Semi-supervised Software Defect Prediction Model Based on Tri-training

Fanqi Meng[1,2], Wenying Cheng[1*], Jingdong Wang[1*]
[1]School of Computer Science, Northeast Electric Power University,
Jilin 132012, China
[e-mail: 2202000697@neepu.edu.cn, wangjingdong@neepu.edu.cn]
[2]Guangdong Atv Academy For Performing Arts,
Guangdong 523710, China
[e-mail: mengfanqi@neepu.edu.cn]
[*]Corresponding author: Wenying Cheng, Jingdong Wang

Received September 7, 2021; accepted October 24, 2021;
published November 30, 2021

Abstract

Aiming at the problem of software defect prediction difficulty caused by insufficient software defect marker samples and unbalanced classification, a semi-supervised software defect prediction model based on a tri-training algorithm was proposed by combining feature normalization, over-sampling technology, and a Tri-training algorithm. First, the feature normalization method is used to smooth the feature data to eliminate the influence of too large or too small feature values on the model's classification performance. Secondly, the oversampling method is used to expand and sample the data, which solves the unbalanced classification of labelled samples. Finally, the Tri-training algorithm performs machine learning on the training samples and establishes a defect prediction model. The novelty of this model is that it can effectively combine feature normalization, oversampling techniques, and the Tri-training algorithm to solve both the under-labelled sample and class imbalance problems. Simulation experiments using the NASA software defect prediction dataset show that the proposed method outperforms four existing supervised and semi-supervised learning in terms of Precision, Recall, and F-Measure values.

Keywords: Feature Normalization, Oversampling Techniques, Software Defect Prediction, Semi-supervised Learning, Unbalanced Classification

This work is supported by the Science and Technology Research Project of Jilin Provincial Department of Education "Research on Overtime Risk Assessment and Early Warning Technology of Industrial Control Code" (No. JJKH20210097KJ).

http://doi.org/10.3837/tiis.2021.11.009 **ISSN : 1976-7277**

1. Introduction

Software defects are the antithesis of software quality and threaten it [1]. As software development progresses, the size and structure of programs become larger and more complex, making defects hide deeper and more challenging to detect. Undetected defects can lead to more server problems as the software iterates, resulting in more server consequences when defects break out. Research and practice have long shown that the earlier software defects are found, the lower the cost of fixing them, and the more damage can be recovered [2]. The cost of finding and fixing defects in the early coding phase of software development is approximately one to two orders of magnitude lower than in the later testing or release phase of software development. As a result, academia and industry are interested in finding software defects early and fixing them at a lower cost. In this context, the accurate prediction of defective modules at the early stages of software development has become a critical technical problem that needs to be addressed.

Generally, software defect prediction is divided into dynamic defect prediction and static defect prediction. Dynamic defect prediction generally requires a running program. However, in the early coding stages of software development, dynamic defect prediction is not applicable in the early stages of software development as the code is not yet ready to run. Static defect prediction can predict defective modules in a software system without running the program, based on defect-related metrics, using methods such as machine learning [3]. Therefore, static defect prediction is ideal for identifying defects in a program at the early stages of software development.

Standard machine learning methods for static defect prediction include logistic regression, decision trees, Bayesian methods, artificial neural networks, and support vector machines [4]. These methods require learning a large number of marker samples in order to build a defect prediction model. However, marker samples need to be created by manually reviewing the code, which is time-consuming. Furthermore, it is almost impossible to obtain many marker samples from a new project without a historical version when the amount of code in the early stages of development is already tiny. As a result, defect prediction in the early stages of software development often faces insufficient marker samples.

In order to solve the above problems, academics have started to experiment with semi-supervised methods to predict software defects. Compared to supervised learning, semi-supervised learning can make full use of unlabeled samples to achieve defect prediction and better classification results with only a small number of labelled samples [5]. Furthermore, in software testing, 80% of defects are found in 20% of the code, meaning that most software defects are concentrated in a few software modules. Therefore, software defect history data is characterized by significant "class imbalance"[6], leading to poor learning and inaccurate prediction.

In order to simultaneously solve the problems of insufficient labelled samples and class imbalance, this paper proposes a semi-supervised software defect prediction model based on Tri-training (Tri_SSDPM), which firstly uses feature normalization to smooth the feature data. The first step is to smoothen the feature data by using the feature normalization method to eliminate the impact of too large or too small feature values on the model's classification performance. Secondly, the SOMTE sampling method expands and samples the data to alleviate the unbalanced classification of labelled samples. Finally, the labelled and unlabeled training sets are randomly selected by setting the index, and the labelled and unlabeled training sets are input to the Tri-Training algorithm for machine learning and building the defect prediction model. It not only solves the class imbalance problem of labelled samples but also

makes full use of unlabeled samples to improve the prediction performance of the classifier.

The rest of the paper is organized as follows: Section 2 reviews related work; Section 3 describes a semi-supervised software prediction model based on Tri-training, including the general framework, data pre-processing and the Tri-training algorithm; Section 4 tests the validity of the model by describing the experimental procedure; Section 5 summarizes the work of the paper and describes the focus of the next steps.

2. Related Work

From recent research in the field of static defect prediction, it is easy to see that supervised machine learning methods rely on a large amount of learning data, and the prediction accuracy of such methods is low when there is not enough learning data. In practical applications, unlabeled data is readily available, but labelled data is challenging to obtain. Researchers have noticed semi-supervised learning in this context and gradually applied it to software defect prediction [4].

In recent years, different types of semi-supervised learning techniques have been applied to the field of software defects [7]. In general, semi-supervised methods for defect prediction can usually be classified into the following categories: expectation maximization [9], constraint-based semi-supervised clustering [10], naive bayesian algorithm [11], labelled propagation [12], sample-based approaches [13-17] and preprocessing strategy [18-20]. These studies have shown the practical value of semi-supervised methods in defect prediction [8].

Seliya et al. [9] proposed a semi-supervised model based on the EM algorithm to achieve good prediction performance on the NASA public dataset. The model applied the EM algorithm to label the unlabeled data in the training set, thus solving difficult access to labelled data. Seliya et al. [10] also proposed a semi-supervised clustering method, a constraint-based semi-supervised clustering method, using k-means as the base clustering algorithm. Catal et al. [11] studied a semi-supervised defect prediction model using the naive bayesian algorithm. Their results show that the naive bayesian algorithm is the best choice for building semi-supervised defect prediction models for small-scale datasets, and the proposed two-stage YATSI method can improve the performance of plain Bayesian on large-scale datasets. Zhang et al. [12] proposed a label propagation method based on non-negative sparse graphs, which uses a small amount of labelled data and a large amount of unlabeled data to improve generalization ability.

Sample-based approaches [13-17] and preprocessing strategy [18-20] are widely used in software defects prediction. Li et al. [13] used an active sampling method to find samples that are prone to misclassification in the training samples and then manually labelled them, solving the problem that these samples are mislabeled during semi-supervised learning reduced model performance. This method achieved better prediction performance than traditional machine learning methods on the PROMISE dataset. Lu et al. [14] proposed an iterative semi-supervised method FTF, which first used the model to set labels for all unlabeled instances to ensure that all instances in the sample were labelled and then trained the model on the whole dataset. The results show that FTF has relatively apparent advantages over traditional supervised methods. Abadi et al. [15] proposed an automated software defect prediction model based on the semi-supervised hybrid self-organizing mapping. The model is a semi-supervised model based on self-organizing mapping and artificial neural networks, which can predict the labels of modules in a semi-supervised manner using software measurement thresholds in the presence of insufficient label data. Experiments have shown that the model has good prediction results. Jiang et al. [16] proposed a semi-supervised software defect prediction method

ROCUS. The method utilizes a large number of unlabeled examples and solves the class imbalance problem. Thung et al. [17] proposed a semi-supervised defect prediction method. The method selects a fraction of different information-rich defect examples to be labelled, and it can improve model performance in the presence of insufficient labelled samples. Lu et al. [18] proposed a semi-supervised software defect prediction model. A dimensionality reduction method was incorporated in the method to reduce the dimensional complexity of the software metric. Experimental results showed that the semi-supervised learning algorithm with dimensionality reduction preprocessing outperformed random forests, one of the best performing supervised learning algorithms, in the few cases where labelled samples were available for training. They [19] also tried to incorporate a feature scaling method and obtained better prediction results. Ma et al. [20] improved the Tri-training algorithm by using a random under-sampling method combined with Tri-training, which effectively reduced the impact of class imbalance and the insufficient number of labelled samples on the model prediction.

Most of the above methods only address one problem in software defect prediction, i.e., they only study the case of low marker samples or unbalanced class distribution, and there are relatively few studies that consider both cases together. Furthermore, relatively little research combines feature normalization, SMOTE sampling methods, and Tri-training algorithms in software defect prediction. The SMOTE sampling method can achieve better results in solving class imbalance and the insufficient number of training samples, while the Tri-training method can effectively use unlabeled samples to improve the prediction performance of the classifier. Therefore, this paper innovatively and effectively combines the three methods to solve insufficient labelled samples and class imbalance simultaneously.

3. Model construction

3.1 Overall Framework

The overall architecture of the model in this paper is divided into two main parts: the training model phase and the prediction phase. As shown in **Fig. 1**, the feature data is first normalized and compressed to a specific interval in the training phase.

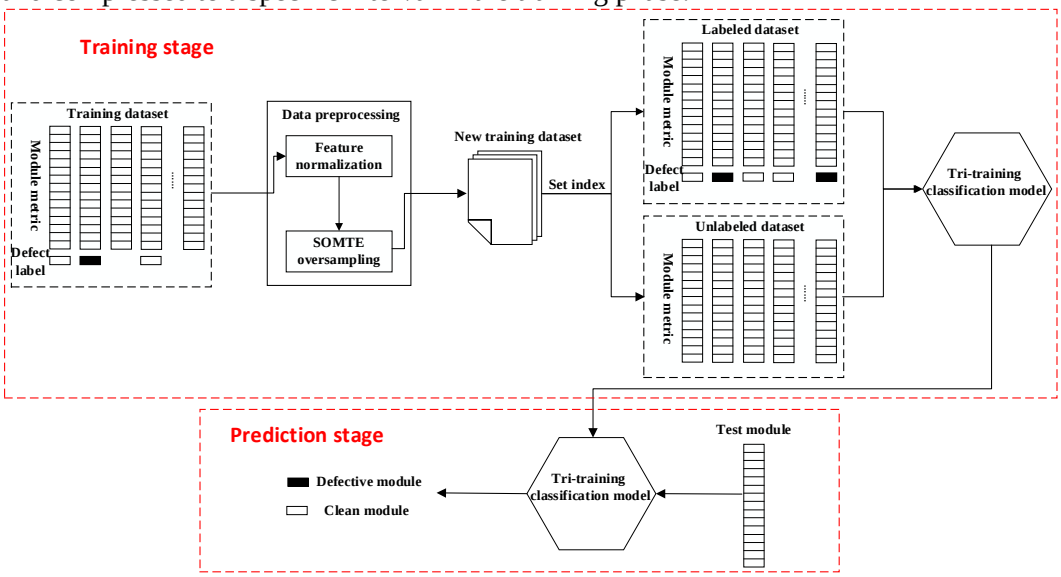

Fig. 1. General architecture diagram for semi-supervised software defect prediction based on Tri-Training

The data is expanded and sampled using the SOMTE sampling method to generate a new training data set, and then labelled and unlabeled training sets are randomly selected by proportionally setting the index and the labelled and unlabeled training sets are input to the Tri-training algorithm for learning. In the prediction phase, the test module is input to the trained classifier to predict whether the module has defects.

3.2 Data preprocessing

3.2.1 Feature normalization

The software metrics commonly used for static software defect prediction are shown in **Table 1**. 21 feature metrics extracted from the project's source code, generated by metrics such as McCabe [21], Halstead [22], which objectively characterize the quality features associated with software quality. These 21 feature metrics are used as independent variables in this experiment, and the dependent variable is a binary variable (0 or 1) to indicate whether the code is defective or non-defective. As some feature values are too large or too small, they can affect the classification results of the final model. The data with a relatively significant skewness can first be transformed using the $log1p$ function to compress the feature data to a specific interval, making it more obedient to the Gaussian distribution while avoiding the problem of complex values, which may lead to a good result for our subsequent classification results.

Table 1. Feature metrics

Feature	Description
McCabe's line count of code	It counts the lines of code in module.
McCabe "cyclomatic complexity"	It indicates complexity of the module on basis of number of linearly independent paths.
McCabe "essential complexity"	It indicates the extent to which a flowgraph can be reduced.
McCabe "design complexity"	It indicates cyclomatic complexity of its reduced flowgraph.
Halstead total operators +operands	It gives the count of operators and operands used in the module.
Halstead "volume"	It measures the product of length and log of vocabulary on base.
Halstead "program length"	It indicates the length of the program.
Halstead "difficulty"	It is related to the difficulty of the program to write or understand. Also computed as reciprocal of length.
Halstead "intelligence"	It determines amount of intelligence presented in the module.
Halstead "effort"	It translates into actual coding time.
Halstead	It is a base Halstead measure.
Halstead's time estimator	It evaluates the testing time of C/C++codes.
Halstead's line count	It indicates the numbers of lines in the code.
Halstead's count of lines of comments	It indicates the number of lines of comments.
Halstead's count of blank	It indicates the number of lines of comments.
Lines of code and comments	It gives the lines of code and comment in the module.
Unique operators	It counts the total number of distinct operators in the module.

Unique operands	It counts the total number of operators in the module.
Total operators	It counts the total number of operands in the module.
Branch count of the flow graph	It gives the count of branches in the flow graph.

3.2.2 SOMTE oversampling

In practice, there are usually fewer defective instances than non-defective ones, known as the class imbalance problem in software defect prediction [23]. If a random division of the dataset or an under-sampling preprocessing approach is used directly, the training dataset will likely contain very little or even no defective data, and it is not easy to train a better prediction model using such data the training set. The basic idea is to generate more samples with fewer labels according to the pattern of samples with fewer labels, thus making the data more balanced and solving insufficient initial samples. A typical oversampling type is the SMOTE (Synthetic Minority Oversampling Technique) algorithm proposed by Chawal [24]. The steps of this algorithm are as follows.

(1) For each sample x in the minority class, calculate its Euclidean distance to all samples in the minority class sample set S_{sin}, To obtain its k-nearest neighbours.

(2) Set a sampling ratio according to the sample imbalance ratio to determine the sampling multiplier N. For each minority class sample x, select some samples at random from its k nearest neighbours, assuming the selected nearest neighbours are x_n.

(3) For each randomly selected nearest neighbor x_n, construct a new sample with the original sample respectively according to the following formula.

$$x_{new} = x + rand(0,1) * |x_n - x| \tag{1}$$

3.2.3 Tri-training

Zhou and Li [25] proposed the Tri-training algorithm to improve the co-training algorithm Co-training proposed by A. Blum and T. Mitchell [26]. The Co-training algorithm requires two fully redundant dataset views, but this is not easy to achieve in practice. The Tri training algorithm does not require fully redundant views and different supervised learning algorithms, which is more widely applicable. The basic idea is to first repeatably sample the labelled example set to obtain three labelled training sets, generate one classifier from each training set, and then use these three classifiers to generate pseudo-labelled samples in a 'majority rule' fashion. For example, if two classifiers predict an unlabeled sample to be defective and a third classifier predicts it to be non-defective, that sample is provided to the third classifier for learning as a defective sample. Specifically, if both classifiers predict the same unlabeled sample equally, the sample is considered to have high confidence in the labelling and is added to the labelled training set of the third classifier after labelling. After the final training is completed, the three classifiers are used as a single classifier integration through a voting mechanism. Thus, Tri-training uses both semi-supervised and integrated learning mechanisms, resulting in a further improvement in learning performance. **Algorithm 1** shows a simplified pseudo-code of the Tri-training method, and more details can be found in the literature [25].

Algorithm 1: Pseudo code of simplified Tri-training.

Input: Training set $L = \{x_i, y_i\}_{i=1}^{l}, U = \{x_i\}_{i=1}^{u}$
Three classifiers h_1, h_2, h_3
Output: Ensemble h using majority vote;

```
for i = 1,2,3 do
    Train h_i on L;
end
while any of {h_1, h_2, h_3} changes do
    for i = 1,2,3 do
        L_i = ∅ ;
        for x ∈ U do
            if h_j(x) = h_k(x)(j, k ≠ i) then
                L_i = L_i ∪ (x, h_j(x));
            end
        end
    end
    for i = 1,2,3 do
        Train h_i on L ∪ L_i;
    end
end
```

The Tri-training algorithm labels unlabeled samples by first labelling any unlabeled sample $x \in U$ by any two of the classifiers h_1 and h_2. If both classifiers have the same labelling result for x, one of the labelling results is chosen as the training sample for the classifier h_3, i.e.: $h_1(x) = h_2(x)$, then $L_i = \{(x, h_2(x))\}, x \in U$ is added to the training set of h_3.

The training result will be optimized if h_3 can achieve high accuracy during the training process; on the contrary, it will introduce noise into the training set of h_3 and reduce the labeling performance of the classifier. Based on this, Zhou et al. [25] made the following proof.

$$|L \cup L^t|\left(1 - 2\frac{\eta_L|L| + \widetilde{e_1}^t|L^t|}{|L \cup L^t|}\right)^2 > |L \cup L^t|\left(1 - 2\frac{\eta_L|L| + \widetilde{e_1}^{t-1}|L^{t-1}|}{|L \cup L^{t-1}|}\right)^2 \tag{2}$$

where L^t: the newly labelled training samples of h_1, h_2 for h_3 at the tth iteration. $\widetilde{e_1}^t$: the ratio of the number of incorrectly labelled samples in h_1, h_2 labelled samples at the tth iteration. η_L: the noise rate of the initial training set L.

In the PAC learnable framework, if the newly labelled training samples are large enough and can satisfy the conditions in equation (2), the classification performance of the hypothesis will improve when h_3 is trained again.

The classification noise rate after the tth iteration can be expressed by the following equation.

$$\eta^t = \frac{\eta|L| + \widetilde{e_1}^t|L^t|}{|L \cup L^t|} \tag{3}$$

When $0 < \widetilde{e_1}^t, \widetilde{e_1}^{t-1} < 0.5$ and $|L^t| > |L^{t-1}|$, it follows that:

$$\widetilde{e_1}^t|L^t| < \widetilde{e_1}^{t-1}|L^{t-1}| \tag{4}$$

Thus, equation (2) can be transformed as follows.

$$0 < \frac{\widetilde{e_1}^t}{\widetilde{e_1}^{t-1}} < \frac{|L^t - 1|}{L^t} \tag{5}$$

In the process of labelling unlabeled samples, equation (5) can be used to determine whether the samples $\{(x, h_2(x))\}$ labelled by classifiers h_1 and h_2 can be added to the classifier h_3 as training samples for a new round.

4. Experiment and Analysis

4.1 Experimental Data

The NASA MDP defect dataset from the PROMISE library [27], which develops software defect datasets, was selected for experimentation in this paper. NASA implements these datasets in C or java. They contain descriptions of the modules in the dataset with or without defective annotations and descriptions of the dataset's attributes generated by McCabe [21], Halstead [22], and other metrics. McCabe, Halstead can objectively characterize the quality features associated with software quality. **Table 2** gives basic information about the five sub-datasets selected, including the dataset name, system, the total number of modules, number of defective modules and defect rate.

Table 2. NASA Defect Prediction Dataset

Datasets	System	Module number	Defects	Defects rate /%
CM1	Spacecraft instrument	498	49	9.8
PC1	Flight software	1109	77	6.9
KC1	Storage management	2109	326	15.5
KC2	Scientific data processing	522	107	20.5

4.2 Experimental Evaluation Index

Software defect propensity prediction is whether a software module is a defective or clean module, and it is a binary classification problem. Therefore, model evaluation metrics for classification problems in machine learning can be used, and the confusion matrix is a presentation tool to evaluate how good a classification model is [1]. As shown in **Fig. 2**, TP indicates that the true class of the instance is a defective module and the prediction result of the prediction model is also a defective module. FN indicates that the true class of the instance is a defective module and the prediction result of the prediction model is a defect-free module. FP indicates that the true class of the instance is a defect-free module and the prediction result of the prediction model is a defective module. TN indicates that the true class of the instance is a defect-free module and the prediction result of the prediction model is a defect-free module.

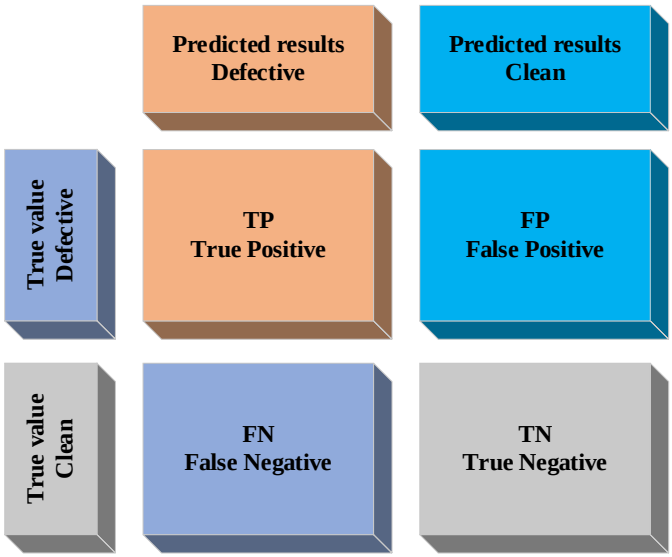

Fig. 2. Confusion matrix

Other evaluation metrics can be calculated based on the confusion matrix, such as accuracy, recall and accuracy, and the overall evaluation metric F-value.

Accuracy represents the ratio of the number of correct predictions by the prediction model to the total number of instances in the data set. The following formula calculates it:

$$Accuracy = \frac{TP + TN}{TP + FN + FP + TN} \tag{6}$$

In general, the higher the Precision, the better the prediction model is. The following formula calculates it:

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

Recall indicates the number of instances predicted by the prediction model to be defective as a proportion of true defective instances. In general, a higher Recall indicates that the model correctly predicts more defective modules and that the prediction model is more effective, but a higher Recall is often coded for reduced accuracy. The following formula calculates it:

$$Recall = \frac{TP}{TP + FN} \tag{8}$$

The F-measure is a composite metric that provides a trade-off between recall and precision, with higher F-measure values indicating better model performance. The following formula calculates it:

$$F = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{9}$$

4.3 Analysis of Results

In this paper, two classical learning algorithms, Decision Tree and NaiveBayes, an Adaboost integrated learning algorithm and the semi-supervised algorithm S4VM+ proposed in the literature [4] were selected for analysis of the NASA MDP dataset and compared with the Tri_SSDPM proposed in this paper to calculate the prediction accuracy, recall, accuracy and overall evaluation index F. The default values in sklearn were used for the running parameters of Decision Tree, NaiveBayes and Adaboost learning algorithms.

This experiment was set up as follows, with each expanded dataset randomly divided into a training set and a test set according to a 4:1 ratio. Then the training set was further randomly divided into a labelled training set and an unlabeled training set according to a specific labelling rate R. The larger the R, the more labelled modules in the training set. In this experiment, three different R values were used, and R was taken to be 0.2, 0.3 and 0.4, respectively. In order to eliminate chance from the experimental results, the experiment was repeated 20 times for each tagging rate R for the method in this paper and the chosen comparison method, and the average of the 20 times was taken as the final result of the experiment.

Table 4. Experimental results of each classifier on the CM1 dataset with different sample labelling rates

Classifier	Accuracy			Precision			Recall			F-Measure		
	R=0.2	R=0.3	R=0.4	R=0.2	R=0.3	R=0.4	R=0.2	R=0.3	R=0.4	R=0.2	R=0.3	R=0.4
DecisionTree	0.835	0.838	0.845	0.190	0.196	0.164	0.182	0.228	0.178	0.174	0.199	0.162
NaiveBayes	0.705	0.696	0.710	0.505	0.582	0.591	0.174	0.207	0.208	0.250	0.268	0.287
Adaboost	0.854	0.874	0.864	0.103	0.187	0.168	0.134	0.246	0.214	0.107	0.199	0.177
S4VM+	0.879	0.880	0.878	0.261	0.338	0.298	0.142	0.146	0.150	0.177	0.191	0.195
Tri_SSDPM	0.850	0.867	0.885	0.865	0.896	0.916	0.775	0.818	0.830	0.816	0.854	0.870

As shown in **Table 4**, Tri_SSDPM does not always obtain optimal values compared to the other four methods, but its Accuracy value differs from the other methods by a maximum of 0.069 (0.879-0.810), which still gives comparable results to the comparison methods. Since the proposed method obtained the best values for Precision, Recall and F-Measure for all three marking rates. Therefore, overall, our proposed method outperforms the other four methods in terms of prediction performance.

Table 5. Experimental results of each classifier on the PC1 dataset with different sample labelling rates

Classifier	Accuracy			Precision			Recall			F-Measure		
	R=0.2	R=0.3	R=0.4	R=0.2	R=0.3	R=0.4	R=0.2	R=0.3	R=0.4	R=0.2	R=0.3	R=0.4
DecisionTree	0.888	0.893	0.901	0.317	0.314	0.327	0.254	0.243	0.301	0.271	0.265	0.304
NaiveBayes	0.761	0.772	0.775	0.496	0.497	0.483	0.138	0.167	0.149	0.213	0.245	0.225
Adaboost	0.910	0.915	0.925	0.198	0.238	0.238	0.331	0.340	0.429	0.236	0.266	0.285
S4VM+	0.926	0.930	0.929	0.435	0.489	0.477	0.146	0.155	0.149	0.207	0.230	0.218
Tri_SSDPM	0.935	0.943	0.939	0.937	0.941	0.959	0.869	0.889	0.915	0.901	0.914	0.936

As shown in **Table 5**, Tri_SSDPM in this paper obtained optimal values at all three labelling rates compared to the other four methods.

Table 6. Experimental results of each classifier on the KC1 dataset with different sample labelling rates

Classifier	Accuracy			Precision			Recall			F-Measure		
	R=0.2	R=0.3	R=0.4	R=0.2	R=0.3	R=0.4	R=0.2	R=0.3	R=0.4	R=0.2	R=0.3	R=0.4
DecisionTree	0.796	0.809	0.810	0.335	0.345	0.351	0.344	0.363	0.376	0.337	0.352	0.361
NaiveBayes	0.743	0.745	0.737	0.643	0.659	0.665	0.324	0.331	0.328	0.430	0.439	0.438
Adaboost	0.831	0.842	0.846	0.268	0.241	0.231	0.452	0.509	0.517	0.333	0.321	0.317
S4VM+	0.842	0.850	0.851	0.494	0.478	0.526	0.284	0.295	0.290	0.357	0.361	0.371
Tri_SSDPM	0.822	0.833	0.847	0.824	0.861	0.872	0.773	0.814	0.831	0.797	0.836	0.851

As shown in **Table 6**, Tri_SSDPM does not always obtain the optimal value in Accuracy, but its F-Measure values are higher than those of the comparison methods. Therefore, our proposed method still outperforms the other four methods in terms of prediction performance.

Table 7. Experimental results of each Classifier on the KC2 dataset with different sample labelling rates

Classifier	Accuracy			Precision			Recall			F-Measure		
	R=0.2	R=0.3	R=0.4	R=0.2	R=0.3	R=0.4	R=0.2	R=0.3	R=0.4	R=0.2	R=0.3	R=0.4
DecisionTree	0.797	0.790	0.784	0.516	0.416	0.451	0.503	0.488	0.499	0.499	0.443	0.466
NaiveBayes	0.770	0.784	0.778	0.748	0.786	0.796	0.476	0.476	0.468	0.577	0.590	0.584
Adaboost	0.787	0.801	0.810	0.433	0.422	0.501	0.476	0.493	0.558	0.445	0.452	0.512
S4VM+	0.820	0.815	0.815	0.562	0.546	0.576	0.438	0.434	0.433	0.487	0.486	0.490
Tri_SSDPM	0.854	0.850	0.861	0.817	0.874	0.875	0.797	0.802	0.823	0.803	0.835	0.848

As shown in **Table 7**, Tri_SSDPM obtained optimal values at all three labelling rates.

The data in **Tables 4** to **Tables 7** were counted, and the Tri-training-based semi-supervised software defect prediction model obtained optimal values 43 times out of a total of 48 evaluations performed at the three labelling rates, accounting for 89.58%. At the same time, the F-Measure values of the supervised learning algorithm and the unsupervised learning algorithm increased as the R-value increased, indicating that the prediction effect subsequently became better when there were more and more labelled modules in the training set.

Table 8. Average of each classifier under the four metrics

Classifier	Accuracy	Precision	Recall	F-Measure
DecisionTree	0.832	0.327	0.330	0.320
NaiveBayes	0.748	0.621	0.287	0.379
Adaboost	0.855	0.269	0.391	0.304
S4VM+	0.866	0.457	0.340	0.314
Tri_SSDPM	**0.874**	**0.886**	**0.828**	**0.855**

The results of each classifier under the four metrics were averaged, as shown in **Table 8**. In order to visualize the classification effect of each classification model, the average values of Accuracy, Precision, Recall and F-Measure for each classification model under the three labelling rates are represented in a bar chart. As can be seen from **Fig. 3**, the proposed method achieved the highest average values for Accuracy, Precision, Recall and F-Measure compared to the other methods. Also, combined with **Table 8**, it can be concluded that each metric improved by at least 0.008 (0.874-0.866), 0.265 (0.886-0.621), 0.437 (0.828-0.3910) and 0.476 (0.855-0.379) respectively.

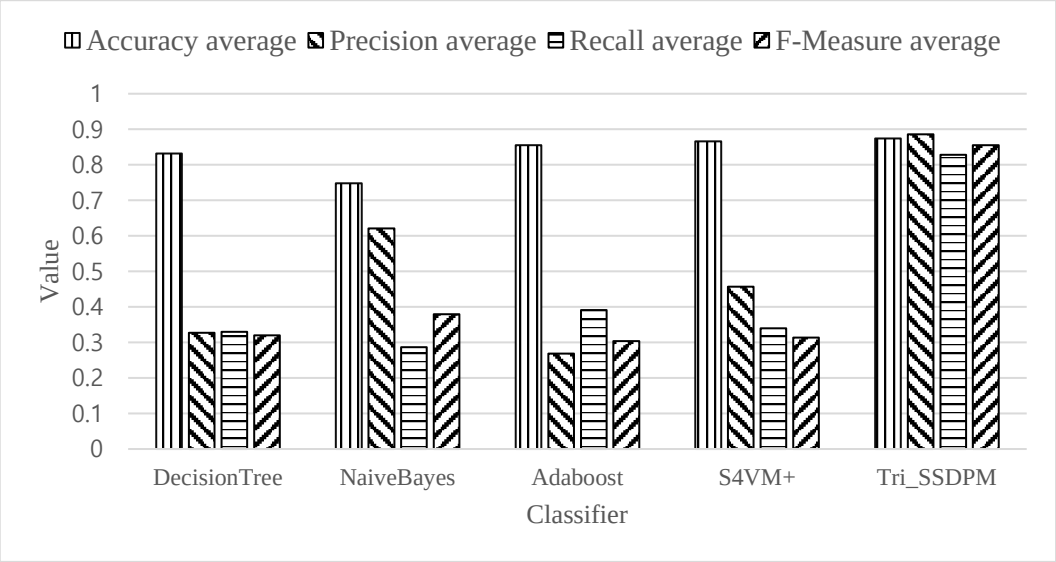

Fig. 3. Average of each classifier under the four metrics

The Tri_SSDPM model achieves higher accuracy than the four existing methods because the Decision Tree and NaiveBayes algorithms do not consider the lack of labelled samples and classification imbalance, resulting in poor classification results. S4VM+ semi-supervised model can make full use of unlabeled samples to improve the model's prediction performance, but it suffers from defective samples in the sampling stage. The number of defective samples is too low in proportion to the total number of samples, and it also does not consider the impact of too large and too small feature values on the model learning. However, the Tri_SSDPM model proposed in this paper ensures that the number of defective samples in the training samples is not too small, solves the problem of unbalanced classification of training samples, makes full use of the unlabeled data in the learning phase, improves the model's defect prediction performance, and also eliminates the influence of too large or too small feature values on the model learning, so that the optimal values of all indicators are obtained in the experimental results.

Combining the experimental results in **Tables 4** to **Tables 8**, it can be concluded that feature normalization, expansion of training samples, and the use of unlabeled samples can improve the prediction model's performance to a certain extent. Our proposed method is an effective semi-supervised software defect prediction model with better learning capability than several other methods.

5. Conclusion

This paper proposes a semi-supervised software defect prediction model based on Tri-training, which not only makes full use of a large number of unmarked samples in software defect prediction and solves the problem of insufficient marked samples but also uses the oversampling method to expand and sample the marked samples to solve the problem of unbalanced classification of marked samples. The problem of unbalanced classification of tagged samples is also solved. In addition, the feature metric values are normalized to eliminate the effect of too large or too small feature values on model learning. Experimental results show that the method achieves better prediction performance on the four NASA public datasets, outperforming DecisionTree, NaiveBayes, Adaboost and S4VM+ for different labelled

training samples.

Considering that the larger the dimensionality of the sample features, the worse the training effect of the model will be. However, academics have continued to increase the number of feature metrics to accurately represent software defects, resulting in sample features of excessive dimensionality. Therefore, the focus of the following work is to explore the impact of each feature metric on software defect prediction, select metric elements with high correlation with software defects, and thus reduce the feature dimensionality to improve the effectiveness of software defect prediction.

References

[1] L. N. Gong, S. J. Jiang and L. Jiang, "Research progress of software defect prediction," *Journal of Software*, vol. 30, no. 10, pp. 3090–3114, 2019. Article (CrossRef Link)

[2] L. Cai, Y. R. Fan, M. Yan and X. Xia, "Just-in-time software defect prediction: Literature review," *Journal of Software*, vol. 30, no. 5, pp. 1288−1307, 2019. Article (CrossRef Link)

[3] X. Chen, Q. Gu, W. S. Liu, S. L. Liu and C. Ni, "Survey of static software defect prediction," *Journal of Software*, vol. 27, no. 1, pp. 1−25, 2016. Article (CrossRef Link)

[4] S. P. Liao, L. Xu and M. Yan, "Software defect prediction using semi-supervised support vector machine with sampling," *Computer Engineering and Applications*, vol. 53, no. 14, pp. 161−166, 2017. Article (CrossRef Link)

[5] T. J. Wang, F. Wu and X. Y. Jing, "Semi-supervised Ensemble Learning Based Software Defect Prediction," *Pattern Recognition and Artificial Intelligence*, vol. 30, no. 7, pp. 646−652, 2017. Article (CrossRef Link)

[6] X. Zhang and L. M. Wang, "Semi-supervised Ensemble Learning Approach for Software Defect Prediction," *Journal of Chinese Computer Systems*, vol. 39, no. 10, pp. 2138−2145, 2018. Article (CrossRef Link)

[7] Z. W. Zhang, X. Y. Jing and F. Wu, "Twice Learning Based Semi-supervised Dictionary Learning for Software Defect Prediction," *Pattern Recognition and Artificial Intelligence*, vol. 30, no. 3, pp. 242−250, 2017. Article (CrossRef Link)

[8] W. W. Li, W. Z. Zhang, X. Y. Jia and Z. Q. Huang, "Effort-Aware Semi-Supervised Just-in-Time Defect Prediction," *Information and Software Technology*, vol. 126, 2020. Article (CrossRef Link)

[9] N. Seliya and T. M. Khoshgoftaar, "Software quality estimation with limited fault data: a semi-supervised learning perspective," *Software Quality Journal*, vol. 15, no. 3, pp. 327−344, 2007. Article (CrossRef Link)

[10] N. Seliya and T. M. Khoshgoftaar, "Software Quality Analysis of Unlabeled Program Modules With Semisupervised Clustering," *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, vol. 37, no. 2, pp. 201-211, March 2007. Article (CrossRef Link)

[11] C. Catal and B. Diri, "Unlabeled extra data do not always mean extra performance for semi-supervised fault prediction," *Expert Systems*, vol. 26, no. 5, pp. 458-471, 2009. Article (CrossRef Link)

[12] Z. W. Zhang, X. Y. Jing and T. J. Wang, "Label propagation-based semi-supervised learning for software defect prediction," *Automated Software Engineering*, vol. 24, no. 1, pp. 47-69, 2017. Article (CrossRef Link)

[13] M. Li, H. Y. Zhang, R. X. Wu and Z. H. Zhou, "Sample-based software defect prediction with active and semi-supervised learning," *Automated Software Engineering*, vol. 19, no. 2, pp. 201-230, 2012. Article (CrossRef Link)

[14] H. Lu, B. Cukic and M. Culp, "An iterative semi-supervised approach to software fault prediction," in *Proc. of the 7th International Conference on Predictive Models in Software Engineering*, pp. 1-10, 2011. Article (CrossRef Link)

[15] G. Abadi, A. Selamat and H. Fujita, "An empirical study based on semi-supervised hybrid self-organizing map for software fault prediction," *Knowledge-Based Systems*, vol. 74, pp. 28-39, 2015. Article (CrossRef Link)
[16] Y. Jiang, M. Li and Z. H. Zhou, "Software Defect Detection with Rocus," *Journal of Computer Science & Technology*, vol. 26, no. 2, pp. 328-342, 2011. Article (CrossRef Link)
[17] F. Thung, X. D. Le and D. Lo, "Active Semi-supervised Defect Categorization," in *Proc. of 2015 IEEE 23rd International Conference on Program Comprehension*, pp. 60-70, May 2015. Article (CrossRef Link)
[18] H. Lu, B. Cukic and M. Culp, "Software defect prediction using semi-supervised learning with dimension reduction," in *Proc. of 27th IEEE/ACM International Conference on Automated Software Engineering*, pp. 314-317, Sept 2012. Article (CrossRef Link)
[19] H. Lu, B. Cukic and M. Culp, "A Semi-supervised Approach to Software Defect Prediction," in *Proc. of 2014 IEEE 38th Annual Computer Software and Applications Conference*, pp. 416-425, July 2014. Article (CrossRef Link)
[20] Y. Ma, W. W. Pan, S. Z. Zhu, H. Y. Yin and J. Lou, "An improved semi-supervised learning method for software defect prediction," *Journal of Intelligent & Fuzzy Systems*, vol. 27, no. 5, pp. 2473-2480, Jan. 2014. Article (CrossRef Link)
[21] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308-320, Dec. 1976. Article (CrossRef Link)
[22] M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*, New York: Elsevierence, 1977.
[23] S. Feng, J. Keung, X. Yu, X. Yan and M. Zhang, "Investigation on the stability of SMOTE-based oversampling techniques in software defect prediction," *Information and Software Technology*, vol. 139, June 2021. Article (CrossRef Link)
[24] N. V. Chawla, K. W. Bowyer, L. O. Hall and W. P. Kegelmeyer, "SMOTE: Synthetic Minority Over-sampling Technique," *Journal of artificial intelligence research*, vol. 16, pp. 321-357, Jun. 2002. Article (CrossRef Link)
[25] Z. H. Zhou and M. Li, "Tri-training: exploiting unlabeled data using three classifiers," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 11, pp. 1529-1541, Nov. 2005. Article (CrossRef Link)
[26] A. Blum and T. Mitchell, "Combining labelled and unlabeled data with co-training," in *Proc. of Eleventh Conference on Computational Learning Theory*, pp. 92-100, July 1998. Article (CrossRef Link)
[27] M. Shepperd, Q. Song, Z. Sun and C. Mair, "Data Quality: Some Comments on the NASA Software Defect Datasets," *IEEE Transactions on Software Engineering*, vol. 39, no. 9, pp. 1208-1215, Sept. 2013. Article (CrossRef Link)

FANQI MENG received the B.E. degree in computer science and technology from Northwest Agriculture and Forest University, Yangling, in 2003 and the M.E. degree in computer application technology from Northeast Electric Power University, Jilin, in 2010 and the Ph.D. degree in computer application technology from Harbin Institute of Technology, Harbin, in 2018.He has been working at the Northeast Electric Power University since 2003. He is currently an Associate Professor in the School of Computer Science. His research interests include software safety, natural language processing, fault diagnosis of electric power equipment and other aspects, involve software engineering, artificial intelligence, data mining and other fields.

WENYING CHENG received the B.S. degree from Binzhou Medical College in 2020. He is currently pursuing a master's degree in the School of Computer Science, Northeastern Electric Power University. His main research interests are software defect prediction and software defect localization.

JINGDONG WANG received the B.E. degree and M.E. degree in computer science and technology from Northeast Electric Power University, Jilin and the Ph.D. degree in information science from University of Science and technology of China, in 2017.He has been working at the Northeast Electric Power University since 2008. From 2008 to 2011, he was a Teaching Assistant. From 2011 to 2016, he was a Lecturer. Since 2017, he has been an Associate Professor with the School of Computer Science. His research interests include public security, natural language processing, text mining, knowledge graph and other aspects, involve software engineering, artificial intelligence, emotional analysis and other fields.