Contents lists available at ScienceDirect

Applied Soft Computing

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

Software defect prediction using cost-sensitive neural network

Ömer Faruk Arar[a,*], Kürşat Ayan[b]

[a] TUBITAK BILGEM, Information Technologies Institute, Kocaeli, Turkey
[b] Department of Computer Engineering, Faculty of Computer and Information Science, Sakarya University, Sakarya, Turkey

ARTICLE INFO

Article history:
Received 29 August 2014
Received in revised form 1 March 2015
Accepted 20 April 2015
Available online 30 April 2015

Keywords:
Machine learning
Software defect prediction
Artificial Neural Network
Artificial Bee Colony
Software quality
Cost-sensitive classification

ABSTRACT

The software development life cycle generally includes analysis, design, implementation, test and release phases. The testing phase should be operated effectively in order to release bug-free software to end users. In the last two decades, academicians have taken an increasing interest in the software defect prediction problem, several machine learning techniques have been applied for more robust prediction. A different classification approach for this problem is proposed in this paper. A combination of traditional Artificial Neural Network (ANN) and the novel Artificial Bee Colony (ABC) algorithm are used in this study. Training the neural network is performed by ABC algorithm in order to find optimal weights. The False Positive Rate (FPR) and False Negative Rate (FNR) multiplied by parametric cost coefficients are the optimization task of the ABC algorithm. Software defect data in nature have a class imbalance because of the skewed distribution of defective and non-defective modules, so that conventional error functions of the neural network produce unbalanced FPR and FNR results. The proposed approach was applied to five publicly available datasets from the NASA Metrics Data Program repository. Accuracy, probability of detection, probability of false alarm, balance, Area Under Curve (AUC), and Normalized Expected Cost of Misclassification (NECM) are the main performance indicators of our classification approach. In order to prevent random results, the dataset was shuffled and the algorithm was executed 10 times with the use of *n*-fold cross-validation in each iteration. Our experimental results showed that a cost-sensitive neural network can be created successfully by using the ABC optimization algorithm for the purpose of software defect prediction.

1. Introduction

Worldwide software spending amounted to $3.7 trillion in 2013 according to one published report [1]. A survey indicated that 23% of this cost was spent on quality assurance and testing [2]. A notorious example of the importance of testing would be the $125 million NASA spacecraft which was lost in space because of a small data conversion bug [3]. The Department of Defense in the U.S. spends $4 billion every year because of software failures [4]. All these numbers show the importance of software testing and quality assurance, and the need to do it carefully and effectively. Fixing a software bug after deployment is 100 times more costly than fixing it during development [5]. Despite these facts, only 30% of companies allocate a testing and quality assurance budget for their projects [3].

Software quality discipline was introduced with the growth of software systems complexities and user expectations. ISO/IEC 9126 is an international standard for the evaluation of software quality. According to this standard, quality characteristics of a software product are gathered from internal and external metrics. Six quality characteristics defined in the standard are functionality, reliability, usability, efficiency, maintainability, and portability. Internal metrics are measured by considering only the product itself, without dealing with its behavior. External metrics, on the other hand, are related to the behavior of the product only. The focus of this study is internal metrics: the source code of software systems, but not its behavior or functionalities. Software defect prediction activities can be discussed as part of software quality discipline. More quality software means less defect-prone software. For more than two decades software defect prediction models have been of considerable interest to researchers. These models are used to detect defect-prone modules in an automated way before manual testing. The project test team could then spend their time and budget more effectively based on the results of the model. Defect predicted modules are required to have more focus than non-defect predicted modules. Fig. 1 represents the overview role of defect predictors

* Corresponding author at: Information Technologies Institute of TUBITAK, BILGEM Gebze, Kocaeli, Turkey. Tel.: +90 262 675 30 23; fax: +90 262 646 31 87.
E-mail addresses: omer.arar@tubitak.gov.tr (Ö.F. Arar), kayan@sakarya.edu.tr (K. Ayan).

http://dx.doi.org/10.1016/j.asoc.2015.04.045
1568-4946/© 2015 Elsevier B.V. All rights reserved.

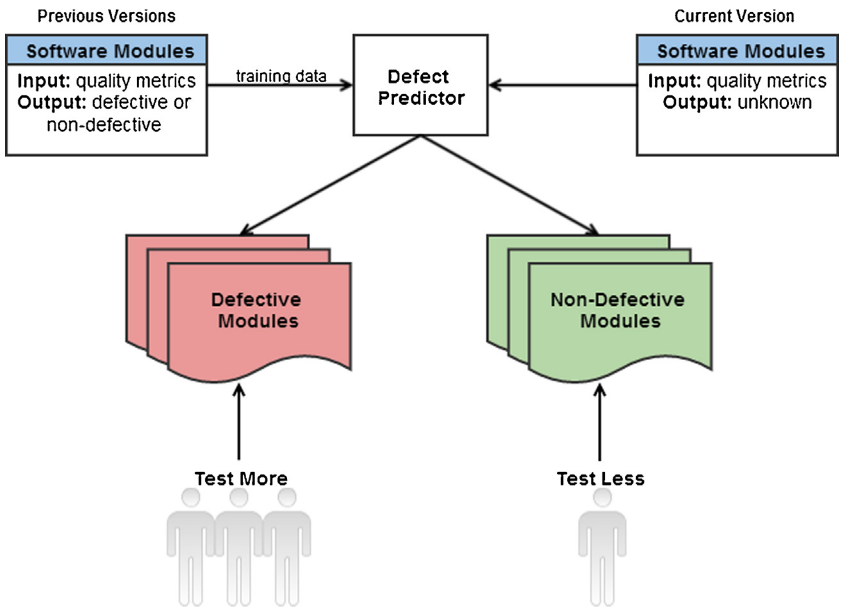

Fig. 1. Role of software defect prediction models on testing.

on testing facilities. Classification of modules is done according to their software quality metrics, which will be explained in later sections. Software quality metrics contain information about defect proneness of a module [6]. All research papers written in this area including this study are proofs of this idea.

Most machine learning algorithms assume that the misclassification cost of each class is equally important. However, the misclassification cost of a minority class is higher than that of a majority class in most cases. For example, predicting a defective module as non-defect prone (ndp) can cause more expensive fixing activities, since the related software may have been deployed for field use with this defect. On the other hand, predicting a non-defective module as defect prone (dp) can cause wasteful testing activities for the project, but that is generally more acceptable than the previous case. Cost-sensitive classifiers take into consideration both of these cases and try to minimize total misclassification costs, but not total misclassification error.

This paper makes the following contributions: (1) a hybrid model, "Artificial Neural Network (ANN) optimized by Artificial Bee Colony (ABC)", is introduced for the software defect prediction domain, and (2) a cost-sensitivity feature is added to ANN by using a parametric fitness function. A trade-off can be made between the classification performance of minority and majority classes by the change of cost coefficients.

The validation of the proposed hybrid model is presented using the following: (1) Commonly used Area Under Curve (AUC), probability of detection (pd), probability of false alarm (pf), balance (bal), and accuracy (acc) performance results are shown, as well as Receiver Operating Characteristics (ROC) curves. (2) AUC results are compared with some other algorithms. (3) The pd, pf, and acc results are shown for different cost values. (4) Normalized Expected Cost of Misclassification (NECM) results are shown according to different cost ratios and compared with some other cost-sensitive neural networks.

The remainder of this paper is organized as follows: The related work is summarized in Section 2. In Section 3, the datasets used in this paper are described. Section 4 explains the algorithms and the proposed prediction model. A brief description of performance indices and the results of the study are given in Section 5. Research conclusions are presented in Section 6.

2. Related work

Different statistical and machine learning methods have been used to solve the software defect prediction problem. Random Forest (RF) [7], Artificial Immune System (AIS) [8], Naive Bayes (NB) [9–11], J48 [12], tree based methods [7,13,14], Case-based Reasoning (CBR) [15], Support Vector Machines (SVM) [16], and Logistic Regression (LR) [17,18] are some of the algorithms used for this problem. Menzies et al. [9] stated that NB with log filtering results in the best performance in terms of True Positive Rate (TPR) and False Positive Rate (FPR).

Besides these basic classification algorithms, some optimization algorithms have also been performed on this problem. They are Genetic Programming (GP) [19], Particle Swarm Optimization (PSO) [20], and Ant Colony Optimization (ACO) [21]. Likewise, ANN has also been used in a few studies and that performance has been investigated [22,23]. Khoshgoftaar et al.'s study [24] is one of the primary studies using the neural network in software quality prediction. In this study, a telecommunication system was used as a benchmarking dataset and the output of their prediction model was the classification of modules as dp or ndp. They concluded that the neural network outperforms statistical methods. Kanmani et al. [25] also used a neural network with object-oriented software quality metrics in their study. The results of this study were compared with two statistical methods.

The majority of software defects are detected in a small portion of its modules [26]. Boehm stated that approximately 20% of modules include 80% of the bugs of a software product (80:20 rule) [27]. The unbalanced distribution of defective and non-defective modules causes poor performance, especially for the minority class (defective modules) [28,29]. This problem has been addressed at data or algorithm levels [30]. At the data level, various oversampling and undersampling methods have been applied in order to balance the distribution of classes [31,32]. These methods, such as random over/under-sampling and SMOTE [33], are simple and efficient; however, the problem domain and the used classification algorithm are main factors of effectiveness [34]. On the other hand, algorithm-level methods address the class imbalance problem by changing their training mechanism with the aim of reaching better accuracy on the minority class [31]. Two examples are one-class

Table 1
Description of the datasets.

Name	Language	Modules (#)	Non-defectives (#)	Defectives (#)	Defect rate (%)
KC1	C++	2109	1783	326	15.45
KC2	C++	522	415	107	20.49
CM1	C	505	449	49	9.83
PC1	C	1109	1032	77	6.94
JM1	C	10,885	8779	2106	19.35

learning [35] and cost-sensitive learning algorithms [31,36]. The proposed model in this paper handles the class imbalance problem at the algorithm level with the use of cost-sensitive ANN. So there is no need to oversample or undersample the dataset as a preprocess step of the predictor model.

Most defect prediction models proposed so far have not considered the cost of the misclassification of defective modules and non-defective modules, with a few exceptions [36,37]. However, in real-world settings misclassification of defective modules is far more important than the misclassification of non-defective modules. The importance levels of these misclassifications are defined by associated costs. A hybrid usage of genetic algorithms and decision trees for cost-sensitive classification is proposed by Turney [38]. There are some attempts to make neural networks cost-sensitive by using oversampling, undersampling or threshold moving [39]. Oversampling and undersampling rates are decided according to associated costs. The number of training examples from each class affects the associated class accuracy. A threshold for the output of a neural network is moved until an optimal value is reached by taking the cost matrix into account. Zhou et al. [36] show that threshold moving is a good choice to make neural networks cost-sensitive.

ABC is an optimization algorithm which has been recently introduced [40]. In this paper, there is a section which explains details of the ABC algorithm. It has been applied to ANN for weight optimization of neuron connections [41]. This hybrid methodology is applied to UCI machine learning datasets for classification purposes and compared with well-known conventional and evolutionary algorithms. Results indicate that the ABC algorithm can be used efficiently to train neural networks for classification purposes [42]. In this paper this hybrid methodology is applied successfully to the software defect prediction problem by a change of fitness function, such that a cost-sensitive neural network is achieved.

3. Datasets

NASA MDP dataset is commonly used as a benchmarking dataset for the defect prediction problem [43]. These software projects were developed for satellite flight control, storage management for ground data, and spacecraft instrumentation. Five of the most used datasets from this repository are used in this study. Each dataset consists of several software modules with quality metrics as input. Each module includes an output label of defective or non-defective, which indicates if any bugs were found in respective modules. Labeling of a module is done manually after a testing phase. Table 1 shows the characteristics of five datasets used in this study. The percentage of defective modules shows an unbalanced distribution of data varying from 6.94% to 20.49%. These NASA projects were developed by C/C++ language.

NASA datasets contain 21 method-level metrics from McCabe [44], (basic and derived) Halstead [45]. Some researchers have expressed the opinion that derived Halstead metrics do not contribute to software defect prediction, so they generally do not use those. McCabe and Halstead metrics are widely used for measuring the quality of software modules. A module may be represented by procedures/methods/functions in procedural languages (C) and by class in object-oriented languages (C++, Java). McCabe metrics collect information about complexity by counting pathways through a module. Halstead metrics forecast readability by counting number operators and operands in a module. A more complex or less readable module is believed to more likely be dp [31]. Table 2 summarizes the 21 method-level metrics collected for each module.

Table 2
Method-level metrics.

Type	Metric	Definition
McCabe	loc	Total lines of code
	v(g)	Cyclomatic complexity
	ev(g)	Essential complexity
	iv(g)	Design complexity
Basic Halstead	lOCode	Count of statement lines
	lOComment	Count of comment lines
	lOBlank	Count of blank lines
	lOCodeAndComment	Count of code and comment lines
	uniqOp	Number of unique operators
	uniqOpnd	Number of unique operands
	totalOp	Number of total operators
	totalOpnd	Number of total operands
	branchCount	Total number of branch count
Derived Halstead	n	Total number of operators and operands
	v	Volume
	l	Program length = (v/n)
	d	Difficulty = (1/l)
	i	Intelligence
	e	Effort to write program = (v/l)
	b	Effort estimate
	t	Time estimator = E/18 s

In this study, a subset of features is used as input instead of using all quality metrics of the dataset. Correlation-based Feature Selection (CFS) [46] technique is applied by using WEKA tool [47]. The same feature selection technique has been applied by different papers [8,16]. Table 3 represents the selected features after the execution of CFS on these five datasets. As Menzies et al. [9] stated, the importance of the classifier is more than that of feature selection, so feature selection technique is not the main focus of this study. Also, some other feature selection techniques such as Principal Component Analysis (PCA) and Info Gain Attribute Evaluation were used in experiments, but results from CFS were the best.

Table 3
Selected metrics for each dataset (after Correlation-based Feature Selection).

Dataset	Metrics (#)	Selected metrics
KC1	8	v, d, i, lOCode, lOComment, lOBlank, uniqOpnd, branchCount
KC2	3	ev(g), b, uniqOpnd
CM1	7	loc, iv(g), i, lOComment, lOBlank, uniqOp, uniqOpnd
PC1	6	v(g), I, lOComment, lOCodeAndComment, lOBlank, uniqOpnd
JM1	8	loc, v(g), ev(g), iv(g), i, lOComment, lOBlank, lOCodeAndComment

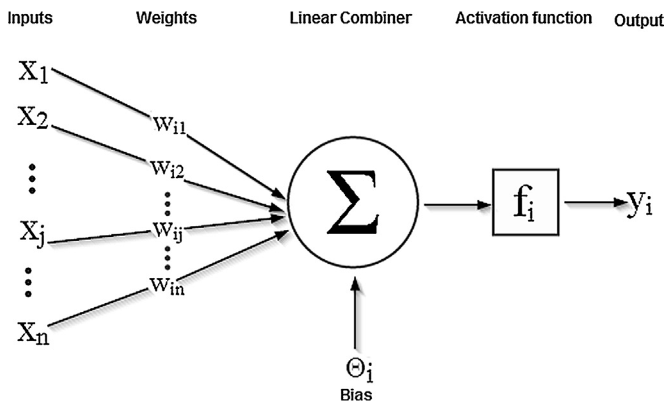

Fig. 2. A neuron with its elements.

On the other hand, the proposed model performs better when using the CFS technique instead of all 21 metrics as input. A bi-directional searching parameter was chosen for CFS.

4. The proposed system

4.1. Artificial Neural Network

A neural network is a model of reasoning based on the human brain [48]. The ANN consists of a set of key information processing units, named neurons. The network consists of three layers which are an input (at least one), a hidden, and an output. The input data (signal) are propagated in a forward direction on a layer-by-layer principle. Connections between neurons are represented by weights. They reflect the importance (or strength) of each neuron. A neural network learns through iterated adjustments of these weights.

Each neuron receives inputs from the neurons of the preceding layer, and each of these inputs is multiplied by its weight value. Then, results of these multiplications are summed. A neuron computes a new activation level from this sum and sends it as an output signal to the following layer. An output signal can be either the final solution of the network or the input of another neuron. Fig. 2 shows a typical neuron with all elements.

An activation function can be a step, sign, sigmoid or linear function. Choosing the activation function is defined according to the expected task of the network, i.e. classification or regression. The output of a neuron which is in ith layer can be described by Eq. (1).

$$y_i = f_i\left(\sum_{j=1}^{n} w_{ij}x_j + \theta_i\right) \tag{1}$$

where y_i is the output of a neuron, n is the total number of inputs to this neuron, x_j is the jth input, w_{ij} is the weight between the current neuron and jth input, and θ_i is the bias of the neuron. f_i represents the activation function of this layer. Generally, the activation function is a nonlinear function such as sigmoid, Gaussian and so on. This enables ANN to model nonlinear relationships. Associations between software quality metrics and module defect proneness are often complex and nonlinear, so ANN is an appropriate choice for software defect prediction problem.

The optimization goal of the network is to minimize the error function by optimizing the network weights (all w_{ij}). The network error at each iteration is calculated by using different methods such as the root mean squared error, mean absolute error, relative absolute error, and root relative squared error. This error is propagated backward in the network and weights are adjusted to minimize the error. The iteration continues until a stopping criterion is met. Stopping criteria can be either a maximum iteration number or minimum error value.

The neural network created for the PC1 dataset is shown in Fig. 3.

In this paper, error adjustments are done with the ABC algorithm, and a specific error function which enables cost-sensitivity is created.

4.2. Artificial Bee Colony

The ABC algorithm simulates the foraging behavior of honey bees. It was proposed by Karaboga for the solution of numerical optimization problems [40]. This algorithm is also used for supervised [42] and unsupervised learning [49] purposes. The ABC algorithm has competitive results with other well-known population-based algorithms, such as the Genetic Algorithm (GA), Differential Evolution (DE), and PSO [50]. The training of ANNs with the ABC algorithm has also been used successfully in several recent studies and has had better results than traditional back propagation algorithms such as gradient descent (GD) and Levenberg–Marquardt (LM) [41].

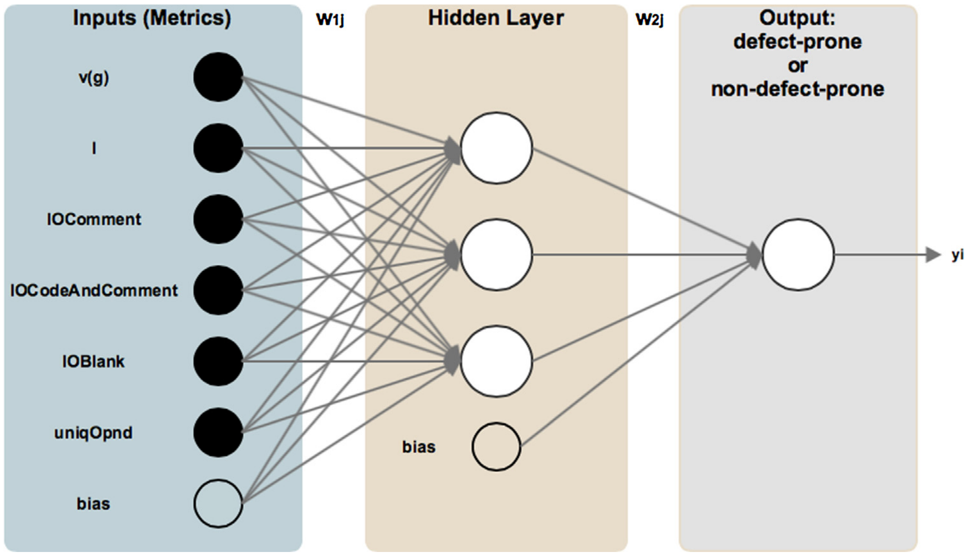

Fig. 3. Architecture of the neural network for PC1 dataset.

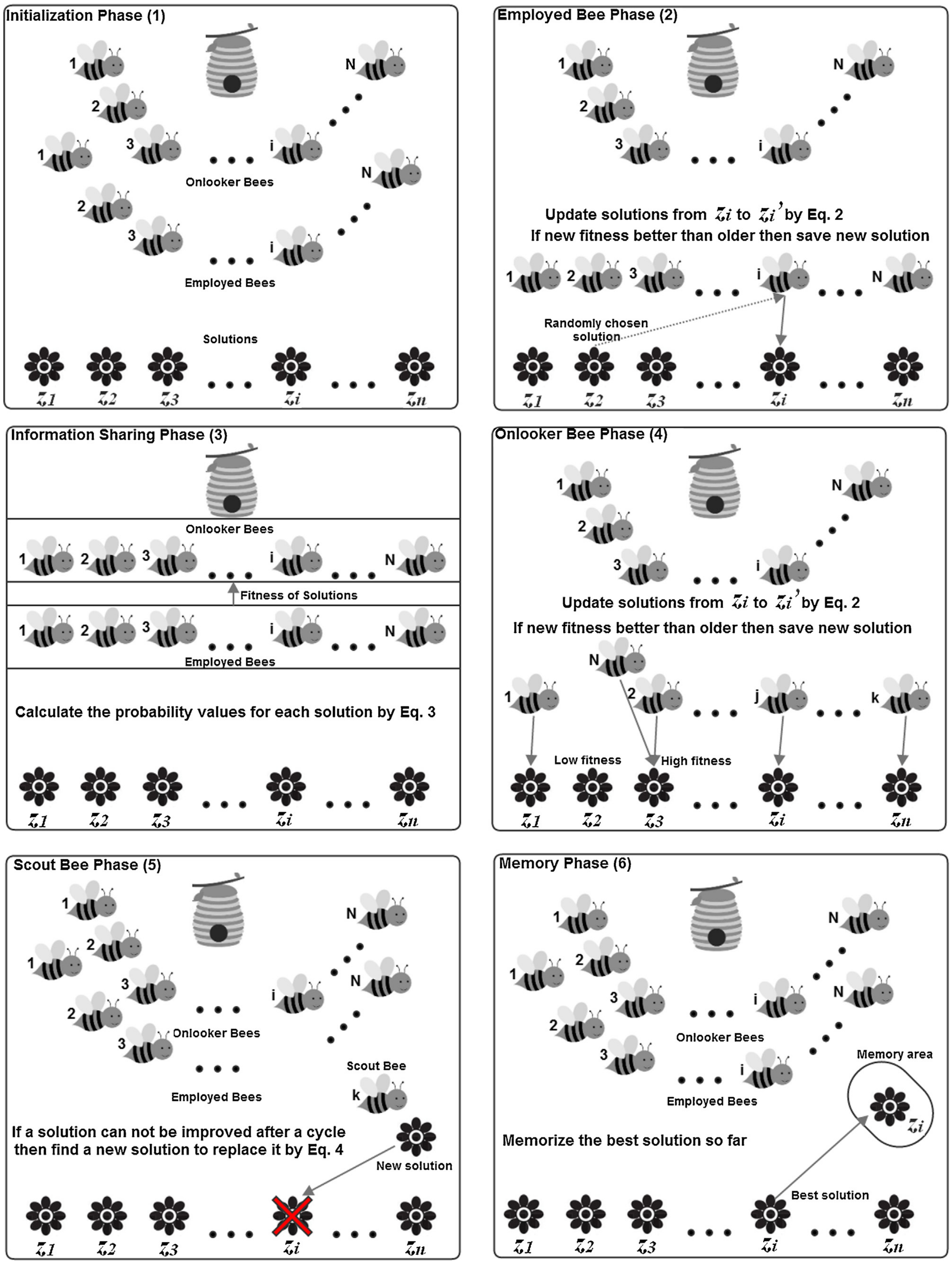
Initialization Phase (1)
Onlooker Bees
Employed Bees
Solutions
Employed Bee Phase (2)
Update solutions from z_i to z_i' by Eq. 2
If new fitness better than older then save new solution
Randomly chosen solution
Information Sharing Phase (3)
Onlooker Bees
Fitness of Solutions
Employed Bees
Calculate the probability values for each solution by Eq. 3
Onlooker Bee Phase (4)
Update solutions from z_i to z_i' by Eq. 2
If new fitness better than older then save new solution
Low fitness
High fitness
Scout Bee Phase (5)
Onlooker Bees
Employed Bees
Scout Bee
If a solution can not be improved after a cycle then find a new solution to replace it by Eq. 4
New solution
Memory Phase (6)
Onlooker Bees
Employed Bees
Memory area
Memorize the best solution so far
Best solution
z_1 z_2 z_3 z_i z_n

Fig. 4. Illustration of the ABC algorithm.

In the swarm-based ABC optimization algorithm, three types of bee populations (employed, onlooker, and scout) look to exploit food sources (candidate solutions) for the best source of nectar (solution). Employed bees visit food sources with the aim of increasing nectar amounts (fitness); onlooker bees wait at the dance area to collect information about food sources from the employed bees. If there is no increase in the amount of nectar for a food source after some attempts, then the responsible employed bee abandons that food source and becomes a scout. The illustration of the ABC algorithm is given in Fig. 4. In previous studies, explanations of the ABC algorithm were shown by flowcharts, pseudo-code, or simply as text; this paper uses an illustration for the first time, as it makes easier to understand the running mechanism of the algorithm.

Phase 1: For each food source there is assigned only one employed bee. The number of employed and onlooker bees are equal. N solutions are randomly generated as are bees. Each solution z_i (i = 1, 2, ..., N) contains a D-dimensional vector where D represents the number of parameters to be optimized within the problem. After Phase 1, the Employed Bee Phase (Phase 2), Information-Sharing Phase (Phase 3), Onlooker Bee Phase (Phase 4), Scout Bee Phase (Phase 5), and Memory Phase (Phase 6) are iterated for a Maximum Cycle Number (*MCN*) in order to improve the solutions and choose the best one at last.

Phase 2: An employed bee makes a modification on the corresponding food source (solution) by changing a randomly chosen parameter and taking same parameter value from one randomly chosen neighbor into account. A new parameter value z'_{ij} is created by Eq. (2).

$$z'_{ij} = z_{ij} + \varphi_{ij}(z_{ij} - z_{kj}) \tag{2}$$

where $j \in \{1, 2, \ldots, D\}$ is a randomly chosen parameter id and $k \in \{1, 2, \ldots, N\}$ is a randomly chosen neighbor solution id which is different than current solution id i – for example, z_{kj} represents the jth parameter value of the kth solution. ϕ_{ij} is a random real number between −1 and +1. The new nectar amount (fitness value) is calculated by using the changed parameter value. If the new fitness value is better than old one, then the old parameter value is forgotten and the new one is memorized. Otherwise, the old parameter value does not change and Failed Improvement Trial (*FIT*) is incremented by one. In other words, a greedy selection manner is performed as the selection process between the old and the candidate solution. As can be seen from Eq. (2), as the parameter z_{kj} gets closer to the parameter z_{ij}, the disturbance on the solution z_i decreases too. Hence, as the exploration approaches the optimum solution, the step length is decreased simultaneously [42].

Phase 3: After completion of Phase 2, employed bees return to the dance area to share nectar amounts of food sources (i.e. fitness values of solutions) with onlooker bees. The probability of visits by onlooker bees is calculated in this phase by Eq. (3).

$$p_i = \frac{fit_i}{\sum_{n=1}^{N} fit_n} \tag{3}$$

where fit_i is the fitness value of solution i.

Phase 4: Onlooker bees select food sources depending on the probability values of food sources which are calculated in the previous phase. Note that food sources with a low fitness value may not be visited by any onlooker bee; on the other hand food sources with high fitness value may be visited more than one bee. Onlooker bees apply same modification as in Phase 2.

Phase 5: Another parameter in the ABC algorithm is the limit which defines when to abandon a food source. If *FIT* value reaches predetermined limit value, then a scout bee discovers a new food source to be substituted for the abandoned one. The creation of new solution is implemented by Eq. (4).

$$z_i^j = z_{\min}^j + rand(0,1)(z_{\max}^j - z_{\min}^j) \tag{4}$$

where abandoned source is z_i and $j \in \{1, 2, \ldots, D\}$. $z_{\max}^j$ and $z_{\min}^j$ are the upper and lower bounds of parameter values that are created. The upper and lower bounds are also used during Phases 2 and 4 when adjusting parameter values.

Phase 6: The solution that has the best fitness value is memorized.

As can be seen from the above description, the original ABC algorithm has three control parameters: colony size ($2 \times N$), abandonment limit, and maximum number of search cycles (*MCN*).

According to Bullinaria et al., colony size and limit have little effect on the results achieved. *MCN* and upper/lower bounds of search space, however, have a big impact on performance [51]. Karaboga et al. used colony size as 30, abandonment limit as 1000, and upper/lower bounds as [−2, +2] for all datasets [42]. But optimizing these parameters based on each dataset can give better results and removing upper and lower bounds of parameters can improve performance [50]. Based on these claims we tried to optimize control parameters (especially upper/lower bounds, *MCN*) of the ABC algorithm according to each NASA dataset.

4.3. The proposed classifier

A cost-sensitive ANN based on the ABC algorithm for software defect prediction is proposed in this paper. In order to convert ANN into a cost-sensitive learner, a different error function is used. The expected cost of misclassification (ECM) [52] is a singular formula considering costs and defective module rate as shown in Eq. (5). C_{FP} represents the cost related to false positive (FP) error (classifying a non-defective module incorrectly as defect prone), and C_{FN} represents the cost of an false negative (FN) error (classifying a defective module incorrectly as non-defect prone). P_{ndp} and P_{dp} indicate prior percentage of ndp modules and dp modules, respectively. It is not easy to define properly misclassification costs for FPR and FNR. The ECM formula is normalized according to C_{FP} and Normalized Expected Cost of Misclassification (NECM) is acquired [53] as shown in Eq. (6). Defining cost ratio by software companies is easier than defining individual costs. The goal of the ABC algorithm used in this study is to minimize the NECM.

$$ECM = C_{FP} \times FPR \times P_{ndp} + C_{FN} \times FNR \times P_{dp} \tag{5}$$

$$NECM = FPR \times P_{ndp} + \frac{C_{FN}}{C_{FP}} \times FNR \times P_{dp} \tag{6}$$

Pseudo-code of the algorithm is given in Fig. 5. *RunABC* is the main function of the classifier which calls other functions like *EmployedBeesFly*, *OnlookerBeesFly*, and *ScoutBeeFly*. Variable definitions of the ABC algorithm and ANN as well as the loading of dataset are done from lines 1 to 7. Dataset is divided into training and test sets by using cross-validation. Details of the cross-validation process are given in Section 5.2. Since the value ranges of the software metrics widely vary, a normalization preprocess is required. Menzies et al. get better results with a log-filter preprocessor [9]. However, in our experiments, min–max (0–1) normalization returned better results than log filtering. In some studies, normalization is applied to the overall dataset instead of training and test sets individually. This is a mistake that effects the performance of the classifier. We obtained minimum and maximum values of each metric from the training set and normalized each test set instance based on these values. From lines 8 to 17, a training phase is applied to find an optimal set of neural network weights, and the performance of algorithm is calculated on the test set by best weights (lines 19–20).

```
function RunABC
        */ INITIALIZATION
1       initialize the variables of the ANN */ inputs, hidden neurons, output
2       initialize the variables of the ABC
3       */ solution size (N), Foods, MCN, upper/lower bounds, limit, Fitness
        */ Foods include weights and it is N x D size matrix
        */ Fitness includes fitness values of each Food
4       FIT = 0     */ Failed Improvement Trial
5       load the dataset
6       shuffle the dataset
7       split dataset as trainSet and testSet (cross-validation)
        */ TRAIN
8       repeat
9       |   normalize all metrics in the range [0.0, 1.0] */ trainSet
10      |   initialize Foods */ Random weight values between upper and lower bounds
11      |   [Foods, Fitness] = EmployedBeesFly ()
12      |   prob = Fitness / sum (Fitness)
13      |   [Foods, Fitness] = OnlookerBeesFly ()
14      |   [Foods, Fitness] = ScoutBeeFly ()
15      |   i = find ( max(Fitness) )
16      |   bestSol = Foods (i)
17      until MCN times
        */ TEST
18      normalize all metrics in the range [0.0, 1.0] */ testSet
19      [tp, tn, fp, fn] = RunANN (testSet, bestSol)
20      calculate performance results (AUC, fr, pd)
```

Fig. 5. Pseudo-code of the main function of the hybrid classifier.

```
function RunANN
        Input : data, sol
        Output : error, tp, tn, fp, fn

1       y = calculate output of the network with logsig activation functions using sol weights
2       y2 = round (y)
3       calculate tp, tn, fp, fn by comparing data output and y2
4       error = apply (6)
```

Fig. 6. Pseudo-code of the Neural Network.

RunANN function is used for running the network according to specified weights and error of the network is calculated. Error value is calculated by Eq. (6). The cost ratio of C_{FN} and C_{FP} is defined according to expectation from the algorithm. The higher the cost ratio, the more important the C_{FN}. Pseudo-code of *RunANN* function is given in Fig. 6. If the output of the network ($y2$) is bigger than 0.5, it is assumed that this module is dp; otherwise, ndp.

EmployedBeeFly function corresponds to Phase 2 of the ABC algorithm. Pseudo-code of this function is given in Fig. 7.

Line 12 of the main function corresponds to Phase 3 of the ABC algorithm.

OnlookerBeesFly function corresponds to Phase 4 of the ABC algorithm. Pseudo-code of this function is given in Fig. 8.

Pseudo-code of Phase 5, the *ScoutBeeFly* function, is given in Fig. 9.

Lines 15 and 16 of the main function are for memorizing the best weight set from all candidate solutions (Phase 6 of the ABC algorithm). As the training of ANN is completed, the testing is performed by this weight set.

5. Experiments and results

5.1. Performance measurement indices

The performance of prediction models for binary classification problems (e.g. dp or ndp) is generally evaluated using a confusion matrix, which is shown in Table 4. The dp modules are regarded as positive (YES) and ndp modules as negative (NO). This matrix includes four possible outputs: If a module is dp and is predicted same, it is marked as true positive (TP); if it is incorrectly predicted as ndp, it is marked as FN. Conversely, an ndp module is marked as true negative (TN) if it is predicted correctly or as FP otherwise. Emam et al. explain different performance indicators which can be

Table 4
Confusion matrix.

	YES (predicted)	NO (predicted)
YES (actual)	TP (true positive)	FN (false negative)
NO (actual)	FP (false positive)	TN (true negative)

```
function EmployedBeesFly
    Input : Foods, Fitness
    Output : Foods, Fitness

1   for i = 1 to N
2       sol = Foods(i)
3       newWeight = apply (2)
4       error = RunANN (trainSet, sol) */ sol include also newWeight
5       newFitness = 1 / error
6       if newFitness > oldFitness
7               Foods(i) = sol  */ Change current Food with newWeight value
8               FIT(i) = 0
9               Fitness(i) = newFitness
10      else */ No improvement on current Food
                */ No change in Foods and Fitness
11              FIT(i)++
12      end
13  end
```

Fig. 7. Pseudo-code of the Employed Bee Phase.

built from these four basic definitions [54]. In this study, probability of detection (pd), probability of false alarms (pf), balance (bal), and accuracy (acc) are used for evaluating the results. As well as above indicators, ROC and AUC are also very popular in the software defect prediction domain, so we used them to show and compare our results with other studies [8,31,55].

- pd (also known as recall) is the percentage of dp modules that are classified correctly, and is computed by Eq. (7). It is also the complement of FNR.

$$pd = recall = \frac{TP}{TP + FN} = 1 - FNR \tag{7}$$

```
function OnlookerBeesFly
    Input : Foods, Fitness
    Output : Foods, Fitness

1   i = 0; t = 0
2   while t < N
3       if rand < prob(i)  */ rand is a real value between 0 and 1
4               t++
5               sol = Foods(i)
6               newWeight = apply (2)
7               error = RunANN (trainSet, sol) */ sol include also newWeight
8               newFitness = 1 / error
9               if newFitness > oldFitness
                        */ Change current Food with newWeight value
10                      Foods(i) = sol
11                      FIT(i) = 0
12                      Fitness(i) = newFitness
13              else */ No improvement on current Food
                        */ No change in Foods and Fitness
14                      FIT(i)++
15              end
16              i++
17      end

18      if i == N +1
19              i = 1
20      end
21  end
```

Fig. 8. Pseudo-code of the Onlooker Bee Phase.

```
function ScoutBeeFly
    Input : Foods, Fitness
    Output : Foods, Fitness

1   $i$ = find (max($FIT$))
2   if $FIT(i)$ > $limit$
3       $newFood$ = create a $Food$
        */ random weight values between $upper$ and $lower$ bounds
4       $Foods$ ($i$) = $newFood$
5       $error$ = RunANN ($trainSet$, $sol$) */ $sol$ include also $newWeight$
6       $newFitness$ = 1 / $error$
7       $Fitness(i)$ = $newFitness$
8   end
```

Fig. 9. Pseudo-code of the Scout Bee Phase.

- pf value indicates incorrectly classified ndp modules. Formula shown in Eq. (8).

$$pf = FPR = \frac{FP}{FP + TN} \tag{8}$$

- bal is equivalent to the normalized Euclidean distance from the desired point (0, 1) to (pf, pd) in a ROC curve as formulated in Eq. (9) [56].

$$bal = 1 - \frac{\sqrt{(1-pd)^2 + (0-pf)^2}}{\sqrt{2}} \tag{9}$$

- acc formula is simply the ratio of correctly classified modules to all modules, as shown in Eq. (10).

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

Note that accuracy is not appropriate for datasets which have uneven class distribution [57]. For example, the PC1 dataset contains dp modules by 6.94%. A learner could score 93.06% acc, even if it predicts all modules as ndp. So it should be considered taking other indicators into account.

The ROC curve is a 2D representation of pd on the y-axis versus pf on the x-axis, as shown in Fig. 10. The ROC curve is created by altering the discrimination threshold over all possible values [58]. Each ROC curve passes through the points (0, 0), stating a classifier that predicts ndp at all times, and (1, 1), stating a classifier that always predicts dp [9]. A line from (0, 0) to (1, 1) provides no information. Points toward the upper left side of the graph are the most acceptable, i.e. obtaining a high pd with low pf. The advantages of the ROC curves are their robustness toward unbalanced datasets and different misclassification costs [59]. Software defect datasets exhibit these characteristics, so it is well suited for this domain [53]. AUC is calculated to get a numeric result for comparison purposes generally. AUC is a widely used performance indicator for imbalanced datasets. Huang et al. [60] presented experimentally that AUC is statistically more discriminating than accuracy for both balanced and unbalanced datasets.

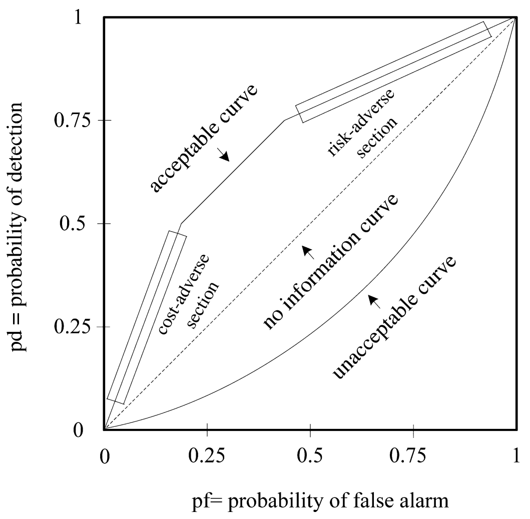

Fig. 10. ROC curves with cost-adverse, risk-adverse and no information sections.

The main goal of a defect predictor is to detect as many dp modules as possible while avoiding false alarms (high pd and low pf). However, there is a trade-off between pd and pf with changing the coefficients C_{FN} and C_{FP} of error function Eq. (6). Different businesses prefer different goals. Having high pf as well as high pd still has still practical usage for mission-critical or risk-adverse systems, where misclassifying a non-defective module is far less important than that of defective module ($C_{FP} \ll C_{FN}$) [61]. On the other hand, (cost-adverse) software projects that have limited time/budget for testing may expect as low pf as possible, i.e. misclassifying a non-defective module is more important than that of defective module ($C_{FN} < C_{FP}$). Risk-adverse and cost-adverse sections of the ROC curve are shown in Fig. 10 [62].

5.2. Experiments

The commonly used N-fold cross-validation technique [63] was used to assess the performance of the proposed classifier. This technique divides the dataset into N parts, each of which consist of an equal number of samples within the original dataset. The algorithm runs N times; in each run training is performed with $(N-1)$ parts and the testing is done with the remaining part. The number of fold (N) is generally chosen as 10. However, when the defect rate is very low, some folds may then include no samples from the minority class. To overcome this problem, N was set according to the defective module rate (dmr) as given in Table 5. After using this approach, N was derived for KC1, KC2, and JM1 dataset experiments as 10;

Table 5
Fold number based on defective module rate (dmr).

Case (IF)	Fold number (#)
dmr < 10	5
10 ≤ dmr < 15	7
dmr ≥ 15	10

Table 6
Control parameters of ABC algorithm.

Parameter	KC1	KC2	CM1	PC1	JM1
Colony size	40	40	40	40	40
Upper, lower	1.0, −3.0	1.0, −2.8	1.0, −3.5	1.0, −4.0	1.0, −2.8
Limit	100	100	100	100	100
MCN	310	70	250	520	310

CM1 and PC1 datasets as 5. Experiments were repeated 10 times to produce statistically reliable results based on the Hall et al.'s study [64]. Fisher et al. stated that some classifiers are affected from order of samples where certain ordering may improve or degrade performance dramatically [65]. Thus, the ordering of samples was shuffled at each run. Therefore, each result given is the average value of 100 experiments.

The main structure of the ANN includes the number of neurons in the hidden layer and the activation functions at the hidden and output layers. They were selected using a trial and error method. The sigmoid function [0,1] was set as an activation function at the hidden and output layers. The number of neurons at the hidden layer was decided as 3. This is the number which returned the best results for five datasets, as several experiments were carried out using different numbers of neurons, varying from 2 to 20.

The control parameters of the ABC algorithm were explored in an automated manner. Source code was implemented to execute the algorithm by control parameters, which were changed at each iteration automatically with results saved to a file. A neural network in nature is a slow algorithm. Repeating one experiment 100 times and exploring best control parameters for the ABC algorithm and ANN requires executing the code many times which may take hours. This automated approach enabled us to run different parameter combinations labor-effectively.

The control parameters selected for the ABC algorithm are given in Table 6. Note that control parameters chosen in this study are different from the original ABC algorithm. As Bullinaria et al. created "Optimized ABC" by optimal parameters for some UCI machine learning datasets [51], we also created "Optimized ABC" for software defect prediction datasets.

5.3. Results

In order to show the performance of the proposed classifier in a more effective manner, this section is divided into two subsections: one regardless of cost-sensitivity, and the other regarding cost-sensitivity. In each section, results were compared with different studies (including cost-sensitive and non-cost-sensitive classifications).

5.3.1. Results regardless of cost-sensitivity

Results in this subsection were obtained without using the costs of FPR and FNR. In other words, costs for FP and FN were assumed to have equal weight by default ($C_{FN}/C_{FP} = 1$). Prior percentages of dp and ndp (P_{dp} and P_{ndp}) were also ignored. With that assumption, the error function of the ABC algorithm changes to Eq. (11)

$$NECM = FPR \times 0.5 + FNR \times 0.5 \quad (11)$$

Results in this subsection were obtained without the cost-sensitivity feature. In this study, five NASA datasets, explained previously, were used. The proposed classifier resulted in ROC graphs for each dataset as shown in Fig. 11. A ROC curve was generated for each execution of the fold, i.e. if the number of folds in the cross-validation process is 10, then the generated ROC curve number is also 10. They were combined to generate one single curve by using curve-fitting techniques based on a 6th degree polynomial.

Table 7
AUC, pd, pf, bal, and acc results with standard deviations of our classifier on the five NASA datasets.

Dataset	AUC	pd (%)	pf (%)	bal (%)	acc (%)
KC1	0.80 ± 0.0019	79 ± 0.93	33 ± 0.67	72 ± 0.56	69 ± 0.60
KC2	0.85 ± 0.0068	79 ± 1.26	21 ± 0.60	79 ± 0.61	79 ± 0.46
CM1	0.77 ± 0.0156	75 ± 2.93	33 ± 1.17	71 ± 1.11	68 ± 0.09
PC1	0.82 ± 0.0083	89 ± 2.31	37 ± 1.84	73 ± 1.07	65 ± 1.63
JM1	0.71 ± 0.0021	71 ± 1.22	41 ± 1.47	64 ± 0.44	61 ± 0.97
Mean	0.79	78.6	33.0	71.8	68.4

Table 8
Comparison of different classifiers in terms of AUC.

Classifier	KC1	KC2	CM1	PC1	JM1	Avg. ranking
Proposed	**0.80**	**0.85**	**0.77**	0.82	0.71	**1.4**
NB	0.79	0.82	0.75	0.70	0.68	2.6
RF	**0.80**	0.82	0.74	**0.85**	**0.75**	1.6
C4.5	0.64	0.67	0.53	0.68	0.61	5.0
Immunos	0.71	0.73	0.63	0.64	0.63	4.0
AIRS	0.60	0.67	0.53	0.58	0.56	5.6

The change of cost ratio has a minor effect on ROC, since these types of graphs in nature show pd, pf pairs for different thresholds. In other words, it shows how good instances with same class output are grouped.

AUC, pd, pf, bal, and acc performance results of these datasets are shown in Table 7. All AUC values are higher than 0.5, which means the proposed classifier has acceptable results. KC1 and KC2 datasets fared better than other datasets, with 0.80 and 0.85 AUC values. The standard deviation of results after 10 executions of the algorithm is shown after the ± sign. The results indicate robustness of the algorithm. CM1 and PC1 have the maximum standard deviation for AUC values with 0.0156 and 0.0083, respectively, since these two datasets are more unbalanced than the others.

A comparison of the proposed classifier with other algorithms in terms of an AUC indicator is shown in Table 8. The result which indicated the best performance is noted in boldface. Results of NB and RF algorithms were taken from Wang et al.'s study [31], the C4.5 result was taken from Sun et al.'s study [55], and Immunos and Artificial Immune Recognition System (AIRS) results were taken from Experiment #3 of Catal et al.'s study [8]. Experiment #3 used the same metric set as we used in this study. Immunos and AIRS are classification algorithms inspired by the human immune system. All these results are from original algorithms, not from any that have been modified. Other studies using the same algorithms have get more or less the same results. Ranking was done according to results from each dataset. The proposed algorithm produced the best results from KC1, KC2, and CM datasets, while it gave second best results from PC1 and JM1 datasets. The RF algorithm had the same best result with the proposed one in the KC1 dataset. The ranking of each algorithm for each dataset was summed and then divided by the number of datasets to get the average ranking. The last column of the table represents the average ranking of each classification algorithm. A lower average ranking value indicates a better performance on related dataset. The proposed classifier earns the best average ranking, with a value of 1.4. The RF value also has a very close performance with 1.6. The NB has a result of 2.6, and other algorithms result in poor performance.

We also evaluated the statistical significance of our classifier with other algorithms by using the non-parametric Friedman test, as was used by Vandecruys et al.'s study [21]. The Friedman test returned result below significance level, meaning at least two of the classifiers are significantly different from each other. Later on, pairwise comparisons were performed according to Conover [66].

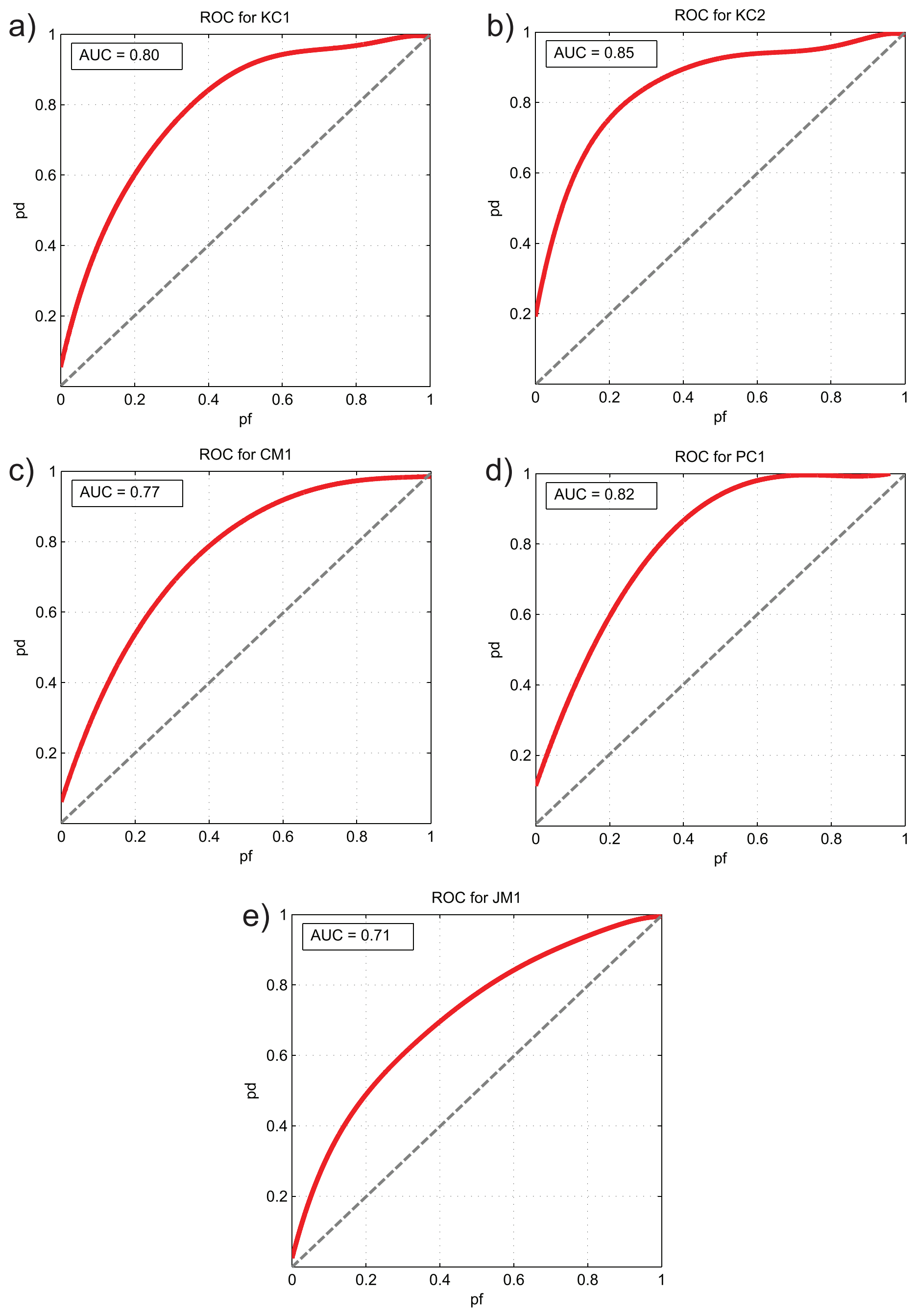

Fig. 11. ROC curves for (a) KC1, (b) KC2, (c) CM1, (d) PC1 and (e) JM1.

Results of these comparisons are shown in Table 9. As can be seen from the table, our classifier's performance is significantly different than all of the others, except the RF, at the 99% level. However, this does not mean it is worse than the RF.

Table 9
Friedman test comparisons [66].

I	J	Differences $(I-J)$
Proposed	NB	1.2[*]
	RF	0.3
	C4.5	3.5[**]
	Immunos	2.7[**]
	AIRS	4.3[**]

[*] $p < 0.01$.
[**] $p < 0.001$.

5.3.2. *Results regarding cost-sensitivity*

The cost-sensitivity feature is another advantage of the proposed classifier. In this subsection, results using cost-sensitivity feature are presented. Different pd, pf pairs could be accomplished by using specific cost ratios (C_{FN}/C_{FP}). Fig. 12 represents pd, pf, and acc results according to four different cost ratios. As can be seen from the figure, when the cost of FN (C_{FN}) decreases (lower cost ratio), pd performance decreases also. And as the cost of FP (C_{FP}) increases (higher cost ratio), the pf performance also increases, as expected. Note that lower pf values indicate a better performance in terms of false alarm. The pf performance has a direct relation with acc performance, because the majority class label is Negative (ndp modules). A lower pf means less error in Negative classes, which enhances acc results.

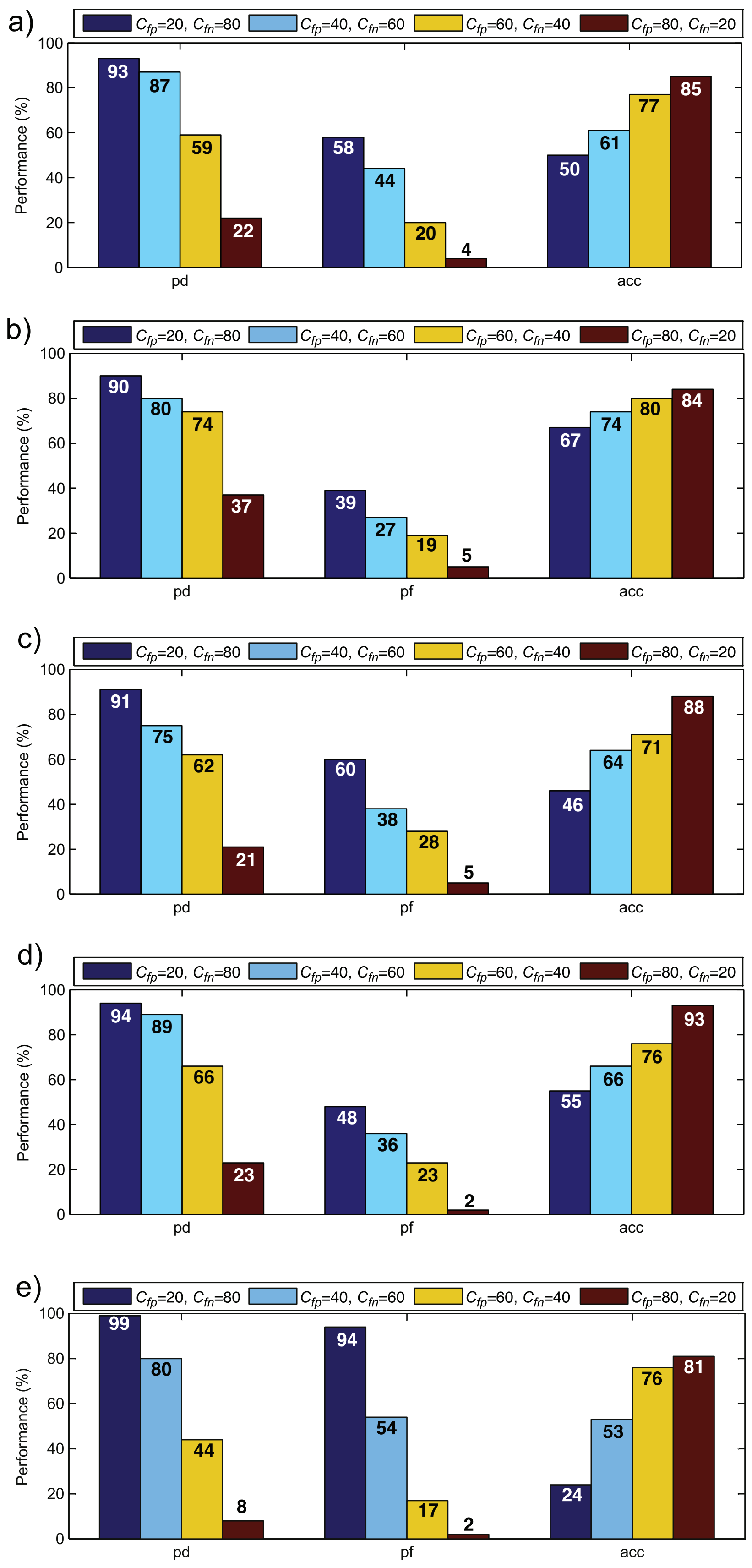

Fig. 12. pd, pf, and acc performances based on different cost values for (a) KC1, (b) KC2, (c) CM1, (d) PC1 and (e) JM1.

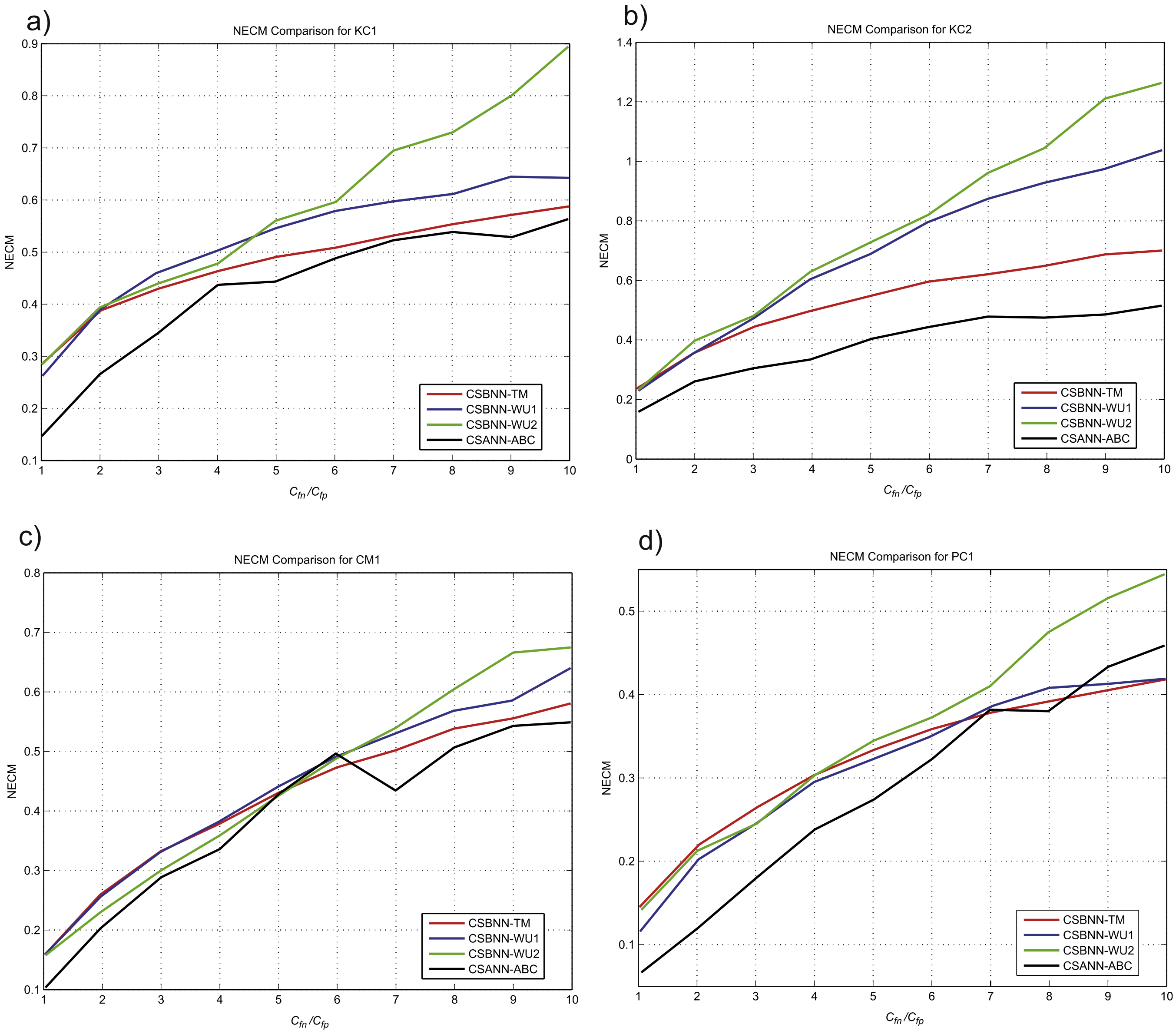

Fig. 13. NECM comparison of the proposed algorithm with others for datasets (a) KC1, (b) KC2, (c) CM1 and (d) PC1.

NECM is a singular performance indicator used for cost-sensitive algorithms. Results of the proposed algorithm were compared with different types of neural networks from a study by Zheng [32]. He used two types of neural networks: Cost-Sensitive Boosting Neural Networks with Threshold-Moving (CSBNN-TM) and Cost-Sensitive Boosting Neural Networks with Weight-Updating (CSBNN-WU). He made two modifications from the original weight updating process. These modified algorithms are called CSBNN-WU1 and CSBNN-WU2. Comparison of the proposed algorithm with these three cost-sensitive algorithms in terms of NECM is shown in Fig. 13. In this study, results of the JM1 dataset are not included, so comparisons were done by other four datasets. In this figure, the proposed algorithm is represented by CSANN-ABC. NECM results according to different cost ratios (varying from 1 to 10) are shown in Fig. 13. The *x*-axis shows cost ratio and the *y*-axis shows related NECM result. The proposed algorithm gave the best result for all four datasets and nearly for all cost ratios. Only when the cost ratio C_{FN}/C_{FP} is 6 for the CM1 dataset, and when cost ratio C_{FN}/C_{FP} is 7 for PC1 dataset, does it give worse results. Note that lower NECM outputs indicate better results.

The ABC algorithm has different control parameters, which affect the performance of the classifier. Performance enhances with the increase of *MCN* until a specific value, and then it reaches a balance, i.e. the increase of *MCN* does not improve performance. So it can be said that the effect of *MCN* on performance is very little. Upper and lower bounds of the search space are the other important parameters. As can be seen from Table 6, their values are not symmetrical, which is different from previous studies that used the ABC algorithm. If value 0 is assumed as the discriminant point, then the negative range is broader than the positive range; that makes the algorithm more powerful against unbalanced datasets. There is a relation between the unbalance rate of the dataset and the lower bound value. For example, PC1 is the most unbalanced dataset and it has the least lower bound; on the other hand, KC2 and JM1 are least unbalanced, so their lower bounds are the highest. Bullinaria et al. claim "Optimized UABC" which has unconstrained bounds, gives better results for most of the UCI Machine Learning datasets [51]. However, the performance is degraded when applying the same approach to software defect prediction datasets.

6. Conclusions

This paper proposes a hybrid classifier for the software defect prediction problem. ANN connection weights are optimized by the ABC algorithm, which is a modeling of the foraging behavior of honey bee swarms. The parametric cost-sensitivity feature is added to ANN by the introduction of a new error function. The costs of misclassifying Positive and Negative classes are set with related coefficients. The value of these costs has an effect on performance of pd, pf and acc. Cost-adverse and risk-adverse businesses may define appropriate cost coefficients for their own software projects.

The performance of the proposed classifier is compared with other algorithms on five NASA datasets and the obtained results show its performance is better than the others. However, the performance difference is not significant. There should be more focus on data preprocessing, feature selection, or other data mining techniques instead of finding a better classifier.

References

[1] J.D. Lovelock, IT Spending Forecast, 4Q13 Update: What Will Make Headlines in 2014? Gartner Inc., 2014 (retrieved 18.02.15) http://www.gartner.com/newsroom/id/2643919
[2] World Quality Report 2013–14, 2014 (retrieved 18.02.15) http://www.capgemini.com/resources/world-quality-report-2013-14
[3] P. Michaels, Faulty Software Can Lead to Astronomic Costs, 2008, http://www.computerweekly.com/opinion/Faulty-software-can-lead-to-astronomic-costs, ComputerWeekly.com (retrieved 23.02.14).
[4] S. Dick, A. Meeks, M. Last, H. Bunke, A. Kandel, Data mining in software metrics databases Fuzzy Sets Syst. 145 (1) (2004) 81–110.
[5] L. Pelayo, S. Dick, Applying novel resampling strategies to software defect prediction, in: IEEE Fuzzy Information Processing Society, NAFIPS'07, San Diego, USA, June 24–27, 2007, pp. 69–72.
[6] V.R. Basili, L.C. Briand, W.L. Melo, A validation of object-oriented design metrics as quality indicators, IEEE Trans. Softw. Eng. 22 (10) (1996) 751–761.
[7] L. Guo, Y. Ma, B. Cukic, H. Singh, Robust prediction of fault-proneness by random forests, in: IEEE 15th International Symposium on Software Reliability Engineering (ISSRE'04), 2004, pp. 417–428.
[8] C. Catal, B. Diri, Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem, Inf. Sci. 179 (8) (2009) 1040–1058.
[9] T. Menzies, J. Greenwald, A. Frank, Data mining static code attributes to learn defect predictors, IEEE Trans. Softw. Eng. 33 (1) (2007) 2–13.
[10] F. Padberg, T. Ragg, R. Schoknecht, Using machine learning for estimating the defect content after an inspection, IEEE Trans. Softw. Eng. 30 (1) (2004) 17–28.
[11] T. Zimmermann, R. Premraj, A. Zeller, Predicting defects for eclipse, in: IEEE International Workshop on Predictor Models in Software Engineering (PROMISE'07), Minneapolis, USA, 2007, pp. 9–14.
[12] A.G. Koru, H. Liu, An investigation of the effect of module size on defect prediction using static measures, in: IEEE International Workshop on Predictor Models in Software Engineering (PROMISE'05), St. Louis, MO, 2005, pp. 1–5.
[13] T.M. Khoshgoftaar, E.B. Allen, W.D. Jones, J.P. Hudepohl, Classification tree models of software quality over multiple releases, IEEE Trans. Reliab. 49 (1) (2000) 4–11.
[14] R.W. Selby, A.A. Porter, Learning from examples: generation and evaluation of decision trees for software resource analysis, IEEE Trans. Softw. Eng. 14 (12) (1988) 1743–1756.
[15] T.M. Khoshgoftaar, N. Seliya, Analogy based practical classification rules for software quality estimation, Empir. Softw. Eng. 8 (4) (2003) 325–350.
[16] K.O. Elish, M.O. Elish, Predicting defect-prone software modules using support vector machines, J. Syst. Softw. 81 (5) (2008) 649–660.
[17] H.M. Olague, S. Gholston, S. Quattlebaum, Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes, IEEE Trans. Softw. Eng. 33 (6) (2007) 402–419.
[18] T. Gyimóthy, R. Ferenc, I. Siket, Empirical validation of object-oriented metrics on open source software for fault prediction, IEEE Trans. Softw. Eng. 31 (10) (2005) 897–910.
[19] M. Evett, T. Khoshgoftaar, P. Chien, E. Allen, GP-based software quality prediction, in: Proceedings of the Third Annual Conference Genetic Programming, San Francisco, USA, 1998, pp. 60–65.
[20] A.B. Carvalho, A. Pozo, S.R. Vergilio, A symbolic fault-prediction model based on multiobjective particle swarm optimization, J. Syst. Softw. 83 (5) (2010) 868–882.
[21] O. Vandecruys, D. Martens, B. Baesens, C. Mues, M.D. Backer, R. Haesen, Mining software repositories for comprehensible software fault prediction models, J. Syst. Softw. 81 (5) (2008) 823–839.
[22] M.M.T. Thwin, T.S. Quah, Application of neural networks for software quality prediction using object-oriented metrics, J. Syst. Softw. 76 (2) (2005) 147–156.
[23] D.E. Neumann, An enhanced neural network technique for software risk analysis, IEEE Trans. Softw. Eng. 28 (9) (2002) 904–912.
[24] T. Khoshgoftaar, E. Allen, J. Hudepohl, S. Aud, Application of neural networks to software quality modeling of a very large telecommunications system, IEEE Trans. Neural Netw. 8 (4) (1997) 902–909.
[25] S. Kanmani, V.R. Uthariaraj, V. Sankaranarayanan, P. Thambidurai, Object-oriented software prediction using neural networks, Inf. Softw. Technol. 49 (5) (2007) 482–492.
[26] B.W. Boehm, P.N. Papaccio, Understanding and controlling software costs, IEEE Trans. Softw. Eng. 14 (10) (1988) 1462–1477.
[27] B.W. Boehm, Industrial software metrics top 10 list, IEEE Softw. 4 (5) (1987) 84–85.
[28] T. Hall, S. Beecham, D. Bowes, D. Gray, S. Counsell, A systematic literature review of fault prediction performance in software engineering, IEEE Trans. Softw. Eng. 38 (6) (2011) 1276–1304.
[29] E. Arisholm, L.C. Briand, E.B. Johannessen, A systematic and comprehensive investigation of methods to build and evaluate fault prediction models, J. Syst. Softw. 83 (1) (2010) 2–17.
[30] R. Moser, W. Pedrycz, G. Succi, A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction, in: Proceedings of the 30th International Conference on Software Engineering (ICSE'08), Leipzig, Germany, May 10–18, 2008, pp. 181–190.
[31] S. Wang, X. Yao, Using class imbalance learning for software defect prediction, IEEE Trans. Reliab. 62 (2) (2013) 434–443.
[32] J. Zheng, Cost-sensitive boosting neural networks for software defect prediction, Expert Syst. Appl. 37 (6) (2010) 4537–4543.
[33] N.V. Chawla, K.W. Bowyer, L.O. Hall, W.P. Kegelmeyer, Smote: synthetic minority over-sampling technique, J. Artif. Intell. Res. 16 (2002) 321–357.
[34] A. Estabrooks, T. Jo, N. Japkowicz, A multiple resampling method for learning from imbalanced data sets, Comput. Intell. 20 (1) (2004) 18–36.
[35] N. Japkowicz, C. Myers, M.A. Gluck, A novelty detection approach to classification, in: Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI'95), Montreal, Canada, August 20–25, 1995, pp. 518–523.
[36] Z.H. Zhou, X.Y. Liu, Training cost-sensitive neural networks with methods addressing the class imbalance problem, IEEE Trans. Knowl. Data Eng. 18 (1) (2006) 63–77.
[37] E. Arisholm, L.C. Briand, Predicting fault-prone components in a Java legacy system, in: ACM/IEEE International Symposium on Empirical Software Engineering (IESE'06), New York, USA, 2006, pp. 8–17.
[38] P.D. Turney, Cost-sensitive classification: empirical evaluation of a hybrid genetic decision tree induction algorithm, J. Artif. Intell. Res. 2 (1995) 369–409.
[39] Y.M. Zhou, H. Leung, Empirical analysis of object-oriented design metrics for predicting high and low severity faults, IEEE Trans. Softw. Eng. 32 (10) (2006) 771–789.
[40] D. Karaboga, B. Basturk, A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm, J. Glob. Optim. 39 (3) (2007) 459–471.
[41] D. Karaboga, B. Akay, C. Ozturk, Artificial bee colony (ABC) optimization algorithm for training feed-forward Neural Networks Modeling Decisions for Artificial Intelligence, LNCS, vol. 4617/2007, Springer-Verlag, 2007, pp. 318–329.
[42] D. Karaboga, C. Ozturk, Neural networks training by artificial bee colony algorithm on pattern classification, Neural Netw. World 19 (3) (2009) 279–292.
[43] M. Chapman, P. Callis, W. Jackson, Metrics Data Program, Technical Report, NASA IV and V Facility, 2004 http://mdp.ivv.nasa.gov/
[44] T. McCabe, A complexity measure, IEEE Trans. Softw. Eng. 2 (4) (1976) 308–320.
[45] M. Halstead, Elements of Software Science, Elsevier Science Inc., New York, 1977.
[46] M.A. Hall, Correlation-based Feature Selection for Machine Learning (Ph.D. dissertation), Department of Computer Science, University of Waikato, Hamilton, New Zealand, 1998.
[47] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I.H. Witten, The WEKA data mining software: an update, ACM SIGKDD Explor. Newlett. 11 (1) (2009) 10–18.
[48] X. Yao, Evolutionary artificial neural networks, Int. J. Neural Syst. 4 (3) (1993) 203–222.
[49] D. Karaboga, C. Ozturk C, A novel clustering approach: artificial bee colony (ABC) algorithm, Appl. Soft Comput. 11 (1) (2011) 652–657.
[50] D. Karaboga, B. Akay, A comparative study of artificial bee colony algorithm, Appl. Math. Comput. 214 (1) (2009) 108–132.
[51] J.A. Bullinaria, Y. Khulood, Artificial bee colony training of neural networks, in: Nature Inspired Cooperative Strategies for Optimization (NICSO 2013), Springer International Publishing, 2014, pp. 191–201.
[52] R.A. Johnson, D.W. Wichern, Applied Multivariate Statistical Analysis, 2nd edition, Prentice Hall, Englewood Cliffs, NJ, USA, 1992.
[53] T.M. Khoshgoftaar, N. Seliya, Comparative assessment of software quality classification techniques: an empirical case study, Empir. Softw. Eng. 9 (3) (2004) 229–257.
[54] K.E. Emam, S. Benlarbi, N. Goel, S.N. Rai, Comparing case-based reasoning classifiers for predicting high-risk software components, J. Syst. Softw. 55 (3) (2001) 301–320.
[55] Z. Sun, Q. Song, X. Zhu, Using coding-based ensemble learning to improve software defect prediction, IEEE Trans. Syst. Man Cybern. Part C: Appl. Rev. 42 (6) (2012) 1806–1817.

[56] Y. Jiang, B. Cukic, Y. Ma, Techniques for evaluating fault prediction models, Empir. Softw. Eng. 13 (5) (2008) 561–595.
[57] A.S. Nickerson, N. Japkowicz, E. Milios, Using unsupervised learning to guide resampling in imbalanced data sets, in: Proceedings of the Eighth International Workshop on Artificial Intelligence and Statistics, FL, USA, January 4–7, 2001, pp. 261–265.
[58] T. Fawcett, An introduction to ROC analysis, Pattern Recognit. Lett. 27 (8) (2006) 861–874.
[59] F. Provost, T. Fawcett, Robust classification for imprecise environments, Mach. Learn. 42 (3) (2001) 203–231.
[60] J. Huang, C.X. Ling, Using AUC and accuracy in evaluating learning algorithms, IEEE Trans. Knowl. Data Eng. 17 (3) (2005) 299–310.
[61] Q. Song, Z. Jia, M. Shepperd, S. Ying, J. Liu, A general software defect-proneness prediction framework, IEEE Trans. Softw. Eng. 37 (3) (2011) 356–370.
[62] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, A. Bener, Defect prediction from static code features: current results, limitations, new approaches, Autom. Softw. Eng. 17 (4) (2010) 375–407.
[63] R. Kohavi, A study of cross-validation and bootstrap for accuracy estimation and model selection, in: Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI'95), Montreal, Canada, August 20–25, 1995, pp. 1137–1143.
[64] M. Hall, G. Holmes, Benchmarking attribute selection techniques for discrete class data mining, IEEE Trans. Knowl. Data Eng. 15 (6) (2003) 1437–1447.
[65] D.H. Fisher, L. Xu, N. Zard, Ordering effects in clustering, in: Proceedings of the 9th International Workshop Machine Learning, Aberdeen, Scotland, 1992, pp. 162–168.
[66] W.J. Conover, Practical Nonparametric Statistics, 3rd edition, John Wiley & Sons, New York, USA, 1999, pp. 367–373.