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 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 (Basil, 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), 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...
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 (Khoshgostaar, 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; Vaene & 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. Khoshgostaar 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.

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 ∈ {−1, 1}) denotes the cost of misclassifying an example of class i to class j. C−1,1 and C1,−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 C1,−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.
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:

$$h_f(x) = \arg \max_{y \in \mathcal{Y}} \sum_{i:t \neq y} \alpha_i h_i(x)$$

where $\mathcal{Y} = \{-1, 1\}$ is the class label.

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 $y_n \in \mathcal{Y} = \{-1, 1\}$ is the class label.

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

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 $h_t$:

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

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

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

4. Update and normalized the weight vector

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

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:

$$h_f(x) = \text{sign}\left(\sum_{i:t \neq y} \alpha_i h_i(x)\right)$$

Fig. 2. Boosting neural networks with AdaBoost.

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.

### Table 1
Cost matrix for software defect prediction problem.

<table>
<thead>
<tr>
<th>Predict defect-prone</th>
<th>Actual defect-prone</th>
<th>Predict not-defect-prone</th>
<th>Actual not-defect-prone</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_{(1,1)}$</td>
<td>$C_{1,1}$</td>
<td>$C_{(1,-1)}$</td>
<td>$C_{1,-1}$</td>
</tr>
<tr>
<td>$C_{(1,0)}$</td>
<td>$C_{1,0}$</td>
<td>$C_{(1,1)}$</td>
<td>$C_{1,1}$</td>
</tr>
<tr>
<td>$C_{(-1,1)}$</td>
<td>$C_{-1,1}$</td>
<td>$C_{(-1,-1)}$</td>
<td>$C_{-1,-1}$</td>
</tr>
</tbody>
</table>


4539
between CSBNN-WU1 and CSBNN-WU2 is that CSBNN-WU1 does not use the weight-updating parameter \( \alpha \) 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 (Kkoshgoftaar et al., 1997). The misclassification of the prediction 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 (ErrI), and type II error (ErrII) can be obtained as:

\[
\text{MR} = \frac{FP + FN}{TP + TN + FP + FN} \quad (10)
\]

\[
\text{ErrI} = \frac{FP}{TN + FP} \quad (11)
\]

\[
\text{ErrII} = \frac{FN}{TP + FN} \quad (12)
\]

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 Kkoshgoftaar 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 (Kkoshgoftaar & Seliya, 2004).

\[
\text{ECM} = C_I \text{ErrI} P_{\text{ndf}} + C_{II} \text{ErrII} P_{\text{df}} \quad (13)
\]

\[
\text{NECM} = \text{ErrI} P_{\text{ndf}} + \frac{C_{II}}{C_I} \text{ErrII} P_{\text{df}} \quad (14)
\]

In Eqs. (13) and (14), \( C_I \) and \( C_{II} \) are the costs for type I and type II errors which are equal to \( C_{11} \) and \( C_{1,1} \) in the cost matrix, respectively. \( P_{\text{ndf}} \) and \( P_{\text{df}} \) are the prior probabilities of the not-defect-prone and defect-prone modules in the dataset.

#### Table 2

Software defect datasets.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Language</th>
<th># Modules</th>
<th>% Defective</th>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>KC1</td>
<td>C++</td>
<td>2,109</td>
<td>15.5</td>
<td>Storage management</td>
</tr>
<tr>
<td>KC2</td>
<td>C++</td>
<td>522</td>
<td>20.5</td>
<td>Scientific data processing</td>
</tr>
<tr>
<td>CM1</td>
<td>C</td>
<td>496</td>
<td>9.7</td>
<td>NASA spacecraft instrument</td>
</tr>
<tr>
<td>PC1</td>
<td>C</td>
<td>1,107</td>
<td>6.9</td>
<td>Flight software</td>
</tr>
</tbody>
</table>

#### Table 3

Software metrics used in this study.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>Line count of code</td>
<td>L</td>
<td>Halstead's length</td>
</tr>
<tr>
<td>v(G)</td>
<td>McCabe's cyclomatic complexity</td>
<td>I</td>
<td>Halstead's content</td>
</tr>
<tr>
<td>ev(G)</td>
<td>McCabe's essential complexity</td>
<td>E</td>
<td>Halstead's effort</td>
</tr>
<tr>
<td>iv(G)</td>
<td>McCabe's design complexity</td>
<td>B</td>
<td>Halstead's error estimate</td>
</tr>
<tr>
<td>N1</td>
<td>Total number of operators</td>
<td>T</td>
<td>Halstead's programming time</td>
</tr>
<tr>
<td>N2</td>
<td>Total number of operands</td>
<td>LOCb</td>
<td>Number of blank lines</td>
</tr>
<tr>
<td>( \mu_1 )</td>
<td>Number of unique operators</td>
<td>LOCc</td>
<td>Number of comment-only lines</td>
</tr>
<tr>
<td>( \mu_2 )</td>
<td>Number of unique operands</td>
<td>LOCe</td>
<td>Number of code-only lines</td>
</tr>
<tr>
<td>N</td>
<td>Halstead's length</td>
<td>LOCe</td>
<td>Number of lines with both code and comments</td>
</tr>
<tr>
<td>V</td>
<td>Halstead's volume</td>
<td>BR</td>
<td>Number of branches</td>
</tr>
<tr>
<td>( D )</td>
<td>Halstead's difficult</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 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.

#### 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 \( C_{II}/C_I \) from 1 to 10. From these plots, we have the following observations: (1) Among the three algorithms,
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-
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 weight-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


