Information Sciences 264 (2014) 260–278

Contents lists available at ScienceDirect

Information Sciences

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

# Software defect prediction using relational association rule mining

Gabriela Czibula *, Zsuzsanna Marian, Istvan Gergely Czibula

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

ARTICLE INFO

*Article history:*
Received 30 July 2013
Received in revised form 31 October 2013
Accepted 29 December 2013
Available online 7 January 2014

*Keywords:*
Software engineering
Defect prediction
Data mining
Association rule

ABSTRACT

This paper focuses on the problem of defect prediction, a problem of major importance during software maintenance and evolution. It is essential for software developers to identify defective software modules in order to continuously improve the quality of a software system. As the conditions for a software module to have defects are hard to identify, machine learning based classification models are still developed to approach the problem of defect prediction. We propose a novel classification model based on relational association rules mining. Relational association rules are an extension of ordinal association rules, which are a particular type of association rules that describe numerical orderings between attributes that commonly occur over a dataset. Our classifier is based on the discovery of relational association rules for predicting whether a software module is or it is not defective. An experimental evaluation of the proposed model on the open source NASA datasets, as well as a comparison to similar existing approaches is provided. The obtained results show that our classifier overperforms, for most of the considered evaluation measures, the existing machine learning based techniques for defect prediction. This confirms the potential of our proposal.

© 2014 Elsevier Inc. All rights reserved.

## 1. Introduction

*Software quality* [43] is considered of great importance in the software engineering field. Nevertheless, building software of high quality is very expensive. Thus, in order to increase the efficiency and usefulness of quality assurance and testing, software defect prediction is used to identify defect-prone modules in a forthcoming version of a software system and help allocate the effort on those modules [8].

Association rule mining [38] means searching attribute–value conditions that occur frequently together in a dataset [19,52]. Ordinal association rules [33] are a particular type of association rules. Given a set of records described by a set of attributes, the ordinal association rules specify ordinal relationships between record attributes that hold for a certain percentage of the records. However, in real world datasets, attributes with different domains and relationships between them, other than ordinal, do actually exist. In such situations, ordinal association rules are not strong enough to describe data regularities. Consequently, *relational association rules* were introduced in [48] in order to be able to capture various kinds of relationships between record attributes.

This paper proposes a novel classification model for the problem of defect prediction, based on the idea of discovering *relational association rules* within a dataset. Predicting whether a software module is defective or not is of major importance

* Corresponding author. Tel.: +40 264 405 327; fax: +40 264 591 906.
*E-mail addresses:* gabis@cs.ubbcluj.ro (G. Czibula), marianzsu@cs.ubbcluj.ro (Z. Marian), istvanc@cs.ubbcluj.ro (I.G. Czibula).
*URLs:* http://www.cs.ubbcluj.ro/~gabis (G. Czibula), http://www.cs.ubbcluj.ro/~istvanc (I.G. Czibula).

0020-0255/$ - see front matter © 2014 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.ins.2013.12.031

for the maintenance and evolution of software systems, as developers are continuously interested in improving the software quality. The results obtained by evaluating the classification model proposed in this paper do confirm that applying relational association rule mining for defect detection is promising and indicate the potential of our proposal. Moreover, the use of relational association rules in classifying software entities as being defective or not, is a novel approach.

The rest of the paper is organized as follows. The motivation for our research is presented in Section 2, followed by the description of the problem of *defect prediction* and its relevance, as well as existing machine learning based approaches for solving this problem in Section 3. Section 4 presents the concept of relational association rules. Section 5 introduces our model for defect prediction based on relational association rule mining. An experimental evaluation of our approach is reported in Section 6, and an analysis of the proposed model and comparison with similar existing approaches are given in Section 7. Conclusions and further work are outlined in Section 8.

## 2. Motivation

Identifying the software entities (such as classes, modules, methods, and functions) that are defective is of major importance as it facilitates further software evolution and maintenance. Although many models have previously been proposed in the software defect prediction literature, this problem has not been completely solved and researchers are still focusing on developing more accurate defect predictors. Recent results show that researchers should concentrate on improving the quality of the data in order to overcome the limits of the existing software prediction models [8].

*Relational association rules* [48] were introduced as an extension to association rules, in order to be able to discover various kinds of relations or correlations that exist between data in large datasets. A software module from a software system can be characterized by a set of relevant software metrics values. These software metrics may be relevant for deciding if a module is defective or not. Consequently, a software module can be visualized as a high dimensional vector and the entire software system can be represented as a dataset consisting of the high dimensional vectors corresponding to the system's software modules. Within this dataset, where the records are the (high dimensional) software modules and the features are the software metrics, significant information can be extracted from the software metrics values characterizing the modules. Different types of relationships between the numerical feature values can be defined and a relational association mining process can be performed on the dataset representing the software system. Such a mining process can provide interesting patterns that would be useful for predicting if a software module is or it is not defective.

In our proposal, we have started from the intuition that when deciding if a software entity is defective, relational association rules may be effective, as relationships between the software metrics values characterizing the software entities may be relevant. These relationships may express quantitative information that may exist in the vector characterizing a software entity. It is likely that these relationships could provide significant information regarding defective entities.

Although many methods for software defect prediction do exist within the software engineering literature, recent researches are still carried out for proposing more accurate software defect predictors and for overcoming the drawbacks and limitations of the existing models. Relational association rule mining has not been applied so far for predicting if a software entity is defective or not. We therefore aim in this paper at developing a novel method based on relation association rules, whose effectiveness will be shown through the experimental results.

## 3. Defect prediction

In this section we aim at presenting the problem of *defect prediction* and its relevance, as well as existing machine learning based approaches for solving the considered problem.

### 3.1. Problem statement and relevance

The automated estimation of software, in terms of defect prediction, is of major importance to the software engineering community and researchers are continuously focusing on building accurate and trustworthy predictors using legacy data.

In order to deliver high quality software on time, software project managers, quality managers and software developers need to continuously monitor, detect and correct software defects at all stages of the development process. Defects such as faults or bugs represent a major factor in planning the on-time-delivery and the quality of the released product especially during the maintenance and evolution of a software project.

The product quality is highly correlated with the (absence of) defects. It is therefore of high importance for developers and project managers to assure a software development with as few errors as possible. In this direction, *software defect prediction* helps in detecting, tracking and resolving software anomalies that might have an effect on human safety and lives, particularly in safety critical systems. Defect prediction also allows changes to be made earlier in the software life-cycle, assuring this way a lower software cost and improving customer satisfaction [25].

A *software defect* represents any error or deficiency in a software artifact or a software process and a major focus is on predicting those defects that influence project or product performance. Many software defect predictors use *software metrics* [12] to measure the software quality in order to predict software defects. Thus, *software defect prediction* is the task of

classifying software modules into the fault-prone and the non-fault-prone ones by using metric-based classification [5]. Most defect prediction techniques used in planning are based on historical data, hence rely on supervised classification.

### 3.2. Related work

Although association rules [3] are usually used in an unsupervised learning scenario, different extensions for classification are presented in the literature. One of them is the *CBA* method, presented in [29], where class association rules, i.e. association rules whose consequent is a class label, are mined. [30] presents an extension of this method, called *CBA*2, where rules predicting different classes can have a different minimum support, to solve the data imbalance problem.

Ma et al. use in [2] the CBA2 method for predicting defective software modules. Experiments on the NASA datasets [36] are performed and comparisons with other rule based classification methods are provided. The authors also investigate whether the association rule sets that were generated based on the data from one software project can be used to predict defective software modules in other, similar software projects [2].

Kamei et al. present in [24] a hybrid model, which combines association rule mining and logistic regression. When a new instance needs to be classified, if there are rules that fit the given instance, they will be used for classification, otherwise the result is provided by a logistic regression model.

Rodriguez et al. introduce in [45] a *Subgroup Discovery (SD)* algorithm named *EDER-SD* (Evolutionary Decision Rules for Subgroup Discovery) that is based on evolutionary computation and generates rules describing only fault-prone modules. The experiments performed on datasets from NASA showed that the *EDER-SD* algorithm performs well in most cases when compared to three other well known SD algorithms.

Besides rule-based methods, many different machine learning algorithms have been applied to the problem of defect prediction. One such work is that of Menzies et al., [35], in which they evaluate the Naive Bayes classifier, OneR and J48. They have also experimented with different filters and concluded that, on average, logarithmic filtering and Naive Bayes produced the best results on the 8 used NASA datasets. [44] presents a literature study about a couple of methods that are often used for defect prediction: simple equations, machine learning methods and defect density prediction models. Similarly, Kaur et al. in [26] shortly present clustering, classification and association mining as software defect prediction methods.

Challagulla et al. evaluate in [7] some different predictor models on four different real-time software defect data sets that were taken from the NASA repository [36]. The experimental results have shown that a combination of 1-rule classification [21] and Instance-based Learning using Consistency based Subset Evaluation technique provides a relatively better consistency in accuracy prediction compared to other models [7].

Haghighi et al. provide in [16] a comparative analysis of 37 different classifiers in fault detection systems and use the NASA datasets for performing experiments. The results showed that, on average, the *Bagging* classifier achieved a higher performance and accuracy compared to the others.

A disagreement-based semi-supervised learning method, called ROCUS, is presented by Jiang et al. in [23]. They use semi-supervised learning because in defect prediction there is a limited amount of labeled data, whereas gathering unlabeled data is easy. They also use under-sampling to solve the class imbalance problem. ROCUS is evaluated on 8 datasets from the NASA repository, and the results are compared to other methods which are either semi-supervised methods that do not take into consideration the imbalanced data, or class-imbalance learning methods that cannot exploit unlabeled data.

Another disagreement-based semi-supervised learning model is presented by Li et al. in [28]. This method, called ACoForest, requires only a percentage of the modules to be labeled, and tries to label the rest of them based on a model built from the labeled data. The novelty of the method is that it uses active-learning, so it can suggest which data to be labeled (the ones on which the learners mostly disagree). Tests performed on publicly available datasets showed that this method outperforms conventional machine learning algorithms.

Guo et al. present in [15] a method that uses Random Forests for predicting if a module contains faults or not. They perform tests on five NASA datasets and compare their method to different statistical and machine learning algorithms. They conclude that Random Forests have higher overall prediction accuracy and/or higher defect detection rate than most of the other algorithms, and also that they work especially well on large datasets.

In the last years the focus on software defect prediction seems to have shifted towards formalizing and standardizing defect prediction methods. For example, in [49] Song et al. define a framework for software defect prediction based on learning schemes (made of a data pre-processing method, an attribute selection method and the actual learning algorithm). The framework consists of two parts: scheme evaluation, where the learning scheme is evaluated and a model is learnt, and secondly, defect prediction, when the learnt model is used. In [10] a benchmark for defect prediction in the form of publicly available datasets is presented. They introduce this benchmark to facilitate the comparison of different approaches. Menzies and Shepperd in [37] investigate the conclusion instability of the prediction systems (i.e. the fact that something true for a project is not true for a different one) by presenting possible reasons and solutions for these problems.

Metrics that evaluate the results of the methods have also been analysed for the same reason of comparing different approaches, to show which ones are more suitable for describing the results (although there is still no clear consensus about them). Ref. [22] presents a list of different performance measures, numerical, graphical and statistical, claiming that the best one can be project specific. However, almost all papers agree that accuracy alone is not very suitable when data is so skewed. Ref. [2] claims that AUC is a more suitable metric to be reported, and Gray et al. in [13] demonstrate that precision should

also be reported, because good values for probability of detection and probability of false alarm can also be achieved for a method with very low precision.

## 4. Relational association rules: Background

Classical association rules discard any quantitative information that may exist between record attributes in datasets, but many times this type of information can give valuable insights into the problem at hand. The record attributes may be in an ordinal relationship, if the domains of the attributes are similar or comparable. Otherwise, when the attributes do not have commensurable values, more general relations are needed, ones that are strong enough to capture various interesting relationships between data. Therefore, the extension of classical association rules towards *ordinal* and more general, *relational* association rules allows the uncovering of much stronger rules that consequently achieve superior data mining, or classification.

In order to be able to capture various kinds of relationships between record attributes, we have extended in [48] the definition of ordinal association rules [33,6] towards *relational association rules*.

In *relational association rule mining*, the objective is to find several relationships between the attributes that tend to hold over a large percentage of records. In a binary classification problem, if attribute A is in relation with attribute B for a great number of positive instances, then a record in which attribute A is not in relation with attribute B may be a negative instance. It may not mean very much if only one rule including B is not fulfilled, but it increases the likelihood that the instance in question belongs to the negative class if many such rules are broken.

The following will briefly review the concept of *relational association rules*, as well as the mechanism for identifying the relevant relational association rules that hold within a dataset.

Let $R = \{r_1, r_2, \ldots, r_n\}$ be a set of *instances* (entities or records in the relational model), where each instance is characterized by a list of $m$ attributes, $(a_1, \ldots, a_m)$. We denote by $\Phi(r_j, a_i)$ the value of attribute $a_i$ for the instance $r_j$. Each attribute $a_i$ takes values from a domain $D_i$, which contains the empty value denoted by $\varepsilon$. Between two domains $D_i$ and $D_j$ relations can be defined (not necessarily ordinal relations), such as: less or equal ($\leqslant$), equal (=), greater or equal ($\geqslant$), etc. We denote by $M$ the set of all possible relations that can be defined on $D_i x D_j$.

A *relational association rule* [48] is an expression $(a_{i_1}, a_{i_2}, a_{i_3}, \ldots, a_{i_\ell}) \Rightarrow (a_{i_1}\ \mu_1\ a_{i_2}\ \mu_2\ a_{i_3} \cdots \mu_{\ell-1}\ a_{i_\ell})$, where $\{a_{i_1}, a_{i_2}, a_{i_3}, \ldots, a_{i_\ell}\} \subseteq \mathcal{A} = \{a_1, \ldots, a_m\}, a_{i_j} \neq a_{i_k},\ j,k = 1 \ldots \ell,\ j \neq k$ and $\mu_i \in M$ is a relation over $D_{i_j} \times D_{i_{j+1}}, D_{i_j}$ is the domain of the attribute $a_{i_j}$. If:

(a) $a_{i_1}, a_{i_2}, a_{i_3}, \ldots, a_{i_\ell}$ occur together (are non-empty) in $s\%$ of the $n$ instances, then we call $s$ the *support* of the rule, and
(b) we denote by $R' \subseteq R$ the set of instances where $a_{i_1}, a_{i_2}, a_{i_3}, \ldots, a_{i_\ell}$ occur together and the relations $\Phi(r_j, a_{i_1})\ \mu_1\ \Phi(r_j, a_{i_2}), \Phi(r_j, a_{i_2})\ \mu_2\ \Phi(r_j, a_{i_3}) \ldots \Phi(r_j, a_{i_{\ell-1}})\ \mu_{\ell-1}\ \Phi(r_j, a_{i_\ell})$ hold for each instance $r_j$ from $R'$; then we call $c = |R'|/|R|$ the *confidence* of the rule.

We call the *length* of a relational association rule the number of attributes in the rule. The length of a relational association rule can be at most equal to the number $m$ of the attributes describing the data.

The users usually need to uncover interesting relational association rules that hold in a dataset; they are interested in relational rules which hold in a minimum number of instances, that is, rules with support at least $s_{min}$, and confidence at least $c_{min}$ ($s_{min}$ and $c_{min}$ are user-provided thresholds).

We call a relational association rule in $R$ *interesting* if its support $s$ is greater than or equal to a user-specified minimum support, $s_{min}$, and its confidence $c$ is greater than or equal to a user-specified minimum confidence, $c_{min}$.

We have previously introduced in [6] an A-Priori [1] like algorithm, called *DOAR* (Discovery of Ordinal Association Rules), that efficiently finds all ordinal association rules (i.e. relational association rules in which the relations are ordinal) of any length, that hold over a dataset.

In the following a brief description of the idea of discovering interesting ordinal association rules will be given [6]. The mechanism of discovering interesting ordinal association rules in a dataset will be extended in our approach towards identifying relational association rules.

This algorithm identifies ordinal association rules using an iterative process that consists in length-level generation of candidate rules, followed by the verification of the candidates for minimum support and confidence compliance. *DOAR* performs multiple passes over the dataset $R$. In the first pass, it calculates the support and confidence of the 2-length rules and determines which of them are interesting, (i.e. verify the minimum support and confidence requirement). Every subsequent pass over the data consists of two phases. The first phase starts with a seed set of $(k-1)$-length ($k \geqslant 3$) interesting rules, found in the previous pass. This set is used to generate new possible $k$-length interesting rules, called candidate rules. The candidate generation process is a key element of the *DOAR* algorithm. During the second phase, a scan over the $R$ data is performed in order to compute the actual support and confidence of the candidate rules. At the end of this step, the algorithm keeps the rules that are deemed interesting (have minimum support and satisfy the confidence requirements), which will be used in the next iteration. The process stops when no new interesting rules were found in the latest iteration.

The *DOAR* algorithm significantly prunes the exponential search space of all possible interesting ordinal association rules, due to the candidate generation technique. The candidate generation restricts the search to those regions of the search space

where it is possible that interesting rules may exist, pruning out all the regions where it is impossible to find any interesting rules. The search space reduction depends on the data being analyzed. The larger the number of interesting rules in the dataset is, the larger the size of the candidate sets will be.

We have proven that the proposed algorithm is correct and complete and we have shown that it efficiently explores the search space of the possible rules. More about the *DOAR* algorithm and its theoretical validation is given in [6].

The *DOAR* algorithm is extended in our approach towards the *DRAR* algorithm (*Discovery of Relational Association Rules*) for finding interesting relational association rules, i.e. association rules which are able to capture various kinds of relationships between record attributes.

Our current implementation provides two functionalities:

- Finds all interesting relational association rules of any length.
- Finds all maximal interesting relational association rules of any length, i.e. if an interesting rule $r$ of a certain length $l$ can be extended with one attribute and it remains interesting (its confidence is greater than the threshold), only the extended rule is kept.

### 4.1. Example

In order to better explain the concept of relational association rules and the extension of the *DOAR* algorithm [6] that is used for discovering relational association rules, we give an example of a relational association rule mining task within a software system.

Let us consider the Java code example shown in Fig. 1. The example is taken from [47] and was used by the authors in order to illustrate the *Move Method* refactoring.

As we have described in Section 3.1, the dataset considered in the mining process consists of a set of *software entities* (instances), each software entity being characterized by a set of *software metrics* (features characterizing the instances).

We consider in our example that a software entity can be either an application class, or a method from an application class. The software metrics considered in our experiment are:

1. Depth in Inheritance Tree (DIT) [9].
2. Number of Children (NOC) [9].
3. Fan-In (FI) [20,32].
4. Fan-Out (FO) [20,32].

We have previously used these software metrics in [34] for a clustering based automatic identification of refactorings that would improve the internal structure of a software system.

Using the above mentioned software metrics, each software entity from the system presented in Fig. 1 can be represented as a 4-dimensional vector, having as components the values of the considered metrics. The corresponding dataset is given in Table 1.

```
public class Class_A {
  public static int attributeA1;
  public static int attributeA2;

  public static void mA1(){
      attributeA1 = 0;
      mA2();
  }

  public static void mA2(){
          attributeA2 = 0;
          attributeA1 = 0;
    }

  public static void mA3(){
          attributeA2 = 0;
          attributeA1 = 0;
          mA1();
          mA2();
  }
}
```

```
public class Class_B {
  private static int attributeB1;
  private static int attributeB2;

  public static void mB1(){
          Class_A.attributeA1=0;
          Class_A.attributeA2=0;
          Class_A.mA1();
    }

  public static void mB2(){
          attributeB1=0;
          attributeB2=0;
    }

  public static void mB3(){
          attributeB1=0;
          mB1();
          m2();
  }
}
```

**Fig. 1.** Code example.

**Table 1**
Sample dataset.

| Entity | *DIT* | *NOC* | *FI* | *FO* |
|---|---|---|---|---|
| Class_A | 1 | 0 | 3 | 1 |
| Class_B | 1 | 0 | 0 | 2 |
| mA1 | 1 | 0 | 2 | 1 |
| mA2 | 1 | 0 | 2 | 0 |
| mA3 | 1 | 0 | 0 | 2 |
| mB1 | 1 | 0 | 1 | 1 |
| mB2 | 1 | 0 | 1 | 0 |
| mB3 | 1 | 0 | 0 | 2 |

**Table 2**
Interesting relational association rules.

| Length | Rule | Confidence |
|---|---|---|
| 2 | DIT > NOC | 1 |
| 2 | NOC < FI | 0.625 |
| 2 | NOC < FO | 0.75 |
| 2 | FI > FO | 0.5 |
| 3 | DIT > NOC < FI | 0.625 |
| 3 | DIT > NOC < FO | 0.75 |
| 3 | NOC < FI > FO | 0.5 |
| 4 | DIT > NOC < FI > FO | 0.5 |

**Table 3**
Maximal interesting relational association rules.

| Length | Rule | Confidence |
|---|---|---|
| 3 | DIT > NOC < FO | 0.75 |
| 4 | DIT > NOC < FI > FO | 0.5 |

As all attributes in our experiment have numerical values, we have defined two possible binary relations between the attributes: < and >.

We executed the *DRAR* algorithm with minimum support threshold of 0.9 and minimum confidence threshold of 0.4. The discovered interesting relational rules are shown in Table 2 and the maximal interesting association rules are given in Table 3. For each discovered rule, its confidence is also provided.

As it can be seen from the results above, interesting relational association rules can be discovered within the set of software entities. Further analysis of these relational association rules may provide relevant information regarding the analyzed software system.

## 5. Methodology

In this section we introduce a novel supervised method for detecting software entities with defects, based on relational association rule mining, called *DPRAR* (*Defect Prediction using Relational Association Rules*).

### 5.1. Theoretical model

The main idea of this approach is to represent the entities (classes, modules, methods, functions) of a software system as a multidimensional vector, whose elements are the values of different software metrics applied to the given entity. In order to give a formal definition, we consider that a software system $S$ is a set of components (called *entities*) $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$.

It is well known that software metrics are widely used to measure the software quality [12]. As we aim at identifying software entities having defects, we consider a set of software metrics (the *feature* set in a vector space model based approach) relevant for deciding if a software entity is or not defective. Consequently, we have a feature set of software metrics $\mathcal{SM} = \{sm_1, sm_2, \ldots, sm_k\}$ and thus each entity $s_i \in \mathcal{S}$ from the software system can be represented as a $k$-dimensional vector, having as components the values of the software metrics from $\mathcal{SM}, s_i = (s_{i1}, s_{i2}, \ldots, s_{ik})$ ($s_{ij}$ represents the value of the software metric $sm_j$ applied to the software entity $s_i$).

### 5.2. Our approach

The problem that we are focusing on is a binary classification problem. There are two possible classes, denoted in the following by "+" and "−". By "+" we denote the class corresponding to software entities having defects, and the entities that belong to the "+" class will be referred to as *positive instances* or *defects*. By "−" we denote the class corresponding to software entities that are not defective, and the entities that belong to the "−" class will be referred to as *negative instances* or *non-defects*.

The main idea of our approach is the following. In a supervised learning scenario for predicting defective software entities, two sets containing positive and negative instances are given. Considering the vector space model presented in Section 5.1, these datasets consist of $k$-dimensional software entities from a software system. These sets will be used for training the classifier. During training, the *DRAR* algorithm will be used. We detect in the training datasets all the interesting relational rules, with respect to the user-provided support and confidence thresholds. After the training was completed, when a new instance (software entity) has to be classified (as "+" or "−"), we reason as follows. Considering the rules discovered during training in the set of *positive* and *negative* instances, two scores, $score_+$ (indicating the similarity degree of the instance to the positive class) and $score_-$ (indicating the similarity degree of the instance to the negative class), are computed. If $score_+$ is greater than $score_-$, then the query instance will be classified as a *positive* instance, otherwise it will be classified as a *negative* instance.

The process takes place in two phases that reflect the principles of a supervised learning algorithm: *training* and *testing*. During training a classification model will be built, and during testing, the model built during training will be applied for classifying an unseen instance. As mentioned above, we consider for training two datasets: $DS_+$ consisting of *positive* $k$-dimensional instances (software entities that are defective) and $DS_-$ consisting of *negative* $k$-dimensional instances (software instances that are not defective). These datasets are used in the **training step** of the *DPRAR* classifier and a classification model consisting of the discovered relational association rules is built. At the classification time, when a new instance (software entity) $e$ has to be classified, the model learned during the training step will be used for computing the similarity degrees of the instance $e$ to the *positive* and *negative* classes, i.e. to predict if the query instance is or not defective.

For classifying a a software entity as being or not defective, the following steps will be performed:

1. Data pre-processing.
2. Training/building the *DPRAR* classifier.
3. Testing/classification.

The following will describe these steps.

### 5.3. Data pre-processing

During this step, the training data are scaled to $[0, 1]$ and a statistical analysis is carried out on the training datasets $DS_+$ and $DS_-$ in order to find a subset of features that are correlated with the target output. The statistical analysis on the features is performed in order to reduce the dimensionality of the input data, by eliminating features which do not significantly influence the output value.

To determine the dependencies between features and the target output, the Spearman's rank correlation coefficient [50] is used. A Spearman correlation of 0 between two variables $X$ and $Y$ indicates that there is no tendency for $Y$ to either increase or decrease when $X$ increases. A Spearman correlation of 1 or $-1$ results when the two variables being compared are monotonically related, even if their relationship is not linear. At the statistical analysis step we remove from the feature set those features that have no significant influence on the target output, i.e. are slightly correlated with it.

In order to decide which feature(s) to remove, we reason as follows. For each feature (software metric) $sm_i \in \mathcal{SM}$ we compute the Spearman correlation ($cor(sm_i, target)$) between the feature and the target output (defect or non-defect). Let us denote by $m$ the average value and $stdev$ the standard deviation of the correlations between all features and the target output. We consider that a feature $sm_i$ is slightly correlated with the target classification output and will be removed from the feature set if the absolute value of the correlation is less than $m - stdev$, i.e. $abs(cor(sm_i, target)) < m - stdev$.

The dataset pre-processed this way can now be used for building the relational association rule based classification model.

### 5.4. Training

First, we define a set of relations between the feature values that will be used in the relational association rule mining process. More exactly, we are focusing on identifying relations between two software metrics (features), relations that would be relevant for deciding if a software entity is or not defective, and consequently would be useful in the mining process. After the relations were defined, the interesting relational association rules are discovered in the training datasets.

More exactly, the training consists of the following steps:

- Determine from $DS_+$, using the *DRAR* algorithm, the set $RAR_+$ of relational association rules having a minimum *support* and *confidence*.
- Determine from $DS_-$, using the *DRAR* algorithm, the set $RAR_-$ of relational association rules having a minimum *support* and *confidence*.
- For each rule $r$ from the sets $RAR_+$ and $RAR_-$ determined as indicated above, the support (denoted by $supp(r)$) and the confidence (denoted by $conf(r)$) of the rule are computed. We denote in the following by $ratio(r)$ the value obtained by dividing the confidence of the rule to its support, i.e. $ratio(r) = \frac{conf(r)}{supp(r)}$.

### 5.5. Classification

At the classification stage, after the training was completed and the *DPRAR* was built, when a new software entity $e$ has to be classified, we calculate the scores $score_+(e)$ (the similarity of $e$ to the *positive* class) and $score_-(e)$ (the similarity of $e$ to the *negative* class). In computing these scores we started based on the intuition that the similarity of an instance $e$ to the *positive* class, for example, is very likely to be influenced by the rules from $RAR_+$ that are verified in the entity $e$ but also by the rules from $RAR_-$ that are not verified in the entity $e$. In this way, $score_+$ measures not only how "close" the entity is to the positive instances, but also how "far" it is from the negative ones.

We propose the following steps for computing the scores:

- Determine $n_+$ as the average values of $ratio(r)$ for each rule $r$ from $RAR_+$ that is verified in the entity $e$ and $n_-$ as the average values of $ratio(r)$ for each rule $r$ from $RAR_-$ that is not verified in the entity $e$.
- Calculate $score_+$ as $score_+ = n_+ + n_-$.
- Determine $m_-$ as the average values of $ratio(r)$ for each rule $r$ from $RAR_-$ that is verified in the entity $e$ and $m_+$ as the average values of $ratio(r)$ for each rule $r$ from $RAR_+$ that is not verified in the entity $e$.
- Calculate $score_-$ as $score_- = m_- + m_+$.

The above presented score computation method takes into consideration the strength of the verified and unverified rules (by using the value of *ratio*, which increases as the confidence of the rule increases), but there are other possibilities for score computation as well: using only the number of these rules, or computing one single score, which can be transformed into a class label with the use of a threshold. In the future we will investigate other score computation formulas.

At the classification stage of a new instance $e$ if $score_+ > score_-$ then instance $e$ will be classified as a *positive* instance (defect), otherwise it will classified as a *negative* instance (non-defect).

### 5.6. Testing

For evaluating the performance of our classifier, a cross-validation using a "leave-one-out" methodology will be applied. As for a binary classification task, the confusion matrix for the two possible outcomes (positive and negative) is computed. The confusion matrix [51] consists of: the number of *true positives* (*TP* – the number of actual positive instances predicted as positive), the number of *false positives* (*FP* – the number of actual negative instances predicted as positive), the number of *true negatives* (*TN* – the number of actual negative instances predicted as negative) and the number of *false negatives* (*FN* – the number of actual positive instances predicted as negative).

The literature gives us different evaluation measures whose values are computed based on the values from the confusion matrix. Not all papers from the work related to defect prediction report the same evaluation measures. That is why, in order to better compare our method to the existing ones, we are going to use in this paper a union of the measures that were used in the literature to evaluate software defect predictors.

When considering the values computed from the confusion matrix, the following evaluation measures for defect detectors will be used in this paper:

1. The classification *accuracy* (denoted by *Acc*) measures the percentage of instances that are classified correctly (or wrongly) by a classifier, i.e. $Acc = \frac{TP+TN}{TP+TN+FP+FN}$.
2. The *probability of detection* (denoted by *Pd*), or the *recall/sensitivity* of the classifier computes the proportion of actual positives which are predicted positive, i.e. $Pd = \frac{TP}{TP+FN}$.
3. The *specificity* of the classifier (denoted by *Spec*) computes the proportion of actual negatives which are predicted negative, i.e. $Spec = \frac{TN}{TN+FP}$.
4. The classification *precision* (denoted by *Prec*) computes the proportion of predicted positives which are actual positive, i.e. $Prec = \frac{TP}{TP+FP}$.
5. The *Area under the ROC curve* measure (*AUC*) is indicated in the literature [27,11] as one of the best evaluation measure to compare different classifiers and it is recommended as the primary accuracy indicator for comparative studies in software defect prediction [16]. The *ROC* (Receiver Operating Characteristics) curve is a two-dimensional plot of *sensitivity* vs. (1-*specificity*). ROC curves are usually constructed for classifiers which, instead of directly returning the class of an instance, return a score that is transformed into a label using a threshold. In such cases, different (*sensitivity*, 1-*specificity*) pairs are obtained for each threshold, which are represented on the ROC curve. In case of classifiers returning the class directly, the

ROC space has a single point. As presented in [31,11] this point can be linked to the points at (0,0) and (1,1), thus producing a curve, for which the AUC measure can be computed. The ROC curves constructed for our method are presented on Fig. 3;

Ideally, detectors have high *Pd*, *specificity* and AUC. These measures have to be maximized in order to obtain better detectors.

In the experimental part of the paper (Section 6), these evaluation measures will be used for comparing the results provided by the *DPRAR* classifier to the results of the classifiers already existing in the software engineering literature for defect prediction.

## 6. Experimental evaluation

This section aims at experimentally evaluating our approach for defect detection using relational association rules, as well as providing a comparison with other existing similar approaches. The case studies used in our experiment, the methodology used, as well as the obtained results are presented in the following. The datasets used in our experiments are open source and available at [41], a software engineering repository made publicly available in order to encourage repeatable, verifiable, and improvable predictive models of software engineering. These 13 public fault data repositories, out of which we will use 10, often called NASA datasets, were originally published at NASA's Independent Verification and Validation (IV&V) Facility website [40], but are no longer available there. They were taken over by the PROMISE (PRedictOr Models In Software Engineering) repository [36], which has recently moved to a new address, and the old one is no longer available. Since they were freely available for anyone who wanted to build or test defect prediction models, they became very popular. A recent study found that out of 208 defect prediction studies 58 used at least one NASA dataset [17]. In 2011 Gray et al. in [14] describe that these datasets need serious data cleaning before analysis, because they contain duplicated and inconsistent instances, especially the Promise version of the datasets. Based on that paper, Shepperd et al. in [46] first present that the datasets on the original IV&V website and the ones at the Promise site differ both in number of instances and number of attributes. Then, they identify possible problems with attributes (for example, constant value for all instances, missing values, violating different constrains, etc.) and instances (for example, repeated instances, inconsistent instances, cases with implausible values, etc.), and present an algorithm that cleans the data. Both the implementation and the cleaned datasets are available online at the NASA – Software Defect Datasets webpage [41]. There are actually two cleaned versions for each datasets: $DS'$ – where duplicated and inconsistent instances are kept, and $DS''$ – where duplicated and inconsistent instances are eliminated as well. These cleaned datasets are currently available in the Promise repository [36] as well. In all our analyses we have used the $DS''$ version of the datasets, taken from [41].

In our evaluation we are focusing on detecting software modules that are likely to be defective, thus an *entity* (Section 5.1) is considered to be a *module*, which can be a function, procedure or method, depending on the programming language used to write the system. We mention that the *DPRAR* classifier is general, and that it can also be used for detecting possible defective *application classes*, *subprograms*, etc., if an appropriate representation of these entities is provided.

The methodology presented in Section 5 is applied for each case study. The first stage, the **data pre-processing** step that depends on the considered dataset will be detailed for each case study. The other steps of *DPRAR*, namely **building the** *DPRAR* **classifier** and the **testing** step are applied as described in SubSection 5.2. The datasets pre-processed as indicated above, are used for building the *DPRAR* classifier (Section 5). For all the experiments, we have considered two possible relations between the software metrics characterizing a software entity: $\leqslant$ and $>$ (we have considered that the relations are not defined between zero valued software metrics) and we executed the classification algorithm introduced in Section 5 with minimum support threshold $s_{min} = 0.9$ and different values for the minimum confidence thresholds for the dataset of positive and negative instances. The minimum confidence threshold considered for the dataset $DS_+$ is denoted by $c^+_{min}$ and the minimum confidence threshold considered for the dataset $DS_-$ is denoted by $c^-_{min}$.

When conducting the case studies, we used a software framework that we have designed for binary classification, based on the discovery of interesting relational association rules. This interface implements the *DRAR* algorithm (a variation of the *DOAR* algorithm previously introduced in [6]) developed for detecting relational association rules in a dataset.

For evaluating the performance of the *DPRAR* classifier, a cross-validation using a "leave-one-out" methodology was applied and the performance measures presented in Section 5.2 used.

### *6.1. The **CM1** dataset*

The CM1 dataset represents a NASA spacecraft instrument written in the C programming language. It consists of 42 positive instances (defects) and 285 negative instances (non-defects), meaning that there are 12.84% positive instances and 87.16% negative instances. Each instance has 37 features and the class label.

#### *6.1.1. DPRAR results*

Fig. 2 shows the absolute values of the correlations between the features (software metrics) and the target output (defect or correct) for the **CM1** dataset.

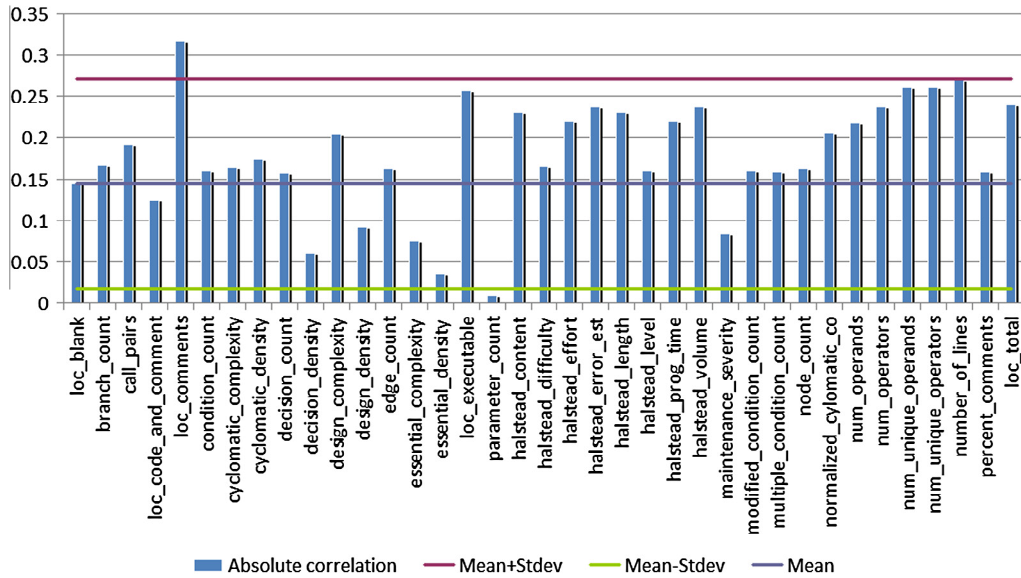


**Fig. 2.** Correlations for the **CM1** dataset.

**Table 4**
Obtained results for all datasets.

| Case study | $c^+_{min}$ | $c^-_{min}$ | Length | Acc | Pd | Spec | Prec | AUC |
|---|---|---|---|---|---|---|---|---|
| CM1 | 0.927 | 0.94 | Any | 0.8716 | 0.929 | 0.8632 | 0.5 | 0.896 |
| KC1 | 0.8 | 0.822 | 2 | 0.823 | 0.818 | 0.825 | 0.628 | 0.822 |
| KC3 | 0.885 | 0.96 | 2 | 0.83 | 0.889 | 0.8165 | 0.5246 | 0.85225 |
| PC1 | 0.95 | 0.995 | 2 | 0.956 | 0.885 | 0.963 | 0.692 | 0.924 |
| JM1 | 0.95 | 0.995 | Any | 0.96 | 0.842 | 0.992 | 0.967 | 0.917 |
| MC2 | 0.96 | 0.99 | Any | 0.896 | 0.773 | 0.9632 | 0.919 | 0.868 |
| MW1 | 0.97 | 0.975 | Any | 0.941 | 0.889 | 0.947 | 0.667 | 0.918 |
| PC2 | 0.95 | 0.995 | Any | 0.984 | 0.938 | 0.985 | 0.577 | 0.962 |
| PC3 | 0.95 | 0.995 | 2 | 0.967 | 0.85 | 0.983 | 0.877 | 0.917 |
| PC4 | 0.95 | 0.995 | 2 | 0.961 | 0.814 | 0.985 | 0.894 | 0.899 |

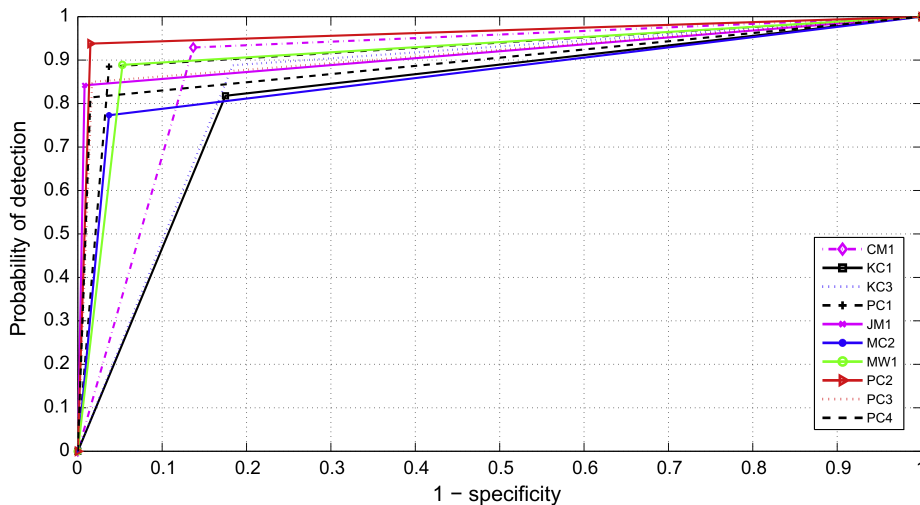


**Fig. 3.** ROC curves.

As a result of the analysis indicated in SubSection 5.3, we concluded that the 17th feature (software metric) *parameter_count* is slightly correlated with the target output and it should therefore be removed from the feature set.

Table 4 presents the best results obtained by the *DPRAR* classifier for all datasets considered for evaluation (preprocessed as indicated below). We should mention that the maximal interesting relational association rules of various lengths (i.e. 2-length rules vs. any length rules) were considered for this case study together with different values for the minimum confidence thresholds. Fig. 3 shows the ROC curves obtained on the NASA datasets used in our experiments.

### 6.2. The **KC1** dataset

The KC1 dataset contains data for a C++ system implementing storage management for receiving and processing ground data. It consists of 314 positive instances (defects) and 869 negative instances (non-defects), meaning that there are 26.54% positive instances and 73.46% negative instances. Each instance has 21 features and the class label.

#### 6.2.1. DPRAR results

Fig. 4 shows the absolute values of the correlations between the features (software metrics) and the target output (defect or correct) for the **KC1** dataset.

As a result of the analysis indicated in Section 5.3, we concluded that the *third* feature (software metric) *loc_code_and_comment* is slightly correlated with the target output and it should therefore be removed from the feature set.

Table 4 presents the best result obtained by the *DPRAR* classifier for the **KC1** dataset (preprocessed as indicated below). We should mention that the maximal interesting relational association rules of various lengths (i.e. 2-length rules vs. any length rules) were considered for this case study together with different values for the minimum confidence thresholds.

### 6.3. The **KC3** dataset

The KC3 dataset contains data about a system written in Java for processing and delivery of satellite metadata. It consists of 36 positive instances (defects) and 158 negative instances (non-defects), meaning that there are 18.56% positive instances and 81.44% negative instances. Each instance has 39 features and the class label.

#### 6.3.1. DPRAR results

As a result of the analysis indicated in Section 5.3, we concluded that there is no slightly correlated feature (software metric) to the target output for the **KC3** dataset. Consequently, the feature set remained the same and no features were removed from it.

Table 4 presents the best result obtained by the *DPRAR* classifier for the **KC3** dataset (preprocessed as indicated below). We should mention that the maximal interesting relational association rules of various lengths (i.e. 2-length rules vs. any length rules) were considered for this case study together with different values for the minimum confidence thresholds.

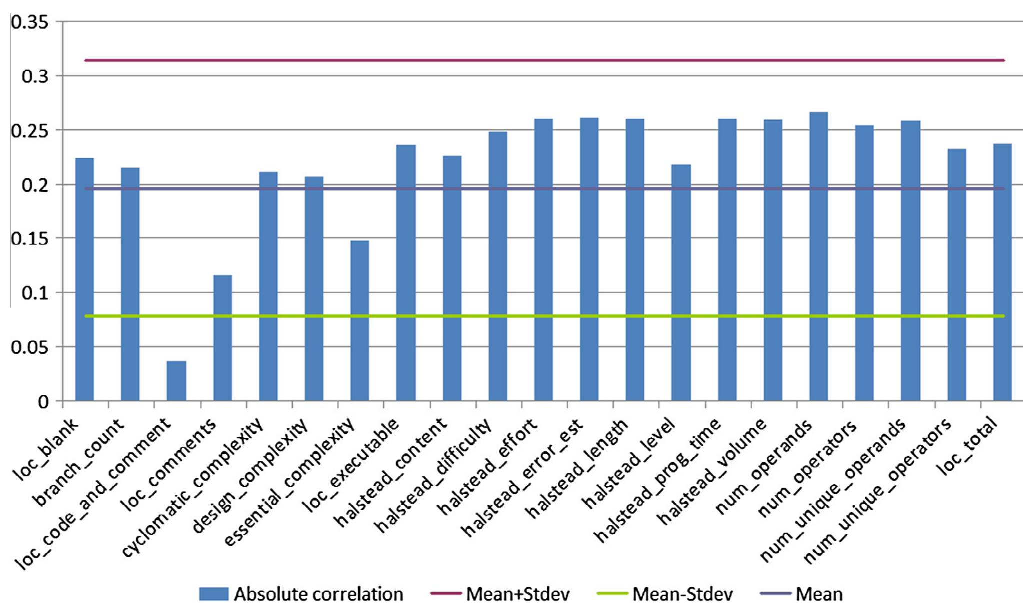


**Fig. 4.** Correlations for the **KC1** dataset.

### 6.4. The **PC1** dataset

The PC1 dataset is built for functions from a flight software for earth orbiting satellite, written in C. It consists of 61 positive instances (defects) and 644 negative instances (non-defects), meaning that there are 8.65% positive instances and 91.35% negative instances. Each instance has 37 features and the class label.

#### 6.4.1. DPRAR results

As a result of the analysis indicated in Section 5.3, we concluded that there is no slightly correlated feature (software metric) to the target output for the **PC1** dataset. Consequently, the feature set remained the same and no features were removed from it.

Table 4 presents the best result obtained by the *DPRAR* classifier for the **PC1** dataset (preprocessed as indicated below). We should mention that the maximal interesting relational association rules of various lengths (i.e. 2-length rules vs. any length rules) were considered for this case study together with different values for the minimum confidence thresholds.

### 6.5. The **JM1** dataset

The JM1 dataset contains data about a real-time predictive ground system, which uses simulations to generate predictions, written in C. It consists of 1672 positive instances (defects) and 6110 negative instances (non-defects), meaning that there are 21.49% positive instances and 78.51% negative instances. Each instance has 21 features and the class label.

#### 6.5.1. DPRAR results

As a result of the analysis indicated in Section 5.3, we concluded that there is no feature (software metric) slightly correlated with the target output for the **JM1** dataset. Consequently, the feature set remained the same and no features were removed from it.

Table 4 presents the best result obtained by the *DPRAR* classifier for the **JM1** dataset (preprocessed as indicated below). We should mention that the maximal interesting relational association rules of various lengths (i.e. 2-length rules vs. any length rules) were considered for this case study together with different values for the minimum confidence thresholds.

### 6.6. The **MC2** dataset

The MC2 dataset contains data about a video guidance system, written in C/C++. It consists of 44 positive instances (defects) and 81 negative instances (non-defects), meaning that there are 35.2% positive instances and 64.8% negative instances. Each instance has 39 features and the class label.

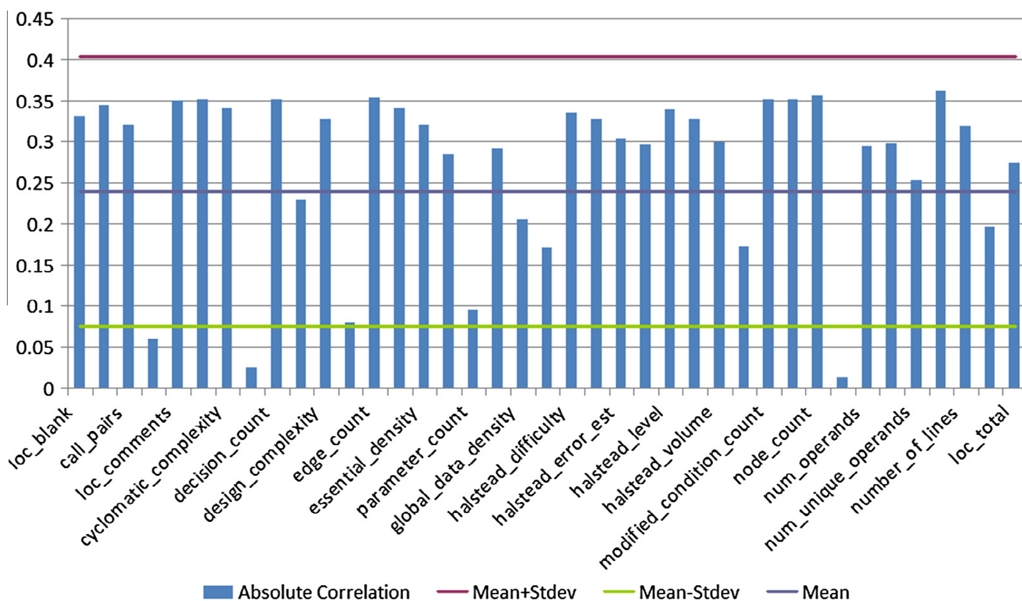


**Fig. 5.** Correlations for the **MC2** dataset.

*6.6.1. DPRAR results*

Fig. 5 shows the absolute values of the correlations between the features (software metrics) and the target output (defect or correct) for the **MC2** dataset.

As a result of the analysis indicated in Section 5.3, we concluded that features (software metrics) 4 (*loc_code_and_comment*), 8 (*cyclomatic_density*) and 32 (*normalized_cyclomatic_complexity*) are slightly correlated with the target output and they should therefore be removed from the feature set.

Table 4 presents the best result obtained by the *DPRAR* classifier for the **MC2** dataset (preprocessed as indicated below). We should mention that the maximal interesting relational association rules of various lengths (i.e. 2-length rules vs. any length rules) were considered for this case study together with different values for the minimum confidence thresholds.

## 6.7. The **MW1** dataset

The MW1 dataset contains data about a zero gravity experiment related to combustion, written in C. It consists of 27 positive instances (defects) and 226 negative instances (non-defects), meaning that there are 10.67% positive instances and 89.33% negative instances. Each instance has 37 features and the class label.

*6.7.1. DPRAR results*

Fig. 6 shows the absolute values of the correlations between the features (software metrics) and the target output (defect or correct) for the **MW1** dataset.

As a result of the analysis indicated in SubSection 5.3, we concluded that features (software metrics) 4 (*loc_code_and_comment*), 12 (*design_density*) and 17 (*parameter_count*) are slightly correlated with the target output and they should therefore be removed from the feature set.

Table 4 presents the best result obtained by the *DPRAR* classifier for the **MW1** dataset (preprocessed as indicated below). We should mention that the maximal interesting relational association rules of various lengths (i.e. 2-length rules vs. any length rules) were considered for this case study together with different values for the minimum confidence thresholds.

## 6.8. The **PC2** dataset

The PC2 dataset contains data for a dynamic simulator for attitude control systems, written in C. It consists of 16 positive instances (defects) and 729 negative instances (non-defects), meaning that there are 2% positive instances and 98% negative instances. Each instance has 36 features and the class label.

*6.8.1. DPRAR results*

As a result of the analysis indicated in Section 5.3, we concluded that there is no feature (software metric) slightly correlated with the target output for the **PC2** dataset. Consequently, the feature set remained the same and no features were removed from it.

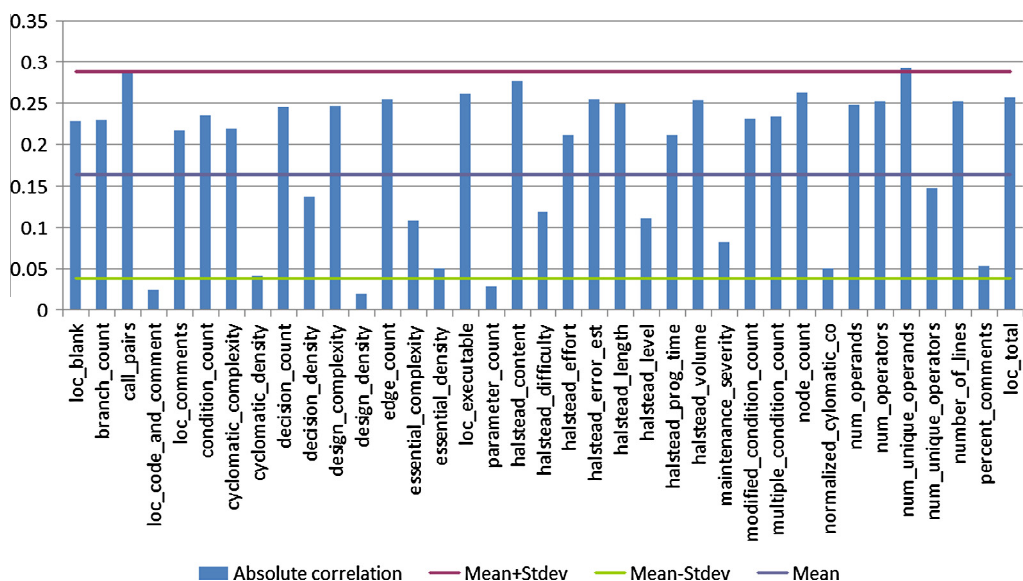


**Fig. 6.** Correlations for the **MW1** dataset.

Table 4 presents the best result obtained by the *DPRAR* classifier for the **PC2** dataset (preprocessed as indicated below). We should mention that the maximal interesting relational association rules of various lengths (i.e. 2-length rules vs. any length rules) were considered for this case study together with different values for the minimum confidence thresholds.

### 6.9. The **PC3** dataset

The PC3 dataset contains data about a flight software for earth orbiting satellite, written in C. It consists of 134 positive instances (defects) and 943 negative instances (non-defects), meaning that there are 12.4% positive instances and 87.6% negative instances. Each instance has 37 features and the class label.

#### 6.9.1. DPRAR results

Fig. 7 shows the absolute values of the correlations between the features (software metrics) and the target output (defect or correct) for the **PC3** dataset.

As a result of the analysis indicated in Section 5.3, we concluded that feature (software metric) 15 (*essential_density*) is slightly correlated with the target output and it should therefore be removed from the feature set.

Table 4 presents the best result obtained by the *DPRAR* classifier for the **PC3** dataset (preprocessed as indicated below). We should mention that the maximal interesting relational association rules of various lengths (i.e. 2-length rules vs. any length rules) were considered for this case study together with different values for the minimum confidence thresholds.

### 6.10. The **PC4** dataset

The PC4 dataset, like PC3, contains data about a flight software for earth orbiting satellite, written in C. It consists of 177 positive instances (defects) and 1110 negative instances (non-defects), meaning that there are 13.8% positive instances and 86.2% negative instances. Each instance has 37 features and the class label.

#### 6.10.1. DPRAR results

As a result of the analysis indicated in Section 5.3, we concluded that there is no feature (software metric) slightly correlated with the target output for the **PC4** dataset. Consequently, the feature set remained the same and no features were removed from it.

Table 4 presents the best result obtained by the *DPRAR* classifier for the **PC4** dataset (preprocessed as indicated below). We should mention that the maximal interesting relational association rules of various lengths (i.e. 2-length rules vs. any length rules) were considered for this case study together with different values for the minimum confidence thresholds.

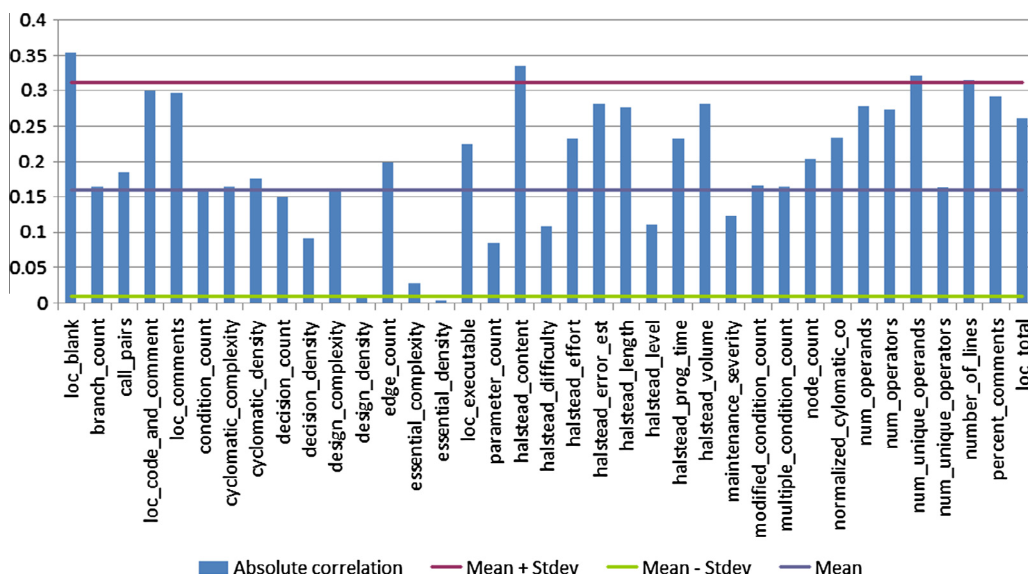

**Fig. 7.** Correlations for the **PC3** dataset.

## 7. Discussion

In this section we aim at analyzing the method proposed in this paper by emphasizing its advantages and drawbacks, as well as comparing the *DPRAR* classifier to other similar approaches existing in the software engineering defect detection literature.

The following provides a comparison between the *DPRAR* method introduced in this paper and the *CBA*2 method [2], the 1*R* classifier [7], the *Bagging* classifier [16] and the *EDER-SD* [45]. The main reason for selecting the *CBA*2, 1*R*, *Bagging* and the *EDER-SD* methods for comparison is that they were applied on datasets from the NASA repository [36], thus a comparison of the obtained results is possible is most cases. Another reason is that *CBA*2 is a classification method based on association rule mining (as *DPRAR* is), *EDER-SD* is rule based (as *DPRAR* is) and 1*R* and *Bagging* were identified in [7,16] as the classifiers with the highest accuracy among the classifiers that were experimented on the NASA datasets.

Table 5 shows comparatively, for all the case studies considered for evaluation, the *classification accuracy* (*Acc*), the *probability of detection* (*Pd*), the *specificity* (*Spec*), the *precision* (*Prec*) and the *AUC* measures obtained for the *DPRAR*, *CBA*2, 1*R*, *Bagging* and the *EDER-SD* methods. If an evaluation measure is not available for a particular classifier, this

**Table 5**
Comparative results.

| Data | Acc | | Pd | | Spec | | Prec | | AUC | |
|---|---|---|---|---|---|---|---|---|---|---|
| CM1 | CBA2 | 0.8036 | CBA2 | 0.2 | CBA | 0.885 | CBA2 | n/a | CBA2 | 0.598 |
| | 1R | 0.8816 | 1R | n/a | 1R | n/a | 1R | n/a | 1R | n/a |
| | **Bagging** | **0.8995** | Bagging | n/a | Bagging | n/a | Bagging | n/a | Bagging | 0.720 |
| | EDER-SD | 0.88 | EDER-SD | n/a | **EDER-SD** | **0.947** | EDER-SD | 0.357 | EDER-SD | n/a |
| | DPRAR | 0.8716 | **DPRAR** | **0.929** | DPRAR | 0.8632 | **DPRAR** | **0.5** | **DPRAR** | **0.896** |
| KC1 | CBA2 | 0.8371 | CBA2 | 0.461 | CBA2 | 0.910 | CBA2 | n/a | **CBA2** | **0.836** |
| | 1R | 0.831 | 1R | n/a | 1R | n/a | 1R | n/a | 1R | n/a |
| | Bagging | 0.8568 | Bagging | n/a | Bagging | n/a | Bagging | n/a | Bagging | 0.809 |
| | **EDER-SD** | **0.859** | EDER-SD | n/a | **EDER-SD** | **0.980** | EDER-SD | 0.596 | EDER-SD | n/a |
| | DPRAR | 0.823 | **DPRAR** | **0.818** | DPRAR | 0.825 | **DPRAR** | **0.628** | DPRAR | 0.822 |
| KC3 | CBA2 | 0.9091 | CBA2 | 0.333 | CBA2 | 0.962 | CBA2 | n/a | CBA2 | 0.696 |
| | 1R | n/a | 1R | n/a | 1R | n/a | 1R | n/a | 1R | n/a |
| | Bagging | n/a | Bagging | n/a | Bagging | n/a | Bagging | n/a | Bagging | n/a |
| | **EDER-SD** | **0.935** | EDER-SD | n/a | **EDER-SD** | **1** | **EDER-SD** | **0.643** | EDER-SD | n/a |
| | DPRAR | 0.83 | **DPRAR** | **0.889** | DPRAR | 0.8165 | DPRAR | 0.5246 | **DPRAR** | **0.85225** |
| PC1 | CBA2 | 0.9178 | CBA2 | 0.44 | CBA2 | 1 | CBA2 | n/a | CBA2 | 0.827 |
| | 1R | 0.9369 | 1R | n/a | 1R | n/a | 1R | n/a | 1R | n/a |
| | Bagging | 0.9332 | Bagging | n/a | Bagging | n/a | Bagging | n/a | Bagging | 0.915 |
| | EDER-SD | 0.943 | EDER-SD | n/a | **EDER-SD** | **1** | EDER-SD | 0.577 | EDER-SD | n/a |
| | **DPRAR** | **0.956** | **DPRAR** | **0.885** | DPRAR | 0.963 | **DPRAR** | **0.692** | **DPRAR** | **0.924** |
| JM1 | CBA2 | 0.7352 | CBA2 | 0.461 | CBA2 | n/a | CBA2 | n/a | CBA2 | 0.688 |
| | 1R | 0.7987 | 1R | n/a | 1R | n/a | 1R | n/a | 1R | n/a |
| | Bagging | 0.8104 | Bagging | n/a | Bagging | n/a | Bagging | n/a | Bagging | 0.742 |
| | EDER-SD | n/a | EDER-SD | n/a | EDER-SD | n/a | EDER-SD | n/a | EDER-SD | n/a |
| | **DPRAR** | **0.96** | **DPRAR** | **0.842** | **DPRAR** | **0.99**2 | **DPRAR** | **0.967** | **DPRAR** | **0.917** |
| MC2 | CBA2 | 0.6981 | CBA2 | 0.333 | CBA2 | 0.886 | CBA2 | n/a | CBA2 | 0.671 |
| | 1R | n/a | 1R | n/a | 1R | n/a | 1R | n/a | 1R | n/a |
| | Bagging | n/a | Bagging | n/a | Bagging | n/a | Bagging | n/a | Bagging | n/a |
| | EDER-SD | 0.759 | EDER-SD | n/a | **EDER-SD** | **1** | EDER-SD | 0.529 | EDER-SD | n/a |
| | **DPRAR** | **0.896** | **DPRAR** | **0.773** | DPRAR | 0.9632 | **DPRAR** | **0.919** | **DPRAR** | **0.868** |
| MW1 | CBA2 | 0.9104 | CBA2 | 0.5 | CBA2 | 0.919 | CBA2 | n/a | CBA2 | 0.86 |
| | 1R | n/a | 1R | n/a | 1R | n/a | 1R | n/a | 1R | n/a |
| | Bagging | n/a | Bagging | n/a | Bagging | n/a | Bagging | n/a | Bagging | n/a |
| | **EDER-SD** | **0.941** | EDER-SD | n/a | **EDER-SD** | **1** | EDER-SD | 0.456 | EDER-SD | n/a |
| | **DPRAR** | **0.941** | **DPRAR** | **0.889** | DPRAR | 0.947 | **DPRAR** | **0.667** | **DPRAR** | **0.918** |
| PC2 | **CBA2** | **0.992** | CBA2 | 0.455 | **CBA2** | **0.994** | CBA2 | n/a | CBA2 | 0.809 |
| | 1R | n/a | 1R | n/a | 1R | n/a | 1R | n/a | 1R | n/a |
| | Bagging | n/a | Bagging | n/a | Bagging | n/a | Bagging | n/a | Bagging | n/a |
| | EDER-SD | n/a | EDER-SD | n/a | EDER-SD | n/a | EDER-SD | n/a | EDER-SD | n/a |
| | DPRAR | 0.984 | **DPRAR** | **0.938** | DPRAR | 0.985 | **DPRAR** | **0.577** | **DPRAR** | **0.962** |
| PC3 | CBA2 | 0.8648 | CBA2 | 0.255 | CBA2 | 0.934 | CBA2 | n/a | CBA2 | 0.821 |
| | EDER-SD | n/a | EDER-SD | n/a | EDER-SD | n/a | EDER-SD | n/a | EDER-SD | n/a |
| | **DPRAR** | **0.967** | **DPRAR** | **0.85** | **DPRAR** | **0.983** | **DPRAR** | **0.877** | **DPRAR** | **0.917** |
| PC4 | CBA2 | 0.8396 | CBA2 | 0.648 | CBA2 | 0.866 | CBA2 | n/a | CBA2 | 0.885 |
| | EDER-SD | n/a | EDER-SD | n/a | EDER-SD | n/a | EDER-SD | n/a | EDER-SD | n/a |
| | **DPRAR** | **0.961** | **DPRAR** | **0.814** | **DPRAR** | **0.985** | **DPRAR** | **0.894** | **DPRAR** | **0.899** |

will be marked with "n/a". 1*R* and *Bagging* classifiers do not provide results for *PC*3 and *PC*4 datasets, therefore are not shown in Table 5. For each case study, the best result is marked with bold characters.

An accurate comparison between the *DPRAR* classifier and other defect detectors existing in the literature is not entirely possible, due to the following reasons:

- Firstly, despite using the NASA-datasets for the case study, other classifiers might have used different versions of these datasets. As we have mentioned at the beginning of Section 6, we used the cleaned version of these datasets, the one without duplicates. Other defect detector case studies do not report which version they use, but they usually give a short description of the datasets (number of instances, number of features) which does suggest the used version. Based on these descriptions, [2,7] uses the original (not cleaned) datasets and they do not mention any data cleaning steps either. [16] does not present the description of the used datasets, and does not mention data cleaning. [45] uses the uncleaned datasets as well, but for each used dataset they mention the number of inconsistent and duplicate cases, without mentioning whether they were eliminated or not. The problem with the uncleaned version of the datasets is the presence of the duplicates, which offers no guarantee that testing data would not contain instances present in the training data. Although not all methods are equally affected by duplicate instances, [14] presents an experiment with an artificial dataset,where different quantities of duplicate instances were introduced. When using a random forest decision tree learner, with 25% duplicate instances, the accuracy grew from 48.50% (value for the test after training on the dataset with no duplicates) to 65.20%, whereas for a higher percent of duplicates, the accuracy grew even higher (up to 93.50%, for 100% duplicates – each instance was present twice in the artificial dataset).
- A further reason that may affect the comparison is that different methodologies were used for testing the classifiers. For *DPRAR* a *leave-one out* cross-validation methodology is used. As indicated in [2,7], the results of the *CBA*2 and the 1*R* method on a dataset were obtained by using 70% instances from the dataset for training and the remaining 30% instances for testing. Similarly, [45] reports that two-thirds of the data was used for training and one-third was used for testing. [16] does not mention what kind of testing method was used, except that they used the WEKA [18] tool, where different settings for testing can be chosen.
- Not all evaluation measures we have used for comparison on each considered dataset are available for all classifiers. *Accuracy* is the only measure reported for every method, but since the datasets are imbalanced, accuracy alone is not sufficient. The rest of the metrics are reported only for some methods, for example *AUC* is reported for *CBA*2 and *Bagging*, and *precision* is reported for *EDER-SD*. Similarly, only *CBA*2 uses all 10 datasets that we have used, whereas *EDER-SD* uses only 6 of them, while *Bagging* and 1*R* use only 4.

From Table 5 one can observe the following:

- Considering *accuracy*, our *DPRAR* classifier beats the other classifiers on six of the datasets (PC1, JM1, MC2, MW1 – tie with *EDER-SD*, PC3 and PC4). For the rest of the datasets, the results vary as follows: for CM1, *DPRAR* is only the fourth out of the five methods, for KC1, KC3 and PC2 it is the last out of the 5, 3 and 2 methods, respectively. Still, in case of PC2 the difference is very small, only 0.008.
- *Probability of detection* is only reported for the *CBA*2 classifier, and here our *DPRAR* classifier is a lot better (differences between 0.166 and 0.729) for every dataset.
- In respect to *specificity DPRAR* has the best value for only 3 datasets (PC3, PC4 and JM1), and there are 6 datasets where *EDER-SD* has the highest *specificity*, with a value of 1 in four of the cases. For the PC2 dataset CBA2 has the highest value, but the difference between DPRAR and CBA2 is only 0.009.
- *Precision* is reported only for *EDER-SD* and *DPRAR*, and the highest values are for the *DPRAR* classifier, except for the KC3 dataset.
- Another frequently reported metric in case of classifiers is the *AUC*. For this metric, our *DPRAR* has the highest value for 9 datasets, only in case of KC1 is the *AUC* of *CBA*2 the highest.

Considering all metrics and all datasets, we can say that the *DPRAR* classifier has less good results for the **KC1** and **KC3** datasets. The reason for this may be that *DPRAR*, *CBA*2 and 1*R* were not tested using the same methodology. As it was already mentioned, the *DPRAR* method was tested using the *leave-one out* cross-validation methodology, but *CBA*2 and 1*R* do not use *cross-validation* for testing. Another cause may be that the proportion of *positive* instances is much smaller than the proportion of *negative* instances for the considered datasets and this may as well be a cause of low detection probabilities and detection accuracies.

**Table 6**
Difficulty of the used NASA datasets.

| Dataset | CM1 | JM1 | KC1 | KC3 | MC2 | MW1 | PC1 | PC2 | PC3 | PC4 |
|---|---|---|---|---|---|---|---|---|---|---|
| Difficulty | 0.1865 | 0.308 | 0.3178 | 0.3041 | 0.392 | 0.2015 | 0.1475 | 0.0430 | 0.195 | 0.2005 |

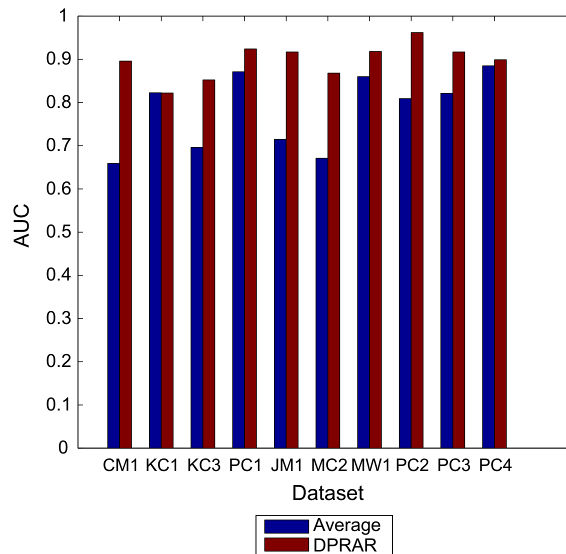


**Fig. 8.** *DPRAR* performance.

A third possible cause for the results can be related to the *difficulty* of datasets. *Difficulty* is a metric introduced by Boetticher in [4], computed for a dataset, whose value depends on how many instances are in the dataset whose nearest neighbor (excluding the class label, when computing the distances) has the same label as the instance itself (their number is denoted by *Matches*). The exact formula is: $Overall\ Difficulty = 1 - \frac{Matches}{Total\ Data\ Instances}$. The difficulties computed for the datasets used in the experimental evaluation are presented on Table 6.

From Table 6 we can see that the KC1 and the KC3 dataset have a higher difficulty than most of the other datasets, even if MC2 and JM1 have also high difficulty and the *DPRAR* classifier has good results for those datasets. Still, difficulty might have to do something with the performance of the *DPRAR* classifier, because, with some exceptions, datasets with higher difficulty have lower AUC values, which is shown by the following: if we sort the datasets decreasingly by their difficulty we have the following list: MC2, KC1, JM1,KC3, MW1, PC4, PC3, CM1, PC1, PC2; if we sort them increasingly, based on the AUC value of the *DPRAR* classifier we will have: KC1, MC2, KC3, CM1, JM1, PC3, MW1, PC1, PC2. Basically, the same datasets can be found in the first part of both lists and also in the last part of both lists (exceptions are PC4, JM1 and CM1).

We can conclude that, taking into account all evaluation measures for all considered case studies, *DPRAR* performed better in 45 measures, similarly in 1 of them and worse in 23 out of the 69 evaluation measures. Moreover, the AUC measure reported by the *DPRAR* classifier (considered in the literature one of the best evaluation measures to compare classifiers) outperforms the average AUC value reported by existing defect detectors on all considered case studies. This indicates a very good efficiency of the *DPRAR* classifier. This comparison is illustrated in Fig. 8, where the red bars correspond to the AUC values of the *DPRAR* classifier.

The results described above bring us to the conclusions that our *DPRAR* technique provides a good performance compared to other existing software defect detector models and that applying relational association rule mining for defect detection is promising. Further improvements will, certainly, increase the accuracy of the obtained results. Moreover, our *DPRAR* method is general and it can therefore be used for detecting possible defective *software entities* such as *application classes* and *subprograms*, if an appropriate representation of these entities is provided.

The *DPRAR* technique may be further extended such that for the identified defective entities to provide useful information regarding the software metrics that are likely to cause the defect. This way, software developers may obtain indications regarding the source of the defect. The current implementation of our proposal does not provide this functionality, but further work will deal with this issue.

## 8. Conclusions and further work

We have introduced in this paper a classification model based on relational association rule discovery for detecting in software systems *software entities* that are likely to be defective. Experiments were conducted in order to detect defective software modules, and the obtained results have shown that our classifier is better than, or comparable to, the classifiers already applied for software defect detection, indicating the potential of our proposal.

Further work in the relational association rules discovery will be made in order to identify and consider different types of relations between the software metrics, relations that may be relevant in the mining process. We will also investigate how the length of the rules and the confidence of the relational association rules discovered in the training data may influence the

accuracy of the classification task. Directions to hybridize our classification model, by combining it with other machine learning based predictive models [39] will be considered too. We also plan to extend our model considering fuzzy relational association rules [42] and investigating their usefulness in software defect detection.

## Acknowledgements

The authors would like to thank to the editor and the anonymous reviewers for their valuable comments and suggestions to improve the paper and the presentation. They also thank for the assistance received from the *PROMISE Software Engineering Repository* (available at: https://code.google.com/p/promisedata/). Special thanks are due to the authors from the *NASA – Software Defect Data Sets* (available at: http://nasa-softwaredefectdatasets.wikispaces.com/) who made public the cleaned versions of the NASA datasets that were used in the experimental part of the paper.

## References

[1] R. Agrawal, R. Srikant, Fast algorithms for mining association rules in large databases, in: Proceedings of the 20th International Conference on Very Large Data Bases, Morgan Kaufman Publishers Inc., San Francisco, CA, USA, 1994, pp. 487–499.
[2] M. Baojun, K. Dejaeger, J. Vanthienen, B. Baesens, Software defect prediction based on association rule classification, Open Access publications from Katholieke Universiteit Leuven urn:hdl:123456789/296322, Katholieke Universiteit Leuven (February 2011).
[3] E. Baralis, L. Cagliero, T. Cerquitelli, P. Garza, Generalized association rule mining with constraints, Inform. Sci. 194 (2012) 68–84.
[4] G.D. Boetticher, Advances in Machine Learning Applications in Software Engineering, IGI Global, 2007 (Ch. Improving the Credibility of Machine Learner Models in Software Engineering).
[5] L.C. Briand, W.L. Melo, J. Wust, Assessing the applicability of fault-proneness models across object-oriented software projects, IEEE Trans. Softw. Eng. 28 (7) (2002) 706–720.
[6] A. Campan, G. Serban, T.M. Truta, A. Marcus, An algorithm for the discovery of arbitrary length ordinal association rules, DMIN (2006) 107–113.
[7] V.U.B. Challagulla, F.B. Bastani, I.-L. Yen, R.A. Paul, Empirical assessment of machine learning based software defect prediction techniques, in: Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, WORDS '05, IEEE Computer Society, Washington, DC, USA, 2005, pp. 263–270.
[8] R. hua Chang, X. Mu, L. Zhang, Software defect prediction using non-negative matrix factorization, J. Softw. 6 (11) (2011) 2114–2120.
[9] S.R. Chidamber, C.F. Kemerer, Towards a metrics suite for object-oriented design, in: Conference of the Proceedings on Object Oriented Programming Systems, Languages, and Applications, 1991, pp. 197–211.
[10] M. D'Ambros, M. Lanza, R. Robbes, Evaluating defect prediction approaches: a benchmark and an extensive comparison, Int. J. Emp. Softw. Eng. (2011) 1–47.
[11] T. Fawcett, An introduction to roc analysis, Pattern Recogn. Lett. 27 (8) (2006) 861–874.
[12] R.B. Grady, Practical Software Metrics for Project Management and Process Improvement, Prentice-Hall Inc., Upper Saddle River, NJ, USA, 1992.
[13] D. Gray, D. Bowes, N. Davey, Y. Sun, B. Christianson, Further thoughts on precision, in: 15th Annual Conference on Evaluation & Assessment in Software Engineering, 2011, pp. 129–133.
[14] D. Gray, D. Bowes, N. Davey, Y. Sun, B. Christianson, The misuse of the NASA metrics data program data sets for automated software defect prediction, in: Proceedings of the Evaluation and Assesment in Software Engineering, 2011, pp. 96–103.
[15] L. Guo, Y. Ma, B. Cukic, H. Singh, Robust prediction of fault-proneness by random forests, ISSRE (2004) 417–428.
[16] 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, WCE 2012, vol. I, IEEE Computer Society, Washington, DC, USA, 2012, pp. 1–5.
[17] 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) (2012) 1276–1304.
[18] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I.H. Witten, The weka data mining software: an update, SIGKDD Explor. 11 (1) (2009) 10–18.
[19] J. Han, Data Mining: Concepts and Techniques, Morgan Kaufman Publishers Inc., San Francisco, CA, USA, 2005.
[20] S. Henry, D. Kafura, Software structure metrics based on information flow, IEEE Trans. Softw. Eng. 7 (5) (1981) 510–518.
[21] R.C. Holte, Very simple classification rules perform well on most commonly used datasets, Mach. Lear. (1993) 63–91.
[22] Y. Jiang, B. Cukic, Y. Ma, Techniques for evaluating fault prediction models, Emp. Softw. Eng. 13 (5) (2008) 561–595.
[23] Y. Jiang, M. Li, Z.-H. Zhou, Software defect detection with rocus, J. Comp. Sci. Technol. 26 (2) (2011) 328–342.
[24] Y. Kamei, A. Monden, S. Morisaki, K. ichi Matsumoto, A hybrid faulty module prediction using association rule mining and logistic regression analysis, in: Proceedings of the International Symposium on Empirical Software Engineering and Measurements (ESEM), 2008, pp. 279–281.
[25] K. Kaminsky, G.D. Boetticher, How to predict more with less defect prediction using machine learners in an implicitly data starved domain <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.83.4402>.
[26] M.P.J. Kaur, M. Pallavi, Data mining techniques for software defect prediction, Int. J. Softw. Web Sci. 1 (3) (2013) 54–57.
[27] N. Lavrač, B. Kavšek, P. Flach, L. Todorovski, Subgroup discovery with cn2-sd, J. Mach. Learn. Res. 5 (2004) 153–188.
[28] M. Li, H. Zhang, R. Wu, Z.-H. Zhou, Sample-based software defect prediction with active and semi-supervised learning, Auto. Softw. Eng. 19 (2) (2012) 201–230.
[29] B. Liu, W. Hsu, Y. Ma, Integrating classification and association rule mining, in: Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining (KDD), 1998, pp. 80–86.
[30] B. Liu, Y. Ma, C.-K. Wong, Data Mining for Scientific and Engineering Applications, Kluwer Academic, 2001 (Ch. Classification Using Association Rules: Weaknesses and Enhancements).
[31] V. López, A. Fernández, S. García, V. Palade, F. Herrera, An insight into classification with imbalanced data: empirical results and current trends on using data intrinsic characteristics, Inform. Sci. 250 (2013) 113–141.
[32] S.G. Maisikeli, Aspect mining using self-organizing maps with method level dynamic software metrics as input vectors, Ph.D. thesis, Graduate School of Computer and Information Sciences Nova Southeastern University, 2009.
[33] A. Marcus, J.I. Maletic, K.-I. Lin, Ordinal association rules for error identification in data sets, in: Proceedings of the Tenth International Conference on Information and Knowledge Management, CIKM '01, ACM, New York, NY, USA, 2001, pp. 589–591.
[34] Z. Marian, G. Czibula, I.G. Czibula, Using software metrics for automatic software design improvement, Stud. Inform. Control 21 (3) (2012) 249–258.
[35] T. Menzies, J. Greenwald, A. Frank, Data mining static code attributes to learn defect predictors, IEEE Trans. Softw. Eng. 33 (1) (2007) 2–13.
[36] T. Menzies, B. Caglayan, Z. He, E. Kocaguneli, J. Krall, F. Peters, B. Turhan, The promise repository of empirical software engineering data, June 2012 <http://promisedata.googlecode.com>.
[37] T. Menzies, M. Shepperd, Special issue on repeatable results in software engineering prediction, Emp. Softw. Eng. 17 (2012) 1–17.
[38] B. Minaei-Bidgoli, R. Barmaki, M. Nasiri, Mining numerical association rules via multi-objective genetic algorithms, Inform. Sci. 233 (2013) 15–24.
[39] T.M. Mitchell, Machine Learning, McGraw-Hill, New York, 1997.
[40] NASA independent verification& validation facility <http://www.nasa.gov/centers/ivv/home/index.html>.

[41] NASA software defect datasets <http://nasa-softwaredefectdatasets.wikispaces.com/>.
[42] Bin Pe, Suyun Zhao, Hong Chen, Xuan Zhou, Dingjie Chen, FARP: Mining fuzzy association rules from a probabilistic quantitative database, Inform. Sci. 237 (2013) 242–260.
[43] N.J. Pizzi, A fuzzy classifier approach to estimating software quality, Inform. Sci. 241 (2013) 1–11.
[44] M.S. Rawat, S.K. Dubey, Software defect prediction models for quality improvement: a literature study, Int. J. Comp. Sci. Iss. 9 (2) (2012) 288–296.
[45] D. Rodríguez, R. Ruiz, J.C. Riquelme, J.S.N. Aguilar-Ruiz, Searching for rules to detect defective modules: a subgroup discovery approach, Inform. Sci. 191 (2012) 14–30.
[46] M. Shepperd, Q. Song, Z. Sun, C. Mair, Data quality: some comments on the NASA software defect data sets, IEEE Trans. Softw. Eng. 99 (2013) 1 (PrePrints).
[47] F. Simon, F. Steinbruckner, C. Lewerentz, Metrics based refactoring, in: CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering, IEEE Computer Society, Washington, DC, USA, 2001, pp. 30–38.
[48] 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.
[49] Q. Song, Z. Jia, M. Shepperd, S. Ying, J. Liu, A general software defect proneness prediction framework, IEEE Trans. Softw. Eng. 37 (3) (2011) 356–370.
[50] C. Spearman, The proof and measurement of association between two things, Am. J. Psychol. 15 (1904) 72–101.
[51] S. Stehman, Selecting and interpreting measures of thematic classification accuracy, Rem. Sens. Environ. 62 (1) (1997) 77–89.
[52] P.-N. Tan, M. Steinbach, V. Kumar, Introduction to Data Mining, first ed., Addison-Wesley, Longman Publishing Co., Inc., Boston, MA, USA, 2005.