Searching for Rules to Detect Defective Modules: A Subgroup Discovery Approach

D. Rodríguez[a,*], R. Ruiz[b], J.C. Riquelme[c], J.S. Aguilar–Ruiz[b]

[a] *Department of Computer Science, University of Alcalá, Ctra. Barcelona, Km. 31.6, 28871 Alcalá de Henares, Madrid, Spain*

[b] *School of Engineering, Pablo de Olavide University, Ctra. Utrera km. 1, 41013 Seville, Spain*

[c] *Department of Computer Science, University of Seville, Avda. Reina Mercedes s/n, 41012 Seville, Spain*

Abstract

Data mining methods in software engineering are becoming increasingly important as they can support several aspects of the software development life-cycle such as quality. In this work, we present a data mining approach to induce rules extracted from static software metrics characterising *fault-prone* modules. Due to the special characteristics of the defect prediction data (imbalanced, inconsistency, redundancy) not all classification algorithms are capable of dealing with this task conveniently. To deal with these problems, Subgroup Discovery (SD) algorithms can be used to find groups of statistically different data given a property of interest. We propose EDER-SD (Evolutionary Decision Rules for Subgroup Discovery), a SD algorithm based on evolutionary computation that induces rules describing only fault-prone modules. The rules are a well-known model representation that can be easily understood and applied by project managers and quality engineers. Thus, rules can help them to develop software systems that can be justifiably trusted. Contrary to other approaches in SD, our algorithm has the advantage of working with continuous variables as the conditions of the rules are defined using intervals. We describe the rules obtained by applying our algorithm to seven publicly available datasets from the PROMISE repository

[*] Corresponding author

Email addresses: daniel.rodriguezg@uah.es (D. Rodríguez), robertoruiz@upo.es (R. Ruiz), riquelme@lsi.us.es (J.C. Riquelme), aguilar@upo.es (J.S. Aguilar–Ruiz)

showing that they are capable of characterising subgroups of fault-prone modules. We also compare our results with three other well known SD algorithms and the EDER-SD algorithm performs well in most cases.

Keywords:
Defect prediction, subgroup discovery, imbalanced datasets, rules

1. Introduction

Software Quality remains an important topic of research within the Software Engineering community. There are many definitions of software quality, but in this context we refer to software reliability, which is generally defined as the probability of failure-free software operation for a specified period of time in a given environment [43]. One way of improving software reliability and guiding the testing effort is through static metrics [15] capable of predicting fault-prone modules.

There is a wide range of defect prediction techniques using statistical methods and more recently, data mining techniques (see Section 5). This is due to the creation of a number of publicly available data repositories obtained from real projects that allow researchers and practitioners to apply data mining techniques. Examples of such repositories include PROMISE[1] and FLOSSMetrics[2]. There are however several issues that need to be considered when applying data mining techniques to defect prediction data.

First, datasets can be imbalanced. In fact, like most datasets in defect prediction, the datasets used in this work are highly imbalanced, i.e., samples of non–defective modules vastly outnumber the defective ones. In this situation, many data mining algorithms generate poor models because they try to optimize the overall accuracy but perform badly in classes with very few samples. For example, if the number of non-defective samples outnumbers the defective samples by 90%, an algorithm that always predicts a module as non-defective will obtain a very high accuracy. As a result, many data mining algorithms obtain biased models that do not take into account the minority class (defective modules in this case).

Second, although in theory having more attributes could provide further discriminant power, experience with data mining algorithms shows just the

[1] `http://promisedata.org/`

[2] `http://flossmetrics.org/`

opposite [36]. Removing irrelevant, redundant or noisy data provides immediate benefits including performance improvements with respect to speed, predictive accuracy and comprehensibility of the results.

Finally, another problem when applying data mining techniques is that data can have duplicated (i.e., identical attribute values including the class) and contradictory cases (i.e., instances with same attribute values but the class).

Recently a new set of descriptive induction algorithms categorised as Subgroup Discovery (SD) algorithms [30, 58, 59] have been proposed to discover statistically different subgroups of data with respect to a property of interest, making these types of algorithms suitable for extracting knowledge from imbalanced datasets.

In this work, we tackle the defect prediction problem through a descriptive induction process using SD. The objective is to generate useful models represented through rules characterising fault–prone modules. The induced rules allow us to determine software metrics and their thresholds that increase the probability of detecting fault–prone modules. The rules are obtained with an Evolutionary Algorithm (EA), called EDER-SD (Evolutionary Decision Rules for Subgroup Discovery). EAs have the advantage that it is possible to optimise different fitness functions, such as precision, recall or coverage, depending on the characteristics of the domain knowledge from experts.

Empirical work was performed using seven publicly available NASA datasets (CM1, KC1, KC2, KC3, MC2, MW1 and PC1) related to software defect prediction from the PROMISE repository [5]. The induced rules from EDER-SD show that our technique generates understandable and useful models that can be used by project managers or quality assurance personnel to guide the testing effort and improve the quality of software development projects. EDER-SD is also compared with three other standard techniques in SD, performing well in most cases.

The organization of the paper is as follows. Section 2 presents the most relevant concepts related to the process and techniques used in this work. Section 3 explains the modifications to a hierarchical classification algorithm to be adapted to subgroup discovery in defect prediction. Section 4 describes the experimental work and discusses the results. Section 5 describes the related work in defect prediction. Finally, Section 6 concludes the paper and outlines future research work.

2. Background

Data mining techniques can be grouped into *predictive* and *descriptive* depending on the problem at hand. From the predictive point of view, patterns are found to predict future behaviour. In fault prediction, it would correspond to the generation of classification models to predict whether a software module will be defective based on metrics from historical project data. From the descriptive point of view, the idea is to find patterns capable of characterising the data represented in such a way that domain experts can understand them. A comprehensive framework for data mining is described by Peng et al. [46].

The application of data mining in the context of defect prediction aims to extract useful and applicable knowledge from the datasets obtained from the software modules in such a way that it can be used for decision making. There are different possible representations of the learning–based models including decisions trees, decision rules, rules with exceptions, fuzzy rules, artificial neural networks, among others. However, when the output needs to be directly interpreted by end–users (e.g., project manager, quality assurance manager, testers), the readability and understandability of the representation needs to be considered (for example, in the case of rules, hierarchical rules are much harder to understand than non–hierarchical rules). In this respect, we selected rules as a representation as they are considered far more simple and intuitive than other representations.

In the following subsections we describe data mining concepts used in this work.

2.1. Reduction of Imbalanced Datasets

The existence of irrelevant and redundant features in the datasets has negative impact in most data mining algorithms. As we are dealing with datasets of collected metrics from modules, we need to consider the following. First, the larger the number of metrics collected per module, the larger the need of data samples to ensure the quality of the learned patterns due to statistical variability between patterns of different class. This problem is known as the *curse of dimensionality* [17].

Second, redundant or irrelevant features may mislead learning algorithms or cause them to overfit the data [36]. Hence, the obtained classifier is in general less accurate than the one learned from the relevant data. Conversely, a dataset with less dimensionality can in most cases improve the accuracy of models, generate simple, understandable models and the data mining algorithms can be run faster.

As stated previously like most datasets in defect prediction, the datasets used in this work are highly imbalanced, i.e., samples of non–defective modules vastly outnumber the cases of defective modules. Under this situation, when the imbalanced data is not considered, many learning algorithms generate distorted models for which (i) the impact of some factors can be hidden and (ii) the prediction accuracy can be misleading. This is due to the fact that most data mining algorithms assume balanced datasets. When dealing with imbalanced datasets, there are two alternatives, either (i) sampling or balancing techniques: *over–sampling* algorithms aimed at balancing the class distribution increasing the minority class, or *under–sampling* algorithms that balance the class removing instances from the majority classes; and (ii) to apply algorithms that are robust to this problem [51, 23, 35, 16].

A few authors have applied Feature Subset Selection (FSS) techniques to software engineering data. In the case of effort estimation, it has been reported that reduced datasets improve the estimation accuracy [29, 9, 34, 8]. It is known, however, that feature selection algorithms do not perform well with imbalanced datasets, resulting in a selection of metrics that cannot be adequate for the learning algorithms, decreasing the quality and usefulness of the rules. In previous work, we analysed the application of FSS to the datasets used in this work [50]. In particular, we applied CFS (Correlation–based Filter Selection) a feature selection algorithm based on non–linear correlations, CNS (Consistency–based filter selection) and wrappers [36]. Although the classification accuracy increased using FSS, few common metrics were selected from the datasets. This can also be observed in other works such as [12]. From the software engineering point of view, these very heterogeneous results were confusing when used to generate predictive models, for example, giving greater importance to metrics such as the number of blank lines or number of commented lines than more intuitive metrics such as complexity. Another alternative to the use of FFS is to consider weights in conjunction with the attributes. For example, Turhan and Bener [53] reports positive results with the application of *Infogain* [49] to the naïve Bayes classifiers in software defect prediction.

2.2. Subgroup Discovery

Subgroup Discovery (SD) aims to find subgroups of data that are statistically different given a property of interest [30, 58, 59, 21]. SD lies between

Table 1: Confusion Matrix for Two Classes

		Actual		
		Positive	*Negative*	
Prediction	*Positive*	True Positive (TP)	False Positive (FP) Type I error (False alarm)	*Positive Predictive Value* (PPV)= $Confidence =$ $Precision =$ $= \frac{TP}{TP+FP}$
	Negative	False Negative (FN) Type II error	True Negative (TN)	*Negative Predictive Value* (NPV)= $\frac{TN}{FN+TN}$
		$Recall =$ $Sensitivity =$ $TP_r = \frac{TP}{TP+FN}$	$Specificity =$ $TN_r = \frac{TN}{FP+TN}$	

predictive (finding rules given historical data and a property of interest) and descriptive tasks (discovering interesting patterns in data). An important difference with classification tasks is that the SD algorithms only focus on finding subgroups (e.g., inducing rules) for the property of interest and do not necessarily describe all instances in the dataset.

In general, subgroups are represented through rules with the form $Cond \rightarrow Class$ having as consequent ($Class$) a specific value of an attribute of interest. The antecedent ($Cond$) is usually composed of a conjunction of attribute–value pairs through relational operators. Discrete attributes can have the form of $att = val$ or $att \neq val$ and for continuous attributes ranges need to be defined, i.e., $val_1 \leq att \leq val_2$.

An important aspect of SD is how to measure the quality of the rules that define the subgroups in order to both (i) guide the search process of the SD algorithms and (ii) to compare them. To do so, we first describe standard measures used in classification and later modifications to those measures in SD.

One common way to evaluate the performance of classifiers is through the values of the confusion matrix [19]. Table 1 shows the possible outcomes for two classes; *True Positives* (TP) and *True Negatives* (TN) are respectively the number of positive and negative instances correctly classified, *False Positives* (FP) is the number of negative instances misclassified as positive, and *False Negatives* (FN) is the number of positive instances misclassified as negative.

Based on the previously defined values: the *true positive rate* ($TP_r =$

$\frac{TP}{TP+FN}$) is the proportion of positive instances correctly classified (also called *recall* or *sensitivity*); the *False negative rate* ($FN_r = \frac{FN}{TP+FN}$) is the proportion of positive instances misclassified as belonging to the negative class; the *True negative rate* ($TN_r = \frac{TN}{FP+TN}$) is the proportion of negative instances correctly classified (*specificity*); and finally, the *false positive rate* ($FP_r = \frac{FP}{FP+TN}$) is the proportion of negative cases misclassified (also called *false alarm rate*).

There is a trade-off between *true positive rate* and *true negative rate* as the objective is to maximise both metrics. They can be combined to form single metrics. For example, the predictive *accuracy* (*Acc*) is defined as:

$$Acc = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

Another widely used metric when measuring the performance of classifiers is the $f - measure$ [57] as an harmonic median of these two proportions:

$$f - measure = \frac{2 \cdot precision \cdot recall}{precision + recall} = \frac{2 \cdot TP}{2 \cdot TP + FP + FN} \tag{2}$$

where precision ($precision = \frac{TP}{TP+FP}$) is the proportion of positive predictions that are correct and *recall* is the TP_r previously defined.

There are also some classification measures adapted to rules and SD. We next describe the most widely used measures for SD evaluation [19, 21]:

- *Coverage* of a rule (*Cov*) is the percentage of instances covered by a rule of the induced set of rules

$$Cov(R_i) = p(Cond) = \frac{n(Cond)}{N} = \frac{TP + FP}{N} \tag{3}$$

  where R_i is a single rule, $n(Cond)$ is the number of instances covered by condition *Cond* and N is the total number of instances.

- The *Support* of a rule refers to the ratio between the number of instances satisfying both the antecedent and the consequent part of a rule and the total number of instances.

$$Sup(R_i) = \frac{n(Cond \cdot Class)}{N} = \frac{TP}{N} \tag{4}$$

where the $n(Cond \cdot Class)$ corresponds to the TP and N is the total number of instances.

- The *Specificity* is the proportion of negative cases correctly classified.

$$Spec(R_i) = \frac{\neg n(Cond \cdot Class)}{\neg n(Class)} = \frac{TN}{FP + TN} \tag{5}$$

where the $\neg n(Cond \cdot Class)$ corresponds to instances which do not satisfy both condition and target class (TN). The $n(Class)$ is the number of instances that satisfy the target class and $\neg n(Class)$ is the number of those that do not satisfy the target class.

- The *Complexity* refers to the number of tests or antecedents (conjunction attribute-value pairs) in the condition ($Cond$) of a single rule.

- *Confidence* ($Conf$), also known as *Precision* or *Positive Predictive Value* (PPV) of a rule is the percentage of positive instances of a rule, i.e. relative frequency of the number of instances satisfying the both the $Cond$ and the target $Class$ and the number of instances satisfying the condition.

$$Conf(R_i) = \frac{n(Cond \cdot Class)}{n(Cond)} = \frac{TP}{TP + FP} \tag{6}$$

- Rule *Unusualness* is measured through the *Weighted Relative Accuracy* ($WRAcc$).

$$WRAcc(R_i) = \frac{n(Cond)}{N} \cdot \left(\frac{n(Cond \cdot Class)}{n(Cond)} - \frac{n(Class)}{N} \right) \tag{7}$$

This measure represents a trade–off between the coverage of a rule, i.e., its generality ($p(Cond)$) and its accuracy gain ($p(Cond \cdot Class) - p(Class)$).

- *Significance* for a rule is measured by the likelihood ratio of a rule.

$$Sig(R_i) = 2 \cdot \sum_{k=1}^{n_c} n(Cond \cdot Class_k) \cdot log \frac{n(Cond \cdot Class_k)}{n(Class_k) \cdot p(Cond)} \tag{8}$$

where n_c is the number of values of the target class. Therefore, considering a binary problem as in this case:

$$Sig(R_i) = 2 \cdot \left(TP \cdot log \frac{TP}{Def \cdot (\frac{TP+FP}{N})} + TN \cdot log \frac{TN}{NonDef \cdot (\frac{TP+FP}{N})} \right) \tag{9}$$

where Def is the number of faulty modules in the dataset and $NonDef$ is the number of non-defective modules contained in the dataset.

- $Lift$ (also known as $interest$) measures how many times more often the $Cond$ and the $Class$ occur together than expected if they where statistically independent.

$$Lift(R_i) = \frac{Conf(R_i)}{Sup(Cond)} = \frac{p(Cond \cdot Class)}{p(Cond) \cdot p(Class)} = \frac{TP \cdot N}{(TP + FP) \cdot Def} \tag{10}$$

Currently, a number of SD algorithms have been proposed since the concept was introduced by Wrobel [58] with the EXPLORA algorithm. A comprehensive survey of SD algorithms can be found in Herrera et al. [21]. In this work, we also compare our algorithm, EDER-SD, described in the next section with the following classical algorithms.

The Subgroup Discovery algorithm SD [18] is a covering rule induction algorithm that using beam search aims to find rules that maximise $q_g = \frac{TP}{FP+g}$, where g is a generalisation parameter that allow us to control the *specificity* of a rule, i.e., the balance between the complexity of a rule and its accuracy.

The CN2-SD [32] algorithm is an adaptation of the CN2 classification rule algorithm [11]. CN2-SD, like the original algorithm, consists of a search procedure based on beam search but the CN2-SD algorithm uses *unusualness* ($WRAcc$) as a quality measure of the induced rules and incorporates weights into the samples. Discretisation is also required for continuous attributes and an entropy based discretisation method (entropy MDL – Minimum Description Length) is used internally by the algorithm so different rules can have different ranges for the same attribute. Therefore, there is no need to discretise continuous attributes as a preprocessing step.

```
if Conditions then Class
  else if Conditions then Class
     else if Conditions then Class
     ...
       else "Unknown class"
```

Figure 1: Hierarchical Set of Rules (HIDER)

APRIORI-SD [24] is also an adaptation of the APRIORI-C [22] which in turn is a modification for classification tasks of the well–known rule APRIORI association algorithm [1].

There are other approaches to SD that do not fit with the representation needed for defect prediction (e.g. Železný and Lavrač [56] describe how relational rule learning is adapted to subgroup discovery) or improvements to these algorithms that are out of the scope of this paper (e.g., Cano et al. [7] make the previous CN2-SD algorithm scalable to large size datasets using instance selection).

3. EDER-SD

In this work, we propose EDER-SD (Evolutionary Decision Rules for Subgroup Discovery), an Evolutionary Algorithm (EA) [41] to characterise the minority class. EDER-SD is a robust algorithm written in C++ capable of dealing with imbalanced data that generates rules only for the defective modules, the class we are interested in.

To do so, we modified HIDER (HIerarchical DEcision Rules) [2], a sequential covering EA that produces a hierarchical set of rules, i.e., an instance will be classified by the ith rule if it does not match the conditions of the $(i-1)$th precedent rules (Figure 1). The rules are sequentially obtained until the search space is totally covered.

In order to apply EAs to optimisation problems, we need to (i) select an internal representation of the space to be searched in and (ii) define a function that assigns fitness to candidate solutions. Both components are of paramount importance for the successful application of the EAs to the problem of interest.

The representation of an individual, i.e., a rule in our case, in HIDER is a tuple of real values as is shown in Figure 2. The l_i and u_i values represent the interval for the the ith attribute. The last position is the label representing

l_1	u_1	l_2	u_2	...	l_n	u_n	$Class$
a_1		a_2			a_n		

Figure 2: Representation of Rules

the $Class$. When l_i for an attribute a_i is equal to its minimum value ($l_i = min(a_i)$) such constrain will not be part of the rule and equally when the u_i value for an attribute is equal to its maximum value ($u_i = max(a_i)$). For example, in the first case the rule would be $[l_i, v]$ and in the second case $[v, u_i]$, where v is any value within the range of the attribute. If both values are equal to their respective boundaries, it means that the attribute is not relevant and will not appear in the rule. EDER-SD maintains the same representation for individuals.

HIDER follows an *Iterative Rule Learning* methodology such that in each iteration an execution of the EA is performed to induce a rule [55]. Instances covered by the induced rule after an execution of the EA are removed from the dataset for the next execution so that only new instances will be covered (and as a consequence, it produces the hierarchy due to fact that rules need to be applied in the induced order). This process is repeated until all instances are covered. HIDER is an algorithm that induces a set of decision rules for all classes. Therefore, in the random generation of the individuals of the initial population individuals are selected to cover all existing labels, and it is the evolutionary process which will provide the best option (rule) in each iteration.

The main differences between EDER-SD and HIDER so that EDER-SD is suitable for subgroup discovery are as follows. The first one is in the process of selecting the initial population at the beginning of each evolutionary process. HIDER generates individuals (rules) for each class, however EDER-SD does not search for rules to classify the all data but only to describe one particular class. In EDER-SD, the generation of each rule in the initial population randomly selects an instance that corresponds to the class of interest (a defective module in this case). From this instance, a rule is generated so that each interval $[l_i, u_i]$ includes the attribute values for that instance. More formally, let e be an instance from the training dataset with label def_module and attribute values $(a_1, a_2, \cdots, a_n)$. Then, n weights are randomly generated $w_i \in [0, 1]$. Finally, a rule is generated as follows:

$$l_i = a_i - w_i \cdot range(a_i)$$

$$u_i = a_i + w_i \cdot range(a_i)$$

$$class = def\_module$$

where $range(a_i)$ is the $max(a_i) - min(a_i)$ for each attribute.

The second difference lies in the fact that HIDER removes the covered instances from the training data after each execution of the EA (i.e., the instances covered by the rule selected) and as a consequence, the induced rules are hierarchical. EDER-SD, on the other hand, does not remove instances from the training file but those instances covered by the rule are penalised.

Adding weights to instances instead of removing them produces the induction of more complex rules (i.e., with larger number of conditions) increasing the precision of the rules. This can also generate pyramidal rules, where the first rule (with fewer conditions) covers a large number of instances (high support) but with low precision. The rest of the rules in the pyramid keep adding conditions decreasing the support but increasing the precision.

The final difference between HIDER and EDER-SD is related to the fitness function. While in the original HIDER, the fitness function used is to maximise *accuracy*, in EDER-SD the fitness function can be any of the metrics used in SD, such as $WRAcc$ or $Lift$. Thus, rules are adapted to whatever measure is the most adequate in the domain. Figure 3 shows the EDER-SD algorithm where the classification algorithm was transformed into an algorithm for the extraction of subgroups to generate descriptive models based on rules.

The EDER-SD procedure works over a set of weighted of instances whose weights are initialised to one. Each execution of the evolutionary algorithm (*EvoAlg*) generates one rule, r, which is added to the total set of rules, R. The weights of the covered instances are decreased by 10% each time and will be used to evaluate the fitness of posterior rules (*Evaluation* procedure within *EvoAlg*). The functions inside the *EvoAlg* procedure are the classical evolutionary algorithm functions:

- The *Inititialise* function generates an initial set of rules covering a set of the target class (as previously explained).

- The *Evaluation* function assigns to each rule a value according to its fitness. As stated previously, different evaluation functions can be selected for each execution such as *accuracy*, *sensitivity*, $significance$, $f - measure$, $lift$ or $WRAcc$. In order to calculate such measures,

```
**Procedure** EDER-SD(E,R)
  $E$ is instance set $\bigcup$ Weights
  $R$ is the set of rules
  $c$ is the target class
  $P$ is the population

  $R := \emptyset$
  $E' := E$
  **while** $|E'| > |\{e \in E|\text{class}(e) == c\}|$
    $r := \text{EvoAlg}(E')$
    $R := R \oplus \{r\}$
    $E' := modifyWeights(E', R)$
  **end_while**
**end**

**Procedure** *EvoAlg*(E)
  $i := 0$
  $P_0 := \textit{Initialise}(E)$
  $\textit{Evaluation}(P_0, E)$
  **while** $i <$ num_generations
    $i := i + 1$
    **for** $j \in \{1, ..., |P_{i-1}|\}$
      $\bar{x} := \textit{Selection}(P_{i-1}, i, j)$
      $P_i := P_i + \textit{Recombination}(\bar{x}, P_{i-1}, i, j)$
    **end_for**
    $\textit{Evaluation}(P_i, E)$
  **end_while**
  **return** $\textit{best\_of}(P_i)$
**end_*EvoAlg***
```

Figure 3: EDER-SD Algorithm

the weighted instances are used for the $n(Cond)$ and $\neg n(Cond)$ expressions. The $n(Cond)$ represents the sum of weights that satisfy the condition (antecedent of the rule) and the $n(Cond \cdot Class)$ is the sum of those instances that satisfy the condition and belong to the target class. Therefore, during the first iteration of the *EvoAlg* procedure, all instances are equally considered. However, as more iterations are performed, the fitness values of the instances already covered are penalised.

- The *Selection* function selects the rules in a generation to be recombined according a to their fitness measure. As in the HIDER algorithm, EDER-SD uses the roulette-wheel algorithm.
- The *Recombination* function is the crossover operator. We have also applied the real code crossovers [2], an extension of BLX-alpha adapted to individuals coded as interval [13].
- Finally, the *best_of* function returns the best rule according to the fitness measure used.

4. Experimental Work

In this section we firstly describe the datasets used in this work. Secondly, we present the induced rules characterising defective modules for each of the datasets. Thirdly, we compare the rules induced using EDER-SD and three other well-known SD algorithms as well as a validation considering a splitting criterion. Finally, threats to validity of the empirical work are considered.

4.1. Datasets

In this paper, we have used the CM1, KC1, KC2, KC3, MC2, MW1 and PC1 datasets available in the PROMISE repository [5], to generate models for defect classification. These datasets were created from projects carried out at NASA[3].

Table 2 shows the number of instances for each dataset, the number of defective, non-defective modules and their percentage, number of duplicates, inconsistencies (equal values for all attributes of an instance but the class)

[3] `http://mdp.ivv.nasa.gov/`

Table 2: Datasets used in this Work

Data	*# inst*	*Non-def*	*Def*	*% def*	*Dupl*	*Inconst*	*Lang*
CM1	498	449	49	9.83	56	1	C
KC1	2,109	1,783	326	15.45	897	20	C++
KC2	522	415	107	20.49	147	6	C++
KC3	458	415	43	9.38	132	1	Java
MC2	161	109	52	32.30	3	1	C++
MW1	434	403	31	7.69	21	3	C++
PC1	1,109	1,032	77	6.94	155	6	C

and programming language. It can be observed that all datasets are highly imbalanced, varying from approximately 7% to 20% with a large number of duplicate instances.

All datasets contain the same 22 attributes composed of 5 different metrics for lines of code, 3 McCabe metrics [37], 4 base Halstead metrics [20] and 8 derived Halstead metrics that have been discarded (see subsection 2.1), a branch–count, and the last attribute is *problems* with 2 classes (whether a module has reported defects). Table 3 summarizes the metrics selected from the datasets in this study.

The McCabe metrics are based on the count of the number of paths contained in a program based on its graph. To find the complexity, the program, module or method of class in an object oriented program is represented as a graph, and its complexity is calculated as $v(g) = e - n + 2$, where e is the number of edges of the graph and n is the number of nodes in the graph.

The cyclomatic complexity metric measures quantity, but McCabe also defined *essential complexity*, $ev(g)$, to measure the quality of the code (penalising what is known as *spaghetti* code). Structured programming only requires sequences, selection and iteration structures, and the essential complexity is calculated in the same manner as cyclomatic complexity but from a simplified graph where such structures have been removed. The *design complexity* metric ($iv(g)$) is similar but takes into account the calls to other modules.

The other metrics used in this experiment are the Halstead's *Software Science* metrics. They are based on simple counts of tokens grouped into (i) *operators* such as keywords from programming languages, arithmetic operators, relational operators and logical operators and (ii) *operands* that include variables and constants.

Table 3: Attribute Definition Summary

	Metric	*Definition*
McCabe	*LoC*	McCabe's Lines of code
	$v(g)$ $ev(g)$ $iv(g)$	Cyclomatic complexity Essential complexity Design complexity
Halstead base	*uniqOp* *uniqOpnd* *totalOp* *totalOpnd*	Unique operators, n_1 Unique operands, n_2 Total operators, N_1 Total operands, N_2
Branch	$branchCount$	No. branches of the flow graph
Class	true, false	Reported defects?

These sets of metrics (both McCabe and Halstead) have been used for quality assurance during (i) development to obtain quality measures, code reviews etc., (ii) testing to focus and prioritize testing effort, improve efficiency etc. and (iii) and maintenance as indicators of comprehensibility of the modules etc. Generally, the developers or maintainers use rules of thumb or threshold values to keep modules, methods etc. within certain range. For example, if the cyclomatic complexity ($v(g)$) of a module is between 1 and 10, it is considered to have a very low risk of being defective; however, any value greater than 50 is considered to have an unmanageable complexity and risk. For the essential complexity ($ev(g)$), the threshold suggested is 4 etc. Although these metrics have been used for long time, there are no clear thresholds, for example, although McCabe suggests a threshold of 10 for $v(g)$, NASA's in–house studies for this metric concluded that a threshold of 20 can be a better predictor of a module being defective.

4.2. Rules found with EDER-SD

We next show the most relevant rules obtained with our tool EDER-SD for each dataset. In order to run the genetic algorithms, we need to define a number of parameters. For all datasets, the population size was 100 individuals and each execution of the evolutionary algorithm ran 100 generations. We must also define a minimum support which is approximately 10% of the number of defective modules contained in the dataset. The rules presented

Table 4: Selected EDER-SD Rules for the CM1 dataset

#	*Rule*	*# Def*	*# Non Def*
1	$6 \leq v(g) \wedge 35 \leq uniqueOpnd \wedge 64 \leq totalOpnd$	22	62
2	$82 \leq LoC \wedge 22 \leq uniqueOp$	13	21
3	$82 \leq LoC \wedge 22 \leq uniqueOp \wedge 190 \leq totalOpnd$	12	16
4	$71 \leq LoC$	17	27
5	$71 \leq LoC \wedge 22 \leq uniqueOp$	16	25
6	$71 \leq LoC \wedge 22 \leq uniqueOp \wedge 190 \leq totalOpnd$	12	18

here are the result of several executions with different fitness functions but for the sake of brevity and space, only the most relevant ones are shown.

4.2.1. CM1 Dataset

As it can be seen in Table 4 in relation to the CM1 dataset, the ratio between defective and non-defective modules for the first rule is about 26% (22/84). Although it seems a low value, it is worth noting that the percentage of unbalance of this dataset is 10%, with only 49 defective modules out of 498 modules contained in the dataset. Therefore, the probability of finding a defective module has been increased considerably. This rule was obtained by maximising *sensitivity* (0.44).

The second and third rules cover fewer modules than the first rule but the ratio between defective and non-defective modules increases to 38% and 43% respectively. These rules were obtained optimising *accuracy* (0.89) or $lift$ (4.35) and rule 3 can also be obtained maximising the $f - measure$ (0.36).

Rules 4 to 6 show the effect of decreasing the weights of instances already covered by the EA, adding a new condition to the precedent rule and showing a pyramidal effect. Rule 4 (obtained by maximising the $f - measure$) just considers *LoC* as single condition achieving a precision of 38%, which means that almost 40% of the modules with more than 71 *LoC* will be defective. Such a threshold is relatively close to the 60 *LoC* suggested by the McCabe IQ[4] tool and the NASA repository. In rules 5 and 6, when new conditions are added, precision increases (64% and 67% for rules 5 and 6 respectively) but support decreases (41 and 30 modules respectively out of the 498 contained in the dataset). These rules can be obtained maximising *precision* (0.4), $WRAcc$ (0.25) or $significance$ (11.17).

[4] `http://www.mccabe.com/`

Table 5: Selected EDER-SD Rules for KC1 dataset

#	*Rule*	*# Def*	*# Non Def*
1	$93 \leq LoC$	40	33
2	$93 \leq LoC \wedge 17 \leq uniqOp$	39	29
3	$4 \leq iv(g) \wedge 69 \leq totalOpnd$	76	54
4	$4 \leq iv(g) \wedge 69 \leq totalOpnd \wedge LoC \leq 78$	27	10
5	$3 \leq ev(g) \wedge 4 \leq v(g)$	100	159
6	$3 \leq ev(g) \wedge 4 \leq v(g) \wedge 17 \leq uniqOp$	71	72
7	$5 \leq branchCount$	204	394
8	$9 \leq branchCount$	134	196
9	$9 \leq branchCount \wedge 9 \leq uniqOp$	134	194

4.2.2. KC1 Dataset

As shown by the first two rules in Table 5, modules with a large number of *LoC* or *uniqOp* have a higher probability of being defective. The limits found by EDER-SD are 93 and 17 respectively, maximising $specificity$ (0.98). Although the number of $defective$ modules covered by the rules is larger than the non defective, both rules have low support as they cover a relatively small number of modules: 73 for the first rule with a single condition ($93 \leq LoC$) and 68 for the second one with both conditions ($93 \leq LoC \wedge 17 \leq uniqOp$).

EDER-SD also found rules combining lines of code and complexity. For modules with large complexity but a relatively small number of *LoC*, the probability of the module being defective increases. For example, for rule 3 in Table 5 ($4 \leq iv(g)$ and $69 \leq totalOpnd$), its ratio is 58% (76 out of 130). However, when the size of the module is limited to 78 *LoC*, the ratio of defective modules is 72%. In other words, rule 4 states that small modules with high complexity tend to be fault-prone. This rule was obtained by maximising *precision* (0.73) whereas rule 3 was obtained by maximising either *accuracy* (0.86) or $significance$ (54.48).

The combined threshold values for $ev(g)$ and $v(g)$ complexity metrics are 3 and 4 respectively (rule 5 obtained maximising *sensitivity* (0.3)), achieving a ratio of 38% for defective modules. Adding a new constraint about unique operators ($17 \leq uniqOp$) to these complexity values increases the ratio to 50% with more than 70 modules covered by the rule. For this dataset, the ratio of defective vs. non defective modules is just 15%, therefore the probability of finding a defective module also increases considerably when compared with random selection.

Other rules found by EDER-SD relate the number of branches with

Table 6: Selected EDER-SD Rules for KC2 Dataset

#	*Rule*	*# Def*	*# Non Def*
1	$58 \leq LoC \wedge 5 \leq ev(g) \wedge 17 \leq branchCount$	36	5
2	$17 \leq uniqOpnd$	85	78
3	$17 \leq uniqOpnd \wedge 30 \leq totalOpnd$	82	75
4	$17 \leq uniqOpnd \wedge 50 \leq totalOpnd$	70	41
5	$30 \leq totalOpnd$	84	84
6	$71 \leq totalOpnd$	51	24
7	$120 \leq totalOpnd$	31	4
8	$9 \leq v(g)$	46	23
9	$9 \leq v(g) \wedge 4 \leq ev(g)$	42	14
10	$9 \leq v(g) \wedge 4 \leq ev(g) \wedge 5 \leq iv(g)$	31	12
11	$9 \leq v(g) \wedge 4 \leq ev(g) \wedge 5 \leq iv(g) \wedge 75 \leq totalOpnd$	27	0
12	$9 \leq v(g) \wedge 4 \leq ev(g) \wedge 5 \leq iv(g) \wedge uniqOp \leq 24 \wedge$ $75 \leq totalOpnd$	31	4

unique operators. The rule ($5 \leq brachCount$) covers 204 defective modules out of 598 modules (34%) but if this threshold is increased to 9, the ratio also increases to 40%. However, when considering both conditions ($9 \leq brachCount \wedge 9 \leq uniqOp$), the rule covers 194 defective modules (almost 2/3 of the 313 defective samples included in the dataset) with ratio of 60%. That is to say that modules with branch count and unique operator values larger than 9 have a 60% of probability of being defective. The last three rules can be obtained by maximising the $f-measure$ (0.44), $significance$ (54.94) or $WRAcc$ (0.05).

4.2.3. KC2 Dataset

The first rule for the KC2 dataset in Table 6 combines three parameters (LoC, $ev(g)$ and $branchCount$), which are very close to the thresholds suggested by the McCabe IQ tool and the empirical values from the NASA repository (60, 4 and 19 respectively). This single rule already covers approximately 1/3 of the defective modules (36 out of a total of 107) and it was obtained maximising the $specificity$ (0.99).

Rules 2 to 7 in Table 6 show the relationship between *uniqOpnd* and *totalOperands* attributes. The threshold suggested by the McCabe IQ tool for *uniqOpnd* is 20, which is quite close to the value of 17 suggested by these rules. As in previous datasets, when new conditions are added to a rule, the number of modules that are covered by the rule (*support*) decreases. In this case, adding the $50 \leq totalOperands$ condition decreases the number

Table 7: Selected EDER-SD Rules for KC3 Dataset

#	*Rule*	*# Def*	*# Non Def*
1	$3 \leq iv(g) \wedge 12 \leq uniqOp \wedge 40 \leq totalOp$	32	94
2	$20 \leq loc \wedge 3 \leq iv(g)$	25	80
3	$79 \leq loc \wedge 10 \leq iv(g) \wedge uniqOp \leq 25 \wedge totalOpnd \leq 308 \wedge branchCount \leq 41$	7	0
4	$79 \leq loc \wedge 10 \leq iv(g) \wedge uniqOp \leq 25$	7	1
5	$3 \leq v(g) \wedge ev(g) \leq 3 \wedge iv(g) \leq 5 \wedge uniqOp \leq 20 \wedge 32 \leq UniqOpnd$	7	1
6	$88 \leq loc \wedge 20 \leq uniqOp \wedge UniqOpnd \leq 106 \wedge branchCount \leq 41$	7	1
7	$ev(g) = 1 \wedge 58 \leq UniqOpnd$	5	0
8	$57 \leq loc \wedge ev(g) \leq 3 \wedge 58 \leq UniqOpnd$	6	0
9	$ev(g) = 1 \wedge 3 \leq iv(g) \wedge 12 \leq uniqOp \leq 14 \wedge 40 \leq totalOp \wedge totalOpnd \leq 39$	7	1
10	$10 \leq iv(g) \wedge uniqOp \leq 25 \wedge 48 \leq UniqOpnd \wedge 22 \leq branchCount$	7	2

of modules covered from 163 (rule 2) to 111 (rule 4), but rule's *precision* is increased from 52% (82 out of 163) to 63% (60 out of 111). Rules 2 to 4 were obtained maximising the $f - measure$ (0.64), *accuracy* (0.85) or $WRAcc$ (0.099), rules 5 and 6 *sensitivity* (0.78), and $specificity$ (0.99) for rule 7.

Rules 8 to 12 do not modify the limits but keep adding conditions to the precedent rule creating the pyramidal effect previously mentioned. Again, there is a trade-off between support and precision, slightly reducing the number of modules covered by the rule, increases the probability of the module being defective. These rules were obtained maximising the $lift$ (4.32).

4.2.4. KC3 Dataset

Table 7 shows the selected rules for the KC3 dataset. The first two rules are obtained using measures that favour high support, e.g., TP_r or $WRAcc$. With such measures, the rules obtained cover a large number of defective modules (32 out of 43) but penalising *Accuracy* and $Specificity$ measures. On the other hand, the rest of the rules try to maximise the latter measures resulting in a very low rate of false positives at the expense of a low support (only seven defective modules). In addition, the rules are composed of a large number of conditions (with the exception of rules 4 and 7).

Table 8: Selected EDER-SD Rules for MC2 Dataset

#	*Rule*	*# Def*	*# Non Def*
1	$2 \leq iv(g) \wedge 15 \leq uniqOp$	27	23
2	$15 \leq uniqOp$	32	36
3	$5 \leq v(g) \wedge 2 \leq iv(g)$	30	23
4	$14 \leq uniqOp \wedge 11 \leq UniqOpnd \wedge totalOpnd \leq 38$	9	2
5	$9 \leq loc \leq 15 \wedge 11 \leq UniqOpnd \wedge 31 \leq totalOp$	8	1
6	$32 \leq loc \wedge 5 \leq ev(g) \wedge 2 \leq iv(g) \wedge 18 \leq uniqOp$	17	4
7	$8 \leq v(g) \wedge 3 \leq iv(g)$	22	8
8	$8 \leq v(g) \wedge 137 \leq totalOpnd$	17	6
9	$totalOp \leq 24 \wedge 11 \leq totalOpnd \wedge branchCount \leq 1$	5	0

Table 9: Selected EDER-SD Rules for MW1 Dataset

#	*Rule*	*# Def*	*# Non Def*
1	$4 \leq iv(g) \wedge 38 \leq uniqOpnd$	17	34
2	$10 \leq uniqOp \wedge 38 \leq uniqOpnd \wedge 54 \leq totalOpnd$	18	36
3	$52 \leq loc \wedge v(g) \leq 26 \wedge ev(g) \leq 10 \wedge 8 \leq iv(g) \wedge 86 \leq totalOpnd \wedge 25 \leq branchCount$	8	1
4	$52 \leq loc \wedge v(g) \leq 26 \wedge 8 \leq iv(g) \wedge 86 \leq totalOpnd \wedge 25 \leq branchCount$	10	4
5	$loc \leq 67 \wedge 12 \leq v(g) \wedge 6 \leq ev(g) \wedge 53 \leq UniqOpnd \wedge branchCount \leq 33$	7	0
6	$10 \leq uniqOp \leq 17 \wedge 38 \leq UniqOpnd \wedge 54 \leq totalOpnd \leq 110 \wedge branchCount \leq 25$	8	2
7	$43 \leq loc \wedge v(g) \leq 15 \wedge ev(g) \leq 1 \wedge 5 \leq iv(g) \wedge 11 \leq uniqOp \leq 36 \wedge 38 \leq UniqOpnd$	6	1

4.2.5. MC2 Dataset

The selected rules for the MC2 dataset are shown in Table 8. As with the KC3 dataset, the first three rules maximise $WRAcc$ or TP_r while the remaining rules maximise $Specificity$ or $Precision$. However, it is worth noting that the differences between the *Accuracy* values in both sets of rules are very small, i.e, 0.6 to 0.7 for the first set and 0.7 to 0.75 for the second one. Rules 6, 7 and 8 maximise *Accuracy* and $Significance$ resulting in similar values for $iv(g)$ and $UniqOp$.

4.2.6. MW1 Dataset

Table 9 shows the results for the MW1 dataset. Again, the first two rules maximise $WRAcc$ and TP_r and the rest of the rules $Lift$ or $Precision$. However, the differences of the values of the *Accuracy* measure are also very small (0.88 for the first of rules and 0.94 for the rest of the rules). The same

Table 10: Selected EDER-SD Rules for PC1 Dataset

#	*Rule*	*# Def*	*# Non Def*
1	$4 \leq v(g) \wedge 13 \leq uniqOp \wedge 18 \leq uniqOpnd \wedge 40 \leq totalOp \wedge 29 \leq totalOpnd$	48	258
2	$3 \leq iv(g) \wedge 7 \leq branchCount$	41	285
3	$64 \leq LoC \wedge 3 \leq iv(g)$	20	56
4	$84 \leq LoC \wedge 3 \leq iv(g)$	16	24
5	$90 \leq LoC \wedge 17 \leq v(g) \wedge 3 \leq iv(g) \wedge 21 \leq uniqOp \leq 35 \wedge uniqOpnd \leq 102 \wedge branchCount \leq 57$	11	2
6	$90 \leq LoC \wedge 17 \leq v(g) \wedge 3 \leq iv(g) \wedge 20 \leq uniqOp \leq 34 \wedge totalOpnd \leq 304 \wedge branchCount \leq 57$	9	1
7	$58 \leq LoC \wedge 16 \leq uniqOp \leq 31 \wedge 59 \leq uniqOpnd \wedge 159 \leq totalOp \wedge 125 \leq totalOpnd \wedge branchCount \leq 24$	10	1
8	$84 \leq LoC \wedge ev(g) \leq 9 \wedge 20 \leq uniqOp \leq 35 \wedge 66 \leq uniqOpnd \wedge totalOp \leq 544$	12	1

occurs in $Specificity$ (0.9 vs. 0.98) or $Significance$ (12 vs. 15). As with the previous dataset, rules with very low values of false positives are composed of a large number of conditions.

4.2.7. PC1 Dataset

Table 10 shows rules for the PC1 dataset. The first rule shows the threshold found by EDER-SD relating for the cyclomatic complexity and metrics related to operands and operators. This rule covers a large proportion of the defective modules (62%) contained in the dataset but provides a low precision of about 16%. It worth noting that this dataset is the most imbalanced one with only 77 samples of defective modules out of 1109 (7% of the total) and therefore when this rule applies, the probability of finding a defective module is more than double than using random selection.

The second rule establishes the limits for the design complexity ($iv(g)$) and the number of branches ($branchCount$). Like the previous one, this rule covers a large number of the defective modules (53% of the total) but has low precision (14%).

EDER-SD found two limits for the number of lines of code LoC, 64 and 84, as shown by the third and fourth rules. The latter limit is more restrictive providing higher precision with a relatively low support as it covers 16 defective modules out of 77 (21% of the total). Finally, the rest of the rules are further examples of rules providing high precision but low support.

4.2.8. General Observations across all Datasets

Each dataset provides examples of concrete applications within a domain (real-time, instruments, control systems) and programming language (C, C++ or Java). Thus, it is difficult to extrapolate the rules found for any of the datasets. It is possible, however, to observe some common trends. As it can be observed from Tables 4 to 10, almost all induced rules are formed by conditions providing lower limits for the metrics that compose the rules. As expected, we found defective modules for high values of the metrics (e.g. lines of code, complexity, number of operands, operators and branches). This is especially true for rules with high support values in each of the datasets. After analysing the rules, it can be seen in general that the limit for LOC is 60; for the complexity measures, the limit of $v(g)$ is between 4 and 9, and for $iv(g)$ is from 2 to 4. The number of operators and operands have wider ranges (*uniqOp*: 10-15, *uniqOpnd*: 17-38, *totalOp*: 40, *totalOpnd*: 29-69). Finally, the lower limit for the $branchCount$ metric is between 5 and 7. On the rare occasions where an upper limit appears, the number of defective modules covered by those rules (support) is very low, which means that they are very specific rules adjusted to a particular subset of the data (e.g., high complex modules with very few operators) and they are also very difficult to generalise.

It is worth noting the consistency of the values established by EDER-SD for the lower threshold values in relation to those obtained by McCabe IQ tool and the NASA MDP web site (*LoC*: 60, $v(g)$: 10, $ev(g)$: 4, $iv(g)$: 7, *uniqOp*: 20, *uniqOpnd*: 20, *totalOp*: 30, *totalOpnd*: 30, $branchCount$: 19) but such thresholds set by the McCabe IQ tool are meant to be used individually. However, EDER-SD thresholds are delineated in several dimensions (metrics), allowing better adjustment of the values that lead to faulty modules.

4.3. Comparison of EDER-SD with other Algorithms

As stated previously, defect prediction datasets are in general highly imbalanced and with a large number of inconsistencies (duplicates or contradictory cases). To deal with these problems using SD, we have developed an evolutionary algorithm called EDER-SD that aims to find simple and easily understandable rules (few conditions) but capable of predicting fault-prone modules. In this subsection, we now compare EDER-SD with three other well-known SD algorithms widely cited in the literature, SD, CN2-SD and

APRIORI-SD (see Section 2.2). These algorithms have been implemented in the Orange data mining toolkit[5] as part of a SD plug-in[6].

An important issue with the application of the APRIORI-SD algorithm is that we were only capable of inducing rules for the KC2 dataset which corresponds with the more balanced one (20%). For the rest of the datasets the algorithm could not find any rules, no matter the variations in the parameters of the algorithm. The reason may reside in the fact that datasets are so highly imbalanced that rules do not achieve the minimum support and precision required, i.e., the number of samples covered by a rule is not enough to fulfil the quality criteria. Furthermore, the APRIORI-SD algorithm not only discretises continuous attributes but discrete ones are also binerised.

Tables 11 to 13 show the set of rules induced for the KC2 dataset using the default parameters used in the Orange tool (*minimal support*=5%; *minimal precision*=80% *beam width*=20; *generalization* parameter, $g = 5$ and no. of times a covered instance can be used before removed, $k = 5$). We do not show the rules for rest of the datasets for the sake of brevity and space. These three algorithms are deterministic when the parameters are fixed.

The first observation is the disparity in the number of rules induced. The CN2-SD algorithm generates very few rules for all four datasets (just two or three rules) but those rules are in general quite good. In order to validate EDER-SD against these classical SD algorithms from a machine learning perspective, we also used a splitting criterion. We divided the datasets into training and testing with two-thirds and one-third of the samples respectively. Table 14 shows the comparative results for the maximum values obtained for the evaluation measures described in Section 2.2 using all algorithms and testing datasets, where the $+$, $-$ and $=$ corresponds to EDER-SD performing better, worst or equal to other algorithm respectively. The reason for using maximum values is that all algorithms induce a very different number of rules (CN2-SD induces only two or three rules for these datasets). Therefore, it would not be *fair* to compare average values. Furthermore, the different algorithms optimise different functions. EDER-SD performed better than the other three algorithms counting the number of times that it obtained maximum values (the no. of pluses, minuses and equals in Table 14 shows if EDER-SD performed better, worst or equal than the other

[5] http://www.ailab.si/orange/

[6] http://kt.ijs.si/petra_kralj/SubgroupDiscovery/

Table 11: SD induced rules for the KC2 Dataset

#	*Rule*	*# Def*	*# Non Def*
1	$ev(g) > 4 \wedge totalOpnd > 117$	28	5
2	$iv(g) > 8 \wedge uniqOpnd > 34 \wedge ev(g) > 4$	31	6
3	$loc > 100 \wedge uniqOpnd > 34 \wedge ev(g) > 4$	31	7
4	$loc > 100 \wedge iv(g) > 8 \wedge ev(g) > 4$	29	5
5	$loc > 100 \wedge iv(g) > 8 \wedge totalOpnd > 117$	27	5
6	$iv(g) > 8 \wedge uniqOp > 11 \wedge totalOp > 80$	33	11
7	$iv(g) > 8 \wedge uniqOpnd > 34$	32	8
8	$totalOpnd > 117$	31	6
9	$loc > 100 \wedge iv(g) > 8$	31	9
10	$ev(g) > 4 \wedge iv(g) > 8$	32	7
11	$ev(g) > 4 \wedge uniqOpnd > 34$	39	12
12	$loc > 100 \wedge ev(g) > 4$	31	7
13	$iv(g) > 8 \wedge uniqOp > 11$	34	13
14	$ev(g) > 4 \wedge totalOp > 80 \wedge v(g) > 6 \wedge uniq_{O}p > 11$	46	19
15	$iv(g) > 8 \wedge totalOp > 80$	35	12
16	$ev(g) > 4 \wedge totalOp > 80 \wedge uniqOp > 11$	46	19
17	$ev(g) > 4 \wedge totalOp > 80 \wedge v(g) > 6$	47	19
18	$loc > 100 \wedge uniqOpnd > 34$	34	9
19	$ev(g) > 4 \wedge totalOp > 80$	47	19
20	$iv(g) > 8$	35	13

Table 12: CN2-SD induced Rules for the KC2 Dataset

#	*Rule*	*# Def*	*# Non Def*
1	$uniqOpnd > 34 \wedge ev(g) > 4$	39	12
2	$totalOp > 80 \wedge ev(g) > 4$	47	19
3	$uniqOp > 11$	86	117

Table 13: APRIORI-SD induced rules for the KC2 Dataset

#	*Rule*	*# Def*	*# Non Def*
1	$v(g) > 6 \wedge iv(G) > 8 \wedge uniqOp > 11 \wedge totalOp > 80$	35	13
2	$iv(G) > 8 \wedge uniqOp > 11 \wedge totalOp > 80$ $\wedge\, branchCount > 11$	34	12
3	$LoC > 100 \wedge uniqOp > 11 \wedge uniqOpnd > 34$ $\wedge\, totalOpnd > 117$	31	6
4	$LoC > 100 \wedge uniqOp > 11 \wedge totalOp > 80 \wedge totalOpnd > 117$	31	6
5	$LoC > 100 \wedge uniqOpnd > 34 \wedge totalOp > 80$ $\wedge\, totalOpnd > 117$	31	6
6	$uniqOp > 11 \wedge uniqOpnd > 34 \wedge totalOp > 80$ $\wedge\, totalOpnd > 117$	31	6
7	$ev(g) > 4 \wedge uniqOp > 11 \wedge uniqOpnd > 34$	39	12
8	$v(g) > 6 \wedge ev(g) > 4 uniqOpnd > 34$	39	12
9	$ev(g) > 4 \wedge uniqOpnd > 34 \wedge branchCount > 11$	37	12
10	$v(g) > 6 \wedge iv(G) > 8 \wedge uniqOp > 11 \wedge branchCount > 11$	33	13
11	$LoC > 100 \wedge uniqOp > 11 \wedge uniqOpnd > 34 \wedge totalOp > 80$	34	9
12	$ev(g) > 4 \wedge totalOp > 80$	47	19

algorithms in their respective metrics). Table 14 shows that the EDER-SD performed better than the SD algorithm in 39 out of 70 evaluation measures, similar results in 14 and worst results in 17 evaluation measures. It also preformed better in five out of the seven datasets for most evaluation measures. The best results were obtained in the PC1 dataset (the most imbalanced one) where EDER-SD outperformed the SD in all the evaluation measures. Similar results can be observed when comparing EDER-SD with the CN2-SD algorithm. It outperformed CN2-SD in 35 evaluation measures, obtained similar values in 10 occasions, and 25 values were worst. EDER-SD almost always performed better in the $specificity$, $accuracy$ and $lift$ measures and obtained similar or better results in $WRAcc$. The EDER-SD algorithm also performed better than the APRIORI-SD algorithm with the KC2 dataset and MC2 datasets (the only two that we were able to obtain results) for all evaluation measures with the exception of *precision* which was very similar. Taking into account all evaluation measures, EDER-SD performed better in 98 measures, similarly in 24 and worst in 43 out of the 160 evaluation measures.

In the case of software prediction, there is an ongoing discussion about the evaluation of defect prediction models [60, 39]. Menzies et al. suggest the use of probability of detection (*pd* or *recall*) and probability of false alarm (pf) arguing that a low precision acceptable in this domain and with

Table 14: Comparison of EDER-SD with other SD Algorithms

		$WRAcc$	Cov	Sup	Acc	Sig	$Prec$	TP_r	$Spec$	$Lift$	$fmsr$
EDER-SD	CM1	.025	.175	.042	.880	3.027	.357	.438	.947	3.705	.333
	KC1	.057	.287	.101	.859	2.767	.596	.657	.980	3.878	.458
	KC2	.090	.333	.155	.839	11.301	.769	.771	.978	3.824	.603
	KC3	.034	.268	.059	.935	6.219	.643	.643	1	10.928	.643
	MC2	.062	.389	.167	.759	3	.529	.529	1	3.176	.529
	MW1	.027	.119	.037	.941	3.07	.455	.455	1	12.273	.455
	PC1	.024	.259	.041	.943	7.125	.577	.577	1	14.231	.577
SD	CM1	.023	.313	.048	.855	2.067	.278	.5	.933	2.882	.294
		=	-	=	+	+	+	-	+	+	+
	KC1	.050	.213	.083	.859	21.843	.600	.537	.975	3.906	.450
		+	+	+	=	-	=	+	=	-	=
	KC2	.079	.322	.144	.839	9.854	.750	.714	.986	3.729	.549
		+	+	+	=	+	+	+	-	+	+
	KC3	.042	.523	.078	.856	2.84	.857	.857	.928	2.277	.857
		-	-	-	+	+	-	-	+	+	-
	MC2	.066	.815	.278	.741	2.171	.882	.882	.973	2.541	.882
		=	-	-	+	+	-	-	+	+	-
	MW1	.030	.111	.037	.933	5.542	.455	.455	.992	9.205	.455
		=	=	=	+	-	=	=	+	+	=
	PC1	.016	.097	.022	.897	4.341	.308	.308	.948	3.558	.308
		+	+	+	+	+	+	+	+	+	+
CN2-SD	CM1	.029	.325	.06	.88	1.719	.238	.625	.973	2.47	.286
		=	-	-	=	+	+	-	-	+	+
	KC1	.052	.418	.117	.859	21.843	.588	.759	.975	3.829	.408
		=	-	-	=	-	+	-	=	+	+
	KC2	.071	.391	.149	.839	9.257	.667	.743	.950	3.314	.522
		+	-	+	=	+	+	+	+	+	+
	KC3	.040	.346	.072	.856	2.84	.786	.786	.928	2.268	.786
		=	-	-	+	+	-	-	+	+	-
	MC2	.042	.278	.13	.685	.769	.412	.412	.838	1.588	.412
		+	+	+	+	+	+	+	+	+	+
	MW1	.032	.148	.044	.904	3.489	.545	.545	.952	4.909	.545
		=	-	-	+	-	-	-	+	+	-
	PC1	.029	.322	.051	.927	4.763	.731	.731	.997	3.558	.731
		=	-	-	+	+	-	-	=	+	-
APRIORI-SD	KC2	.043	.091	.061	.831	7.3	.681	.306	.963	3.384	.416
		+	+	+	+	+	+	+	+	+	+
	MC2	.027	.12	.065	.694	1.018	.559	.206	.919	1.775	.295
		+	+	+	+	+	-	+	+	+	+

the datasets used in this work. In any case, there is a trade-off between these two measures as stated by Menzies et al., increasing the *recall*, also increases the pf, and viceversa. Also, Khoshgoftaar and Seliya [28] chose to to minimise Type II errors (a *fault-prone* error misclassified as *non-fault-prone*) in accordance with a project manager of the system studied. When using EDER-SD in comparison with other tools, it is possible to generate rules according to the criteria of project managers or quality engineers.

We do not apply HIDER or other classification techniques due to the fact that quality measures are related to accuracy and this measure might not always be the most appropriate when data are imbalanced. Furthermore, in the case of hierarchical classification rules, those are harder to interpret and apply by domain experts than the rules obtained using EDER-SD. For example, Vandecruys et al. [54] show the rules obtained using their AntMiner+ tool for the PC1 and KC1 datasets. AntMiner+ extracts hierarchical classification rules, a chain of $if \ldots then \ldots else \ldots$ which are harder to interpret and apply than the rules obtained using EDER-SD. For example, Table 15 shows the AntMiner+ rules for the CM1 dataset. Although the first rule is easy to interpret, for the rest of the rules it becomes increasingly harder to extract useful knowledge. For modules not covered by the first rule we need to apply the second one and so on and so forth. Therefore, the number of conditions to consider is 3 for the first rule, 3+4 for the second rule, 3+4+5 for the third rule etc. Furthermore, the rules presented by Vandercruys to identify faulty modules have in general a low support and and the high level of accuracy for the Antminer+ rules seems to be consequence of the final *"else"* default branch, which covers the non–defective modules.

4.4. Threats to Validity

There are some threats to validity that need to be considered in this study as in all empirical studies.

Construct validity is the degree to which the variables used in the study accurately measure the concepts they to measure. Although there seems to be an agreement about the practical usefulness of static metrics, there are critics to their effectiveness as predictors of quality. Here, we can also highlight the point that is difficult to avoid an *unfair* comparison between SD algorithms as they induce a different number of rules, use different quality measures etc. For the comparison, the rules were obtained using the default parameters provided by the tool but other parameters could generate better sets of rules.

Table 15: AntMiner+ Rules for the PC1 Dataset [54]

```
if LOC_Blank $\geq$ 16 and LOC_Code_And_Comment $\geq$ 2 and
Normalized_Cyclomatic_Complexity $\geq$ 0.17
then class = Erroneous module
else if LOC_Code_And_Comment $\geq$ 1 and
LOC_Comments $\geq$ 5 and
Normalized_Cyclomatic_Complexity $\geq$ 0.23 and
Num_Unique_Operands $\geq$ 38
then class = Erroneous module
else if Halstead_Content $\geq$ 50.37 and
Halstead_Error_Est $\geq$ 0.6 and
LOC_Blank $\geq$ 16 and LOC_Comments $\geq$ 14 and
LOC_Executable $\geq$ 53
then class = Erroneous module
else if Halstead_Content $\geq$ 50.37 and
LOC_Blank $\geq$ 16 and LOC_Code_And_Comment $\geq$ 2 and
LOC_Comments $\geq$ 14 and
Normalized_Cyclomatic_Complexity $\geq$ 0.08
then class = Erroneous module
else if Halstead_Content $\geq$ 50.37 and
LOC_Code_And_Comment $\geq$ 2 and
LOC_Comments $\geq$ 5 and
Normalized_Cyclomatic_Complexity $\geq$ 0.17
then class = Erroneous module
else class = Correct module
```

Internal validity is the degree to which conclusions can be drawn. This work consisted in a small number of datasets and all came from the same domain. There is some consistency among the attributes used within each dataset but they vary among datasets and so do the thresholds. In this work, we have generated a set of rules for each dataset but from a practical point of view, it could be interesting to generalise the rules across the datasets in order to facilitate their application by project managers or quality engineers. One approach to do this is to join all datasets and generate generic rules even if some degree of performance is lost in individual datasets. It can be observed that the threshold values across (the different datasets are close to each other (e.g., LoC varies between 60 and 80; $iv(g)$ between 3 and 5, etc.).

External validity is the degree to which the results of the research can be generalised to the population under study and other research settings. According to Menzies et al. [40], the NASA repository can be generalised to the industry in general. However, it is probably better to calibrate the rules to different domains or organisations. Finally, as with other empirical studies, this approach needs to be replicated with further datasets and SD algorithms.

5. Related Work

Initially, some statistical approaches were proposed to deal with defect prediction. For example, Munson and Khoshgoftaar [42] explore discriminant analysis techniques on two commercial datasets composed with many of the Halstead and McCabe metrics used in this work. Basili et al. [4] analysed the applicability of Chidamber and Kemerer's Object Oriented set of metrics [10] with logistic regression to predict fault-prone code classes. Khoshgoftaar and Allen [27] also analysed logistic regression extended with prior probabilities and of misclassification costs.

More recently, a number of researchers have focused on machine learning approaches. Khoshgoftaar et al. [26] described the use of neural networks for quality prediction. The authors used a dataset from a telecommunications system and compare the neural networks results with a non-parametric model. Also, Khoshgoftaar et al. [25] applied regression trees as classification model to the same problem.

However, there are still large discrepancies regarding the assessment of the goodness of the different techniques and the reasons for such discrepancies [44, 60, 39]. For example, Lessmann et al. [33] compare 22 classifiers grouped into statistical, nearest neighbour methods, neural networks, support vector machine, decision trees and ensemble methods over ten datasets from the NASA repository. The authors discuss several performance metrics such as TP_r and FP_r but advocate the use of AUC as the best indicator to compare the different classifiers. Arisholm et al. [3] compare different data mining techniques (classification tree algorithm (C4.5), a coverage rule algorithm (PART), logistic regression, back–propagation neural work and support vector machines) over 13 releases of a Telecom middleware software developed in Java using three types metrics: (i) object oriented metrics, (ii) delta measures, amount of change between successive releases, and (iii) process measures from a configuration management system. The authors concluded that although there are no significant differences regarding the techniques used, large differences can be observed depending on the criteria used to compare them. The authors also propose a cost–effectiveness measure based on the AUC and number of statements so that larger modules are more expensive to test. The same approach of considering module size in conjunction with the AUC as evaluation measure has been explored by Mende and Koschke [38] using NASA datasets and three versions of Eclipse[7] and random forests [6] as classification technique. Koru and Liu [31] use the C4.5 [49] implementation of Weka for defect prediction on the NASA datasets to analyse the relationships between defects and module size. Khoshgoftaar and Seliya [28] recognise the problem of imbalanced data and use Case-based Reasoning to deal with this problem, considering Type I error when a non-faulty module is classified as faulty and Type II error occurs when a faulty module is classified as non-faulty. This approach is also considered by Ostrand and Weyuker [45].

Also in this respect but in the domain of cost estimation, Shepperd and Kadoda [52] analyse the influence of different data characteristics (dataset size; number, type and independence of features; and type of distribution, in.) using simulated data over a number of different types of classifiers (regression, rules induction, nearest neighbour and neural networks). The authors conclude there is no best classifier as the characteristics of the data highly affect the outcomes.

There are a number of other works using subsets of the NASA repository. Peng et al. [47] propose a performance metric to evaluate the merit of

[7] `http://www.eclipse.org/`

classification algorithms using a broad selection of classification algorithms and performance measures. The experimental results, using 13 classification algorithms with 11 measures over 11 software defect datasets, indicate that the classifier which obtains the best result for a given dataset according to a given measure may perform poorly on a different measure. The results of the experiment indicate that support vector machines, k-nearest neighbor algorithm and C4.5 algorithm ranked the top three classifiers. Menzies et al. [40] applied J48 (the Weka implementation of the C4.5) and Naïve Bayes to several datasets of the PROMISE repository for defect prediction. The authors concluded that such technique can be used as good defect estimators and suggest bound exploration as part of future work. Elish and Elish [12] applied the Support Vector Machine (SVM) technique to a subset of the NASA repository. The authors concluded that SVM is capable of improving or at least obtaining similar result than other techniques such as logistic regression, neural networks, Bayesian networks or decision trees. Recently, Peng et al. [48] have also analysed ten NASA datasets using four Multicriteria Decision Making methods to rank classification algorithms, highlighting that the boosting of CART and the boosting of C4.5 decision tree are ranked as the most appropriate algorithms to deal with defect prediction.

Finally, some authors criticize the use of only static metrics with statistical techniques as an approach to defect prediction. For example, Fenton and Neil [14] advocate the use of Bayesian network approaches as a probabilistic technique to estimate defects among other parameters.

6. Conclusions and Future Work

In this work, we applied Subgroup Discovery (SD), a data mining approach used to find groups of statistically different data given a property of interest, to the problem of software defect prediction. SD is a sensible approach to deal with imbalanced, inconsistent or redundant data.

To do so, we developed a genetic algorithm, EDER-SD (Evolutionary Decision Rules for Subgroup Discovery), used to induces rules describing only fault-prone modules. EDER-SD has the advantage of working with continuous variables as the conditions of the rules are defined using intervals. Furthermore, the fitness function of the genetic algorithm can be adapted to optimise the most suitable quality measure of the domain. EDER-SD was applied to seven publicly available datasets from the PROMISE repository. The results shows that the induced rules are capable of characterising subgroups of fault-prone modules. We also found that many of the thresholds found by the metrics are close to those defined in the literature or the McCabe IQ tool used to obtain the datasets. Furthermore, the simplicity of the induced rules and their readability facilitates the application of this approach helping project managers or quality engineers with testing and software quality assurance activities. We also compared EDER-SD with three other well-known SD algorithms. The results of the comparison showed the advantages of been able to adapt the fitness function of the evolutionary algorithm as (in general) EDER-SD did perform better than the other algorithms.

In relation to future work, we will further explore this approach with other datasets and different metrics, for example, datasets with object oriented metrics. There is also room for improvement of the genetic algorithm, for example exploring further quality measures or multiobjective approaches.

Acknowledgements

The authors are grateful to the anonymous reviewers and Prof Rachel Harrison for their useful comments. This work has been supported by the projects TIN2007-68084-C02-00 and TIN2010-21715-C02-01 (Spanish Ministry of Education and Science). D. Rodríguez carried out part of this work as a visiting research fellow at Oxford Brookes University, UK.

References

[1] R. Agrawal, R. Srikant, Fast algorithms for mining association rules in large databases, in: VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994, pp. 487–499.

[2] J. Aguilar-Ruiz, I. Ramos, J. Riquelme, M. Toro, An evolutionary approach to estimating software development projects, Information and Software Technology 43 (2001) 875–882.

[3] E. Arisholm, L.C. Briand, E.B. Johannessen, A systematic and comprehensive investigation of methods to build and evaluate fault prediction models, Journal of Systems and Software 83 (2010) 2–17.

[4] V.R. Basili, L.C. Briand, W.L. Melo, A validation of object-oriented design metrics as quality indicators, IEEE Transactions on Software Engineering 22 (1996) 751–761.

[5] G. Boetticher, T. Menzies, T. Ostrand, Promise repository of empirical software engineering data, 2007.

[6] L. Breiman, Random forests, Machine Learning 45 (2001) 5–32.

[7] J. Cano, F. Herrera, M. Lozano, S. García, Making CN2-SD subgroup discovery algorithm scalable to large size data sets using instance selection, Expert Systems with Applications (2008).

[8] C. Catal, B. Diri, Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem, Information Sciences 179 (2009) 1040–1058.

[9] Z. Chen, T. Menzies, D. Port, B. Boehm, Finding the right data for software cost modeling, IEEE Software 22 (2005) 38–46.

[10] S.R. Chidamber, C.F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering 20 (1994) 476–493.

[11] P. Clark, T. Niblett, The CN2 induction algorithm, Machine Learning 3 (1989) 261–283.

[12] K. Elish, M. Elish, Predicting defect-prone software modules using support vector machines, Journal of Systems and Software 81 (2008) 649–660.

[13] L. Eshelman, J. Schaffer, Real-coded genetic algorithms and interval-schemata, in: L. Whitley (Ed.), Foundations of Genetic Algorithms, volume 2.

[14] N.E. Fenton, M. Neil, A critique of software defect prediction models, IEEE Transactions on Software Engineering 25 (1999) 675–689.

[15] N.E. Fenton, S.L. Pfleeger, Software Metrics: A Rigorous and Practical Approach, PWS Publishing Co., Boston, MA, USA, 2nd edt. edition, 1998.

[16] A. Fernández, M.J. del Jesus, F. Herrera, On the 2-tuples based genetic tuning performance for fuzzy rule based classification systems in imbalanced data-sets, Information Sciences 180 (2010) 1268–1291.

[17] J.H. Friedman, From statistics to neural networks, From Statistics to Neural Networks, Springer Verlag, 1994, pp. 1–61.

[18] D. Gamberger, N. Lavrač, Expert-guided subgroup discovery: Methodology and application, Journal of Artificial Intelligence Research 17 (2002) 501–527.

[19] L. Geng, H.J. Hamilton, Interestingness measures for data mining: A survey, ACM Computing Surveys 38 (2006) paper 9.

[20] M. Halstead, Elements of Software Science, Elsevier Computer Science Library. Operating And Programming Systems Series; 2, Elsevier, New York; Oxford, 1977.

[21] F. Herrera, C.J. Carmona del Jesus, P. González, M.J. del Jesus, An overview on subgroup discovery: Foundations and applications, Knowledge and Information Systems Published online first: http://www.springerlink.com/content/y3g719412258w058/ (2010).

[22] V. Jovanoski, N. Lavrač, Classification rule learning with APRIORI-C, in: EPIA '01: Proceedings of the 10th Portuguese Conference on Artificial Intelligence on Progress in Artificial Intelligence, Knowledge Extraction, Multi-agent Systems, Logic Programming and Constraint Solving, Springer-Verlag, London, UK, 2001, pp. 44–51.

[23] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, K. Matsumoto, The effects of over and under sampling on fault–prone module detection, in: Empirical Software Engineering and Measurement (ESEM), pp. 196–204.

[24] B. Kavšek, N. Lavrač, APRIORI-SD: Adapting association rule learning to subgroup discovery, Applied Artificial Intelligence 20 (2006) 543–583.

[25] T.M. Khoshgoftaar, E. Allen, J. Deng, Using regression trees to classify fault-prone software modules, IEEE Transactions on Reliability 51 (2002) 455–462.

[26] T.M. Khoshgoftaar, E. Allen, J. Hudepohl, S. Aud, Application of neural networks to software quality modeling of a very large telecommunications system, IEEE Transactions on Neural Networks 8 (1997) 902–909.

[27] T.M. Khoshgoftaar, E.B. Allen, Logistic regression modeling of software quality, International Journal of Reliability, Quality and Safety Engineering 6 (1999) 303–317.

[28] T.M. Khoshgoftaar, N. Seliya, Analogy-based practical classification rules for software quality estimation, Empirical Software Engineering 8 (2003) 325–350.

[29] C. Kirsopp, M. Shepperd, Case and feature subset selection in case-based software project effort prediction, in: Proceedings of 22nd International Conference on Knowledge-Based Systems and Applied Artificial Intelligence (SGAI'02).

[30] W. Klösgen, EXPLORA: a multipattern and multistrategy discovery assistant (1996) 249–271.

[31] A.G. Koru, H. Liu, Building effective defect-prediction models in practice, IEEE Software 22 (2005) 23–29.

[32] N. Lavrač, B. Kavšek, P. Flach, L. Todorovski, Subgroup discovery with CN2-SD, The Journal of Machine Learning Research 5 (2004) 153–188.

[33] S. Lessmann, B. Baesens, C. Mues, S. Pietsch, Benchmarking classification models for software defect prediction: A proposed framework and novel findings, IEEE Transactions on Software Engineering 34 (2008) 485–496.

[34] Y. Li, M. Xie, T. Goh, A study of mutual information based feature selection for case based reasoning in software cost estimation, Expert Systems with Applications 36 (2009) 5921–5931.

[35] Z. Li, M. Reformat, A practical method for the software fault–prediction, in: IEEE International Conference Information Reuse and Integration (IRI), pp. 659–666.

[36] H. Liu, L. Yu, Toward integrating feature selection algorithms for classification and clustering, IEEE Trans. on Knowledge and Data Eng. 17 (2005) 1–12.

[37] T. McCabe, A complexity measure, IEEE Transactions on Software Engineering 2 (1976) 308–320.

[38] T. Mende, R. Koschke, Effort-aware defect prediction models, in: 14th European Conference on Software Maintenance and Reengineering (CSMR'10).

[39] T. Menzies, A. Dekhtyar, J. Distefano, J. Greenwald, Problems with precision: A response to comments on data mining static code attributes to learn defect predictors, IEEE Transactions on Software Engineering 33 (2007) 637–640.

[40] T. Menzies, J. Greenwald, A. Frank, Data mining static code attributes to learn defect predictors, IEEE Transactions on Software Engineering 33 (2007) 2–13.

[41] T. Mitchell, Machine Learning, McGraw Hill, 1997.

[42] J.C. Munson, T.M. Khoshgoftaar, The detection of fault-prone programs, IEEE Transactions on Software Engineering 18 (1992) 423–433.

[43] J.D. Musa, A. Iannino, K. Okumoto, Software reliability: measurement, prediction, application, McGraw-Hill, Inc., New York, NY, USA, 1989.

[44] I. Myrtveit, E. Stensrud, M. Shepperd, Reliability and validity in comparative studies of software prediction models, IEEE Transactions on Software Engineering 31 (2005) 380–391.

[45] T.J. Ostrand, E.J. Weyuker, How to measure success of fault prediction models, in: SOQUA'07: Fourth international workshop on Software quality assurance, ACM, New York, NY, USA, 2007, pp. 25–30.

[46] Y. Peng, G. Kou, Y. Shi, Z. Chen, A descriptive framework for the field of data mining and knowledge discovery, International Journal of Information Technology & Decision Making (IJITDM) 07 (2008) 639–682.

[47] Y. Peng, G. Kou, G. Wang, H. Wang, F. Ko, Empirical evaluation of classifiers for software risk management, International Journal of Information Technology & Decision Making (IJITDM) 08 (2009) 749–767.

[48] Y. Peng, G. Wang, H. Wang, User preferences based software defect detection algorithms selection using MCDM, Information Sciences In Press. (2010) –.

[49] J. Quinlan, C4.5: Programs for machine learning, Morgan Kaufmann, San Mateo, California, 1993.

[50] D. Rodríguez, R. Ruiz, J. Cuadrado, J. Aguilar-Ruiz, Detecting fault modules applying feature selection to classifiers, in: IEEE International Conference on Information Reuse and Integration (IRI 2007), pp. 667–672.

[51] C. Seiffert, T.M. Khoshgoftaar, J.V. Hulse, A. Folleco, An empirical study of the classification performance of learners on imbalanced and noisy software quality data, Information Sciences In Press. (2011) –.

[52] M. Shepperd, G. Kadoda, Comparing software prediction techniques using simulation, IEEE Transactions on Software Engineering 27 (2001) 1014–1022.

[53] B. Turhan, A. Bener, Analysis of naïve bayes' assumptions on software fault data: An empirical study, Data & Knowledge Engineering 68 (2009) 278–290.

[54] O. Vandecruys, D. Martens, B. Baesens, C. Mues, M. De Backer, R. Haesen, Mining software repositories for comprehensible software fault prediction models, Journal of Systems and Software 81 (2008) 823–839.

[55] G. Venturini, SIA: A supervised inductive algorithm with genetic search for learning attributes based concepts, in: ECML'93: Proceedings of the European Conference on Machine Learning, Springer-Verlag, London, UK, 1993, pp. 280–296.

[56] F. Železný, N. Lavrač, Propositionalization-based relational subgroup discovery with RSD, Machine Learning 62 (2006).

[57] I. Witten, E. Frank, Data Mining: Practical machine learning tools and techniques, Morgan Kaufmann, San Francisco, 2005.

[58] S. Wrobel, An algorithm for multi-relational discovery of subgroups, in: Proceedings of the 1st European Symposium on Principles of Data Mining, pp. 78–87.

[59] S. Wrobel, Relational data mining, Relational Data Mining, 2001, pp. 74–101.

[60] H. Zhang, X. Zhang, Comments on "data mining static code attributes to learn defect predictors", IEEE Transactions on Software Engineering 33 (2007) 635–637.