# Enhancing RBF-DDA Algorithm's Robustness: Neural Networks Applied to Prediction of Fault-Prone Software Modules

Miguel E. R. Bezerra[1], Adriano L. I. Oliveira[2], Paulo J. L. Adeodato[1], and Silvio R. L. Meira[1]

[1] Center of Informatics, Federal University of Pernambuco, P.O. Box 7851, 50.732-970, Cidade Universitaria, Recife-PE, Brazil {merb,pjla,srlm}@cin.ufpe.br
[2] Department of Computing Systems, Polytechnic School, University of Pernambuco, Rua Benfica, 455, Madalena, 50.750-410, Recife-PE, Brazil adriano@dsc.upe.br

Many researchers and organizations are interested in creating a mechanism capable of automatically predicting software defects. In the last years, machine learning techniques have been used in several researches with this goal. Many recent researches use data originated from NASA (National Aeronautics and Space Administration) IV&V (Independent Verification & Validation) Facility Metrics Data Program (MDP). We have recently applied a constructive neural network (RBF-DDA) for this task, yet MLP neural networks were not investigated using these data. We have observed that these data sets contain inconsistent patterns, that is, patterns with the same input vector belonging to different classes. This paper has two main objectives, (i) to propose a modified version of RBF-DDA, named RBF-eDDA (RBF trained with enhanced Dynamic Decay Adjustment algorithm), which tackles inconsistent patterns, and (ii) to compare RBF-eDDA and MLP neural networks in software defects prediction. The simulations reported in this paper show that RBF-eDDA is able to correctly handle inconsistent patterns and that it obtains results comparable to those of MLP in the NASA data sets.

## 1 Introduction

Machine learning techniques have already been used to solve a number of software engineering problems, such as software effort estimation [5], organization of libraries of components [17] and detection of defects in software [3, 9]. This paper is concerned with the detection of defects in software. We aim to predict if a software module contains some defect, without regard of how many defects it contains. For detecting defects in current software projects, a classifier needs to be trained previously with information about defects of past projects. In many papers, the data used during the experiments are obtained from a public repository made available by NASA [1] and by the Promise Repository [15].

These repositories contain static code measures and defect information about several projects developed by NASA.

Bezerra et. al [3] have evaluated the performance of some classifiers in the problem of defect detection. They reported that some patterns of the NASA data sets of the Promise Repository were inconsistent, that is, they had the same input vector and a class different from that of the other replicas. Inconsistent patterns are an inherent feature of some defect detection data sets and therefore should be handled appropriately by the classifiers. For instance, we can have two different software modules characterized by exactly the same features (such as (i) number of operators, (ii) number of operands, (iii) lines of code, etc). The problem is that the first can have defect and the other can be free of defects. This leads to inconsistent patterns (same input vector but different classes). Note that this situation arises because the input information regarding the software modules is not sufficient to differentiate them.

Bezerra et. al [3] have used the RBF-DDA classifier in their experiments and reported an important drawback in the DDA (Dynamic Decay Adjustment) algorithm: it does not work with inconsistent patterns. Therefore, in their experiments, patterns that had repetitions with conflicting classes (that is, inconsistent patterns) were discarded [3], which means loosing information. One of the contributions of this paper is to propose a modified version of RBF-DDA, referred to as RBF-eDDA, which is able to handle inconsistent patterns.

The DDA algorithm was originally proposed for constructive training of RBF neural networks [2]. The algorithm has achieved performance comparable to MLPs in a number of classification tasks and has a number of advantages for practical applications [2, 13], including the fact that it is a constructive algorithm which is able to build a network in only 4 to 5 epochs of training [2].

The use of neural networks for software defect prediction is rare in comparison to other techniques such as Decision Trees J4.8 [4, 7, 10, 12], k-Nearest Neighbor (kNN) [4, 7, 10], and Naive Bayes [7, 10, 12]. Furthermore, multi-layer perceptron (MLP) neural networks was not used for the detection of fault-prone modules using the NASA data. MLPs were used for software defect prediction, yet using other data (not from NASA), such as in [9]. In this way, our experiments utilize MLP neural networks trained with backpropagation to evaluate its performance in the detection of software defects and to compare it to RBF-eDDA in this task.

In summary, the main contributions of this paper are: (i) to propose a modified version of the RBF-DDA algorithm, named RBF-eDDA, which aims to handle inconsistent patterns, (ii) to apply RBF-eDDA to software defect prediction in the NASA data sets, and (iii) to apply MLP to software defect prediction in the NASA data sets and to compare the results obtained to those of RBF-eDDA.

The rest of this paper is organized as follows. Section 2 briefly reviews the standard RBF-DDA network and describes the proposed method, RBF-eDDA. Sections 3 presents the methods used to assess and compare the classifiers whereas Section 4 presents the experiments and discusses the results obtained.

Finally, the Section 5 presents the conclusions and suggestions for future research.

## 2 The Proposed Method

RBF-DDA neural networks have a single hidden layer, whose number of units is automatically determined during training. In this way, during training, the topology starts with an empty hidden layer. Next, the neurons are dynamically included on it until a satisfactory solution has been found [2, 13]. The activation $R_i(\overrightarrow{x})$ of a hidden neuron $i$ is given by the Gaussian function (Eq. 1), where $\overrightarrow{x}$ is the input vector, $\overrightarrow{r_i}$ is the center of the $i$th Gaussian and $\sigma_i$ denotes its standard deviation, which determines the Gaussian's width.

$$R_i(\overrightarrow{x}) = exp\left(-\frac{\|\overrightarrow{x} - \overrightarrow{r_i}\|^2}{\sigma_i^2}\right) \tag{1}$$

RBF-DDA uses 1-of-n coding in the output layer, with each unit of this layer representing a class. Classification uses a *winner-takes-all* approach. In this way, the output unit with the highest activation gives the class. Each hidden unit is connected to exactly one output unit and has a weight $A_i$, whose value is determined by the training algorithm. Output units use linear activation functions with values computed by $f(\overrightarrow{x}) = \sum_{i=1}^{m} A_i \times R_i(\overrightarrow{x})$, where $m$ is the number of RBFs connected to that output. In this paper, each output is normalized as proposed by et al. Bezerra in [3]. Thus, the RBF-DDA becomes capable to produce continuous outputs that represent the probability of a module being fault-prone.

The DDA algorithm has two parameters, namely, $\theta^+$ and $\theta^-$, whose default values are 0.4 and 0.1, respectively [2]. These parameters are used to decide on the introduction of new neurons in the hidden layer during training. The DDA training algorithm for one epoch is presented in the Algorithm 1 [2].

It was originally believed that the parameters $\theta^+$ and $\theta^-$ would not influence RBF-DDA performance. Yet, Oliveira et al. have recently demonstrated that the value of $\theta^-$ may significantly influence classification performance [13]. Therefore, in this paper we use the default value for $\theta^+$ ($\theta^+ = 0.4$) and select the best value of $\theta^-$ for each data set via cross-validation, as in [13].

### *2.1 Training RBF Networks with the Enhanced DDA Algorithm*

Before the explanation of the modifications in the DDA algorithm, it is necessary to understand its drawbacks regarding inconsistent patterns. For this task, we use an example that has a one-dimensional training set composed by

**Algorithm 1** DDA algorithm (one epoch) for RBF Training

1: **for all** prototypes $p_i^k$ **do** ▷ Reset weights
2: $A_i^k = 0.0$
3: **end for**
4: **for all** training pattern $(\overrightarrow{x}, c)$ **do** ▷ Train one complete epoch
5: **if** $\exists p_i^c : R_i^c(\overrightarrow{x}) \geq \theta^+$ **then**
6: $A_i^c += 1.0$
7: **else**
8: add new prototype $p_{m_c+1}^c$ with: ▷ Commit: introduce new prototype
9: $\overrightarrow{r}^c_{m_c+1} = \overrightarrow{x}$
10: $\sigma^c_{m_c+1} = max_{k \neq c \wedge 1 \leq j \leq m_k}\{\sigma : R^c_{m_c+1}(\overrightarrow{r}^k_j) < \theta^-\}$
11: $A^c_{m_c+1} = 1.0$
12: $m_c += 1$
13: **end if**
14: **for all** $k \neq c, 1 \leq j \leq m_k$ **do** ▷ Shrink: adjust conflicting prototypes
15: $\sigma_j^k = max\{\sigma : R_j^k(\overrightarrow{x}) < \theta^-\}$
16: **end for**
17: **end for**

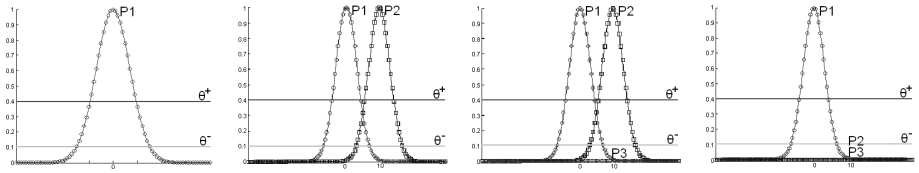
three patterns: (*P1*)=0, class=$C1$; (*P2*)=10, class=$C0$; (*P3*)=10, class=$C1$. The patterns *P2* and *P3* have the same input, but are from distinct classes.

Following Algorithm 1, when *P1* is presented, there is no neuron in the hidden layer, and then a new Gaussian (RBF) is created with $\overrightarrow{r_{P_1}} = [0]$, $A_{P_1} = 1$ and $class = C1$ (Fig. 1(a)). When *P2* is encountered, the DDA algorithm introduces a new Gaussian with $\overrightarrow{r_{P_2}} = [10]$, $A_{P_2} = 1$ and $class = C0$, since there was no neuron from class $C0$.

After *P2*'s Gaussian have been included, all others Gaussians with class different from $C0$ have their width shrank (Fig. 1(b)). When *P3* is presented, the algorithm realizes that it has the same class of *P1* and that its activation is less than $\theta^+$, then a new prototype should be introduced. At this moment, DDA's drawback can be observed. The *P3*'s Gaussian conflicts with that of *P2*, because both have the same center but distinct classes. Thus, the Euclidean distance between the centers of *P2* and *P3* is *zero*. In other words, Eq. 1 gives $\|\overrightarrow{x_{P_3}} - \overrightarrow{r_{P_2}}\| = 0$, and that makes the standard deviation of *P3* equal to *zero* as well ($\sigma_{P_3} = 0$), causing a division by *zero* (see line 10 of Algorithm 1). The result can be seen in Fig. 1(c).

After *P3* is introduced, the algorithm shrank the width of Gaussians with a class different of $C1$. In this way, $\sigma_{P_2}$ receive *zero* because $\|\overrightarrow{x_{P_3}} - \overrightarrow{r_{P_2}}\| = 0$, then it will happen to *P2* the same as *P3*, as can be seen in Fig. 1(d). Therefore, the algorithm never converges to one solution, because the final SSE (Sum of Squared Errors) is always calculated as *NaN* (Not-a-Number).

To handle the problem caused by inconsistent patterns just described, we propose a modified version of RBF-DDA, named RBF-eDDA, which aims to turn it robust enough to treat the problem of inconsistent patterns. We began from the principle that the hidden layer could not hold this type of conflict, because it will influence the inclusion of new Gaussians and the adjustment of its




(a) (b) (c) (d)

**Fig. 1** An example of the DDA Algorithm's drawback: (a) *P1* is encountered and a new RBF is created; (b) *P2* leads to a new prototype for class $C0$ and shrinks the radius of the existing RBFs of class $C1$; (c) *P3* Gaussian is presented and cannot be calculated because $\sigma_{P_3} = 0$; (d) during the shrink of the Gaussians, $\sigma_{P_2} = 0$ and *P2* also cannot be calculated.

width. In RBF-eDDA, whenever a new Gaussian is included, the algorithm will verify if it is conflicting with some pre-existent Gaussian. If it is not conflicting, the algorithm will create the Gaussian normally; otherwise, the algorithm will not create a new Gaussian. Instead, the algorithm will reduce the weight $A_i = A_i - 1$ of the pre-existent Gaussian that is conflicting. If the Gaussian weight becomes *zero*, it is removed from the hidden layer.

## 3 Assessing the Performance

The defect detection is a cost-sensitive task whereby a misclassification is more costly than correct classification. Other problem is that the data sets utilized to train the predictors have a skewed class distribution, that is, these data sets have more modules with defects than modules without defects. Then, we need to use evaluation metrics capable to assess the performance of the classifiers, and that handle these constraints. The ROC curve [18] is the best way to deal with cost-sensitive problems and unbalanced datasets because it depicts the performance of a classifier regardless of the class distribution or the error costs [18]. Thus, in order to assess the classifiers, we use ROC's AUC (Area Under Curve). The best classifier is the one with the higher AUC [18].

A defect predictor is a binary classifier that has four possible outcomes, as shown in Fig. 2, which depicts the confusion matrix. Considering the defect detection problem, the columns of the Fig. 2 represent the actual class of a software module while the rows represent the class predicted by the classifier. Thus, the *NO column* represents the modules that do not have defects while the *YES column* represents the inverse. Conversely, the *no row* represents the modules labeled as fault-free by the classifier, while the *yes row* represents the modules labeled as fault-prone.

The confusion matrix is the core of several evaluation metrics. The *Accuracy* (Acc) is the proportion of the total number of modules that were correctly classified (Eq. 2). The *Probability of Detection* (PD) is the proportion of defective

| | | Actual Class | |
|---|---|---|---|
| | | **NO** | **YES** |
| Prediced Class | **no** | True Negative | False Negative |
| | **yes** | False Positive | True Positive |

**Fig. 2** Confusion Matrix of a Binary Classifier.

modules that were correctly identified (Eq. 3). In the other case, the *Probability of False Alarm* (PF) (Eq. 4), is the proportion of correct modules that were incorrectly identified. Another metric is the *Precision* (Prec), that is the proportion of the predicted defective modules that were correct (Eq. 5).

$$Acc = \frac{(TP + TN)}{(TP + TN + FP + FN)} \tag{2}$$

$$PD = \frac{TP}{(TP + FN)} \tag{3}$$

$$PF = \frac{FP}{(FP + TN)} \tag{4}$$

$$Prec = \frac{TP}{(TP + FP)} \tag{5}$$

These four metrics are used during our experiments to evaluate the performance, yet their values depend on the adjustment of the classifiers' operation point (threshold), because it modifies the class memberships and, consequently, the confusion matrix distribution. To choose the classifier's best threshold, we also use the ROC curves; the threshold is the best ROC's point, which is the one closer to the point ($x$-*axis*=0, $y$-*axis*=1) [6].

## 4 Experiments

This Section presents the experiments carried out with the proposed method, RBF-eDDA, as well as with MLP neural networks. In the experiments with RBF-eDDA, we adjusted the parameter $\theta^-$ to find the best performance of the classifier [13]. We employed the following values for this parameter: 0.2, $10^{-1}$, $10^{-2}$, $10^{-3}$, $10^{-4}$, $10^{-5}$ and $10^{-6}$. These values were chosen because they were used successfully in previous papers [13, 3].

In this study, we compare the performance of RBF-eDDA with other neural network, the MLP trained with Backpropagation [8]. MLP is a feedforward neural network. As its architecture is not constructive, it is necessary to vary its topology to choose the one that performs well for a given problem. In the experiments we varied three parameters of the MLP: the number of neurons

in the hidden layer, the *learning rate* ($\eta$) and the *momentum*. The simulations using MLP networks were carried out using different topologies, that is, number of neurons in the hidden layer. The values used were: 1, 3, 5, 9 and 15. The values used for the *learning rate* were 0.001 and 0.01; for the *momentum* we used 0.01 and 0.1.

In our experiments we are concerned with the reproducibility, then, we decided that the five data sets used – $CM1$, $JM1$, $KC1$, $KC2$ and $PC1$ – should be obtained from the Promise Repository [15], since it stores data sets that can be reused by other researches. Each dataset has 21 input features based on software metrics such as cyclomatic complexity, lines of comments, total operators, total lines of codes, etc. Detailed information about each feature can be obtained freely on the MDP web site [1].

To compare two or more classifiers, it is important that the data used in the evaluation are exactly the same. Then, to guarantee the reproducibility of our results, all experiments utilized the same data set separation. In order to do this, the fold separation and the stratification were made by the Weka [18]. We set the Weka's random seed to 1 and made the separation of the datasets in 10 stratified folds. Then, using the Weka framework and Promise datasets, other researchers can reproduce the experiments reported here.

Before the simulations, we made a preprocessing in the datasets. This is a procedure whereby the data are cleaned and prepared for training and for testing the classifiers. Initially we observed that some patterns had missing values. These patterns with missing values were removed, since they represented a very small sample of the total number of patterns in each data set (less than 0.1%) and therefore could be discarded [11]. The next step of the preprocessing was the dataset normalization because the values of each feature had different amplitudes in relation to the others, and this could induce skewed results. Table 1 summarizes the characteristics of each dataset after preprocessing. Notice that all datasets have a small percentage of modules with some defect.

**Table 1** Summary of the datasets used in this paper.

| Dataset | #Modules | %Modules with defects |
|---|---|---|
| CM1 | 498 | 9.83 |
| JM1 | 10885 | 19.35 |
| KC1 | 2109 | 15.45 |
| KC2 | 522 | 20.50 |
| PC1 | 1109 | 6.94 |

### *4.1 Analysis of Results*

In our comparison between RBF-eDDA and MLP, the AUC is used as the main criterion; we also report the PD, PF, Acc and Prec obtained by the classifiers for comparison. The AUC was selected as the most important criterion because it summarizes the global performance of the classifier in a single scalar value. The simulations' results of the RBF-eDDA and MLP are reported in Table 2.

**Table 2** Results of the classifiers RBF-eDDA and MLP with backpropagation.

| | RBF-eDDA | | | | | | | MLP with BackPropagation | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Dataset | $\theta^-$ | #Units | AUC | PD | PF | Acc | Prec | AUC | PD | PF | Acc | Prec |
| CM1 | $10^{-3}$ | 336 | 0.773 | 0.837 | 0.347 | 0.671 | 0.208 | 0.760 | 0.755 | 0.327 | 0.681 | 0.201 |
| JM1 | $10^{-2}$ | 7238 | 0.596 | 0.653 | 0.461 | 0.561 | 0.254 | 0.718 | 0.652 | 0.326 | 0.670 | 0.324 |
| KC1 | $10^{-5}$ | 999 | 0.712 | 0.801 | 0.384 | 0.644 | 0.276 | 0.792 | 0.755 | 0.309 | 0.701 | 0.309 |
| KC2 | $10^{-1}$ | 252 | 0.744 | 0.766 | 0.313 | 0.703 | 0.387 | 0.825 | 0.757 | 0.186 | 0.803 | 0.513 |
| PC1 | $10^{-2}$ | 613 | 0.859 | 0.805 | 0.218 | 0.784 | 0.216 | 0.819 | 0.805 | 0.300 | 0.707 | 0.167 |
| Average | | | 0.737 | 0.772 | 0.345 | 0.673 | 0.268 | 0.783 | 0.745 | 0.289 | 0.712 | 0.303 |

For RBF-eDDA, Table 2 shows the best $\theta^-$ for each dataset and the number of neurons of the hidden layer. For the MLP networks the configuration that obtained the best stability and performance across all datasets was the configuration with 3 neurons in the hidden layer, *learning rate* set to 0.01 and *momentum* set to 0.1. Using the AUC for the comparison, notice that the MLP outperformed the RBF-eDDA classifier on the JM1, KC1 and KC2 datasets; on the other hand, the RBF-eDDA outperformed the MLP in the CM1 and the PC1. In the CM1 dataset, the difference between the classifiers was small, but in the PC1 dataset it was higher.

Fig. 3 shows the ROC curve of the classifiers for each dataset along with the best operation points. The values of PD, PF, Acc and Prec were computed using these operation points. A prominent result of the MLP has occurred in the KC2 dataset, with PD=75.7% and a high accuracy and small amount false alarms. In the case of RBF-eDDA, the best result occurred in the PC1 dataset, with a PD=80.5%, PF=21.8% and a high accuracy.

## 5 Conclusions

This paper contributes by proposing RBF-eDDA, an enhanced version of the DDA algorithm. RBF-eDDA aims to handle inconsistent patterns, which occur in software defect detection data sets. The original RBF-DDA algorithm was not able to train if the data set contains inconsistent patterns. We report a number of experiments that have shown that RBF-eDDA handles inconsistent patterns

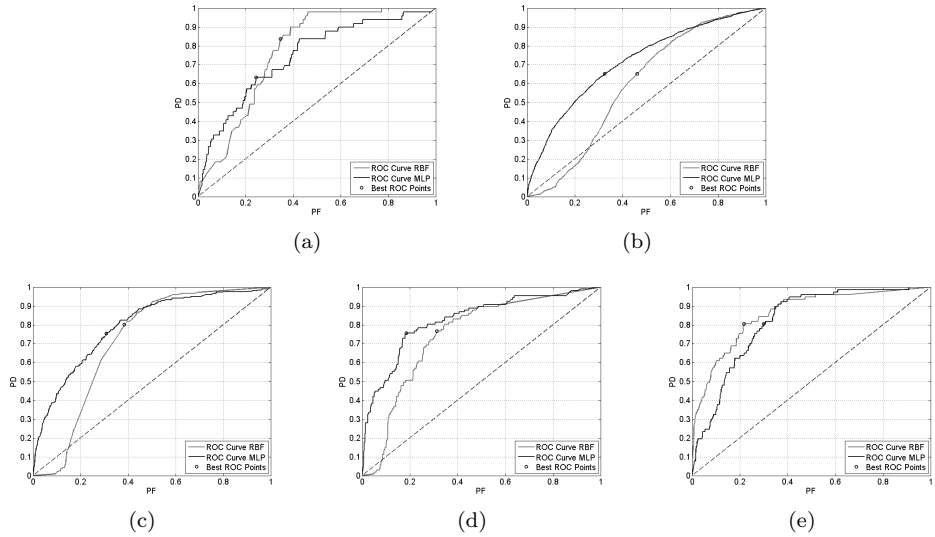



**Fig. 3** ROC Curves obtained by the MLP and RBF-eDDA classifiers for the datasets CM1(a), JM1(b), KC1(c), KC2(d) and PC1(e).

adequately. Our experiments also aimed to compare the proposed method to MLP networks for software defect prediction. The experiments have shown that RBF-eDDA and MLP have similar performance in this problem. RBF-eDDA offers an advantage over MLP since it has only one critical parameter ($\theta^-$) whereas MLP has three.

Considering the study of Shull et al. [16], which asserts that in a real development environment a peer review catches between 60-90% of the defects, our results are useful, since RBF-eDDA achieved mean PD of 77.2% and the MLP achieved 74.5%.

We endorse the conclusions of Menzies et al.[12], since they state that these predictors would be treat as *indicators* and not as definitive *oracles*. Therefore, the predictors are suitable tools to guide test activities, aiding on the prioritization of resources and, hence, in the reduction of costs in software factories where the development resources are scarce.

As future work, we propose to investigate a committee machine composed of RBF-eDDA and MLP networks to achieve a better classification performance; this was already used with success in time series novelty detection [14]. The motivation for such a committee is that in some data sets RBF-eDDA outperforms MLP whereas in others the inverse occurs.

## Acknowledgments

The authors would like to thank the CNPq (Brazilian Research Agency) for its financial support. They also would like to thank NeuroTech (Brazilian Data Mining Company) for allowing the use of its MLP simulator.

## References

1. Metrics data program. URL `http://mdp.ivv.nasa.gov`
2. Berthold, M.R., Diamond, J.: Boosting the performance of RBF networks with dynamic decay adjustment. In: Adv. in Neural Inf. Proc. Syst., vol. 7, pp. 521–528 (1995)
3. Bezerra, M.E.R., Oliveira, A.L.I., Meira, S.R.L.: A constructive RBF neural network for estimating the probability of defect in software modules. In: IEEE Int. Joint Conference on Neural Networks, pp. 2869–2874. Orlando, USA (2007)
4. Boetticher, G.D.: Nearest neighbor sampling for better defect prediction. SIGSOFT Softw. Eng. Notes **30**(4), 1–6 (2005)
5. Braga, P.L., Oliveira, A.L.I., Ribeiro, G., Meira, S.R.L.: Bagging predictors for estimation of software project effort. In: IEEE International Joint Conference on Neural Networks (IJCNN2007), pp. 1595–1600. Orlando-Florida, USA (2007)
6. Fawcett, T.: An introduction to ROC analysis. Pattern Recognition Letters **27**(8), 861–874 (2006)
7. Guo, L., et al.: Robust prediction of fault-proneness by random forests. In: 15th International Symposium on Software Reliability Engineering, pp. 417–428 (2004)
8. Haykin, S.: Neural Networks: A Comprehensive Foundation, 2nd edn. Prentice Hall (1998)
9. Kanmani, S., et al.: Object-oriented software fault prediction using neural networks. Inf. Softw. Technol. **49**(5), 483–492 (2007)
10. Khoshgoftaar, T.M., Seliya, N.: The necessity of assuring quality in software measurement data. In: IEEE METRICS, pp. 119–130. IEEE Computer Society (2004)
11. Lakshminarayan, K., Harp, S.A., Samad, T.: Imputation of missing data in industrial databases. Appl. Intell. **11**(3), 259–275 (1999)
12. Menzies, T., Greenwald, J., Frank, A.: Data mining static code attributes to learn defect predictors. IEEE Trans. Software Engineering **33**(1), 2–13 (2007)
13. Oliveira, A.L.I., et al.: On the influence of parameter $\theta^-$ on performance of RBF neural networks trained with the dynamic decay adjustment algorithm. Int. Journal of Neural Systems **16**(4), 271–282 (2006)
14. Oliveira, A.L.I., Neto, F.B.L., Meira, S.R.L.: Combining MLP and RBF neural networks for novelty detection in short time series. In: MICAI, Lecture Notes in Computer Science, vol. 2972, pp. 844–853. Springer (2004)
15. Shirabad, J., Menzies, T.: The promise repository of software engineering databases (2005). URL `http://promise.site.uottawa.ca/SERepository`
16. Shull, F., et al.: What we have learned about fighting defects. In: Eighth IEEE Int. Symposium on Software Metrics, pp. 249–258 (2002)
17. Veras, R.C., de Oliveira, A.L.I., de Moraes Melo, B.J., de Lemos Meira, S.R.: Comparative study of clustering techniques for the organization of software repositories. In: IEEE Int. Conf. on Tools with Artificial Intelligence, pp. 210–214 (2007)
18. Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann (2005)