Information Sciences 441 (2018) 152–170

Contents lists available at ScienceDirect

Information Sciences

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

A novel approach for software defect prediction through hybridizing gradual relational association rules with artificial neural networks

Diana-Lucia Miholca, Gabriela Czibula*, Istvan Gergely Czibula

Department of Computer Science, Babeş-Bolyai University, 1, M. Kogalniceanu Street, Cluj-Napoca 400084, Romania

ARTICLE INFO

Article history:
Received 20 September 2017
Revised 11 February 2018
Accepted 14 February 2018
Available online 15 February 2018

Keywords:
Artificial neural network
Gradual relational association rule
Machine learning
Software defect prediction

ABSTRACT

The growing complexity of software projects requires increasing consideration of their analysis and testing. Identifying defective software entities is essential for software quality assurance and it also improves activities related to software testing. In this study, we developed a novel supervised classification method called *HyGRAR* for software defect prediction. *HyGRAR* is a non-linear hybrid model that combines *gradual relational association rule mining* and *artificial neural networks* to discriminate between *defective* and *non-defective* software entities. Experiments performed based on 10 open-source data sets demonstrated the excellent performance of the *HYGRAR* classifier. *HyGRAR* performed better than most of the previously proposed approaches for software defect prediction in performance evaluations using the same data sets.

1. Introduction

Data mining, [23] techniques are now employed widely to extract meaningful knowledge from large volumes of data in different domains. Data mining applications include various real world scenarios such as assessing cybersecurity awareness [21], detecting communities in social networks based on user frequency pattern mining [31], and software process improvement to produce high-quality software [22]. Data mining-based solutions for predicting software defects have also been investigated [12,17,29].

Software defect prediction involves identifying defective software modules in new versions of a software system, which makes the testing process more efficient by focusing on testing and analyzing the modules identified as defective [10]. Identifying software defects is also useful for guiding code review, which is often employed as a quality assurance activity in agile methodologies for software development.

Despite its importance and extensive applicability, the detection of defective software modules is a complex task, particularly in complex and large-scale software systems. In particular, *cross-project* prediction is one of the main challenges in the field of defect prediction. The *cross-project* prediction strategy involves training the prediction model based on past data from different software projects and then using it to predict the defective software entities in a new project [34]. From the perspective of supervised learning, software defect prediction remains a difficult problem, mainly because the data used

* Corresponding author.
E-mail addresses: mdir1308@scs.ubbcluj.ro (D.-L. Miholca), gabis@cs.ubbcluj.ro (G. Czibula), istvanc@cs.ubbcluj.ro (I.G. Czibula).
URL: http://www.cs.ubbcluj.ro/~gabis (G. Czibula), http://www.cs.ubbcluj.ro/~istvanc (I.G. Czibula)

https://doi.org/10.1016/j.ins.2018.02.027
0020-0255/

for training are highly imbalanced, where there is a very small number of *defective* instances compared with the number of *non-defective* instances. Numerous approaches have been proposed for software defect prediction and detection, but there is still a need to develop accurate and robust defect detectors, as well as software metrics for distinguishing between defective and non-defective software modules.

The main contribution of this study is a novel solution for software defect prediction called *HyGRAR*, which detects defective modules in software systems. *HyGRAR* is a hybrid supervised learning method based on a combination of *gradual association rule mining,ijicic* and *artificial neural networks* (ANNs) [40]. ANNs learn *gradual relations* between software metrics that characterize the software entities. These learned relations are effectively expressed using previously trained neural networks and these relations are used in the subsequent mining process. The *gradual relational association rules* (GRARs) obtained from the mining process are then used to distinguish between *defective* and *non-defective* software entities.

In the proposed *HyGRAR* classifier, we intuitively assume that the relations between relevant software metrics values may be significant for indicating vulnerability to defects. Furthermore, we consider that learning their relations is preferable to predefining them because of their adaptability and expressive power. The proposed *HyGRAR* is novel from the perspectives of both search-based software engineering and data mining.

The remainder of this paper is organized as follows. In Section 2, we highlight the importance of the software defect detection problem. Brief background details of *GRARs* and *neural networks* are given in Section 3. In Section 4, we review research into *software defect prediction*. We introduce our hybrid model for software defect prediction in Section 5. In Section 6, we present the results of computational experiments performed using 10 open source data sets. The results obtained are analyzed in Section 7 and we compare them with those produced using related approaches for software defect prediction. We give our conclusions and suggestions for extending the proposed approach in Section 8.

2. Problem statement and importance

Software defect detection involves identifying software modules that contain errors.

Predicting software reliability is very important for software development, particularly in large scale and complex software projects [46]. Clark and Zubrow [11] analyzed the importance of predicting software defects. One important reason for defect prediction is that it helps software managers to measure how software projects evolve. In addition, it supports process management by assessing a software product's quality and the performance of a process [11]. Moreover, information about software defects can be employed to increase the effectiveness of related testing activities to improve the quality of the next version of the software. Defect prediction is essential for guiding the code review process by indicating the locations of lines of source code that are highly likely to be defective and require particular attention.

From the perspective of supervised learning, the prediction of defects is a difficult task because the training data used for training defect predictors in a supervised manner are highly imbalanced. The faulty modules in a software system are greatly outnumbered by the error-free modules. Thus, the conventional learning algorithms are often biased toward the non-defective class. In the worst case, defective examples are treated as outliers and ignored, so the algorithm simply generates a trivial classifier that classifies every software module as non-defective. Therefore, there is great interest in building unbiased classifiers that start from imbalanced software defect data.

3. Background

Next, we consider the background concepts related to *GRARs*, which we introduced in a previous study [13]. We also provide a brief review of *ANNs*.

3.1. GRARs

We introduced the concept of *GRARs* in a previous study [13] by extending *relational association rules*, [38] via the use of *fuzzy relations*.

In the following, we briefly explain the concept of GRARs [13].

Let us consider that $\mathcal{E} = \{e_1, e_2, \ldots, e_n\}$ is a set of *instances* (entities, records), where each instance is characterized by a list of m attributes (features), $\mathcal{A} = (a_1, \ldots, a_m)$. Each attribute a_i takes values from a non-empty and non-fuzzy domain D_i, which also contains a *null* (*empty*) value. We denote $\Phi(e_j, a_i)$ as the value of attribute a_i in the instance e_j.

We define a fuzzy binary relation [7] R between two domains D_i and D_j as $R = \{< (x, y), \mu_R(x, y) >: x \in D_i, y \in D_j\}$, where μ_R: $D_i \times D_j \rightarrow [0, 1]$ is a membership function that associates each pair (x, y), $x \in D_i$, $y \in D_j$ with the *membership degree* $\mu_R(x, y)$ to measure the degree to which the relation R is satisfied. $\mu_R(x, y) = 1$ denotes that x and y are completely related according to R, whereas $\mu_R(x, y) = 0$ means that x and y are fully unrelated [7]. We denote $\mathcal{F}$ as the set of all possible binary fuzzy relations that can be defined between any two non-fuzzy attribute domains.

Definition 1. *[13]* A **GRAR**, *gRule*, is a sequence $(a_{i_1}\ R_1\ a_{i_2}\ R_2\ a_{i_3} \ldots R_{\ell-1}\ a_{i_\ell})$, where $\{a_{i_1}, a_{i_2}, a_{i_3}, \ldots, a_{i_\ell}\} \subseteq \mathcal{A} = \{a_1, \ldots, a_m\}$, $a_{i_j} \neq a_{i_k},\ j, k = 1.\ell,\ j \neq k$ and $R_j \in \mathcal{F}$ is a binary fuzzy relation over $D_{i_j} \times D_{i_{j+1}}$. D_{i_j} represents the domain of the attribute a_{i_j}.

The *membership degree* of the GRAR *gRule* for an instance $e \in \mathcal{E}$ is defined as $\mu_{gRule}(e) = \min\{\mu_{R_j}(\Phi(e, a_{i_j}), \Phi(e, a_{i_{j+1}}))$, $j = 1, 2, \ldots, \ell - 1\}$ and expresses the grade or the magnitude to which the relation is satisfied.

Table 1
Sample data set [20].

a_1	a_2	a_3
3.93	3.99	3.96
3.78	3.96	3.94
3.88	3.96	4.02
3.93	4.03	4.06
3.84	4.10	3.94
3.75	4.02	4.09
3.98	4.06	4.17
3.84	3.92	4.12

Table 2
Interesting *GRARs*.

Gradual relational association rule	Confidence	Membership
$a_1 \lesssim a_2$	1	0.907
$a_1 \approx a_2$	1	0.923
$a_1 \lesssim a_3$	1	0.913
$a_2 \approx a_3$	1	0.967

a) If $a_{i_1}, a_{i_2}, a_{i_3}, \ldots, a_{i_\ell}$ are non-missing in k instances from the data set, then we call $s = \frac{k}{n}$ the *support* of the rule.
b) If we denote $E' \subseteq \mathcal{E}$ as the set of instances where $a_{i_1}, a_{i_2}, a_{i_3}, \ldots, a_{i_\ell}$ are non-missing and $\mu_{gRule}(e) > 0$ for each instance e from E', then we call $c = \frac{|E'|}{n}$ the *confidence* of the rule.
c) Using the notation from (b), we call $m = \frac{\sum_{e \in E'} \mu_{gRule}(e)}{n}$ the *membership* of *gRule*.

A *GRAR* is considered *interesting* if its support s and confidence c are greater than or equal to user-specified thresholds, i.e., $s \geq s_{\min}$ and $c \geq c_{\min}$.

An A-priori like algorithm called GRANUM was introduced by [13] for discovering *interesting* GRARs within a data set. The algorithm starts by computing the interesting 2-length gradual rules. Then, as in the *Apriori* algorithm, *GRANUM* generates candidate rules of length k ($k \geq 3$) from the rules of length $k-1$. Next, it prunes the candidates that violate the minimum support and confidence constraint. As a consequence, only the interesting k-length rules are obtained, which are used in the next iteration. The iterative process stops when no new interesting rules are found.

For more details of GRARs and the GRANUM algorithm for mining GRARs, we refer the reader to [13].

3.1.1. Example

To illustrate the concept of *GRARs*, we consider the following simple example.

The data set used in our example (from [20]) corresponds to a real experiment conducted to compare three different feeding treatments for birds. Table 1 gives the actual data set, where each instance is characterized by values of three attributes denoted by a_1, a_2, and a_3. Each attribute in the table represents the average weight (expressed in kilograms) of birds with one of the feeding treatments.

The *GRANUM* algorithm is employed to discover the binary interesting *GRARs* where the support and confidence are greater than or equal to $s_{\min} = 1$ and $c_{\min} = 0.9$, respectively. We also constrain the algorithm to only list those rules where the *membership* is greater than or equal to 0.9, which ensures that the relations obtained are strongly satisfied.

The fuzzy relations considered in this experiment are: $\lesssim$, $\gtrsim$, and $\approx$. Table 2 presents the results obtained after mining interesting GRARs from the data set in Table 1.

3.2. ANNs

ANNs, [44] are adaptive learning systems and they are useful for approximating a non-linear mapping between input and output data. The output values represent variables that are difficult to quantify, but which are known to depend in some complex manner on the input variables. ANNs are powerful supervised learning models and they have been used in different domains, which range from software defect prediction to the analysis of chemical processes [3,4,15].

The adaptability of an ANN means that its parameters are autonomously adjusted during a *training phase*. In a supervised learning scenario, some examples are provided to the network in the form of (input, output) pairs during the *training phase*. The input vector and its corresponding desired output vector [37] are provided by an external supervisor.

In the *testing phase*, after the ANN model has been built, the learned model can generate an output vector when presented with a previously unseen input instance.

The *multilayer perceptron* (MLP) is a feedforward ANN comprising multiple fully interconnected layers of neurons (computational units). The MLP is trained using an inductive learning procedure called the *backpropagation* algorithm [30]. After an input–output example is provided to the network, the error obtained is fed back into the network and its parameters are adjusted accordingly. This process is repeated until acceptable performance is obtained.

Another neural network architecture is the *radial basis function network* (RBFN). According to Mitchell [30], the hidden units generate an output determined by a Gaussian activation function centered at some input instance. The output of a RBFN is obtained as a linear combination of the hidden unit activations.

The form of the hypothesis learned by a RBFN for an input x is shown in formula (1) [30].

$$\hat{f}(x) = \sum_{k=1}^{K} w_k \phi_k(d(c_k, x)) \tag{1}$$

In Formula (1), w_i are positive real numbers representing the parameters (weights) of the radial basis functions, K denotes the predefined number of radial basis functions, and c_k is the center (prototype) chosen for the kth radial basis function. The function $\phi_k(d(c_k, x))$ is commonly selected as a Gaussian function centered at the point c_k with a variance ${\sigma_k}^2$ (see formula (2)) and the metric d is usually the Euclidean distance.

$$\phi_k(d(c_k, x)) = e^{\frac{d^2(c_k, x)}{2 \cdot {\sigma_k}^2}} \tag{2}$$

The kernel width of c_k is set to:

$$\sigma_k = \alpha \frac{1}{p} \sum_{i=1}^{p} d(c_k, c_i^k), \tag{3}$$

where p is a constant parameter, c_i^k is the ith nearest prototype of c_k, and α is a heuristically chosen selected parameter.

4. Software defect prediction: literature review

The prediction of defects in software systems is a highly active area of software engineering research and there is great interest in the development of novel high-performance software defect predictors. Indeed, 208 studies of defect prediction published between 2000–2010 were considered by Hall et al. in a systematic review of *software defect prediction*, [19].

Many machine learning-based techniques have been employed for predicting software defects, but we only review the approaches that were evaluated based on the same data sets used in our experiments.

Yu and Mishra [45] investigated the problem of developing cross-project models for software detection. These models are created based on data extracted from a software system, but applied and tested with different software. Binary logistic regression is applied to the data sets related to projects from a Turkish white-goods manufacturer (Ar) and two models are built: the *self-assessment* model, which is tested using the data set employed for its creation, and the *forward-assessment* model, where certain data sets are used for creating the model and a different data set is used for its testing. It was shown that the self-assessment model performs better but the forward-assessment model provides a more realistic estimate of the actual performance of the binary logistic regression model.

Afzal et al. [2] used multiple linear regression and genetic programming to evaluate the performance of different resampling methods in the software defect detection problem. The Ar data sets were used as case studies to compare five different resampling methods: tenfold cross validation, hold-out, repeated random sub-sampling, non-parametric bootstrapping, and leave-one-out (LOO) cross-validation. They concluded that there were no significant differences in the performance of the resampling methods according to the area under the receiver operating characteristic (ROC) curve (AUC) performance measure. They suggested that this may have been due to the imbalanced nature of the data sets considered or the high number of features (software metrics) employed.

Abaei et al. [1] introduced an unsupervised learning approach for predicting software defects based on a combination of self-organizing maps (SOMs) and threshold values. After building the SOM, the nodes from the map are labeled based on threshold values, where the nodes with weight vectors that have elements greater than the corresponding threshold are considered to be defective. In the classification stage, the best matching unit for the given test instance is considered and its label gives the target classification value.

Clustering has been used in previous studies to group software entities into defective and non-defective types. In particular, Bishnu and Bhattacherjee [5] employed the *K-means* clustering algorithm after identifying the centers of the clusters using quad trees. The same clustering algorithm was applied by Varade and Ingle [42] but they used hyper-quad trees to initialize the centers of the clusters.

Sevim et al. [9] proposed a software defect prediction approach based on the X means clustering algorithm from Weka [18]. After determining the clusters, threshold values for software metrics are used to identify the clusters that represent faulty and non-faulty software entities. The Xmeans and Expectation–Maximization (EM) clustering algorithms were used by Park and Hong [35] together with several feature selection methods implemented in Weka.

Malhotra [26] compared statistical and machine learning methods for software defect prediction. In particular, logistic regression was compared with six machine learning approaches comprising decision trees, ANNs, support vector machines,

cascade correlation networks, group method of data handling (GMDH) polynomial networks, and gene expression programming. These learning models were evaluated based on two Ar data sets and the best performance was obtained using decision trees.

The problem of cross-project software defect detection was also addressed by Jaechang and Sunghunin in [32], who investigated the situation when the software metrics used for building the prediction model are different to the software metrics computed for the software under evaluation. They proposed an approach based on matching the software metrics obtained from the data sets by considering their correlations, distribution, and other characteristics. In order to compare their proposed approach with existing methods, they used 28 data sets, including the Ar data sets and logistic regression from Weka.

Malhotra [25] employed multivariate logistic regression to detect defective software entities in the *Ant* system. First, the data were preprocessed by detecting and removing outliers. Multivariate logistic regression based on ten-fold cross-validation was then applied by considering only two software metrics from the data set: response for class (RFC) and cyclomatic complexity (CC).

Canfora et al. [8] proposed a cross-project defect prediction approach where the problem was formulated as a multi-objective optimization problem. Two different objectives were considered: the number of defective entities detected and the cost of analyzing the predicted defective application classes. This approach [8] was based on a multi-objective genetic algorithm and it was experimentally evaluated based on 10 data sets, including JEdit and Ant. They compared the multi-objective approach with a single-objective method and concluded that the formed obtained better performance.

An approach for software defect detection using a Bayesian approach was presented by Okutan and Yildiz [33]. Nine publicly available data sets (including JEdit, Ant, and Tomcat) were used in the experimental evaluation together with the Weka implementation of the K2 algorithm [18]. The software metrics used in the Promise data repository were added as two new software metrics to the data sets: number of developers (NOD) and lack of coding quality (LOCQ). The efficiencies of different software metric pairs at defect detection were analyzed, where the RFC–LOC (Lines of code), RFC–LOCQ, and RFC–WMC (Weighted methods for class) pairs were identified as the most effective.

Panichella et al. [34] proposed a combined approach called COmbined DEfect Predictor (CODEP), which combines the classifications provided by different machine learning techniques to improve the detection of defective entities. Ten open source software systems, including JEdit 4.0, Ant 1.7, and Tomcat 6.0, were evaluated using CODEP in the context of cross-project defect prediction. They concluded that the accuracy of the predictions was improved by combining different classifiers.

Xuan et al. [43] investigated the performance of within-project defect prediction based on 10 defect data sets from the PROMISE repository using six state-of-the-art machine learning approaches. Ten-fold cross-validation was performed based on each data set, including JEdit 4.0, Ant 1.7, and Tomcat 6.0, and several evaluation measures were reported.

In order to cope better with noise and imprecise information, Marian et al. [28] investigated a fuzzy approach for software defect prediction. A fuzzy decision tree method was proposed and the JEdit and Ant open source software systems were used in the experimental evaluation. Experiments were performed based on the original data sets and data sets enhanced with defective software entities from the Tomcat data set. The results demonstrated the effectiveness of the fuzzy approach compared with the non-fuzzy method.

5. Our approach: methodology

Next, we introduce *HyGRAR* as our supervised learning model for predicting defective software entities. *HyGRAR* is a combination of *GRAR mining* and *ANNs*. ANNs learn *gradual relations* between software metrics. These relations expressed as trained neural networks are then used in the mining process. The *GRARs* obtained from the mining process are used to distinguishing between *defective* and *non-defective* software modules.

We consider a software system $\mathcal{S}yst$ described by a set of software *entities* (classes, modules, methods, functions), $\mathcal{S}yst = \{e_1, e_2, \ldots, e_n\}$. The software entities are represented as vectors. Several software metrics are used as features (attributes) characterizing the software entities, $\mathcal{F} = \{ftr_1, ftr_2, \ldots, ftr_l\}$. Thus, each element from the vector associated with an entity represents the value of a software metric applied to that entity. A software entity $e_i \in \mathcal{S}yst$ is represented as an l-dimensional vector, $e_i = (e_{i1}, e_{i2}, \ldots, e_{il})$, where e_{ij} expresses the value of the software metric ftr_j computed for the software entity e_i.

5.1. HyGRAR *classifier for defect prediction*

There are two possible target classes for the software defect classifier: the *positive* ("+") class and the *negative* ("−") class. We denote "+" as the defective class and "−" as the non-defective class in the following.

The main idea of the *HyGRAR* classifier is explained as follows.

We learn from the experience of previous projects or previous versions of the current project to detect vulnerability to defects by considering the existing relations between two software measurements. The *gradual relations* expressing these dependencies are represented by ANNs trained based on labeled data.

In the next phase, we exploit the learned relations by considering them in the relational association rule mining algorithm. Assuming that the data set corresponding to the current software project is partitioned into a training subset and

a testing subset (software entities for testing), we separately mine the interesting *GRARs* from both the defective and non-defective training subsets.

In the testing phase, based on the interesting GRARs mined during the previous phase, we determine whether the entity under testing is defective or non-defective.

In our proposed method, the following steps are performed to determine whether a software entity contains a defect.

1. Data pre-processing.
2. Building the *HyGRAR* classifier (the training step).
3. Classification stage.

The previously mentioned steps are described in the following.

5.2. Data pre-processing

To train our *HyGRAR* classifier, we employ two disjoint sets: X and Y. Both X and Y comprise a set of software systems, which are characterized by a number of l software metrics. Thus, each software system is represented by l-dimensional software entities (as indicated previously). Each software entity from X and Y is labeled as *defective* or *non-defective*. The training data sets X and Y are highly imbalanced, where the number of defects is much smaller than the number of non-defects. The data from X are used for learning the *gradual relations*, whereas Y is used for discovering the interesting *GRARs* that characterize each of the *defective* and *non-defective* classes.

The first pre-processing step is *feature selection*. This feature selection step is data dependent and it uses the l software metrics to characterize the systems from X and Y as those that are relevant for distinguishing faulty and non-faulty software entities. Details of this step are given in Section 6.1. After the feature selection step, f features are considered as the most relevant for the defect classification task. The f software metrics identified after the feature selection step are considered for all the systems from X and Y.

After reducing the data by selecting only the most relevant f features, we prepare the data from X for training the ANNs. Perfectly balanced subsets are built starting with the data sets describing the software projects. We split a data set with n_- non-defective instances and n_+ defective instances ($n_+ \ll n_-$) according to the following steps.

1. We partition the set of all non-defective instances into $s = [n_-/n_+]$ subsets (buckets) of n_+ elements and one subset containing the remaining $r = n_- - s \cdot n_+$ instances.
2. To the subset with cardinality r, we add $s - r$ randomly selected non-defective instances. To this set, we then add non-defective instances from the complete buckets of $2 \cdot n_+$ instances so the maximum difference between the number of instances selected from two different buckets is at most 1.
3. Finally, to learn the gradual relations between each two attributes, we generate all of the subsets with two attributes (software metrics) from all of the buckets. Thus, for each pair of software metrics, from each bucket, we extract one subset with only these two attributes.

Thus, the preliminary data pre-processing step yields a collection $\mathcal{D}_1$ of multiple perfectly balanced sets containing defective and non-defective instances, which are characterized by only two relevant software metrics, and an unbalanced data set $\mathcal{D}_2$ representing the software entities from Y characterized by the same f relevant software metrics identified after the feature selection step.

5.3. Training

The data sets $\mathcal{D}_1$ and $\mathcal{D}_2$ obtained after the pre-processing step described in Section 5.2 are used for building the *HyGRAR* classification model.

The propose hybrid model has two distinct training phases.

1. **Phase 1.** Learning gradual relations from the data set $\mathcal{D}_1$ between any two software metrics with respect to the vulnerability to defects.
2. **Phase 2.** Mining from $\mathcal{D}_2$ interesting GRARs characterizing each of the *defective* and *non-defective* classes.

First, using each of the perfectly balanced subsets of two attributes from $\mathcal{D}_1$, we train two ANN models: a MLP and RBFN.

For a subsequent test instance, both classifiers output two subunit values denoting the likelihood of the instance belonging to the two classes. We semantically translate the output values into *membership degrees* and the trained neural networks themselves into pairs of *gradual relations*.

Thus, based on the previously trained ANN, *Ann*, where the output neurons produce two values in [0, 1], m_- and m_+, we derive two *gradual relations*. According to m_-, one relation expresses the extent to which the combination of two actual software metric values (v_1 and v_2) characterizes the non-defective class, i.e., the degree to which a software entity with values v_1 and v_2 is non-faulty for the specific software metrics. According to m_+, the second relation indicates the vulnerability to defects for the same combination of software metric values (v_1 and v_2).

The first phase of learning the *gradual relations* semantically corresponds to learning based on the experience of past software projects and/or past versions of the current project according to the extent to which combinations of software

measurement values predispose to defects. Thus, in order to train the neural networks, we use data from projects (or versions) other than the project (or version) considered to evaluate the performance of *HyGRAR* classifier.

In the second learning phase, the *GRANUM* algorithm considers the gradual relations expressed by trained neural networks to mine the interesting *GRARs* characterizing the defective and non-defective classes from both the defective and non-defective subsets of the data set $\mathcal{D}_2$. In particular, for each pair (ftr_j, ftr_k) of attributes (software metrics), the *GRANUM* mining algorithm described in Section 3.1 tests the set of all fuzzy relations, irrespective of class (defective or non-defective), derived from the ANNs trained for ftr_j and ftr_k. *GRANUM* is applied once to the defective subset $\mathcal{D}_2^+$ from the data $\mathcal{D}_2$ and once to the non-defective subset $\mathcal{D}_2^-$ in order to discover the *GRARs* that characterize the defective software entities and non-defective entities, respectively. Thus, the output from the mining phase comprises two sets of interesting *GRARs*, i.e., $GRAR_+$ and $GRAR_-$, based on which the discriminative sets $(GRAR_+ \setminus (GRAR_+ \cap GRAR_-)$ and $GRAR_- \setminus (GRAR_+ \cap GRAR_-))$ are used as classification criteria.

The training step for *HyGRAR* is summarized in algorithm $HyGRAR - Train$.

Algorithm $HyGRAR - Train$ **is:**

Input:

- pre-processed data sets $\mathcal{D}_1$ and $\mathcal{D}_2$.

Output:

- $GRAR_+$ set of interesting GRARs that hold over the *defective* instances from $\mathcal{D}_2$
- $GRAR_-$ set of interesting GRARs that hold over the *non-defective* instances from $\mathcal{D}_2$

Begin

1. $Relations \leftarrow \emptyset$
   For each balanced data set $d \in \mathcal{D}_1$ do
    For each distinct pair (sm_1, sm_2) of software metrics from $\mathcal{F}$ do
     Train two ANNs to express two *gradual relations* R_1 and R_2 between sm_1 and sm_2
     $Relations \leftarrow Relations \bigcup \{R_1, R_2\}$
    EndFor
   EndFor
2. Keep the defective entities from $\mathcal{D}_2$ in $\mathcal{D}_2^+$
   Keep the non-defective entities from $\mathcal{D}_2$ in $\mathcal{D}_2^-$
   $GRAR_+ \leftarrow$ set of interesting GRARs from $\mathcal{D}_2^+$
   $GRAR_- \leftarrow$ set of interesting GRARs from $\mathcal{D}_2^-$

End $HyGRAR - Train$

It should be noted that we use disjoint data sets in the two phases of the training process, i.e., $\mathcal{D}_1 \cap \mathcal{D}_2 = \emptyset$, which is useful to prevent the *HyGRAR* model from overfitting the training data.

5.4. Classification

After the *HyGRAR* classification model has been built as described in Section 5.3, during testing, a new *f*-dimensional software entity $e_{test} = (e_{test}^1 \cdots e_{test}^f)$ (characterized by the same software metrics as the training data sets *X* and *Y*) is received and it must be classified as *defective* or *non-defective*.

HyGRAR classifies the current test instance e_{test} according to the following method. For each of the "+" and "−" classes, from the learned sets of *GRARs* $GRAR_+$ and $GRAR_-$, we keep only the most interesting *noRules* rules, i.e., the rules with membership degrees that are among the largest memberships (*noRules* is a parameter of the *HyGRAR* classifier). Thus, we sort and filter the rules from $GRAR_+$ and $GRAR_-$ to obtain two sets of *noRules gradual association rules*: $R_1^+, R_2^+, \cdots R_{noRules}^+$ and $R_1^-, R_2^-, \cdots R_{noRules}^-$, respectively. Intuitively, the rules from $GRAR_+$ are representative for the "+" class, whereas the rules from $GRAR_-$ are representative for the "−" class, thereby making them useful for differentiating between the *defective* and *non-defective* classes.

For the test instance e_{test}, we compute two values to indicate the distance of e_{test} from the target classes, where $diff^+(e_{test})$ expresses the dissimilarity degree of the instance relative to the positive class and $diff^-(e_{test})$ denotes the dissimilarity degree of the instance with respect to the negative class. The dissimilarity of a test instance relative to a class considers the differences between the memberships of the representative GRARs for that class and the memberships computed for the test instance e_{test} relative to these representative rules. If $diff^+(e_{test})$ is smaller than $diff^-(e_{test})$, then the query instance is likely to be closer to "+" class and it is classified as a *positive* instance; otherwise, it is classified as a *negative* instance.

Formulae 4 and 5 define $diff^{+}(e_{test})$ and $diff^{-}(e_{test})$, where m_i^{class} is the membership of the rule R_i^{class} described in Definition 1, $\forall 1 \leq i \leq noRules$, and $class \in \{+, -\}$.

$$diff^{+}(e_{test}) = \frac{\sum_{i=1}^{noRules} |m_i^{+} - \mu_{R_i^{+}}(e_{test})|}{noRules} \tag{4}$$

In Formula (4), R_i^{+} represents, $\forall 1 \leq i \leq noRules$, the ith *GRAR* from the filtered set $GRAR_{+}$ of rules characterizing the defective instances.

$$diff^{-}(e_{test}) = \frac{\sum_{i=1}^{noRules} |m_i^{-} - \mu_{R_i^{-}}(e_{test})|}{noRules} \tag{5}$$

In Formula (5), R_i^{-} represents, $\forall 1 \leq i \leq noRules$, the ith *GRAR* from the filtered set $GRAR_{-}$ of rules characterizing the non-defective instances.

The classification step in *HyGRAR* is shown in Algorithm $HyGRAR - Classify$.

Algorithm $HyGRAR - Classify$ **is:**

Input:

- $GRAR_{+}$ set of interesting GRARs that hold over the *defective* class
- $GRAR_{-}$ set of interesting GRARs that hold over the *non-defective* class
- e_{test} instance that needs to be classified
- $noRules$ number of relevant GRARs considered for classification

Output:

- $class$ predicted class for e_{test}

Begin

```
Sort the rules from GRAR+ in decreasing order of
 their membership
For i ← 1, noRules do
  R_i^+ ← the ith rule from GRAR+
EndFor
Sort the rules from GRAR- in decreasing order of
 their membership
For i ← 1, noRules do
  R_i^- ← the ith rule from GRAR-
EndFor
Compute diff^+(e_test) according to Formula (4)
Compute diff^-(e_test) according to Formula (5)
If diff^+(e_test) < diff^-(e_test), then
      class ← +
else
      class ← -
EndIf
```

End $HyGRAR - Classify$

The *HyGRAR* classifier is depicted in Fig. 1.

6. Computational experiments

Our experimental evaluation of the *HyGRAR* classifier for detecting faulty software entities is presented in the following. We performed the evaluation based on the open-source data sets used in previous software defect prediction studies.

6.1. Data sets

Ten open source data sets were used in our experiments. The first five data sets comprised publicly available data sets [41] called Ar1, Ar3, Ar4, Ar5, and Ar6. These data sets were obtained from embedded software implemented in C, which is a Turkish white-goods manufacturer [27]. They contained functions and methods represented as 29-dimensional vectors, which comprised the values for different McCabe and Halstead software metrics [14]. For each software entity in the data sets, the class label denoting whether the entity was *defective* or *non-defective* was known. We reduced the dimensionality of the feature set characterizing the software entities based on the analysis carried out in a previous study [27] of the Ar

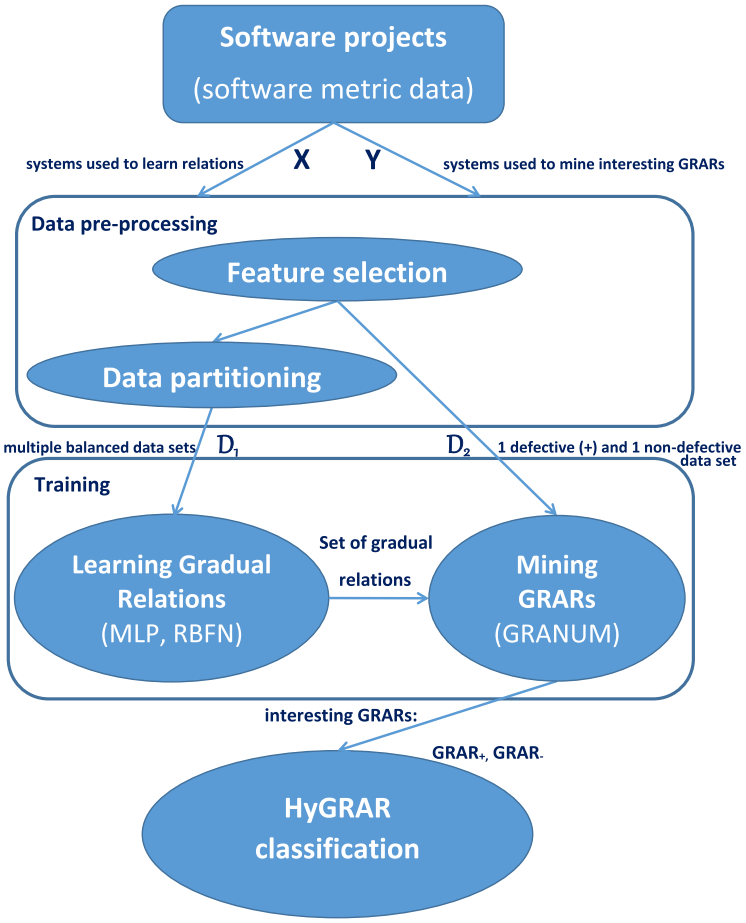

Fig. 1. *HyGRAR* classifier.

data sets to identify the relevant software metrics for software defect prediction. In a previous study [27], the importance of the software metrics employed to describe the Ar1, Ar3, and Ar4 data sets was determined using the *information gain* (IG) measure. Thus, we considered the software metrics with the maximum IG measures. The first three software metrics with the highest IG values (larger than 0.22) comprised *halstead_vocabulary*, *unique_operators*, and *unique_operands*, which we used as features for defect prediction in the Ar data sets. From a software engineering perspective, the selected software metrics are mutually dependent. The statistical dependence between the software metrics was confirmed by a *Chi-square* test performed using each pair of features in a data set comprising all of the Ar data sets. For all the pairs of features, the *p*-values obtained were smaller than 0.00001, thereby confirming the statistical dependence (relationship) of the features at a significance level of 0.01. Despite the degree of dependence between the selected features, it was shown previously [27] that they were highly correlated with the target classification output (i.e., *defective* or *non-defective* class).

In addition to the Ar data sets, we considered five publicly available data sets extracted from Java software systems: JEdit (versions 4.0, 4.2 and 4.3), Ant (version 1.7), and Tomcat (version 6.0) [41]. We selected version 1.7 of the Ant data set because most previous studies used this version. However, a single version of the JEdit data set was not used in most previous studies, so we considered the versions of JEdit that have been employed most frequently (versions 4.0, 4.2, and 4.3). A single version of the Tomcat data set (6.0) was used in all previous studies. The software entities from JEdit, Ant, and Tomcat were characterized by 20 object-oriented software metrics and they were labeled with the number of bugs. The number of bugs was initially transformed into a binary feature denoting whether the entity was faulty or not. In order to reduce the dimensionality of the data sets (i.e., the number of software metrics), we considered the conclusions of a previous study [36] regarding the suitability of different software metrics for defect prediction. Among the software metrics reported previously as strongly correlated with the existence of software defects, only three were found in our data sets. Thus, we decided to retain only these three metrics in our experiments: WMC, coupling between objects (CBO), and RFC. WMC sums the complexities of the methods from a class, CBO counts the number of classes that are coupled to a class, and RFC represents the size of the response set for a class. Thus, there is a degree of dependence between all three of these software metrics. A *Chi-square* test was performed for each pair of features. For (WMC, RFC) and (CBO, RFC), the *p*-values

Table 3
Descriptions of the data sets used in the evaluation.

Data set	#Defects	#Non-defects	%Defects	Difficulty
Ar1	9	112	7.4%	0.666
Ar3	8	55	12.7%	0.625
Ar4	20	87	18.69%	0.7
Ar5	8	28	22.22%	0.375
Ar6	15	86	14.85%	0.666
JEdit 4.0	75	231	24.5%	0.69
JEdit 4.2	48	319	13.08%	0.666
JEdit 4.3	10	482	2.03%	0.9
Ant 1.7	166	579	22.28%	0.572
Tomcat 6.0	77	781	8.97%	0.818

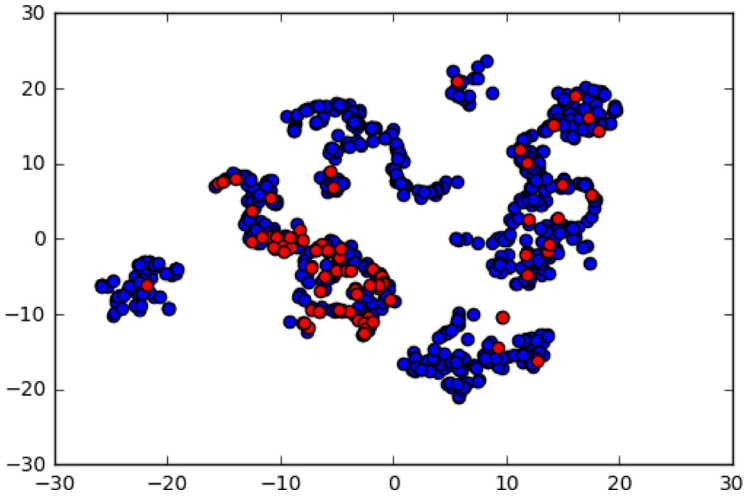

Fig. 2. t-SNE plot for the Tomcat data set.

obtained were smaller than 0.00001, thereby confirming the statistical dependence (relationship) between these two pairs of features at a significance level of 0.01. There was no statistical dependence between WMC and CBO at a significance level of 0.01, where the *p*-value obtained was 0.95. Despite the degree of dependence between WMC, CBO, and RFC, a previous study [36] demonstrated that they are effective for defect prediction.

Table 3 describes the data sets used in our experiments. For each data set corresponding to a software system, the table shows the number of *defective* and *non-defective* entities, the percentage of software *defects*, and the system's difficulty. The *difficulty* [6] of a data set is computed as the proportion of defective instances for which the nearest neighbor is a non-defective instance. The *difficulty* of a data set is a measure of how challenging the classification task is based on that data set. When the *difficulty* of a defect data set is higher, it is more difficult to distinguish between its *defective* and *non-defective* software entities.

Table 3 clearly shows that all of the data sets were highly imbalanced, where the number of *defective* instances was much smaller than the number of *non-defective* entities. Thus, the supervised classification task was difficult because the classifier was trained to recognize mostly non-defective entities, thereby making it biased toward classifying the entities as *non-defective*, i.e., labeling them as belonging to the majority class. Table 3 shows that JEdit 4.3 and Tomcat were the most complex data sets from a defect classification viewpoint because their *difficulty* was high and the proportion of defects in the data set was very small. Fig. 2 illustrates the difficulty of the software defect prediction task for the Tomcat system based on a two-dimensional representation of the data obtained using T-distributed stochastic neighbor embedding (t-SNE) [24]. t-SNE is a non-linear dimensionality reduction method for visualizing high-dimensional data, which maps multi-dimensional data onto two or more dimensions, and it better reflects the initial structure of the data than the Principal Component Analysis (PCA) method [24]. The *x*-axis and *y*-axis in Fig. 2 represent the first and the second dimensions, respectively, of the Tomcat data set reduced to two dimensions using t-SNE.

Fig. 2 emphasizes the complexity of detecting defects in the Tomcat data set, where it is very difficult to distinguish between *defects* and *non-defects* (the non-defects are marked in *blue* and the defects in *red*). Despite the t-SNE graph was obtained only for the Tomcat data set, the situation was similar for all of the data sets used in our experiments.

Table 4
Case studies.

Case study	X
Ar1	{Ar3, Ar4, Ar5, Ar6}
Ar3	{Ar1, Ar4, Ar5, Ar6}
Ar4	{Ar1, Ar3, Ar5, Ar6}
Ar5	{Ar1, Ar3, Ar4, Ar6}
Ar6	{Ar1, Ar3, Ar4, Ar5}
JEdit 4.0	{JEdit 3.2, JEdit 4.1, JEdit 4.2, JEdit 4.3, Ant 1.7, Tomcat 6.0}
JEdit 4.2	{JEdit 3.2, JEdit 4.0, JEdit 4.1, JEdit 4.3, Ant 1.7, Tomcat 6.0}
JEdit 4.3	{JEdit 3.2, JEdit 4.0, JEdit 4.1, JEdit 4.2, Ant 1.7, Tomcat 6.0}
Ant 1.7	{JEdit 3.2, JEdit 4.0, JEdit 4.1, JEdit 4.2, JEdit 4.3, Tomcat 6.0}
Tomcat 6.0	{JEdit 3.2, JEdit 4.0, JEdit 4.1, JEdit 4.2, JEdit 4.3, Ant 1.7}

Table 5
MLP parameter settings.

Model preferences	
Activation function	Sigmoid
Number of input units	2
Number of hidden layers	1
Number of hidden units	2
Number of output units	2
Learning preferences	
Learning rate	0.1
Momentum term	0.1
Decay factor	0.1
Validation set proportion	0.2

6.2. Training and testing

As described in Section 5, two distinct sets *X* and *Y* were provided to our *HyGRAR* classifier. Both *X* and *Y* comprised a set of software systems, which were pre-processed as explained in Section 5.2 and then used for training. The pre-processed set *X* was used for learning the *gradual relations* and the pre-processed set *Y* was employed for the *GRAR mining* process. Table 4 shows the training set *X* for each example. For the JEdit software, in addition to the versions used in our evaluation (4.0, 4.2, and 4.3), we found two additional versions (3.2 and 4.1) and we decided to use these additional versions in the *HyGRAR* training step for the JEdit, Ant, and Tomcat case studies.

In order to evaluate the performance of our *HyGRAR* classifier, we employed the LOO cross-validation method [47] in each case study. When applying LOO to a data set with *n* instances, the *HyGRAR* model was trained based on *n*-1 instances and then tested with the remaining instance. This process was repeated *n* times for each instance in the data set.

In each step of the LOO cross-validation process, the instances excluding the current test example comprised the training set *Y* used for *GRAR* mining, which was split into defective and non-defective subsets. The *GRANUM* algorithm separately mined the interesting binary *GRARs* from the two subsets of *Y*.

The confusion matrix was computed during the cross-validation process, where the *precision, specificity*, and *sensitivity* measures were calculated based on this matrix. A suitable measure for evaluating the performance of the software defect classifiers is the *AUC,Fawcett*, which is a measure that must be maximized in order to obtain better defect predictors. In general, the AUC measure is employed for approaches that yield a single value, which is then converted into a class label using a threshold. Thus, for each threshold value, the point $(1 - specificity, sensitivity)$ is represented on a plot and the AUC is computed as the area under this curve. For the approaches where the direct output of the defect classifier was the class label, there is only one $(1 - specificity, sensitivity)$ point, which is linked to the (0, 0) and (1, 1) points, and the AUC measure representing the area under the trapezoid is computed using Formula (6).

$$AUC = \frac{sensitivity + specificity}{2} \tag{6}$$

6.3. Results

The *HyGRAR* classifier proposed in Section 5 was applied in the case studies described in Section 6.1. Each of the 10 case studies was tested using LOO cross-validation with the HyGRAR model constructed based on the training data (Section 6.2), which were pre-processed as described in Section 5.2.

The MLP and RBFN configurations, and the learning parameters are shown in Tables 5 and 6, respectively.

Fig. 3 shows convergence curves for the *mean square error* (MSE) vs. epochs based on both the training and validation data. The curves in Fig. 3 were drawn while learning the *gradual relations* based on one balanced subset from *Ant 1.7*.

Table 6
RBFN parameter settings.

Model preferences	
Radial function	Gaussian
Number of radial centers	2
Learning preferences	
α	0.5
p	1

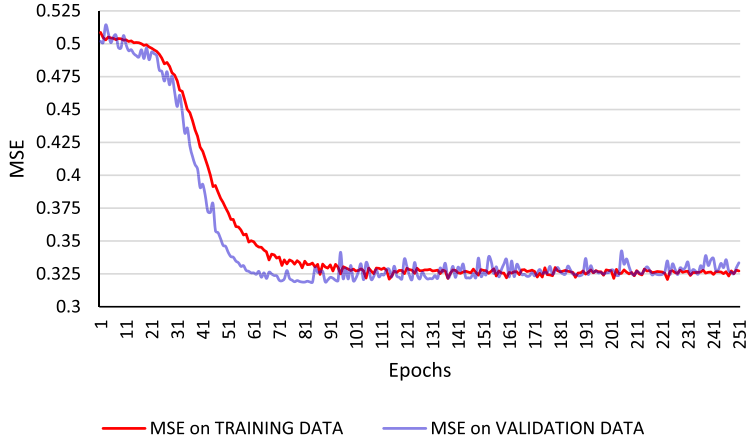

Fig. 3. Curves of *MSE* vs. epochs during the training of the MLP (i.e., learning the *gradual relations*) based on one balanced subset from *Ant 1.7*.

Table 7
Performance of HyGRAR with the 10 data sets.

Case study	TP	FP	TN	FN	Acc	Sens	Spec	Prec	AUC	FPR	FNR	OE
Ar1	5	1	111	4	0.959	0.556	0.991	0.833	**0.773**	0.009	0.444	0.041
Ar3	8	11	44	0	0.825	1.000	0.800	0.421	**0.900**	0.200	0.000	0.175
Ar4	18	1	86	2	0.972	0.900	0.989	0.947	**0.944**	0.012	0.100	0.028
Ar5	7	1	27	1	0.944	0.875	0.964	0.875	**0.920**	0.036	0.125	0.056
Ar6	10	1	85	5	0.941	0.667	0.988	0.909	**0.828**	0.012	0.333	0.059
JEdit 4.0	54	1	230	21	0.928	0.72	0.996	0.982	**0.858**	0.004	0.280	0.072
JEdit 4.2	39	2	317	9	0.970	0.813	0.994	0.951	**0.903**	0.006	0.188	0.030
JEdit 4.3	7	3	478	4	0.986	0.636	0.994	0.700	**0.815**	0.006	0.364	0.014
Ant 1.7	142	2	577	24	0.965	0.855	0.997	0.986	**0.926**	0.004	0.145	0.035
Tomcat 6.0	65	2	779	12	0.984	0.844	0.997	0.970	**0.921**	0.003	0.156	0.016

In the second phase where the *GRANUM* algorithm mined the interesting *GRARs*, we used a minimum confidence threshold of 0.5 and a minimum support threshold of 1.

The parameter *noRules* used by the *HyGRAR* classifier for determining the number of *GRARs* employed for distinguishing between defects and non-defects was set to 3.

Table 7 summarizes the experimental results obtained by *HyGRAR*. The confusion matrix is indicated for each data set. Several suitable evaluation measures for expressing the performance of a classifier were also computed based on the values in the confusion matrix: accuracy (Acc), sensitivity (Sens), specificity (Spec), precision (Prec), false positive rate (FPR), false negative rate (FNR), and overall error (OE). All of these evaluation measures range from 0 to 1. The measures shown in the last three columns had to be minimized to obtain a better classifier, whereas larger values corresponded to better defect predictors for the other measures.

The results presented in Table 7 show that the AUC values were very good. Except for the Ar1 data set, all of the AUC values were above 0.8, thereby indicating the very good performance of *HyGRAR*. The ROC curves obtained for our *HyGRAR* classifier using the data sets are presented in Fig. 4. For each data set, the point on the ROC curve in Fig. 4 connected to (0, 0) and (1, 1) denotes the (1-*Spec, Sens*) point, where *Spec* and *Sens* represent the *specificity* and *sensitivity* values for the corresponding predictor, respectively.

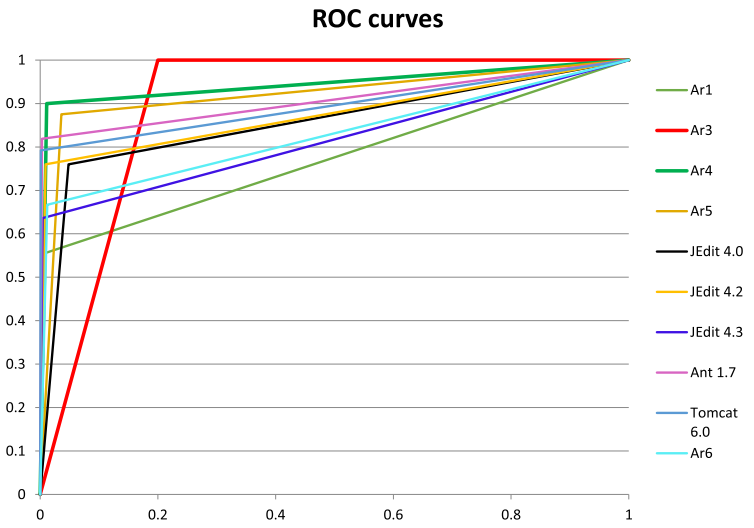

Fig. 4. ROC curves obtained for the data sets considered in this study.

7. Discussion

In the following, we analyze the experimental results obtained after applying the *HyGRAR* classifier to detect software defects in the case studies described in Section 6.1. We also compare the results produced using *HyGRAR* with the results obtained with similar approaches proposed in previous studies.

7.1. Analysis of HyGRAR

In this study, we developed a novel hybrid classification method *HyGRAR* for *software defect prediction*. *HyGRAR* is based on identifying the *gradual relations* between software metrics that characterize *defective* and *non-defective* software entities. The *gradual relations* learned using neural networks are used to discriminate between *non-faulty* and *faulty* entities.

The training of the *HyGRAR* classifier proposed in Section 5 is slow because it is an eager classifier, but the classification step is very fast. In terms of the time complexity required for training *HyGRAR*, we note that it is influenced greatly by **Phase 2**, which involves mining GRARs that characterize the *defective* and *non-defective* software entities. The *GRANUM* algorithm searches efficiently for interesting GRARs but its time complexity is still exponential with respect to the number of features that characterize the mined data set (the number of features is f in our method, which represents the number of software metrics identified after the feature selection process). Parallel implementations would be useful in both phases of the training process in order to reduce the training time for *HyGRAR*. Training multiple neural networks in parallel would be effective for learning the gradual relations and a distributed solution would be efficient for mining the GRARs. The training process is computationally expensive, but the classification step of *HyGRAR* is very fast with a linear time complexity of $\theta(noRules)$, where *noRules* represents the number of *gradual association rules* used for classification.

In order to experimentally evaluate our *HyGRAR* classifier, we selected 10 open-source data sets that have often been used in previous software defect prediction studies. Among the data sets considered as case studies, five were Ar data sets (Ar1, Ar3, Ar4, Ar5, and Ar6) and three were data sets extracted from Java software systems: JEdit (versions 4.0, 4.2, and 4.3), Ant (version 1.7), and Tomcat (version 6.0). We selected several software metrics as features for characterizing the software entities: *halstead_vocabulary*, *unique_operators*, and *unique_operands* for the Ar data sets, and WMC, CBO, and RFC for the JEdit, Ant, and Tomcat systems. Previous studies [27,36] have demonstrated that the selected software metrics are highly correlated with the defect prediction task. From a statistical perspective, the features have a certain degree of dependence, but this does not significantly affect the classification accuracy of HyGRAR. Previous studies have shown that even highly correlated features can provide valuable information that may be useful for classification. In the future, we will investigate how to automatically extract features (software metrics) using *deep learning* architectures, instead of using engineered ones.

In the experiments, we tested the learned hypothesis using the LOO cross-validation method. The AUC measure was used to estimate the performance of the proposed *HyGRAR* classifier due to the highly imbalanced nature of the software defect data. The AUC values obtained from the case studies indicated the very good performance of *HyGRAR*. Overall, out of 10 AUC values, six were over 0.9, three ranged between 0.8 and 0.9, and only one was between 0.75 and 0.8. An average AUC value of 0.879 was obtained for our *HyGRAR* classifier, which indicates very good performance, especially in the difficult task of *defect prediction* based on highly imbalanced training data.

The main challenge of software defect prediction is minimizing the number of defective software entities that are misclassified, particularly decreasing the *FNR*, which is equivalent to maximizing the *sensitivity* because an undetected software

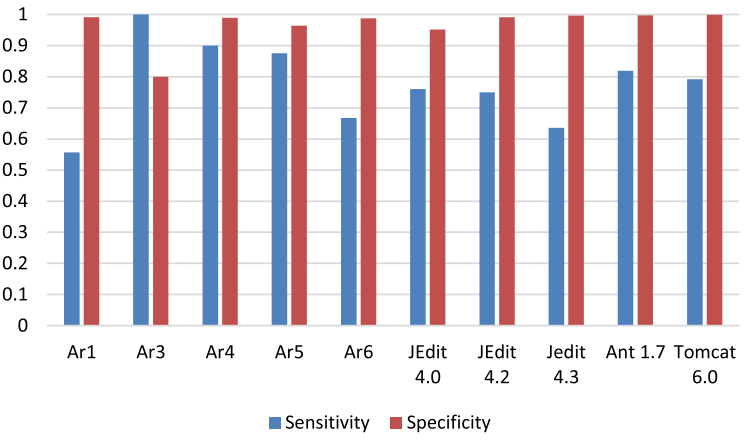

Fig. 5. Plot of *specificity* vs. *sensitivity* for *HyGRAR* based on each of the case studies.

defect can have serious consequences during software deployment. It is more dangerous to have false negatives than false positives in the software defect prediction problem. The misclassification of a non-defect would lead to time being wasted systematically testing a fault-free entity, which is not as dangerous as the alternative situation. Thus, increasing the *sensitivity* of a defect predictor is more desirable than increasing its *specificity*. The AUC measure combines both *specificity* and *sensitivity* measures, and thus it can be considered the best evaluation measure for representing the defect prediction performance.

Fig. 5 plots the *sensitivity* vs. *specificity* obtained for *HyGRAR* based on each data set. In each case, the *specificity* values are sufficiently large and they exceed the *sensitivity* values. However, sensitivity values larger than 0.6 were obtained, except with the Ar1 data set. In six cases, our *HyGRAR* classifier obtained a sensitivity value above 0.8, which corresponds to a very good *FPR*, where this is desirable for defect predictors.

In addition to its superior performance in experiments, *HyGRAR* has various other advantages, as summarized in the following. *HyGRAR* is a two-phase learning hybrid model so it benefits from the ability of the ANN to detect complex nonlinear relations between variables and the capacity of the mining algorithm to discover the exact rules that characterize the mined data starting with the learned relations. From a software engineering perspective, this avoids limiting the defect prediction model by predefining *gradual relations* that may be irrelevant, or by using artificially selected membership functions, but instead it can learn them automatically based on the experience of past projects or past versions of the current project. Furthermore, the mining phase discriminates between all of these automatically learned relations to focus only on the *GRARs* that are specifically important for the software system or version under analysis. A drawback of *HyGRAR* is that the readability of the classifications rules is lost, despite the adaptability gained by using ANNs. A second limitation is the classification methodology which is based on predefined membership values. In order to avoid this issue, we propose the use of supervised *learning* for the classification methodology as a future improvement to *HyGRAR*. An additional indirect limitation of the model may be the inadequate relevance of the software metrics used for prediction. We propose to address this limitation by automatically extracting relevant software characteristics from software artifacts.

7.2. Comparisons with related methods

The AUC evaluation measure is widely considered the best measure for comparing defect predictors [16], so it was used to compare *HyGRAR* with similar software defect prediction approaches literature.

Tables 8 and 9 compare the AUC values obtained by our *HyGRAR* classifier and the values obtained in previous studies using similar approaches, as described in Section 4. Table 8 shows the results obtained with the Ar data sets, and Table 9 provides the AUC values produced with JEdit, Ant, and Tomcat. We use "–" to indicate the approaches that did not provide results for a particular data set. Some studies did not report the value of the AUC measure, but instead they presented only the confusion matrix or other evaluation measures such as *FPR* and *FNR*. Thus, if possible, we computed the AUC based on these measures using Formula (6). The highest AUC values are indicated in bold in the tables.

In Tables 8 and 9, the best AUC value obtained for each data set is highlighted and the best two values are underlined. We conducted a one-tailed *t*-test [39] to test the alternative hypothesis that the mean AUC with *HyGRAR* was greater than the mean AUC using each of the other approaches (presented in Tables 8 and 9) that obtained results based on at least three of the data sets. Before conducting the *t*-test, an F-test was applied to test the variance of the samples. In each row of Tables 8 and 9, the last column depicts the *p*-value obtained from the *t*-test to compare *HyGRAR* and the other corresponding method in that row.

Table 8
Comparison with other methods based on the Ar data sets. The AUC value is shown for each approach. In each row, the last column depicts the *p*-value obtained from the *t*-test conducted to compare HyGRAR and the alternative corresponding method in that row.

Approach	Ar1	Ar3	Ar4	Ar5	Ar6	*p*-value
Our HyGRAR	0.773	**0.9**	0.944	**0.92**	**0.828**	
SOM with threshold [1]	–	0.88	**0.95**	0.84	–	0.2202
K-means with quad-trees [5]	–	0.70	0.75	0.87	–	0.0456
Clustering Xmeans [35]	–	0.84	0.69	0.86	–	0.0696
Clustering EM [35]	–	0.82	0.69	0.80	–	0.0268
Clustering Xmeans [9]	–	0.70	0.75	0.87	–	0.0456
K-means [42]	–	0.834	0.292	0.92	0.628	0.0784
Genetic programming [2]	0.530	0.67	0.65	0.67	0.630	0.0002
Multiple linear regression [2]	0.550	0.61	0.62	0.55	0.590	0.0001
Binary logistic regression [45]	0.551	0.87	0.73	0.39	0.722	0.0189
Logistic regression [32]	0.734	0.82	0.82	0.91	0.640	0.0778
Logistic regression [26]	0.494	–	–	–	0.538	–
Artificial neural networks [26]	0.711	–	–	–	0.774	–
Support vector machines [26]	0.717	–	–	–	0.721	–
Cascade correlation networks [26]	**0.786**	–	–	–	0.758	–
GMDH network [26]	0.744	–	–	–	0.702	–
Gene expression programming [26]	0.547	–	–	–	0.688	–
Simple cart [18]	0.500	0.732	0.685	0.777	0.500	0.0060

Table 9
Comparisons with results obtained using other methods based on the JEdit, Ant, and Tomcat data sets. The AUC value is depicted for each approach. In each row, the last column shows the *p*-value obtained from the *t*-test conducted to compare HyGRAR and the alternative corresponding method in that row.

Approach	JEdit 4.0	JEdit 4.2	JEdit 4.3	Ant 1.7	Tomcat 6.0	*p*-value
Our HyGRAR	**0.858**	**0.903**	**0.815**	**0.926**	**0.921**	
Fuzzy decision tree - original [28]	–	0.666	–	0.646	–	–
Fuzzy decision tree - enhanced [28]	0.7	0.735	–	0.707	–	0.0019
Weka - decision tree [18]	0.695	0.455	0.5	0.629	0.577	0.0004
Multivariate logistic Regression [25]	–	–	–	0.829	–	–
Multiobjective [8]	(0.13, 0.33) 0.305	–	–	(0.51, 0.39) 0.641	(0.31, 0.18) 0.570	0.0096
	(0.18, 0.64) 0.346	–	–	(0.43, 0.77) 0.739	(0.23, 0.82) 0.775	0.0565
	(0.18, 0.54) 0.369	–	–	(0.43, 0.43) 0.633	(0.30, 0.58) 0.723	0.0197
Bayesian networks [33]	–	0.732	0.658	0.820	0.766	0.0076
Logistic [34]	0.77	–	–	0.81	0.81	0.0109
RBFNetwork [34]	0.71	–	–	0.74	0.76	0.0026
Alternative decision tree (ADTree) [34]	0.74	–	–	0.75	0.79	0.0041
DecisionTable [34]	0.78	–	–	0.75	0.78	0.0072
MultiLayer perceptron [34]	0.78	–	–	0.81	0.82	0.0137
BayesNet [34]	0.49	–	–	0.65	0.51	0.0015
$CODEP_{Log}$ [34]	0.85	–	–	0.88	0.87	0.1233
$CODEP_{Bayes}$ [34]	0.84	–	–	0.86	0.87	0.0826

According to Table 8, the results obtained using the multiple linear regression and genetic programming approaches by Afzal et al. [2] were the best values reported. Varade and Ingle [42] obtained the FPR and OE values by applying their approach to the Ar3–Ar6 data sets. Based on the values reported, we computed the AUC values using formula 6.

Malhotra [26] reported the results obtained by applying the SimpleCart implementation from Weka software [18] based on the Ar1 and Ar6 data sets. Using the LOO methodology, we tested SimpleCart with all of the Ar data sets and computed the AUC measures (Formula (6)) based on the reported confusion matrix. The last line in Table 8 shows the AUC values obtained.

Most of the previous approaches that tested the Ant data set used version 1.7, but a common version of the JEdit data set was not considered in most previous studies. Therefore, as shown in Table 9, we tested three versions (4.0, 4.2, and 4.3) for the JEdit data set. A single version of the Tomcat data set (6.0) was used in all previous studies.

The third line in Table 9 shows the results obtained using the decision tree (DT) classifier in Weka freely available machine learning software [18]. The results for JEdit 4.2 and Ant 1.7 were reported by Marian et al. [28]. For the remaining data sets, we applied the DT classifier in Weka and computed the AUC measure according to Formula (6) based on the reported confusion matrix.

Canfora et al. [8] reported different (precision, recall) pairs for the proposed multi-objective approach. Thus, based on the reported precision and recall values, we obtained the values from the confusion matrix and the AUC measure values.

Table 10
Performance evaluation of *HyGRAR* based on the AUC measure.

	Ar1	Ar3	Ar4	Ar5	Ar6	JEdit 4.0	JEdit 4.2	JEdit 4.3	Ant 1.7	Tomcat 6.0
Average from related work	0.624	0.770	0.693	0.769	0.658	0.644	0.647	0.579	0.743	0.740
Our *HyGRAR*	0.773	0.900	0.944	0.920	0.828	0.858	0.903	0.815	0.926	0.921
Winner	No	**Yes**	No	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
Top 2	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**

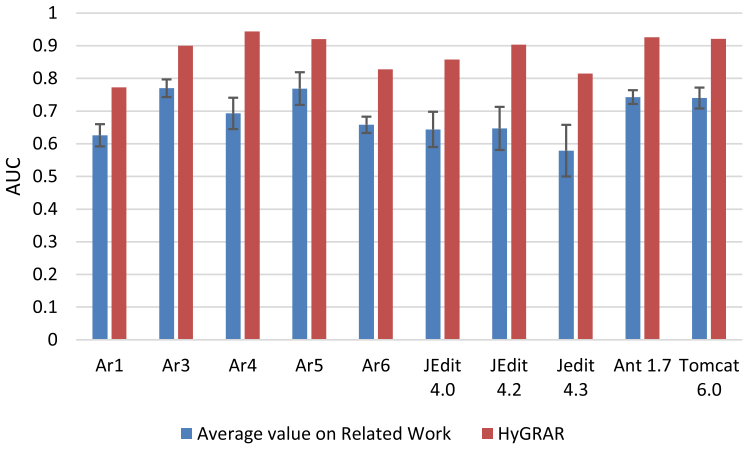

Fig. 6. Performance of *HyGRAR* based on the AUC measure. Standard error bars are shown for the average AUC measures reported in previous studies.

We also indicate the corresponding (precision, recall) pair for the computed AUC values. Canfora et al. [8] used version 4.0 of JEdit, so we also applied our *HyGRAR* model to JEdit 4.0.

Okutan and Yildiz [33] reported the AUC values obtained using JEdit version 4.3, which is why we considered the AUC values reported by Marian et al. [28] for JEdit version 4.2. Marian et al. [28] applied the Weka implementation of Bayesian networks to reproduce the experiments of Okutan and Yildiz [33] based on JEdit version 4.2, so the AUC values were computed from the confusion matrix.

The CODEP classifier proposed by Panichella et al. [34] was applied to the JEdit 4.0, Ant 1.7, and Tomcat 6.0 data sets in the context of cross-project defect prediction. Several stand-alone machine learning techniques (including MLP, Bayes Net, and RBFN) and their combinations based on logistic regression ($CODEP_{Log}$) and Bayesian networks ($CODEP_{Bayes}$) were tested by Panichella et al. [34].

The results in Table 8 show that the *HyGRAR* approach had the highest AUC values for the Ar3, Ar5, and Ar6 data sets, and the second highest values for the Ar1 and Ar4 data sets. Even with the hardest data sets, i.e., Ar1 and Ar4, the AUC values obtained using our approach were very close to the best AUC values reported in previous studies with these data sets (0.773 vs. 0.786 for Ar1 and 0.944 vs. 0.95 for Ar4). Table 9 also highlights the very good performance of *HyGRAR* with the JEdit, Ant and Tomcat data sets. The AUC values obtained by our classifier were the highest values compared with other studies using these data sets.

The overall performance of *HyGRAR* in all 10 case studies is summarized in Table 10. For each data set, the first line represents the average AUC value obtained in other studies excluding our approach. The next two lines in Table 10 show whether the *HyGRAR* classifier was the best performer or if it was ranked in the best two compared with other methods based on the AUC measure.

The results showed that among the 10 case studies, *HyGRAR* was the best classifier of eight data sets and it was one of the top two classifiers for the other two data sets. *HyGRAR* performed better than previously proposed approaches in 102 cases and it only achieved an inferior AUC measure than other methods in only 2 cases, i.e., it produced better AUC values than those reported in previous studies in 98% of the cases. For each case study, the AUC measure obtained by *HyGRAR* was significantly better than the average AUC values achieved by previously proposed defect detectors, thereby indicating the very high efficiency of the *HyGRAR* classifier. Fig. 6 shows the average AUC value reported in previous studies (bar on the left-hand side) and the AUC value obtained by *HyGRAR* (bar on the right-hand side) for each case study considered in our evaluation. Standard errors bars are shown for the average AUC values reported in previous studies.

According to Tables 8 and 9, we make the following observations regarding the statistical significance of the comparisons with previously proposed methods. In 92% of the cases (10 out of 11 comparisons for the Ar data sets, and 13 out of 14 comparisons for the JEdit, Ant, and Tomcat data sets), the p-value was less than $\alpha = 0.1$, i.e., the mean performance of HyGRAR was better than the mean performance of other approaches (in terms of AUC) at a significance level of 0.1.

Considering the analyses given in Sections 7.1 and 7.2, we conclude that our proposed hybrid classifier is effective for predicting software defects. Future improvements to *HyGRAR* may yield even better results.

8. Conclusions and future work

In this study, we proposed a hybrid supervised learning model called *HyGRAR* to solve the software defect prediction problem. *HyGRAR* is a combination of *GRARs* and *ANNs*. The proposed model is novel from the perspectives of both search-based software engineering and data mining. We demonstrated the potential of our proposed method in computational experiments based on 10 open source data sets, where the results obtained showed that *HyGRAR* performed better than similar previously proposed methods in most cases.

We plan to extend our experimental evaluation of *HyGRAR* by considering additional case studies based on real software systems in order to better demonstrate its effectiveness. We also aim to expand the *HyGRAR* model by considering additional machine learning methods for learning the *gradual relations* such as support vector machines,mitchell. Other improvements that could enhance *HyGRAR* include learning (using ANNs or support vector machines) the classification methodology based on the membership values, thereby avoiding predetermining them, and automatically extracting software characteristics (e.g., by applying unsupervised learning methods such as Doc2Vec based on the source code) as substitutes for the software metrics.

Acknowledgments

The authors would like to thank the editor and the anonymous reviewers for their valuable comments and suggestions, which improved our paper and its presentation.

References

[1] G. Abaei, Z. Rezaei, A. Selamat, Fault prediction by utilizing self-organizing map and threshold, in: 2013 IEEE International Conference on Control System, Computing and Engineering (ICCSCE), 2013, pp. 465–470.
[2] W. Afzal, R. Torkar, R. Feldt, Resampling methods in software quality classification, Int. J. Softw. Eng. Knowl. Eng. 22 (2) (2012) 203–223.
[3] A. Asfaram, M. Ghaedi, M.H.A. Azqhandi, A. Goudarzi, S. Hajati, Ultrasound-assisted binary adsorption of dyes onto mn@ cus/zns-NC-AC as a novel adsorbent: application of chemometrics for optimization and modeling, J. Ind. Eng. Chem. 54 (Supplement C) (2017) 377–388. http://www.sciencedirect.com/science/article/pii/S1226086X17303052.
[4] A.R. Bagheri, M. Ghaedi, A. Asfaram, S. Hajati, A.M. Ghaedi, A. Bazrafshan, M.R. Rahimi, Modeling and optimization of simultaneous removal of ternary dyes onto copper sulfide nanoparticles loaded on activated carbon using second-derivative spectrophotometry, J. Taiwan Inst. Chem. Eng. 65 (Supplement C) (2016) 212–224.
[5] P. Bishnu, V. Bhattacherjee, Software fault prediction using quad tree-based k-means clustering algorithm, IEEE Trans. Knowl. Data Eng. 24 (6) (2012) 1146–1150.
[6] G.D. Boetticher, Advances in machine learning applications in software engineering, in: IGI Global, 2007, Ch. Improving the Credibility of Machine Learner Models in Software Engineering,, pp. 52–72.
[7] H. Borzecka, Multi-criteria decision making using fuzzy preference relations, Oper. Res. Decis. 3 (2012) 5–21.
[8] G. Canfora, A.D. Lucia, M.D. Penta, R. Oliveto, A. Panichella, S. Panichella, Multi-objective cross-project defect prediction, in: Proceedings of the 6th International Conference on Software Testing, Verification and Validation, 2013, pp. 252–261.
[9] C. Catal, U. Sevim, B. Diri, Software fault prediction of unlabeled program modules, in: Proceedings of the World Congress on Engineering (WCE), 2009, pp. 212–217.
[10] R.h. Chang, X. Mu, L. Zhang, Software defect prediction using non-negative matrix factorization, J. Softw. 6 (11) (2011) 2114–2120.
[11] B. Clark, D. Zubrow, Software Engineering Symposium, Carreige Mellon University, 2001, pp. 1–35.
[12] G. Czibula, Z. Marian, I.G. Czibula, Software defect prediction using relational association rule mining, Inf. Sci. 264 (Supplement C) (2014) 260–278.
[13] G. Czibula, I.G. Czibula, D.L. Miholca, Enhancing relational association rules with gradualness, Int. J. Innov. Comput. Commun. Control 13 (1) (2017) 289–305.
[14] I.G. Czibula, G. Czibula, Z. Marian, V.S. Ionescu, A novel approach using fuzzy self-organizing maps for detecting software faults, Stud. Inform. Control 25 (2) (2016) 207–216.
[15] M. Dastkhoon, M. Ghaedi, A. Asfaram, M.H.A. Azqhandi, M.K. Purkait, Simultaneous removal of dyes onto nanowires adsorbent use of ultrasound assisted adsorption to clean waste water: chemometrics for modeling and optimization, multicomponent adsorption and kinetic study, Chem. Eng. Res. Des. 124 (Supplement C) (2017) 222–237.
[16] T. Fawcett, An introduction to ROC analysis, Pattern Recognit. Lett. 27 (8) (2006) 861–874.
[17] A.S. Haghighi, M.A. Dezfuli, S. Fakhrahmad, Applying mining schemes to software fault prediction: a proposed approach aimed at test cost reduction, in: Proceedings of the World Congress on Engineering 2012 Vol I, WCE 2012, IEEE Computer Society, Washington, DC, USA, 2012, pp. 1–5.
[18] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I.H. Witten, The WEKA data mining software: an update, SIGKDD Explor. Newsl. 11 (1) (2009) 10–18.
[19] T. Hall, S. Beecham, D. Bowes, D. Gray, S. Counsell, A systematic literature review on fault prediction performance in software engineering, IEEE Trans. Softw. Eng. 38 (6) (2011) 1276–1304.
[20] Y. Hand, F. Daly, K. McConway, D. Lunn, E. Ostrowski, A Handbook of Small Data Sets, Vol. 1, CRC Press, 1993.
[21] N.A.R. Hayani, S. Hamid, M.L.M. Kiah, S. Shamshirband, S. Furnell, A systematic review of approaches to assessing cybersecurity awareness, Kybernetes 44 (4) (2015) 606–622.
[22] J. Iqbal, R.B. Ahmad, M.H. Nasir, M. Niazi, S. Shamshirband, M.A. Noor, Software smes' unofficial readiness for cmmi®-based software process improvement, Softw. Quality J. 24 (4) (2016) 997–1023.
[23] J. Li, X. Ma, J. Zhang, J. Tao, P. Wang, X. Guan, Mining repeating pattern in packet arrivals: metrics, models, and applications, Inf. Sci. 408 (Supplement C) (2017) 1–22.
[24] L. van der Maaten, G. Hinton, Visualizing data using t-SNE, J. Mach. Learn. Res. 9 (2008) 2579–2605.
[25] R. Malhotra, A defect prediction model for open source software, in: Proceedings of the World Congress on Engineering, Vol II, 2012, pp. 880–884.
[26] R. Malhotra, Comparative analysis of statistical and machine learning methods for predicting faulty modules, Appl. Soft Comput. 21 (2014) 286–297.
[27] Z. Marian, G. Czibula, I.G. Czibula, S. Sotoc, Software Defect Detection using Self-Organizing Maps, Studia Universitatis Babes-Bolyai, Informatica LX, 2015, pp. 55–69.

[28] Z. Marian, I.G. Mircea, I.G. Czibula, G. Czibula, A novel approach for software defect prediction using fuzzy decision trees, in: Proceedings of SYNASC'16, Symbolic and Numeric Algorithms for Scientific Computing, 2016, pp. 1–8.
[29] T. Menzies, J. Greenwald, A. Frank, Data mining static code attributes to learn defect predictors, IEEE Trans. Softw. Eng. 33 (1) (2007) 2–13.
[30] T.M. Mitchell, Machine Learning, McGraw-Hill, Inc. New York, USA, 1997.
[31] S.A. Moosavi, M. Jalali, N. Misaghian, S. Shamshirband, M.H. Anisi, Community detection in social networks using user frequent pattern mining, Knowl. Inf. Syst. 51 (1) (2017) 159–186.
[32] J. Nam, S. Kim, Heterogeneous defect prediction, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, 2015, pp. 508–519.
[33] A. Okutan, O.T. Yildiz, Software defect prediction using bayesian networks, Empir. Softw. Eng. 19 (1) (2014) 154–181.
[34] A. Panichella, R. Oliveto, A.D. Lucia, Cross-project defect prediction models: l'union fait la force, in: IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week, 2014, pp. 164–173.
[35] M. Park, E. Hong, Software fault prediction model using clustering algorithms determining the number of clusters automatically, Int. J. Softw. Eng. Appl. 8 (7) (2014) 199–205.
[36] D. Radjenović, M. Heričko, R. Torkar, A. Živkovič, Software fault prediction metrics: a systematic literature review, Inf. Softw. Technol. 55 (8) (2013) 1397–1418.
[37] Y. Ren, D. Ji, Neural networks for deceptive opinion spam detection: an empirical study, Inf. Sci. 385–386 (Supplement C) (2017) 213–224.
[38] G. Serban, A. Câmpan, I.G. Czibula, A programming interface for finding relational association rules, Int. J. Comput. Commun. Control I (S.) (2006) 439–444.
[39] Social science statistics, http://www.socscistatistics.com/tests/.
[40] S.C. Tan, S. Wang, J. Watada, A self-adaptive class-imbalance TSK neural network with applications to semiconductor defects detection, Inf. Sci. 427 (Supplement C) (2018) 1–17.
[41] Tera-promise repository. http://www.openscience.us/repo/.
[42] S. Varade, M. Ingle, Hyper-quad-tree based k-means clustering algorithm for fault prediction, Int. J. Comput. Appl. 76 (5) (2013) 6–10.
[43] X. Xuan, D. Lo, X. Xia, Y. Tian, Evaluating defect prediction approaches using a massive set of metrics: An empirical study, in: Proceedings of the 30th Annual ACM Symposium on Applied Computing, SAC '15, ACM, New York, NY, USA, 2015, pp. 1644–1647.
[44] T. Yang, A.A. Asanjan, M. Faridzad, N. Hayatbini, X. Gao, S. Sorooshian, An enhanced artificial neural network with a shuffled complex evolutionary global optimization with principal component analysis, Inf. Sci. 418–419 (Supplement C) (2017) 302–316.
[45] L. Yu, A. Mishra, Experience in predicting fault-prone software modules using complexity metrics, Qual. Technol. Quant. Manage. 9 (4) (2012) 421–433.
[46] J. Zheng, Predicting software reliability with neural network ensembles, Expert Syst. Appl. 36 (2, Part 1) (2009) 2116–2122.
[47] G. Wahba, Y. Lin, H. Zhang, GACV for support vector machines, or, another way to look at margin-like quantities, Adv. Large Margin Classifiers (2000) 297–309.

Diana-Lucia Miholca is currently a PhD. student in the first year of study at the Faculty of Mathematics and Computer Science from the Babeş-Bolyai University, city of Cluj-Napoca, Romania. Her main research interests are Machine Learning and Software Engineering.

Gabriela Czibula works as a professor in the Computer Science Department, Faculty of Mathematics and Computer Science from the Babeş-Bolyai University, city of Cluj-Napoca, Romania. She published more than 160 papers in prestigious journals and conferences proceedings. Her research interests include artificial intelligence, machine learning, multiagent systems, software engineering.

István Gergely Czibula works as a professor at the Computer Science Department, Faculty of Mathematics and Computer Science from the Babeş-Bolyai University, city of Cluj-Napoca, Romania. He received his PhD degree in Computer Science in 2009. He published more than 90 papers in various journals and conferences proceedings. His main research interest is Search-based Software Engineering.