various defect prediction methods. The intuition here is that the more complicated the module, the greater the probability that the module contains defects. Chidamber and Kemerer (1991, 1994) proposed a suite of six meaningful CK features. They claim that such features can help users understand the design complexity of the source code, which helps to detect design flaws. The CK features can be automatically extracted with an open-source tool, CKJM, developed by Spinellis (2005). We use the six features because they are the common ones in both original PROMISE and AEEEM datasets.

(2) Process features: Process features, also called change features, measure the development change history of the software projects. Studies have shown that process features are better measurements for software defects (Rahman and Devanbu, 2013; Graves et al., 2000; Moser et al., 2008). For each of the above-mentioned projects, Song et al. (2018) extracted 11 process features derived from Moser et al. (2008).

(3) Network features: Network features are generally used to quantitatively measure the dependency relationships between the various components of a software module. These features can be extracted from the dependency graph of each module, in which the nodes represent the components (such as the methods) in a class, and the edges represent the call dependencies between the components. Song et al. (2018) used two open-source tools (i.e., DependencyFinder[1] and UCINET[2]) to extract the network features. More specifically, DependencyFinder is first used to extract the call dependency graph of the components for each module, and the UCINET tool is then applied to calculate the network features from the graph. Song et al. (2018) extracted 24 features of three types of network measurements, including 12 Ego network measurements, four structural measurements and eight centrality measurements. The brief descriptions of the three types of measurements are as follows:

Ego network measurements: Each node (a method in the class corresponds to a node) has an ego network that contains itself (called ego), its neighbors (called alters), and some specific edges that present the call relationships between nodes. These edges only include those between the ego and the alters, and those among the alters.

[1] http://depfind.sourceforge.net/.
[2] http://www.analytictech.com/ucinet/.

Structural measurements: Burt (2009) proposed four features from the structural holes based on the ego network. The features are calculated for all nodes in the network by taking each in turn as the ego.

Centrality measurements: Centrality evaluates the importance of a node or edge for the connectivity or the information flow of the network.

The detailed meanings of the six CK features, 11 process features, and 24 network features are available on our online supplementary materials[3].

4.3. Evaluation indicators

A series of evaluation indicators has been proposed to measure the performance of various defect prediction methods, such as Accuracy, Recall, Precision, F-measure, and AUC. However, Accuracy is not an appropriate performance indicator for defect prediction. For example, given defect data with 90 non-defective modules and 10 defective modules and a SDP method that classifies all modules as non-defective, in this case, even though the Accuracy (90%) is high, this SDP method is meaningless because no defective modules are identified. Precision and Recall are fundamental indicators in SDP studies (Fu and Menzies, 2017). In general, increasing the Recall values, however, may result in an oscillating decline in the Precision values. We thus use the harmonic mean of Precision and Recall, that is, the F-measure, as our performance indicator. Greater Recall and Precision will lead to a higher F-measure. These indicators have been widely used in previous studies (Nam and Kim, 2015; Jing et al., 2014; Nam et al., 2013). Another commonly used performance indicator in SDP is the **A**rea **U**nder the **C**urve (**AUC**). However, Song et al. (2018) recently noted that AUC is not suitable for defect prediction to determine which specific SDP method should be selected. In addition, Vickers and Elkin (2006) verified that methods with different AUC values can be comparable and that methods with higher AUC values can sometime lead to inferior performance. Thus, we do not use AUC as the performance measurement in this study, however, we make the AUC results of our LDFR framework available in our online materials for comparison in future studies. Before introducing the Recall and F-measure, we first describe four typical outputs of a binary classification as follows: **T**rue **P**ositive (**TP**) denotes the number of defective modules that are correctly predicted; **T**rue **Negative** (**TN**) denotes the number of predicted non-defective modules that are correctly predicted; **F**alse **P**ositive (**FP**) denotes the number of predicted defective modules that are incorrectly predicted; and **F**alse **Negative** (**FN**) denotes the number of predicted non-defective modules that are incorrectly predicted.

Recall is defined as the ratio of defective modules that are correctly predicted to the total number of real defective modules, i.e.,

$$\text{Recall} = \text{TP}/(\text{TP} + \text{FN}). \tag{9}$$

Precision is defined as the ratio of defective modules that are correctly predicted to the total number of defective modules that are correctly and incorrectly predicted, i.e.,

$$\text{Precision} = \text{TP}/(\text{TP} + \text{FP}). \tag{10}$$

Then, F-measure is calculated as

$$\text{F-measure} = \frac{(1+\theta^2) \times \text{Precision} \times \text{Recall}}{\theta^2 \times \text{Precision} + \text{Recall}}, \tag{11}$$

where θ is a bias parameter toward Precision and Recall (i.e., to measure the relative importance of the two terms). In this work, we set θ as 2, which highlights Recall because we want to correctly detect as many real defective modules as possible, which is consistent with the definition of Recall. In addition, two types of misclassification errors are encountered in defect prediction: type I misclassification occurs when non-defective modules are predicted as defective ones, and type II misclassification occurs when defective modules are predicted as non-defective ones. Type I misclassification leads to a waste in test resources by checking the real non-defective modules, and type II misclassification is associated with risk cost by missing the real defective modules (Jing et al., 2014). The losses generated by type II misclassification are much higher than the type I misclassification, even though the latter may have a negative impact on the developers' trust on the approach. To reduce type II misclassification, we must reduce the FN value, which leads to greater Recall according to its definition. For these reasons, we follow previous studies (Jing et al., 2017; Jiang et al., 2008; Liparas et al., 2012; Li et al., 2018) in using the skewed F-measure with parameter $\theta = 2$.

However, these three indicators are traditional performance measurements that do not consider the inspecting efforts required in the software testing process because they assume that sufficient efforts are available to test all modules (Chen et al., 2015; Zimmermann et al., 2009; Turhan et al., 2009). To assess SDP performance in a more practical scenario in which efforts for testing are limited, Mende et al. Mende and Koschke (2010) proposed effort-aware performance indicators. In this study, we use three effort-aware indicators to evaluate the performance of our LDFR framework for SDP.

In quality assurance activity, the testers always desire the maximum profit within a certain amount of test efforts Arisholm et al. (2010); Kamei et al. (2013); Yang et al. (2016). Previous studies used 20% of LOC to define the test efforts involved in inspecting the software modules and treated the ratio of discovered defective modules as the profit (Yang et al., 2015; Xia et al., 2016; Jiang et al., 2013).

In this study, we refer to the study by Huang et al. (2017) to calculate the three effort-aware indicators. Fig. 5 depicts the calculation process. We describe the main steps as follows:

In step 1, a classification model is learned on the training set to predict the test set as two groups: the predicted defective group and the predicted non-defective group. In step 2, the modules in the two groups are ranked in ascending order according to their LOC values individually. In step 3, the two ranking results are merged and the predicted defective group is put in front of the other group. In step 4, the cumulative percentage of LOC is calculated until it reaches 20%. In step 5, some statistics are counted to calculate the indicators.

Here, we first define three basic terms: t_{nd} denotes the number of defective modules in the test set; t'_n denotes the number of modules that have been checked after inspecting 20% of LOC; and t'_{nd} denotes the number of real defective modules that have been discovered after inspecting 20% of LOC.

Based on the three statistics, the three effort-aware indicators are defined as follows

Effort-**A**ware **Recall** (**EARecall**) is defined as the ratio of the defective modules detected when inspecting 20% LOC to all real defective modules, i.e.,

$$\text{EARecall} = \text{t}'_{\text{nd}}/\text{t}_{\text{nd}}. \tag{12}$$

Effort-**A**ware **Precision** (**EAPrecision**) is defined as the ratio of the real defective modules that have been detected to all modules that have been checked when inspecting 20% LOC, that is,

$$\text{EAPrecision} = \text{t}'_{\text{nd}}/\text{t}'_{\text{n}}. \tag{13}$$

Then **E**ffort-**A**ware **F-measure** (**EAF-measure**) is defined as the harmonic mean of EAPrecison and EARecall. Like the traditional F-

[3] https://sites.google.com/view/jss-ldfr.

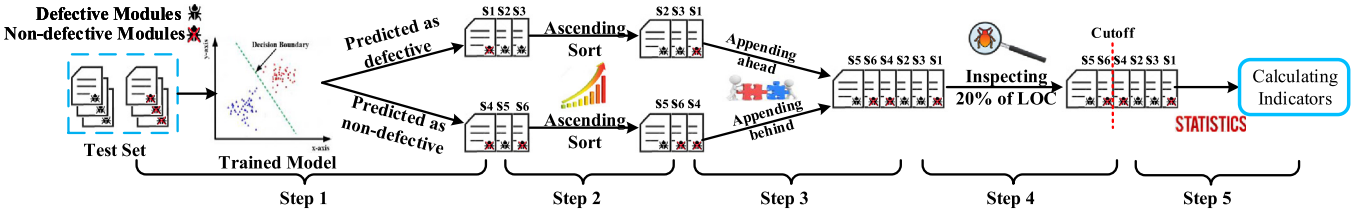

Fig. 5. The flow chart to calculate the effort-aware indicators.

measure, the general form of EAF-measure is formulated as:

$$\text{EAF-measure} = \frac{(1+\theta_1^2) \times \text{EAPrecision} \times \text{EARecall}}{\theta_1^2 \times \text{EAPrecision} + \text{EARecall}}. \quad (14)$$

In this study, we also set θ_1 as 2 for EAF-measure. In Fig. 5, the values of t_{nd}, t'_n, and t'_{nd} are equal to 3, 4, and 2, respectively. Thus, EARecall and EAPrecision are equal to 2/3 and 1/2, respectively.

4.4. Parameter configuration

To facilitate the reproduction of our experiments in future studies, we detail the network structure and parameter configuration of our LDFR framework as follows.

Our DNN model consists of one input layer, multiple hidden layers and one output layer. The inputs of DNN are the module features. The output layer has two nodes that denote the probability distribution for the module labels. The configuration for the hidden layers is important because a simple network design cannot promise enough learning capacity and a complicated network structure may lead to over-fitting. In this study, we focus mainly on the network structure design, that is, the number of hidden layers and the number of hidden nodes in each hidden layer that have a relatively larger impact on performance. In terms of network parameters (i.e., the weights and biases), the former are initialized with the Xavier method (Glorot and Bengio, 2010), which is a popular initialization technique for weights, and the latter are initialized with 0. In terms of the hyperparameters, we fix the batch size as 16, and assign the number of iterations to 20,000, in which we set the learning rate for the first 5000 iterations as 1e-5 and decay it to 1e-6 for the other 15,000 iterations.

In terms of the number of hidden layers and hidden nodes, we perform a case study on Project ant with version 1.3 to determine the two terms with a grid search based on F-measure. More specifically, we divide the data into two equal parts with a stratified sampling strategy and treat one part as the test set. We divide the other part again into two equal parts with a stratified sampling strategy and treat them as the training set and the validation set. We set the number of hidden layers with five options {1, 2, 3, 4, 5} and the number of hidden nodes with six options {5, 10, 15, 20, 25, 30}. We then use the grid search on the two sets of options to select the optimum options. We conduct 30 (5*6) experiments and found that the network structure with two hidden layers and 10 hidden nodes achieve the best overall performance. Considering that (1) the more hidden layers and hidden nodes, the greater the complexity of the model and the slower the model training; that (2) seeking the optimum parameters with a grid search for each defect data on each data partition is time-consuming; and that (3) the model with different parameters for distinct defect data shows no good generalization ability, in this study, we fix the neural network with two hidden layers and 10 hidden nodes for all defect data. Thus, we simply divide our defect data into two equal parts as the training set and test set, respectively, without a validation set for parameter-tuning in our experiments.

In terms of the weight in the weighted cross-entropy loss, we set the weight of classifying a real defective module as non-defective as double that of classifying a real non-defective module as defective. The source code is publicly available in our online supplementary materials.

4.5. Dataset partition

In our experiment, we use a 50/50 split with the stratified sampling strategy to divide the original data into training and test sets. More specifically, we randomly sample half of the defective and non-defective modules as the training set and use the remaining half of the modules as the test set. This process ensures that the number of defective and non-defective modules in the two sets are the same and helps to reduce sampling biases by avoiding sampling of a set with only non-defective modules. This sampling strategy is commonly adopted in previous SDP settings (Wang et al., 2016b; Jing et al., 2014; Hryszko et al., 2017; Ryu et al., 2016). To reduce the effects of random division on the experimental results, we repeat the sampling process 30 times and report the average value for each indicator.

4.6. Basic classifier

In this study, we select the random forest as our basic classifier which is widely-used in previous studies (Yang et al., 2017; Tantithamthavorn et al., 2018a; 2018b). In the model building phase, random forest constructs multiple decision trees with sampled module subsets from the training set. In the model application phase, random forest feeds the new module into the constructed decision trees and the module label is determined by the majority voting of the outputs of these decision trees. In our study, the final output of the last hidden layer is used as the input to the random forest classifier because this layer outputs the learned feature representation.

4.7. Statistical test

To carry out a statistical comparison of the differences in performance among our method (LDFR) and the compared baseline methods, we follow previous studies (Zhang et al., 2016; Tantithamthavorn et al., 2017; Xu et al., 2016; Ghotra et al., 2015; Li et al., 2017b) in using a state-of-the-art statistical test called the Scott-Knott test. Unlike several commonly used post hoc test methods (such as Nemenyi's test (Demšar, 2006)) that have a confounding issue by partitioning multiple methods into overlapping groups (Ghotra et al., 2015), the Scott-Knott test exploits a hierarchical clustering algorithm to partition the methods into significantly different groups (significance level $\alpha = 0.05$). The methods within the same group have no significant differences, whereas the methods across groups include significant differences. In this study, we use a variant of the Scott-Knott test called **Scott-Knott E**ffect **S**ize **D**ifference (**Scott-Knott ESD**) (Tantithamthavorn et al., 2017) for significance analysis. The Scott-Knott ESD corrects the non-normal distribution of the test inputs and merges the groups that have statistical differences with a negligible effect size into one group. In this study, we conduct a two-round Scott-Knott ESD: in the first round, the Scott-Knott ESD is applied to each project with the inputs of 30 indicator values of each method. As each

method receives a ranking on each project, we can obtain 27 rankings for each method after this round; in the second round, the Scott-Knott ESD is applied to all methods with the inputs of the 27 rankings and then outputs the overall ranking for each method and the corresponding group. The method with lower ranking performs better.

4.8. Experimental environment

We conduct the experiments on our server, which is equipped with a 16-core Intel Xeon E5-2620@2.1GHz CPU, 16RAM, 512GB SSD, and two 1080Ti GPUs. We implement the DNN using Pytorch 0.3.0 under Ubuntu 18.04.1 LTS.

5. Evaluation result

Learning discriminative feature representation that can easily distinguish the two classes indicates that most of modules in both classes will be correctly predicted. From this perspective, feature representation learning is also helpful to address the class imbalance issue. Thus, in this study, we mainly investigate whether our LDFR framework works better than other imbalance learning methods for SDP. We empirically design the following two research questions.

5.1. RQ1: How effective is our combined loss function compared with its downgraded loss functions?

Motivation: As described in Section 3.4, our LDFR framework uses a novel hybrid loss function that consists of a triplet loss and a weighted cross-entropy loss for DNN training to learn the deep feature representation of the defect data. This question is proposed to investigate whether this kind of combined loss function can more effectively achieve better SDP performance than some of its variant loss functions.

Method: To answer this question, we design three baseline methods with different downgraded loss functions to learn the feature representation for the defect data. In addition, we also use the method without feature representation learning as the most basic method for comparison. The four baseline methods are described as follows. **NONE** method means that we use only the random forest classifier without any feature learning process to conduct SDP on the original defect data. We treat this method as a special variant of LDFR. **CE** method uses the original **C**ross-**E**ntropy loss (i.e., the unweighted version) to train the DNN for feature representation learning. **WCE** method applies the **W**eighted **C**ross-**E**ntropy loss to train the DNN for feature representation learning. **CET** method combines the unweighted **C**ross-**E**ntropy loss with the **T**riplet loss function to train the DNN for feature representation learning.

Because each project studied consists of CK metrics, process metrics, and network metrics, for each research question, we analyze the experimental results on defect data with each type of metric individually for each research question.

Results:

5.1.1. The results of LDFR and its four variants on defect data with CK metrics

Fig. 6 depicts the radar chart of average values of the six indicators for our LDFR framework and its four variants under defect data with CK metrics. The radar chart is a graphical representation used to display multivariate data (i.e., the average indicator values in this study) in a two-dimensional chart in which the variables represented on the axes begin from the same center point with a value of 0. The value of each variable is depicted by the node on the axis. A line is drawn connecting the nodes to form a polygon corresponding to a method with a specific color. The radar chart easily shows all average indicator values for multiple methods at once. The detailed results for the methods are available on our online supplementary materials. From this figure, it can be seen that LDFR achieves the best average values in terms of five indicators (all except Precision) across the 27 defect data, whereas the average Precision by LDFR is only inferior to that by the NONE method. Compared with the best average indicator values among the four baseline methods, LDFR achieves average improvements of 7.6%, 13.5%, 15.3%, 7.1%, and 11.6% in terms of Recall, F-measure, EAPrecision, EARecall, and EAF-measure, respectively.

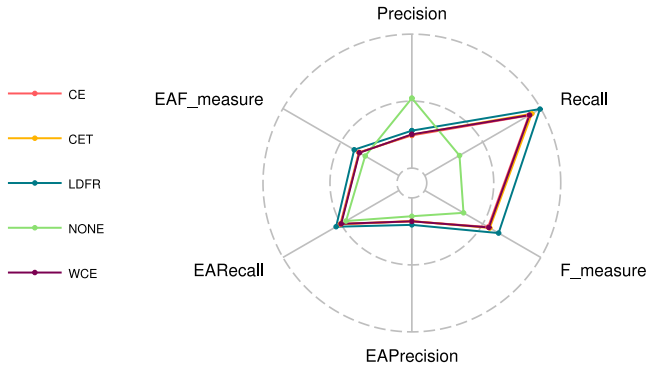

Fig. 6. Radar charts of average indicator values for LDFR and its four variants under defect data with CK metrics.

Fig. 7 visualizes the corresponding results of the Scott-Knott ESD for each indicator. The methods that have significant differences are drawn in different colors. The figure shows that LDFR ranks first and performs significantly better than all baseline methods in terms of five indicators (all except Precision) and ranks second and performs significantly better than three baseline methods (all except the NONE method) in terms of Precision.

5.1.2. The results of LDFR and its four variants on defect data with process metrics

Fig. 8 depicts the results of radar chart for these five methods under defect data with process metrics. From this figure, it can be seen that LDFR obtains the best average values in terms of five indicators (all except Precision), whereas the average Precision by LDFR is only worse than that by the NONE method. Compared with the best average indicator values among the four baseline methods, LDFR achieves average improvements of 5.1%, 11.5%, 16.0%, 9.8%, and 12.6% in terms of Recall, F-measure, EAPrecision, EARecall, and EAF-measure, respectively.

Fig. 9 visualizes the corresponding statistical test results for each indicator. Fig. 9 shows that LDFR ranks first and has significant differences in performance toward all baseline methods in terms of five indicators (all except Precision), whereas ranks second and has significant differences in performance toward three baseline methods (all except the NONE method) in terms of Precision.

5.1.3. The results of LDFR and its four variants on defect data with network metrics

Fig. 10 depicts the results of radar chart for these five methods under defect data with network metrics. From this figure, it can be seen that LDFR obtains the best average values in terms of five indicators (all except Precision), whereas the average Precision by LDFR is only lower than that by the NONE method. Compared with the best average indicator values among the four baseline methods, LDFR achieves average improvements of 6.4%, 10.1%, 14.7%, 7.9%, and 11.8% in terms of Recall, F-measure, EAPrecision, EARecall, and EAF-measure, respectively.

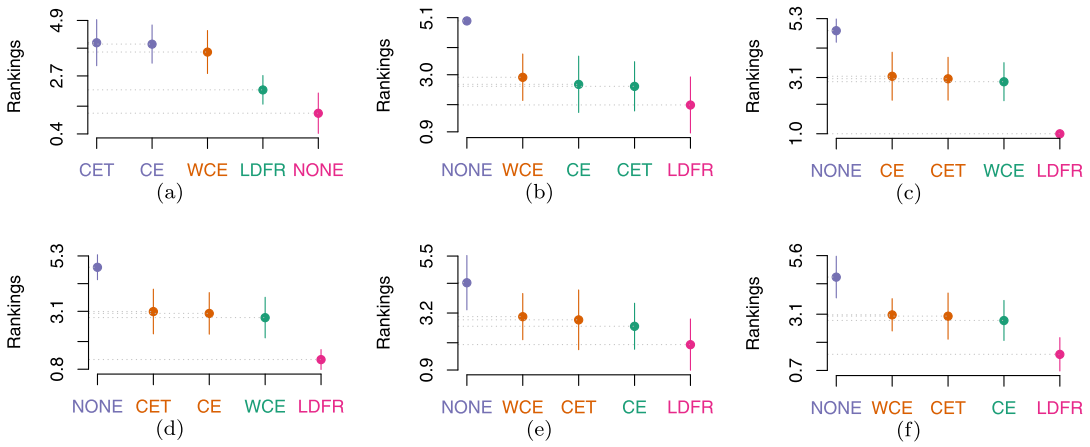

Fig. 7. Scott-Knott ESD test for LDFR and its four variants on defect data with CK metrics. (a) Precision. (b) Recall. (c) F-measure. (d) EAPrecision. (e) EARecall. (f) EAF-measure.

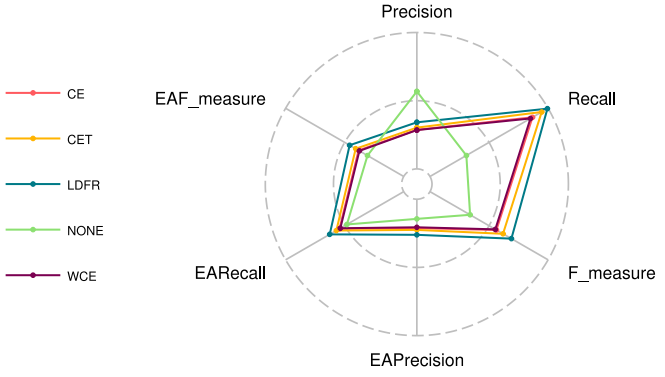

Fig. 8. Radar charts of average indicator values for LDFR and its four variants under defect data with process metrics.

Fig. 11 visualizes the corresponding statistical test results for each indicator. Fig. 11 shows that LDFR ranks first and is significantly superior to all baseline methods in terms of five indicators (all except Precision), whereas ranks second and is significantly superior to three baseline methods (all except the NONE method) in terms of Precision.

Analysis: From the above observations, it can be seen that our method, LDFR, combining the triplet loss and weight cross-entropy loss is superior to the variant methods overall under defect data with three types of metrics (all except the NONE method) in terms of Precision. Compared against the NONE method with lower Recall and higher Precision, LDFR with higher Recall and lower Precision means that more real defective modules are correctly identified, whereas more real non-defective are incorrectly identified. The reason is that LDFR that considers the class imbalance of the data is biased to the identification of minority class modules, which leads to a certain number of real non-defective modules being incorrectly identified, whereas the NONE method does not consider class imbalance, thus less majority class modules and more minority class modules will be incorrectly identified, which causes higher Precision but lower Recall. Among CE, CET, and WCE, the three groups of results show that CET is superior to other two methods overall in most cases. The reason why our method, LDFR, is superior to CET is that LDFR uses a weighted cross-entropy loss to further make the learned features bias to identifying minority class modules. The reason why LDFR outperforms WCE is that LDFR further improves the representation ability of the features by using a metric learning technique. It also implies the necessity of using the hybrid loss function. The fact that CET is superior to CE also implies that the feature learning process can further promote the performance improvement. In addition, the fact that CET is superior to WCE implies that the introduction of the feature representation is more effective than the weighting scheme.

In summary, LDFR shows the superiority on defect prediction performance compared with its downgraded version in terms

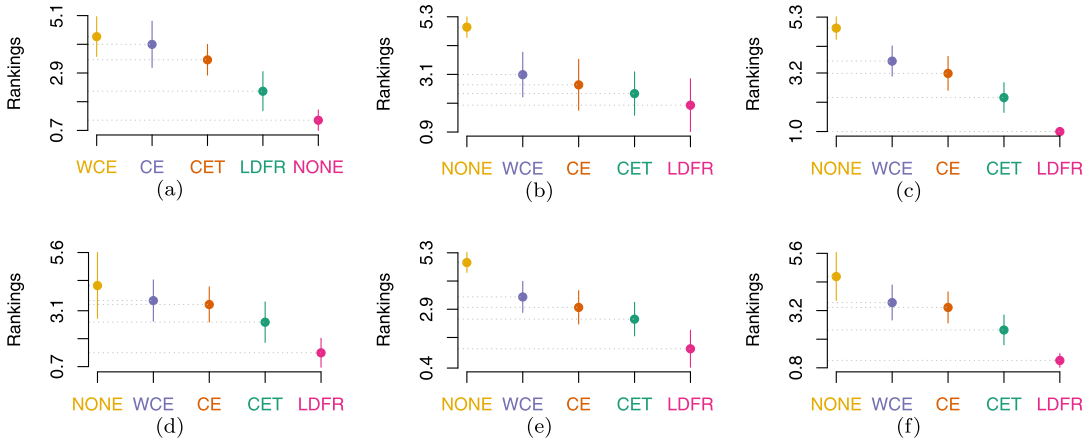

Fig. 9. Scott-Knott ESD test for LDFR and its four variants on defect data with process metrics. (a) Precision. (b) Recall. (c) F-measure. (d) EAPrecision. (e) EARecall. (f) EAF-measure.

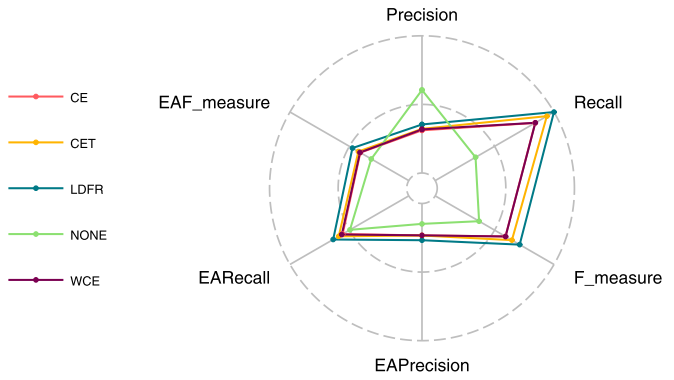

Fig. 10. Radar charts of average indicator values for LDFR and its four variants under defect data with network metrics.

of five indicators (all except Precision), whereas the most basic method, NONE, performs the best only in terms of Precision on three types of defect data.

5.2. RQ2: Is our framework LDFR more effective than exiting imbalanced learning methods for SDP?

Motivation: There are various method families for imbalanced data, such as sampling-based, ensemble-based, and cost-sensitive-based methods. This question is designed to explore whether our LDFR framework is better than previous imbalanced learning methods in improving the prediction performance on the imbalanced defect data.

Method: To answer this question, we select six sampling-based, six ensemble-based, two statistical models, nine cost-sensitive-based methods. The sampling-based methods include **R**andom **U**nder-**S**ampling (**RUS**), **R**andom **O**ver-**S**ampling (**ROS**), **S**ynthetic **M**inority **O**versampling **TE**chnique (**SMOTE**) (Chawla et al., 2002), **ADA**ptive **SYN**thetic (**ADASYN**) sampling (He et al., 2008), **SMO** with **E**dited **N**earest **N**eighbors (**SMOENN**) (Batista et al., 2004), and **SMO** with **T**omek **L**inks (**SMOTL**) (Batista et al., 2004). The ensemble-based methods consist of **Bag**ging (**Bag**), **B**alanced**Bag**ging (**BBag**), **AdaB**oost (**AdaB**), **RUS** with Ada**B**oost (**RUSB**), **EasyE**nsemble (**EasyE**), **Bal**ance**C**ascade (**BalC**). The two statistical models are **P**artial **L**east **S**quares **C**lassifier (**PLSC**) and **A**symmetric **PLSC** (**APLSC**). The nine cost-sensitive-based methods are derived from the combinations of three different decision forest methods (i.e., a **S**ystematically developed **F**orest of multiple decision trees (**SF**) (Islam and Giggins, 2011), **C**ost-**S**ensitive decision **F**orest (**CSF**) (Siers and Islam, 2015), and **B**alanced **CSF** (**BCSF**) (Siers and Islam, 2015)) and three different voting strategies (i.e., **C**ascading-and-sharing based **V**oting (**CV**)(Li and Liu, 2003), **M**aximally diversified multiple decision tree based **V**oting (**MV**) (Hu et al., 2006), and **C**ost-**S**ensitive **V**oting (**CSV**) (Siers and Islam, 2015)). Combining each decision forest method with a voting strategy, we obtain nine total baseline methods, that is, SFCV, SFMV, SFCSV, CSFCV, CSFMV, CSFCSV, BCSFCV, BCSFMV, and BCSFCSV. The source code of the nine methods was provided by Siers and Islam (2015).

Results:

5.2.1. The results of LDFR and 23 imbalanced learning methods on defect data with CK metrics

Fig. 12 depicts the results of radar chart for LDFR and 23 baseline methods under defect data with CK metrics. To observe the average values more clearly, we use three radar charts to present the results. From Fig. 12(a), compared with six sampling-based methods, LDFR gets the best average values in terms of 5 indicators (all except Precision), whereas the average Precision by LDFR is lower than that by other methods. Compared with the best average indicator values among the six methods, LDFR achieves average improvements of 48.3%, 14.7%, 45.2%, 17.7%, and 31.5% in terms of Recall, F-measure, EAPrecision, EARecall, and EAF-measure, respectively. From Fig. 12(b), compared with six ensemble-based methods and two statistic models, LDFR gets the best average values in terms of F-measure, EARecall, and EAF-measure, whereas the average Precision, Recall, and EAPrecision by LDFR are lower than that by seven, one, and three baseline methods, respectively. Compared with the best average indicator values among the eight methods, LDFR achieves average improvements of 5.5%, 17.4%, and 23.2% in terms of F-measure, EARecall, and EAF-measure, respectively. From Fig. 12(c), compared with 9 cost-sensitive-based methods, LDFR gets the best average values in terms of five indicators (all except Precision), whereas the average Precision by LDFR is lower than that by other methods. Compared with the best average indicator values among the nine methods, LDFR achieves average improvements of 59.5%, 15.6%, 46.4%, 16.5%, and 31.5% in terms of Recall, F-measure, EAPrecision, EARecall, and EAF-measure, respectively.

Fig. 13 visualizes the corresponding statistical test results for each indicator. Fig. 13 shows that LDFR ranks first in terms of Recall, F-measure, EARecall, and EAF-measure and is significantly superior to all baseline methods (all except BalC in terms of Recall), whereas ranks third and is not significantly superior to two statistical models and BalC in terms of EAPrecision.

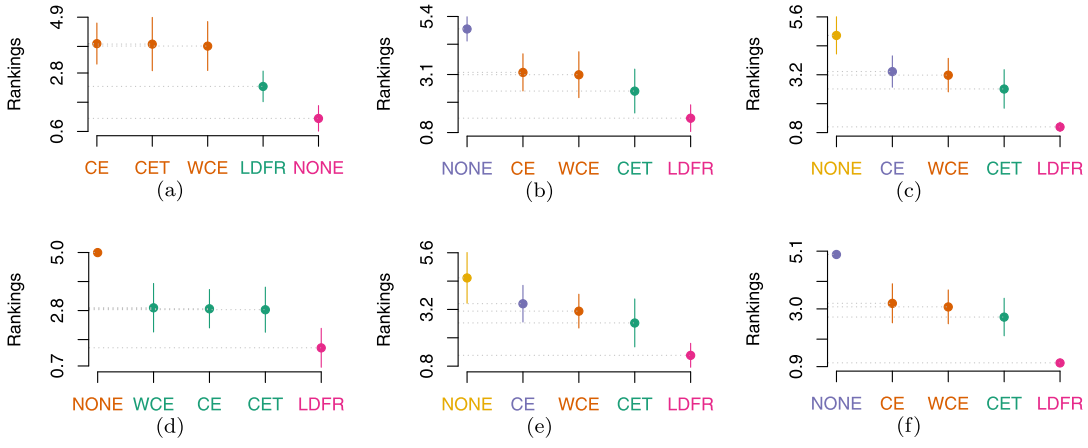

Fig. 11. Scott-Knott ESD test for LDFR and its four variants on defect data with network metrics. (a) Precision. (b) Recall. (c) F-measure. (d) EAPrecision. (e) EARecall. (f) EAF-measure.

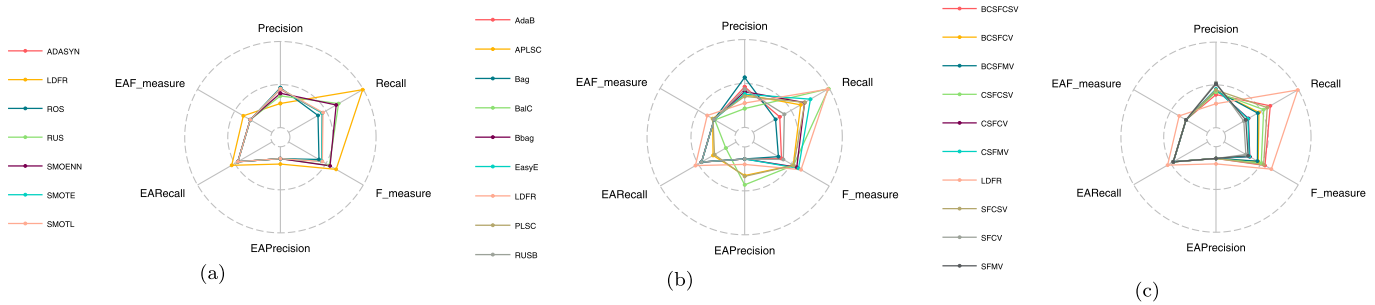

Fig. 12. Radar charts of the six average indicator values across 27 project versions with CK features. (a) LDFR and six sampling-based methods. (b) LDFR, six ensemble-based methods and two statistical models.(c) LDFR and nine cost-sensitive based methods.

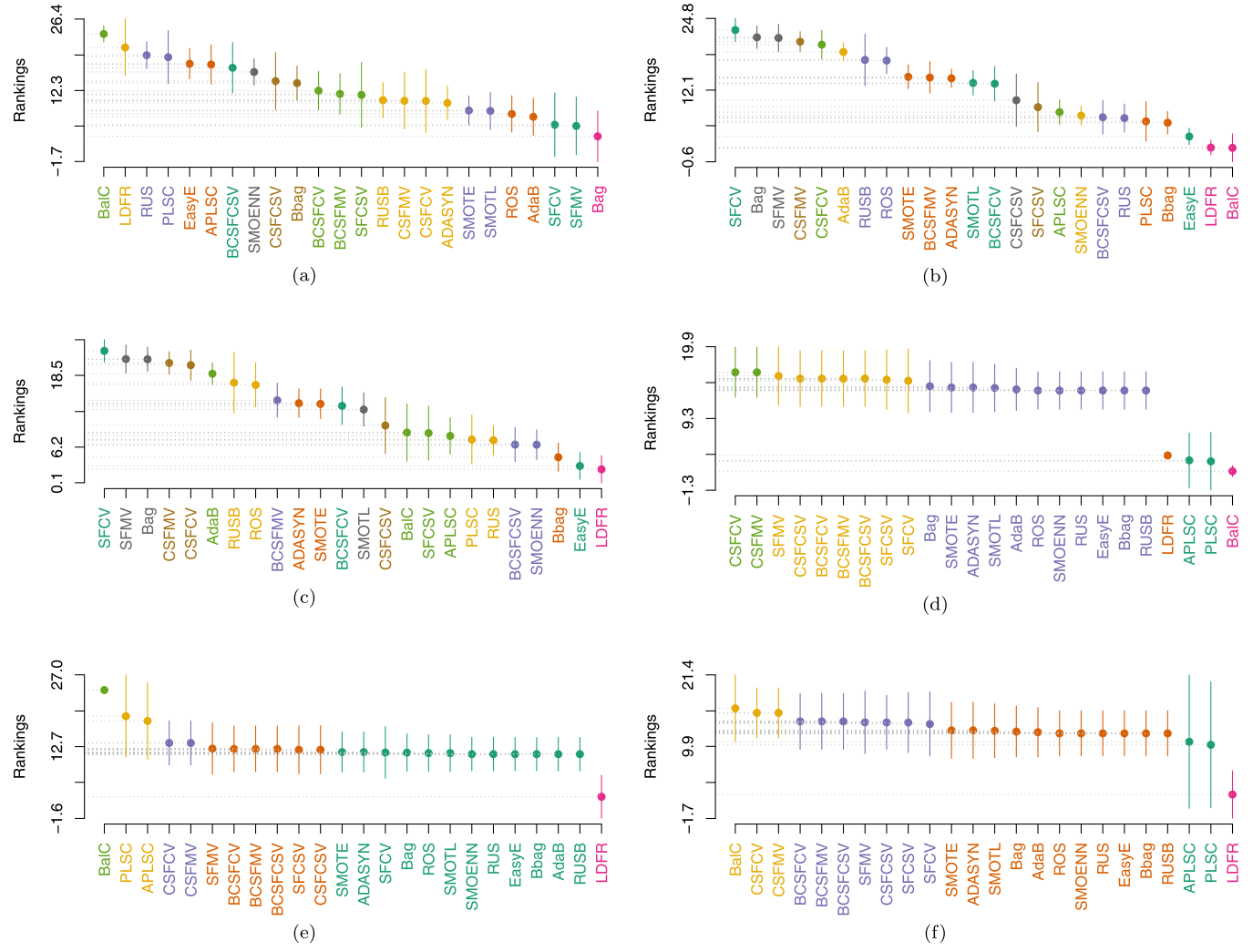

Fig. 13. Scott-Knott ESD test for LDFR and 23 imbalanced learning methods on defect data with CK metrics. (a) Precision. (b) Recall. (c) F-measure. (d) EAPrecision. (e) EARecall. (f) EAF-measure.

5.2.2. The results of LDFR and 23 imbalanced learning methods on defect data with process metrics

Fig. 14 depicts the results of radar chart for these 24 baseline methods under defect data with process metrics. From Fig. 14(a), compared with six sampling-based methods, LDFR obtains the best average values in terms of five indicators (all except Precision), whereas the average Precision by LDFR is inferior to that by other methods. Compared with the best average indicator values among the six methods, LDFR achieves average improvements of 40.4%, 18.6%, 81.6%, 29.8%, and 50.0% in terms of Recall, F-measure, EAPrecision, EARecall, and EAF-measure, respectively. From Fig. 14(b), compared with 6 ensemble-based methods and two statistic models, LDFR obtains the best average values in terms of F-measure, EARecall, and EAF-measure, whereas the average Precision, Recall, and EAPrecision by LDFR are lower than that by six, one, and three baseline methods, respectively. Compared with the best average indicator values among the eight methods, LDFR achieves average improvements of 13.2%, 29.8%, and 22.4% in terms of F-measure, EARecall, and EAF-measure, respectively. From Fig. 14(c), compared with 9 cost-sensitive-based methods, LDFR obtains the best average values in terms of five indicators (all except Precision), whereas the average Precision by LDFR is lower than that by other methods.

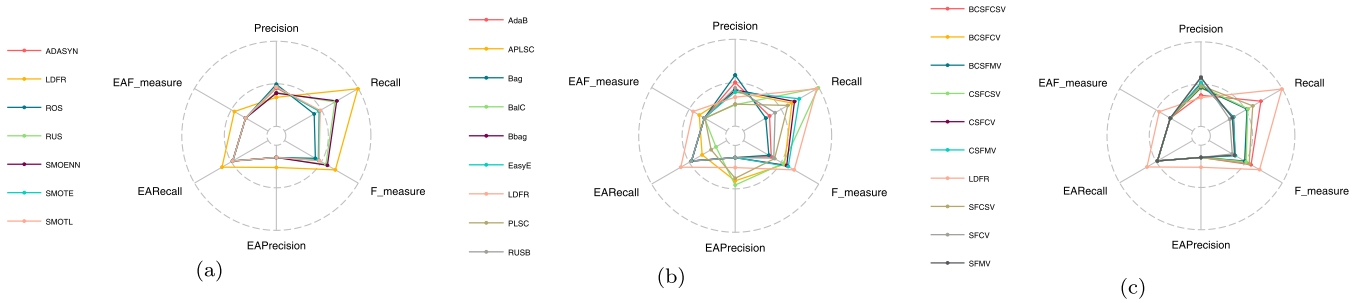

Fig. 14. Radar charts of the 6 average indicator values across 27 project versions with process features. (a) LDFR and six sampling-based methods. (b) LDFR, six ensemble-based methods and two statistical models.(c) LDFR and nine cost-sensitive-based methods.

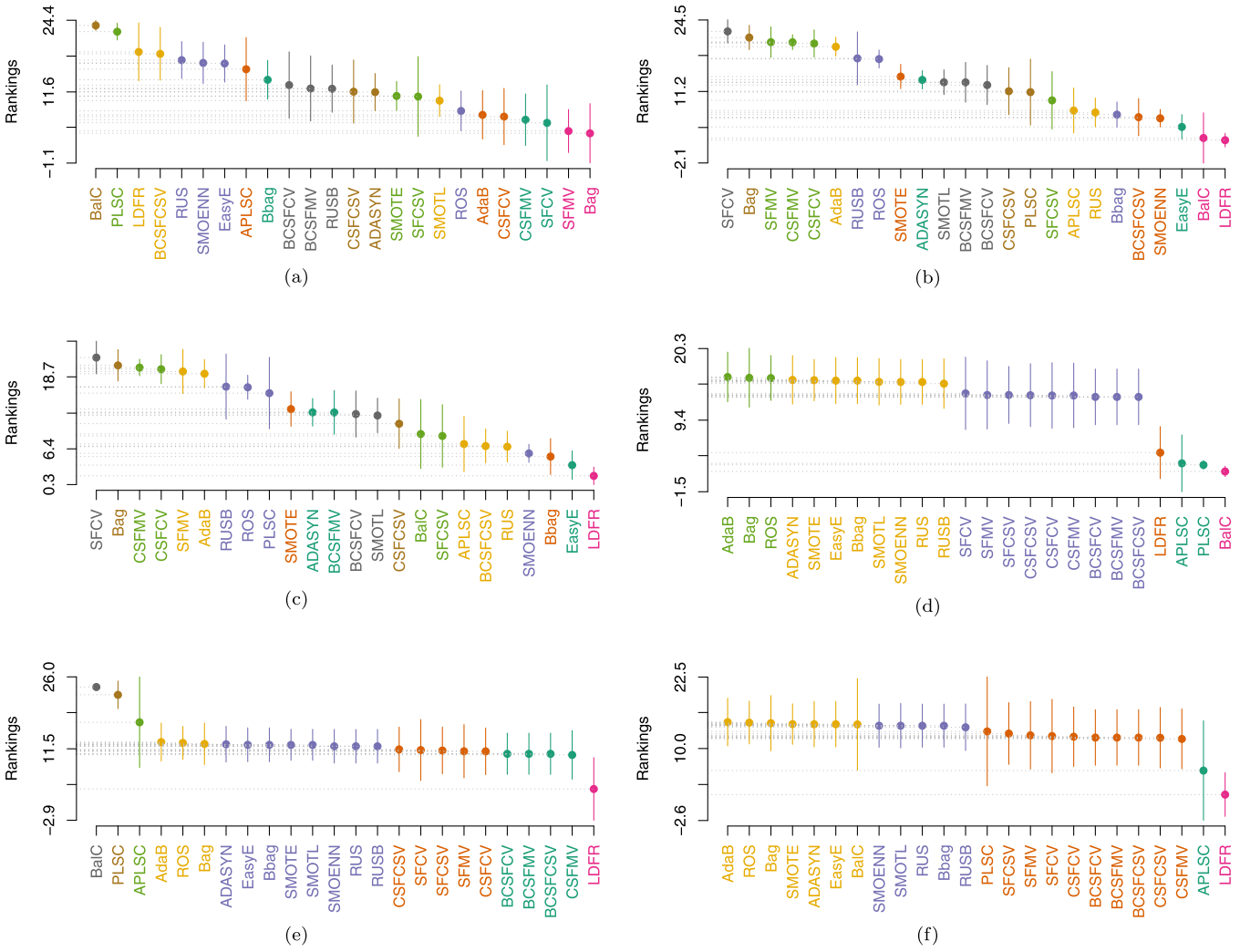

Fig. 15. Scott-Knott ESD test for LDFR and 23 imbalanced learning methods on defect data with process metrics. (a) Precision. (b) Recall. (c) F-measure. (d) EAPrecision. (e) EARecall. (f) EAF-measure.

Compared with the best average indicator values among the nine methods, LDFR achieves average improvements of 40.4%, 20.2%, 80.2%, 29.5%, and 48.9% in terms of Recall, F-measure, EAPrecision, EARecall, and EAF-measure, respectively.

Fig. 15 visualizes the corresponding statistical test results for each indicator. Fig. 15 shows that LDFR ranks first in terms of Recall, F-measure, EARecall, and EAF-measure, and performs significantly better than all baseline methods (all except BalC in terms of Recall), whereas ranks third and is not significantly superior to two statistical models and BalC in terms of EAPrecision.

5.2.3. The results of LDFR and 23 imbalanced learning methods on defect data with network metrics

Fig. 16 depicts the results of radar chart for these 24 baseline methods under defect data with network metrics. From Fig. 16(a), compared with six sampling-based methods, LDFR obtains the best average values in terms of five indicators (all except Precision), whereas the average Precision by LDFR is inferior to that by other methods. Compared with the best average indicator values among the six methods, LDFR achieves average improvements of 33.3%, 13.7%, 78.6%, 27.7%, and 49.1% in terms of Re-

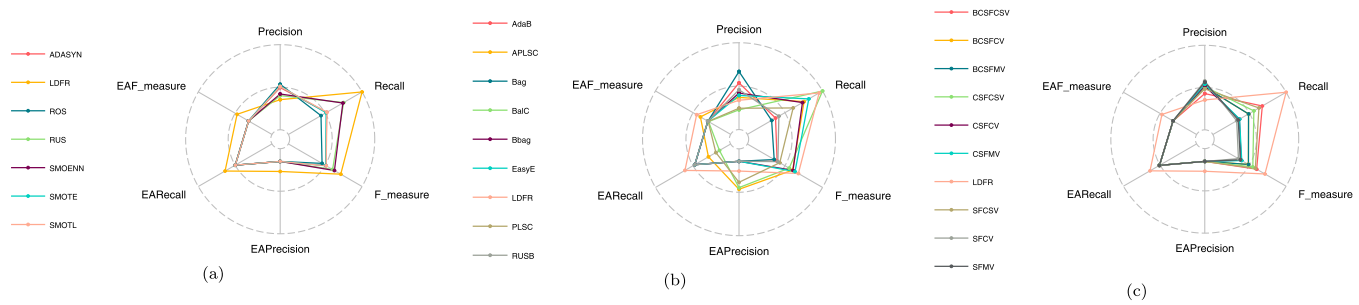

Fig. 16. Radar charts of the 6 average indicator values across 27 project versions with network features. (a) LDFR and six sampling-based methods. (b) LDFR, six ensemble-based methods and two statistical models.(c) LDFR and nine cost-sensitive-based methods.

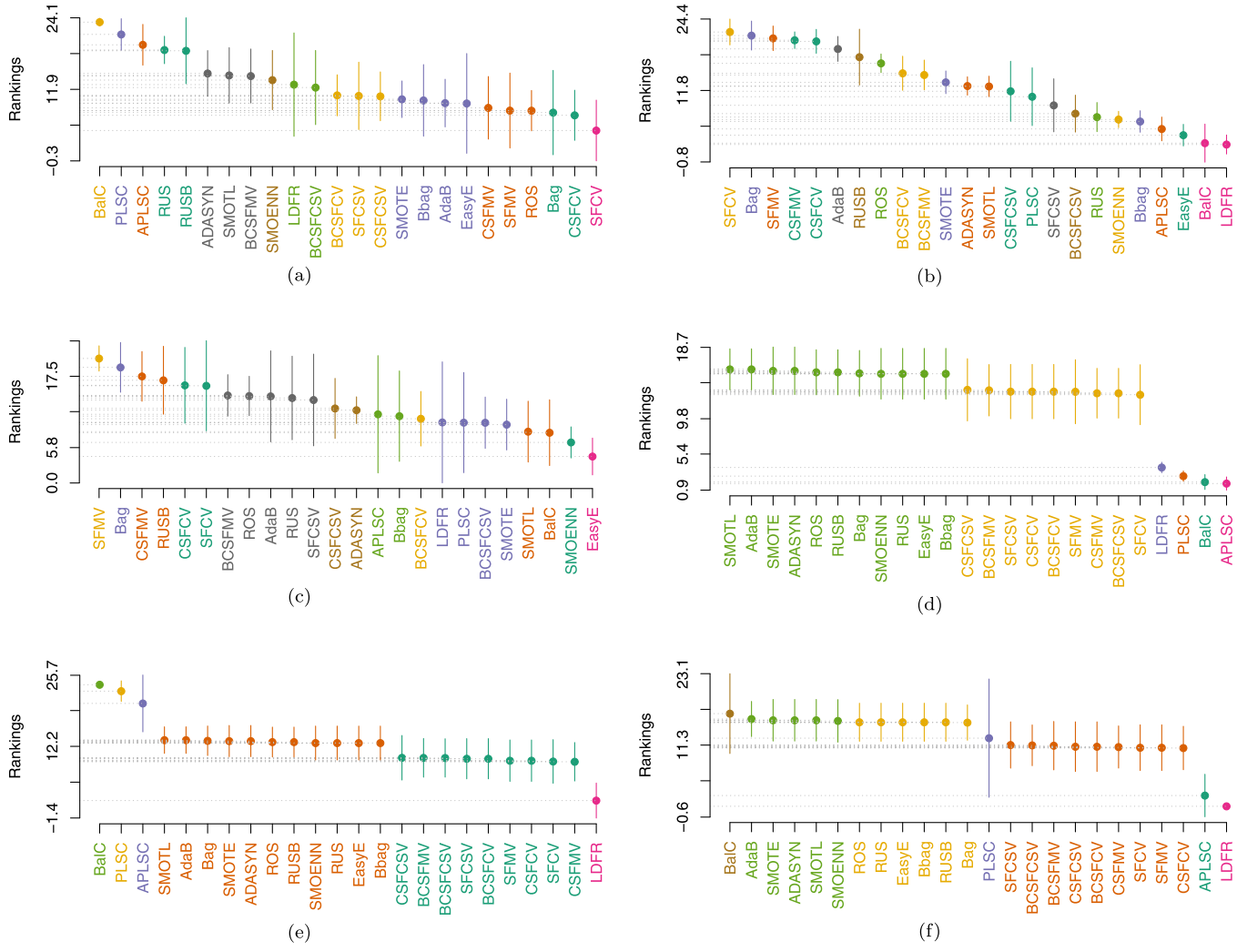

Fig. 17. Scott-Knott ESD test for LDFR and 23 imbalanced learning methods on defect data with network metrics. (a) Precision. (b) Recall. (c) F-measure. (d) EAPrecision. (e) EARecall. (f) EAF-measure.

call, F-measure, EAPrecision, EARecall, and EAF-measure, respectively. From Fig. 16(b), compared with six ensemble-based methods and two statistical models, LDFR obtains the best average values in terms of F-measure, EARecall, and EAF-measure, whereas the average Precision, Recall, and EAPrecision by LDFR are lower than that by six, one, and three baseline methods, respectively. Compared with the best average indicator values among the eight methods, LDFR achieves average improvements of 7.0%, 27.4%, and 12.4% in terms of F-measure, EARecall, and EAF-measure, respectively. From Fig. 16(c), compared with nine cost-sensitive-based methods, LDFR obtains the best average values in terms of five indicators (all except Precision), whereas the average Precision by LDFR is lower than that by other methods. Compared with the best average indicator values among the nine methods, LDFR achieves average improvements of 48.1%, 18.9%, 77.2%, 25.3%, and 46.9% in terms of Recall, F-measure, EAPrecision, EARecall, and EAF-measure, respectively.

Fig. 17 visualizes the corresponding statistical test results for each indicator. Fig. 17 shows that LDFR ranks first in terms of Recall, EARecall, and EAF-measure and performs significantly bet-

ter than all baseline methods (all except BalC in terms of Recall), whereas ranks fourth in terms of F-measure and EAPrecision.

Analysis: From the above observations, it can be seen that our method LDFR performs better than the 23 baseline methods overall under defect data with three types of metrics except in terms of Precision. Unlike the sampling-based methods that change the data distribution of the original data to re-balance the module numbers of two classes, LDFR uses a weighted loss function to make the identification bias to the minority class modules and maintains the original data distribution. Unlike the ensemble-based methods that rely on a set of classification decisions with multiple iterations, LDFR focuses on learning new feature representation with powerful discriminant ability for the classification model. Compared with the nine cost-sensitive-based methods that are mainly based on the original feature set, LDFR has an additional feature refining process by the metric learning technique. In addition, among the six sampling-based methods, they achieve similar average performance in terms of three effort-aware indicators, and RUS and SMOENN achieve better Recall and F-measure than the others on three types of defect data. Among the six ensemble-based methods and two statistical models, they also obtain similar average performance in terms of three effort-aware indicators for most methods, whereas five new imbalanced learning methods (i.e., BBag, EasyE, BalC, PLSC, and APLSC) get better Recall and F-measure than the traditional methods (like Bag and AdaB) on three types of defect data. Among the nine cost-sensitive-based methods, they also get similar average performance in terms of three effort-aware indicators, whereas the three different decision forest methods with the cost-sensitive voting scheme perform better than those with other two schemes. This implies that the cost-sensitive voting scheme is more effective to improve the defect prediction performance.

Table 2
The Parameter Settings of the 6 Basic Classifiers.

Classifier	Parameter settings
RF	Number of trees in the forest: 10
NB	Distribution Type: kernel density estimation
LR	Norm used in the penalization: l_2 penalties
DT	Minimum module number splitting an internal node: 2
NN	Number of neighbors to query: 8
SVM	Penalty parameter of the error term: 1; kernel type: rbf

Overall, LDFR outperforms nearly all baseline methods in terms of three effort-aware indicators, and is superior to most baseline methods in terms of Recall and F-measure, but is inferior to most baseline methods in terms of Precision on three types of defect data.

6. Discussion

6.1. Performance on various classifiers

We use the random forest classifier as our basic prediction model. In this subsection, we discuss the effects of various classifiers on the performance of our LDFR framework for SDP. For this purpose, we choose five more commonly used machine learning classifiers for comparison: **N**aive **B**ayesian (**NB**), **L**ogistic **R**egression(**LR**), **D**ecision **T**ree (**DT**), **N**earest **N**eighbour (**NN**), and **S**upport **V**ector **M**achine (**SVM**). Table 2 lists the parameter setting of the 6 basic classifiers.

Figs. 18 –20 visualize the corresponding statistical test results for each indicator under defect data with CK metrics, process metrics, and network metrics, respectively. Note that the method term

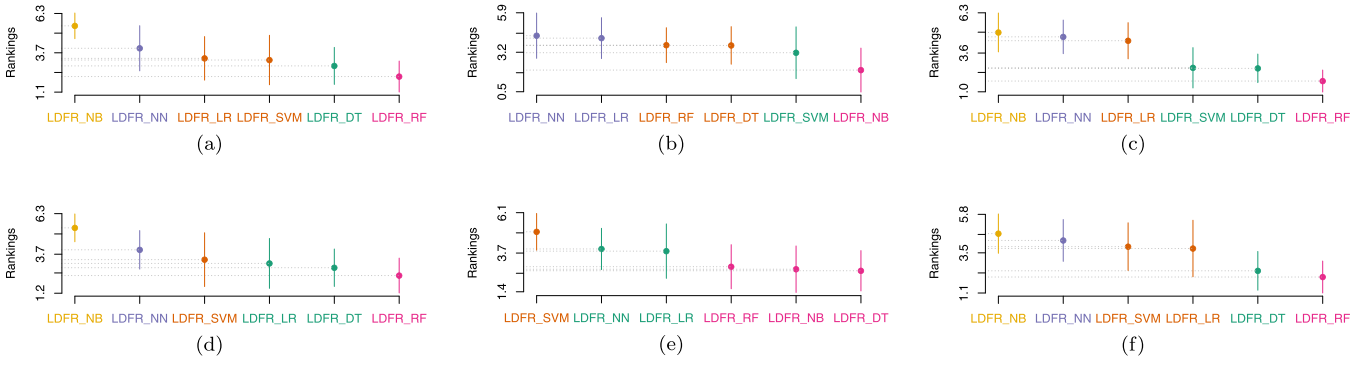

Fig. 18. Scott-Knott ESD test for LDFR with different classifiers on defect data with CK metrics. (a) Precision. (b) Recall. (c) F-measure. (d) EAPrecision. (e) EARecall. (f) EAF-measure.

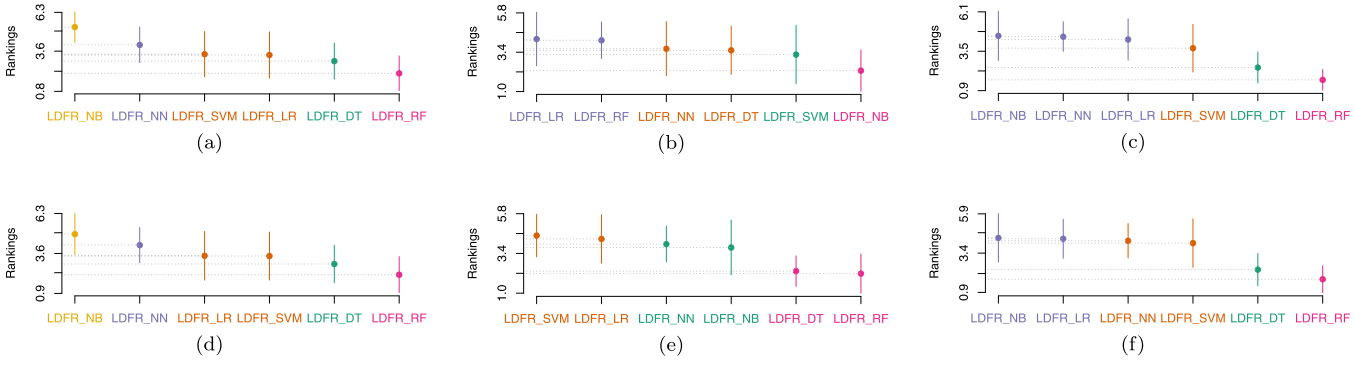

Fig. 19. Scott-Knott ESD test for LDFR with different classifiers on defect data with process metrics. (a) Precision. (b) Recall. (c) F-measure. (d) EAPrecision. (e) EARecall. (f) EAF-measure.

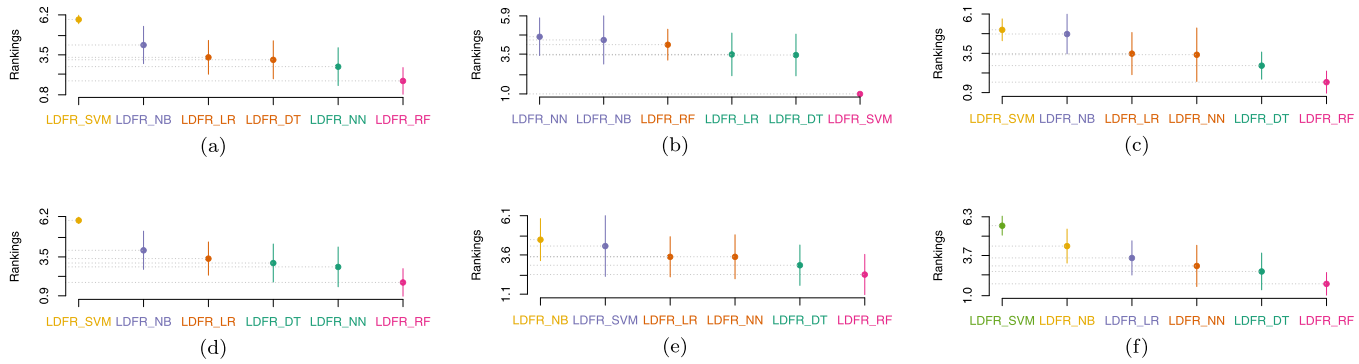

Fig. 20. Scott-Knott ESD test for LDFR with different classifiers on defect data with network metrics. (a) Precision. (b) Recall. (c) F-measure. (d) EAPrecision. (e) EARecall. (f) EAF-measure.

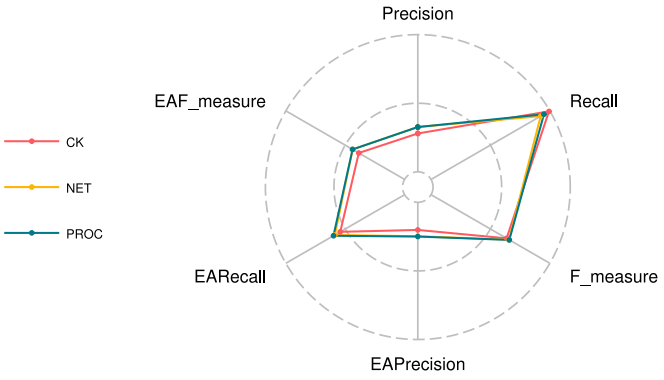

Fig. 21. Radar charts of average indicator values for LDFR under defect data with 3 types of metrics.

'LDRF_NB' in the figures denotes that the NB classifier is used as the basic classifier for our LDFR method. From Fig. 18, it can be seen that on defect data with CK metrics, LDFR_RF ranks first in terms of Precision, F-measure, EAPrecision, EARecall, and EAF-measure; performs significantly better than all baseline methods (all except LDFR_DT and LDFR_NB in terms of EARecall); and ranks fourth in terms of Recall. From Fig. 19, it can be seen that on defect data with process metrics, LDFR_RF ranks first in terms of 5 indicators (all except Recall); performs significantly better than all baseline methods (all except LDFR_DT in terms of EARecall); and ranks third in terms of Recall. From Fig. 20, it can be seen that on defect data with network metrics, LDFR_RF ranks first in terms of 5 indicators (all except Recall); performs significantly better than all baseline methods; and ranks third in terms of Recall.

Overall, LDFR_RF achieves better average indicator values than the other five methods (all except Recall). Thus, the random forest classifier is an ideal choice for our feature representation learning method LDFR.

6.2. Performance on different types of metrics

Fig. 21 depicts the radar chart of average values of the six indicators for our LDFR framework under defect data with three types of metrics. Fig. 21 shows that the performance on defect data with process metrics and network are very similar in terms of six indicators and they are higher than the performance on defect data with CK metrics in terms of five indicators (all except Recall). This implies that LDFR tends to achieve better Recall values on defect data with CK metrics and obtains better other five indicator values on defect data with process and network metrics.

7. Threats to validity

7.1. External validity

The threats to external validity focus on the generalization of the experimental results. To mitigate these external threats, we choose a publicly available open-source benchmark dataset that was provided in a top journal in the software engineering domain. This dataset contains 27 software project versions, and three types of metrics were collected for each project. Thus, conducting experiments on such a large dataset reinforces the representativeness of our conclusions. However, the projects studied were all developed with the Java language. Thus, replication experiments on the projects developed with other programming languages, such as C, C++, and Python, may prove fruitful.

7.2. Internal validity

The threats to internal validity concern the faults that may occur in the implementation of the studied methods. To minimize the internal validity, we implement our LDFR framework by pair programming in an attempt to avoid the mistakes in programming. For the baseline methods, we make full use of the methods available in the third-party libraries of the Python and the source codes provided in previous studies. Future studies should explore state-of-the-art imbalanced learning methods. However, the implementations of such specialized methods are seldom available and our implemented versions may include differences and errors compared with the original ones. Hence, we choose to compare against some off-the-shelf imbalanced learning methods. For the effects of dividing data into training and test sets, we use stratified sampling and repeat the process 30 times to reduce the variability of the random division. This is a commonly used setting for SDP studies.

7.3. Construct validity

The threats to construct validity concern the bias of the evaluation indicators used. In this study, we choose widely used traditional indicators and effort-aware indicators to show the empirical evaluation of our LDFR framework for SDP. Other indicators, such as g-mean, Balance, and AUC are not used, but we have reported their detailed results with our LDFR framework in the online supplementary materials for future studies to compare.

7.4. Reliability validity

The threats to reliability validity concern the possibility of repeating of our proposed framework. To enable further replication

and comparison with results from future studies, we have made the used benchmark dataset and source code available in our online supplementary materials.

8. Related work

Software defect data have the natural class imbalance trait, which means that the data comprise only a small number of defective modules (i.e., minority class) and the non-defective modules (i.e., majority class) account for the majority. As the class imbalance issue can prevent defect prediction methods from achieving satisfactory performance, researchers have proposed various imbalanced learning methods to mitigate such negative effects. The existing methods can be divided into three main categories: sampling-based, ensemble learning-based, and cost-sensitive-based imbalanced learning methods.

8.1. Sampling-based methods for SDP

For sampling-based imbalanced learning methods, three schemes can be used to mitigate the imbalance issue. The first is to decrease the non-defective modules, the second is to increase the defective modules with redundant or synthetic ones, and the third is to simultaneously increase the defective modules and decrease the non-defective modules until an approximately identical proportion status is reached. Kamei et al. (2007) studied the effects of four sampling methods on the performance of four classifiers. They chose two industry legacy software systems as a benchmark dataset. Their results showed that these sampling methods can promote the performance of linear and logistic models, but not neural network and classification tree models. Bennin et al. (2016) investigated the effects of four sampling methods on the performance of effort-aware based SDP models. They used 10 classification models and performed experiments on 10 software projects. Their results showed that these sampling techniques could improve the performance of all prediction models when the percentage of fault-prone modules lies in 20%∼30%. In addition, the results indicated that the proportions of defective and non-defective modules in the test set have a significant impact on the performance of effort-aware based models. Bennin et al. (2017b) explored the effects of six sampling methods on the performance of five classification models. The experimental results on 10 projects showed that these sampling methods had statistical and practical effects in terms of false positive, Recall, and G-mean, but not in terms of AUC. To generate new minority class modules that could contribute to the diversity of the defect data distribution, Bennin et al. (2018) introduced a novel over-sampling method MAHAKIL based on the chromosomal theory of inheritance. This method used the Mahalanobis distance to partition the minority class modules into two groups as the parents and generate synthetic modules that inherit the parents' diverse peculiarity. Their experimental results manifested the superiority of MAHAKIL over some classical sampling methods. Bennin et al. (2017a) examined the effects of a configurable parameter (i.e, the percentage of defective modules) in the sampling methods on the performance of SDP. They studied seven sampling methods with five classifiers on 10 projects. Their results illustrated that the configurable parameter had a great impact on the performance of these classifiers in terms of the used indicators (except for AUC). Tantithamthavorn et al. (2018a) assessed the effects of four sampling techniques on the performance and interpretation of seven classifiers. They performed a large-scale empirical experiment on four benchmark datasets using 10 performance indicators. Their results indicated that these sampling methods increased the completeness of Recall, but had little effect on AUC.

Sampling-based methods are limited in their ability to change the data distribution of the original data. In addition, such methods are averse to understanding the interpretation of the SDP models as described in Tantithamthavorn et al. (2018a). Unlike sampling-based methods, our LDFR framework does not change the module number; it just embeds the initial features into a more discriminable space.

8.2. Ensemble-based methods for SDP

Ensemble based imbalanced learning methods combine multiple weak classifiers into a new one to enhance the overall performance on SDP Valentini and Masulli (2002). Wang and Yao (2013) investigated how class imbalance learning methods enhance defect prediction performance. They compared sampling-based, threshold moving-based, and ensemble learning-based methods and found that a variant of AdaBoost achieved the best overall performance. Sun et al. (2012) proposed a coding-based ensemble learning method by transforming the imbalanced binary class data into multiple-class data and then used a special coding strategy to build a prediction model. Laradji et al. Laradji et al. (2015) proposed a two-variant ensemble learning method with or without feature selection methods to alleviate the class imbalance of the defect data. Petrić et al. (2016) built an ensemble model by first using a weighted accuracy diversity technique to exploit the diversity of four classifiers and then combining these classifiers with a stacking ensemble technique. Experiments on eight projects illustrated that their method can achieve better performance.

Unlike ensemble-based methods that usually iteratively sample multiple module subsets to enhance the classifier performance, our LDFR framework aims to learn discriminative features to enhance the classifier performance.

8.3. Cost-sensitive-based methods for SDP

The cost-sensitive-based methods increase the penalties when the minority class modules are classified as majority class modules. It is thus conducive to amend the bias in which classifiers prefer to predict the modules as the majority class on imbalanced defect data. To the best of our knowledge, Khoshgoftaar et al. (1999, 2002) were the first to introduce cost-sensitive imbalance learning into SDP and to build a prediction model with a boost method. Zheng (2010) proposed three cost-sensitive boosting neural networks methods for SDP. Their experimental results indicated that the threshold-moving-based method achieved better performance, especially on object-oriented software projects. Liu et al. (2014) proposed a two-stage cost-sensitive learning method for SDP by considering the cost information during both the classification and feature selection stages. Siers and Islam (2015) proposed a combined of decision tree-based cost-sensitive method and a cost-sensitive voting method for SDP. Both methods aimed to minimize the classification cost. Their experiments with six projects of the NASA dataset showed clear improvements compared with six baseline methods. Yang et al. (2017) proposed a two-layer ensemble learning method for just-in-time defect prediction. In the inner layer, decision tree and bagging were combined to construct a random forest model. In the outer layer, a random under-sampling technique was used to train various random forest models, and a stacking technique was used to assemble these models.

In addition to assigning the weights to various misclassification cases like general cost-sensitive-based methods, our LDFR framework integrates the metric learning process into cost-sensitive learning.

9. Conclusions

To address the challenges of learning capable features and alleviating the class imbalance issue in SDP studies, we propose a novel defect prediction framework, LDFR, by using a hybrid loss function to train a DNN to learn top-level feature representation. The hybrid loss function consists of a triplet loss and a weighted cross-entropy loss. The former loss can preserve locality by shrinking the distances among modules with the same label and can learn diacritical feature representation by enlarging the distances between the modules with different labels. The latter loss assigns more penalties when real defective modules are predicted as non-defective ones, which is helpful in reducing the negative impact of class imbalance. We evaluate the performance of our LDFR framework on 27 project version data (each with three types of features) with both traditional and effort-aware indicators. The experimental results show that compared with 27 baseline methods, LDFR usually performs significantly better in terms of five indicators.

In the future, we will adapt our deep feature representation learning framework to cross-project defect prediction by combining transfer learning techniques. In addition, we plan to incorporate more information in our model in an attempt to understand the potential reasons for defective software modules.

Acknowledgement

This work was supported by National Key Research and Development Program of China (2018YFC1604000), the grands of the National Natural Science Foundation of China (Nos. 61572374, 61602258, U163620068), Natural Science Foundation of Heilongjiang Province (No. LH2019F008), the Open Fund of Key Laboratory of Network Assessment Technology from Chinese Academy of Sciences, Hong Kong GRC Project (Nos. PolyU 152223/17E, PolyU 152239/18E), the General Research Fund of the Research Grants Council of Hong Kong (No. 11208017), the research funds of City University of Hong Kong (Nos. 9678149, 7005028), and the Research Support Fund by Intel (No. 9220097).

Supplementary material

Supplementary material associated with this article can be found, in the online version, at 10.1016/j.jss.2019.110402.

References

Arisholm, E., Briand, L.C., Johannessen, E.B., 2010. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. J. Syst. Softw. (JSS) 83 (1), 2–17.

Arora, I., Saha, A., 2018. Software defect prediction: A comparison between artificial neural network and support vector machine. In: Advanced Computing and Communication Technologies. Springer, pp. 51–61.

Batista, G.E., Prati, R.C., Monard, M.C., 2004. A study of the behavior of several methods for balancing machine learning training data. ACM SIGKDD Explor. Newsl. 6 (1), 20–29.

Bennin, K.E., Keung, J., Monden, A., 2017. Impact of the distribution parameter of data sampling approaches on software defect prediction models. In: Proceedings of the 24th Asia-Pacific Software Engineering Conference (APSEC). IEEE, pp. 630–635.

Bennin, K.E., Keung, J., Monden, A., Kamei, Y., Ubayashi, N., 2016. Investigating the effects of balanced training and testing datasets on effort-aware fault prediction models. In: Proceedings of the 40th Annual Computer Software and Applications Conference (COMPSAC), 1. IEEE, pp. 154–163.

Bennin, K.E., Keung, J., Monden, A., Phannachitta, P., Mensah, S., 2017. The significant effects of data sampling approaches on software defect prioritization and classification. In: Proceedings of the 11th International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE Press, pp. 364–373.

Bennin, K.E., Keung, J., Phannachitta, P., Monden, A., Mensah, S., 2018. Mahakil: diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction. IEEE Trans. Softw. Eng. (TSE) 44 (6), 534–550.

Burt, R.S., 2009. Structural Holes: The Social Structure of Competition. Harvard university press.

Challagulla, V.U.B., Bastani, F.B., Yen, I.-L., Paul, R.A., 2008. Empirical assessment of machine learning based software defect prediction techniques. Int. J. Artif. Intell. Tool. 17 (02), 389–400.

Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P., 2002. Smote: synthetic minority over-sampling technique. J. Artif. Intell. Res. 16, 321–357.

Chen, L., Fang, B., Shang, Z., Tang, Y., 2015. Negative samples reduction in cross–company software defects prediction. Inform. Softw. Technol. (IST) 62, 67–77.

Cheng, D., Gong, Y., Zhou, S., Wang, J., Zheng, N., 2016. Person re-identification by multi-channel parts-based cnn with improved triplet loss function. In: Proceedings of the 29th Conference on Computer Vision and Pattern Recognition (CVPR), pp. 1335–1344.

Chidamber, S.R., Kemerer, C.F., 1991. Towards a Metrics Suite for Object Oriented Design, 26. ACM.

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Trans. Softw. Eng. (TSE) 20 (6), 476–493.

Collobert, R., Weston, J., 2008. A unified architecture for natural language processing: Deep neural networks with multitask learning. In: Proceedings of the 25th International Conference on Machine Learning (ICML). ACM, pp. 160–167.

Dam, H. K., Tran, T., Pham, T., Ng, S. W., Grundy, J., Ghose, A., 2017. Automatic feature learning for vulnerability prediction. https://arxiv.org/abs/1708.02368.

Dam, H.K., Tran, T., Pham, T.T.M., Ng, S.W., Grundy, J., Ghose, A., 2018. Automatic feature learning for predicting vulnerable software components. IEEE Trans. Softw. Eng.

De Boer, P.-T., Kroese, D.P., Mannor, S., Rubinstein, R.Y., 2005. A tutorial on the cross-entropy method. Ann. Operat. Res. 134 (1), 19–67.

Demšar, J., 2006. Statistical comparisons of classifiers over multiple data sets. J. Mach. Learn. Res. 7 (Jan), 1–30.

Dong, Y., Li, J., 2017. Video retrieval based on deep convolutional neural network. https://arxiv.org/abs/1712.00133.

DAmbros, M., Lanza, M., Robbes, R., 2012. Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empir. Softw. Eng. (ESE) 17 (4–5), 531–577.

Fenton, N.E., Ohlsson, N., 2000. Quantitative analysis of faults and failures in a complex software system. IEEE Trans. Softw. Eng. (TSE) 26 (8), 797–814.

Fu, W., Menzies, T., 2017. Revisiting unsupervised learning for defect prediction. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE). ACM, pp. 72–83.

Ghotra, B., McIntosh, S., Hassan, A.E., 2015. Revisiting the impact of classification techniques on the performance of defect prediction models. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1. IEEE Press, pp. 789–800.

Ghotra, B., McIntosh, S., Hassan, A.E., 2017. A large-scale study of the impact of feature selection techniques on defect classification models. In: Proceedings of the 14th International Conference on Mining Software Repositories (MSR). IEEE, pp. 146–157.

Glorot, X., Bengio, Y., 2010. Understanding the difficulty of training deep feedforward neural networks. In: Proceedings of the 13th International Conference on Artificial Intelligence and Statistics, pp. 249–256.

Graves, T.L., Karr, A.F., Marron, J.S., Siy, H., 2000. Predicting fault incidence using software change history. IEEE Trans. Softw. Eng. (TSE) 26 (7), 653–661.

He, H., Bai, Y., Garcia, E.A., Li, S., 2008. Adasyn: Adaptive synthetic sampling approach for imbalanced learning. In: Proceedings of the 2008 International Joint Conference on Neural Networks (IJCNN). IEEE, pp. 1322–1328.

Hryszko, J., Madeyski, L., Dabrowska, M., Konopka, P., 2017. Defect prediction with bad smells in code. https://arxiv.org/abs/1703.06300.

Hu, H., Li, J., Wang, H., Daggard, G., Shi, M., 2006. A maximally diversified multiple decision tree algorithm for microarray data classification. In: Proceedings of the 2006 workshop on Intelligent Systems for Bioinformatics-Volume 73. Australian Computer Society, Inc., pp. 35–38.

Huang, C., Li, Y., Change Loy, C., Tang, X., 2016. Learning deep representation for imbalanced classification. In: Proceedings of the 29th Conference on Computer Vision and Pattern Recognition (CVPR), pp. 5375–5384.

Huang, Q., Xia, X., Lo, D., 2017. Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction. In: Proceedings of the International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 159–170.

Islam, Z., Giggins, H., 2011. Knowledge discovery through sysfor: a systematically developed forest of multiple decision trees. In: Proceedings of the 9th Australasian Data Mining Conference-Volume 121. Australian Computer Society, Inc., pp. 195–204.

Jiang, T., Tan, L., Kim, S., 2013. Personalized defect prediction. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE Press, pp. 279–289.

Jiang, Y., Cukic, B., Ma, Y., 2008. Techniques for evaluating fault prediction models. Empir. Softw. Eng. (ESE) 13 (5), 561–595.

Jing, X., Wu, F., Dong, X., Qi, F., Xu, B., 2015. Heterogeneous cross-company defect prediction by unified metric representation and cca-based transfer learning. In: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (FSE). ACM, pp. 496–507.

Jing, X.-Y., Wu, F., Dong, X., Xu, B., 2017. An improved sda based defect prediction framework for both within-project and cross-project class-imbalance problems. IEEE Trans. Softw. Eng. (TSE) 43 (4), 321–339.

Jing, X.-Y., Ying, S., Zhang, Z.-W., Wu, S.-S., Liu, J., 2014. Dictionary learning based software defect prediction. In: Proceedings of the 36th International Conference on Software Engineering (ICSE). ACM, pp. 414–423.

Jureczko, M., Spinellis, D., 2010. Using object-oriented design metrics to predict

software defects. Model. Method. Syst. Dependability. Oficyna Wydawnicza Politechniki Wrocławskiej 69–81.
Kamei, Y., Monden, A., Matsumoto, S., Kakimoto, T., Matsumoto, K.-i., 2007. The effects of over and under sampling on fault-prone module detection. In: Proceedings of the 1st International Symposium onEmpirical Software Engineering and Measurement (ESEM). IEEE, pp. 196–204.
Kamei, Y., Shihab, E., Adams, B., Hassan, A.E., Mockus, A., Sinha, A., Ubayashi, N., 2013. A large-scale empirical study of just-in-time quality assurance. IEEE Trans. Softw. Eng. (TSE) 39 (6), 757–773.
Khoshgoftaar, T.M., Allen, E.B., Jones, W.D., Hudepohl, J.P., 1999. Data mining for predictors of software quality. Int. J. Softw. Eng. Knowl. Eng. (IJSEKE) 9 (05), 547–563.
Khoshgoftaar, T.M., Geleyn, E., Nguyen, L., Bullard, L., 2002. Cost-sensitive boosting in software quality modeling. In: Proceedings of the 7th International Symposium on High Assurance Systems Engineering (HASE). IEEE, pp. 51–60.
Laradji, I.H., Alshayeb, M., Ghouti, L., 2015. Software defect prediction using ensemble learning on selected features. Inform. Softw. Technol. (IST) 58, 388–402.
Lessmann, S., Baesens, B., Mues, C., Pietsch, S., 2008. Benchmarking classification models for software defect prediction: a proposed framework and novel findings. IEEE Trans. Softw. Eng. (TSE) 34 (4), 485–496.
Li, J., He, P., Zhu, J., Lyu, M.R., 2017. Software defect prediction via convolutional neural network. In: Proceedings of the 2017 International Conference on Software Quality, Reliability and Security (QRS). IEEE, pp. 318–328.
Li, J., Liu, H., 2003. Ensembles of cascading trees. IEEE, p. 585.
Li, Z., Jing, X.-Y., Wu, F., Zhu, X., Xu, B., Ying, S., 2018. Cost-sensitive transfer kernel canonical correlation analysis for heterogeneous defect prediction. Autom. Softw. Eng. (ASE) 25 (2), 201–245.
Li, Z., Jing, X.-Y., Zhu, X., Zhang, H., Xu, B., Ying, S., 2017. On the multiple sources and privacy preservation issues for heterogeneous defect prediction. IEEE Trans. Softw. Eng. (TSE).
Liao, W., Yang, M.Y., Zhan, N., Rosenhahn, B., 2017. Triplet-based deep similarity learning for person re-identification. In: Proceedings of the 30th Conference on Computer Vision and Pattern Recognition (CVPR), pp. 385–393.
Liparas, D., Angelis, L., Feldt, R., 2012. Applying the mahalanobis-taguchi strategy for software defect diagnosis. Autom. Softw. Eng. (ASE) 19 (2), 141–165.
Liu, M., Miao, L., Zhang, D., 2014. Two-stage cost-sensitive learning for software defect prediction. IEEE Trans. Reliab. 63 (2), 676–686.
Manjula, C., Florence, L., 2018. Deep neural network based hybrid approach for software defect prediction using software metrics. Clust.Comput. 1–17.
Mende, T., Koschke, R., 2009. Revisiting the evaluation of defect prediction models. In: Proceedings of the 5th International Conference on Predictor Models in Software Engineering. ACM, p. 7.
Mende, T., Koschke, R., 2010. Effort-aware defect prediction models. In: Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR). IEEE, pp. 107–116.
Ming, Z., Chazalon, J., Luqman, M.M., Visani, M., Burie, J.-C., 2017. Simple triplet loss based on intra/inter-class metric learning for face verification. In: Proceedings of the 30th Conference on Computer Vision and Pattern Recognition (CVPR), pp. 1656–1664.
Moser, R., Pedrycz, W., Succi, G., 2008. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: Proceedings of the 30th International Conference on Software Engineering (ICSE). ACM, pp. 181–190.
Nam, J., Kim, S., 2015. Clami: Defect prediction on unlabeled datasets (t). In: Proceedings of the 30th International Conference on Automated Software Engineering (ASE). IEEE, pp. 452–463.
Nam, J., Pan, S.J., Kim, S., 2013. Transfer defect learning. In: Proceedings of the 35th International Conference onSoftware Engineering (ICSE). IEEE, pp. 382–391.
Okutan, A., Yıldız, O.T., 2014. Software defect prediction using bayesian networks. Empir. Softw. Eng. (ESE) 19 (1), 154–181.
Parchami, M., Bashbaghi, S., Granger, E., 2017. Video-based face recognition using ensemble of haar-like deep convolutional neural networks. In: Proceedings of the 2017 International Joint Conference on Neural Networks (IJCNN). IEEE, pp. 4625–4632.
Parkhi, O.M., Vedaldi, A., Zisserman, A., et al., 2015. Deep face recognition.. In: BMVC, 1, p. 6.
Petrić, J., Bowes, D., Hall, T., Christianson, B., Baddoo, N., 2016. Building an ensemble for software defect prediction based on diversity selection. In: Proceedings of the 10th International Symposium on Empirical Software Engineering and Measurement (ESEM). ACM, p. 46.
Phan, A.V., Nguyen, M.L., Bui, L.T., 2018. Convolutional neural networks over control flow graphs for software defect prediction. arXiv preprint arXiv:1802.04986.
Rahman, F., Devanbu, P., 2013. How, and why, process metrics are better. In: Proceedings of the 35th International Conference on Software Engineering (ICSE). IEEE, pp. 432–441.
Rumelhart, D.E., Hinton, G.E., Williams, R.J., 1986. Learning representations by back-propagating errors. Nature 323 (6088), 533.
Ryu, D., Choi, O., Baik, J., 2016. Value-cognitive boosting with a support vector machine for cross-project defect prediction. Empir. Softw. Eng. (ESE) 21 (1), 43–71.
Salakhutdinov, R., 2015. Learning deep generative models. Annual Review of Statistics and Its Application 2, 361–385.
Schroff, F., Kalenichenko, D., Philbin, J., 2015. Facenet: A unified embedding for face recognition and clustering. In: Proceedings of the 28th conference on Computer Vision and Pattern Recognition (CVPR), pp. 815–823.
Shepperd, M., Bowes, D., Hall, T., 2014. Researcher bias: the use of machine learning in software defect prediction. IEEE Trans. Softw. Eng. (TSE) 40 (6), 603–616.
Siers, M.J., Islam, M.Z., 2015. Software defect prediction using a cost sensitive decision forest and voting, and a potential solution to the class imbalance problem. Information Systems 51, 62–71.
Song, Q., Guo, Y., Shepperd, M., 2018. A comprehensive investigation of the role of imbalanced learning for software defect prediction. IEEE Trans. Softw. Eng. (TSE).
Song, Q., Jia, Z., Shepperd, M., Ying, S., Liu, J., 2011. A general software defect-proneness prediction framework. IEEE Trans. Softw. Eng. (TSE) 37 (3), 356–370.
Spinellis, D., 2005. Tool writing: a forgotten art?(software tools). IEEE Software 22 (4), 9–11.
Su, C., Yan, Y., Chen, S., Wang, H., 2017. An efficient deep neural networks training framework for robust face recognition. In: Proceedings of the 2017 International Conference on Image Processing (ICIP). IEEE, pp. 3800–3804.
Sun, Z., Song, Q., Zhu, X., 2012. Using coding-based ensemble learning to improve software defect prediction. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 42 (6), 1806–1817.
Sze, V., Chen, Y.-H., Yang, T.-J., Emer, J.S., 2017. Efficient processing of deep neural networks: a tutorial and survey. Proceedings of the IEEE 105 (12), 2295–2329.
Tantithamthavorn, C., Hassan, A. E., Matsumoto, K., 2018a. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. arXiv:1801.10269.
Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K., 2017. An empirical comparison of model validation techniques for defect prediction models. IEEE Trans. Softw. Eng. (TSE) 43 (1), 1–18.
Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K., 2018. The impact of automated parameter optimization on defect prediction models. IEEE Trans. Softw. Eng..
Turhan, B., Menzies, T., Bener, A.B., Di Stefano, J., 2009. On the relative value of cross-company and within-company data for defect prediction. Empir. Softw. Eng. (ESE) 14 (5), 540–578.
Valentini, G., Masulli, F., 2002. Ensembles of learning machines. In: Italian Workshop on Neural Nets. Springer, pp. 3–20.
Vickers, A.J., Elkin, E.B., 2006. Decision curve analysis: a novel method for evaluating prediction models. Med. Decis. Mak, 26 (6), 565–574.
Wang, S., Liu, T., Tan, L., 2016. Automatically learning semantic features for defect prediction. In: Proceedings of the 38th International Conference on Software Engineering (ICSE). ACM, pp. 297–308.
Wang, S., Yao, X., 2013. Using class imbalance learning for software defect prediction. IEEE Trans. Reliab. 62 (2), 434–443.
Wang, T., Zhang, Z., Jing, X., Zhang, L., 2016. Multiple kernel ensemble learning for software defect prediction. Automated Software Engineering (ASE) 23 (4), 569–590.
Xia, X., Lo, D., Pan, S.J., Nagappan, N., Wang, X., 2016. Hydra: massively compositional model for cross-project defect prediction. IEEE Trans. Softw. Eng. (TSE) 42 (10), 977–998.
Xu, Z., Li, S., Tang, Y., Luo, X., Zhang, T., Liu, J., Xu, J., 2018. Cross version defect prediction with representative data via sparse subset selection. In: Proceedings of the 26th Conference on Program Comprehension (ICPC). ACM, pp. 132–143.
Xu, Z., Liu, J., Yang, Z., An, G., Jia, X., 2016. The impact of feature selection on defect prediction performance: An empirical comparison. In: Proceedings of the 27th International Symposium on Software Reliability Engineering (ISSRE). IEEE, pp. 309–320.
Yang, B., Yin, Q., Xu, S., Guo, P., 2008. Software quality prediction using affinity propagation algorithm. In: Proceedings of the 2008 International Joint Conference on Neural Networks (IJCNN). IEEE, pp. 1891–1896.
Yang, X., Lo, D., Xia, X., Sun, J., 2017. Tlel: a two-layer ensemble learning approach for just-in-time defect prediction. Inform. Softw. Technol. (IST) 87, 206–220.
Yang, X., Lo, D., Xia, X., Zhang, Y., Sun, J., 2015. Deep learning for just-in-time defect prediction. In: Proceedings of the International Conference on Software Quality, Reliability and Security (QRS). IEEE, pp. 17–26.
Yang, Y., Zhou, Y., Liu, J., Zhao, Y., Lu, H., Xu, L., Xu, B., Leung, H., 2016. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE). ACM, pp. 157–168.
Zhang, C., Koishida, K., 2017. End-to-end text-independent speaker verification with triplet loss on short utterances.. In: Interspeech, pp. 1487–1491.
Zhang, F., Xin, S., Feng, J., 2017. Combining global and minutia deep features for partial high-resolution fingerprint matching. Patt. Recognit. Lett..
Zhang, F., Zheng, Q., Zou, Y., Hassan, A.E., 2016. Cross-project defect prediction using a connectivity-based unsupervised classifier. In: Proceedings of the 38th International Conference on Software Engineering (ICSE). ACM, pp. 309–320.
Zheng, J., 2010. Cost-sensitive boosting neural networks for software defect prediction. Expert Syst. Appl. 37 (6), 4537–4543.
Zhong, S., Khoshgoftaar, T.M., Seliya, N., 2004. Unsupervised learning for expert-based software quality estimation.. In: HASE. Citeseer, pp. 149–155.
Zimmermann, T., Nagappan, N., Gall, H., Giger, E., Murphy, B., 2009. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (FSE). ACM, pp. 91–100.

Zhou Xu is a joint Ph.D. student in the School of Computer Science at Wuhan University, China and in the Department of Computing at The Hong Kong Polytechnic University, Hong Kong. His research interests include software defect prediction, feature engineering, data mining.

Shuai Li is a Ph.D. student in the Department of Computing at The Hong Kong Polytechnic University. He received his BS degree in Computer Science from Dalian University of Technology in 2017. His research interests are in classification and detection both in theory and applications

Jun Xu received the B.Sc. degree in Pure Mathematics and the M. Sc. Degree in Information and Probability both from the School of Mathematics Science, Nankai University, China, in 2011 and 2014, respectively. He got his Ph.D. in Computer Science from the Department of Computing, The Hong Kong Polytechnic University. He is currently a lecturer at College of Computer Science, Nankai University. His research interests include image restoration, subspace clustering, sparse and low rank models.

Jin Liu received the Ph.D. degree in computer science from the State Key Lab of Software Engineering, Wuhan University, China, in 2005. He is currently a professor at School of Computer Science, Wuhan University. He has authored or coauthored a number of research papers in international journals. His research interests include mining software repositories and intelligent information processing.

Xiapu Luo received the Ph.D. degree in computer science from The Hong Kong Polytechnic University. He was a post-doctoral research fellow in the Georgia Institute of Technology. He is currently an associate professor in the Department of Computing and an associate researcher with the Shenzhen Research Institute, The Hong Kong Polytechnic University. His current research focuses on smartphone security and privacy, network security and privacy, and Internet measurement.

Yifeng Zhang is a master student in the School of Computer Science at Wuhan University, China. His research interests include software engineering and data mining.

Tao Zhang is an associate professor in Faculty of Information and Technology at Macau University of Science and Technology. He spent one year as a postdoctoral research fellow in the Department of Computing at The Hong Kong Polytechnic University. He got his Ph.D. in Computer Science from the University of Seoul, South Korea while he received his master degree and bachelor degree at Northeastern University, China. His research interest includes mining software repositories and empirical software engineering.

Jacky Keung received the BSc (Hons) degree in computer science from the University of Sydney, and the Ph.D. degree in software engineering from the University of New South Wales, Australia. He is an associate professor in the Department of Computer Science, City University of Hong Kong. His main research interests include software effort and cost estimation, empirical modeling and evaluation of complex systems, and intensive data mining for software engineering datasets. He has published papers in prestigious journals including the IEEE Transactions on Software Engineering, the Empirical Software Engineering, and many other leading journals and conferences. He is a member of the IEEE.

Yutian Tang received his Ph.D. degree from The Hong Kong Polytechnic University in 2018 and his BSc from Jilin University in 2013. Currently, he is a research staff in The Hong Kong Polytechnic University. His research interests include software product line, system security and machine learning.