International Journal of Software Engineering
and Knowledge Engineering
Vol. 24, No. 1 (2014) 61–90

DOI: 10.1142/S021819401450003X

World Scientific
www.worldscientific.com

Investigating Associative Classification for Software Fault Prediction: An Experimental Perspective

Baojun Ma
School of Economics and Management
Beijing University of Posts and Telecommunications
Beijing 100876, P. R. China
mabaojun@bupt.edu.cn

Huaping Zhang
School of Computer Science & Technology
Beijing Institute of Technology
Beijing 100081, P. R. China
kevinzhang@bit.edu.cn

Guoqing Chen*
School of Economics and Management
Tsinghua University, Beijing 100084, P. R. China
chengq@sem.tsinghua.edu.cn

Yanping Zhao
School of Management and Economics
Beijing Institute of Technology
Beijing 100081, P. R. China
zhaoyp@bit.edu.cn

Bart Baesens
Faculty of Business and Economics
Katholieke Universiteit Leuven
Leuven B-3000, Belgium
Bart.Baesens@econ.kuleuven.be

Received 30 April 2013
Revised 19 June 2013
Accepted 24 September 2013

It is a recurrent finding that software development is often troubled by considerable delays as well as budget overruns and several solutions have been proposed in answer to this observation, software fault prediction being a prime example. Drawing upon machine learning techniques,

*Corresponding author.

software fault prediction tries to identify upfront software modules that are most likely to contain faults, thereby streamlining testing efforts and improving overall software quality. When deploying fault prediction models in a production environment, both prediction performance and model comprehensibility are typically taken into consideration, although the latter is commonly overlooked in the academic literature. Many classification methods have been suggested to conduct fault prediction; yet associative classification methods remain uninvestigated in this context. This paper proposes an associative classification (AC)-based fault prediction method, building upon the CBA2 algorithm. In an empirical comparison on 12 real-world datasets, the AC-based classifier is shown to achieve a predictive performance competitive to those of models induced by five other tree/rule-based classification techniques. In addition, our findings also highlight the comprehensibility of the AC-based models, while achieving similar prediction performance. Furthermore, the possibilities of cross project prediction are investigated, strengthening earlier findings on the feasibility of such approach when insufficient data on the target project is available.

Keywords: Software fault prediction; associative classification; prediction performance; comprehensibility; cross project validation.

1. Introduction

Computers have become omnipresent in everyday life, and also software development has become a key economical activity, which aims to deliver high-quality and reliable software with as few errors or defects as possible [1]. A defect can be defined as any situation in the code, which prevents the software system from operating according to its specifications. Defects are also commonly referred to as bugs or faults, and we will use the terms interchangeably [2]. Since defective software modules could cause software failures, increase development and maintenance costs, and decrease customer satisfaction, effective defect prediction models can help developers focus quality assurance activities on defect-prone modules, thereby improving software quality by using resources more efficiently [3].

Software defect prediction is typically regarded as a binary classification task, which involves categorizing modules, represented by a set of software metrics or code attributes, into two classes, i.e. fault-prone (*fp*) and non-fault-prone (*nfp*), by means of a classification model derived from data of previous development projects [4]. There has been ample work discussing the different classifiers used in the domain of software fault prediction, which can be broadly divided into two groups, i.e. statistical approaches including discriminant analysis [5, 6], logistic regression [7, 8] and Bayesian networks [9, 10], as well as machine learning methods such as rule/tree-based methods [3, 11], SVM [12], neural networks [13, 14], fuzzy clustering methods [15], analogy-based approaches [16] and ensemble methods [17]. Using statistical models validated with a specific organization's data, is not convenient to be used in another company because statistical models are seriously dependent on data and different organizations may have different data features [18]. According to the literature review of Catal and Diri [19], machine learning algorithms have become the most popular approach to fault prediction. Furthermore, it was pointed out that certain complex classifiers, such as SVM and neural networks, result in the unclear relationship between input and prediction, which are also referred to as black box

solutions [20]. Lessmann *et al.* [21] showed that simple classifiers, such as Naive-Bayes or decision trees, are often competitive with more sophisticated approaches, i.e. not significantly inferior, suggesting that most methods do not differ significantly in terms of predictive performance. Consequently, the assessment and selection of a classification model should not solely be based on predictive performance but also on alternative criteria like computational efficiency, ease of use, and especially *comprehensibility*. Models providing insight into factors influencing the presence of software faults and helping improve our overall understanding of software failures and their sources, which in return, may enable the development of novel predictors of fault-proneness [5, 21].

From this perspective, tree/rule-based classifiers seem to be the preferred choice over, e.g. SVM and neural networks, given their white box nature and similar classification performance. However, traditional tree rule-based classifiers, such as C4.5 [22], RIPPER [23], CART [24] and DT [25], derive *local* sets of rules in a *greedy* manner, which means these rules are discovered from partitions of the training data set instead of from the whole training set [26] and this could result in many interesting and useful rules not being discovered [27, 28]. In contrast, associative classification (AC) methods, first proposed by Liu *et al.* [27], combine classification and association rule mining, exploring the complete training data set in an attempt to construct a global classifier [26]. Several studies have provided evidence that AC algorithms are able to build classifiers competitive with those produced by tree/rule-based methods or probabilistic approaches such as Bayesian networks, SVM and neural networks [27–29]. Obviously, AC classifiers are typically extremely comprehensible as they result in a collection of classification rules in the form of "$X \Rightarrow C$", where X is a set of data items, and C is the class label.

Both proprietary and public datasets have been investigated in the domain of software fault prediction. Recently however, there has been a shift towards using public data sets in software engineering studies, as it has been argued that studies relating to proprietary data are more difficult to verify [18, 19]. Public datasets mostly originate from PROMISE [30] and NASA MDP (Metrics Data Program) [31] repositories which are freely accessible by researchers, and as such attracted considerable attention [18, 19, 32, 33]. Nevertheless, recent work often involve only a small selection of these public datasets, such as three datasets in [34], two in [35], five in [17] and [36].

Software fault prediction models are assessed on the basis of confusion matrix based criteria, including *accuracy*, *recall*, *precision*, *sensitivity*, *specificity*, *F-measure*, *G-mean*, or the more recently introduced *AUC* measure. In addition, other aspects are also important when deploying fault prediction models in a development environment, including ease of use, computational efficiency and especially model comprehensibility. Despites its importance, the latter remains an often-overlooked aspect of classification models [37, 38].

In this paper, we investigate an associative classification approach based on CBA2, proposed by Liu *et al.* [28], and adjust it to the specifics of the domain of

software fault prediction. The proposed AC-based approach is contrasted with five other tree/rule-based classification methods, i.e. C4.5, Decision Table, CART, ADT and RIPPER, across twelve public-domain benchmark data sets obtained from the NASA MDP repository and the PROMISE repository. Comparisons are mainly based on the Area under the ROC curve (AUC) and three comprehensibility metrics. As argued later in this paper, the AUC represents the most informative indicator of predictive accuracy within the field of software defect prediction. Furthermore, we also investigated a cross project validation approach to assess whether rule sets learned on one data set are applicable to other data sets.

In summary, the main contributions of this paper are as follows:

- The investigation of associative classification based software fault prediction, which allows to account for specific characteristics encountered in this domain such as the skewed class distribution. Note that an associative classification based approach has not been adopted in this field to date, despite the promising results obtained in other settings [39]. This associative classification approach is contrasted against five other tree/rule-based methods in an extensive benchmarking experiment involving 12 public data sets, answering the call made by, e.g. [40] concerning repeatable results.
- The aspect of model *comprehensibility*, which is an important but often overlooked criterion when deploying classification models, is defined from the point of view of tree/rule-based classifiers and is subsequently considered during the benchmarking experiments.
- A silent assumption made in software fault prediction research is that sufficient data is available to learn a classification model. However, when developing new software, only historic data from previous, completed projects will be available. This observation constitutes the rationale for considering a cross project validation approach in which data from different projects are used to learn and validate the classification models. This also allows to investigate the generalization capabilities of the proposed AC classifier.

This paper is organized as follows. In Sec. 2, we introduce the research background, including software fault prediction and associative classification. Section 3 specifies our proposed associative classification based fault prediction method from four stages. Section 4 discusses software defect data and performance evaluation measures. In Sec. 5, we provide the experimental setup and discussion of the results. Finally, a conclusion is presented.

2. Related Work

2.1. *Software fault prediction*

A key finding to software testing is that faults tend to cluster; i.e. to be contained in a limited subset of software modules. Examples hereof can be found in the work of, e.g.

Gyimothy *et al.* [41] who investigated the open source software web and email suite Mozilla and found that bugs were present in 42.04% of all modules and Ostrand *et al.* [42] who reported even more skewed class distributions. This observation, paired with the fact that correcting errors later in the software development life cycle becomes increasingly more expensive [43], lead to the investigation of early warning mechanisms which try to identify upfront fault prone regions in the code. Several approaches have hereto been proposed, including, e.g. software reliability modeling which try to predict which components will fail first [44]. Such approach however considers a different timing, assuming software already to be released. Software fault prediction on the other hand takes a different perspective by situating itself right after coding, before software testing and deployment.

Several so-called static code features can be obtained from the code base of a software project using automated methods and, together with historic data on which modules were faulty in, e.g. a previous release, form the basis on which software fault prediction models can learned. Such models typically discriminate between modules, which are likely to contain errors or are fault prone (*fp*) and those that are not fault prone (*nfp*). Given the extensive code bases software project managers are often faced with, an automatable approach to identify software modules that are likely to contain faults seems especially feasible to allow to streamline the testing process, an activity which can otherwise take up to 50% of the total development budget [45].

During the software development process, accurately identifying the software quality of software systems plays a critical role in targeting quality improvement efforts to the high-risk modules. However, only predicting the exact amount of faults is too risky, especially in the beginning of a software project when too little information is available [46]. Generally, software managers and researchers commonly adopt approaches that classifying software modules into two groups, *fp* or *nfp*, in order to identify the *fp* category easily [47]. Hence, providing an accurate and reliable software fault prediction model has become more and more important for effectively assuring quality in software systems.

There has been ample work applying various types of classifiers to construct fault prediction models, and software project managers often have difficulties in choosing an appropriate approach from the many alternatives discussed in the literature as results regarding the superiority of one method over another are not always consistent across studies [48]. Moreover, the aspect of *comprehensibility* is often neglected, which is, however, often of critical importance for model acceptance [5, 21]. Almeida and Matwin [20] state that some statistical models are to be considered black-box solutions because the relationship between inputs and responses is unclear, and also signpost their data dependency [20]. When considering machine learning methods, tree/rule-based algorithms seem the most attractive to software project managers due to their comprehensibility and satisfactory predictive effectiveness [5, 21, 49]. However, traditional tree-based or rule-based classifiers, such as C4.5 [22], RIPPER [23], CART [24] and ADT [25] derive *local* sets of rules in a *greedy* manner, which implies that many interesting and useful rules would not be discovered. The

derived rules are *local* because when a rule is discovered, all training data objects associated with it are discarded and the process continues until the next rule found has an unacceptable error rate. This means rules are discovered from partitions of the training data set and not from the whole training data set. The search process for the rules is greedy as most rule induction algorithms look for the rule that maximizes some statistical measure [26].

2.2. *Associative classification*

Associative classification (AC), which was firstly proposed by Liu *et al.* [27] with the CBA algorithm, aims to integrate classification rule mining and association rule mining to build a classifier based on a special discovered subset of association rules (i.e. class association rules, CARs). Association rule mining can be defined as follows [50]: Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of items and D be a set of transactions (the dataset), where each transaction t (a data record) is a set of items such that $t \subseteq I$. An association rule is an implication of the form, $X \Rightarrow Y$, where $X \subset I, Y \subset I$ are called itemsets, and $X \cap Y = \phi$. A transaction t is called to contain X, if $X \subseteq t$. The rule $X \Rightarrow Y$ holds in the transaction set D with confidence *conf* if *conf*% of transactions in D that supports X also supports Y. The rule has support *supp* in D if *supp*% of the transactions in D contains $X \cup Y$. Given a set of transactions D (the dataset), the problem of mining association rules is to discover all the rules that have support and confidence greater than the user-specified minimum support (denoted as *minsup*) and minimum confidence (denoted as *minconf*). An efficient algorithm for mining association rules is the Apriori algorithm, which was proposed by Agrawal and Srikant [50].

A class association rule or CAR takes the form $X \Rightarrow C$, where X is a set of data items, and C is the class (label) and a predetermined target. With such a rule, a transaction or data record t in a given database could be classified into class C if t contains X. Apparently, a class association rule could be regarded as an association rule of a special kind. To build a classifier using an AC algorithm, the complete set of class association rules (CARs) is first discovered from the training data set and a subset is selected to form the classifier.

Since the introduction of the CBA algorithm in 1998, many studies have investigated alternative associative classification approaches, such as CBA2 [28], CMAR [51], CPAR [52], MMAC [53], CAAR [54] and GARC [29]. The CBA algorithm directly employs the Apriori-type approach for mining classification rules in form of $X \Rightarrow C$ and uses them to predict new data records based on user-defined threshold values of *minsup* and *minconf* [27]. This study investigates the CBA2 algorithm, which modifies the way the *minsup* threshold is set during rule generation [28]. More specifically, CBA2 allows to define different *minsup* values depending on the class (i.e. each class is assigned a different *minsup*), rather than using only a single *minsup* value as is the case in CBA, which can potentially improve classification performance in case of an unbalanced class distribution. This is also the main reason of its

application to software defect prediction. Associative classifiers have been applied to many other domains, including document classification [55] and recommendation systems [39, 56].

There are several points about associative classification needing attention. Small value of *minsup* may cause too many CARs to be generated, resulting overfitting, which requires the pruning of redundant rules [28, 29]. Furthermore, the two user-specified predefined thresholds (i.e. *minsup* and *minconf*), especially for *minsup*, have a considerable impact on the generation of CARs. Although sensitive analysis can be utilized to quantify the impact of both parameters [29], it is often impractical to apply this strategy in real software fault prediction applications. Some researchers suggested to adopt cross-validation or bootstrapping to address this problem [21, 48]. Finally, classification datasets often contain continuous attributes. Association rule mining techniques on the other hand usually involve transactional data and as such, a discretization of these continuous attributes is required [57].

3. Associative Classification Based Fault Prediction

This section introduces our associative classification approach which is based on the CBA2 algorithm previously proposed by Liu *et al.* [28]. Software fault prediction is often characterized by a skewed class distribution, motivating the choice of CBA2 as basis in this study. In contrast to e.g. the CBA algorithm, CBA2 allows to determine a specific *minsup* threshold for each class, potentially improving classification performance in case of an unbalanced class distribution.

There are four main stages in the processing of our proposed associative classification based fault prediction method, i.e. data preprocessing, rule generation, classifier building and prediction, as well as prediction results evaluation, which are illustrated in Fig. 1. Since data preprocessing is conducted on the experimental datasets and prediction results evaluation mainly discusses prediction performance and classifier comprehensibility, to avoid duplicate statements, these two parts would be introduced and discussed in Secs. 4.2 and 4.3 in detail respectively. Thus in this section, we focus on the stages of rule generation (see Sec. 3.2) as well as classifier building and prediction (see Sec. 3.2).

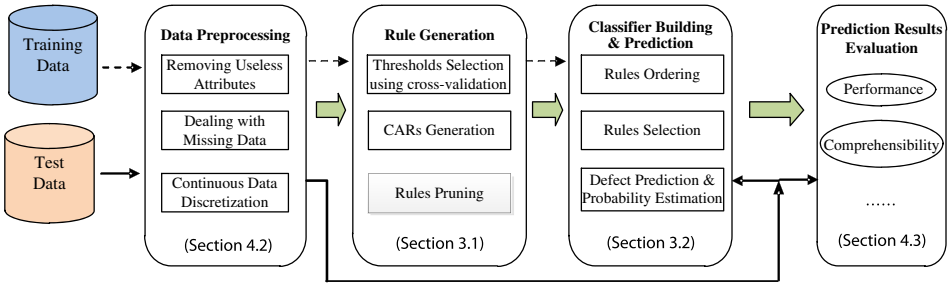


Fig. 1. The processing of the associative classification based fault prediction method.

3.1. *Rule generation*

The rule generation process, resulting in the CAR set, constitutes the main part of the associative classification inference. The typical approach adopted in, e.g. CBA [27], CBA2 [28], GARC [29], is to use a predefined *minsup* and *minconf* threshold. These parameters can however seriously impact classifier performance as e.g. valid rules are being rejected if the first parameter is set too high and as such, the CARs may fail to cover all the training cases. As a result, the generated rule set will be unable to properly discriminate the minority class [27]. By contrast, if the *minsup* value is set too low, overfitting might occur [28]. Although the CBA2 algorithm already includes a multiple *minsup* values strategy [28] and a sensitive analysis by Chen *et al.* [29] investigated the impact of both parameters, an approach in which these thresholds are determined in advance seems most feasible, especially given the skewed class distribution in the domain of software fault prediction.

In this study, a 10-fold cross-validation approach [58] was adopted to address the above concern, see Algorithm 1. More specifically, a grid-search procedure was implemented, based on a set of candidate values for both parameters. This model

Algorithm 1 Optimal Thresholds Selection Algorithm

Inputs:

$\mathbf{D_{tr}}$: training data set;

CT: candidates threshold pairs set, $\boldsymbol{CT} = \{<minsup_1, minconf_1>, \ldots, <minsup_m, minconf_m>\}$;

Outputs:

Optimal threshold pair, $<minsup_{opt}, minconf_{opt}>$;

$\mathbf{D_{tr}} \rightarrow_{\text{randomly}} \mathbf{D_{tr1}}, \mathbf{D_{tr2}}, \ldots, \mathbf{D_{tr9}}$;
$opt \leftarrow 1$, $\text{AUC}_k = 0$ $(k = 1, 2, \ldots, m)$;
for $i = 1$ to m do
  for $j = 1$ to 9 do
    $\text{CARSet}_j \leftarrow \text{MiningCARs}(\mathbf{D_{tr}}, \mathbf{D_{tr}}_j, <minsup_i, minconf_i>)$;
    $\text{Classifier}_j \leftarrow \text{BuildingClassifier}(\text{CARSet}_j, \mathbf{D_{tr}})$;
    $\text{AUC}_j \leftarrow \text{Evaluation}(\text{Classifier}_j, \mathbf{D_{tr}}_j)$;
  end for
  $\text{AUC}_i \leftarrow \text{mean}(\text{AUC}_j)$
  if $i > 1$ then
    if $\text{AUC}_i > \text{AUC}_{opt}$ then
      $opt \leftarrow i$;
    end if
  end if
end for
Return $<minsup_{opt}, minconf_{opt}>$.

selection step is guided by the *AUC* as criterion of choice, see also Sec. 4.3.1. That is, the training data set is randomly divided into 10 parts of approximately equal size (line 1). Iteratively, 9 of the 10 parts are used for training by employing the current *minsup* and *minconf*, resulting in a CAR set and classifier (line 5–6). The last part is then used to evaluate the candidate solution (line 7) and the combination of hyper parameters resulting in the best performance on this independent test set is selected as optimal threshold during the remainder of the model building (lines 10–16).

After determining the optimal thresholds, the CARs are generated on all training data by application of the CBA2 algorithm. In fact, the generation of the CARs in CBA2 is based on the CBA-RG algorithm of CBA. The basic idea lies in generating all possible frequent rule-items by making multiple passes over the training data based on $minsup_{opt}$ and producing CARs from this set of frequent rule-items based on $minconf_{opt}$, see also [27]. However, to extract useful CARs related to the minority class, the global $minsup_{opt}$ value was multiplied by the fraction of each class, resulting in $minsup^{c}_{opt}$. Any rule related to class c satisfying the $minsup^{c}_{opt}$ would be regarded as a frequent CAR.

During the CARs generation process, several rule pruning strategies can be applied to substantially reduce the size of the generated rule set. In this study, the pessimistic error rate pruning proposed in the context of C4.5 [22] is adopted. This procedure goes as follows: if rule r's pessimistic error rate is higher than the pessimistic error rate of rule r', obtained by deleting one condition from rule antecedent of rule r, then rule r is replaced by the shorter rule r'. Note it is also possible to remove redundant and/or conflicting rules and during the process of rule generation, see e.g. [29].

3.2. *Classifier building and prediction*

The CARs generated in Sec. 3.1 allow to construct the final classifier and subsequently allow for inference on the test data.

Hereto, an ordering on all CARs is imposed based on the following definition of rule precedence:

Definition 1: Given two rules, r_i and r_j, r_i precedes $r_j (r_i \succ r_j)$ if

(a) Confidence(r_i) > Confidence(r_j), or
(b) Confidence(r_i) = Confidence(r_j), but Support(r_i) > Support(r_j), or
(c) Confidence(r_i) = Confidence(r_j) and Support(r_i) = Support(r_j), but r_i is generated earlier than r_j.

Then, the final classifier is constructed by selecting a set of high precedence CARs (denoted as R) to cover the training data set. As such, the classifier generated has the following format: $< r_1, r_2, \ldots, r_m,$ *default_class*$>$, where $r_i \in R, r_a \succ r_b$ *if* $b > a$, which is similar to that of C4.5. Note that the confidence of each rule can be used to estimate the prediction probability. Each rule is marked with its rule confidence,

which could be used to estimate prediction probability needed when calculating the *AUC*.

With the above classifier, any unseen module could be classified as either *fp* or *nfp* by traversing the ordered rule set until a rule with matching rule antecedent is found; if no rule fires, the default class is assigned.

4. Software Defect Data and Performance Evaluation Measurements

4.1. *Software defect datasets*

The data sets used in this study stem from the NASA MDP repository [31]. A total of twelve public software fault prediction data sets are analyzed, including those investigated by [21] as well as two additional data sets (i.e. MC1 and MC2). Each data set contains details on several software modules (e.g. java source files), together with their number of faults and static code characteristics. Besides LOC counts, the NASA MDP data sets contain several Halstead attributes as well as McCabe complexity measures. The former estimates complexity by counting the number of operators and operands in a module, whereas the latter is derived from a module's flow graph, quantifying the number of independent paths through a program. The reader is referred to [38] for a more detailed description on the MDP data sets.

4.2. *Data pre-processing*

A first important step in each data mining exercise is preprocessing the data. In order to contrast the proposed associative classification technique discussed in Sec. 3 against the other rule based learners presented in Sec. 5.1.2, the same preprocessing steps are applied to each of the twelve data sets. Each observation (software module or file) in the data sets consists of a unique ID, several static code features and an error count. First, the data used to learn and validate the models are selected and thus, the *ID* as well as attributes exhibiting zero variance is discarded. The *error count* indicates the number of faults associated with each module and is discretized into a Boolean value where 0 indicates that no errors were recorded for this software module or file and 1 otherwise, in line with, e.g. [21] and [37]. The error density is also removed, as this attribute indicates the number of faults per line of code and thus perfectly correlates with the target attribute.

While static code features can be mined in an automated way, some data sets exhibit missing values for the decision density attribute. To retain as much data as possible, an average value imputation schema is used where appropriate.

As the proposed associative classification approach is unable to cope with continuous features, a discretized version of each data set was constructed using the algorithm of Fayyad and Irani [59]. This supervised discretization algorithm uses entropy to select subintervals that are as pure as possible with respect to the target attribute.

Finally, it should be noted that machine learning techniques typically perform better if more data to learn from are available. On the other hand, part of the data needs to be put aside as an independent test set in order to provide a realistic assessment of the performance. As can be seen from Table 2, the smallest data set contains 125 observations, while the largest contains up to 9,537 observations. Each of the data sets is randomly partitioned into two disjoint sets, i.e. a training and a test set consisting of respectively 2/3 and 1/3 of the observations, using stratified sampling in order to preserve the class distribution [11, 21, 37, 47]. To account for a potential sampling bias, this partitioning procedure is repeated ten times in total.

After performing these steps, the data sets are passed to the learners described in Sec. 3 and the reference learners discussed in Sec. 5.1.2.

4.3. *Evaluation measurements*

4.3.1. *Prediction performance*

Binary classifiers (i.e. classifiers with dichotomous outcomes) are routinely assessed using a confusion matrix (also called a contingency table). A confusion matrix summarizing the number of modules correctly or incorrectly classified as fault prone (*fp*) or not fault prone (*nfp*) by the classifier is shown in Fig. 2. In our study, we call the software modules with defects "positive" cases, whereas the modules without faults are termed "negative" cases. If *TP*, *TN*, *FP*, and *FN* represent respectively the number of true positive, true negative, false positive and false negative, than several common evaluation metrics can be defined as follows.

The *accuracy* is the proportion of total modules that were predicted correctly:

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

The *recall* (also called *hit rate* or *sensitivity*) is the percentage of fault-prone modules that are correctly predicted:

$$recall = \frac{TP}{TP + FN}. \tag{2}$$

The *precision* is the proportion of correctly predicted fault-prone modules in all the predicted fault-prone modules:

$$precision = \frac{TP}{TP + FP}. \tag{3}$$

		Actual Class	
		fp	***nfp***
Predicted Class	***fp***	**T**rue **P**ositive (**TP**)	**F**alse **P**ositive (**FP**)
	nfp	**F**alse **N**egative (**FN**)	**T**rue **N**egative (**TN**)

Fig. 2. Confusion matrix.

The *specificity* is the proportion of correctly identified defect-free modules:

$$specificity = \frac{TN}{FP + TN}. \tag{4}$$

The *F-measure* is the harmonic mean of *recall* and *precision*, defined as follows:

$$F_\beta - measure = \frac{(\beta^2 + 1) \cdot recall \cdot precision}{\beta^2 \cdot recall + precision}. \tag{5}$$

In Eq. (5), β takes any non-negative value and is used to control the weight assigned to *recall* and *precision*. If β is set to one, than recall and precision are weighted equally, giving the F_1-*measure*.

In the domain of software fault prediction, the number of fault-prone modules is typically much smaller than the number of non-defective modules; for example, PC2 and MC1 only contain 0.51% and 1.47% *fp* modules respectively, as can be seen in Table 2. Furthermore, it is often stated that the misclassification cost of *fp* modules is different from that of *nfp* modules [38]. Therefore, recent work advices against the usage of accuracy, especially in contexts characterized by skewed data distribution and unequal classification costs [60–62]. Moreover, Menzies *et al.* [63] stated that also precision is not a useful parameter for software engineering problems.

Besides the F_1-*measure*, there are several others that take the class imbalance into account, such as *geometric mean* (*G-mean*) [64] and *Area under ROC curve* (*AUC*) [65]. $G\text{-}mean_1$ and $G\text{-}mean_2$ are defined as follows:

$$G - mean_1 = \sqrt{recall \cdot precision}. \tag{6}$$

$$G - mean_2 = \sqrt{recall \cdot specificity}. \tag{7}$$

Although Ma *et al.* [66] suggested using the *G-mean* and *F-measure* to evaluate the performance of imbalanced datasets, the above confusion matrix metrics still suffer from the limitation that they require a predefined cut-off value for predicted probabilities, which may leave considerable room for bias and cause inconsistencies across studies [11, 21].

As such, there is a growing trend in fault prediction research to quantify classification performance in terms of the *AUC* measure [67].

The ROC graph is a two-dimensional plot, in which *sensitivity* is plotted on the *Y* axis against 1-*specificity*, see Fig. 3. A ROC curve depicts the relative tradeoff between benefits (true positives) and costs (false positives) and allows one to assess performance of a prediction model in general, regardless of any particular cut-off value [11]. In addition, the Area under ROC curve (*AUC*) has the potential to significantly improve convergence across empirical experiments in software defect prediction because it separates predictive performance from operating conditions, i.e. class and cost distributions, and thus represents a general measure of predictability [21]. Furthermore, the *AUC* has a clear statistical interpretation: it measures the probability that a classifier ranks a randomly chosen *fp* module higher than a randomly chosen *nfp* module, which is equivalent to the Wilcoxon test of ranks [61].

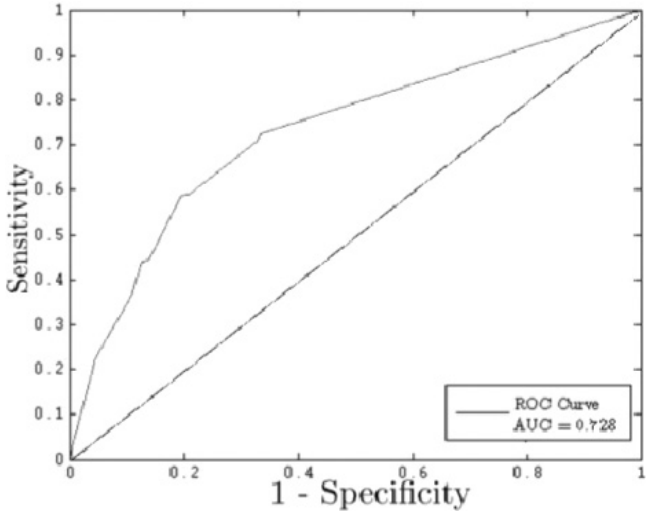


Fig. 3. ROC curve (model trained on the KC1 data set, evaluated on the JM1 data set).

In this study, our experimental comparisons mainly focus on the AUC. In order to illustrate the ineffectiveness and inconsistency between other metrics and the AUC, the values of *accuracy, recall, precision*, *specificity*, F_1*-measure*, $G\text{-}mean_1$ and $G\text{-}mean_2$ are also calculated.

4.3.2. *Comprehensibility*

Comprehensibility can be a key requirement for a classification model, demanding that the end user can understand the rationale behind the model's prediction. Although defining comprehensibility for a classification model is close to being a philosophical discussion, literature indicates that the type of output and the size of output play an important role [68]. Although the comprehensibility of a specific output type is largely domain dependent, rule-based or tree-based classifiers can be considered as the most comprehensible while nonlinear classifiers are typically regarded as less comprehensible. Smaller models are also to be preferred over more elaborate ones [68].

As this study only compares different kinds of tree-based and rule-based classifiers, only the size of the classifier output needs to be considered. For rule-based classifiers, the outputs are classification rule sets. While referring to tree-based classifiers, we could also regard each leave with all its related previous decision nodes in the tree as a classification rule. First, for a given rule output, the comprehensibility decreases with the size, i.e. the number of rules, implying that simple models are preferred over more complex ones [69]. Second, speaking in a rule-based context, the more the conditions (i.e. rule antecedents), the harder it is to understand [70]. In

addition, for a given number of conditions, it is better to have many rules with a low average number of conditions per rule than few rules with many conditions [68].

Based on the above discussion, the *number of rules, total number of conditions in the rule set as well as average number of conditions per rule is* adopted as assessment criteria for classifier comprehensibility.

5. Experiments and Results

In this section, the experimental setup is described and subsequently, the empirical results are presented in detail, together with a discussion of possible limitations and threats to validity of this study.

5.1. *Experimental setup*

5.1.1. *The details of our proposed AC-based method*

In the experimental study, we first aim to investigate the performance of the proposed AC-based software fault prediction method on 12 NASA MDP data sets. Although the open source CBA package is perhaps the most popular AC toolkit, it lacks specific functionalities key to our research. CBA[a] does not allow for evaluation criteria others than accuracy during the cross-validation stage, which is infeasible given the skewed class distribution of software fault prediction data sets. Instead, we opted for the LUCS-KDD implementation of the CBA algorithm [71] and made several modifications and improvements to achieve our goals. Firstly, LUCS-KDD only implements the basic CBA algorithm with a single *minsup* value [71]; the ability of defining multiple *minsup* values was added to account for the class imbalances problem. Hereto, we considered the principles outlined by the CBA2 algorithm. Secondly, as the original implementation of LUCS-KDD still utilized *accuracy* as its evaluation metric, seven evaluation metrics were added: *precision*, *recall*, *specificity*, F_1-*measure*, G-$mean_1$, G-$mean_2$ and AUC. Furthermore, a cross-validation procedure was added to LUCS-KDD, which is based on the AUC to obtain optimal *minsup* and *minconf* thresholds before building the final prediction model.

Note that this cross validation procedure is only adopted during parameter selection; the final performance is estimated using a random hold-out testing procedure considering 2/3 of all available data for model building while the remaining 1/3 of the data is set aside for testing. Besides providing an unbiased estimate of the classifier's generalization performance, the split-sample setup offers the advantage of enabling easy replication, which constitutes an important part of empirical research [48, 72]. Furthermore, its choice is motivated by the fact that the split-sample setup is the prevailing approach to assess performance in the domain of software defect prediction [16, 21, 73, 74].

[a] http://www.comp.nus.edu.sg/∼dm2/

All the modifications and improvements described above were implemented by using Java language, and the experimental environment is a PC with a quad-core CPU 2.50 GHz, 3.96 GB RAM, running Microsoft Windows Server 2003 Standard Edition.

5.1.2. *Benchmarking learners*

As the *comprehensibility* of classification models when deploying decision models in a business environment is often of critical importance, different rule and decision tree learners were also considered. These are C4.5 [22], Classification and regression tree (CART) [24], Alternating Decision Trees (ADT) [25] and Repeated Incremental Pruning to Produce Error Reduction (RIPPER) [23]. Also the paradigm of decision tables (DT) [75] was investigated as it was previously found that such approach also can result in compact and comprehensible models. These benchmark learners are all implemented in WEKA, an open source package available under the GNU public license [76]. Specifics on these learners are given in Table 1; further details can be found in academic literature on this topic.

Note that the above classifiers exhibit adjustable parameters, also termed hyperparameters, which enable the algorithm to be adjusted to a specific problem setting, see Table 1. Similar to our approach to tuning the hyperparameters for the AC learner, a grid-search approach is adopted during this selection phase, again adopting the AUC as criterion of choice. Besides AUC, also calculated the *accuracy*, *recall*, *precision*, *specificity*, F_1*-measure*, $G\text{-}mean_1$ *and* $G\text{-}mean_2$ are calculated. Comprehensibility on the other hand is assessed by the *number of rules*, *total number of conditions in rule set* and *average number of conditions per rule*.

5.1.3. *Cross project validation*

Finally, a cross project validation setup was also considered, training a model on data stemming from one specific NASA project while validating using data from other (NASA) projects. In principle, the scope of such cross project validation is not limited to data stemming from the NASA MDP repository, but as such validation approach requires an identical input space to learn from, many alternative data sources are infeasible. Moreover, as there exist also differences in the input space of NASA MDP data sets, the cross project validation is investigated on the only 9 of the 12 data sets

Table 1. Summary of five selected rule/tree-based classifiers.

Classifier	WEKA path	Hyperparameters
C4.5	weka.classifiers.trees.J48	confidenceFactor, minNumObj
DT	weka.classifiers.rules.DecisionTable	evaluationMeasure, search
CART	weka.classifiers.trees.SimpleCart	minNumObj, numFoldsPruning,
ADT	weka.classifiers.trees.ADTree	numOfBoostingIterations
RIPPER	weka.classifiers.rules.Jrip	folds, optimizations

(i.e. CM1, KC3, MC1, MC2, MW1, PC1, PC2, PC3, PC4). More specifically, for these 9 data sets, the classification model is trained on one data set (e.g. CM1), while the other 8 data sets are used to validate the learned model (e.g. KC3, MC1, MC2, MW1, PC1, PC2, PC3, PC4). Note that for reasons of brevity, only the result of the cross project validation when applying the novel AC approach is discussed.

5.2. *Experimental results*

5.2.1. *Results of classifier performance*

As outlined in Fig. 1, the first step of the proposed AC approach involves the empirical derivation of optimal *minsup* and *minconf* thresholds, the results of which are shown in Table 2. It can be observed that the *fp* percentage of some data sets is relatively close to the *minsup* value, as is the case for the CM1, KC3, MW1 and PC1 data sets. Other data sets however exhibit larger differences between the two values, especially for data sets with a small amount of *fp* modules, e.g. MC1 and PC2.

The predictive performance of the different classifiers is summarized in Table 3; the best value is indicated in bold font. The last column of Table 3 displays the Average Rank (i.e. AR) for each algorithm. The AR is calculated by ranking all techniques according to their performance on each dataset, rank 1 indicating the best performance and rank 6 the worst. The ARs are then obtained by averaging the ranks across all twelve data sets.

First of all, although all classification methods perform dissimilar, it can be observed that some data sets are "easier" to learn from than others, as is, e.g. the case for the KC4 data set versus the CM1 data set. When considering the different classification performance measures discussed in Sec. 4.3, it can be noted that alternative conclusions can be drawn depending on the metric of choice. More specifically, DT and CART classifiers outperform the other classifiers on *accuracy* and *specificity* measures while AC and RIPPER perform better than other methods when considering F_1-*measure* and G-$mean_1$. AC and DT show clear advantage over the other methods on AUC; the AC approach itself yields decent performance on *recall* and G-$mean_2$. In addition, it seems that there are no obvious differences amongst the six classifiers on *precision*. The above results also imply that comparing classifiers performance by means of multiple evaluation metrics results in an inconclusive outcome. Some of these measures, such as *classification accuracy*, are known to be inappropriate to the domain of software fault prediction [11, 21, 63]. Referring

Table 2. The optimal thresholds (i.e. *minsup*, *minconf*) of AC-based method on each NASA MDP data set.

	CM1	JM1	KC1	KC3	KC4	MC1	MC2	MW1	PC1	PC2	PC3	PC4
$minsup_{opt}$	9%	30%	14%	7%	35%	45%	50%	10%	5%	10%	20%	40%
$minconf_{opt}$	95%	85%	85%	95%	70%	95%	80%	70%	50%	95%	95%	95%

Table 3. Experimental results of AC, C4.5, DT, CART, ADT and RIPPER algorithms on different metrics.

Measures	Classifier	CM1	JM1	KC1	KC3	KC4	MC1	MC2	MW1	PC1	PC2	PC3	PC4	AR
Accuracy (%)	AC	80.36	73.52	83.71	90.91	85.37	95.00	**69.81**	91.04	91.78	99.20	86.48	83.96	4.3
	C4.5	**85.12**	80.31	81.34	85.31	78.05	98.70	60.38	90.30	88.39	99.00	89.26	**88.64**	4.6
	DT	83.93	81.11	**83.76**	**91.61**	**87.80**	98.51	**69.81**	**93.28**	**93.20**	99.27	88.07	87.75	2.3
	CART	84.52	**81.52**	83.05	**91.61**	**87.80**	**98.90**	66.04	91.04	**93.20**	**99.67**	**89.86**	88.08	**1.9**
	ADT	83.33	81.05	83.48	81.82	80.49	98.83	67.92	91.04	**93.20**	99.27	**89.86**	85.97	3.3
	RIPPER	84.52	80.89	82.91	89.51	**87.80**	98.83	66.04	91.79	92.07	99.13	89.07	**88.64**	3.2
Recall (Sensitivity)	AC	0.200	**0.461**	**0.445**	**0.333**	0.722	**0.500**	0.333	**0.500**	**0.440**	**0.455**	0.255	**0.648**	**1.5**
	C4.5	0.200	0.231	0.107	0.167	0.556	0.300	**0.500**	0.250	0.160	0.000	0.235	0.426	3.3
	DT	0.050	0.119	0.157	0.083	**0.778**	0.250	0.222	0.125	0.200	0.000	0.078	0.148	4.1
	CART	0.200	0.086	0.091	0.000	**0.778**	0.250	0.167	0.250	0.160	0.000	0.000	0.298	4.1
	ADT	0.050	0.021	0.083	0.167	0.722	0.250	0.333	0.375	0.280	0.000	0.000	0.593	3.7
	RIPPER	**0.300**	0.243	0.182	**0.333**	**0.778**	0.350	0.167	0.250	0.240	0.091	**0.333**	0.500	2.2
Precision	AC	0.190	0.357	0.480	0.444	0.929	0.130	0.600	**0.333**	0.423	**0.455**	0.302	0.398	3.7
	C4.5	0.308	0.479	0.361	0.154	0.909	0.500	0.429	0.222	0.167	0.000	0.444	0.535	4.3
	DT	0.111	0.550	0.613	**0.500**	**0.933**	0.385	**0.667**	**0.333**	0.556	0.000	0.235	0.471	**2.7**
	CART	0.286	0.667	0.550	0.000	**0.933**	**0.714**	0.500	0.250	**0.571**	0.000	0.000	**0.649**	2.9
	ADT	0.100	**0.882**	**0.667**	0.111	0.813	0.625	0.545	0.300	0.538	0.000	0.000	0.438	3.6
	RIPPER	**0.333**	0.503	0.512	0.364	**0.933**	0.583	0.500	0.286	0.400	0.250	**0.447**	0.529	2.9
Specificity	AC	0.885	0.801	0.910	0.962	**0.957**	0.956	0.886	0.919	0.954	0.994	0.934	0.866	5.1
	C4.5	0.939	0.940	0.960	0.916	**0.957**	0.996	0.520	0.944	0.939	0.998	0.967	0.949	4.1
	DT	**0.946**	0.977	0.979	0.992	**0.957**	0.995	**0.943**	**0.984**	0.988	**1.000**	0.971	**0.977**	2.0
	CART	0.932	0.990	0.985	**1.000**	**0.957**	**0.999**	0.914	0.952	**0.991**	**1.000**	**1.000**	0.974	**1.8**
	ADT	0.939	**0.999**	**0.991**	0.878	0.870	0.998	0.857	0.944	0.982	**1.000**	**1.000**	0.896	3.1
	RIPPER	0.919	0.943	0.964	0.947	**0.957**	0.997	0.914	0.960	0.973	0.998	0.954	0.939	3.5
G-mean$_1$	AC	0.195	**0.406**	**0.463**	**0.385**	0.819	0.255	0.447	**0.408**	**0.431**	**0.455**	0.278	0.508	2.3
	C4.5	0.248	0.333	0.197	0.160	0.711	0.387	**0.463**	0.236	0.163	0.000	0.323	0.477	3.8
	DT	0.074	0.256	0.310	0.204	**0.852**	0.310	0.385	0.204	0.333	0.000	0.135	0.264	3.8
	CART	0.239	0.240	0.224	0.000	**0.852**	0.422	0.289	0.250	0.302	0.000	0.000	0.440	4.1
	ADT	0.071	0.136	0.235	0.136	0.766	0.395	0.426	0.335	0.388	0.000	0.000	0.510	3.8
	RIPPER	**0.316**	0.350	0.305	0.348	**0.852**	**0.452**	0.289	0.267	0.310	0.151	**0.386**	**0.514**	**2.2**

(*Continued*)

Table 3. (*Continued*)

Measures	Classifier	CM1	JM1	KC1	KC3	KC4	MC1	MC2	MW1	PC1	PC2	PC3	PC4	AR
G-mean$_2$	AC	0.421	**0.608**	**0.637**	**0.566**	0.831	**0.691**	**0.543**	**0.678**	**0.648**	**0.672**	0.488	**0.749**	**1.6**
	C4.5	0.433	0.466	0.320	0.391	0.729	0.547	0.510	0.486	0.388	0.000	0.477	0.636	3.8
	DT	0.217	0.341	0.392	0.287	**0.863**	0.499	0.458	0.351	0.445	0.000	0.275	0.380	4.2
	CART	0.432	0.292	0.299	0.000	**0.863**	0.500	0.391	0.488	0.398	0.000	0.000	0.539	4.3
	ADT	0.217	0.145	0.287	0.383	0.793	0.499	0.534	0.595	0.524	0.000	0.000	0.729	3.9
	RIPPER	**0.525**	0.479	0.419	0.561	**0.863**	0.591	0.391	0.490	0.483	0.301	**0.563**	0.685	2.3
F_1-Measure	AC	0.195	**0.402**	**0.462**	**0.381**	0.813	0.206	0.429	**0.400**	**0.431**	**0.455**	0.277	0.493	2.3
	C4.5	0.242	0.312	0.166	0.160	0.690	0.375	**0.462**	0.235	0.163	0.000	0.308	0.474	3.4
	DT	0.069	0.195	0.250	0.143	**0.848**	0.303	0.333	0.128	0.294	0.000	0.118	0.225	4.1
	CART	0.235	0.152	0.156	0.000	**0.848**	0.370	0.250	0.250	0.250	0.000	0.000	0.409	4.2
	ADT	0.067	0.042	0.147	0.133	0.765	0.357	0.414	0.333	0.368	0.000	0.000	0.504	4.1
	RIPPER	**0.316**	0.328	0.268	0.348	**0.848**	**0.438**	0.250	0.267	0.300	0.133	**0.382**	**0.514**	**2.0**
AUC	AC	0.598	0.688	**0.836**	0.696	0.835	0.862	**0.671**	**0.860**	**0.827**	**0.809**	**0.821**	**0.885**	2.3
	C4.5	0.645	0.710	0.711	0.597	0.874	0.817	0.573	0.597	0.601	0.783	0.726	0.817	4.0
	DT	0.642	**0.718**	0.778	**0.749**	**0.906**	**0.896**	0.660	0.737	0.758	0.795	0.793	0.867	**1.9**
	CART	0.614	0.673	0.694	0.500	0.901	0.784	0.540	0.604	0.575	0.500	0.5	0.817	4.9
	ADT	**0.653**	0.705	0.745	0.700	0.852	0.819	0.580	0.765	0.805	0.794	0.743	0.842	2.8
	RIPPER	0.613	0.593	0.572	0.644	0.901	0.673	0.540	0.605	0.604	0.544	0.642	0.723	4.8

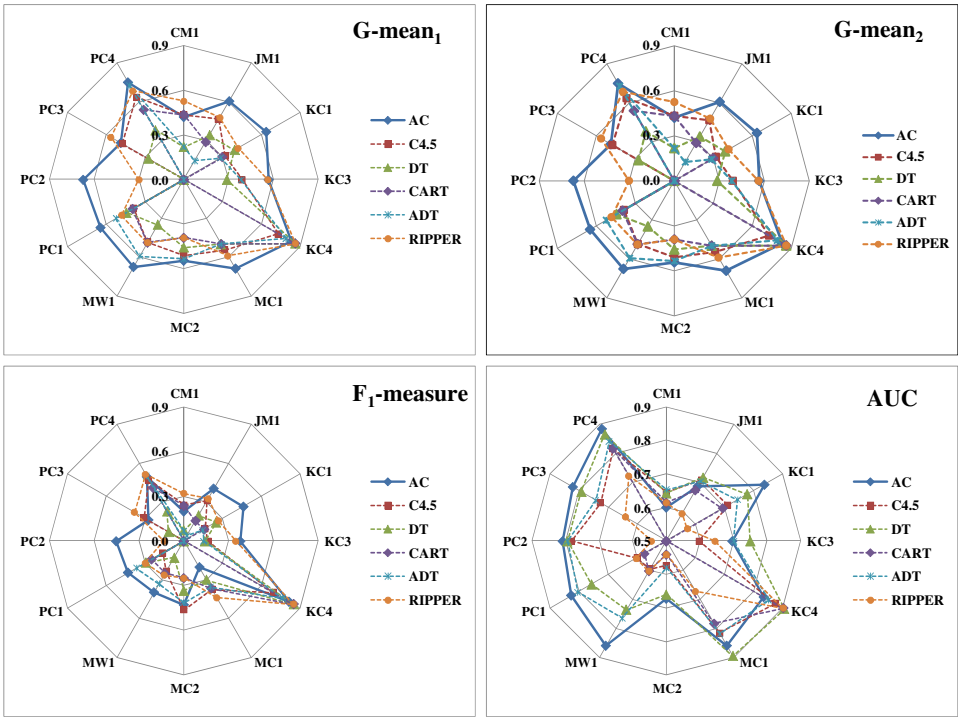


Fig. 4. Radar diagrams comparing AC, C4.5, DT, CART, ADT and RIPPER on G-mean, F1 and AUC.

to results on measures considering imbalance of data (i.e. F_1-*measure*, G-$mean_2$, G-$mean_1$ and AUC), radar diagrams are derived for easier comparison of the results, see Fig. 4. It is found that the proposed AC-based method outperforms almost all the other five classifiers on F_1-*measure*, G-$mean_2$, G-$mean_1$ and AUC measures, as the circle representing AC method almost lies outside all the other circles in each radar diagram.

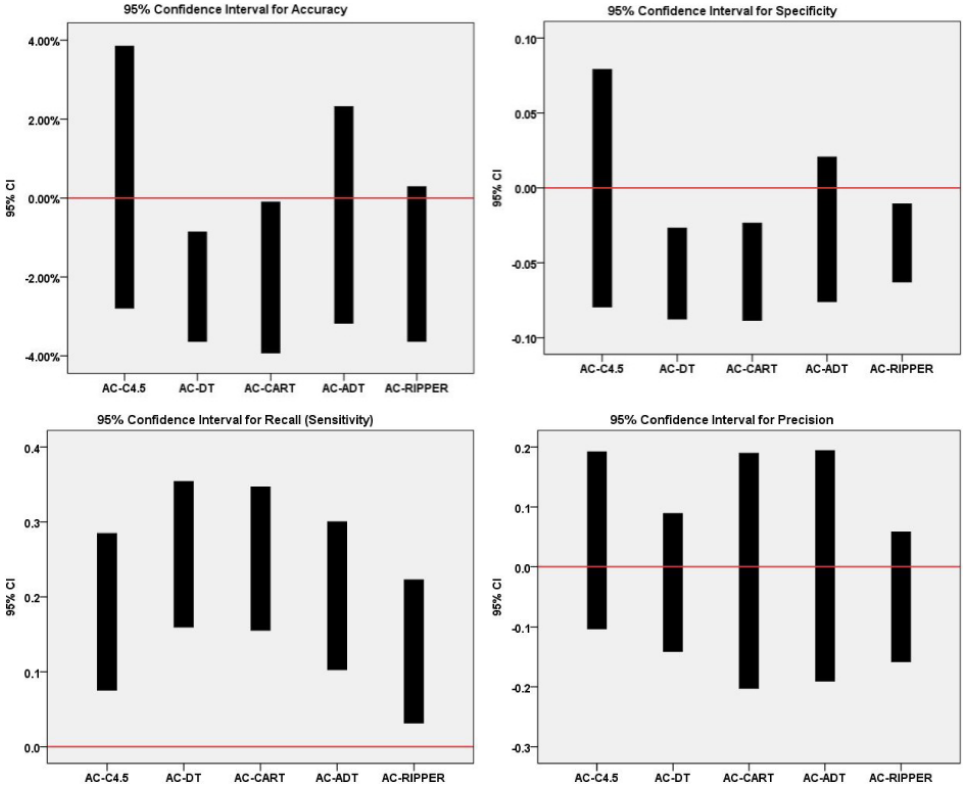
The above presented empirical results are submitted to statistical analysis to verify whether specific results are due to luck, or in fact reflect underlying trends. Hereto, the 95% confidence interval (denoted as 95%-CI) on the results are calculated for each method. When comparing the results of two classifiers on a specific metric, it suffices to consider the 95%-CI of both techniques; a difference between two methods on a specific measure is statistically significant at the 95% level when both confidence intervals do not overlap one another. The results of this statistical analysis are provided in Fig. 5, comparing the proposed AC learner with the benchmark learners. Note that an analogous analysis was also performed considering a significance level of 90%, which resulted in similar findings; due to space constraints, these are however not further detailed.

Based on this statistical analysis, it can be seen that, for G-$mean_1$ and F_1-*measure*, the proposed AC method obtains superior results over other methods except

RIPPER, which can be attributed to the fact that $G\text{-}mean_1$ and $F_1\text{-}measure$ are both combinations of *recall* and *precision*. Note that AC outperforms all the other methods in terms of $G\text{-}mean_2$. Most importantly, our AC method beats all other approaches except DT when considering the AUC, which should be the preferred evaluation metric when dealing with imbalanced class distributions. As such, we can conclude that AC and DT significantly outperform tree and rule based learners in the domain of software fault prediction.

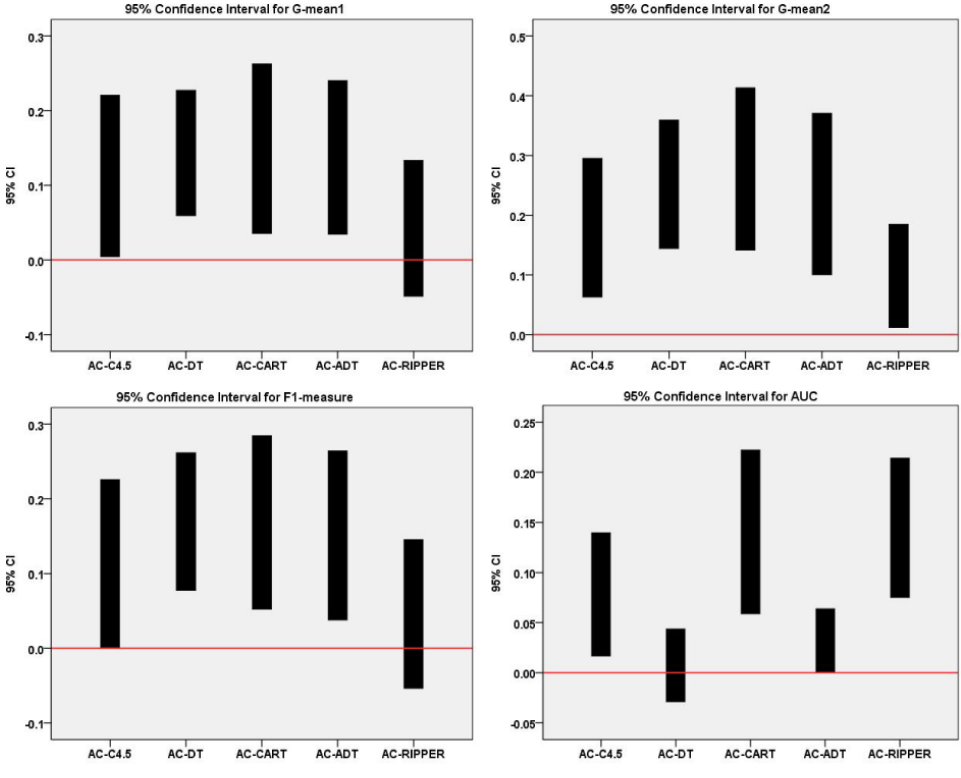
5.2.2. *Results of classifier comprehensibility*

Section 4.2.2 defined three metrics which, in the case of tree or rule based classifiers, can serve as a proxy to the somewhat intangible concept of classifier comprehensibility, i.e. *number of rules*, *number of conditions* in the rule set and *average number of conditions per rule*. Based on these metrics, the different classifiers can be ordered according to this concept, based on the following definition of comprehensibility



(a)

Fig. 5. 95% Confidence intervals of differences between AC and benchmark classifiers.



(b)

Fig. 5. (*Continued*)

precedence of rule set for tree-based or rule-based classifiers:

Definition 2: Given two rule sets of tree/rule-based classifiers, R_i and R_j, R_i has a higher comprehensibility precedence than R_j if

(a) R_i has fewer *number of rules* than R_j, or
(b) R_i has equal *number of rules* with R_j, but R_i has fewer *number of conditions* than R_j, or
(c) R_i has equal *number of rules* with R_j, and R_i has equal *number of conditions* with R_j, but R_i has fewer *average number of conditions per rule* than R_j.

Table 4 reports on the results of classifier comprehensibility while Fig. 6 presents again the 95%-CIs of differences between AC and the benchmark classifiers on model comprehensibility. It is clear from Table 4 and Fig. 6 that the comprehensibility of CART and RIPPER is notably better than that of the other four classifiers and that the AC classifier results in significantly better comprehensible models than DT and C4.5 classifiers. Finally, the difference of model comprehensibility between AC and ADT classifiers is not significant. When examining the rules of CART and RIPPER

Table 4. Experimental results of model comprehensibility for different classifiers.

	Classifier	CM1	JM1	KC1	KC3	KC4	MC1	MC2	MW1	PC1	PC2	PC3	PC4	AR
No. of rules	AC	9	13	17	13	4	31	9	11	34	19	48	8	4.0
	C4.5	9	71	23	14	4	14	7	**2**	16	11	23	14	3.6
	DT	22	104	65	18	4	31	16	44	41	20	29	32	5.4
	CART	4	7	3	**1**	**3**	4	**2**	3	**2**	3	**1**	**4**	1.5
	ADT	21	21	13	21	21	21	21	21	13	13	13	13	4.4
	RIPPER	**3**	**4**	**4**	4	**3**	**3**	**2**	**2**	4	**2**	4	**4**	**1.3**
No. of Conditions	AC	14	34	38	18	5	69	21	24	103	35	155	14	3.8
	C4.5	32	602	163	73	9	59	27	**2**	101	44	132	73	4.5
	DT	132	312	390	72	8	93	96	704	205	100	116	128	5.5
	CART	9	27	**5**	**0**	5	**9**	2	5	**2**	**0**	**0**	**6**	**1.5**
	ADT	44	64	22	48	46	42	44	38	22	22	20	16	4.0
	RIPPER	**5**	**10**	16	11	**2**	**9**	**1**	3	9	3	10	12	1.6
Average No. of Conditions per rule	AC	**1.56**	2.62	2.24	1.38	1.25	**2.23**	2.33	2.18	3.03	1.84	3.23	1.75	3.0
	C4.5	3.56	8.48	7.09	5.21	2.25	4.21	3.86	**1.00**	6.31	4.00	5.74	5.21	5.3
	DT	6.00	3.00	6.00	4.00	2.00	3.00	6.00	16.00	5.00	5.00	4.00	4.00	5.0
	CART	2.25	3.86	**1.67**	**0.00**	1.67	2.25	1.00	1.67	**1.00**	**0.00**	**0.00**	1.50	**2.3**
	ADT	2.10	3.05	1.69	2.29	2.19	2.00	2.10	1.81	1.69	1.69	1.54	**1.23**	2.8
	RIPPER	1.67	**2.50**	4.00	2.75	**0.67**	3.00	**0.50**	1.50	2.25	1.50	2.50	3.00	2.6

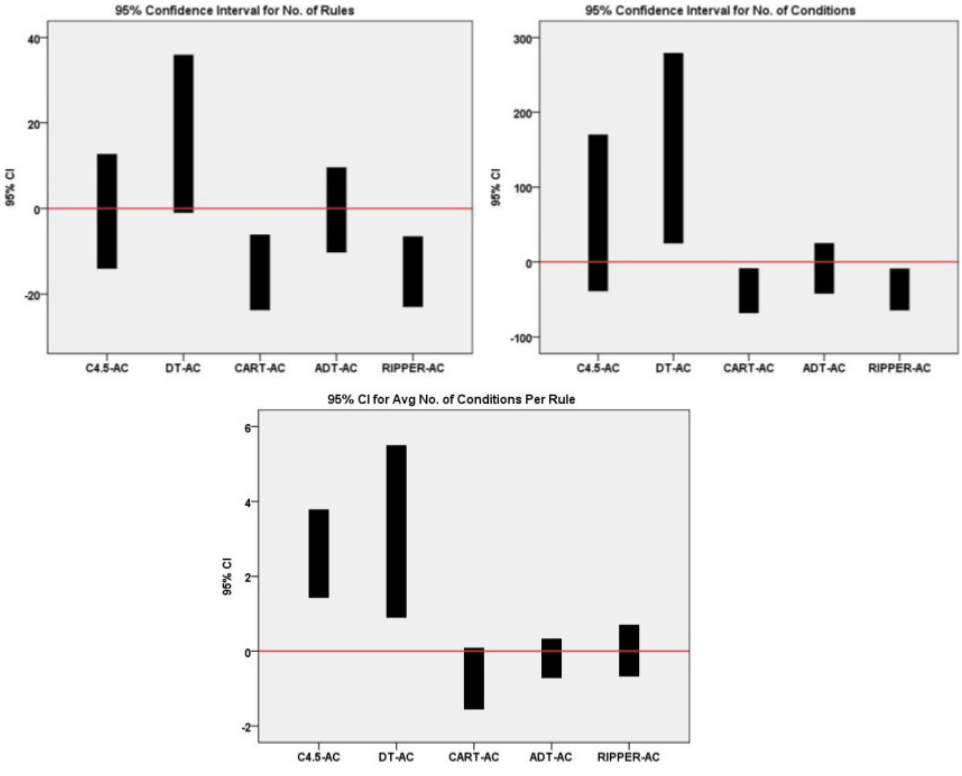


Fig. 6. 95% confidence intervals of differences between AC and benchmark classifiers on comprehensibility.

classifiers, it was found that their rules are usually overly simple and direct. For instance, there is only one rule, i.e., *all modules being nfp* or *default_class*, for CART classifier on KC3 and PC3 data sets, which results in poor performance (AUC = 0.5).

Based on the above discussion, any satisfactory classifier requires the making of a tradeoff between prediction performance and model comprehensibility. We post that the proposed AC method performed adequately in both perspectives. Table 5 summaries the prediction performance and model comprehensibility for the benchmark classifiers compared to the AC-based method. The proposed approach notably outperforms four alternative methods, with DT being the notable exception, while resulting in highly comprehensible models.

Furthermore, we argue that, when comparing tree-based or tree-based classifiers, prediction performance should be the dominant factor to be taken into account, since their comprehensibility is already often satisfactory. As such, comprehensibility could be an additional metric to be considered. From this perspective, the AC method still surpasses other methods. When examining prediction performance, AC and DT show clear advantage over other classifiers while DT results in significantly less comprehensible models.

Table 5. Summary of prediction performance and comprehensibility for five selected classifiers compared to AC-based method.

Classifiers	Prediction performance (AUC)	Comprehensibility
DT	0	−−
CART, RIPPER	−−	++
C4.5	−−	−−
ADT	−−	0

Notes: ++, significantly better than AC; 0, without significant difference; −−, significantly worse than AC.

5.2.3. *Results of cross project validation*

Table 6 presents the performance of the proposed AC-based classifier in terms of the AUC; Table 7 compares these results with those presented above. As indicated in Sec. 5.1.3, only 9 of the 12 data sets are considered here, since other three data sets, i.e. JM1, KC1 and KC4, exhibit a different input space. For both tables, the horizontal dataset names represent the training data while the vertical are the validation sets utilized to obtain the classification performance.

In the context of Tables 6 and 7, we observe that classifiers trained from four data sets, i.e. MC2, MW1, PC3 and PC4, achieve acceptable prediction performance on other data sets with relative AUC values above 90%. In the meanwhile, the stability of prediction effectiveness, which expressed by the standard deviation (denoted as StDev in Table 7), needs also to be taken into consideration. Hence, as can be seen, from a combination point of view of performance and stability, AC classifier trained by PC4 data set possesses a relative high capability of generalization, due to its high mean value and low standard deviation (see Table 7).

5.3. *Threats to validity*

When conducting an empirical study, it is important to be aware of potential threats to the validity of the obtained results and derived conclusions. A first

Table 6. Experimental results of cross project validation on absolute AUC value.

Training / Test	CM1	KC3	MC1	MC2	MW1	PC1	PC2	PC3	PC4
CM1	—	0.709	0.667	0.775	0.735	0.720	0.630	0.759	0.727
KC3	0.627	—	0.644	0.730	0.803	0.646	0.596	0.758	0.658
MC1	0.736	0.745	—	0.718	0.703	0.751	0.560	0.776	0.835
MC2	0.631	0.547	0.604	—	0.654	0.575	0.598	0.665	0.571
MW1	0.651	0.664	0.605	0.727	—	0.590	0.659	0.744	0.650
PC1	0.662	0.654	0.660	0.699	0.644	—	0.607	0.762	0.703
PC2	0.534	0.702	0.761	0.796	0.787	0.561	—	0.522	0.766
PC3	0.620	0.696	0.720	0.695	0.677	0.711	0.587	—	0.746
PC4	0.591	0.701	0.822	0.588	0.589	0.610	0.557	0.720	—

Table 7. Experimental results of cross project validation on relative *AUC* value (%).

Training / Test	CM1	KC3	MC1	MC2	MW1	PC1	PC2	PC3	PC4
CM1	—	118.56	111.52	129.55	122.76	120.41	105.38	126.79	121.51
KC3	89.99	—	92.51	104.82	115.33	92.83	85.63	108.84	94.48
MC1	85.38	86.48	—	83.33	81.61	87.13	64.96	90.04	96.90
MC2	94.02	81.43	90.00	—	97.47	85.58	89.13	99.04	85.04
MW1	75.73	77.26	70.29	84.56	—	68.59	76.65	86.52	75.54
PC1	80.00	79.11	79.84	84.51	77.84	—	73.35	92.14	85.04
PC2	65.96	86.81	94.09	98.35	97.30	69.38	—	64.49	94.73
PC3	75.52	84.71	87.69	84.67	82.47	86.57	71.52	—	90.87
PC4	66.81	79.18	92.86	66.45	66.58	68.95	62.94	81.31	—
Mean	79.18	86.69	89.85	92.03	92.67	84.93	78.69	93.65	**93.02**
StDev	9.55	12.50	11.13	17.70	17.99	16.18	13.18	17.42	**12.59**

possible source of bias relates to the data used, e.g. whether the data is representative to the domain in question and whether results can be generalized. As the data used in this study stems from the public domain, our results can be compared to others, and can be subjected to replication if necessary. In addition, several authors have argued in favor of the appropriateness of the NASA MDP repository and/or used some of its data sets for their experiments [21, 33, 37, 49]. Therefore, we consider the obtained results to be relevant to the software defect prediction community.

Despite the general suitability of the data, the sampling procedure might bias results and prevent generalization. We consider a generic split-sample setup with randomly selected test records (1/3 of the available data set). This is a well-established approach for comparative classification experiments and the size of the MDP data sets seems large enough to justify this setting. Compared to cross validation or bootstrapping, the split sample setup saves a considerable amount of computation time, which, in turn, can be invested into model selection to ensure that the classifiers are well tuned to each data set. It would be interesting to quantify possible differences between a split-sample setup and cross-validation/ bootstrapping setups by means of empirical experimentation. However, this step is left for future research.

The selection of classifiers is another possible source of bias. Given the variety of available tree-based and rule-based learning algorithms, many others could have been considered. Our selection is guided by the aim of finding the most frequently used approaches. Regarding to the proposed AC-based method, we adopted the core idea of CBA and CBA2 algorithms, which contributed the basic point of view for associative classification. However, other AC-based approaches, e.g. CMAR [51], CPAR [52], MMAC [53], CAAR [54], GARC [29], etc. are left as a topic for future investigation. Furthermore, since our experiments are conducted among tree/rule-based classification methods, generalization of our conclusions towards other types of classifiers needs to done cautiously.

6. Conclusions and Future Work

This paper offered several insights into the discipline of software fault prediction. Firstly, we pointed out that software fault prediction models needed to be evaluated from multiple perspectives, not only prediction performance but also comprehensibility. Accuracy may be a misleading performance metric as faulty modules are likely to form the minority of all modules. In addition, other metrics depending on setting a specific threshold on the scores outputted by a classifier can result in arbitrary results being reported, and as such, we advocate to use a different set of evaluation metrics, e.g. AUC and comprehensibility metrics, when comparing classification algorithms.

We also proposed a novel software defect prediction approach. This approach is based on two variants of the associative classification algorithms, i.e. CBA and CBA2 and this is the first attempt to make a connection between associative classification and software fault prediction. It is valuable for real-world applications in software fault prediction as our proposed AC-based method runs efficiently on large data sets and provides high level of predictive power without overfitting. Moreover, it provides classification models with acceptable comprehensibility, which is essential and vital for software managers and users. Implementing effective early warning mechanisms when developing software is believed to decrease costs and enable faster time to market for software developers.

Another important characteristic of this study is that it has compared results from several classification algorithms on 12 public software engineering data sets. The results of our experiments provide a "proving grounds" for new methods that will be developed in the future. All our experiments can be repeated, as the tools and data sets of this study are situated in the public domain. Therefore, new methods for software fault prediction can and should be compared with our results in a clear and consistent way.

Finally, in order to evaluate the software fault prediction results in a combined perspective of classification performance and comprehensibility, we have compared the AC-based method with five other rule/tree-based classification algorithms. Nevertheless, considering other complicated classification methods, such as support vector machines, neural networks, Bayesian classifications, etc. in the comparative experiments would gain important insights for the domain of software fault prediction, which is left as the future research for us.

Acknowledgements

This work was partly supported by the National Natural Science Foundation of China (61272362/71372044/71110107027), the Fundamental Research Funds for the Central Universities (No. 2014RC0601) and the Doctoral Fund of Ministry of Education of China (No. 20120005120001). We also give thanks to the anonymous reviewers for their thoughtful comments and suggestions.

References

1. C. Nan-Hsing, Combining techniques for software quality classification: An integrated decision network approach, *Expert Systems with Applications* **38** (2011) 4618–4625.
2. R. Bell, T. Ostrand and E. Weyuker, The limited impact of individual developer data on software defect prediction, *Empirical Software Engineering* **18** (2013) 478–505.
3. A. G. Koru and H. Liu, Building effective defect-prediction models in practice, *IEEE Software* **22** (2005) 23–29.
4. N. F. Schneidewind, Methodology for validating software metrics, *IEEE Transactions on Software Engineering* **18** (1992) 410–422.
5. L. C. Briand, V. R. Brasili and C. J. Hetmanski, Developing interpretable models with optimized set reduction for identifying high-risk software components, *IEEE Transactions on Software Engineering* **19** (1993) 1028–1044.
6. N. F. Schneidewind, Investigation of logistic regression as a discriminant of software quality, in *Seventh International Symposium on Software Metrics*, Washington, DC, 2001, pp. 328–337.
7. G. Denaro, M. Pezzè and S. Morasca, Towards industrially relevant fault-proneness models, *Int. J. Software Engineering & Knowledge Engineering* **13** (2003) 395–414.
8. T. Khoshgoftaar, N. Seliya and K. Gao, Assessment of a new three-group software quality classification technique: An empirical case study, *Empirical Software Engineering* **10** (2005) 183–218.
9. T. Menzies, J. DiStefano, A. Orrego and R. Chapman, Assessing predictors of software defects, in *Predictive Software Models Workshop*, 2004, pp. 1–4.
10. B. Turhan and A. Bener, Analysis of naive Bayes' assumptions on software fault data: An empirical study, *Data & Knowledge Engineering* **68** (2009) 278–290.
11. E. Arisholm, L. C. Briand and 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.
12. F. Xing, P. Guo and M. R. Lyu, A novel method for early software quality prediction based on support vector machine, in *Sixteenth IEEE International Symposium on Software Reliability Engineering*, Chicago, IL, USA, 2005, pp. 213–222.
13. S. Kanmani, V. R. Uthariaraj, V. Sankaranarayanan and P. Thambidurai, Object oriented software quality prediction using general regression neural networks, *SIGSOFT Softw. Eng. Notes* **29** (2004) 1–6.
14. Z. Jun, Cost-sensitive boosting neural networks for software defect prediction, *Expert Systems with Applications* **37** (2010) 4537–4543.
15. S. Zhong, T. M. Khoshgoftaar and N. Seliya, Unsupervised learning for expert-based software quality estimation, in *Eighth IEEE International Symposium on High Assurance Systems Engineering*, 2004, pp. 149–155.
16. K. Ganesan, T. M. Khoshgoftaar and E. B. Allen, Case-based software quality prediction, *International Journal of Software Engineering & Knowledge Engineering* **10** (2000) 139–152.
17. C. Catal and 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.
18. C. Cagatay, Software fault prediction: A literature review and current trends, *Expert Systems with Applications* **38** (2011) 4626–4636.
19. C. Catal and B. Diri, A systematic review of software fault prediction studies, *Expert Systems with Applications* **36** (2009) 7346–7354.

20. M. A. Almeida and S. Matwin, Machine learning method for software quality model building, in *Eleventh International Symposium on Methodologies for Intelligent Systems*, 1999, pp. 565–573.
21. S. Lessmann, B. Baesens, C. Mues and S. Pietsch, Benchmarking classification models for software defect prediction: A proposed framework and novel findings, *IEEE Transactions on Software Engineering* **34** (2008) 485–496.
22. J. R. Quinlan, *C4.5: Programs for Machine Learning* (Morgan Kaufmann, 1993).
23. W. W. Cohen, Fast effective rule induction, in *Proceedings of the 12th International Conference on Machine Learning* (Lake Tahoe, CA, 1995), pp. 115–123.
24. L. Breiman, J. H. Friedman, R. A. Olshen and C. J. Stone, *Classification and Regression Trees* (Wadsworth International Group, Belmont, 1984).
25. Y. Freund and L. Mason, The alternating decision tree learning algorithm, in *Proceedings of the 16th International Conference on Machine Learning* (Bled, Slovenia, 1999), pp. 124–133.
26. F. Thabtah, A review of associative classification mining, *Knowledge Engineering Review* **22** (2007) 37–65.
27. B. Liu, W. Hsu and Y. Ma, Integrating classification and association rule mining, in *Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining* (New York, 1998), pp. 80–86.
28. B. Liu, Y. Ma and C. Wong, Improving an association rule based classifier, in *Proceedings of the 4th European Conference on Principles and Practice of Knowledge Discovery in Databases*, 2000, pp. 504–509.
29. G. Chen, H. Liu, L. Yu, Q. Wei and X. Zhang, A new approach to classification based on association rule mining, *Decision Support Systems* **42** (2006) 674–689.
30. J. S. Shirabad and T. J. Menzies, The PROMISE Repository of Software Engineering Databases, 2005, from http://promise.site.uottawa.ca/SERepository.
31. M. Chapman, P. Callis and W. Jackson, Metrics Data Program, 2004, from http://mdp.ivv.nasa.gov/.
32. B. Cukic and Y. Ma, Predicting fault-proneness: Do we finally know how? in *Reliability Analysis of System Failure Data*, Cambridge, UK, 2007.
33. T. Menzies, J. Greenwald and A. Frank, Data mining static code attributes to learn defect predictors, *IEEE Transactions on Software Engineering* **33** (2007) 2–13.
34. Y. Jiang, B. Cukic and T. Menzies, Fault prediction using early lifecycle data, in *18th IEEE International Symposium on Software Reliability*, 2007, pp. 237–246.
35. S. Shafi, S. M. Hassan, A. Arshaq, M. J. Khan and S. Shamail, Software quality prediction techniques: A comparative analysis, in *4th International Conference on Emerging Technologies*, 2008, pp. 242–246.
36. J. Riquelme, R. Ruiz, D. Rodríguez and J. Moreno, Finding defective modules from highly unbalanced datasets, in *Actas del 8 taller sobre el apoyo a la decisión en ingeniería del software*, 2008, pp. 67–74.
37. O. Vandecruys, D. Martens, B. Baesens, C. Mues, M. De Backer and R. Haesen, Mining software repositories for comprehensible software fault prediction models, *Journal of Systems and Software* **81** (2008) 823–839.
38. K. Dejaeger, T. Verbraken and B. Baesens, Towards comprehensible software fault prediction models using Bayesian network classifiers, *IEEE Transactions on Software Engineering* **39** (2013) 237–257.
39. Y. Jiang, J. Shang and Y. Liu, Maximizing customer satisfaction through an online recommendation system: A novel associative classification model, *Decision Support Systems* **48** (2010) 470–479.

40. T. Menzies and M. Shepperd, Special issue on repeatable results in software engineering prediction, *Empirical Software Engineering* **17** (2012) 1–17.
41. T. Gyimothy, R. Ferenc and I. Siket, Empirical validation of object-oriented metrics on open source software for fault prediction, *IEEE Transactions on Software Engineering* **31** (2005) 897–910.
42. T. J. Ostrand, E. J. Weyuker and R. M. Bell, Predicting the location and number of faults in large software systems, *IEEE Transactions on Software Engineering* **31** (2005) 340–355.
43. B. W. Boehm and P. N. Papaccio, Understanding and controlling software costs, *IEEE Transactions on Software Engineering* **14** (1988) 1462–1477.
44. S. G. Swapna, Architecture-based software reliability analysis: Overview and limitations, *IEEE Transactions on Dependable and Secure Computing* **4** (2007) 32–40.
45. M. J. Harrold, Testing: A roadmap, in *Proceedings of the Conference on the Future of Software Engineering* (Limerick, Ireland, 2000), pp. 61–72.
46. A. Bhattacharya, A. Konar, S. Das, C. Grosan and A. Abraham, Hardware software partitioning problem in embedded system design using particle swarm optimization algorithm, in *International Conference on Complex, Intelligent and Software Intensive Systems*, 2008, pp. 171–176.
47. L. Chen and S. Huang, Accuracy and efficiency comparisons of single- and multi-cycled software classification models, *Information and Software Technology* **51** (2009) 173–181.
48. I. Myrtveit, E. Stensrud and M. Shepperd, Reliability and validity in comparative studies of software prediction models, *IEEE Transactions on Software Engineering* **31** (2005) 380–391.
49. A. G. Koru and H. Liu, An investigation of the effect of module size on defect prediction using static measures, *ACM SIGSOFT Software Engineering Notes* **30** (2005) 1–5.
50. R. Agrawal and R. Srikant, Fast algorithms for mining association rules, in *Proceeding of the 20th International Conference on Very Large Data Bases (VLDB'94)*, 1994, pp. 487–499.
51. W. Li, J. Han and J. Pei, CMAR: Accurate and efficient classification based on multiple class-association rules, in *Proceedings of the International Conference on Data Mining* (San Jose, CA, 2001), pp. 369–376.
52. X. Yin and J. Han, CPAR: Classification based on predictive association rules, in *Proceedings of the SIAM International Conference on Data Mining* (San Francisco, CA, 2003), pp. 369–376.
53. F. A. Thabtah, P. Cowling and P. Yonghong, MMAC: A new multi-class, multi-label associative classification approach, in *4th IEEE International Conference on Data Mining* (Brighton, UK, 2004), pp. 217–224.
54. X. Xiaoyuan, H. Guoqiang and M. Huaqing, A novel algorithm for associative classification of image blocks, in *4th International Conference on Computer and Information Technology* (Shiguo, China, 2004), pp. 46–51.
55. Y. Yoon and G. G. Lee, Efficient implementation of associative classifiers for document classification, *Information Processing & Management* **43** (2007) 393–405.
56. J. Pinho Lucas, S. Segrera and M. N. Moreno, Making use of associative classifiers in order to alleviate typical drawbacks in recommender systems, *Expert Systems with Applications* **39** (2012) 1273–1283.
57. J. Dougherty, R. Kohavi and M. Sahami, Supervised and unsupervised discretization of continuous features, in *Proceedings of the 12th International Conference on Machine Learning* (San Francisco, CA, 1995), pp. 194–202.
58. R. Kohavi, A study of cross-validation and bootstrap for accuracy estimation and model selection, in *International Joint Conference on Artificial Intelligence*, 1995, pp. 1137–1145.

59. U. M. Fayyad and K. B. Irani, Multi-interval discretization of continuous-valued attributes for classification learning, in *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, 1993, pp. 1022–1027.
60. F. Provost, T. Fawcett, Robust classification for imprecise environments, *Machine Learning* **42** (2001) 203–231.
61. F. Tom, An introduction to ROC analysis, *Pattern Recognition Letters* **27** (2006) 861–874.
62. A. Mahaweerawat, P. Sophatsathit and C. Lursinsap, Adaptive self-organizing map clustering for software fault prediction, in *4th International Joint Conference on Computer Science and Software Engineering* (Khon Kaen, Thailand, 2007), pp. 35–41.
63. T. Menzies, A. Dekhtyar, J. Distefano and 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.
64. M. Kubat, R. C. Holte and S. Matwin, Machine learning for the detection of oil spills in satellite radar images, *Machine Learning* **30** (1998) 195–215.
65. B. Andrew, The use of area under the ROC curve in the evaluation of machine learning algorithms, *Pattern Recognition* **30** (1997) 1145–1159.
66. Y. Ma, L. Guo and B. Cukic, A statistical framework for the prediction of fault-proneness, in *Advances in Machine Learning Application in Software Engineering* (Idea Group, 2006), pp. 237–265.
67. T. Hall, S. Beecham, D. Bowes, D. Gray and S. Counsell, A systematic literature review on fault prediction performance in software engineering, *IEEE Transactions on Software Engineering* **38** (2012) 1276–1304.
68. D. Martens and B. Baesens, Building acceptable classification models, in R. Stahlbock, S. F. Crone and S. Lessmann (Eds.) *Data Mining: Special Issue in Annals of Information Systems* (Springer, 2010), pp. 53–74.
69. I. Askira-Gelman, Knowledge discovery: Comprehensibility of the results, in *Proceedings of the 31st Hawaii International Conference on System Sciences*, Washington, DC, USA, 1998, pp. 247–255.
70. J. Huysmans, K. Dejaeger, C. Mues, J. Vanthienen and B. Baesens, An empirical evaluation of the comprehensibility of decision table, tree and rule based predictive models, *Decision Support Systems* **51** (2011) 141–154.
71. F. Coenen, LUCS KDD implementation of CBA (Classification Based on Associations), 2004, from http://www.csc.liv.ac.uk/∼frans/KDD/Software/CMAR/cba.html.
72. C. Andersson, A replicated empirical study of a selection method for software reliability growth models, *Empirical Software Engineering* **12** (2007) 161–182.
73. K. El Emam, S. Benlarbi, N. Goel and S. N. Rai, Comparing case-based reasoning classifiers for predicting high risk software components, *Journal of Systems and Software* **55** (2001) 301–320.
74. K. El Emam, W. Melo and J. C. Machado, The prediction of faulty classes using object-oriented design metrics, *Journal of Systems and Software* **56** (2001) 63–75.
75. R. Kohavi, The power of decision tables, in *8th European Conference on Machine Learning*, 1995, pp. 174–189.
76. I. H. Witten, E. Frank and M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*, *3rd Ed.* (Morgan Kaufmann, 2011).