Information and Software Technology 96 (2018) 94–111

Contents lists available at ScienceDirect

Information and Software Technology

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

Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning

Haonan Tong[a,b], Bin Liu[a,b], Shihai Wang[a,b,*]

[a] School of Reliability and Systems Engineering, Beihang University, Beijing 100083, China
[b] Science & Technology on Reliability & Environmental Engineering Laboratory, China

ARTICLE INFO

Keywords:
Software defect prediction
Stacked denoising autoencoders
Ensemble learning
Software metrics
Deep learning

ABSTRACT

Context: Software defect prediction (SDP) plays an important role in allocating testing resources reasonably, reducing testing costs, and ensuring software quality. However, software metrics used for SDP are almost entirely traditional features compared with deep representations (DPs) from deep learning. Although stacked denoising autoencoders (SDAEs) are powerful for feature learning and have been successfully applied in other fields, to the best of our knowledge, it has not been investigated in the field of SDP. Meanwhile, class-imbalance is still a pressing problem needing to be addressed.

Objective: In this paper, we propose a novel SDP approach, *SDAEsTSE*, which takes advantages of SDAEs and ensemble learning, namely the proposed two-stage ensemble (TSE).

Method: Our method mainly includes two phases: the deep learning phase and two-stage ensemble (TSE) phase. We first use SDAEs to extract the DPs from the traditional software metrics, and then a novel ensemble learning approach, TSE, is proposed to address the class-imbalance problem.

Results: Experiments are performed on 12 NASA datasets to demonstrate the effectiveness of DPs, the proposed TSE, and *SDAEsTSE*, respectively. The performance is evaluated in terms of F-measure, the area under the curve (AUC), and Matthews correlation coefficient (MCC). Generally, DPs, TSE, and *SDAEsTSE* contribute to significantly higher performance compared with corresponding traditional metrics, classic ensemble methods, and benchmark SDP models.

Conclusions: It can be concluded that (1) deep representations are promising for SDP compared with traditional software metrics, (2) TSE is more effective for addressing the class-imbalance problem in SDP compared with classic ensemble learning methods, and (3) the proposed *SDAEsTSE* is significantly effective for SDP.

1. Introduction

Software testing, aiming to detect as many defects as possible before the software is released, plays an important role in ensuring software quality. However, with the growth of software scale and complexity, testing cost and duration of traditional software testing are increasing dramatically. How to improve testing efficiency with limited testing resources to assure software quality is a great challenge to practitioners and researchers. Software defect prediction (SDP) technique was proposed to help to allocate testing resources reasonably, determine the testing priority of different software modules, and improve software quality. By using the results of SDP, software practitioners can efficiently judge that which software modules are more likely to be defective, the possible number of defects in a module, or other information related to software defects before software testing. Great achievements have been made in the field of SDP. Existing SDP studies can be divided into four categories: (1) Classification, (2) Regression, (3) Mining association rules, (4) Ranking.

The first category study aim to classify software entities (classes, functions, modules, files, etc.) into defect proneness and non-defect proneness or various levels of defect severity by using statistical techniques (such as discriminant analysis [1], and logistic regression [2]) and machine learning methods (such as support vector machines [3], and artificial neural networks [4]). The second category study aims to estimate the number of defects in the software entities by utilizing various methods, such as genetic programming [5], and support vector regression [6]. The third category study uses association rule mining approaches, such as relational association rule [7], and CBA2 algorithm [8], to mine the relationship between the defectiveness of software entities and software metrics. The fourth category study attempts to rank the software entities according to the number of defects in entities or directly optimizing the ranking performance, i.e., faults percentage

* Corresponding author.
E-mail addresses: tonghaonan@buaa.edu.cn (H. Tong), liubin@buaa.edu.cn (B. Liu), wangshihai@buaa.edu.cn (S. Wang).

https://doi.org/10.1016/j.infsof.2017.11.008
Received 7 January 2017; Received in revised form 13 October 2017; Accepted 13 November 2017
Available online 13 December 2017

average (FPA) [9] according to existing SDP studies [10,11].

Unfortunately, SDP is facing two major challenges [12,13]: class imbalance and high dimensionality. Class-imbalance data means that the number of non-defective modules (majority class) is much more than that of defective modules (minority class). Bohem and Basili [14] pointed out that in most cases, 20% of the modules can result in 80% of the software defects. In this case, SDP models may perform significantly worse on the minority class (i.e. the defective modules). Highly imbalanced datasets are well-known to reduce the ability of a machine learning algorithm to predict the infrequent class [15]. The imbalanced distribution is a major factor accounting for the poor performance of certain machine learning methods, especially for the minority class [16]. There are two popular kinds of methods to address the class-imbalance problem [16,17]: data-level (also known as sampling methods), and algorithm-level. Sampling methods have been widely investigated in previous SDP studies [2,18,19] by reducing the samples from the majority class (i.e. under-sampling) or adding samples to the minority class (i.e. over-sampling) to balance class distribution. Algorithm-level methods mainly refer to cost-sensitive learning (CSL) which considers the cost caused by a misclassified minority-class sample is greater than that of a misclassified majority-class sample and aims at minimizing the expected cost of misclassification instead of the traditional total misclassification error. CSL has drawn increasing attention in the field of SDP [17,20-22]. According to existing SDP studies, in addition to these two kinds methods, ensemble approaches (e.g., dynamic AdaBoost.NC [16], average probability ensemble [23]) also were used for addressing the class-imbalance problem.

Another challenge, i.e., high dimensionality, is caused by software metrics. SDP models were built on various software metrics (also called features), such as static code metrics [24-26], process metrics [27], etc. Software metrics extracted from software entities are often redundant and correlational, and this may have a negative impact on the performance of SDP models. As shown in previous SDP studies [13,28-31], feature selection and feature extraction methods do help to deal with this problem. However, almost all of these metrics are traditional metrics compared with deep features extracted by deep learning.

Deep learning has drawn more and more attention, because of its powerful feature learning capability, and has been successfully used in many domains, such as speech recognition [32], image classification [33], etc. Stacked denoising autoencoders (SDAEs) model is one of the most famous deep learning architectures. However, to the best of knowledge, SDAEs method has not been investigated in the field of SDP.

To bridge these gaps, we propose a novel SDP model based on deep learning and ensemble learning. The main contributions of this paper are three-fold:

(1) We use the stacked denoising autoencoders (SDAEs), a famous deep learning architecture, to extract deep representations from the traditional software metrics. To the best of our knowledge, it is the first time that SDAEs are used in the field of software defect prediction.
(2) To solve class imbalance and overfitting problems, we propose a novel ensemble learning approach, called two-stage ensemble (TSE).
(3) We evaluate and compare the proposed approach with existing state-of-the-art SDP models on 12 public software defect datasets from NASA projects. The experiment results show that the deep representations, the proposed TSE method, and proposed SDAEsTSE approach are effective.

The remainder of this paper is organized as follows. We review related work in Section 2. We elaborate our method in detail in Section 3. We present the experiments and results in Section 4. Discussions about our research are presented in Section 5. Finally, this study is concluded in Section 6.

2. Related work

2.1. Software defect prediction

Software defect prediction (SDP) refers to the techniques that use the historical defect data to build the relationship between software metrics and software defects.

Many software metrics have been proposed and used for SDP, including method-level metrics (e.g. the most widely used McCabe [24] and Halstead [25] metrics), Chidamber and Kemerer's (CK) metrics [26] specially designed for object-oriented software, network metrics [34], cascading style sheets metrics [35], web metrics [36], change metrics [37], developer micro interaction metrics [38], mutation-based metrics [39], package modularization metrics [40,41], context-based package cohesion metrics [42], lines of comments [43], and code smell metrics [44].

The most popular and widely used modeling methods for SDP are machine learning methods [45], such as artificial neural networks [46,47], Bayesian networks [48], support vector machines [3], dictionary learning [49], association rule [7], naive Bayes [50], tree-based methods [51], evolutionary algorithm [5], ensemble learning [20,22]. Some researchers thought software defect-proneness has the fuzzy characteristic, therefore they proposed fuzzy inference based SDP models [52,53]. Sometimes labeled samples for training SDP models are very limited even no labeled samples, some researchers proposed semi-supervised methods [19,54,55], unsupervised methods [56,57] for SDP.

2.2. Deep learning

Deep learning mainly has three architectures: convolutional neural networks (CNNs), deep belief networks (DBNs), and staked denoising autoencoders (SDAEs). Convolutional neural networks mainly are used in the field image processing [33,58].

Deep belief networks have been successfully applied and shown significant advantages in many fields, e.g., speech recognition [32]. Furthermore, some researchers [31,59] applied deep belief networks in the field of SDP. Yang et al. [31] used deep belief networks for just-in-time software defect prediction and the experimental results show that the model performed better than the model without deep representation. Wang et al. [59] utilized deep belief networks to learn semantic features automatically and proved that the deep features based method outperforms traditional software metrics.

For data partial corrupted with noise, Vincent et al. [60] stated that SDAEs outperforms deep belief networks because SDAEs help to extracted more robust features. Given this, SDAEs have attracted growing attention and been applied in many fields, such as wind speed prediction [61] and face recognition [62].

2.3. Ensemble learning

Ensemble methods aim to improve predictive performance by combining multiple weak learning learners into a strong learner [63]. There are mainly three famous ensemble methods: bagging [64], random forest [65], and AdaBoost [66]. Breiman [64] proposed bagging (i.e. bootstrap aggregating) method by combining multiple copies of a weak learner trained with bootstrap samples in 1996. Random forest proposed by Breiman [65] is specially developed for decision trees. AdaBoost (short for adaptive boosting) uses the weighted majority voting rule to combine base learners. Freund and Schapire [67] experimentally demonstrated that AdaBoost performs better than bagging.

Ensemble methods have been used and shown effectiveness in SDP domain [16,20,22]. Wang and Yao [16] proposed a dynamic AdaBoost.NC algorithm to address class imbalance problem in SDP. Siers and Islam [20] used the cost-sensitive random forest for SDP. Zheng proposed three SDP models (CSBNN-TM, CSBNN-WU1, and CSBNN-

WU2) which use cost-sensitive AdaBoost to boost neural networks.

3. Methodology

3.1. Variables

Dependent variable: the dependent variable used in this paper is a binary variable $Y \in \{0, 1\}$ where 1 denotes the class label of a software entity (module/class/file, etc.) is defective, 0 indicates the class label of a software entity is non-defective. During software test in the real world, if one or more defects are detected in a software entity, the dependent variable of this software entity is set to 1, otherwise 0. When perform software defect prediction by using a classifier trained on historical labeled defect data and a modeling technology, for each unlabeled software entity, the classifier will predict the probabilities that the dependent variable may be 0 and 1, respectively. If the former is larger than the latter, then the class label is predicted to be non-defective, otherwise defective.

Independent variables: In the field of software defect prediction, software metrics (aka. features or attributes in machine learning) are used as independent variables. A software metric (aka feature) is a quantitative measure of the degree to which a software system, component or process possesses a given property [68]. Catal and Diri divided software metrics into six categories [69]: method-level, class-level, component-level, file-level, process-level, and quantitative-level. The most commonly used metrics are method-level metrics [69] where 21 most commonly used method-level metrics are shown in Table 1.

However, in this paper, we do not directly use aforementioned traditional software metrics (TSM) as independent variables but use deep representations extracted from traditional metrics by using deep networks as the independent variables.

3.2. Motivation and overall architecture

3.2.1. Motivation

According to literature, existing SDP studies have the following problems:

(1) Software metrics used for SDP are almost entirely traditional features compared with deep representations extracted from traditional software metrics by using deep networks.
(2) Data noise problems in software defect datasets.

Table 1
Method-level software metrics. Taken from [17].

Categories	Metrics	Definition
McCabe	loc	Total lines of code
	v(g)	Cyclomatic complexity
	ev(g)	Essential complexity
	Iv(g)	Design complexity
Basic halstead	LOCode	Count of statement lines
	LOComment	Count of comment lines
	LOBlank	Count of blank lines
	LOCodeAndComment	Count of code and comment lines
	uniqOp	Number of unique operators
	uniqOpnd	Number of unique operands
	totalOp	Number of total operators
	totalOpnd	Number of total operands
	branchCount	Total number of branch count
	n	Total number of operators and operands
Derived halstead	v	Volume
	L	Program length = (v/n)
	d	Difficulty = (1/L)
	i	Intelligence
	e	Effort to write program
	b	Effort estimate
	t	Time estimator = E/18s

(3) Class Imbalance problem in software defect datasets.

It has been demonstrated that deep representations have an advantage over traditional features to reveal the essence of research object [32,33]. Although DBNs have been successfully used for SDP by some researchers [31,59], SDAEs have not been investigated in the field of SDP. Vincent et al. [60] stated that SDAEs outperforms deep belief networks, especially for noisy data. Furthermore, some researchers [70,71] have discussed the problem of data quality and have argued that it should be considered during SDP modeling. Ensemble learning is one type of the most widely used methods to solve class imbalance problem in SDP domain [16]. However, it has been demonstrated that ensemble learning methods have a risk of overfitting [72] and suffer from class imbalance [73,74], although they usually perform better than common weak classifiers. Therefore, it is necessary to extract more representative features with the capability of noise immunity and propose a more robust ensemble learning method in order to improve the performance of SDP models, especially in the case of class imbalance.

3.2.2. Overall architecture

The overall framework of the proposed model is shown in Fig. 1. The proposed method mainly includes two phases: deep learning (see Section 3.3.1) and ensemble learning (see Section 3.3.2). Deep learning phase aims to extract deep representations from the traditional software metrics. The second phase, namely two-stage ensemble (TSE), attempts to deal with class imbalance and overfitting problem.

Given a software defect dataset, it will be divided into two parts: training data and testing data. Training data is used to train the model. Testing data is applied to evaluate the predictive performance of the trained model. Before the usage of training and testing data, they will be preprocessed including deletion of duplicated instances, replacement of missing values, and normalization. After preprocessing, both training and testing data are fed into stacked de-noising auto-encoders to extract deep representations. Then the training data characterized by deep representations is used to train the proposed TSE. We next use testing data characterized by deep representations to evaluated trained TSE. The details of the proposed approach are presented in the following Section 3.3.

3.3. Proposed approach

In this section, we first present the preprocessing procedure and then describe the details of our proposed SDAEsTSE approach which mainly includes two phases: deep learning (DL) phase and two-stage ensemble (TSE) phase.

3.3.1. Data preprocessing

NASA MDP datasets are commonly used in the field of SDP. However, some researchers, such as Gray et al. [70], Shepperd et al. [75], and Petric et al. [71], have questioned the quality of these datasets. Inspired by their proposed solutions for dealing with the quality problems of data, some concerns about data preprocessing in this study can be summarized as follows:

(1) Deletion of Duplicated Instances
Duplicated instances refer to software modules which have same software metric values and same class labels (e.g., defective labels). This situation is possible in the real world. Unfortunately, repeated instances can potentially have a negative impact on machine learners. More clearly, duplicated instances can cause over-optimistic performances when they are classified correctly as a part of test data, can cause over-pessimistic performances when they are misclassified as a part of test data. Moreover, they not only can make the training process time-consuming but also cannot make any improvement in the model's performance. Therefore, duplicated

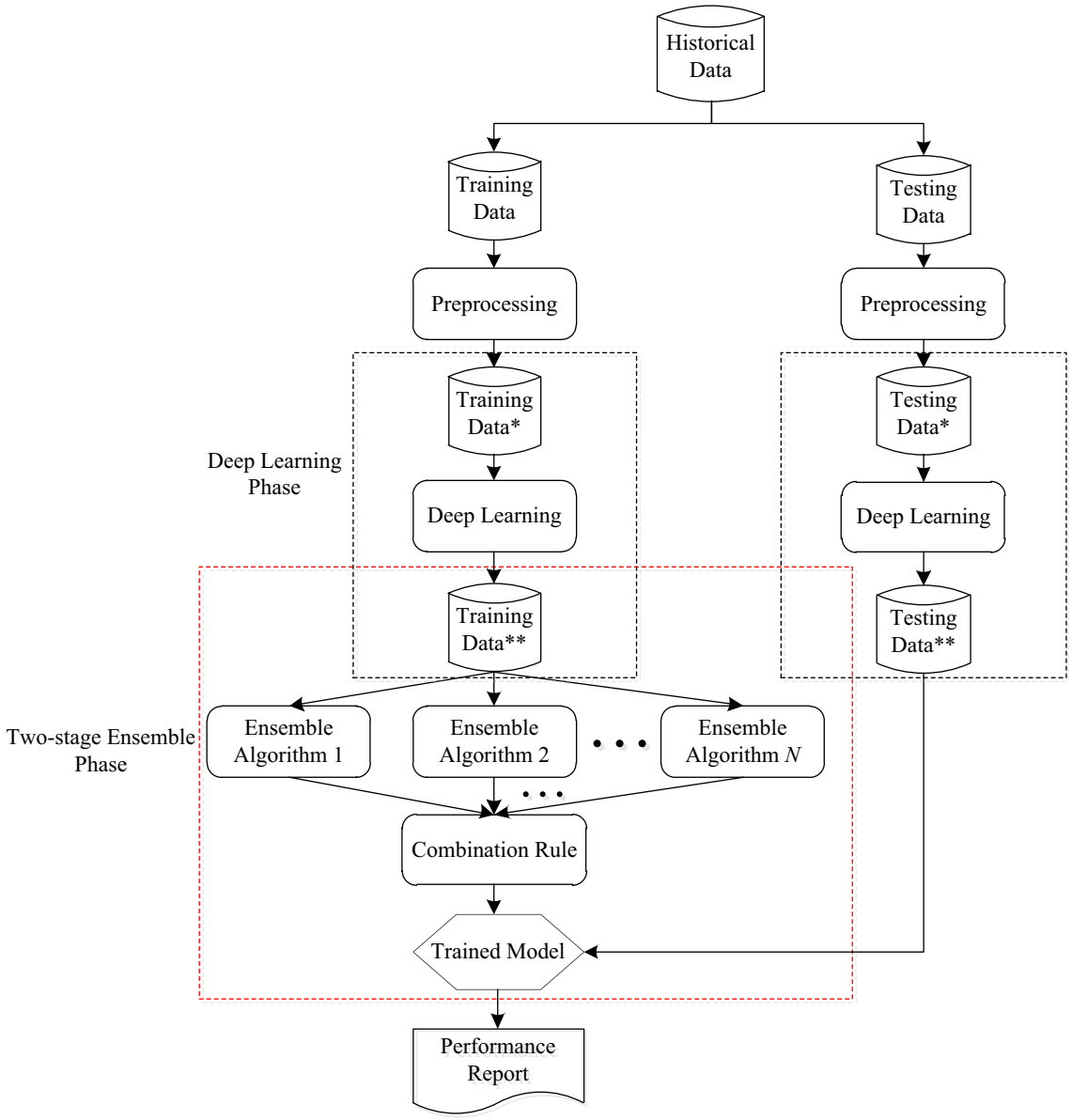

Fig. 1. The Framework of the Proposed ASDEsTSE for Software Defect Prediction.

instances, if any, should be removed.

(2) Replacement of Missing Values

Generally, an instance consists of the values of multiple software metrics. If one or more values in an instance are missing owing to some reasons (e.g., data collector's carelessness), then this instance cannot satisfy the input requirement of our approach. These missing values must be processed. In this study, for a missing value, we replace it with the mean of the corresponding metric. For example, given a metric *mt* and its observations $\{mt_1, \cdots mt_{100}\}$, and suppose that both mt_{99} and mt_{100} are missed (i.e., NaN). Then these two missing values are replaced by:

$$mt_{99} = mt_{100} = \frac{1}{98}\sum_{i=1}^{98} mt_i. \tag{1}$$

(3) Data Normalization

Because the values of different software metrics usually have the different order of magnitude, we conduct data normalization on these metrics. In this paper, we use the commonly used min-max normalization method [76] to normalize the data and transform all values to values in the interval [0, 1]. Given a metric x, its maximum and minimum values are $\max(x)$ and $\min(x)$, respectively. For each value x_i of metric x, the normalized value $\widetilde{x}_i$ is computed as

$$\widetilde{x}_i = \frac{x_i - \min(x)}{\max(x) - \min(x)}. \tag{2}$$

3.3.2. Deep learning phase

A. Stacked denoising autoencoders

Stacked denoising autoencoders (SDAEs) [77] model is a powerful deep learning model, which is made of multiple denoising autoencoders [60] stacked on top of each other. Denoising autoencoder (DAE) is a modified version based on basic autoencoder by adding a corruption process, and an autoencoder is a feedforward neural network with an input layer, an output layer, and a hidden layer. DAE attempts to reconstruct the input from the corrupted input. The framework of a DAE is shown in Fig. 2. DAE is trained as follows:

Given a d-D input vector $x \in R^d$, its corrupted version $\widetilde{x} \in R^d$ can be obtained by first randomly selecting vd elements in x and then setting

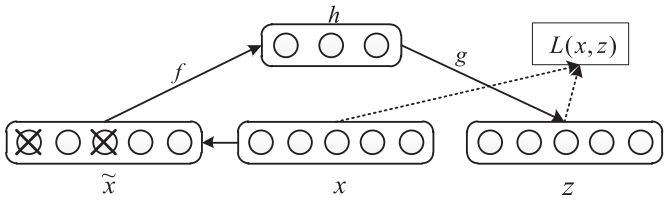

Fig. 2. An example of denoising autoencoder, the original input x is corrupted to $\tilde{x}$, $\tilde{x}$ is mapped to h and then the representation z tries to reconstruct x. The reconstruction error is denoted as $L(x, z)$. Taken from [60].

their value to zero while keeping the reminder unchanged, where v is called as corruption rate. A basic autoencoder first encoders $\tilde{x}$ to a hidden layer h, and then decodes the output of hidden layer to generate a reconstruction z. These two processes can be formularized as follows:

$$\text{Encoding:}\quad h(\tilde{x}) = f_1(W_1\tilde{x} + b_1),$$

$$\text{Decoding:}\quad z(\tilde{x}) = f_2(W_1^1 h(\tilde{x}) + b_2),$$

Where $W_1 \in R^{k_1 \times d}$ and $W_1^1 \in R^{d \times k_1}$ are weight matrices, $b_1 \in R^{k_1}$ and $b_2 \in R^d$ are the corresponding bias terms, $f_1(\cdot)$ and $f_1(\cdot)$ denote two nonlinear activation functions, and k_1 indicates the number of nodes in the hidden layer. In this paper, the sigmoid function is used as activation function by default.

Given a series of inputs $\{x_i\}_{i=1}^N$, to train the model, the idea is to minimize the average reconstruction error between the input x_i and the corresponding reconstruction $z(\tilde{x}_i)$, the parameters of the model are denoted by θ:

$$\theta = \arg\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} L(x_i, z(\tilde{x}_i)), \tag{3}$$

Where L is a function to measure the reconstruction error, we can let L be squared error, i.e., $L(x, z) = \|x - z\|^2$.

Finally, use the back-propagation (BP) algorithm with the stochastic gradient descent technique to update the parameters until the stopping criterion is satisfied, e.g., the reconstruction error is lower than the minimum limit.

Having trained the first DAE the training data is feed-forwarded through the first DAE and the outputs of the hidden layer of the first DAE are used as the training data to train the second DAE, and similarly for the third DAE, etc. In this way, multiple DAEs can be stacked on top of each other and trained in an unsupervised greedy layer-wise fashion [78,79].

B. Structural parameter selection for SDAEs

As discussed in Section A, to get an effective SDAEs model, we need to set three key structural parameters, which are: 1) the number of hidden layers, 2) the number of nodes in each hidden layer, and 3) the value of corruption rate.

To simplify the SDAEs model, we choose three hidden layers, which is a general configuration followed by Vincent et al. [77], and we make the first two hidden layers have the same number of nodes. For the number of nodes in the first two hidden layers, we experiment with four discrete values including 100, 200,500, and, 1000. For the number of nodes in the last hidden layer, we experiment with six discrete values including 5, 10, 20, 40, 100, and, 200. For the values of corruption rate, we experiment with six discrete values including 0, 0.2, 0.3, 0.6, 0.8, and, 1. We first change the number of nodes in the first two hidden layers and fix the other two, and then we change the number of nodes in the last hidden layer and fix the other two, and so on. We perform 5-fold cross validation (CV) [80] 50 times, and use logistic regression as the classifier, which is commonly used in SDP [31,41,42,81].

We found that the number of nodes in the first two hidden layers has no substantial significant effect on the performance of classifier compared with the other two parameters (i.e., the number of nodes in the last hidden layer and the corruption rate). For the sake of simplicity, we just set the number of nodes in the first two hidden layers to 100. For the number of nodes in the last hidden layer, when its value is 5 or 10, the performance of the classifier is obviously worse than the case that it takes other values. In fact, in most cases, when the number of nodes in the last hidden layer ranges from 20 to 100, the effect on the classifier has no significant difference, although there are some slight fluctuations. For the corruption rate, its effect on classifier depends on its value and the specific dataset.

To select the optimal corruption rate for each dataset, CV methods is used again. Concretely, for each dataset, we run 5-fold cross validation (CV) 50 times, for each run, we utilize four out of the five folds to build the predictive model, which will then be tested on the reminding fold. Before the training begins, the four folds are further divided into 5 folds, in which four folds are used as the training dataset, and the reminding fold is used as validation dataset. The optimal corruption rate that results in the best performance on the validation dataset is then obtained. Utilizing the optimal corruption rate, a SDAEs model is trained on the initial training dataset.

C. Generating deep representations

After a SADEs model is trained, we feed the preprocessed training data and test data into the SDAEs respectively and then obtain the corresponding deep representations for training data and test data from the last hidden layer of the SDAEs. Fig. 3 presents an example of using SDAEs to extract deep representations. Given a train SDAEs model, and an instance $(x_{i1}, ..., x_{ik}, ..., x_{id}, y_i)$ where x_{ik} denotes the value of the kth metric of the ith instance and y_i indicates the class label of the ith instance. We input $(x_{i1}, ..., x_{ik}, ..., x_{id})$ into the SDAEs network, feed-forward layer-by-layer, and then we obtain the deep-representation vector $(deepx_{i1}, ..., deepx_{in_4})$ which shares same class label y_i with the original instance. $(deepx_{i1}, ..., deepx_{in_4}, y_i)$ is a deep-representation based instance.

3.3.3. Two-stage ensemble phase

In two-stage ensemble (TSE) phase of SDAEsTSE, we attempt to use the deep representation based training data to construct a strong classifier by using ensemble learning framework and then apply it to deep representation based test data to evaluate the model. Ensemble learning (EL) aims to the technology that it aims to improve predictive performance by combining multiple weak learning learners into a strong

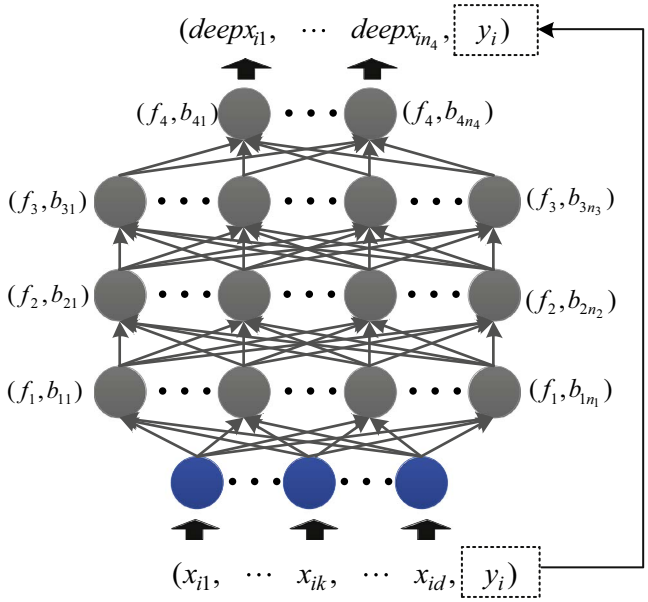

Fig. 3. A schematic of using stacked denoising autoencoders to extract deep representations. Solid circles in the bottom layer are input-layer nodes, solid circles in other layers correspond to hidden-layer nodes. The outputs of the top hidden layer are deep representations (DR). The class label of deep representations is same as that of the inputs.

learner [63]. There are three famous classic EL approaches including bagging [64], random forest [65], and AdaBoost [66].

However, our proposed EL approach, i.e., TSE, is different from existing classic EL methods. The differences are three folds: (1) **base learner**, on one hand, classic EL methods are homogenous, but TSE is heterogeneous because the former use only one kind of base learner and the latter uses three kinds of base learners. On the other hand, classic EL methods use weak classifier as the base learner to obtain strong classifier, but TSE utilizes strong classifier as base learner. (2) **training process**, the training process of TSE is different from any of classic ensemble learning methods. For TSE, the base learners are trained by using same training data, and they are trained independently. (3) **combining rule**, differ from classic EL's combining rules, such as majority vote and weighted majority vote, TSE uses weighted average probabilities and probability adjustment as combining rule. For TSE, the first stage ensemble learning refers to the training of three base learners, i.e., bagging, random forest, and AdaBoost. The second stage ensemble learning refers to the combination of three base learners by using our proposed combining rule.

Since base learner has an important effect on the performance of the obtained strong classifier by ensemble learning, we take strong classifier as base learner. Considering that none of the classic ensemble learning methods always performs the best on every dataset, and none of them always performs the worst on every dataset. Furthermore, it has been demonstrated that heterogeneous ensemble outperforms homogenous ensemble [82]. Therefore, we apply heterogeneous ensemble framework, namely take bagging, random forest, and AdaBoost as base learners. Classic ensemble learning methods suffer from class imbalance [73,74,83], although they have been proven to be superior to weak classifiers. Therefore, we propose a new combining rule for addressing the class imbalance. The details of our TSE are described as follows:

Given training data $D_{tr} = \{(x_i, y_i)\}_{i=1}^{n}$, defective rate π_d, and test data $D_{te} = \{(x_j)\}_{j=1}^{m}$, where x_i, $x_j \in R^d$ are two d-D vectors made of the values of d software metrics, $y_i \in \{0, 1\}$ is equal to 0 if ith software entity is defective (i.e., the minority class), otherwise 1 (i.e., the majority class), and π_d refers to the proportion of the number of defective instances to the total number of instances, which can be empirically estimated according to training data. We first preprocess the training data and test data by removing repeated instances in training data, deleting instances having missing values, and performing data normalization. We next use training data to perform 5-fold cross validation (CV) for each base learner. For the ith base learner, we use the average performance of MCC (see Section 4.3 for details) as the performance of each base learner and denote it by $avgMCC_i (i = 1, 2, 3)$. After CV is performed on every base learner, we compute the weight of each base learner as follows:

$$w_i = \frac{avgMCC_i}{\sum_{i=1}^{3} avgMCC_i},$$

Then training data is used to train every base learner, we denote each trained base learner by $h_i (i = 1, 2, 3)$.

For $x_j \in D_{te}$, the predicted probabilities of $y_j = 0$ and $y_j = 1$ by base classifier h_i are denoted by a tuple $[proND_{ij}, proD_{ij}]$. Then we compute the combined probabilities $[coproND_j, coproD_j]$ as follows:

$$[coproND_j, coproD_j] = \sum_{i=1}^{3} w_i * [proND_{ij}, proD_{ij}],$$

We indicate the maximum value among $\{proD_{ij}\}_{i=1}^{3}$ as $maxProD_{kj}$ where k is the index of the base classifier who predicts the maximum defective probability. For example, if $proD_{3j}$ is the maximum among $\{proD_{ij}\}_{i=1}^{3}$, then $k = 3$.

Next, we need to adjust the combined probabilities according to π_d. Now we define a threshold T, which is determined by π_d, and it can be calculated according to the following sigmoid function:

$$T(\pi_d) = \frac{\alpha}{1 + \exp\{-\beta * (\pi_d - \gamma)\}} + \theta, \quad (4)$$

where α, β, γ and θ are four positive parameters. The value of T ranges from θ to $(\alpha + \theta)$, β decides the slope of the sloped line segment in the sigmoid function $T(\pi_d)$, and γ represents the midpoint of the sloped line segment in the sigmoid function $T(\pi_d)$. The value of θ ranges from 0 to 0.5, and the smaller the value of θ is, the more attention the actual positive instances get. However, if the value of θ is too small, the positive instances will get too much attention, this will result in that more positive instances can be correctly classified but most of the negative instances will be misclassified. If the value of $(\alpha + \theta)$ ranging from 0 to 0.5 is too large, the positive instances cannot get enough attention, which will result in that the positive instances are hard to be correctly classified, especially when the class-imbalance degree of the defect data is serious. If the value of β is too small, the change of threshold T is too gentle which will reduce our method's sensitivity to the class-imbalance degree of defect data. Similarly, if the value of γ, which ranges from 0 to 0.5, is too large, it will result in that our method is too sensitive to the class-imbalance degree of defect data, and then the negative instances may be misclassified easily. In this paper, the four parameters of the sigmoid function $T(\pi_d)$ are set as follows: $\alpha = 0.2$, $\beta = 500$, $\gamma = 0.055$, and $\theta = 0.2$.

If $\pi_d < 0.2$, namely the training data is extremely imbalanced, in this case, if $T < maxProD_{kj} < 0.5$, we reverse the predicted probabilities of kth base classifier and then assign them to the combined probabilities, i.e., $[coproND_j, coproD_j] = [proD_{kj}, proND_{kj}]$; else if $maxProD_{kj} \geq 0.5$, we take the predicted probabilities of kth base classifier as the combined probabilities, namely, $[coproND_j, coproD_j] = [proND_{kj}, proD_{kj}]$. If $\pi_d \geq 0.2$ and $maxProD_{kj} \geq 0.5$, we take the predicted probabilities of kth base classifier as the combined probabilities, namely, $[coproND_j, coproD_j] = [proND_{kj}, proD_{kj}]$. Now the process of adjustment is finished.

Finally, we can predict the class label (i.e., $\tilde{y}_j$) of x_j according to the corresponding combined probabilities as follows: if $coproND_j > coproD_j$, then $\tilde{y}_j = 0$, namely jth instance is predicted to be non-defective, otherwise $\tilde{y}_j = 1$. Repeat above procedures until the class labels of all testing instances are obtained.

To make it easier to understand, let's take an example to present the shortcomings of classic combining rules (i.e., majority vote and weighted majority vote) and the superiority of our proposed combining rule in case of class imbalance. Given a test sample x_j with real class label $y_j = 1$, a class imbalanced dataset with defective rate $\pi_d = 0.12$, the weights of three trained base classifiers are respectively 0.3, 0.4, and 0.3, the predicted probabilities of three base classifiers for x_j are [0.9, 0.1], [0.92, 0.08], and [0.7, 0.3], respectively. Obviously, the classic combining rules will give same misclassified results, i.e., $\tilde{y}_j = 0$. However, our proposed combining rule can give correct predicted class label, i.e., $\tilde{y}_j = 1$. By performing a large number of experiments and observations, the case assumed in the above-mentioned example does often happen and is reasonable and real. It is the root cause that our combining rule outperforms classic combining rules.

Algorithm 1 presents the details of our proposed two-stage ensemble learning approach.

4. Experiments and results

In this section, we evaluate the effectiveness and efficiency of our proposed SDAEsTSE. The experimental environment is an Intel(R) Core (TM) i5-4200U CPU @ 1.60 GHz 2.30 GHz 4.0 GB RAM laptop running Windows 8.1 (64 bit) and MATLAB R2012a (Version 7.14). We first present the used benchmark datasets, and experiment setup, respectively. Then we describe the utilized performance evaluation criteria. Finally, we propose our research questions (RQ) and answer them.

Algorithm 1
Two-stage Ensemble Phase.

Input: Training data D_{tr}, testing data D_{te}, defective rate π_d, ensemble learning algorithm pool ELP = {Random Forest, Bagging, AdaBoost.}.
Output: The predicted class label of each instance in testing data.
Delete repeated instances in both D_{tr} and D_{te}.
Normalize D_{tr} and D_{te} by using min-max method.
for each ensemble algorithm in ELP **do**
    Train it on training data D_{tr}, then give back a hypothesis h_i.
    with respect to h_i, perform 5-fold cross validation on D_{tr}, and calculate the average ***MCC*** (see Section 4.3), $avgMCC_i$.
end for
Set the weight of h_i (i = 1,2,3) as $w_i = \frac{avgMCC_i}{\sum_{i=1}^{3} avgMCC_i}$.
for each instance $x_j \in D_{te}$ **do**
    Use h_i (i = 1,2,3) to predict the probabilities of x_j being non-defective and defective, and denote them as $[proND_{ij}, proD_{ij}]$ where $proND_{ij} + proD_{ij} = 1$.
    Set the combined average probabilities as $[coproND_j, coproD_j] = \sum_{i=1}^{3} w_i * [proND_{ij}, proD_{ij}]$.
    Find the maximum probability of being defective among $\{proD_{ij}\}_{i=1}^{3}$, and express it as $maxProD_{kj}$ where k is equal to i, e.g., if $ProD_{3j}$ is the maximum, then $k = 3$.
    Set threshold T as follows: $\frac{0.2}{1+exp\{-500\times(\pi_d-0.055)\}} + 0.2$
    if $\pi_d < 0.2$ **then**
        if $T < maxProD_{kj} < 0.5$ **then**
            $[coproND_j, coproD_j] = [proD_{kj}, proND_{kj}]$.
        end if
        if $maxProD_{kj} \geq 0.5$ **then**
            $[coproND_j, coproD_j] = [proND_{kj}, proD_{kj}]$.
        end if
    else
        if $maxProD_{kj} \geq 0.5$ **then**
            $[coproND_j, coproD_j] = [proND_{kj}, proD_{kj}]$.
        end if
    end if
    if $coproND_j \leq coproD_j$ **then**
        $\tilde{y}_j = 1$, i.e., j-th testing instance is predicted to be defective.
    else
        $\tilde{y}_j = 0$, i.e., j-th testing instance is predicted to be non-defective.
    end if
end for
Output: The predicted class label of each instance in testing data.

4.1. Datasets

We evaluate *SDAEsTSE* on 12 public NASA datasets, which are CM1, KC1, KC2, KC3, MC1, MC2, MW1, PC1, PC2, PC3, PC4, and JM1. Five datasets including CM1, KC1, KC2, PC1, and JM1 are obtained from the PROMISE software engineering repository [84], and the reminding datasets are obtained from tera-PROMISE Repository [85]. The statistics of these 12 datasets are described in Table 2. Note that all datasets are imbalanced.

We found that JM1 has the missing values and some datasets, such as CM1, KC1, KC2, PC1, PC4, and JM1, have repeated instances. Note that KC1 has the most number of repeated instances.

4.2. Experiment setup

Cross-validation (CV) [80] is a predominant approach to evaluate the generalization performance of a prediction model, which is widely used in software defect prediction domain [31,86]. In this study, we utilize 5-fold CV to evaluate the proposed approach. Each dataset is divided into 5 folds, in which 4 folds are taken as training data, and the remaining one fold is taken as testing data.

Since *SDAEsTSE* involves a degree of randomness (e.g., a dataset is randomly divided into 5 folds), we need to run *SDAEsTSE* multiple times. Arcuri and Briand [87] suggested that sample size should be not less than 30 to make a meaningful comparison. Same as previous SDP study [88], by default, we run our approach 50 times and report the average performance in the form of "mean ± standard deviation".

Table 2
Statistics of used software defect datasets. #M denotes the number of software metrics, #I Bef. the number of instances before data preprocessing, #I Aft. the number of instances after data preprocessing, # NDI the number of non-defective instances, and %DI the percentage of defective instances.

Data	#M	#I		# NDI		% DI	
		Bef.	Aft.	Bef.	Aft.	Bef.	Aft.
CM1	21	498	442	449	394	9.84	10.86
KC1	21	2109	1212	1783	897	15.46	25.99
KC2	21	522	375	415	270	20.50	28.00
KC3	39	194	194	158	158	18.56	18.56
MC1	38	1988	1988	1942	1942	2.31	2.31
MC2	39	125	125	81	81	35.20	35.20
MW1	37	253	253	226	226	10.67	10.67
PC1	21	1109	954	1032	884	6.94	7.34
PC2	36	745	745	729	729	2.15	2.15
PC3	37	1077	1077	943	943	12.44	12.44
PC4	37	1458	1344	1280	1167	12.21	13.17
JM1	21	10,885	8912	8779	6905	19.35	22.50

All experiments are performed in MATLAB environment, and all baselines are realized by invoking WEKA. Unless otherwise stated, we always use the classifiers with the default setting in WEKA. WEKA is an open-source machine learning and data mining software developed by the University of Waikato, which is publicly available and can be obtained from the website http://www.cs.waikato.ac.nz/ml/weka/index.html.

4.3. Performance evaluation criteria

To evaluate the performance of the proposed model, three performance measures including ***F-measure*** (aka. F1 or F1 score), area under ROC curve (i.e., ***AUC***), and Matthews correlation coefficient (i.e., ***MCC***) are used.

They can be defined by using confusion matrix as shown in Table 3. By convention, the defective modules are regarded as positive samples and non-defective modules as negative samples.

According to confusion matrix, the probability of detection (*PD*, also called *recall, TPR, sensitivity*), the probability of false alarm (*PF*, also called *FPR*), *precision* can be defined as.

$$PD = \frac{TP}{TP + FN} \tag{5}$$

$$PF = \frac{FP}{FP + TN} \tag{6}$$

$$precision = \frac{TP}{TP + FP} \tag{7}$$

To evaluate the classifier more comprehensively in the class-imbalance context, ***F-measure*** [1] and ***AUC*** [89] are commonly applied. ***F-measure*** considers both *PD* and *precision*. ***AUC***, varying in [0, 1], measures the area under the ROC curve which describes the trade-off between *PD* and *PF*. The larger ***AUC***, the better the performance of the classifier is. ***F-measure*** and ***AUC*** are can be calculated as follows:

Table 3
Confusion matrix.

		Actual	
		Positive	Negative
Predicted	Positive	*TP*	*FP*
	Negative	*FN*	*TN*

$$F - measure = \frac{2*PD*precision}{PD + precision} \tag{8}$$

$$AUC = \frac{\sum_{ins_i \in PositiveClass} rank(ins_i) - \frac{M(M+1)}{2}}{M*N} \tag{9}$$

Where $\sum_{ins_i \in PositiveClass} rank(ins_i)$ is the sum of ranks of all positive samples, M and N are the number of positive samples and negative samples, respectively. For ***F-measure***, if TP equals 0, ***F-measure*** is NaN.

Matthews Correlation Coefficient (***MCC***) [90] is another overall measure of the quality of binary classification, which takes all true and false positives and negatives into account. The range of its value is the closed interval $[-1, 1]$. The larger ***MCC*** is, the better the model's prediction performance is. It has been used as a well performance measure in existing SDP studies [91,92]. ***MCC*** can be calculated according to the following formula:

$$MCC = \frac{TP*TN - FP*FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{10}$$

4.4. Research questions and results

RQ1: How effective are deep representations?

Motivation: To validate the effectiveness of deep representations, we need to compare the predictive performance of SDP models developing by traditional metrics and deep representations, respectively.

Approach: We use logistic regression (LR) [93] as the modeling technology, which is a probabilistic statistical classification method. We decided to use LR owing to two main reasons. On one hand, LR is a commonly used method to develop software defect prediction models [31,41,42,81]. On the other hand, LR is not only a very simple classification method but also performs well compared with other complex modeling methods for SDP [94,95]. We attempt to evaluate the performance of the classifiers trained by using traditional metrics and deep representations, respectively.

To prove the significant superiority of deep representations compared with corresponding traditional metrics, we perform Wilcoxon Rank-Sum test [96] with 5% significance level and compute the p-value. Wilcoxon Rank-Sum test is an effective non-parametric test method to check whether the differences between two sample sets are statistically significant (which happens when the p-value is less than 0.05). To present the effect size, we also report Cliff's delta [97]. Cliff's delta is a non-parametric effect size measure that quantifies whether the differences are substantial or not, which has been used in previous SDP studies [31,88]. Cliff's delta can be calculated according to the method proposed by Macbeth et al. [98]. Delta's value ranges from -1 to 1, where -1 or 1 denotes the absence of overlap between two approaches, and $\delta = 0$ indicates that two approaches are completely overlapping. In this paper, we utilize Cliff's delta (δ) to compare our proposed approach with the baselines. If delta $>$ 0, it means that our approach is better than the baseline method. If delta $<$ 0, it indicates that the baseline method is better than our approach. Table 4 presents different Cliff's delta values and their corresponding meaning.

Besides logistic regression, we also use random forest and Bayesian network as the modeling technologies, respectively. The results are presented in Appendix A.

Results: Table 5 presents the average performance of the proposed SDAEsTSE and the baselines on each benchmark dataset. We notice that the F1 values of DR vary from 0.156 to 0.545 across all benchmark datasets, the AUC values of DR vary from 0.683 to 0.987, and the MCC values of DR vary from 0.13 to 0.39.

Table 4
Cliff's delta and effect size level [97].

Cliff's delta($\|\delta\|$)	Effect size level
$\|\delta\| < 0.1470$	Negligible
$0.1470 \le \|\delta\| < 0.3300$	Small
$0.3300 \le \|\delta\| < 0.4740$	Medium
$\|\delta\| \ge 0.4740$	Large

Table 6 presents the p-value and Cliff's delta when we compare SDAEsTSE with the baseline methods in terms of three performance measures, i.e., *F1,* ***AUC***, and ***MCC***. We notice that in most cases, DR is significantly better than TSM. When considering both Table 5 and Table 6, we find that:

In terms of *F1*, DR significantly outperforms TSM with a large effect size in four out of 12 datasets (MW1, PC1, PC2, and JM1), with a medium effect size in two datasets (CM1 and KC1), and with a small effect size in three datasets (KC2, MC1, and MC2). TSM is significantly better than DR with a large effect size in three datasets (KC3, PC3, and PC4).

In terms of ***AUC***, DR significantly outperforms TSM with a large effect size in five out of 12 benchmark datasets (KC3, MC1, MC2, MW1, and JM1), and with a small effect size in one dataset (KC1). DR has no significant difference with TSM in one dataset (KC2). TSM is significantly better than DR with a large effect size in five out of 12 datasets (CM1, PC1, PC2, PC3, and PC4).

In terms of ***MCC***, DR is significantly better than TSM with a large effect size in five out of 12 datasets (CM1, KC2, MW1, PC1, and JM1), with a medium effect size in two datasets (MC2 and PC2), and with a small effect size in one dataset (KC1). DR has no significant difference with TSM in one dataset (MC1), and is significantly worse than TSM with a large effect size in three datasets (KC3, PC3, and PC4).

RQ2: How effective is our proposed two-stage ensemble (TSE) algorithm?

Motivation: We have demonstrated the effectiveness of deep representations in RQ1. Now, we need to validate the effectiveness of our proposed TSE.

Approach: Our TSE is benchmarked against three famous ensemble learning methods including Bagging [99], AdaBoost [66], and random forest (RF) [65] in terms of F1, AUC, and MCC on all benchmark datasets. RF is a famous ensemble learning approach and is specially designed for C4.5 decision tree [100] (aka. J48 in WEKA), namely C4.5 decision tree is used as the base learner. To avoid the base learner's potential effect on our research results, we also use C4.5 decision tree as the base learners of AdaBoost and Bagging. For our TSE, bagging, AdaBoost, and random forest are used as the base learners. In this research question, we just use original software metrics, i.e., traditional software metrics, as the independent variables for each dataset. As RQ1, Wilcoxon Rank-Sum test with 5% significance level and Cliff's delta are conducted.

Results: Table 7 presents the results of our TSE and the baseline methods on each benchmark dataset in terms of *F1,* ***AUC***, and ***MCC***. We notice that the F1 values of TSE vary from 0.1525 to 0.6288 across all benchmark datasets, the AUC values of TSE range from 0.6567 to 0.9357, and the MCC values of TSE vary from 0.1242 to 0.5758.

Tables 8 and 9 present the results of p-value and Cliff's delta when we compared our TSE approach with the baseline approaches in terms of *F1,* ***AUC***, and ***MCC***. From Table 8, we notice that except for the case that when compare TSE and random forest on six datasets in terms of ***AUC***, in most cases, the performance of TSE is significantly better than that of all baseline methods in terms of each of the above-mentioned three performance measures. When considering Tables 7–9 together, we notice that:

In terms of *F1*, TSE significantly outperforms RF with a large effect size in 10 out of 12 datasets, with a medium effect size in one dataset (MW1), and with small effect size in one dataset (MC1). TSE is significantly better than Bagging with a large effect size in 11 out of 12 datasets and has no significant differences with Bagging in one dataset (PC2). TSE significantly performs better than AdaBoost with a large effect size in seven out of 12 datasets, with a medium effect size in two datasets (MC1 and MW1), and with a small effect size in one dataset

Table 5
Means and standard deviations of different models with deep representations (DR) and traditional software metrics (TSM) on benchmark datasets.

Dataset	F1		AUC		MCC	
	DR	TSM	DR	TSM	DR	TSM
CM1	0.2808 ± 0.0559	0.2394 ± 0.0489	0.6831 ± 0.0258	0.7584 ± 0.0250	0.2103 ± 0.0559	0.1486 ± 0.0440
KC1	0.3239 ± 0.0124	0.3162 ± 0.0109	0.6839 ± 0.0055	0.6815 ± 0.0050	0.2658 ± 0.0143	0.2597 ± 0.0134
KC2	0.5009 ± 0.0219	0.4881 ± 0.0217	0.7587 ± 0.0126	0.7613 ± 0.0130	0.3900 ± 0.0243	0.3632 ± 0.0246
KC3	0.3176 ± 0.0570	0.4079 ± 0.0597	0.6989 ± 0.0251	0.6683 ± 0.0397	0.1773 ± 0.0537	0.2713 ± 0.0612
MC1	0.2050 ± 0.0513	0.1916 ± 0.0514	0.7570 ± 0.0237	0.7277 ± 0.0296	0.1337 ± 0.1377	0.1146 ± 0.0416
MC2	0.5446 ± 0.0429	0.5251 ± 0.0457	0.7202 ± 0.0368	0.6622 ± 0.0374	0.3462 ± 0.0583	0.2962 ± 0.0664
MW1	0.4251 ± 0.0576	0.3474 ± 0.0626	0.6828 ± 0.0388	0.6565 ± 0.0425	0.3513 ± 0.0662	0.2302 ± 0.0647
PC1	0.3064 ± 0.0359	0.2028 ± 0.0440	0.7166 ± 0.0172	0.8122 ± 0.0159	0.3280 ± 0.0365	0.1646 ± 0.0368
PC2	0.3848 ± 0.1029	0.2843 ± 0.0831	0.7085 ± 0.0454	0.7464 ± 0.0531	0.1452 ± 0.0981	0.0715 ± 0.0721
PC3	0.1561 ± 0.0205	0.2992 ± 0.0187	0.8099 ± 0.0036	0.8192 ± 0.0093	0.1304 ± 0.0285	0.2653 ± 0.0202
PC4	0.5039 ± 0.0109	0.5684 ± 0.0136	0.8970 ± 0.0036	0.9091 ± 0.0043	0.4819 ± 0.0117	0.5352 ± 0.0145
JM1	0.2086 ± 0.0038	0.1937 ± 0.0029	0.6863 ± 0.0012	0.6835 ± 0.0009	0.1919 ± 0.0043	0.1866 ± 0.0036
Win/tie/lose	8/1/3		6/1/5		8/1/3	

Note: Win denotes our DR is significantly better than TSM. Tie means that the difference between our DR and TSM is not significant. Lose represents that TSM is significantly better than our DR.

Table 6
p-value and Cliff's delta between deep representation (DR) and traditional software metric (TSM) in terms of various measures on benchmark datasets.

Dataset	F1		AUC		MCC	
	p-value	Cliff's delta	*p*-value	Cliff's delta	*p*-value	Cliff's delta
CM1	.0003	0.4196	.000*	−0.9728	.000*	0.6136
KC1	.0012	0.3776	.0193	0.2720	.0135	0.2872
KC2	.0048	0.3280	.1962	−0.1504	.000*	0.5688
KC3	.000*	−0.7280	.000*	0.4968	.000*	−0.7520
MC1	.3189	0.1491	.000*	0.5616	.6401	0.0589
MC2	.0498	0.2280	.000*	0.7424	.0004	0.4128
MW1	.000*	0.6576	.0043	0.3320	.000*	0.8232
PC1	.000*	0.9080	.000*	−1.0000	.000*	1.0000
PC2	.000*	0.5701	.0018	−0.3968	.000*	0.4712
PC3	.000*	−1.0000	.000*	−0.5984	.000*	−0.9984
PC4	.000*	−1.0000	.000*	−0.9504	.000*	−0.9968
JM1	.000*	0.9992	.000*	0.9384	.000*	0.6672

('0.000*' denotes a very small number.)

(CM1). AdaBoost significantly performs better than TSE with a large effect size in two datasets (MC1 and PC2).

In terms of ***AUC***, TSE significantly outperforms RF significantly with a large effect size in one dataset and with a medium effect size in two datasets. TSE has no significant differences with RF in six out of 12 datasets, and is worse than RF in three datasets. TSE significantly outperforms Bagging with a large effect size in four out of 12 datasets, with a medium effect size in two datasets, and with a small effect size in two datasets. TSE has no significant differences with Bagging in two datasets and is worse than Bagging in two datasets. TSE significantly outperforms AdaBoost with a large effect size in nine out of 12 datasets and with a medium effect size in one dataset. TSE has no significant differences with AdaBoost in two datasets.

In terms of ***MCC***, TSE significantly outperforms RF with a large effect size in eight out of 12 datasets, with a medium effect size in one dataset, and with a small effect size in one dataset. TSE has no significant differences with RF in two datasets. TSE significantly performs better than Bagging with a large effect size in six out of 12 datasets, with a medium effect size in two datasets, and with a small effect size in two datasets. TSE is worse than Bagging in two datasets. TSE significantly outperforms AdaBoost with a large effect size in eight out of 12 datasets and with a medium effect size in two datasets. TSE has no significant differences with AdaBoost in one dataset and is worse than it in one dataset.

RQ3: How effective the proposed SDP model?

Motivation: We have validated the effectiveness of our SDAEsTSE's two phases in RQ1 and RQ2, respectively. The results demonstrate that both two phases are effective. Now, we need to validate the effectiveness of SDAEsTSE.

Approach: We compare our SDAEsTSE with four state-of-the-art SDP models: random forest (RF) [38,101], Bayesian networks (BN) [102], Naive Bayes (NB) [103] and AdaBoost_NN (AB_NN) [22] in terms of *F1*, ***AUC***, and ***MCC*** on benchmark datasets. AdaBoost_NN, which takes neural networks as base learners, is a non-cost-sensitive version of the proposed CSBNN model by Zheng [22]. For SDAEsTSE, in DL phase, we use the same setting in RQ1, and in EL phase, we use logistic regression as the base learners of Bagging and AdaBoost. To obtain convincing results, we also conduct 5-fold CV 50 runs to report the average results, perform Wilcoxon Rank-Sum test with 5% significance level, and report Cliff's Delta.

Results: Table 10 presents the results of our SDAEsTSE approach and the baseline approaches on each benchmark dataset in terms of *F1*, ***AUC***, and ***MCC***. We notice that the *F1* values of SDAEsTSE vary from 0.215 to 0.587, the ***AUC*** values of SDAEsTSE vary from 0.685 to 0.961, and the ***MCC*** values of SDAEsTSE range from 0.137 to 0.492.

Tables 11 and 12 present the results of p-value and Cliff's delta when we compared our SDAEsTSE approach with the baseline approaches in terms of *F1*, ***AUC***, and ***MCC***. When considering Tables 10–12 together, we find that:

With respect to *F1*, SDAEsTSE significantly outperforms AB_NN with a large effect size in nine out of 12 datasets, with a medium effect size in one dataset (KC2), and with a small effect size in one dataset (PC1). AB_NN outperforms SDAEsTSE in one dataset (PC2). SDAEsTSE significantly outperforms RF with a large effect size in 10 out of 12 datasets, has no significantly differences in one dataset (KC2), and is worse than it in one dataset (MC1). SDAEsTSE significantly outperforms BN with a large effect size in five out of 12 datasets, has no significant differences with BN in four datasets (CM1, KC3, MW1, and PC3), and is worse than BN in three datasets (KC1, KC2, and JM1). SDAEsTSE significantly outperforms NB with a larger effect size in nine out of 12 datasets and with a small effect size in one dataset. In two datasets (CM1 and KC3), SDAEsTSE has no significant differences with NB.

With respect to ***AUC***, SDAEsTSE significantly outperforms AB_NN with a large effect size in ten out of 12 datasets, has no significant differences with AB_NN in one dataset (KC2), and is worse than AB_NN in one dataset (MC2). SDAEsTSE significantly outperforms RF with a large effect size in eight out of 12 datasets, has no significant differences with RF in one dataset (KC3), and is worse than RF in three datasets (KC2, PC4, and MC2). SDAEsTSE is significantly better than BN with a large effect size in 10 out of 12 datasets and with a medium effect size in one dataset (MC2). In one dataset (KC2), BN is better than

Table 7
Means and standard beviations of different models in terms of F1, AUC, and MCC on benchmark datasets. Best performer is in bold font.

Data	Proposed TSE			Random forest			Bagging			AdaBoost		
	F1	AUC	MCC	F1	AUC	MCC	F1	AUC	MCC	F1	AUC	MCC
CM1	**0.2399-** ± **0.04-21**	0.656-7 ± 0.0-492	**0.1333-** ± **0.05-00**	0.166-1 ± 0.0-322	**0.7026-** ± **0.01-97**	0.015-1 ± 0.0-439	0.196-1 ± 0.0-664	**0.6810-** ± **0.03-34**	0.071-5 ± 0.0-581	0.214-8 ± 0.0-490	0.657-0 ± 0.0-313	0.089-7 ± 0-.0527
KC1	**0.4637-** ± **0.01-55**	**0.6717-** ± **0.01-21**	0.240-5 ± 0.0-223	0.372-4 ± 0.0-182	**0.6769-** ± **0.01-00**	0.245-6 ± 0.0-224	0.382-5 ± 0.0-194	**0.6681-** ± **0.01-03**	**0.2622-** ± **0.02-18**	0.387-5 ± 0.0-231	0.648-5 ± 0.0-134	0.210-6 ± 0-.0278
KC2	**0.5548-** ± **0.01-86**	0.764-5 ± 0.0-115	0.358-7 ± 0.0-269	0.513-5 ± 0.0-267	0.764-8 ± 0.0-115	0.366-6 ± 0.0-35	0.529 ± 0.0283	**0.7719-** ± **0.01-59**	**0.3913-** ± **0.03-41**	0.505-6 ± 0.0-256	0.733-9 ± 0.0-185	0.337-4 ± 0-.0343
KC3	**0.4382-** ± **0.04-60**	**0.7262-** ± **0.02-74**	**0.306** ± **0.0587**	0.255-8 ± 0.0-607	**0.7305-** ± **0.02-36**	0.122-7 ± 0.0-874	0.380-6 ± 0.0-713	**0.7223-** ± **0.03-16**	0.268-5 ± 0.0-863	0.361-2 ± 0.0-642	0.703-6 ± 0.0-339	0.237-2 ± 0-.0797
MC1	0.289-1 ± 0.0-249	0.861-1 ± 0.0-282	0.328-7 ± 0.0-319	0.272-2 ± 0.0-528	**0.8795-** ± **0.02-71**	0.268-0 ± 0.0-649	0.210-5 ± 0.0-568	0.812-8 ± 0.0-323	0.159-3 ± 0.0-827	**0.4045-** ± **0.05-69**	0.829-4 ± 0.0-377	**0.41-31** ± **0.06-20**
MC2	**0.565** ± **0.0509**	**0.7352-** ± **0.03-39**	**0.3289-** ± **0.07-36**	0.469-3 ± 0.0-387	0.703-0 ± 0.0-327	0.295-3 ± 0.0-585	0.496-8 ± 0.0-545	0.717-2 ± 0.0-354	0.288-2 ± 0.0-765	0.532-3 ± 0.0-53	**0.7294-** ± **0.03-54**	**0.32-57** ± **0.07-65**
MW1	**0.3969-** ± **0.04-27**	**0.7364-** ± **0.02-92**	**0.3207-** ± **0.05-06**	0.363-9 ± 0.0-678	**0.7401-** ± **0.02-73**	0.275-2 ± 0.0-848	0.355-3 ± 0.0-611	0.714-4 ± 0.0-526	0.264-3 ± 0.0-757	0.353-3 ± 0.0-597	0.699-1 ± 0.0-449	0.233-0 ± 0-.0730
PC1	**0.3622-** ± **0.03-31**	**0.8251-** ± **0.01-56**	**0.3179-** ± **0.03-63**	0.260-1 ± 0.0-362	0.817-0 ± 0.0-169	0.251-9 ± 0.0-420	0.242-4 ± 0.0-388	0.801-3 ± 0.0-204	0.230-5 ± 0.0-424	0.283-6 ± 0.0-448	0.797 ± 0.0195	0.252-4 ± 0-.0467
PC2	0.152-5 ± 0.0-317	**0.7585-** ± **0.06-85**	**0.1242-** ± **0.05-21**	NaN ± NaN	**0.7726-** ± **0.05-36**	-0.011-3 ± 0.0-028	0.333-3 ± 0	0.715-1 ± 0.0-803	-0.002-5 ± 0.0-561	**0.3871-** ± **0.15-16**	0.693-4 ± 0.0-604	0.015-5 ± 0-.0608
PC3	**0.4000-** ± **0.02-57**	**0.8250-** ± **0.00-89**	**0.3095-** ± **0.02-96**	0.178-4 ± 0.0-254	**0.8280-** ± **0.00-67**	0.174-7 ± 0.0-323	0.242-9 ± 0.0-379	0.804-5 ± 0.0-166	0.206-3 ± 0.0-401	0.309-0 ± 0.0-300	0.773-2 ± 0.0-149	0.237-9 ± 0-.0320
PC4	**0.6288-** ± **0.01-85**	0.935-7 ± 0.0-038	**0.5758-** ± **0.02-18**	0.488-0 ± 0.0-284	**0.9376-** ± **0.00-38**	0.473-4 ± 0.0-293	0.553-1 ± 0.0-271	0.918-7 ± 0.0-078	0.500-6 ± 0.0-309	0.573-5 ± 0.0-271	0.914-9 ± 0.0-082	0.519-6 ± 0-.0294
JM1	**0.3957-** ± **0.00-72**	**0.6787-** ± **0.00-38**	**0.2217-** ± **0.00-91**	0.285-7 ± 0.0-07	0.677-0 ± 0.0-034	0.203-5 ± 0.0-076	0.306-9 ± 0.0-099	0.675-4 ± 0.0-045	0.216-2 ± 0.0-104	0.331-2 ± 0.0-097	0.646-7 ± 0.0-063	0.175-8 ± 0-.0103
Win/tie/lose				12/0/0	3/6/3	10/0/2	11/1/0	8/2/2	10/0/2	10/0/2	10/2/0	10/1/1

Table 8
p-value for our proposed TSE compared with the baselines in terms of F1, AUC, and MCC.

Dataset	Proposed TSE vs random forest			Proposed TSE vs bagging			Proposed TSE vs AdaBoost		
	F1	AUC	MCC	F1	AUC	MCC	F1	AUC	MCC
CM1	0.000*	0.000*	0.000*	0.000*	0.0102	0.000*	0.0096	0.8985	0.000*
KC1	0.000*	0.0532	0.1566	0.000*	0.0793	0.000*	0.000*	0.000*	0.000*
KC2	0.000*	0.7960	0.2211	0.000*	0.0029	0.000*	0.000*	0.000*	0.0013
KC3	0.000*	0.4713	0.000*	0.000*	0.4219	0.0132	0.000*	0.0005	0.000*
MC1	0.0080	0.0024	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*
MC2	0.000*	0.000*	0.0296	0.000*	0.0106	0.0179	0.0039	0.3328	0.8931
MW1	0.0009	0.4545	0.0014	0.000*	0.0291	0.0003	0.0002	0.000*	0.000*
PC1	0.000*	0.0377	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*
PC2	NaN	0.2966	0.000*	0.0960	0.0026	0.000*	0.000*	0.000*	0.000*
PC3	0.000*	0.1001	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*
PC4	0.000*	0.0276	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*
JM1	0.000*	0.000*	0.000*	0.000*	0.000*	0.0033	0.000*	0.000*	0.000*

('0.000*' denotes a very small number.)

SDAEsTSE. SDAEsTSE significantly outperforms NB with a large effect size in 10 out of 12 datasets, and is worse than NB in two datasets (KC2 and MC2).

With respect to ***MCC***, SDAEsTSE significantly outperforms AB_NN with a large effect size in seven out of 12 datasets, with a medium effect size in one dataset, and with a small effect size in one dataset. SDAEsTSE has no significantly differences with AB_NN in one dataset (KC1) and is worse than AB_NN in two datasets (KC2 and PC1). SDAEsTSE significantly outperforms RF with a large effect size in six out of 12 datasets and with a medium effect size in two datasets (MC2 and PC4). SDAEsTSE has no significant differences with RF in two datasets, and is worse than RF in two datasets. SDAEsTSE significantly outperforms BN with a large effect size in four out of 12 datasets and a small effect size in one dataset. SDAEsTSE has no significant differences with BN in six out of 12 datasets, and is worse than BN in one dataset (KC2). SDAEsTSE significantly outperforms NB with a large effect size in seven out of 12 datasets and with a medium effect size in two datasets. SDAEsTSE has no significant differences with NB in one dataset, and is worse than NB in two datasets.

RQ4: How much time does it take for using SDAEsTSE?

Motivation: We have examined the effectiveness of the proposed SDAEsTSE and each of its two main phases (i.e., DL phase and EL phase). Time efficiency of a method is also a crucial indicator to evaluate whether the method is good enough or not.

Approach: To investigate SDAEsTSE's the time efficiency, for each dataset, we run SADEsTSE 50 times and measure the average model training time and testing time on each benchmark dataset. Model training time refers to the time taken from data preprocessing to obtaining a strong classifier in the EL phase of SDAEsTSE. The testing time is the time taken from preprocessing testing data until the class labels are predicted. We compare the training and testing time of SDAEsTSE with the baseline methods used in RQ3.

Results: Tables 13 and 14 present the model training and testing time (in seconds) respectively on each benchmark dataset. From Table 13, we note that our proposed SDAEsTSE approach takes the most amount of average training time, i.e., 100.7547 s, across all benchmark datasets, and NB spends the least amount of average training time (0.0504 s). The main reason why SDAEsTSE takes so much training time is that amount of time is taken in DL phase to select the best parameters. From Table 14, we can see that the average testing time across all benchmark datasets of our SDAEsTSE method and the baseline methods, i.e., AB_NN, RF, NB, and BN, are 0.0788 s, 0.0203 s, 0.0207 s, 0.0185 s, and 0.0171 s, respectively. Although SDAEsTSE takes the largest testing time, we believe it is still acceptable. Furthermore, we can use many technologies, e.g., parallel computing, to speed up the training of SDAEsTSE in the future.

5. Discussion

5.1. Effects of corruption rate of SDAEs on classifier

As mentioned in Section 3.3.1, SDAEs have a key parameter, namely corruption rate (CR). The aim of this discussion is to present the effects of different CR values on the performance of classifier more intuitively.

Because of the space limitation, here, we just randomly select CM1 and PC1 as benchmark datasets. We use logistic regression as modeling technology, and evaluate the built model in terms of *F1,* ***AUC***, and ***MCC***. The value of CR is varied from 0 to 1 with the increment of 0.1. To eliminate the influence of randomness and make the results more reliable, we perform 5 folds cross-validation 50 times on each dataset.

Table 9
Cliff's delta for our proposed TSE compared with baselines in terms of F1, AUC, and MCC.

Dataset	Proposed TSE vs random forest			Proposed TSE vs bagging			Proposed TSE vs AdaBoost		
	F1	AUC	MCC	F1	AUC	MCC	F1	AUC	MCC
CM1	0.8185	−0.5944	0.9368	0.5616	−0.2984	0.5824	0.3008	0.0152	0.4584
KC1	1.0000	−0.2248	−0.1648	1.0000	0.2040	−0.5192	0.9976	0.7984	0.5984
KC2	0.8016	−0.0304	−0.1424	0.5560	−0.3456	−0.5384	0.8792	0.8288	0.3728
KC3	0.9704	−0.0840	0.9072	0.5072	0.0936	0.2880	0.6896	0.4024	0.5032
MC1	0.3080	−0.3528	0.6160	0.8009	0.7360	0.9437	−0.9080	0.5000	−0.8424
MC2	0.8632	0.5000	0.2528	0.6432	0.2968	0.2752	0.3352	0.1128	−0.0160
MW1	0.3864	−0.0872	0.3712	0.4784	0.2536	0.4248	0.4336	0.5360	0.6592
PC1	0.9392	0.2416	0.7928	0.9832	0.6440	0.9048	0.8496	0.7408	0.7320
PC2	NaN	−0.1345	0.9600	−1.0000	0.3869	0.9040	−0.9667	0.5321	0.7728
PC3	1.0000	−0.1912	0.9976	1.0000	0.7416	0.9592	1.0000	0.9744	0.8872
PC4	1.0000	−0.2560	0.9976	0.9888	0.9824	0.9720	0.9088	0.9904	0.8728
JM1	1.0000	0.2592	0.8816	1.0000	0.4592	0.3416	1.0000	1.0000	1.0000

Table 10
Means and standard deviations of different models in terms of evaluation measures on benchmark datasets. Best performer is in bold font.

Data	Proposed SDAEsSTE			AdaBoost_NN			Random forest			Bayesian network			Naive Bayes		
	F1	AUC	MCC	F1	AUC	MCC	F1	AUC	MCC	F1	AUC	MCC	F1	AUC	MCC
CM1	**0.2882** ± 0.0460	**0.8373** ± 0.0105	**0.1877** ± 0.0466	0.2163 ± 0.0550	0.6409 ± 0.0403	0.0796 ± 0.0603	0.1713 ± 0.0349	0.7041 ± 0.0240	0.0088 ± 0.0401	**0.2892** ± 0.0164	0.7215 ± 0.0139	**0.2000** ± 0.0228	**0.2962** ± 0.0428	0.7138 ± 0.0199	**0.1863** ± 0.0253
KC1	0.3946 ± 0.0154	**0.7426** ± 0.0064	**0.2532** ± 0.0196	0.3129 ± 0.0257	0.6337 ± 0.0158	**0.2519** ± 0.0218	0.3738 ± 0.0151	0.6768 ± 0.0087	**0.2474** ± 0.0194	**0.4523** ± 0.0094	0.6669 ± 0.0040	0.2338 ± 0.0110	0.3785 ± 0.0061	0.6722 ± 0.0034	0.2281 ± 0.0074
KC2	0.5235 ± 0.0199	0.7281 ± 0.0135	0.3423 ± 0.0263	0.5046 ± 0.0306	0.7276 ± 0.0224	0.3955 ± 0.0278	0.5199 ± 0.0248	0.7648 ± 0.0110	0.3749 ± 0.0316	**0.6007** ± 0.0111	**0.7963** ± 0.0075	**0.4365** ± 0.0166	0.4853 ± 0.0168	0.7823 ± 0.0061	0.4107 ± 0.0205
KC3	**0.3750** ± 0.0442	**0.7237** ± 0.0231	0.2196 ± 0.0596	0.3239 ± 0.0538	0.5818 ± 0.0496	0.1636 ± 0.0757	0.2669 ± 0.0635	**0.7333** ± 0.0236	0.1340 ± 0.0959	**0.3592** ± 0.0664	0.6017 ± 0.0413	0.2167 ± 0.0797	**0.3808** ± 0.0407	0.6717 ± 0.0207	**0.2447** ± 0.0369
MC1	0.2217 ± 0.0375	**0.9614** ± 0.0030	0.1904 ± 0.0369	0.1816 ± 0.0433	0.6138 ± 0.0504	0.1057 ± 0.0654	**0.2668** ± 0.0592	0.8793 ± 0.0268	**0.2614** ± 0.0683	0.1385 ± 0.0160	0.7877 ± 0.0133	0.1529 ± 0.0184	0.1166 ± 0.0174	0.7311 ± 0.0211	0.0980 ± 0.0245
MC2	**0.5871** ± 0.0382	0.6846 ± 0.0325	**0.3547** ± 0.0603	0.5393 ± 0.0571	0.7140 ± 0.0411	0.3264 ± 0.0791	0.4748 ± 0.0359	0.7100 ± 0.0302	0.3059 ± 0.0614	0.5020 ± 0.0473	0.6584 ± 0.0302	**0.3311** ± 0.0656	0.4658 ± 0.0376	**0.7246** ± 0.0191	0.3140 ± 0.0361
MW1	**0.4073** ± 0.0423	**0.8597** ± 0.0148	**0.3343** ± 0.0417	0.3539 ± 0.0574	0.6215 ± 0.0443	0.2391 ± 0.0534	0.3556 ± 0.0514	0.7420 ± 0.0271	0.2597 ± 0.0707	**0.3922** ± 0.0343	0.7400 ± 0.0280	**0.3193** ± 0.0342	0.3769 ± 0.0242	0.7383 ± 0.0305	0.3003 ± 0.0216
PC1	**0.3062** ± 0.0253	**0.9062** ± 0.0042	0.2576 ± 0.0273	0.2949 ± 0.0430	0.7258 ± 0.0328	**0.3054** ± 0.0332	0.2558 ± 0.0326	0.8168 ± 0.0167	0.2488 ± 0.0377	0.2352 ± 0.0116	0.7642 ± 0.0099	0.1899 ± 0.0169	0.2947 ± 0.0165	0.6946 ± 0.0144	0.2361 ± 0.0177
PC2	0.2154 ± 0.0378	**0.9264** ± 0.0067	**0.1374** ± 0.0426	**0.3622** ± 0.1118	0.5874 ± 0.0569	0.0445 ± 0.1099	NaN ± NaN	0.7708 ± 0.0541	-0.0120 ± 0.0024	0.1387 ± 0.0281	0.8191 ± 0.0450	**0.1534** ± 0.0383	0.1521 ± 0.0443	0.7501 ± 0.0516	0.0565 ± 0.0438
PC3	**0.3545** ± 0.0236	**0.8381** ± 0.0059	**0.2661** ± 0.0267	0.2524 ± 0.0455	0.7473 ± 0.0225	0.1881 ± 0.0478	0.1822 ± 0.0276	0.8290 ± 0.0081	0.1790 ± 0.0312	**0.3562** ± 0.0081	0.7744 ± 0.0051	**0.2721** ± 0.0127	0.2705 ± 0.0080	0.7657 ± 0.0078	0.1423 ± 0.0149
PC4	**0.5544** ± 0.0166	0.8846 ± 0.0045	**0.4917** ± 0.0189	0.5062 ± 0.0285	0.8022 ± 0.0397	0.4615 ± 0.0264	0.4893 ± 0.0244	**0.9376** ± 0.0034	0.4732 ± 0.0247	0.4267 ± 0.0075	0.8253 ± 0.0037	0.3457 ± 0.0102	0.4170 ± 0.0235	0.836 ± 0.0034	0.3574 ± 0.0233
JM1	0.3174 ± 0.0061	**0.7731** ± 0.0018	**0.2217** ± 0.0069	0.1626 ± 0.0523	0.6303 ± 0.0111	0.1642 ± 0.0218	0.2862 ± 0.0063	0.6775 ± 0.0027	0.2044 ± 0.0072	**0.4280** ± 0.0015	0.6743 ± 0.0008	0.2191 ± 0.0021	0.2637 ± 0.0018	0.6566 ± 0.0009	0.1903 ± 0.0022
Win/tie/lose				11/0/1	10/1/1	9/1/2	10/1/1	8/1/3	8/2/2	5/4/3	11/0/1	5/6/1	10/2/0	10/0/2	9/1/2

Note: Win denotes our proposed SDAEsTSE is significantly better than the baseline method. Tie means that the difference between our SDAEsTSE and the baseline method is not significant. Lose represents that the baseline method is significantly better than our SDAEsTSE.

Table 11
p-value of our model compared with baseline models in terms of evaluation measures on benchmark datasets.

Dataset	SDAEsTSE vs AdaBoost_NN			SDAEsTSE vs random forest			SDAEsTSE vs Bayesian network			SDAEsTSE vs naive Bayes		
	F1	AUC	MCC	F1	AUC	MCC	F1	AUC	MCC	F1	AUC	MCC
CM1	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*	0.8174	0.000*	0.0893	0.7590	0.000*	0.8659
KC1	0.000*	0.000*	0.9204	0.000*	0.000*	0.1045	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*
KC2	0.0006	0.9094	0.000*	0.4841	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*
KC3	0.000*	0.000*	0.0002	0.000*	0.0566	0.000*	0.1356	0.000*	0.7590	0.8174	0.000*	0.0276
MC1	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*
MC2	0.000*	0.0004	0.0475	0.000*	0.0002	0.0002	0.000*	0.0001	0.1217	0.000*	0.000*	0.0003
MW1	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*	0.1060	0.000*	0.1234	0.000*	0.000*	0.000*
PC1	0.0154	0.000*	0.000*	0.000*	0.000*	0.2685	0.000*	0.000*	0.000*	0.0166	0.000*	0.000*
PC2	0.000*	0.000*	0.000*	NaN	0.000*	0.000*	0.000*	0.000*	0.0817	0.000*	0.000*	0.000*
PC3	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*	0.8605	0.000*	0.2454	0.000*	0.000*	0.000*
PC4	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*
JM1	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*	0.000*	0.0140	0.000*	0.000*	0.000*

('0.000*' denotes a very small number.)

The experimental results are shown in Fig. 4, Fig. 5, and Fig. 6, respectively.

We notice that (1) for any one of the performance measures, i.e., ***F1***, ***AUC***, and ***MCC***, it varies with different CR values, especially for F1 and MCC (see Figs. 4 and 6), there is a big gap between the maximum and minimum. (2) For the same performance measure, its change trend is usually different on different datasets along with the change of CR. However, we also find that for AUC, the change tendency on CM1 is similar to that on PC1. From Table 15, we notice that the effects of different corruption rates on classifier's performance are significant and substantial.

In general, it is very necessary to select suitable corruption rate for constructing SDAEs.

5.2. Why our proposed ensemble learning approach (i.e., TSE) is more effective in addressing class imbalance or/and overfitting?

Overfitting refers to the phenomenon that a model has good performance on the training data set, but has poor performance on the test dataset. Some of the potential key causes of model overfitting include 1) insufficient training data, 2) data noise, and 3) constructing a mode that is too complex than it needs to be [80]. Many methods have been proposed to prevent overfittings, such as early stopping [104], model selection [105], dropout [106], and ensemble learning [65]. Freund et al. [107] have demonstrated that averaging classifiers can protect against overfitting.

Take binary classification as an example, we think overfitting can be divided into two cases: (1) the model performs badly on both positive and negative samples in the test set, (2) model performs well on negative samples but performs badly on positive samples. Class imbalance usually results in the occurrence of the second case [15,16], and in this case the positive class is often called as minority class (e.g., the defective proneness of software modules in SDP), the negative is often called as majority class (e.g., the non-defective proneness of software modules). That is to say, addressing class imbalance problem helps to address overfitting. Methods for addressing class imbalance can be divided into three main categories [23]: (1) resampling, (2) cost-sensitive learning, (3) ensemble learning. Obviously, ensemble learning not only helps to address overfitting but helps to address class imbalance problem. This is the primary reason why we use ensemble learning in our SDAEsTSE. Unfortunately, classic ensemble methods, such as random forest, bagging, and AdaBoost, are not immune to class imbalance [73,74,83].

Our proposed ensemble learning approach, i.e., two-stage ensemble learning (TSE), is different from existing classic ensemble learning methods. According to RQ 2 in Section 4.4, the results show that TSE outperforms the classic ensemble learning methods.

The main reasons that our TSE is more effective to prevent overfitting / class-imbalance are two aspects: (1) **base learner**, on one hand, TSE uses ensemble classifiers as base learners, and it has been demonstrated that ensemble learning help to address class imbalance problem [16,23,107]. On the other hand, TSE is heterogeneous ensemble because different base learners are used, and it has been demonstrated that heterogeneous ensemble outperforms homogenous ensemble [82]. (2) **combining rule**, TSE first uses weighted average probabilities to combine base learners. Considering that classic ensemble learning methods are re not immune to class imbalance [73,74,83], especially when the class imbalance is serious. Then TSE adjusts the combined probabilities according to the severity of class-imbalance of defect dataset. More clearly, for a completely class-

Table 12
Cliff's delta of our model compared with baseline models in terms of evaluation measures on benchmark datasets.

Dataset	SDAEsTSE vs AdaBoost_NN			SDAEsTSE vs random forest			SDAEsTSE vs Bayesian network			SDAEsTSE vs naive Bayes		
	F1	AUC	MCC	F1	AUC	MCC	F1	AUC	MCC	F1	AUC	MCC
CM1	0.6727	1.0000	0.8504	0.9569	1.0000	0.9976	−0.0272	1.0000	−0.1976	−0.0360	1.0000	0.0200
KC1	0.9960	1.0000	0.0120	0.6576	1.0000	0.1888	−0.9984	1.0000	0.5664	0.6408	1.0000	0.7168
KC2	0.4000	0.0136	−0.8432	0.0816	−0.9720	−0.5696	−1.0000	−1.0000	−1.0000	0.8544	−1.0000	−0.9680
KC3	0.5656	0.9960	0.4376	0.8008	−0.2216	0.5248	0.1736	0.9960	0.0360	−0.0272	0.9024	−0.2560
MC1	0.5298	1.0000	0.7424	−0.5056	1.0000	−0.6888	0.9584	1.0000	0.6360	0.9928	1.0000	0.9672
MC2	0.5136	−0.4144	0.2304	0.9584	−0.4304	0.4264	0.8632	0.4416	0.1800	0.9520	−0.6992	0.4216
MW1	0.5804	1.0000	0.8232	0.5892	1.0000	0.6184	0.1880	1.0000	0.1792	0.4760	1.0000	0.5512
PC1	0.2816	1.0000	−0.7264	0.7752	1.0000	0.1288	1.0000	1.0000	0.9776	0.2784	1.0000	0.4560
PC2	−0.7977	1.0000	0.6880	NaN	1.0000	1.0000	0.9176	1.0000	−0.2024	0.7416	1.0000	0.8120
PC3	0.9664	1.0000	0.8664	1.0000	0.6256	0.9704	−0.0208	1.0000	−0.1352	1.0000	1.0000	1.0000
PC4	0.8656	0.9888	0.6552	0.9736	−1.0000	0.4576	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
JM1	1.0000	1.0000	1.0000	1.0000	1.0000	0.9376	−1.0000	1.0000	0.2856	1.0000	1.0000	1.0000

Table 13
Training time (in seconds).

Dataset	AB_NN	RF	NB	BN	SDAEsTSE
CM1	7.0626	0.0838	0.0089	0.0101	66.2530
KC1	18.9663	0.4696	0.0954	0.0982	123.6531
KC2	4.5620	0.0849	0.0083	0.0092	46.2486
KC3	8.7089	0.0481	0.0067	0.0074	26.3430
MC1	35.5941	0.4367	0.1328	0.1370	275.8729
MC2	5.5363	0.0767	0.0472	0.0478	17.2570
MW1	11.1992	0.0716	0.0274	0.0283	34.3603
PC1	13.9622	0.2574	0.0960	0.0978	118.0352
PC2	17.2835	0.1064	0.0358	0.0376	93.4328
PC3	41.2573	0.3203	0.0483	0.0520	134.8502
PC4	36.2074	0.3631	0.0480	0.0523	171.9960
JM1	152.0825	6.0808	0.6314	0.6774	1771.5640
Avg.	29.3685	0.6999	0.0989	0.1046	239.9888

Table 14
Testing time (in seconds).

Dataset	AB_NN	RF	NB	BN	SDAEsTSE
CM1	0.0118	0.0128	0.0107	0.011	0.0518
KC1	0.0281	0.0335	0.0248	0.0244	0.1204
KC2	0.0098	0.0119	0.0090	0.0092	0.0427
KC3	0.0081	0.0070	0.0065	0.0059	0.0275
MC1	0.0418	0.0459	0.0417	0.0378	0.1919
MC2	0.0058	0.0054	0.0048	0.0048	0.0217
MW1	0.0094	0.0080	0.0078	0.0071	0.0319
PC1	0.0218	0.0235	0.0198	0.0195	0.0923
PC2	0.0188	0.0175	0.0173	0.0155	0.0715
PC3	0.0323	0.0278	0.0280	0.0238	0.0977
PC4	0.0351	0.0346	0.0327	0.0286	0.1170
JM1	0.2549	0.3368	0.2158	0.2055	1.1524
Avg.	0.0398	0.0470	0.0349	0.0328	0.1682

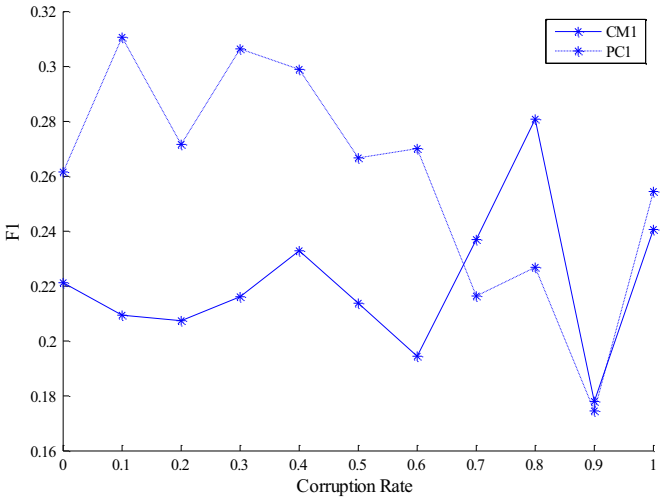

Fig. 4. Effect of corruption rate on F1 for CM1 and PC1.

balanced dataset (i.e., the number of positive instances is equal to the number of negative instances), the default threshold is 0.5, that is to say, if the predicted probability being positive class of an unlabeled instance is larger than 0.5, then this instance is classified as a positive instance. However, for a class-imbalanced dataset, the thread should be changed according to the severity of imbalance. Specifically, for the positive class label, the thread should be smaller than 0.5 (for example the threshold is 0.3), namely if the predicted probability being the positive class of an unlabeled instance is larger than 0.3, then this instance is classified as a positive instance. This is the core idea of

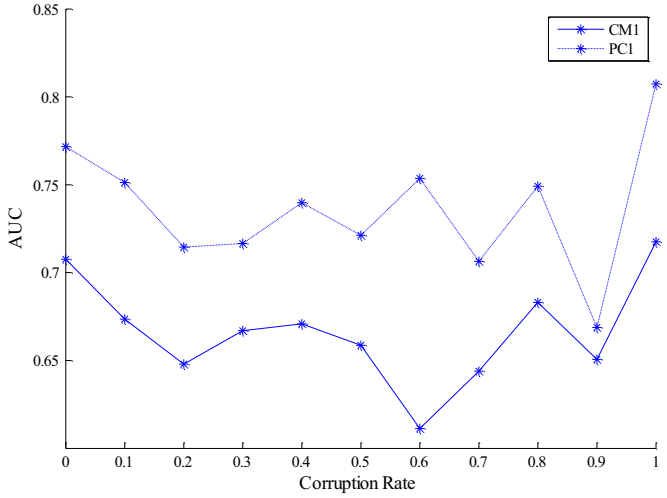

Fig. 5. Effect of corruption rate on AUC for CM1 and PC1.

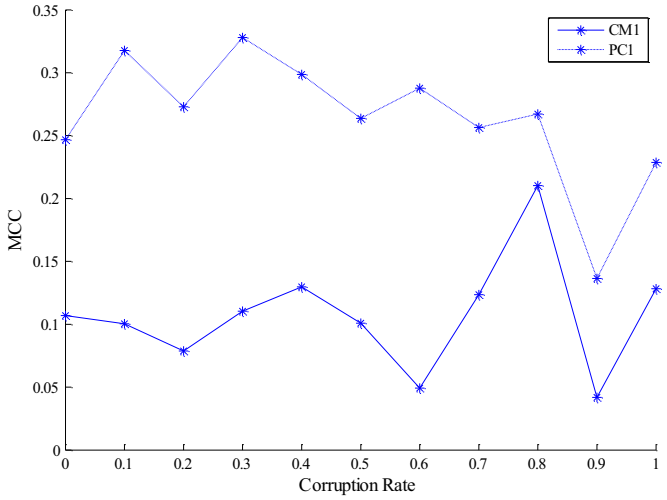

Fig. 6. Effect of corruption rate on AUC for CM1 and PC1.

Table 15
p-value and Cliff's delta compared with different corruption rate (CR) in terms of MCC.

Dataset	CR = 0 vs CR = 0.3		CR = 0 vs CR = 0.6		CR = 0 vs CR = 0.9	
	p-value	Delta	*p*-value	Delta	*p*-value	Delta
CM1	0.9643	0.0056	0.000*	0.6472	0.000*	0.7144
PC1	0.000*	−0.9280	0.000*	−0.5992	0.000*	0.9768

('0.000*' denotes a very small number.)

probability adjustment.

5.3. More insightful discussion on the results

The experimental results have shown that the proposed SDAEsTSE is effective. More discussions about the results are presented as follows: (1) Many performance evaluation measures, such as G-measure, G-mean, AUC, and MCC, have been used to evaluate the performance of SDP models. Different performance evaluation criteria measure the model performance from different perspectives and in different ways. Given two different SDP models, i.e., model-1 and model-2, suppose that model-1 significantly performs better than model-2 in terms of one performance measure, e.g., AUC. However, model-2 might significantly outperform model-1 in terms of

another performance measure, e.g., MCC. This phenomenon has been shown in previous SDP studies [16,91,92]. (2) For F-measure and MCC, there is a certain positive correlation relationship between them. In other words, if a model performs well in terms of F-measure/MCC, the model generally performs well in terms of MCC/F-measure. (3) For AUC and MCC, it seems that there is no positive correlation relationship between them. Although an absolutely good SDP model should perform best in terms of both AUC and MCC on any dataset compared with other models, this kind of models is rare. (4) Multi-objective defect prediction [108,109] may be a promising choice if we hope that the proposed model can perform well in terms of multiple performance measures simultaneously. (5) An SDP model shouldn't be evaluated just by using one of the performance measures on one or a small amount of datasets, any single measure is not enough to demonstrate that a SDP model is a good model.

5.4. Threats to validity

Potential threats to the validity of our research are three folds, which are shown as follows.

Threats to internal validity refer to the errors in the experiments, such as the quality of benchmark datasets, the implementation of baseline methods. We have double checked our experiments, some errors maybe still exist.

Threats to external validity refer to the generalization of our research findings. We have evaluated the effectiveness of deep representations, the proposed two-stage ensemble (TSE), and the proposed SDAEsTSE model with the statistical test on 12 NASA datasets, respectively. Furthermore, we use Wilcoxon rank-sum test and Cliff's delta to perform statistic test and quantify the effect size, respectively. More defect datasets should be considered in the future to reduce the threats to external validity.

Threats to construct validity relate to the suitability of the performance evaluation criteria. According to previous SDP studies [8,17,91], many performance measures, such as *Accuracy, PD, PF, precision*, ***F-measure***, *Balance, G-measure, G-mean1, G-mean2,* ***AUC*** and ***MCC***, have been used to evaluate the performance of binary classification based SDP models. However, we cannot use all these measures, and in fact, no studies do use all these measures to evaluate the SDP model. Considering the fact that highly imbalanced defect datasets, the limited space, and the latest research in SDP area, in this study, three performance measures including ***F-measure***, ***AUC***, and ***MCC*** are utilized. This may pose some threats to the construct validity.

6. Conclusion and future work

Software defect prediction (SDP) plays an important role in allocating testing resources reasonably and ensuring software quality. In this paper, we propose a novel two-phase SDP model, named SDAEsTSE, which is build based on stacked denoising autoencoders (SDAEs) and our proposed two-stage ensemble learning. To obtain more robust and representative features compared with commonly used traditional software metrics in previous SDP studies, we use SDAEs to extract deep representations in the first phase, i.e., DL phase. To address class-imbalance, we take ensemble learning strategy. To eliminate the overfitting problem in existing ensemble methods, we propose a two-stage ensemble (TSE) learning method as the second phase. i.e., EL phase, of our model. To evaluate the effectiveness of deep representations, the proposed TSE, and our SDAEsTSE approach, we performed a series of experiments on 12 public software defect datasets from NASA projects by comparing with the baseline methods in terms of *F1,* ***AUC***, and ***MCC***. We also analyze the time efficiency of the proposed SDEsTSE. In this study, Wilcoxon Rank-Sum test is used for significance test, and Cliff's delta is used for evaluating the effect size.

The experimental results show that (1) deep representations are promising for SDP compared with traditional software metrics, (2) the proposed TSE is effective and outperforms the baseline methods significantly and substantially in the most cases, and 3) the proposed SDAEsTES approach outperforms existing state-of-the-art SDP models significantly and substantially in most of cases in terms of *F1,* ***AUC***, and ***MCC***.

In the future, we plan to extend this method for cross-project defect prediction, which is now a research hotspot in the field of software defect prediction [18,81,88,91,110,111].

Acknowledgments

This research was supported by the Major State Basic Research Development Program of China (973 Program) (No.2014CB744904). We would like to thank the editors and anonymous reviewers for their constructive comments and suggestions for the improvement of this paper. All the results of our analysis can be downloaded from https://github.com/THN-BUAA/SDAEsTSE.

Appendix A

Table 16 presents the results of DP compared with TSM when random forest is used as the modeling method. The results of p-value and Cliff's delta are presented in Table 17

Table 18 presents the results of DP compared with TSM when Bayesian network is used as the modeling method. The results of p-value and Cliff's delta are presented in Table 19.

Table 16
Means and standard deviations of different models with deep representations (DR) and traditional software metrics (TSM).

Dataset	F1		AUC		MCC	
	DR	TSM	DR	TSM	DR	TSM
CM1	0.1865 ± 0.0494	0.1713 ± 0.035	0.7004 ± 0.0268	0.7041 ± 0.024	0.0532 ± 0.0558	0.0088 ± 0.040
KC1	0.3799 ± 0.0135	0.3738 ± 0.015	0.6784 ± 0.0089	0.6768 ± 0.009	0.2617 ± 0.0172	0.2474 ± 0.019
KC2	0.5305 ± 0.0263	0.5199 ± 0.025	0.7414 ± 0.0155	0.7648 ± 0.011	0.3876 ± 0.0321	0.3749 ± 0.032
KC3	0.2704 ± 0.0676	0.2669 ± 0.064	0.6784 ± 0.0330	0.7333 ± 0.024	0.0989 ± 0.0828	0.134 ± 0.096
MC1	0.1936 ± 0.0381	0.2668 ± 0.059	0.7692 ± 0.0368	0.8793 ± 0.027	0.1483 ± 0.0647	0.2614 ± 0.068
MC2	0.5388 ± 0.0417	0.4748 ± 0.036	0.7724 ± 0.0267	0.71 ± 0.0302	0.3631 ± 0.0560	0.3059 ± 0.061
MW1	0.3874 ± 0.0684	0.3556 ± 0.051	0.7366 ± 0.0267	0.742 ± 0.027	0.3447 ± 0.0791	0.2597 ± 0.071
PC1	0.2774 ± 0.0411	0.2558 ± 0.033	0.8262 ± 0.0160	0.8168 ± 0.017	0.2666 ± 0.0465	0.2488 ± 0.038
PC2	NaN ± NaN	NaN ± NaN	0.7726 ± 0.0418	0.7708 ± 0.054	NaN ± NaN	-0.012 ± 0.002
PC3	0.2152 ± 0.0289	0.1822 ± 0.028	0.8028 ± 0.0093	0.829 ± 0.008	0.1976 ± 0.0325	0.179 ± 0.031
PC4	0.4292 ± 0.0197	0.4893 ± 0.024	0.8888 ± 0.0059	0.9376 ± 0.003	0.4328 ± 0.0215	0.4732 ± 0.025
JM1	0.2798 ± 0.0063	0.2862 ± 0.006	0.671 ± 0.0035	0.6775 ± 0.003	0.2025 ± 0.0070	0.2044 ± 0.007

Table 17
p-value and Cliff's delta between deep representation (DR) and traditional software metric (TSM) in terms of various measures.

Dataset	F1		AUC		MCC	
	p-value	Cliff's delta	*p*-value	Cliff's delta	*p*-value	Cliff's delta
CM1	0.1928	0.1627	0.9259	−0.0112	0.0001	0.4504
KC1	0.0318	0.2496	0.4380	0.0904	0.0002	0.4272
KC2	0.0747	0.2072	0.000*	−0.7816	0.0932	0.1952
KC3	0.7433	0.0384	0.000*	−0.8480	0.0603	−0.2184
MC1	0.000*	−0.7363	0.000*	−0.9952	0.000*	−0.8048
MC2	0.000*	0.7456	0.000*	0.9024	0.000*	0.5056
MW1	0.0242	0.2620	0.3574	−0.1072	0.000*	0.5416
PC1	0.0059	0.3200	0.0070	0.3136	0.0207	0.2688
PC2	NaN	NaN	0.9110	−0.0147	NaN	NaN
PC3	0.000*	0.6144	0.000*	−0.9784	0.0102	0.2984
PC4	0.000*	−0.9336	0.000*	−1.0000	0.000*	−0.7792
JM1	0.000*	−0.5584	0.000*	−0.8568	0.1075	−0.1872

('0.000*' denotes a very small number.)

Table 18
Means and standard deviations of different models with deep representations (DR) and traditional software metrics (TSM).

Dataset	F1		AUC		MCC	
	DR	TSM	DR	TSM	DR	TSM
CM1	0.3081 ± 0.0171	0.2892 ± 0.0164	0.7333 ± 0.0113	0.7215 ± 0.0139	0.2232 ± 0.0231	0.2000 ± 0.0228
KC1	0.4450 ± 0.0093	0.4523 ± 0.0094	0.6632 ± 0.0043	0.6669 ± 0.0040	0.2180 ± 0.0110	0.2338 ± 0.0110
KC2	0.6055 ± 0.0239	0.6007 ± 0.0111	0.7785 ± 0.0077	0.7963 ± 0.0075	0.4400 ± 0.0243	0.4365 ± 0.0166
KC3	0.1727 ± 0.0525	0.3592 ± 0.0664	0.5040 ± 0.0135	0.6017 ± 0.0410	−0.0545 ± 0.0786	0.2167 ± 0.0797
MC1	0.1556 ± 0.0222	0.1385 ± 0.0160	0.7552 ± 0.0216	0.7877 ± 0.0133	0.1608 ± 0.0240	0.1529 ± 0.0184
MC2	0.5637 ± 0.0420	0.5020 ± 0.0473	0.7383 ± 0.0275	0.6584 ± 0.0302	0.3427 ± 0.0672	0.3311 ± 0.0656
MW1	0.4269 ± 0.0424	0.3922 ± 0.0343	0.7019 ± 0.0319	0.7400 ± 0.0280	0.3608 ± 0.0392	0.3193 ± 0.0342
PC1	0.2535 ± 0.0140	0.2352 ± 0.0116	0.7649 ± 0.0083	0.7640 ± 0.0099	0.2009 ± 0.0188	0.1899 ± 0.0169
PC2	NaN ± NaN	0.1387 ± 0.0281	0.5708 ± 0.0439	0.8191 ± 0.0450	NaN ± NaN	0.1534 ± 0.0383
PC3	0.4053 ± 0.0151	0.3562 ± 0.0081	0.8037 ± 0.0062	0.7744 ± 0.0051	0.3139 ± 0.0194	0.2721 ± 0.0127
PC4	0.4854 ± 0.0092	0.4267 ± 0.0075	0.8416 ± 0.0032	0.8253 ± 0.0037	0.4050 ± 0.0114	0.3457 ± 0.0102
JM1	0.4159 ± 0.0023	0.428 ± 0.0015	0.6656 ± 0.0010	0.6743 ± 0.0008	0.2192 ± 0.0031	0.2191 ± 0.0021

Table 19
p-value and Cliff's delta between deep representation (DR) and traditional software metric (TSM) in terms of various measures.

Dataset	F1		AUC		MCC	
	p-value	Cliff's delta	*p*-value	Cliff's delta	*p*-value	Cliff's delta
CM1	0.000*	0.5960	0.000*	0.4616	0.000*	0.5248
KC1	0.000*	−0.4552	0.000*	−0.4928	0.000*	−0.7064
KC2	0.0016	0.3672	0.000*	−0.9208	0.0959	0.1936
KC3	0.000*	−0.9547	0.000*	−0.9784	0.000*	−0.9893
MC1	0.000*	0.5608	0.000*	−0.8024	0.0291	0.2536
MC2	0.000*	0.6984	0.000*	0.9496	0.3538	0.1080
MW1	0.000*	0.5344	0.000*	−0.6392	0.000*	0.5992
PC1	0.000*	0.6816	0.9368	0.0096	0.0035	0.3392
PC2	NaN	NaN	0.000*	−0.9943	NaN	NaN
PC3	0.000*	0.9984	0.000*	1.0000	0.000*	0.9176
PC4	0.000*	1.0000	0.000*	0.9944	0.000*	1.0000
JM1	0.000*	−1.0000	0.000*	−1.0000	0.000*	0.0496

('0.000*' denotes a very small number.)

Reference

[1] Y. Ma, K. Qin, S. Zhu, Discrimination analysis for predicting defect-prone software modules, J. Appl. Math. 2014 (1) (2014) 201–211.
[2] Y. Kamei, et al., A large-scale empirical study of just-in-time quality assurance, IEEE Trans. Software Eng. 39 (6) (2013) 757–773.
[3] M.Y. Ricky, et al., Mobile application software defect prediction, 2016 IEEE Symposium on Service-Oriented System Engineering (SOSE), United Kingdom, Oxford, 2016, pp. 307–313.
[4] S. Kanmani, et al., Object-oriented software fault prediction using neural networks, Inf. Software Technol. 49 (5) (2007) 483–492.
[5] S. Singh Rathore, S. Kumar, Predicting number of faults in software system using genetic programming, Proceedings of the 2015 International Conference on Soft Computing and Software Engineering, 2015.
[6] V.B. Singh, K.K. Chaturvedi, et al., Improving the quality of software by quantifying the code change metric and predicting the bugs, in: B Murgante, et al. (Ed.), 13th International Conference on Computational Science and Its Applications – ICCSA 2013, Springer, Berlin, Heidelberg, 2013, pp. 408–426.
[7] G. Czibula, et al., Software defect prediction using relational association rule mining, Inf. Sci. 264 (2014) 260–278.
[8] B. Ma, et al., Investigating associative classification for software fault prediction:

an experimental perspective, Int. J. Software Eng. Knowl. Eng. 24 (1) (2014) 61–90.
[9] E.J. Weyuker, et al., Comparing the effectiveness of several modeling methods for fault prediction, Empirical Software Eng. 15 (3) (2010) 277–295.
[10] X. Yang, et al., A learning-to-rank approach to software defect prediction, IEEE Trans. Reliab. 64 (1) (2015) 234–246.
[11] G. You, M Yutao, A ranking-oriented approach to cross-project software defect prediction: an empirical study, International Conference on Software Engineering & Knowledge Engineering, 2016.
[12] K. Gao, et al., The use of ensemble-based data preprocessing techniques for software defect prediction, Int. J. Software Eng. Knowl. Eng. 24 (9) (2014) 1229–1253.
[13] T.M. Khoshgoftaar, et al., A comparative study of iterative and non-iterative feature selection techniques for software defect prediction, Inf. Syst. Front. 16 (5) (2014) 801–822.
[14] B. Boehm, V.R Basili, Software defect reduction top 10 list. Computer 34 (1) (2001) 135–137.
[15] Z. Mahmood, et al., What is the impact of imbalance on software defect prediction performance, Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering, ACM, Beijing, China, 2015, pp. 1–4.
[16] S. Wang, X. Yao, Using class imbalance learning for software defect prediction, IEEE Trans. Reliab. 62 (2) (2013) 434–443.
[17] Ö.F. Arar, K. Ayan, Software defect prediction using cost-sensitive neural network, Appl. Soft Comput. 33 (2015) 263–277.
[18] Y. Kamei, T. Fukushima, S. Mcintosh, et al., Studying just-in-time defect prediction using cross-project models, Empirical Software Eng. 21 (5) (2016) 2072–2106.
[19] Y. Ma, et al., An improved semi-supervised learning method for software defect prediction, J. Intell. Fuzzy Syst. 27 (5) (2014) 2473–2480.
[20] M.J. Siers, M.Z. Islam, Software defect prediction using a cost sensitive decision forest and voting, and a potential solution to the class imbalance problem, Inf. Syst. 51 (2015) 62–71.
[21] N. Seliya, T.M. Khoshgoftaar, The use of decision trees for cost-sensitive classification: an empirical study in software quality prediction, Wiley Interdiscip. Rev.–Data Mining Knowl. Discovery 1 (5) (2011) 448–459.
[22] J. Zheng, Cost-sensitive boosting neural networks for software defect prediction, Expert Syst. Appl. 37 (6) (2010) 4537–4543.
[23] I.H. Laradji, et al., Software defect prediction using ensemble learning on selected features, Inf. Software Technol. 58 (2014) 388–402.
[24] T. McCabe, A complexity measure, IEEE Trans. Software Eng. 2 (4) (1976) 308–320.
[25] M. Halstead, Elements of Software Science, Elsevier North-Holland, New York, 1997.
[26] S.R. Chindamber, C.F. Keremer, A metrics suite for object oriented design, IEEE Trans. Software Eng. 20 (6) (1994) 476–493.
[27] A. Kaur, et al., Application of machine learning on process metrics for defect prediction in mobile application, in: C.S. Satapathy, et al. (Ed.), Proceedings of Third International Conference Information Systems Design and Intelligent Applications: INDIA 2016, Springer India, New Delhi, 2016, pp. 81–98.
[28] Jiaqiang Chen, et al., A two-stage data preprocessing approach for software fault prediction, 2014 Eighth International Conference on Software Security and Reliability (SERE), 2014, pp. 20–29.
[29] M. Liu, et al., Two-stage cost-sensitive learning for software defect prediction, IEEE Trans. Reliab. 63 (2) (2014) 676–686.
[30] P. He, et al., An empirical study on software defect prediction with a simplified metric set, Inf. Software Technol. 59 (2015) 170–190.
[31] X. Yang, et al., Deep learning for just-in-time defect prediction, Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015 (2015).
[32] G. Hinton, et al., Deep neural networks for acoustic modeling in speech recognition, IEEE Signal Process. Mag. 29 (6) (2012) 82–97.
[33] A. Krizhevsky, et al., ImageNet classification with deep convolutional neural networks, Adv. Neural Inf. Process. Syst. 25 (2) (2012) 2012.
[34] Wanwangying Ma, et al., Empirical analysis of network measures for effort-aware fault-proneness prediction, Inf. Software Technol. 69 (2016) 50–70.
[35] M.S. Bicer, B. Diri, Defect prediction for cascading style sheets, Appl. Soft Comput. 49 (2016) 1078–1084.
[36] M.S. Bicer, B. Diri, Predicting defect prone modules in web applications, 21st International Conference on Information and Software Technologies (ICIST), 2015.
[37] S. Kim, et al., Classifying software changes: clean or buggy, IEEE Trans. Software Eng. 34 (2) (2008) 181–196.
[38] T. Lee, et al., Developer micro interaction metrics for software defect prediction, IEEE Trans. Software Eng. (2016) 1-1.
[39] D. Bowes, et al., Mutation-aware fault prediction, International Symposium on Software Testing and Analysis, 2016.
[40] Santonu Sarkar, et al., Metrics for measuring the quality of modularization of large-scale object-oriented software, IEEE Trans. Software Eng. 34 (5) (2008) 700–720.
[41] Y. Zhao, et al., An empirical analysis of package-modularization metrics: implications for software fault-proneness, Inf. Software Technol. 57 (2015) 186–203.
[42] Y. Zhao, Y. Yang, H. Lu, et al., Understanding the value of considering client usage context in package cohesion for fault-proneness prediction, Autom. Software Eng. 24 (2) (2017) 393–453.
[43] H. Aman, et al., Lines of comments as a noteworthy metric for analyzing fault-proneness in methods, IEICE Trans. Inf. Syst. E98D (12) (2015) 2218–2228.
[44] T. Hall, et al., Some code smells have a significant but small effect on faults, ACM Trans. Software Eng. Methodol. 23 (4) (2014) 1–39.
[45] R. Malhotra, A systematic review of machine learning techniques for software fault prediction, Appl. Soft Comput. 27 (2015) 504–518.
[46] V. Vashisht, Defect prediction framework using neural networks for software enhancement projects, Br. J. Math. Comput. Sci. 16 (5) (2016) 1–12.
[47] C. Jin, S. Jin, Prediction approach of software fault-proneness based on hybrid artificial neural network and quantum particle swarm optimization, Appl. Soft Comput. 35 (2015) 717–725.
[48] A. Okutan, O.T. Yildiz, Software defect prediction using Bayesian networks, Empirical Software Eng. 19 (1) (2014) 154–181.
[49] X. Jing, et al., Dictionary learning based software defect prediction, Proceedings of the 36th International Conference on Software Engineering (ICSE), Hyderabad, India, ACM, 2014.
[50] A. Soleimani, F. Asdaghi, An AIS based feature selection method for software fault prediction, 2014 Iranian Conference on Intelligent Systems (LCIS), 2014.
[51] B. Li, et al., A scenario-based approach to predicting software defects using compressed C4.5 Model, in: C.K. Chang, et al. (Ed.), 2014 IEEE 38th Annual International Computers, Software and Applications Conference, 2014, pp. 406–415.
[52] S. Chatterjee, B. Maji, A new fuzzy rule based algorithm for estimating software faults in early phase of development, Soft Comput. 20 (10) (2016) 4023–4035.
[53] H.B. Yadav, D.K. Yadav, A fuzzy logic based approach for phase-wise software defects prediction using software metrics, Inf. Software Technol. 63 (2015) 44–57.
[54] G. Abaei, et al., An empirical study based on semi-supervised hybrid self-organizing map for software fault prediction, Knowl.-Based Syst. 74 (2015) 28–39.
[55] Z.W. Zhang, X.Y. Jing, T.J. Wang, Label propagation based semi-supervised learning for software defect prediction, Autom. Software Eng. 24 (1) (2016) 1–23.
[56] J. Nam, S. Kim, CLAMI: defect prediction on unlabeled datasets (T), IEEE/ACM International Conference on Automated Software Engineering, 2015, pp. 452–463.
[57] R. Sasidharan, P. Sriram, Hyper-quadtree-based k-means algorithm for software fault prediction, Comput. Intell. Cyber Secur. Comput. Models 246 (2014) 107–118.
[58] A. Karpathy, et al., Large-scale video classification with convolutional neural networks, 2014 IEEE Conference on Computer Vision and Pattern Recognition, 2014.
[59] S. Wang, et al., Automatically learning semantic features for defect prediction, 2016 IEEE/ACM 38th IEEE International Conference on Software Engineering (ICSE), 2016.
[60] P. Vincent, et al., Extracting and composing robust features with denoising autoencoders, Proceedings of the 25th International Conference on Machine Learning, Helsinki, Finland, ACM, 2008, pp. 1096–1103.
[61] Q. Hu, et al., Transfer learning for short-term wind speed prediction with deep neural networks, Renew. Energy 85 (2016) 83–95.
[62] Y. Kang, et al., Stacked denoising autoencoders for face pose normalization, in: M. Lee, et al. (Ed.), Proceedings of 20th International Conference On Neural Information Processing, ICONIP 2013, Springer, Berlin, Heidelberg, 2013, pp. 241–248.
[63] L. Rokach, Ensemble-based classifiers, Artif. Intell. Rev. 33 (1) (2010) 1–39.
[64] L. Breiman, Bagging predictors, Mach. Learn. 24 (2) (1996) 123–140.
[65] L. Breiman, R. Forests, Mach. Learn. 45 (1) (2001) 5–32.
[66] Y. Freund, R.E. Schapire, A decision-theoretic generalization of on-line learning and an application to boosting, J. Comput. Syst. Sci. 55 (1) (1997) 119–139.
[67] Y. Freund, R.E. Shapire, Experiments with a new boosting algorithm, Proceedings of the Thirteenth International Conference on Machine learning, ICML, 1996, pp. 148–156.
[68] IEEE, IEEE Standard Glossary for Software Engineering Terminology, (1990) 610.12–1990.
[69] C. Catal, B. Diri, A systematic review of software fault prediction studies, Expert Syst. Appl. 36 (4) (2009) 7346–7354.
[70] D. Gray, et al., The misuse of the NASA metrics data program data sets for automated software defect prediction, 15th Annual Conference on Evaluation & Assessment in Software Engineering (EASE 2011), 2011.
[71] J. Petric, et al., The jinx on the NASA software defect data sets, Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, Limerick, Ireland, ACM, 2016, pp. 1–5.
[72] A. Vezhnevets, O. Barinova, Avoiding boosting overfitting by removing confusing samples, Proceedings of the 18th European Conference on Machine Learning, Warsaw, Poland, Springer-Verlag, 2007, pp. 430–441.
[73] D.J. Dittman, et al., The effect of data sampling when using random forest on imbalanced bioinformatics data, 2015 IEEE International Conference on Information Reuse and Integration, 2015.
[74] C. Su, et al., Improving random forest and rotation forest for highly imbalanced datasets, Intell. Data Anal. 19 (6) (2015) 1409–1432.
[75] M. Shepperd, et al., Data quality: some comments on the NASA software defect datasets, IEEE Trans. Software Eng. 39 (9) (2013) 1208–1215.
[76] I.H. Witten, et al., Data Mining: Practical Machine Learning Tools and Techniques, Morgan Kaufmann Publishers Inc, 2011, p. 664.
[77] P. Vincent, et al., Stacked denoising autoencoders: learning useful representations in a deep network with a local denoising criterion, J. Mach. Learn. Res. 11 (12) (2010) 3371–3408.
[78] G.E. Hinton, et al., A fast learning algorithm for deep belief nets, Neural Comput. 18 (7) (2006) 1527–1554.
[79] Y. Bengio, et al., Greedy layer-wise training of deep networks, Proceedings of the 19th International Conference on Neural Information Processing Systems, MIT Press, Canada, 2006, pp. 153–160.

[80] P.-N. Tan, et al., Introduction to Data Mining, Addison-Wesley Longman Publishing Co., Inc, 2005.
[81] J. Nam, et al., Transfer defect learning, Proceedings of the 35th International Conference on Software Engineering (ICSE), 2013, IEEE Press, San Francisco, CA, USA, 2013, pp. 382–391.
[82] M. Gashler, et al., Decision tree ensemble: small heterogeneous is better than large homogeneous, 2008 Seventh International Conference on Machine Learning and Applications, 2008.
[83] N.V. Chawla, et al., SMOTEBoost: improving prediction of the minority class in boosting, Lecture Notes in Computer Science 2838 (2003) 107–119.
[84] J. Sayyad Shirabad, T.J. Menzies, The PROMISE Repository of Software Engineering Databases, School of Information Technology and Engineering, University of Ottawa, Canada, 2005 Available from: http://promise.site.uottawa.ca/SERepository.
[85] T. Menzies, et al., The Promise Repository of Empirical Software Engineering Data, Available from, 2015. http://openscience.us/repo, .
[86] W. Li, et al., Three-way decisions based software defect prediction, Knowledge-Based Syst. 91 (2016) 263–274.
[87] A. Arcuri, L. Briand, A practical guide for using statistical tests to assess randomized algorithms in software engineering, Proceedings of the 33rd International Conference on Software Engineering, Honolulu, HI, USA, ACM: Waikiki, 2011, pp. 1–10.
[88] X. Xia, D. Lo, S.J. Pan, et al., HYDRA: massively Compositional model for cross-project defect prediction, IEEE Trans. Software Eng. 42 (10) (2016) 977–998.
[89] A.P. Bradley, The use of the area under the ROC curve in the evaluation of machine learning algorithms, Pattern Recognit. 30 (7) (1997) 1145–1159.
[90] B.W. Matthews, Comparison of the predicted and observed secondary structure of T4 phage lysozyme, Biochim. Biophys. Acta 405 (2) (1975) 442.
[91] L. Chen, et al., Negative samples reduction in cross-company software defects prediction, Inf. Software Technol. 62 (2015) 67–77.
[92] F. Zhang, A. Mockus, I. Keivanloo, et al., Towards building a universal defect prediction model with rank transformed predictors, Empirical Software Eng. 21 (5) (2014) 182–191.
[93] S. Le Cessie, J.C. Van Houwelingen, Ridge estimators in logistic regression, J. R. Stat. Soc. 41 (1) (1992) 191–201.
[94] T. Hall, S. Beecham, D. Bowes, et al., A systematic literature review on fault prediction performance in software engineering. IEEE Trans. Software Eng. 38 (6) (2012) 1276–1304.
[95] S. Lessmann, et al., Benchmarking classification models for software defect prediction: a proposed framework and novel findings, IEEE Trans. Software Eng. 34 (4) (2008) 485–496.
[96] Frank Wilcoxon, Individual comparisons by ranking methods, Biometr. Bull. 1 (6) (1945) 80–83.
[97] Norman Cliff, Ordinal Methods for Behavioral Data Analysis, Lawrence Erlbaum Associates, 1996.
[98] G. Macbeth, et al., Cliff's delta calculator: a non-parametric effect size program for two groups of observations, Univ. Psychol. 10 (2) (2011) 545–555.
[99] L. Breiman, Bagging predictors, Mach. Learn. 24 (2) (1996) 123–140.
[100] J. Ross Quinlan, C4.5: Programs for Machine Learning, Morgan Kaufmann Publishers Inc, 1993, p. 302.
[101] J. Moeyersoms, et al., Comprehensible software fault and effort prediction: a data mining approach, J. Syst. Software 100 (2015) 80–90.
[102] K. Dejaeger, et al., Toward comprehensible software fault prediction models using Bayesian network classifiers, IEEE Trans. Software Eng. 39 (2) (2013) 237–257.
[103] G.H. John, P. Langley, Estimating continuous distributions in Bayesian classifiers, Proceedings of the Eleventh conference on Uncertainty in artificial intelligence, Morgan Kaufmann Publishers Inc, Montréal, Québec, Canada, 1995, pp. 338–345.
[104] A. Weigend, On overfitting and the effective number of hidden units, Proceedings of the 1993 Connectionist Models Summer School, 1993.
[105] J.E. Moody, Number of parameters: an analysis of generalization and regularization in nonlinear learning systems, Advances in Neural Information Processing Systems, 2009.
[106] N. Srivastava, et al., Dropout: a simple way to prevent neural networks from overfitting, J. Mach. Learn. Res. 15 (2014) 1929–1958.
[107] Y. Freund, et al., Why averaging classifiers can protect against overfitting, Proceedings of Theghth International Workshop on Artificial Intelligence & Statistics, 2001.
[108] D. Ryu, J. Baik, Effective multi-objective naïve Bayes learning for cross-project defect prediction, Appl. Soft Comput. 49 (Supplement C) (2016) 1062–1077.
[109] G. Canfora, et al., Multi-objective cross-project defect prediction, 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, 2013, pp. 252–261.
[110] F. Zhang, et al., Cross-project defect prediction using a connectivity-based unsupervised classifier, 2016 IEEE/ACM 38th IEEE International Conference on Software Engineering(ICSE), ACM:, Austin, Texas, 2016, pp. 309–320.
[111] F. Peters, et al., Better cross company defect prediction, Proceedings of the 10th Working Conference on Mining Software Repositories, IEEE Press, San Francisco, CA, USA, 2013, pp. 409–418.