Expert Systems with Applications 37 (2010) 4537–4543

Contents lists available at ScienceDirect

Expert Systems with Applications

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

Cost-sensitive boosting neural networks for software defect prediction

Jun Zheng [*]

Department of Computer Science and Engineering, New Mexico Institute of Mining and Technology, Socorro, NM 87801, United States

ARTICLE INFO

Keywords:
Software defect
Adaboost
Cost-sensitive
Neural networks

ABSTRACT

Software defect predictors which classify the software modules into defect-prone and not-defect-prone classes are effective tools to maintain the high quality of software products. The early prediction of defect-proneness of the modules can allow software developers to allocate the limited resources on those defect-prone modules such that high quality software can be produced on time and within budget. In the process of software defect prediction, the misclassification of defect-prone modules generally incurs much higher cost than the misclassification of not-defect-prone ones. Most of the previously developed predication models do not consider this cost issue. In this paper, three cost-sensitive boosting algorithms are studied to boost neural networks for software defect prediction. The first algorithm based on threshold-moving tries to move the classification threshold towards the not-fault-prone modules such that more fault-prone modules can be classified correctly. The other two weight-updating based algorithms incorporate the misclassification costs into the weight-update rule of boosting procedure such that the algorithms boost more weights on the samples associated with misclassified defect-prone modules. The performances of the three algorithms are evaluated by using four datasets from NASA projects in terms of a singular measure, the Normalized Expected Cost of Misclassification (NECM). The experimental results suggest that threshold-moving is the best choice to build cost-sensitive software defect prediction models with boosted neural networks among the three algorithms studied, especially for the datasets from projects developed by object-oriented language.

1. Introduction

As today's software grows in size and complexity, how to maintain the high quality of the product is one of the most important problems facing the software industry. Software defect predictors are tools to deal with this problem in a cost-effective way (Menzies, Greenwald, & Frank, 2007; Zhou & Leung, 2006). Previous studies have shown that the majority of defects of a software product are only found in a small portion of its modules (Boehm & Papaccio, 1988). Boehm indicated that approximately 20% modules of a software product are responsible for 80% of the error, costs, and rework, i.e. the "80:20" rule (Boehm, 1987). By measuring the defect-proneness of the software modules during the testing process and classifying them into defect-prone and not-defect-prone classes, software project managers can allocate the limited resources to test the defect-prone modules more intensively such that high quality software can be produced on time and within budget.

To predict the defect-proneness of software modules, software metrics are needed to provide the quantitative description of the program attributes. Many software metrics have been developed for this purpose and most of them are based on size and complexity. Lines of code (LOC) is a commonly used size metric for defect prediction (Akiyama, 1971) while McCabe (1976) and Halstead (1977) are the mostly used complexity metrics. Many works have been done to find the correlation of software metrics and defect-proneness by building different predictive models including discriminant analysis (Khoshgoftaar, Allen, Kalaichelvan, & Goel, 1996; Munson & Khoshgoftaar, 1992), logistic regression (Basili, Briand, & Melo, 1996; Gyimothy, Ferenc, & Siket, 2005; Zhou & Leung, 2006), factor analysis (Khoshgoftaar & Munson, 1990; Munson & Khoshgoftaar, 1990, 1992), fuzzy classification (Ebert, 1996), classification trees (Gokhale & Lyu, 1997; Gyimothy et al., 2005; Koru & Liu, 2005; Menzies et al., 2007), Bayesian network (Pai & Dugan, 2007; Zhou & Leung, 2006), artificial neural networks (ANN) (Gondra, 2008; Gyimothy et al., 2005; Kanmani, Uthariaraj, Sankaranarayanan, & Thambidurai, 2007; Khoshgoftaar, Lanning, & Pandya, 1994; Khoshgoftaar, Allen, Hudepohl, & Aud, 1997; Neumann, 2002; Quah & Thet Thwin, 2004) support vector machines (Gondra, 2008; Xing, Guo, & Lyu, 2005), etc. Since the relationship between software metrics and defect-proneness of software modules are often complicated and nonlinear, machine learning methods such as neural networks have been shown more adequate for the problem than traditional linear models (Khoshgoftaar et al., 1994, 1997). Our work is concentrated on applying neural networks for software defect prediction. Especially we investigate the ensemble of multiple neural network classifiers through

* Tel.: +1 575 835 6182.
E-mail address: junzheng@ieee.org.

0957-4174/$ - see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.eswa.2009.12.056

AdaBoost – an adaptive boosting algorithm (Freund, 1995; Freund & Schapire, 1997), which has shown to be an effective ensemble learning method to significantly improve the performance of neural network classifiers (Schwenk & Bengio, 2000).

During the software defect prediction process, two types of misclassification errors can be encountered. The type I misclassification happens when a not-fault-prone module is predicted as fault-prone one while a type II misclassification is that a fault-prone module is classified as not-fault-prone. A type I misclassification will result in the waste of time and resources to review a non-faulty module. A type II misclassification results in the missed opportunity to correct a faulty module that the faults may appear in the system testing or even in the field (Khoshgoftaar, Geleyn, Nguyen, & Bullard, 2002). It can be seen that the cost of a type II misclassification is much higher than that of a type I misclassification. Cost-sensitive learning has shown to be an effective technique for incorporating the different misclassification costs into the classification process (Elken, 2001; Viaene & Dedene, 2005). Several cost-sensitive boosting algorithms have been proposed by combing the cost factors in the boosting procedure to solve the imbalanced data problem (Fan, Stolfo, Zhang, & Chan, 1999; Sun, Kamel, Wong, & Wang, 2007; Ting, 2000). However, most of the existing works use the decision tree classification algorithm as the base classifier and none of them discusses cost-sensitive boosting neural networks. There are also only a few works in the literature that apply cost-sensitive boosting for software defect prediction. Khoshgotaar et al. (2002) built software quality models by using the cost-sensitive boosting algorithms where the C4.5 decision trees and decision stumps were used as the base classifiers. In this paper, we studied three cost-sensitive algorithms for boosting neural networks such that the misclassification costs of type I and II errors can be taken into account in building the software defect prediction models.

The rest of this paper has been organized as follows; in the next section, we briefly introduce the background of the neural network classifier and the *AdaBoost* algorithm. Section 3 describes the cost-sensitive algorithms for boosting neural networks to predict software defects. Section 4 introduces the software defect datasets used in this study and measurements used for assessing the classification performance. Section 5 shows the experimental results and the conclusions are drawn in Section 6.

2. Background

2.1. Neural networks

Neural networks have been used in many pattern recognition applications (Bishop, 1995). Among different neural network architectures, we adopt the back propagation neural network (BPNN) in this study which is the most frequently used architecture in the literature. The BPNN consists of a network of nodes arranged in layers. A typical BPNN consists of three or more layers of processing nodes: an input layer that receives external inputs, one or more hidden layers, and an output layer which produces the classification results. There is no computation involved in the input layer. When data are presented at the input layer, the network nodes perform calculations in the successive layers until an output value is obtained at each of the output nodes. The BPNN used in our study consists of three layers as shown in Fig. 1. The input layer has 21 nodes which correspond to the 21 software metrics extracted from a software module. The number of nodes in the hidden layer is set to 11 in our study. The output layer has one node to indicate the module is defect-prone or not, i.e. "-1" for defect-prone and "1" for not-defect-prone.

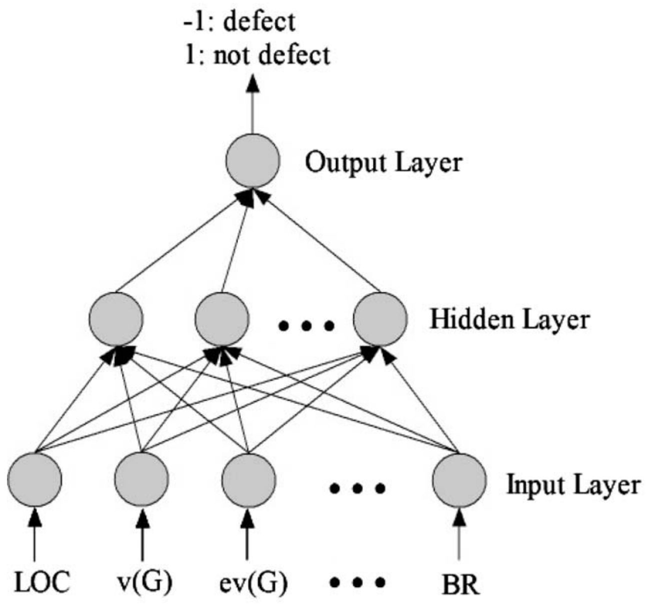

Fig. 1. Architecture of BPNN used in this study.

2.2. AdaBoost

Among different ensemble learning techniques, boosting has shown to be an effective way to produce diverse base classifiers for better classification accuracy (Freund, 1995; Freund & Schapire, 1997). AdaBoost, an adaptive boosting algorithm first introduced in 1995 by Freund (1995), is the most widely used boosting algorithm. AdaBoost constructs a composite classifier by sequentially training individual base classifiers. During the training process, the weights for the training examples are adjusted in the way that the weights of the misclassified examples are increased while the weights of the correctly classified examples are decreased in each training round. This kind of weight adjustment makes the learner to concentrate on different examples in each training round which leads to diverse classifiers. Finally, the constructed individual classifiers are combined to form the composite classifier by weighted or simple voting schemes.

For our two-class software defect prediction problem, the AdaBoost algorithm for boosting neural networks is shown in Fig. 1. Note that neural network can provide a posteriori probabilities of classes instead of a class label. Thus the AdaBoost algorithm shown in Fig. 2 combines the weak hypotheses by summing the probabilistic predictions instead of using a majority voting.

3. Cost-sensitive boosting neural networks

3.1. Cost-sensitive learning

The aim of cost-sensitive learning is to build a classifier that the different costs of the misclassification errors can be taken into account. For the two-class software defect prediction problem, the cost matrix has the structure shown in Table 1. In Table 1, $C(i,j)$ $(i,j \in \{-1,1\})$ denotes the cost of misclassifying an example of class i to class j. $c_{-1,1}$ and $c_{1,-1}$ denote the costs of false negative and false positive. In our case, $c_{-1,1}$ represents the cost of misclassifying a defect-prone software module to not-defect-prone while $c_{1,-1}$ is the cost of misclassifying a not-defect-prone one to defect-prone. The goal of cost-sensitive learning process is to take the cost matrix into consideration and generate a classification model with minimum misclassification cost.

```
Input: A training set T containing N examples ($(x_n, y_n)$), $n = 1, 2, \ldots, N$ where $x_n$ is a vector of attribute
    values and $y_n \in \mathbf{Y} = \{-1, 1\}$ is the class label.

Initialization: Let the weight vector: $W_1(n) = 1/N$ for $n = 1, 2, \ldots, N$, $y \in \mathbf{Y} - \{y_n\}$

Do for $t = 1, 2, \ldots, T$

1. Train neural network using the weight distribution $W_t$ and obtain the hypothesis $h_t \in [-1, 1]$.

2. Calculate the error of $h_t$:

        $\varepsilon_t = \sum_{n:h_t(x_n) \neq y_n} W_t(n)$        (4)

   If $\varepsilon_t > 0.5$, set $T = t - 1$ and abort loop.

3. Set the weight updating parameter $\alpha_t = \frac{1}{2}\log\left(\frac{1-\varepsilon_t}{\varepsilon_t}\right)$.

4. Update and normalized the weight vector

        $W_{t+1}(n) = \frac{W_t(n)\exp(-\alpha_t h_t(x_n) y_n)}{Z_t}$        (5)

   where $n = 1, 2, \ldots, N$, $Z_t$ is a normalization factor chosen so that $W_{t+1}$ becomes a proper distribution.

Output the final classifier:

        $h_f(x) = sign\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$        (6)
```

Fig. 2. Boosting neural networks with AdaBoost.

Table 1
Cost matrix for software defect prediction problem.

	Actual defect-prone	Actual not-defect-prone
Predict defect-prone	$C(-1,-1) = c_{-1,-1}$	$C(1,-1) = c_{1,-1}$
Predict not-defect-prone	$C(-1,1) = c_{-1,1}$	$C(1,1) = c_{1,1}$

There are several methods can be used to make a neural network classifier cost-sensitive including over-sampling, under-sampling, and threshold-moving (Zhou & Liu, 2006). Over-sampling and under-sampling incorporate the cost matrix into the learning by changing the training data distribution where the costs of the examples are conveyed by the appearance of training examples. Over-sampling increases the appearances of high-cost training examples while under-sampling decreases the number of inexpensive examples. Threshold-moving, in a different way, takes the cost matrix into account by moving the output threshold of neural network classifier such that high-cost examples are harder to be misclassified. It is shown in Zhou and Liu (2006) that the threshold-moving is a good choice to train cost-sensitive neural networks among the three methods.

3.2. Cost-sensitive boosting neural networks

AdaBoost provides an effective method to improve neural network classifiers. The direct way to make AdaBoost cost sensitive is to apply the threshold-moving in the final output stage of AdaBoost algorithm. Accordingly, the final hypothesis of the AdaBoost algorithm is modifies as:

$$h_f(x) = \arg\max_{y \in \mathbf{Y}} \sum_{t:h_t(x)=y} C_t \alpha_t h_t(x) \tag{7}$$

where $C_t = C(y, h_t(x))$ for $y \neq h_t(x)$. This modification is denoted as CSBNN-TM (Cost-Sensitive Boosting Neural Networks with Threshold-Moving). CSBNN-TM does not need to retrain the base neural network classifiers when the cost matrix changes.

Another way to make AdaBoost cost-sensitive is to introduce the cost matrix into the weight-updating process. Two modifications can be obtained as in Ting (2000) by changing the weight-update rule (Eq. (5) of Step 4) in the AdaBoost algorithm to:

$$\text{Modification 1}: W_{t+1}(n) = \frac{C_\delta W_t(n)\exp(-h_t(x_n)y_n)}{Z_t} \tag{8}$$

$$\text{Modification 2}: W_{t+1}(n) = \frac{C_\delta W_t(n)\exp(-\alpha_t h_t(x_n)y_n)}{Z_t} \tag{9}$$

where $C_\delta = 1$ if $y_n = h_t(x_n)$ and $C_\delta = C(y_n, h_t(x_n))$ for $y_n \neq h_t(x_n)$. It can be seen that this modification boosts more weights on the samples with higher misclassification cost such that the classification performance on those samples can be improved. We denote these two modifications as CSBNN-WU1 and CSBNN-WU2 (Cost-Sensitive Boosting Neural Networks with Weight-Updating). The difference between CSBNN-WU1 and CSBNN-WU2 is that CSBNN-WU1 doest not use the weight-updating parameter α_t in the formulation. Compared with CSBNN-TM, CSBNN-WU1 and CSBNN-WU2 requires retaining of all base neural network classifiers if the misclassification costs change.

4. Software defect data and performance measurements

4.1. Software defect datasets

Four software defect datasets, KC1, KC2, CM1 and PC1, used in this research are from four mission critical NASA projects that can be obtained freely from NASA IV & V Facility Metrics Data Program (MDP) data repository. The details about these four datasets are shown in Table 2.

For each module in the datasets, there are 21 associated software metrics including lines of code, McCabe, Halstead, and branch count metrics. Table 3 shows the descriptions for the 21 metrics. A module in the datasets is said to be defect-prone if there is one or more reported problems causing the change of the code.

4.2. Performance measurements

The prediction result obtained by any software defect prediction algorithm can be represented as the confusion matrix shown in Fig. 3.

To evaluate the performance of a defect prediction model, many prediction performance measures can be used. The most commonly used measure is the misclassification rate which is defined as the ratio of the number of wrongly classified modules to the total number of modules (Khoshgoftaar et al., 1997). The misclassification of the predication model can be further divided into types: Type I error and type II error as discussed in Section 1. From the confusion matrix, the misclassification rate (MR), type I error ($\mathrm{Err_I}$), and type II error ($\mathrm{Err_{II}}$) can be obtained as:

Table 2
Software defect datasets.

Dataset	Language	# Modules	% Defective	System
KC1	C++	2,109	15.5	Storage management
KC2	C++	522	20.5	Scientific data processing
CM1	C	496	9.7	NASA spacecraft instrument
PC1	C	1,107	6.9	Flight software

Table 3
Software metrics used in this study.

Metric	Description	Metric	Description
LOC	Line count of code	L	Halstead's length
v(G)	McCabe's cyclomatic complexity	I	Halstead's content
ev(G)	McCabe's essential complexity	E	Halstead's effort
iv(G)	McCabe's design complexity	B	Halstead's error estimate
N1	Total number of operators	T	Halstead's programming time
N2	Total number of operands	LOCb	Number of blank lines
µ1	Number of unique operators	LOCc	Number of comment-only lines
µ2	Number of unique operands	LOCe	Number of code-only lines
N	Halstead's length	LOCec	Number of lines with both code and comments
V	Halstead's volume	BR	Number of branches
D	Halstead's difficult		

		Actual	
		Defect	Not Defect
Predicted	Defect	TP	FP
	Not Defect	FN	TN

Fig. 3. Defect prediction confusion matrix, where TP is number of true positives, FP is number of false positives, TN is number of true negatives, and FN is number of false negatives.

$$\mathrm{MR} = \frac{FP + FN}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}} \tag{10}$$

$$\mathrm{Err_I} = \frac{\mathrm{FP}}{\mathrm{TN} + \mathrm{FP}} \tag{11}$$

$$\mathrm{Err_{II}} = \frac{\mathrm{FN}}{\mathrm{TP} + \mathrm{FN}} \tag{12}$$

As the costs for inspecting and correcting type I and type II errors are different, there is a need of a unified measure that can take into account the misclassification costs. In Kkoshgotaar and Seliya (2004), the expect cost of misclassification (ECM) (Johnson & Wichern, 1992) was used as a singular measure to compare the performances of different software quality classification models. The ECM measure is defined in Eq. (13) which includes both the prior probabilities of the two classes and the misclassification costs. Since it is not practical to obtain the individual misclassification costs in many organizations, the ECM measure is usually normalized with respect to C_I as shown in Eq. (14) such that the cost ratio can be used (Kkoshgotaar & Seliya, 2004).

$$\mathrm{ECM} = C_I \mathrm{Err_I} P_{\mathrm{ndf}} + C_{\mathrm{II}} \mathrm{Err_{II}} P_{\mathrm{df}} \tag{13}$$

$$\mathrm{NECM} = \mathrm{Err_I} P_{\mathrm{ndf}} + \frac{C_{\mathrm{II}}}{C_{\mathrm{I}}} \mathrm{Err_{II}} P_{\mathrm{df}} \tag{14}$$

In Eqs. (13) and (14), C_I and C_II are the costs for type I and type II errors which are equal to $c_{1,-1}$ and $c_{-1,1}$ in the cost matrix, respectively. P_{ndf} and P_{df} are the prior probabilities of the not-defect-prone and defect-prone modules in the dataset.

5. Experiments and results

To evaluate the performance of the three cost-sensitive neural network boosting algorithms, a fivefold cross-validation is used where each dataset is randomly divided into five equal sized subsets. Each time one subset is retained as the testing data while other four subsets are used as the training data. This process is then repeated five times (or fivefolds) such that each of the five subsets is used exactly once as the testing data. The final performance estimation is obtained from averaging of the results of the fivefolds. To ensure the low bias of the results, the cross-validation process is repeated for 20 times such that the partitioning of the dataset is different each time. For each performance measure, the mean is computed from the results of these 20 runs. The base BPNN used in the study has three layers with 11 hidden nodes. The iterations of boosting T which indicates the number of neural networks generated for the boosting ensemble is set as 10. Note that the architecture of the base NN and the parameter T are not optimized since the purpose of this study is to compare different cost-sensitive algorithms for boosting neural networks. Thus the relative performance is concerned instead of the absolute performance.

Figs. 4–7 show the prediction results of the three cost-sensitive boosting algorithms by using the four datasets, KC1, KC2, CM1 and PC1, respectively. We evaluate the prediction performance by varying the cost ratio $C_{\mathrm{II}}/C_{\mathrm{I}}$ from 1 to 10. From these plots, we have the following observations: (1) Among the three algorithms,

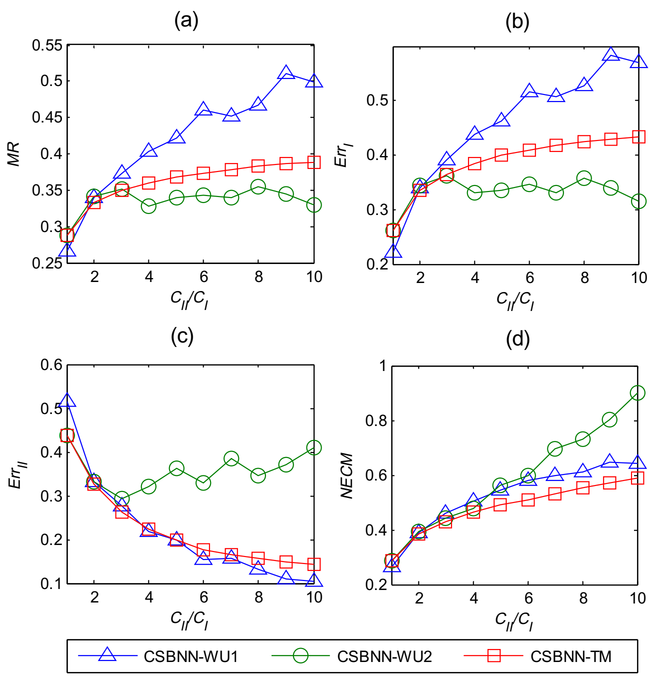

Fig. 4. Performance measurements of three cost-sensitive neural network boosting algorithms on KC1 dataset, (a) MR, (b) Err_I, (c) Err_II, and (d) NECM.

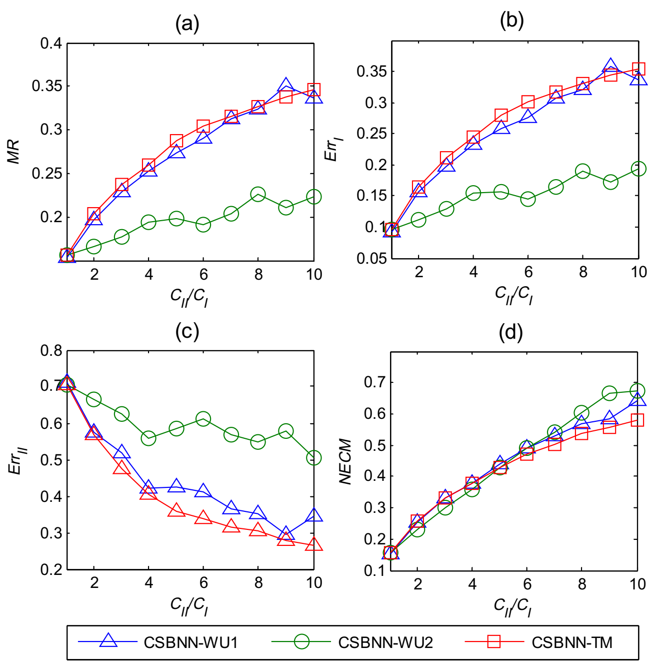

Fig. 6. Performance measurements of three cost-sensitive neural network boosting algorithms on CM1 dataset, (a) MR, (b) Err_I, (c) Err_II, and (d) NECM.

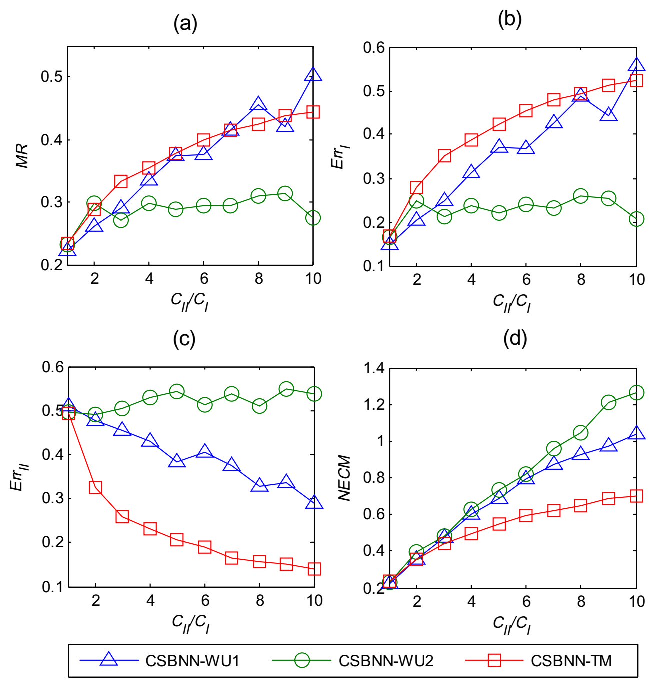

Fig. 5. Performance measurements of three cost-sensitive neural network boosting algorithms on KC2 dataset, (a) MR, (b) Err_I, (c) Err_II, and (d) NECM.

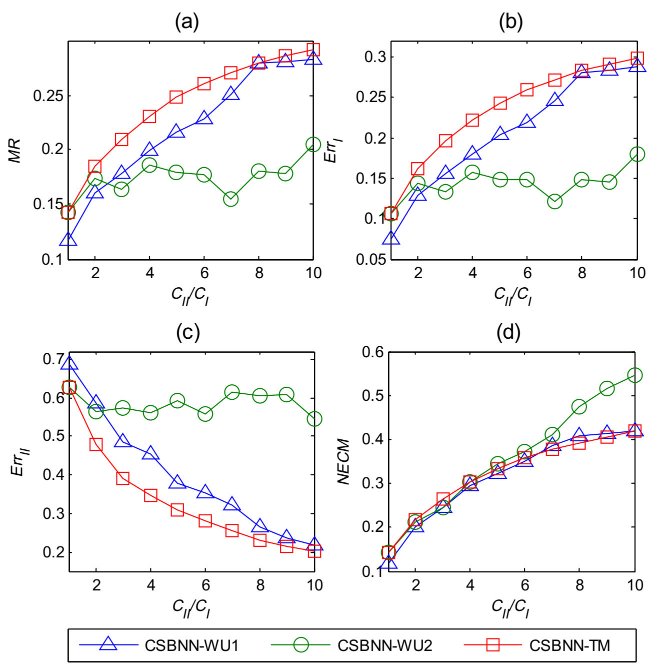

Fig. 7. Performance measurements of three cost-sensitive neural network boosting algorithms on PC1 dataset, (a) MR, (b) Err_I, (c) Err_II, and (d) NECM.

CSBNN-WU2 is the one least sensitive to the varying cost ratio. The type I and type II errors of CSBNN-WU2 are relatively flat when the cost ratio changes compared with other two cost-sensitive boosting algorithms. (2) For the two datasets (KC1 and KC2) from the projects developed by object-oriented languages (C++) where a module is a method, CSBNN-TM achieves significantly better performance than the two weight-updating based algorithms in terms of NECM although its MR is not the lowest. (3) For the two datasets (CM1 and PC1) from the projects developed by procedure language (C) where a module is a function, the performance of CSBNN-TM is slightly worse than that of CSBNN-WU2 in terms of NECM when the cost ratio is not greater than 5. When cost ratio is larger than 5, CSBNN-TM can obtain significantly lower cost than CSBNN-WU2. CSBNN-TM and CSBNN-WU1 achieve comparable performance for the two datasets except for the case that the cost ratio is larger than 5 and the dataset CM1 is used, where the cost obtained by CSBNN-TM is significantly lower than that of CSBNN-

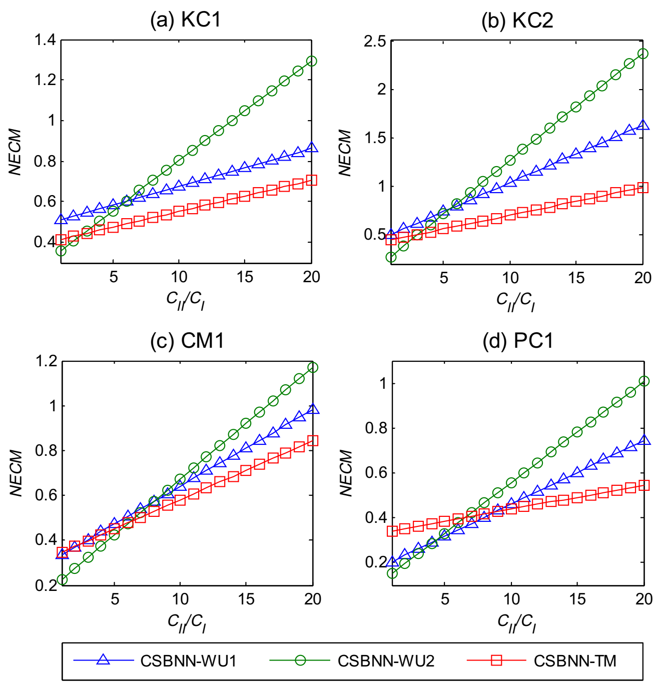

Fig. 8. NECMs of three prediction models built with cost ratio estimated as 10 versus actual cost ratio varying from 1 to 20 for the four datasets, (a) KC1, (b) KC2, (c) CM1, and (d) PC1.

WU2. (4) In most cases, CSB-WU2 achieves the lowest MR but highest NECM which shows that NECM is a performance measure more suitable for software defect prediction than MR as the misclassification costs are taken into account.

During the process of building the software defect prediction model, it is not easy to precisely estimate the cost ratio. Fig. 8 shows the effect of overestimating and underestimating the cost ratio on the performance of the prediction models. The prediction models are built by using three cost-sensitive neural network boosting algorithms with the cost ratio estimated as 10. The NECMs of the prediction models are then calculated with the actual cost ratio varying from 1 to 20. From Fig. 8, it can be observed that for the four datasets we use, the model built by CSBNN-TM always achieves the lowest cost among the three models when the cost ratio is underestimated, i.e. the actual cost ratio is larger than 10. If the cost ratio is overestimated (i.e. the actual cost ratio is less than 10), CSBNN-TM can still obtain good performance for the two datasets, KC1 and KC2, which are from the projects developed by object-oriented language. CSBNN-MU2 achieves the lowest cost when the actual cost ratio is much lower than estimated cost ratio (<3 for KC1 and KC2, and <5 for CM1 and PC1).

In summary, the experimental results suggest that threshold-moving is the best choice to build the cost-sensitive software defect predication models with boosted neural networks among the three algorithms studied, especially for the datasets from the projects developed by object-oriented language.

6. Conclusions

Software defect prediction models, which classify the software modules into two classes: defect-prone and not-defect-prone, play an important role in software development process. As the actual cost of misclassifying a defect-prone module is much higher than the misclassification cost of a not-defect-prone one, it is necessary to incorporate the misclassification costs into the software defect prediction models. In this paper, we build the prediction models by boosting neural networks and three cost-sensitive boosting algorithms are studied empirically on four datasets from NASA mission critical projects. A singular performance measure, NECM, is employed to evaluate the performance of different prediction models which is more suitable than the commonly used MR for software defect prediction. The empirical results indicate that the threshold-moving based algorithm achieves lower cost of misclassification and is more tolerant to the underestimation and overestimation of cost ratio compared with other two weigh-updating based algorithms. Another advantage of threshold-moving is that it is easier to implement as the base neural network classifiers do not need to be retrained when the misclassification costs change. Our study suggests that threshold-moving is a good choice to build cost-sensitive software defect prediction models with boosted neural networks.

References

Akiyama, F. (1971). An example of software system debugging. *Information Processing, 71*, 353–379.
Basili, V. R., Briand, L. C., & Melo, W. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering, 22*(10), 751–761.
Bishop, C. M. (1995). *Neural network for pattern recognition*. Oxford: Oxford University Press.
Boehm, B. W. (1987). Industrial software metrics top 10 list. *IEEE Software, 4*(5), 84–85.
Boehm, B. W., & Papaccio, P. N. (1988). Understanding and controlling software costs. *IEEE Transactions on Software Engineering, 14*(10), 1462–1477.
Ebert, C. (1996). Classification techniques for metric-based software development. *Software Quality Journal, 5*(4), 255–272.
Elken, C. (2001). The foundations of cost-sensitive learning. In *Proceedings of the seventh international joint conference on artificial intelligence (IJCAI'01)* (pp. 973–978).
Fan, W., Stolfo, S. J., Zhang, J., & Chan, P. K. (1999). Adacost: Misclassification cost-sensitive boosting. In *Proceedings of sixth international conference on machine learning (ICML'99)* (pp. 97–105).
Freund, Y. (1995). Boosting a weak learning algorithm by majority. *Information and Computation, 121*(2), 256–285.
Freund, Y., & Schapire, R. E. (1997). A decision theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences, 55*(1), 119–139.
Gondra, I. (2008). Applying machine learning to software fault-proneness prediction. *Journal of Systems and Software, 81*(2), 186–195.
Gokhale, S. S., & Lyu, M. R. (1997). Regression tree modeling for the prediction of software quality. *Proceedings of the third ISSAT international conference on reliability and quality in design* (pp. 31–36).
Gyimothy, T., Ferenc, R., & Siket, I. (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering, 31*(10), 897–910.
Halstead, M. (1977). *Elements of software science*. Elsevier.
Johnson, R. A., & Wichern, D. W. (1992). *Applied multivariate statistical analysis* (2nd ed.). NJ, USA: Englewood, Cliffs Prentice Hall.
Kanmani, S., Uthariaraj, V. R., Sankaranarayanan, V., & Thambidurai, P. (2007). Object-oriented software fault prediction using neural networks. *Information and Software Technology, 49*, 483–492.
Khoshgoftaar, T. M., & Munson, J. C. (1990). Predicting software development errors using complexity metrics. *IEEE Journal of Selected Areas in Communication, 12*(2), 279–291.
Khoshgoftaar, T. M., Lanning, D. L., & Pandya, A. S. (1994). A comparative study of pattern recognition techniques for quality evaluation of telecommunications software. *IEEE Journal of Selected Areas in Communication, 12*(2), 279–291.
Khoshgoftaar, T. M., Allen, E. B., Kalaichelvan, K. S., & Goel, N. (1996). Early quality prediction: A case study in telecommunication. *IEEE Software, 13*(1), 65–71.
Khoshgoftaar, T. M., Allen, E. B., Hudepohl, J. P., & Aud, S. J. (1997). Application of neural networks to software quality modeling of a very large telecommunications system. *IEEE Transaction in Neural Networks, 8*(4), 902–909.
Khoshgotaar, T. M., Geleyn, E., Nguyen, L., & Bullard, L. (2002). Cost-sensitive boosting in software quality modeling. In *Proceedings of the seventh IEEE international symposium on high assurance systems engineering (HASE'02)* (pp. 51–60).
Kkoshgotaar, T. M., & Seliya, N. (2004). Comparative assessment of software quality classification techniques: An empirical case study. *Empirical Software Engineering, 9*, 229–257.
Koru, A. G., & Liu, H.-F. (2005). Building effective defect-prediction models in practice. *IEEE Software, 23*, 29.
McCabe, T. (1976). A complexity measure. *IEEE Transactions on Software Engineering, 2*(4), 308–320.
Menzies, T., Greenwald, J., & Frank, A. (2007). Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering, 33*(1), 2–13.

Munson, J. C., & Khoshgoftaar, T. M. (1990). Regression modeling of software quality: An empirical investigation. *Information and Software Technology, 32*(2), 106–114.

Munson, J. C., & Khoshgoftaar, T. M. (1992). The detection of fault-prone program. *IEEE Transactions on Software Engineering, 18*(5), 410–422.

Neumann, D. E. (2002). An enhanced neural network technique for software risk analysis. *IEEE Transactions on Software Engineering, 28*(9), 904–912.

Pai, G. J., & Dugan, J. B. (2007). Empirical analysis of software fault content and fault proneness using Bayesian method. *IEEE Transactions on Software Engineering, 33*(10), 675–686.

Quah, T.-S., & Thet Thwin, M. M. (2004). Prediction of software development faults in PL/SQL files using neural network models. *Information and Software Technology, 46*, 519–523.

Schwenk, H., & Bengio, Y. (2000). Boosting neural networks. *Neural Computation, 12*(8), 1869–1887.

Sun, Y.-M., Kamel, M. S., Wong, A. K. C., & Wang, Y. (2007). Cost-sensitive boosting for classification of imbalanced data. *Pattern Recognition, 40*, 3358–3378.

Ting, K. M. (2000). A comparative study of cost-sensitive boosting algorithms. In *Proceedings of the 17th international conference on machine learning (ICML'00)* (pp. 983–990).

Viaene, S., & Dedene, G. (2005). Cost-sensitive learning and decision making revisited. *European Journal of Operation Research, 166*, 212–220.

Xing, F., Guo, P., & Lyu, M. R. (2005). A novel method for early software quality prediction based on support vector machine. In *Proceedings of IEEE international conference on software reliability engineering* (pp. 213–222).

Zhou, Y.-M., & Leung, H. (2006). Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Transactions on Software Engineering, 32*(10), 771–789.

Zhou, Z.-H., & Liu, X.-Y. (2006). Training cost-sensitive neural networks with methods addressing the class imbalance problem. *IEEE Transaction in Knowledge and Data Engineering, 18*(1), 63–77.