A Study of Subgroup Discovery Approaches for Defect Prediction

Daniel Rodriguez[a,*], Roberto Ruiz[b], Jose C. Riquelme[c], Rachel Harrison[d]

[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*
[d]*School of Technology, Oxford Brookes University Wheatley Campus, Oxford OX33 1HX, UK*

Abstract

Context: Although many papers have been published on software defect prediction techniques, machine learning approaches have yet to be fully explored.
Objective: In this paper we suggest using a descriptive approach for defect prediction rather than the precise classification techniques that are usually adopted. This allows us to characterise defective modules with simple rules that can easily be applied by practitioners and deliver a practical (or engineering) approach rather than a highly accurate result.
Method: We describe two well-known subgroup discovery algorithms, the SD algorithm and the CN2-SD algorithm to obtain rules that identify defect prone modules. The empirical work is performed with publicly available datasets from the Promise repository and object-oriented metrics from an Eclipse repository related to defect prediction. Subgroup discovery algorithms mitigate against characteristics of datasets that hinder the applicability of classification algorithms and so remove the need for preprocessing techniques.

*Corresponding author
Email addresses: daniel.rodriguezg@uah.es (Daniel Rodriguez), robertoruiz@upo.es (Roberto Ruiz), riquelme@lsi.us.es (Jose C. Riquelme), rachel.harrison@brookes.ac.uk (Rachel Harrison)

Results: The results show that the generated rules can be used to guide testing effort in order to improve the quality of software development projects. Such rules can indicate metrics, their threshold values and relationships between metrics of defective modules.
Conclusions: The induced rules are simple to use and easy to understand as they provide a description rather than a complete classification of the whole dataset. Thus this paper represents an *engineering approach* to defect prediction, i.e., an approach which is useful in practice, easily understandable and can be applied by practitioners.

Keywords:
Subgroup Discovery, Rules, Defect Prediction, Imbalanced datasets.

---

1. Introduction

In the recent past, the application of data mining techniques in software engineering has received a lot of attention. Problems such as planning and decision making, defect prediction, effort estimation, testing and test case generation, knowledge extraction, etc. can be reformulated using a set of techniques under the umbrella of data mining [15, 73, 27]. The extracted patterns of knowledge can assist software engineers in predicting, planning, and understanding various aspects of a project so that they can more efficiently support future development and project management activities.

Data mining provides techniques to analyze and extract novel, interesting patterns from data. Formally, it has been defined as the process of inducing previously unknown and potentially useful information from data collections [23]: "*The two high-level primary goals of data mining in practice tend to be prediction and description*". The former is related to the prediction of unknown or future values (e.g. classification tree models, regression models, etc.), the latter, involves finding interesting patterns that can be easily understood by humans (e.g. association and clustering algorithms). It is worth noting that Fayyad et al. also state that: "*the boundaries between prediction and description are not sharp (some of the predictive models can be descriptive, to the degree that they are understandable, and vice-versa)*". For example, clustering algorithms can be used as classifiers or supervised feature selection methods can be considered descriptive. There are also techniques that hybridise prediction and description as in the case of *supervised descriptive techniques* [43]. The aim of these techniques is to understand

the underlying phenomena rather than to classify new instances; i.e., to find interesting information about a specific value. The information should be useful to the domain experts and easily interpretable by them.

In this work, we tackle the defect prediction problem through a descriptive induction process using Subgroup Discovery (SD) techniques. These kinds of algorithms are designed to find subgroups of data that are statistically different given a property of interest [39, 71, 72, 32]. SD algorithms can be both predictive, finding rules given historical data and a property of interest; and descriptive, discovering interesting patterns in data. For the same purpose, there are other types of supervised descriptive techniques, Contrast Set Mining (CSM) [6] and Emerging Pattern Mining (EPM) [19]. CSM finds contrast sets which are defined as conjunctions of attribute-value pairs that differ significantly in their distributions across groups (class variable). These contrast sets may have a very low support but they must clearly differentiate the different groups. EPM captures emerging patterns (EP) in time-stamped databases or useful contrasts in classification datasets (with a class attribute). EPs are defined as itemsets (using association rule terminology) whose support increases significantly from one dataset, D_1, to another dataset, D_2. EPM searches for characteristics that differentiate two itemsets, D_1 and D_2, based on the $growthRate$[1] (ratio between both supports) as quality measure. These techniques have similarities: they all use rules as a representation techniques and have been proved to be equivalent [43]. However, the SD approaches have better tool support (including the Orange toolkit) and the quality measures used as objective function focus on finding statistically different subgroups (this is explained in § 3).

The main contributions of this paper are as follows. Firstly, to propose a descriptive approach based on Subgroup Discovery for defect prediction which allows us to characterise defective modules with simple rules that can easily be applied by practitioners. Such rules can describe thresholds and relationships between metrics. In this paper, we show how SD algorithms induce rules that can indicate defective software modules with a fairly high probability. To do this, we rely on the fact that SD algorithms mitigate against some of the characteristics of datasets that hinder the applicability of many classical classification algorithms such as (i) imbalanced datasets in which the number of non-defective modules is much larger than the number

[1] $growthRate(itemset) = \frac{support_{D_2}(itemset)}{support_{D_1}(itemset)}$

of defective modules, (ii) duplicated instances and contradictory cases and (iii) redundant and irrelevant attributes. In the literature these problems have mainly been tackled with preprocessing techniques such as sampling and feature selection. SD algorithms can be an alternative to classical classification algorithms without the need of applying preprocessing techniques. Modifying the original data or using preprocessing techniques does not always guarantee better results and can make it more difficult to extract knowledge from the data. SD algorithms, on the other hand, can be applied to the original data without the need for sampling or feature selection techniques and the representation of the rules makes them easy to apply.

In summary, we search for simple models represented as rules capable of detecting defective modules rather than highly accurate models. Thus our research question is: *can subgroup discovery be used to detect the most defective modules in a system?*

We describe and compare two well-known SD algorithms, the Subgroup Discovery (SD) algorithm [25] and the CN2-SD algorithm [44], by applying them to several datasets from the publicly available Promise repository [52], as well as the Bug Prediction Dataset (BPD) created by D'Ambros et al. [16, 17].

The organization of the paper is as follows. Section 2 covers the related work in defect prediction followed by background related to subgroup discovery concepts in Section 3. Next, Section 4 describes the experimental work, including datasets, rule induction, study of the generalisation of the rule induced and discussion of the results. Section 5 covers the threats to the validity. Finally, Section 6 concludes the paper and outlines future research work.

2. Related Work

Defect prediction has been an important research topic for more than a decade with an increasing number of papers including two recent and comprehensive systematic literature reviews [12, 29]. Many studies in defect prediction have been reported using techniques which originated from the field of statistics and machine learning. Such techniques include regression [8], logistic regression [18, 75], Support Vector Machines [20], etc. Others have their origin in machine learning techniques such as classification trees [34]), neural networks [35], probabilistic techniques (such as Naïve Bayes [53] and

Bayesian networks [24]), Case Based Reasoning ([36]), ensembles of different techniques and meta-heuristic techniques such as ant colony optimisation [33, 5, 67], etc.

Work has also been done on using rules as a representation model or decision trees such as C4.5 [59] which can be easily transformed to rules. For example, Koru and Liu [41]) used C4.5 for defect prediction with the NASA datasets to analyse the relationships between defects and module size. Also descriptive rules such as association rules [1] have been applied by Song et al. [66] to predict the defect associations and defect correction effort. In general, rules are easier to understand and apply than many other classification techniques such as neural networks or ensembles of multiple classifiers which behave as black boxes and are difficult to generalise across different datasets even when the same attributes are used. Hierarchical rules, such as chained *if ... then ... else* rules are harder to interpret and use by domain experts than independent rules such as the ones obtained by SD approaches. For example, Vandecruys et al. [67] reported the use of ant colonies as optimization technique for generating rules. A drawback of their approach is that they cannot handle imbalanced datasets appropriately and hierarchical rules can become hard to understand and apply. Azar and Vybihal [5] have also used metaheuristic optimization to induce rules capable of predicting defective modules from a number of static metrics that measure size, cohesion, coupling and inheritance. In this case, the authors recognised and deal with the imbalance by reporting Younden's J_{index} per class. In another work, Azar et al. [4] also combine rules from different algorithms using metaheuristics. Their approach could be used to combine and select rules induced from different SD algorithms (as shown in this work) or as a postprocessing step if a large number of rules are generated.

Several papers have compared multiple techniques with single datasets (e.g.[37]) or multiple datasets with multiple evaluation measures. Peng et al. [57] evaluated 13 classification algorithms with 11 measures over 11 software defect datasets. Although Support Vector Machines, nearest neighbour and the C4.5 algorithm were ranked as the top three classifiers, the authors indicated that a classifier which obtains the best result for a given dataset according to a given measure may perform poorly with a different measure. Also in another work, Peng et al. [58] used ten NASA datasets to rank classification algorithms, showing that a CART boosting algorithm and the C4.5 decision-tree algorithm with boosting are ranked as the optimum algorithms for defect prediction. Another extensive study, Lessman et

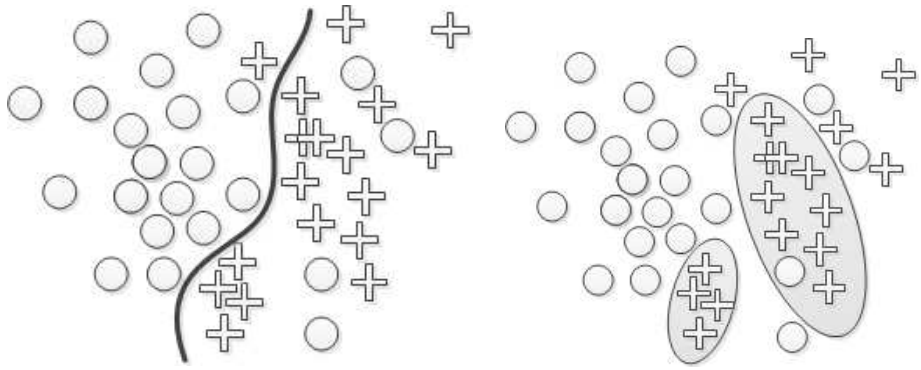
al. [45] compared 22 classifiers grouped into statistical, nearest neighbour, neural networks, support vector machine, decision trees and ensemble methods over ten datasets from the NASA repository. The authors discuss several performance metrics such as TP_{rate} and FP_{rate} but advocate the use of Area Under the ROC (AUC) [21] as the best indicator for comparing the different classifiers.

However, there are discrepancies among the outcomes of these works where (i) no classifier is consistently better than the others; (ii) there is no optimum metric to evaluate and compare classifiers as highlighted in [49, 56, 74, 53]; and (iii) there are quality issues regarding the data such as imbalanced datasets, class overlaps, outliers, transformation issues, etc. that affect different classifiers differently. Menzies et al. [55] argue that we may have reached the limit of what we can do with standard classifiers and new paths need to be explored, for example considering cost. Arisholm et al. [3] compared a classification tree algorithm (C4.5), a coverage rule algorithm (PART), logistic regression, neural networks and support vector machines over 13 releases of a Telecom middleware system developed in Java using three types of metrics: (i) object-oriented metrics, (ii) churn (delta) metrics between successive releases, and (iii) process management metrics 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 new cost-effectiveness metric based on the area-under-the-curve (AUC) and the number of statements so that larger modules are more expensive to test. Arisholm and Briand also considered cost [2]. Mende and Koschke [50] also explored module size in conjunction with the AUC as evaluation metrics in defect prediction using the NASA datasets as well as datasets from three versions of Eclipse[2] using random forests [9] as the classification technique. Here we want to to explore another alternative, Subgroup Discovery, which is explained in detail in the next Section.

3. Subgroup Discovery

Subgroup Discovery (SD) [39, 71, 72, 32] concerns the discovery of statistically distinct subgroups with respect to some property of interest. Subgroups are generally represented through rules, e.g. `if` $LoC > 100$ `and`

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



(a) Classification (b) Subgroup Discovery

Figure 1: Classification (a) vs. Subgroup Discovery (b)

complexity $>$ 4 `then` $defective$. More formally, rules are represented as $Cond \rightarrow Class$, where $Class$ (the consequent) is a specific value of an attribute of interest (e.g. $defective = true$), and $Cond$ (the antecedent) 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, e.g. $val_1 \leq att \leq val_2$. We can generate rules for all values of the attribute class; however, in this paper, we want to focus only on finding rules that identify defect prone modules. Rules have various advantages: they are able to explain the learned models (they provide knowledge of the learned domain), they are easily applicable and they are understandable by domain experts.

As shown in Figure 1(a), classification techniques are concerned with prediction (dividing the data as accurately as possible) while SD algorithms focus on finding subgroups of data related to some property of interest (see Figure 1(b)).

Rules induced by SD algorithms are different from the ones induced by classification algorithms. Figure 2(a) shows rules induced as a classification tree using Weka's implementation of C4.5 and Figure 2(b) shows the rules generated by the SD algorithm CN2-SD. Graphically, we can also observe that the three rules induced by the C4.5 classifier – which can be derived from the classification tree – cover disjoint areas of the search space. The rule R1 divides the search space into two partitions depending on whether the variable `total_loc` is less than or equal to 155. Next, R2.1 and R2.2 spaces are created depending on `design_complexity` being less or equal than 54.

R1	`total_loc` $\leq$ 155: false
	`total_loc > 155`
R2.1		`design_complexity` $\leq$ 54: true
R2.2		`design_complexity` $>$ 54: false

(a) Classification Example

S1	`unique_operators > 15`
S2	`unique_operators > 15 & branch_count > 32`
S3	`unique_operators > 15 & design_complexity > 3`
S4	`unique_operators > 15 & branch_count > 32 & design_complexity > 3`
S5	`cyclomatic_complexity > 17 & design_complexity > 3`

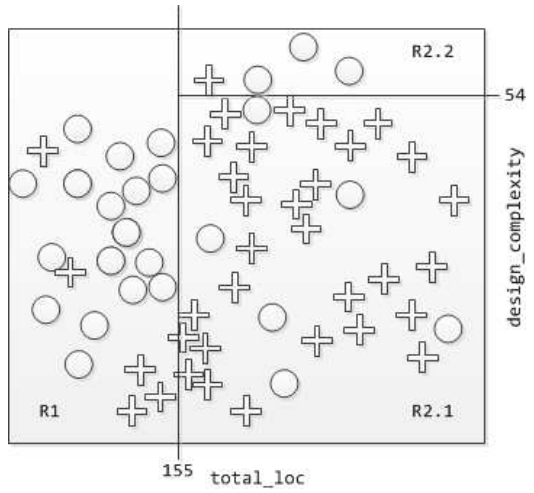
(b) Subgroup Discovery Example

Figure 2: Example of Rules induced with Classification (a) vs. Subgroup Discovery (b)

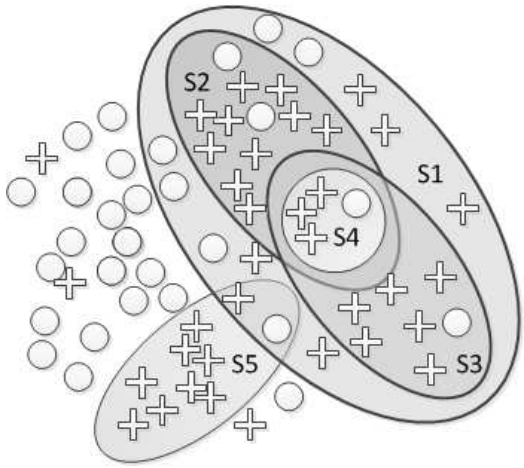
The graphical interpretation of the three disjoint search spaces is shown in Figure 3(a).

On the other hand, SD algorithms can generate redundant rules (see Figure 3). For example, the rule or subgroup S4, `unique_operators > 15 & branch_count > 32 & design_complexity > 3` is more restrictive (specific) than the rules S2 and S3, which in turn are again more restrictive than the first rule S1, `unique_operator > 15`. Figure 3 shows graphically how the SD rules find subgroups of data for the property of interest (in this case, defective modules).

As stated previously, the idea of SD is to label interpretable groups of data in an intuitive manner. Therefore, there is a balance between the specificity and generality of the induced rules. The more conditions rules have (the more specific they are), the fewer false positives the rules have. On the other hand, the fewer conditions the rules have, the more generic the rules are, possibly generating a larger number false positives. For example, Figure 4 shows three different subgroups, the subgroup *S1* covers a small number of instances with only one error. However, subgroup *S2* is more generic (there are fewer conditions in the rules) but these more generic rules increase the number of false positives, and the same for *S3*. In this sense, SD can be compared to a form of cost-sensitive classification which aims to find a high proportion of a particular class. For a comprehensive survey of subgroup discovery, we refer the reader to the work of Herrera et al. [32]. Table 1 summarises the main differences between rules induced by classification and SD techniques.



(a) Classification Example



(b) Subgroup Discovery Example

Figure 3: Graphical Representation: Classification (a) vs. Subgroup Discovery (b)

Table 1: Classification vs. Subgroup Discovery

	Classification	***Subgroup Discovery***
Induction Type	Predictive	Descriptive
Output	Set of classification rules	Individual rules to describe subgroups
Purpose	To learn a model for classification or prediction	To find interesting and interpretable patterns with respect to a specific attribute

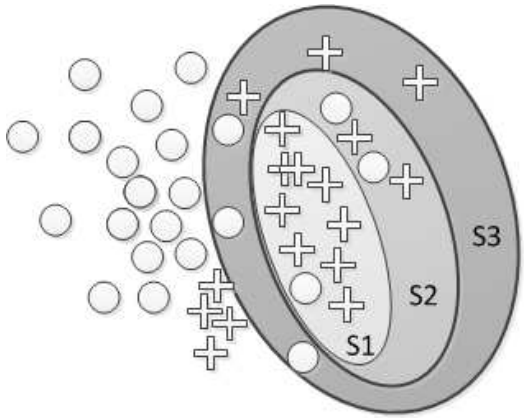


Figure 4: Specificity vs. Generality of the Induced Rules

In this paper, we use two well-known SD algorithms widely cited in the literature and which are implemented in an extension[3] of the Orange data mining tool[4]. We then describe both algorithms including their objective functions.

The Subgroup Discovery algorithm SD [25] is a covering rule induction algorithm [69] that uses beam search (where a set of alternatives are kept while finding optimal solutions) to find rules that maximise the following equation:

$$q_g = \frac{n(Cond \cdot Class)}{n(Cond \cdot \neg Class) + g} = \frac{TP}{FP + g} \tag{1}$$

where $n(Cond \cdot Class)$ represents the number of instances covered by a rule in which both the antecedent ($Cond$) and consequent ($Class$) are true (this is the number of true positives, TP). and $n(Cond \cdot \neg Class)$ is the number of instances in which the antecedent is true but not the consequent (FP). The generalisation parameter g allow us to control the *specificity* of a rule, i.e., the balance between the complexity of a rule and its accuracy, typically between 0.5 and 100. For values smaller than or equal to one, the induced rules will have very high specificity. For values larger than one, the larger the value, the larger the percentage of errors.

[3] `http://kt.ijs.si/petra_kralj/SubgroupDiscovery/`
[4] `http://www.ailab.si/orange/`

Algorithm 1 SD Algorithm [25]

Require: $\mathcal{D}$ dataset; g generalisation parameter; $minSupp$; $beamWidth$
Require: $l \in \mathcal{L}$, where $\mathcal{L}$ is the set of all attribute values
Ensure: set of rules

```
for all rules in Beam and newBeam do
    {} → Cond                                          ▷ Initialisation
    q_g(i) ← 0                                         ▷ i=1 to beamWidth
end for
while there are improvements do
    for all rules in Beam and newBeam do
        for all l ∈ L do
            Cond(i) ← Cond(d) ∧ l                      ▷ add new condition to the beam
            q_g = TP/(FP+g)                            ▷ calculate quality for the new rule
            if TP/|E| ≥ minSupp and q_g ≥ (q_g(i) ∈ newBeam) and relevant then
                replace worst rule with new rule
                sort rules in newBeam according to quality
            end if
        end for
    end for
    Beam ← newBeam
end while
```

The CN2-SD [44] algorithm induces subgroups in the form of rules using as a quality metric the relation between true positives and false positives. It is an adaptation of the CN2 classification rule algorithm [14], which consists of a search procedure using beam search within a control procedure that iteratively performs the search (see Algorithm 2). The CN2-SD algorithm uses the rule *Weighted Relative Accuracy* ($WRAcc$) as a covering measure of the quality of the induced rules:

$$WRAcc(R_i) = \frac{n(Cond)}{N} \cdot \left(\frac{n(Cond \cdot Class)}{n(Cond)} - \frac{n(Class)}{N} \right) \qquad (2)$$

where N is the total number of instances, $n(Cond)$ and $n(Class)$ are respectively the number of instances in a dataset in which the antecedent and consequent hold. This measure, $WRAcc$, represents a trade-off between the coverage of a rule, i.e., its generality or probability of the condition ($p(Cond)$) and its accuracy gain ($p(Class \cdot Cond) - p(Class)$ expressed as probabilities), which is proportional to $TP_{rate} - FP_{rate}$ [43].

Discretisation in both algorithms is required for continuous attributes and so an entropy based discretisation method (Minimum Description Length) [22]

Algorithm 2 CN2-SD Algorithm [44]

Require: $\mathcal{D}$ dataset with discrete attributes.
Ensure: Set of rules
1: $RuleSet \leftarrow \emptyset$
2: **function** UNSORTEDCN2($\mathcal{D}$, $classValue$)
3: **for all** target $\in Class$ **do**
4: $RuleSet \leftarrow RuleSet \bigcup$ **OneClassCN2**($\mathcal{D}$, $classValue$)
5: **end for**
6: **return** $RuleSet$
7: **end function**
8:
9: **function** ONECLASSCN2($\mathcal{D}$, classValue)
10: $rules \leftarrow \emptyset$
11: **repeat**
12: $bestCond \leftarrow$ **findBestCond** ($\mathcal{D}$, $classValue$)
13: **if** $bestCond \neq$ null **then**
14: $rules \leftarrow$ **addRule** ($BestCond$, $classValue$)
15: addWeightsToInstances($\mathcal{D}$, $bestCond$) ▷ Instances covered by the rule
16: **end if**
17: **until** $bestCondition =$ null
18: **return** $rules$
19: **end function**

is used internally by the algorithms.

4. Experimental Work and Discussion

In this section, we first describe the datasets used and then show the rules obtained by the SD algorithms for the individual datasets as well as a simple validation dividing the datasets into training and testing. Together with the rules we show the number of true positives (TP) and false positives (FP) captured by the rules to clarify of rule evaluation. Although there are multiple metrics that can be used to measure the goodness of the induced rules are, those metrics can all be derived from the number of true positives and false positives.

4.1. Datasets

In this paper we have used datasets from the Promise repository[5] [52] related to NASA projects (CM1, KC2, KC2, KC3, MC2, MW1 and PC1)

[5] `http://code.google.com/p/promisedata/`

Table 2: Description of the Datasets

Dataset	*# inst.*	*Non-def*	*Def*	*% Def.*	*Lang*
CM1	498	449	49	9.83	C
KC1	2,109	1,783	326	15.45	C++
KC2	522	415	107	20.49	C++
KC3	458	415	43	9.39	Java
MC2	161	109	52	32.29	C++
MW1	434	403	31	7.14	C++
PC1	1,109	1,032	77	6.94	C
AR	428	368	60	14.01	C
Eclipse JDT Core	997	791	206	20.66	Java
Eclipse PDE-UI	1,497	1,288	209	13.96	Java
Equinox	324	195	129	39.81	Java
Lucene	691	627	64	9.26	Java
Mylyn	1,862	1,617	245	13.15	Java

and projects from a Turkish white-goods manufacturer (AR) as well as from the Bug Prediction Dataset (BDP)[6] collected by D'Ambros et al. [16, 17] from open source Java projects.

Table 2 shows the number of instances (modules or classes) for each dataset with the number of defective and non-defective modules. The percentage of defective modules shows that all are highly imbalanced varying from 7% defective to 40%. The last column is the programming language used to develop those modules.

Table 3 summarizes the non-derived metrics collected for each module. The NASA datasets and the AR dataset from the Promise repository each contain different McCabe [47], Halstead [30] and branch-count metrics together with the binary class attribute (true or false indicating whether a module has reported defects). We have decided to use base metrics, i.e. non-derived metrics, as there is high variability in the metrics selected by feature selection algorithms [61]. The McCabe metrics are based on the count of the number of paths contained in a program based on its control flow graph [48]. The other set of metrics included in these datasets is Halstead's *Software Science* [30].

The NASA datasets were created from projects carried out at NASA and

[6] `http://bug.inf.usi.ch/`

Table 3: Summary of Metrics — Promise Datasets

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

collected under their metrics programme using the MaCabeIQ[7] tool. The AR dataset was obtained by combining several datasets (AR1, AR3, AR4, AR5, AR6) all from embedded systems developed in C by a Turkish white-goods manufacturer as the individual datasets were very small. In this case, the metrics were obtained using the PREST tool[8]. Although both the NASA and the AR datasets share the same metrics, we did not merge them as they were collected using different tools and belong to different domains. It has been reported that different tools can obtain very different results for the same metrics [46].

The Bug Prediction Dataset (BPD) by D'Ambros et al. is composed of Chidamber and Kemerer's object oriented metrics [13] (referred to here as the (C&K) metrics) as well as other metrics such as *fan-in*, *fan-out* and *number of attributes*. As with the previous dataset, we selected the C&K metrics because they are well-known and the most used in defect prediction [60, 10]. Using this set of reduced metrics helped to speed up the learning algorithms as and allows us to analyse and compare our results with previous work. The *wmc* metric represents the sum of the complexity of all methods for a class. If $w = 1$, this is equal to the number of methods. The *dit* metric counts the maximum level of the inheritance hierarchy of a class; the root of the

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

[8] `http://code.google.com/p/prest/`

Table 4: Summary of the C&K Metrics — D'Ambros et al. [17] Dataset

	Metric	*Definition*
C&K	*wmc*	Weighted Method Count
	dit	Depth of Inheritance Tree
	cbo	Coupling Between Objects
	noc	No. of Children
	lcom	Lack of Cohesion in Methods
	rfc	Response For Class
Class	true,false	Does the module contain defects?

inheritance tree is at level zero of the inheritance tree. The *cbo* metric for a class is a count of the number of other classes to which is coupled, i.e., when one class uses methods or variables of another class. The *noc* metric is the count of the number of subclasses belonging to a class. The *lcom* metric represents the extent to which methods reference a class instance data. The *rfc* metric is the count of the set of all methods that can be invoked in response to a message to an object of the class or by some method in the class. Table 4 summarizes the metrics collected for each Java class in the selected projects.

These metrics have been used for software quality assurance during testing and maintenance to prioritise testing effort, to assess comprehensibility and as thresholds. For example, if the *cyclomatic complexity* of a module is between 1 and 10 then it is considered to be a very low risk module; however, any module with a cyclomatic complexity greater than 50 is considered to have an unmanageable complexity and is a high risk module [11]. For essential complexity ($ev(g)$), the threshold value is 4. Similarly, other papers have suggested thresholds for the C&K metrics (e.g., [63, 7, 65, 31]). Although all these metrics have been used for some time, there are no clear thresholds and they are open to interpretation. For example, although McCabe [47, 68] 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. This may indicate the need to use multiple metrics as we suggest in this paper.

4.2. Inducted Rules for the Individual Datasets

We next present examples of rules and their quality measures induced for the KC2 dataset (Table 5 from the Promise repository) and the JDT Core (Table 6 from the BPD dataset) as an example of the rules obtained using the individual datasets. As stated previously, we used the Orange tool and its extension for subgroup discovery.

After carrying out a manual sensitivity analysis we maintained the default parameters in both algorithms as they allowed us to obtain rules with a good balance between specificity and generality. For the SD algorithm we can modify the minimal *support* (5%) and *confidence* (80%), the *generalisation* parameter g (5) and *beam width* (20) which refers to the number of solutions kept in each iteration of the algorithm. For the CN2-SD algorithm, we can modify the k (5) parameter, which represents the number of times an instance is kept before being removed from the training set (CN2-SD is a weighted covering algorithm). The rules were quite stable to modifications of the parameters because once a rules complies with the parameters, i.e., minimal support and confidence, such a rule will be kept in the solution and the best rules did not change as g varies. Also discretisation is an internal preprocessing step of the algorithms and as a result, the intervals created from the metrics remain constant (and consequently so do the rules).

As we can observe from Tables 5 and 6, rules tend to be accurate according to their *Positive Predictive Value* or *precision* rate ($\frac{TP}{TP+FP}$) or when comparing the number of true and false positives (TP or FP) with the total number of defective examples in the dataset. There are differences in the number of induced rules between the subgroup discovery algorithms. The CN2-SD algorithm tends to produce simpler and fewer rules covering more samples (defective and non-defective) as weights are used to increase the probability of inducing rules that cover unseen examples.

As we allow the SD algorithm to generate a large number of rules, some SD rules extend previous rules (rules that contain a previously induced rule) generating a pyramidal effect (the refined rules adding extra conditions a to previously induced rule). Such refined rules with more conditions are more accurate but cover fewer instances. For example, consider the last two rules of the SD algorithm in Table 6 (R_{18}: $rfc > 88$) and (R_{19}: $lcom > 171 \wedge rfc > 88$). The rule R_{19} covers 113 out of 206 modules and 72 false positives; however, R_{18} is more specific as it covers 74 instances with 48 false positives. However, although it is more specific this does not necessarily mean that it is a better rule as we must also consider the other metrics. For a practical

Table 5: Rules for the KC2 Dataset using the SD and CN2-SD Algorithms

	#	TP	FP	Precision	Rule
SD	0	26	2	0.93	$ev(g) > 4 \wedge totalOpnd > 117$
	1	30	3	0.91	$iv(g) > 8 \wedge uniqOpnd > 34 \wedge ev(g) > 4$
	2	29	3	0.91	$loc > 100 \wedge uniqOpnd > 34 \wedge ev(g) > 4$
	3	28	3	0.90	$loc > 100 \wedge iv(g) > 8 \wedge ev(g) > 4$
	4	26	3	0.90	$loc > 100 \wedge iv(g) > 8 \wedge totalOpnd > 117$
	5	33	4	0.89	$iv(g) > 8 \wedge uniqOp > 11 \wedge totalOp > 80$
	6	31	4	0.89	$iv(g) > 8 \wedge uniqOpnd > 34$
	7	31	4	0.89	$totalOpnd > 117$
	8	30	4	0.88	$loc > 100 \wedge iv(g) > 8$
	9	30	4	0.88	$ev(g) > 4 \wedge iv(g) > 8$
	10	37	5	0.88	$ev(g) > 4 \wedge uniqOpnd > 34$
	11	29	4	0.88	$loc > 100 \wedge ev(g) > 4$
	12	33	5	0.87	$iv(g) > 8 \wedge uniqOp > 11$
	13	41	7	0.85	$ev(g) > 4 \wedge totalOp > 80 \wedge v(g) > 6 \wedge uniqOp > 11$
	14	33	6	0.85	$iv(g) > 8 \wedge totalOp > 80$
	15	42	8	0.84	$ev(g) > 4 \wedge totalOp > 80 \wedge uniqOp > 11$
	16	42	8	0.84	$ev(g) > 4 \wedge totalOp > 80 \wedge v(g) > 6$
	17	34	7	0.83	$loc > 100 \wedge uniqOpnd > 34$
	18	43	9	0.83	$ev(g) > 4 \wedge totalOp > 80$
	19	33	7	0.83	$iv(g) > 8$
CN2SD	0	37	5	0.88	$uniqOpnd > 34 \wedge ev(g) > 4$
	1	43	9	0.83	$totalOp > 80 \wedge ev(g) > 4$
	2	83	88	0.49	$uniqOp > 11$

Table 6: Rules for the JDT Core Dataset using the SD and CN2-SD Algorithms

	#	TP	FP	Precision	Rule
SD	0	55	12	0.82	$cbo > 16 \wedge lcom > 171 \wedge wmc > 141 \wedge rfc > 88$
	1	62	14	0.82	$cbo > 16 \wedge rfc > 88 \wedge wmc > 141$
	2	55	13	0.81	$cbo > 16 \wedge lcom > 171 \wedge wmc > 141$
	3	62	15	0.81	$cbo > 16 wmc > 141$
	4	60	16	0.79	$lcom > 171 \wedge wmc > 141 \wedge rfc > 88$
	5	60	17	0.78	$lcom > 171 \wedge wmc > 141$
	6	69	21	0.77	$rfc > 88 \wedge wmc > 141$
	7	69	22	0.76	$wmc > 141$
	8	39	14	0.74	$wmc > 141 \wedge noc = 0$
	9	42	17	0.71	$noc > 0 \wedge rfc > 88 \wedge cbo > 16$
	10	93	38	0.71	$cbo > 16 \wedge rfc > 88$
	11	66	27	0.71	$cbo > 16 \wedge lcom > 171 \wedge rfc > 88$
	12	51	21	0.71	$cbo > 16 \wedge noc = 0 \wedge rfc > 88$
	13	34	17	0.67	$cbo > 16 \wedge wmc \in (68, 141]$
	14	35	18	0.66	$cbo > 16 \wedge lcom > 171 \wedge noc <= 0$
	15	46	24	0.66	$noc > 0 \wedge rfc > 88$
	16	69	38	0.64	$cbo > 16 \wedge lcom > 171$
	17	34	19	0.64	$noc > 0 \wedge rfc > 88 \wedge lcom > 171$
	18	113	72	0.61	$rfc > 88$
	19	74	48	0.61	$lcom > 171 \wedge rfc > 88$
CN2SD	0	62	15	0.81	$wmc > 141 \wedge cbo > 16$
	1	93	38	0.71	$rfc > 88 \wedge cbo > 16$
	4	113	72	0.61	$rfc > 88$

Table 7: Ranking of the C&K Attributes in the JDT Core and BPD Datasets

	JDT Core				*BPD Combined*			
	Inf gain	*Gain ratio*	*ReliefF*	χ^2	*Inf gain*	*Gain ratio*	*ReliefF*	χ^2
wmc	0.1657	0.1018	0.006534	263.872	0.06461	0.03673	0.001645	581.228
rfc	0.1494	0.1721	0.005472	248.415	0.07159	0.04772	0.001598	659.484
cbo	0.1397	0.0835	0.012259	222.905	0.06499	0.04184	0.002236	567.614
noc	0.0754	0.0494	0.007773	24.788	0.00219	0.00335	0.002616	17.236
lcom	0.0168	0.0207	0.000423	120.473	0.04412	0.02727	0.000121	396.321
dit	0	0	0.006563	0	0	0	0.001853	0

approach, we could opt for simpler or more accurate rules depending on the application domain, e.g. safety critical systems vs. management information systems.

It is also important to consider the thresholds defined by the discretisation which is carried out as a pre-processing step. Intervals, which can be seen as thresholds, differ among datasets. Fayyad and Irany's discretisation method [22] is a supervised discretisation method that generates intervals to maximise the probability of the class, i.e., detecting defective modules. The *dit* metric did not provide any classification information, i.e. it does not help to detect defective modules as its discretisation generated a single interval. As a result we discarded this metric. Similar observations have been reported in the literature [63, 26]. The *noc* metric also has poor prediction power and the rules mainly differentiate between classes with and without children. These findings are also corroborated by ranking the C&K attributes of two of the datasets using Weka[9] [69] with respect to various metrics as shown in Table 7. The *Information Gain* and *Gain ratio* [28] are metrics based on the entropy metric [59]. ReliefF [38, 40] is an instance-based based ranking method (we used the default parameter which is 10 neighbours and all instances). The χ^2 ranking method is based on the χ^2 statistic with respect to the class. We did not find such extreme cases in the attributes from the NASA and AR datasets.

4.3. Generalisation

For a project manager it may be interesting to extract general knowledge so that any new expertise is not confined to a single project. We can obtain general rules by applying the SD algorithms to metrics collected from several projects or systems. In our case, datasets in Table 2 are grouped according

[9] `http://www.cs.waikato.ac.nz/ml/weka/`

to their source. In this section, we work with three datasets: AR, NASA and BDP, (these correspond to the datasets described in Section 4.1) in order to determine whether the obtained rules are generalizable. We ran two different experiments. First of all we applied the algorithms to complete databases and then we divided the data into training and testing sets [5, 51].

Tables 8, 9 and 10 show the results for the combined datasets. The probability of a module being correctly identified as defective is the ratio between the number of defective modules and the total number of modules. Therefore, the probability that a random module is correctly identified as defective is 0.13, 0.14 and 0.16 for the NASA, AR and BDP datasets respectively according to Table 2. Out of all possible evaluation measures that can be calculated for a rule, we decided to use *precision* as it is both intuitive and useful. In this context, *precision* can be understood as the probability of a module being correctly identified as defective when covered by the condition(s) of the rule. Thus, a rule with high *precision* values will indicate which modules should be tested most thoroughly.

The rules induced by the SD algorithm correctly identifies defective modules with probability between 0.66 and 0.83 for the AR dataset (Table 8), between 0.43 and 0.47 for the NASA dataset (Table 9) and between 0.25 and 0.55 for the BDP dataset (Table 10). In the case of the CN2-SD algorithm, the range is larger and the rules are sorted from the most specific to the most generic.

Analysing the rules obtained by the SD algorithm, we can observe that they indicate thresholds for the metrics. The combination of the metrics with their threshold values can increase or decrease the specificity of the rule. For example, in the case of the AR dataset (Table 8), the threshold value for lines of code is set to 155 while the threshold values of $iv(g)$ and $v(g)$ are 3 and 17 respectively. In relation to the number of operators and operands, the ratio of the thresholds for *UniqOp* (15) and *UniqOpnd* (47) is approximately 1 to 3, while the ratio for *totalOpnd* (112) and *totalOp* (170) is approximately 1 to 1.5. In the case of the NASA datasets (Table 9), the threshold is lower for *loc* (88) and higher for complexity; the ratios between total operators and operands is maintained (1:1.5) although it is different for unique operators and operands (1:2).

In the case of the BDP dataset (Table 10), the thresholds of the *wmc* metric obtained by the SD algorithm are 24 and 87. Similarly, there are also two thresholds induced by SD for *cbo* (15 and 25). The threshold values for *lcom* and rfc are 45 and 68 respectively. The obtained rules are combina-

tions of these thresholds. The rules induced by the CN2-SD algorithm are much simpler and the first two rules are more specific than the ones induced by the SD algorithm; $wmc > 87 \wedge cbo > 25$ indicates a 0.7 probability of correctly identifying a defective module and when $rfc > 187$ the probability of correctly identifying a defective module is 0.63.

It can be observed that in the BDP datasets some rules define a range (e.g. rules R_2 $lcom \in (45, 1225]$) rather than a threshold (e.g. rules R_0 and R_1). From the software engineering point of view we could use the lower values of these metrics as thresholds (i.e., $lcom > 45$) instead of the intervals induced by the algorithms. The quality of the rule will be very similar if we ignore the upper limit. In relation to *noc* metric, its distribution is also very skewed; most modules do not have children (as stated previously) and so this metric is not a good discriminator for correctly identifying defective modules. Again, from the software engineering point of view, we could ignore this attribute and the rule quality will be very similar (although small differences can be observed in rules R_4 and R_5 using the SD algorithm).

An advantage of using the rules, in addition to simplicity, is that they provide information about how to combine metrics that can indicate error-prone modules. For example, when $cbo > 25$ a module may be identified as error-prone, however when *cbo* is moderate ($cbo \in (15, 25]$) but occurs with with high values of other metrics (e.g. $lcom \in (45, 1225]$) the probability of correctly identifying a defective module is much higher. Finally, generating a large number of rules, as the SD algorithm does, allows us to observe different threshold values and their consequences (c.f. rules R_0 and R_{18} in Table 10).

We can conclude that subgroup discovery rules can produce results for heterogeneous projects to indicate the probability of finding defect-prone modules. These rules are obviously less specific than the ones extracted from single projects but still useful to project managers or quality engineers who need some guidance in the testing process.

With respect to the comparison of SD and CN2-SD, the induced rules are different due to the different approaches of the algorithms. With SD, we can generate a larger number of rules with greater overlap (rules covering the same examples). We could apply some post-processing to select rules as will be explained in Section 4.4 depending on which rules are easier to apply or control. On the other hand, CN2-SD induces a smaller number of rules but with less overlap and similar specificity.

In order to check if the rules are generalizable we studied how the algorithms behave when different datasets are used for training and testing

Table 8: Rules of the Combined AR Dataset using the SD and CN2-SD Algorithms

	#	TP	FP	Precision	Rules
SD	0	19	4	0.83	$totalOpnd > 112 \wedge iv(g) > 3 \wedge uniqOp > 15 \wedge totalOp > 170$
	1	19	4	0.83	$uniqOp > 15 \wedge iv(g) > 3 \wedge v(g) > 17$
	2	20	5	0.8	$uniqOpnd > 47 \wedge v(g) > 17 \wedge uniqOp > 15$
	3	19	5	0.79	$uniqOpnd > 47 \wedge totalOp > 170 \wedge uniqOp > 15$
	4	18	5	0.78	$totalOpnd > 112 \wedge iv(g) > 3 \wedge v(g) > 17$
	5	18	5	0.78	$uniqOpnd > 47 \wedge iv(g) > 3 \wedge branchCount > 32$
	6	18	5	0.78	$loc > 155 \wedge totalOpnd > 112$
	7	18	5	0.78	$loc > 155 \wedge v(g) > 17$
	8	20	6	0.77	$uniqOpnd > 47 \wedge branchCount > 32 \wedge uniqOp > 15$
	9	19	6	0.76	$loc > 155$
	10	19	6	0.76	$totalOp > 170 \wedge iv(g) > 3$
	11	19	6	0.76	$totalOpnd > 112 \wedge iv(g) > 3 \wedge uniqOp > 15$
	12	23	8	0.74	$uniqOp > 15 \wedge iv(g) > 3 \wedge branchCount > 32$
	13	20	7	0.74	$uniqOpnd > 47 \wedge v(g) > 17$
	14	20	7	0.74	$uniqOpnd > 47 \wedge uniqOp > 15$
	15	17	6	0.74	$totalOpnd > 112 \wedge iv(g) > 3 \wedge uniqOpnd > 47$
	16	19	7	0.73	$uniqOpnd > 47 \wedge totalOp > 170$
	17	25	11	0.69	$uniqOp > 15 \wedge iv(g) > 3$
	18	20	9	0.69	$uniqOpnd > 47 \wedge branchCount > 32$
	19	23	12	0.66	$uniqOp > 15 \wedge totalOp > 170$
CN2SD	0	20	5	0.8	$v(g) > 17 \wedge iv(g) > 3$
	1	23	8	0.74	$uniqOp > 15 \wedge branchCount > 32 \wedge iv(g) > 3$
	2	25	11	0.69	$uniqOp > 15 \wedge iv(g) > 3$
	3	29	18	0.62	$uniqOp > 15 \wedge branchCount > 32$
	4	36	34	0.51	$uniqOp > 15$

Table 9: Rules Combining the Promise Datasets using the SD and CN2-SD Algorithms

	#	TP	FP	Precision	Rules
SD	0	132	147	0.47	$v(g) > 10 \wedge totalOp > 103 \wedge totalOpnd > 69 \wedge iv(g) > 7 \wedge branchCount > 20$
	1	133	150	0.47	$iv(g) > 7 \wedge totalOpnd > 69 \wedge branchCount > 20$
	2	138	156	0.47	$v(g) > 10 \wedge totalOp > 103 \wedge iv(g) > 7 \wedge branchCount > 20$
	3	133	155	0.46	$v(g) > 10 \wedge totalOp > 103 \wedge totalOpnd > 69 \wedge iv(g) > 7$
	4	121	142	0.46	$v(g) > 10 \wedge totalOp > 103 \wedge iv(g) > 7 \wedge uniqOpnd > 36$
	5	139	164	0.46	$v(g) > 10 \wedge totalOp > 103 \wedge iv(g) > 7$
	6	134	159	0.46	$iv(g) > 7 \wedge totalOpnd > 69 \wedge v(g) > 10$
	7	123	146	0.46	$v(g) > 10 \wedge iv(g) > 7 \wedge uniqOpnd > 36$
	8	122	145	0.46	$iv(g) > 7 \wedge totalOpnd > 69 \wedge uniqOpnd > 36 \wedge v(g) > 10$
	9	142	171	0.45	$iv(g) > 7 \wedge branchCount > 20$
	10	118	186	0.39	$v(g) > 10 \wedge iv(g) > 7 \wedge uniqOp > 17$
	11	148	186	0.44	$iv(g) > 7 \wedge totalOp > 103 \wedge totalOpnd > 69$
	12	151	192	0.44	$iv(g) > 7 \wedge totalOpnd > 69$
	13	125	159	0.44	$iv(g) > 7 \wedge totalOp > 103 \wedge uniqOp > 17$
	14	123	157	0.44	$iv(g) > 7 \wedge totalOp > 103 \wedge totalOpnd > 69 \wedge uniqOp > 17$
	15	127	163	0.44	$iv(g) > 7 \wedge totalOp > 103 \wedge uniqOpnd > 36$
	16	130	167	0.44	$iv(g) > 7 \wedge totalOpnd > 69 \wedge uniqOpnd > 36$
	17	131	169	0.44	$iv(g) > 7 \wedge uniqOpnd > 36$
	18	143	185	0.44	$v(g) > 10 \wedge iv(g) > 7$
	19	124	162	0.43	$iv(g) > 7 \wedge totalOpnd > 69 \wedge uniqOp > 17$
CN2	1	130	112	0.54	$loc > 88$
	2	251	462	0.35	$totalOp > 103$

Table 10: Rules Combining the BDP Datasets using the SD and CN2-SD Algorithms

	#	*TP*	*FP*	*Precision*	*Rules*
SD	0	181	148	0.55	$wmc > 87$
	1	185	190	0.49	$cbo > 25$
	2	139	251	0.36	$cbo \in (15, 25] \wedge lcom \in (45, 1225]$
	3	117	223	0.34	$cbo \in (15, 25] \wedge wmc \in (24, 87]$
	4	140	279	0.33	$cbo \in (15, 25] \wedge noc = 0$
	5	176	374	0.32	$cbo \in (15, 25]$
	6	227	496	0.31	$rfc \in (68, 187]$
	7	189	414	0.31	$lcom \in (45, 1225] \wedge rfc \in (68, 187]$
	8	124	277	0.31	$lcom \in (45, 1225] \wedge rfc \in (68, 187] \wedge wmc \in (24, 87] \wedge noc = 0$
	9	150	336	0.31	$lcom \in (45, 1225] \wedge rfc \in (68, 187] \wedge noc = 0$
	10	179	408	0.3	$rfc \in (68, 187] \wedge noc = 0$
	11	148	339	0.3	$lcom \in (45, 1225] \wedge rfc \in (68, 187] \wedge wmc \in (24, 87]$
	12	146	336	0.3	$rfc \in (68, 187] \wedge noc = 0 \wedge wmc \in (24, 87]$
	13	174	406	0.3	$rfc \in (68, 187] \wedge wmc \in (24, 87]$
	14	102	261	0.28	$lcom \in (45, 1225] \wedge noc > 0$
	15	230	667	0.26	$lcom \in (45, 1225] \wedge wmc \in (24, 87]$
	16	410	1202	0.25	$lcom \in (45, 1225]$
	17	179	533	0.25	$wmc \in (24, 87] \wedge noc = 0 \wedge lcom \in (45, 1225]$
	18	308	925	0.25	$wmc \in (24, 87]$
	19	308	941	0.25	$lcom \in (45, 1225] \wedge noc = 0$
CN2SD	0	128	56	0.7	$wmc > 87 \wedge cbo > 25$
	1	147	86	0.63	$rfc > 187$
	2	410	1202	0.25	$lcom \in (45, 1225]$

(Table 11). In this case we used half of the modules for training algorithms and the other half for testing. The figures shown in the table are the average results of running this process twice. The second column ($\%Def.Orig.$) shows the percentage of defective modules in each group of datasets. In addition, the table is divided vertically into two parts, one for each SD algorithm. Each part has two columns: the first column indicates the average *precision* of the rules generated by the algorithm; whereas the value of the second column indicates the percentage of defective modules captured ($\%Def.Capt.$) by the generated rules. This table shows that the obtained rules are generalizable and not dependent on the training data. It can be observed that for the NASA group application of SD rules increased the precision rate from the original 0.13 to 0.37. With the AR and BDP data the results are even better because the percision rate is above 50%. Results with CN2-SD are very similar.

The conclusion of this analysis is twofold: first, the software engineer can generate rules to improve the probability of correctly detecting defect prone modules using historical data and apply the rules to future development; and second, these rules provide knowledge about which metrics are best at correctly detecting defective modules and what threshold values we should

Table 11: Rule Generalisation Results

		SD		*CN2-SD*	
Dataset	*% Def.Orig.*	*Precision*	*% Def.Capt.*	*Precision*	*% Def.Capt.*
NASA	0.13	0.37	0.52	0.44	0.29
AR	0.14	0.58	0.55	0.62	0.63
BDP	0.16	0.51	0.40	0.44	0.53

Table 12: Rule Overlap

	SD			*CN2-SD*		
Dataset	*Avg. # Rules*	*% Rules*	*% Def.*	*Avg. # Rules*	*% Rules*	*% Def.*
CM1	13.86	69.32	77.08	3.40	68.00	83.33
KC1	14.69	73.44	34.36	2.54	50.83	62.88
KC2	13.22	66.10	46.73	2.89	57.86	78.50
MC2	11.23	56.17	57.69	2.79	55.86	55.77
MW1	15.69	78.44	51.61	3.95	78.95	61.29
PC1	9.76	48.78	53.25	2.16	43.20	64.94
AR	12.68	63.39	51.67	3.59	71.89	61.67
Eclipse JDT Core	10.07	50.34	57.28	4.02	80.35	54.85
Eclipse PDE-UI	14.47	72.33	62.68	1.03	20.61	62.68
Equinox	10.20	51.02	79.84	1.91	38.13	74.42
Lucene	9.68	48.38	57.81	3.73	74.67	46.88

use. This implies the rules provide a mechanism to establish which modules should be subjected to increased surveillance during the testing phase.

4.4. Overlapping and Rule Selection

From the software engineering point of view, we want to select a set of accurate rules covering a large number of samples in the dataset. However this is not an easy task and expert knowledge is required in most cases to select and understand the most useful rules (as individual pieces of knowledge) in order to apply them successfully. Although it is not the case with the CN2-SD algorithm, which creates a small number of rules, we might need to select rules from those generated by the SD algorithm or combine rules from different algorithms for optimal results.

In order to measure how much the rules generated by the algorithms overlap, we built Table 12. This table shows three values for each algorithm: the average number of rules which cover a unique defective module, the percentage with respect to the total number of rules, and the number of

defective modules covered by all the rules. For example, for CM1, each correctly identified defective module is covered by 13.86 rules on average (69.32% of 20) and 77.08% of all defective modules are covered.

In general, it can be seen that the SD algorithm provides numerous similar rules regarding instances covered, although the metrics and thresholds are different. To select a minimum set of rules that are as different as possible and cover the same records as the complete set of rules we have to define a distance measure between the rules based on their overlap.

Given a set of rules it is possible to measure their overlap by counting the number of different instances covered by the rules. If $c(R_i)$ is the set of instances covered by a rule, R_i, then we define the distance between two rules as:

$$dist(R_i, R_j) = |c(R_i)| + |c(R_j)| - 2 \cdot |(c(R_i) \bigcap c(R_j)| \tag{3}$$

where $|..|$ indicates the cardinality of the set. This value represents the number of different instances covered by any of the two rules and provides a measure of how different the rules are. If both rules cover the same instances, their distance will be equal to zero. The greater the number of different instances covered by the rules, the greater their distance. We can calculate a matrix of distances from a set of rules using Eq.(3) in order to cluster the rules. Each cluster will be composed of a set of rules that cover similar instances and *explain* similar characteristics. As an example, given the set of rules obtained by the algorithm SD from the AR dataset (Table 8), we could form 6 different clusters. The clusters are composed of 7 rules, 5 rules, two clusters with 3 rules and a cluster with a single rule. The question is: what information can these clusters provide? For example, the cluster with five rules is composed of the following rules:

- $totalOpnd > 112 \wedge iv(g) > 3 \wedge uniqOp > 15 \wedge totalOp > 170$
- $uniqOp > 15 \wedge iv(g) > 3 \wedge v(g) > 17$
- $totalOpnd > 112 \wedge iv(g) > 3 \wedge v(g) > 17$
- $totalOp > 170 \wedge iv(g) > 3$
- $totalOpnd > 112 \wedge iv(g) > 3 \wedge uniqOp > 15$

As it can be observed, in this cluster all rules have a common condition $iv(g) > 3$ and a number of different conditions involving the number of operands and operators that vary from rule to rule. Software quality or testing engineers can choose between these rules as they represent similar information. On the other hand, each cluster represents different conditions for the modules that are identified as error-prone. In this example, we could select 6 rules out of the 20 rules, one rule from each cluster, and we will be able to cover a very similar number of instances as with the whole ruleset but with minimal overlap. In this example, we could select for example the following rules, one from each cluster:

- $totalOp > 170 \wedge iv(g) > 3$
- $uniqOpnd > 47 \wedge v(g) > 17$
- $uniqOpnd > 47 \wedge iv(g) > 3 \wedge branchCount > 32$
- $loc > 155$
- $uniqOp > 15 \wedge totalOp > 170$
- $uniqOp > 15 \wedge iv(g) > 3$

Bar charts are a rule visualisation technique for binary classes [42] that can be used to evaluate and select rules in an intuitive way. The first bar represents the distribution of the class attribute (a colour for each value) where the imbalance can be observed with the percentage of positive cases to the right and negative to the left. The rest of the bars represent one rule each with their TP and FP ratios.

For example, Figure 5 shows bar chart representations of the rules obtained using the CN2-SD algorithm for the AR dataset. The first bar in these figures shows the proportion of non defective samples (on the left-hand side of the bar) vs. defective samples (on the right-hand side of the bar) for the entire dataset. The rest of the rules show the proportion of FP and TP for each rule. For example, the first rule for CN2-SD ($uniqOp > 34 \wedge ev(g) > 4$) covers 35% of the defective modules and only 1% of the non-defective ones. A simpler rule like $uniOp > 11$ covers a larger number of defective modules (78%) but also includes more non defective modules (21%). Both colours together represent the size of the subgroup, i.e., the number of instances covered by the rule.

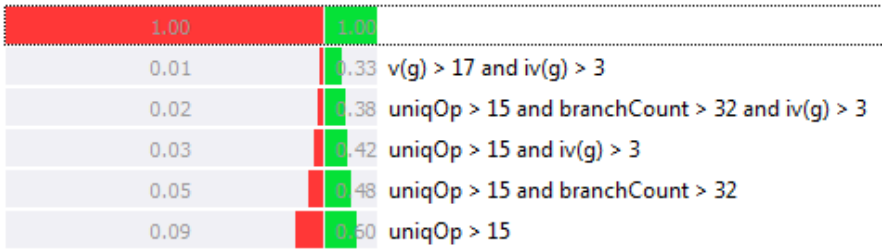


Figure 5: Bar Chart Visualization of Rules for the AR dataset using the CN2-SD Algorithm

5. Threats to Validity

There are some threats to validity [70] that we discuss in this section.

Construct validity is the degree to which the variables used in the study accurately measure the concepts they are intended to measure. As discussed previously, datasets are highly imbalanced and skewed. The algorithms used in this work do not directly work with continuous attributes and so we must discretise continuous attributes during pre-processing. Further work is needed to look into different discretisation techniques and transformation of the data as a preprocessing step before applying the SD technique. We have carried out some work to deal with continuous attributes [62] but the algorithm is not yet integrated with the Orange tool. Also, most of the metrics were selected manually, using non-derived metrics and the C&K metrics for the BDP datasets. The validity of these metrics has been investigated but is still open to debate [60].

We found problems in the datasets such as duplicates, inconsistencies etc. that need to be further investigated. Finally, there seems to be some agreement about the practical usefulness of metrics as predictors of quality, but there have also been some criticisms concerning their effectiveness.

Conclusion validity threats are related to finding the right relations between the treatments and the outcomes. Rules were induced using the algorithms' default parameters provided by the tool. Although results could be further optimised with different parameters, a preliminary sensitivity analysis did not change the most relevant rules. Furthermore, the aim of this paper is to explore the use of SD to generate simple rules that are easy to apply by practitioners. It is probably the case that within some application domains of software engineering, a higher percentage of errors (false positives) is admissible when compared with other disciplines such as medicine. Finally, the induced rules do not cover all the defective modules, i.e. this is

not a complete solution for the whole search space. Future research will analyse other quality metrics for sets of rules such as measures of the dispersion of a set of selected rules in addition to the ones outlined in this work.

Internal validity is the degree to which conclusions can be drawn. We used a relatively small number of datasets and although there is some degree of consistency among the metrics selected by the rules within each dataset, they vary among datasets and so do the thresholds. Shepperd and Kadoda [64] reported the influence of different data characteristics (dataset size, number, type and independence of features, and type of distribution) using simulated data over a number of different types of classifiers concluding there is no one best classifier as the characteristics of the data affect the outcomes.

External validity is the degree to which the results of the research can be generalised to the population under study and other research settings. It is suggested that the NASA repository can be generalised to industry in general [54]. However, the source of some of the data is not known and there are some issues with the quality of the data. The object-oriented datasets come from a unique open source project and therefore further replication studies are needed with other datasets, as well as other SD algorithms and application domains. There is also a problem with the size of the datasets as the complexity of the algorithms is $O(n^2)$. There are new algorithms that have been proposed in the literature [32]. However, they are spread across multiple tools which makes them difficult to compare.

6. Conclusions and Future Work

Subgroup Discovery (SD) as initially proposed by Klösgen is defined as the task of finding groups of individuals given a property of interest. Machine learning approaches to defect prediction in software engineering have mainly focused on classification or clustering techniques. SD algorithms can be both predictive (predicting the future given historical data and a property of interest) and descriptive (discovering interesting patterns in data). In our case, we used two SD algorithms to find rules for correctly predicting modules which are likely to be defective. The algorithms are robust to problems faced by classification techniques such as highly imbalanced data, noisy data, and data with high numbers of inconsistencies and redundancies. These problems are present in most defect prediction datasets in the software engineering domain.

In this paper, we analysed and compared different publicly available datasets using two Subgroup Discovery algorithms, SD and CN2-SD, showing that induced rules are capable of correctly characterising subgroups of modules with a reasonable probability of being defective. The set of induced descriptive rules is in general simpler and easier to use than those obtained through classification algorithms. This can provide a software engineering approach, i.e., an approach which is useful in practice, easily understandable and can be applied by project managers, testers and quality engineers alike.

The experimentation was carried out in three different stages. First, all datasets were analysed independently and the induced rules were found to increase the probability of detecting defective modules. In the second stage, we verified that subgroup discovery algorithms can obtain good results by combining all datasets into three new datasets according to their source. Finally, we analysed whether the rules could be generalised to unseen examples that were not present in the training datasets. The results show that induced rules can be used to correctly identify defect-prone modules. In any of the three stages discussed above, rules induced by subgroup discovery algorithms were simpler and easier to understand and apply. The main difference between these two algorithms, is that the SD algorithm can induce a large number of rules that cover the same instances while the CN2-SD algorithm takes any overlap of rules into account (weighted covering algorithm). However, in general, the rules obtained with both algorithms have similar values for support and precision. The rules obtained by the SD algorithm can be post-processed to select those that minimise overlap through a clustering process that finds sets of rules that cover the same instances (the same process could be used to select rules merged from different algorithms). It is worth noting that knowing the rules that overlap can be an advantage, because practitioners can establish relationships between the metrics and select different metrics that cover the same defects. In conclusion, we answer our research question in the affirmative: subgroup discovery can be used to detect the most defective modules in a system.

Future work will consist of further exploring other SD algorithms, datasets, quality measures (including multi-objective approaches) and the sensitivity of SD algorithms. There is a need for further investigation of issues related to imbalanced data as a part of feature selection as well as methods for evaluating the quality of defect prediction datasets.

Acknowledgements

The authors are grateful to Dr. Petra Kralj Novak for answering some of our questions about the Orange tool and the anonymous reviewers for their useful comments while improving this manuscript. This work has been supported by the projects ICEBERG (IAPP-2012-324356) and MICINN TIN2011-28956-C02-00. Also D. Rodríguez carried out part of this work as a visiting research fellow at Oxford Brookes University, UK.

References

[1] R. Agrawal, T. Imieliński, A. Swami, Mining association rules between sets of items in large databases, SIGMOD Rec. 22 (1993) 207–216.

[2] E. Arisholm, L.C. Briand, Predicting fault-prone components in a Java legacy system, in: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering (IESE'06), ACM, New York, NY, USA, 2006, pp. 8–17.

[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] D. Azar, H. Harmanani, R. Korkmaz, A hybrid heuristic approach to optimize rule-based software quality estimation models, Information and Software Technology 51 (2009) 1365 – 1376.

[5] D. Azar, J. Vybihal, An ant colony optimization algorithm to improve software quality prediction models: Case of class stability, Information and Software Technology 53 (2011) 388 – 393.

[6] S.D. Bay, M.J. Pazzani, Detecting group differences: Mining contrast sets, Data Mining and Knowledge Discovery 5 (2001) 213–246.

[7] S. Benlarbi, K.E. Emam, N. Goel, S.N. Rai, Thresholds for object-oriented measures, in: 11th International Symposium on Software Reliability Engineering (ISSRE 2000), pp. 24–39.

[8] S. Bibi, G. Tsoumakas, I. Stamelos, I. Vlahvas, Software defect prediction using regression via classification, in: IEEE International Conference on Computer Systems and Applications (AICCSA 2006), pp. 330–336.

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

[10] L. Briand, P. Devanbu, W. Melo, An investigation into coupling measures for C++, in: Proceedings of the 19th International conference on Software Engineering (ICSE'97), ICSE'97, ACM, New York, NY, USA, 1997, pp. 412–421.

[11] K. Brune, D. Fisher, J. Foreman, J. Gross, R. Rosenstein, M. Bray, W. Mills, D. Sadoski, J. Shimp, E. van Doren, C. Vondrak, M. Gerken, G. Haines, E. Kean, M.D. Luginbuhl, C4 Software Technology Reference Guide: A Prototype, Technical Report CMU/SEI-97-HB-001, Software Engineering Institute (SEI), Carnegie Mellon University, 1997.

[12] C. Catal, B. Diri, A systematic review of software fault prediction studies, Expert Systems with Applications 36 (2009) 7346 – 7354.

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

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

[15] J. Clarke, J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, M. Shepperd, Reformulating software engineering as a search problem, IEE Software 150 (2003) 161–175.

[16] M. D'Ambros, M. Lanza, R. Robbes, An extensive comparison of bug prediction approaches, in: 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), pp. 31–41.

[17] M. D'Ambros, M. Lanza, R. Robbes, Evaluating defect prediction approaches: a benchmark and an extensive comparison, Empirical Software Engineering 17 (2012) 531–577.

[18] G. Denaro, M. Pezze, An empirical evaluation of fault-proneness models, in: Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on, pp. 241 –251.

[19] G. Dong, J. Li, Efficient mining of emerging patterns: discovering trends and differences, in: Proceedings of the fifth ACM SIGKDD international

conference on Knowledge discovery and data mining, KDD '99, ACM, New York, NY, USA, 1999, pp. 43–52.

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

[21] T. Fawcett, An introduction to roc analysis, Pattern Recogn. Lett. 27 (2006) 861–874.

[22] U.M. Fayyad, K.B. Irani, On the handling of continuous-valued attributes in decision tree generation, Machine Learning 8 (1992) 87–102.

[23] U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, From data mining to knowledge discovery in databases., AI Magazine 17 (1996) 37–54.

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

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

[26] T. Gyimothy, R. Ferenc, I. Siket, Empirical validation of object-oriented metrics on open source software for fault prediction, IEEE Transactions on Software Engineering 31 (2005) 897–910.

[27] M. Halkidi, D. Spinellis, G. Tsatsaronis, M. Vazirgiannis, Data mining in software engineering, Intell. Data Anal. 15 (2011) 413–441.

[28] M. Hall, G. Holmes, Benchmarking attribute selection techniques for discrete class data mining, Knowledge and Data Engineering, IEEE Transactions on 15 (2003) 1437 – 1447.

[29] T. Hall, S. Beecham, D. Bowes, D. Gray, S. Counsell, A systematic literature review on fault prediction performance in software engineering, Transactions on Software Engineering (In Press – 2011).

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

[31] R. Harrison, S. Counsell, R. Nithi, Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems, The Journal of Systems and Software 52 (2000) 173–179.

[32] 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 29 (2011) 495–525.

[33] D.A. Houari A. Sahraoui, Quality estimation models optimization using genetic algorithms: Case of maintainability, in: Proc. of the 2nd European Software Measurement Conference (FESMA'99).

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

[35] 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.

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

[37] T.M. Khoshgoftaar, N. Seliya, Comparative assessment of software quality classification techniques: An empirical case study, Empirical Software Engineering 9 (2004) 229–257.

[38] K. Kira, L.A. Rendell, A practical approach to feature selection, in: Proceedings of the ninth international workshop on Machine learning, ML92, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992, pp. 249–256.

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

[40] I. Kononenko, Estimating attributes: analysis and extensions of relief, in: Proceedings of the European conference on machine learning on Machine Learning (ECML'94), ECML-94, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1994, pp. 171–182.

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

[42] P. Kralj, N. Lavrač, B. Zupan, Subgroup visualization, in: Proceedings of the 8th International Multiconference Information Society (IS 2005), pp. 228–231.

[43] P. Kralj Novak, N. Lavrč, G.I. Webb, Supervised descriptive rule discovery: A unifying survey of contrast set, emerging pattern and subgroup mining, Journal of Machine Learning Research 10 (2009) 377–403.

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

[45] 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.

[46] R. Lincke, J. Lundberg, W. Löwe, Comparing software metrics tools, in: Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA'08), ISSTA'08, ACM, New York, NY, USA, 2008, pp. 131–142.

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

[48] T.J. McCabe, C.W. Butler, Design complexity measurement and testing, Communications of the ACM 32 (1989) 1415–1425.

[49] T. Mende, R. Koschke, Revisiting the evaluation of defect prediction models, in: Proceedings of the 5th International Conference on Predictor Models in Software Engineering (PROMISE'09), ACM, New York, NY, USA, 2009, pp. 1–10.

[50] T. Mende, R. Koschke, Effort-aware defect prediction models, in: Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering (CSMR'10), CSMR'10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 107–116.

[51] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, T. Zimmermann, Local vs. global lessons for defect prediction and effort estimation, IEEE Transactions on Software Engineering Preprint (2012).

[52] T. Menzies, B. Caglayan, E. Kocaguneli, J. Krall, F. Peters, B. Turhan, The PROMISE repository of empirical software engineering data, 2012.

[53] 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.

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

[55] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, A. Bener, Defect prediction from static code features: currentresults, limitations, new approaches, Automated Software Engineering 17 (2010) 375–407. 10.1007/s10515-010-0069-5.

[56] 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.

[57] 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.

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

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

[60] D. Radjenović, M. Heričo, R. Torkar, A. Živkovič, Software fault prediction metrics: A systematic literature review, Information and Software Technology (2013) In press.

[61] D. Rodriguez, R. Ruiz, J. Cuadrado, J. Aguilar-Ruiz, Detecting fault modules applying feature selection to classifiers, IEEE International Conference on Information Reuse and Integration (IRI'07), pp. 667–672.

[62] D. Rodriguez, R. Ruiz, J. Riquelme, J. Aguilar-Ruiz, Searching for rules to detect defective modules: A subgroup discovery approach, Information Sciences 191 (2012) 14–30.

[63] R. Shatnawi, W. Li, J. Swain, T. Newman, Finding software metrics threshold values using roc curves, Journal of Software Maintenance and Evolution: Research and Practice 22 (2010) 1–16.

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

[65] Y. Singh, A. Kaur, R. Malhotra, Empirical validation of object-oriented metrics for predicting fault proneness models, Software Quality Journal 18 (2010) 3–35. 10.1007/s11219-009-9079-6.

[66] Q. Song, M. Shepperd, M. Cartwright, C. Mair, Software defect association mining and defect correction effort prediction, IEEE Transactions on Software Engineering 32 (2006) 69–82.

[67] 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.

[68] A.H. Watson, T.J. McCabe, D.R. Wallace, Structured testing: A testing methodology using the cyclomatic complexity metric, NIST special Publication 500 (1996) 1–114.

[69] I. Witten, E. Frank, M. Hall, Data Mining: Practical machine learning tools and techniques, Morgan Kaufmann, San Francisco, 3rd edition edition, 2011.

[70] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in software engineering: an introduction, Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[71] 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.

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

[73] T. Xie, S. Thummalapenta, D. Lo, C. Liu, Data mining for software engineering, IEEE Computer 42 (2009) 55–62.

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

[75] T. Zimmermann, R. Premraj, A. Zeller, Predicting defects for eclipse, in: International Workshop on Predictor Models in Software Engineering (PROMISE'07), p. 9.