Toward Comprehensible Software Fault Prediction Models Using Bayesian Network Classifiers

Karel Dejaeger, Thomas Verbraken, and Bart Baesens

Abstract—Software testing is a crucial activity during software development and fault prediction models assist practitioners herein by providing an upfront identification of faulty software code by drawing upon the machine learning literature. While especially the Naive Bayes classifier is often applied in this regard, citing predictive performance and comprehensibility as its major strengths, a number of alternative Bayesian algorithms that boost the possibility of constructing simpler networks with fewer nodes and arcs remain unexplored. This study contributes to the literature by considering 15 different Bayesian Network (BN) classifiers and comparing them to other popular machine learning techniques. Furthermore, the applicability of the Markov blanket principle for feature selection, which is a natural extension to BN theory, is investigated. The results, both in terms of the AUC and the recently introduced H-measure, are rigorously tested using the statistical framework of Demšar. It is concluded that simple and comprehensible networks with less nodes can be constructed using BN classifiers other than the Naive Bayes classifier. Furthermore, it is found that the aspects of comprehensibility and predictive performance need to be balanced out, and also the development context is an item which should be taken into account during model selection.

Index Terms—Software fault prediction, Bayesian networks, classification, comprehensibility

1 Introduction

The ubiquitous[1] presence of computers has given rise to novel research fields such as software development, computer engineering, and artificial intelligence [25] while at the same time enabling novel developments in other domains like medicine, telecommunications, and image processing [41], [101], [102]. In spite of all the efforts invested in the field of software engineering, the development of software remains jeopardized by high cancellation rates and considerable delays [59]. The introduction of software testing processes to identify software faults in a timely manner is crucial since corrective maintenance costs increase exponentially if faults are detected later in the software development life cycle [11]. As a result, the importance of software testing has long been recognized, e.g., the waterfall approach, a phased and iterative development methodology, specifies the implementation of a separate testing phase [82]. The first work on the topic of software testing dates from 1975 [44] and, since the pioneering work of Goodenough, numerous books and papers have been published on this topic. Software testing expenses can amount to up to 60 percent of the overall development budget [50], and several approaches to support these efforts have been proposed.

A key finding to software testing is the fact that faults tend to cluster, i.e., to be contained in a limited number of software modules [87]. Gyimóthy et al. found while investigating the open source software web and e-mail suite Mozilla that bugs were present in 42.04 percent of all software classes [46]. Even more skewed distributions have been reported by others, e.g., Ostrand et al., who investigated several successive releases of a large inventory system, stated that "At each release after the first, faults occurred in 20 percent or fewer of the files" [77]. In fact, it has been shown that the distribution of faults over a system can be modeled by a Weibull probability distribution [107]. This motivates the use of software fault prediction models, which provide an upfront indication of whether code is likely to contain faults, i.e., is fault prone. A timely identification of this fault prone code will allow for a more efficient allocation of testing resources and an improved overall software quality. To construct such a prediction model which discriminates between fault-prone code segments and those presumed to be fault-free, the use of static code features characterizing code segments has been advocated [14], [16]. Static code features which can be automatically collected from software source code have proven to be useful [88], and are widely used in academic research as well as in industry settings [74], [99].

A myriad of different approaches to assist in the fault prediction task have previously been proposed, including expert driven methods, statistical models, and machine learning techniques [16]. In spite of the use of various

1. Data collected from different application domains by the ISBSG (International Software Benchmarking Standards Group) indicates that a waterfall-like approach remains commonly used in modern software development, www.ISBSG.org.

- *K. Dejaeger and T. Verbraken are with the Department of Decision Sciences and Information Management, Faculty of Business and Economics, Katholieke Universiteit Leuven, Naamsestraat 69, B-3000 Leuven, Belgium. E-mail: {Karel.Dejaeger, Thomas.Verbraken}@econ.kuleuven.be.*
- *B. Baesens is with the Department of Decision Sciences and Information Management, Faculty of Business and Economics, Katholieke Universiteit Leuven, Naamsestraat 69, B-3000 Leuven, Belgium and the School of Management, University of Southampton, Highfield Southampton, SO17 1BJ, United Kingdom. E-mail: Bart.Baesens@econ.kuleuven.be.*

Manuscript received 16 May 2011; revised 2 Feb. 2012; accepted 16 Feb. 2012; published online 20 Mar. 2012.
Recommended for acceptance by B. Littlewood.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2011-05-0150.
Digital Object Identifier no. 10.1109/TSE.2012.20.

advanced techniques, including association rule mining [9], support vector machines [31], neural networks [80], genetic programming [32], and swarm intelligence [100], it is recognized that their gain compared to simple techniques such as Naive Bayes is limited [74]. The use of Naive Bayes to model the presence of software faults is also advocated by other researchers, citing predictive performance and comprehensibility as its major strengths [17], [36], [97]. Underlying Naive Bayes is the assumption of conditional independency between attributes. Despite the restrictions on the network structure imposed by this conditional independence assumption, Naive Bayes classifiers have been found to perform surprisingly well in, e.g., the medical domain [91]. A similar conclusion was also echoed by Holte, who compared the complexity and accuracy of different rule learners [52]. He noted that simple models are often not outperformed by more complex ones and that in such cases, the simpler model should be selected. The good performance of Naive Bayes compared to other classifiers inspired several modifications relaxing the conditional independence assumption to allow the construction of more complex network structures. Well-known examples include Augmented Naive Bayes and General Bayesian Network (BN) classifiers. The latter even impose no restrictions on the network structure and various algorithms to learn a suitable network structure have been introduced. Two such algorithms, together with various Augmented Naive Bayes classifiers and a selection of benchmark classifiers, constitute the set of techniques under consideration in this study. Additionally, the use of the Markov blanket feature selection procedure, which provides a natural way to reduce the set of available features, is also investigated. The results of the analyses are presented both in terms of AUC (Area Under the ROC Curve) and the novel H-measure and are subjected to rigorous statistical testing to verify their significance.

The rest of the paper is structured as follows: In the next part, our work is positioned against the software fault prediction literature. Section 3 discusses the working of Naive Bayes classifiers and provides a number of extensions hereon. The Markov blanket feature selection procedure is also explained. In Section 4, the empirical setup of the study is detailed, including the rationale for using the novel H-measure. Next, the results are presented in Section 5, together with a discussion on the suitability of extending the Naive Bayes classifier in a software fault prediction context from a comprehensibility point of view. Finally, a short conclusion is provided.

2 Related Work

Software failure is being studied from various viewpoints, for instance, stochastic models to estimate the postdeployment software reliability, expressed, e.g., in terms of the probability of failure each time a software component is executed, is a topic which has attracted considerable attention [42]. Such models typically combine information on the interplay between different components with individual component reliability data. Furthermore, reliability growth models can be applied to estimate the reliability of individual components over time, which can be aggregated into an estimate of the overall reliability of a system. One of the main purposes of these models is to assist in software maintenance budgeting [42].

Some researchers have adopted an alternative viewpoint stemming from the observation that costs incurred to correct faults increase exponentially with the time they remain uncorrected in the system. It is therefore advisable to eliminate as many faults as early as possible during software development [11]. Based hereon, one can also focus on already locating faults during development, before the software goes gold. An example is the development of automatic bug localization techniques that try to establish which software patterns are associated with bugs (referred to as "bug patterns") and subsequently use this information to locate previously unknown bugs in the source code [24]. Another approach to the early identification of faults is *software fault prediction*, which investigates the characteristics of individual code segments to identify those segments that are fault prone [74] or to predict the number of faults in each segment [78]. In the first, software fault prediction is regarded as a classification problem, while the latter approach considers it to be a regression problem. Note that in this study, emphasis is put on the classification point of view. To this purpose, a large number of software code characteristics (also referred to as "static code features") have been introduced to the domain of software fault prediction. These include McCabe and Halstead metrics, metrics adopting object-oriented (OO) programming concepts such as the Chidamber-Kemerer (CK) metrics suite [21] or the Conceptual Cohesion of Classes (C3) measure introduced in [73], as well as various file and component-based metrics [72], [77]. Note that some of these metrics can be collected on different granularity levels, e.g., McCabe and Halstead measures have been explored on the level of software functions, classes, and files. Fenton and Neil [35] conjectured that the most widely used static code features include Lines Of Code (LOC)-based measures, Halstead metrics, and McCabe complexity metrics, which was also echoed by Catal and Diri [16]. Evidence hereon can be found in the publicly available software fault prediction data, e.g., all projects in the NASA MDP repository contain these metrics at the level of software modules and several datasets from other sources also include these metrics [63], [98], [99], [108].

There has been considerable debate about the extent to which software fault prediction models constructed from these metrics actually contribute to supporting software testing processes. It is, e.g., demonstrated by Fenton and Pfleeger that by using different language constructs, source code providing the same functionality can have different static code values [37]. Furthermore, while several studies failed to validate the usefulness of, e.g., McCabe cyclomatic complexity metrics for software fault prediction [85], [86], other studies showed opposite results [16].

More recently, the validity of software fault prediction using static code features has been empirically illustrated by, e.g., Menzies et al., who stated that static code features are *useful*, *easy to use*, and *widely used* [74]. This observation was later also confirmed by other work, e.g., [99].

Useful. Several studies have reported on the inability of real-life fault predictors to obtain similar detection rates as static code-based classification techniques.

- A panel consisting of academic and business experts at the IEEE Metrics 2002 symposium concluded that manual software reviews typically account for around 60 percent of all identified faults, independent of the domain or level of maturity of the organization [88]. Similar (or even worse) defect detection capabilities were observed among other industrial defect detectors [74].
- Empirical evidence comparing an expert driven approach with the use of statistical techniques to locate software faults indicated the superior performance of the latter, stating that "When it comes to comparing both methods we found that statistical models outperformed expert estimations" [92]. Other advantages of adopting static code-based classifiers that were identified include improved fault prediction efficiency and the ability to cope with large datasets, resulting in possibly finer grained fault prediction. The study also found human experts to be unable to grasp or understand the structure of large systems and, as a result, unable to, e.g., provide a ranking of the fault proneness across *all* system components.

  On the flip side, it was reckoned that human experts might be better able to incorporate qualitative information when predicting the fault proneness of a software component; however, this advantage proved not to outweigh the disadvantages.

By contrast, using static code-based classification techniques, noticeably better detection rates have been recorded. For instance, Menzies et al. reported an average detection rate of 71 percent [74].

Easy to use. Static code features such as McCabe and Halstead metrics can be mined from the source code using automated methods. Several tools have been proposed, including McCabe IQ,[2] RUBY [17], EMERALD [53], and Prest [64], to assist practitioners in this effort. In addition to static code features characterizing each code segment, labels indicating whether faults were found are needed to construct software fault prediction models. This often requires a matching between data contained in a bug database such as Bugzilla[3] and the mined source code. Various text mining techniques exist to facilitate this matching effort [38].

Widely used. Static code features have been extensively investigated by researchers [14], [16] and their use in industry has been long reckoned, e.g., [35]. It is argued that some large government software contractors will not review code segments unless they are flagged as fault prone [74]. Moreover, the ability to collect data concerning the software development process is also a requirement when trying to achieve Capability Maturity Model Integration (CMMI) level 2 appraisal. Obtaining such appraisal can be an obligation to compete for (government) contracts [10].

2. www.mccabe.com.
3. www.bugzilla.org.

Researchers have adopted a myriad of different techniques to construct software fault prediction models. These include various statistical techniques such as logistic regression and Naive Bayes which explicitly construct an underlying probability model. Furthermore, different machine learning techniques such as decision trees, models based on the notion of perceptrons, support vector machines, and techniques that do not explicitly construct a prediction model but instead look at a set of most similar known cases have also been investigated. A taxonomy of the most often employed classification techniques for software fault prediction is offered in Fig. 1. References to earlier work using each technique are provided between square brackets. While this overview does not attempt to be exhaustive, it is clear that BN classifiers are in fact one of the most popular techniques to model the presence of software faults in a system. One of the earliest references to BN classifiers in this context can be found in the work of Fenton and Neil, who reckoned that these techniques offer several advantages including the ability to explicitly model uncertainty, the good comprehensibility, and the avoidance of multicollinearity related problems [35]. They also highlighted the possibility of expert driven BN creation, which, however, deviates from how such classifiers are commonly utilized in machine learning literature. Persuaded by these remarks, several authors have used such models [70], [74], [97], [99]. The Naive Bayes classifier has been especially carefully investigated and has been found to perform exceptionally well, despite being a very simple technique. For instance, Menzies et al. found Naive Bayes with logarithmic transformation of the inputs to be the best performing prediction model as compared to two rule induction techniques, i.e., C4.5 and OneR [74]. This result was later partially confirmed by Lessmann et al., who found an ensemble learner, Random Forests (RndFor), to be the best performing technique. BN learners, however, were not found to be statistically outperformed by this ensemble learner [70]. The assumption of conditional independence underlying Naive Bayes is typically not met in a software fault prediction context; different static code features try to measure the same underlying dimensions of the source code. The relaxation of this assumption has been investigated by Turhan et al., who instead used a univariate Gaussian approximation of the unknown distribution of the static code features, a multivariate Gaussian approximation [97]. It was concluded that the independence assumption of Naive Bayes is not harmful for software defect data after the data were preprocessed using Principal Component Analysis (PCA), which maps the data on a set of orthogonal axes. Note that by construction, principal components are not correlated with each other. In our work, the impact of the conditional independence assumption is considered from a different perspective by investigating BN classifiers that explicitly model the conditional independence among attributes. In another study, Turhan and Bener looked into the use of various attribute weighting heuristics (e.g., heuristics based on concepts of Shannon's information theory and statistical methods such as the PCA scores and the Kullback-Leibler Divergence) to improve BN learners [96], [97]. As such, they adopted a two stage approach by first applying these heuristics to rank all attributes and afterward providing this ranking to the BN learner. They

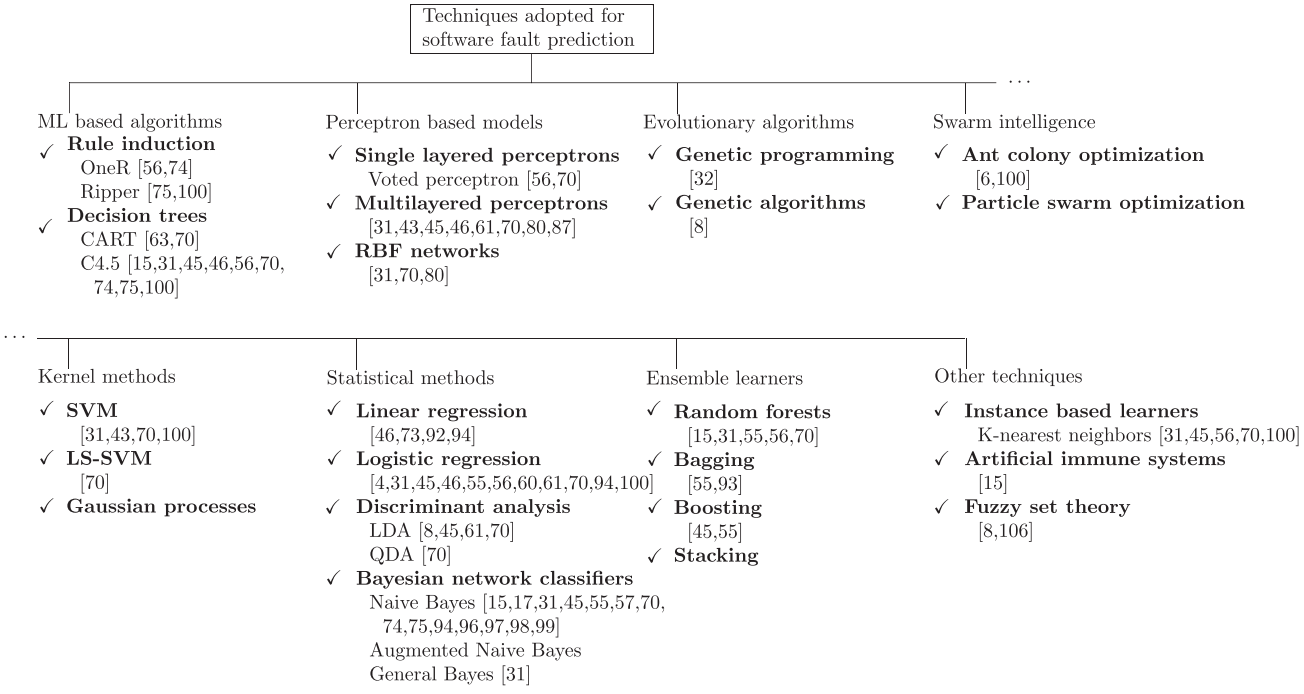


Fig. 1. Supervised classification taxonomy for software fault prediction.

showed that using weighting heuristics based on Shannon's information theory (information gain and gain ratio) or feature selection techniques yields improved results. Their findings motivated the inclusion of the Markov blanket feature selection into this study, which is a feature selection approach rooted in BN theory.

3 Bayesian Network Classifiers

In this section, a general introduction to Bayesian networks is presented, followed by a description of the Naive Bayes classifier. Next, a number of alternative BN classifiers relaxing the assumption of conditional independence are explained. Two alternative machine learning techniques, which serve as a benchmark in this study, are also detailed.

The following notation is adopted throughout the paper:

j	attribute identifier
i	instance identifier
n	number of attributes
N	number of instances
$x_{i(j)} \in \mathbb{R}$	scalar representing the value of the j^{th} attribute on the i^{th} instance
y_i	dichotomous target indicating an instance is faulty ($y_i = 1$) or not ($y_i = 0$)

In line with this notation, the task of learning a software fault prediction model can be defined as follows: Let $\mathrm{D} =_{trn} \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$ be a training set containing N observations, where $\mathbf{x}_i \in \mathbb{R}^n$ represents the static code features characterizing the ith instance and $y_i \in \{0, 1\}$, a label indicating the presence of faults. A software fault prediction model provides a mapping from the instances $\mathbf{x}_i$ to the posterior probability of belonging to the class of fault prone code segments, $P(y_i = 1|\mathbf{x}_i)$. Formally, $f(\mathbf{x}_i) : \mathbb{R}^n \mapsto P(y_i = 1|\mathbf{x}_i)$.

3.1 Bayesian Networks

A BN represents a joint probability distribution over a set of stochastic variables, either discrete or continuous. It can be visualized as a graph consisting of nodes representing the individual variables $x_{(j)}$ and directed arcs indicating the existence of dependencies between variables. Likewise, the absence of an arc between two nodes $x_{(j)}$ and $x_{(j')}$ indicates the conditional independence between both variables given their parents in the graph. Associated with each node is a probability table containing the probability distribution of each variable conditional on the direct parent(s) in the graph [79]. Underlying BN is the Bayes theorem, which formulates the posterior probability of the presence of faults in terms of prior probabilities and the reverse conditional probability:

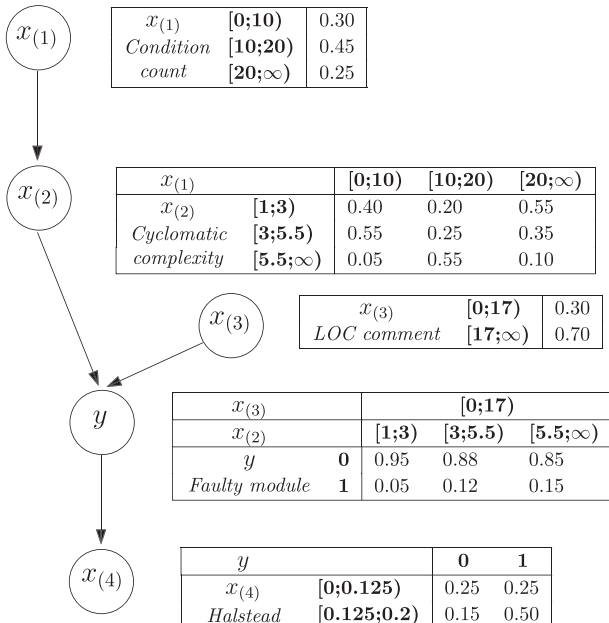
$$P(y_i = 1|\mathbf{x}_i) = \frac{P(\mathbf{x}_i|y_i = 1)P(y_i = 1)}{P(\mathbf{x}_i)}. \quad (1)$$

Note that $P(\mathbf{x}_i)$ acts as a normalizing constant herein and can be ignored.

More formally, a BN comprises two parts $B = \langle G, \Theta \rangle$. G is a directed acyclic graph (DAG) conveying the direct dependence relationships within the dataset, whereas the second part, Θ, represents the conditional probability distribution of each variable. Adopting the notation of Cooper and Herskovits [23], $\Pi_{x_{(j)}}$ represents the set of direct parents of $x_{(j)}$ in G. Θ contains a parameter $\theta_{x_{(j)}|\Pi_{x_{(j)}}} = P_B(x_{(j)}|\Pi_{x_{(j)}})$ for each possible value of $x_{(j)}$, given each possible combination of values of all direct parents. The network B then represents the following joint probability distribution:

$$P_B(x_{(1)}, \ldots, x_{(n)}) = \prod_{j=1}^{n} P_B(x_{(j)}|\Pi_{x_{(j)}}) = \prod_{j=1}^{n} \theta_{x_{(j)}|\Pi_{x_{(j)}}}. \quad (2)$$

Typically, the task of learning a BN can be decomposed into two subtasks which are executed sequentially. First, the



$x_{(1)}$		
Condition count	**[0;10)**	0.30
	[10;20)	0.45
	[20;∞)	0.25

$x_{(1)}$		**[0;10)**	**[10;20)**	**[20;∞)**
$x_{(2)}$	**[1;3)**	0.40	0.20	0.55
Cyclomatic complexity	**[3;5.5)**	0.55	0.25	0.35
	[5.5;∞)	0.05	0.55	0.10

$x_{(3)}$		
LOC comment	**[0;17)**	0.30
	[17;∞)	0.70

$x_{(3)}$		**[0;17)**			**[17;∞)**		
$x_{(2)}$		**[1;3)**	**[3;5.5)**	**[5.5;∞)**	**[1;3)**	**[3;5.5)**	**[5.5;∞)**
y	**0**	0.95	0.88	0.85	0.90	0.87	0.91
Faulty module	**1**	0.05	0.12	0.15	0.10	0.13	0.09

y		**0**	**1**
$x_{(4)}$	**[0;0.125)**	0.25	0.25
Halstead error estimate	**[0.125;0.2)**	0.15	0.50
	[0.2;0.285)	0.10	0.05
	[0.285;∞)	0.50	0.20

Classification example

Condition count, $x_{(1)} \in [20;\infty)$
Cyclomatic complexity, $x_{(2)} \in [3;5.5)$
LOC comment, $x_{(3)} \in [17;\infty)$
Halstead error estimate, $x_{(4)} \in [0.2;0.285)$

Joint probability (Eq. 2)

$$P\left(y=0, x_{(1)} \in [20;\infty), x_{(2)} \in [3;5.5), x_{(3)} \in [17;\infty), x_{(4)} \in [0.2;0.285)\right) = 0.87 \cdot 0.25 \cdot 0.35 \cdot 0.70 \cdot 0.10 = 0.00533$$

$$P\left(y=1, x_{(1)} \in [20;\infty), x_{(2)} \in [3;5.5), x_{(3)} \in [17;\infty), x_{(4)} \in [0.2;0.285)\right) = 0.13 \cdot 0.25 \cdot 0.35 \cdot 0.70 \cdot 0.05 = 0.00040$$

Conditional probability

$$P\left(y=0 | x_{(1)} \in [20;\infty), x_{(2)} \in [3;5.5), x_{(3)} \in [17;\infty), x_{(4)} \in [0.2;0.285)\right) = \frac{0.00533}{0.00533+0.00040} = 0.93$$

$$P\left(y=1 | x_{(1)} \in [20;\infty), x_{(2)} \in [3;5.5), x_{(3)} \in [17;\infty), x_{(4)} \in [0.2;0.285)\right) = \frac{0.00040}{0.00533+0.00040} = 0.07$$

Fig. 2. Bayesian network classification by example.

exact structure G of the network needs to be determined. In general, it is infeasible to iterate over all possible network structures and therefore several constraints can be imposed, leading to different learning algorithms. After establishing the exact network structure G, the parameters associated with each node need to be estimated. In this paper, the empirical frequencies as observed in the training data D_{trn} are used to estimate these parameters:

$$\theta_{x_{(j)}|\Pi_{x_{(j)}}} = \hat{P}_{D_{trn}}(x_{(j)}|\Pi_{x_{(j)}}). \tag{3}$$

It can be shown that these estimates maximize the log likelihood of the network B given the training data D_{trn}. Note that these estimates might be further improved by a smoothing operation, e.g., by using a Laplace correction or an M-estimate [40].

Generally, BN classifiers can be considered as probabilistic white-box classifiers. They allow to calculate the (joint) posterior probability distribution of any subset of unobserved stochastic variables, given that the variables in the complementary subset are observed. This functionality enables the use of BN as statistical classifiers which provide a final classification by selecting an appropriate threshold on the posterior probability distribution of the (unobserved) class node. Alternatively, assuming all misclassification costs are equal, a winner-takes-all rule can be adopted [29]. A pivotal ability of these classifiers is the use of graphical artifacts which facilitates the understanding of complex and seemingly contradictory relationships within the data [35].

A simple example of a BN classifier is given in the left-hand side of Fig. 2, while on the right-hand side an example classifying a specific code segment as (not) fault prone is provided. Note that by considering the characteristics of this segment and the information conveyed in the Bayesian network, the posterior probability that this particular instance will be faulty can be computed as follows:

$$P(y|x_{(1)}, x_{(2)}, x_{(3)}, x_{(4)}) = \frac{P(y, x_{(1)}, x_{(2)}, x_{(3)}, x_{(4)})}{P(x_{(1)}, x_{(2)}, x_{(3)}, x_{(4)})}.$$

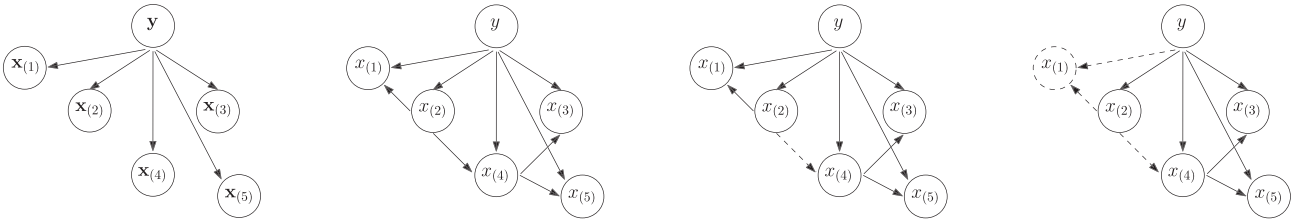
It can be easily observed from Fig. 2 that, according to the winner-takes-all rule, the code segment will be classified as being not fault prone. In what follows, several structure learning algorithms for the construction of BN classifiers will be discussed.

3.2 The Naive Bayes Classifier

A first classifier built on the principle of Bayesian networks is the Naive Bayes classifier [29]. This classifier merits its connotation to the underlying assumption of conditional independence between attributes given the class label. As a result of this assumption, the DAG associated with a Naive Bayes classifier is composed of a single parent (the unobserved class label y) and several children, each corresponding to an observed variable in the dataset. In spite of this often oversimplifying assumption, the Naive Bayes classifier typically performs surprisingly well. For instance, Domingos and Pazzani [28] found the Naive Bayes classifier performance to sometimes be superior to a number of decision tree induction algorithms, even on datasets with considerable variable dependencies. This result was also confirmed in a software fault prediction context by Menzies et al., who found the Naive Bayes classifier to outperform rule-based learners [74].

The Naive Bayes classifier proceeds by calculating the posterior probability of each class given the vector of observed variable inputs $(x_{i(1)}, \ldots, x_{i(n)})$ of each new code segment using Bayes' rule (1). As a result of the conditional independence assumption, the class-conditional probabilities can be restated as:

$$P(x_{i(1)}, \ldots, x_{i(n)}|y_i = y) = \prod_{j=1}^{n} P(x_{i(j)}|y_i = y). \tag{4}$$



(a) Naive Bayes network (b) Selective Tree Augmented Naive Bayes network (c) Selective Forest Augmented Naive Bayes network (d) Selective Forest Augmented Naive Bayes with Discarding network

Fig. 3. Examples of Bayesian network structures.

The probabilities $P(x_{i(j)}|y_i = y)$ are estimated by using frequency counts for the discrete variables and a normal or kernel density-based method for continuous variables [58]. Note that as a result of the simplifying assumption of conditional independence, Naive Bayes classifiers are easy to construct since the network structure is given a priori and no structure learning phase is required. Another advantage is its computational efficiency, especially since the model has the form of a product, which can be converted into a sum by using a logarithmic transformation. In this study, both Naive Bayes using a kernel density estimate for continuous variables as well as Naive Bayes after variable discretization are considered. Fig. 3a provides a graphical representation of a Naive Bayes classifier.

3.3 Augmented Naive Bayes Classifiers

The promising performance of the Naive Bayes classifier inspired several modifications to relax the conditional independence assumption. These modifications are mainly based on adding additional arcs between variables to account for dependencies present in the data or removing irrelevant or correlated variables from the network structure. A well-known example is the algorithm presented by Friedman et al., the *Tree Augmented Naive Bayes (TAN)* classifier, which allows every variable in the network to have one additional parent next to the class node [40]. One other such algorithm is the *Semi-Naive Bayesian* classifier developed by Kononenko [65], which partitions the variables into pairwise disjoint groups. The latter assumes that $x_{(j)}$ is conditionally independent of $x_{(j')}$ if and only if they belong to different groups. By contrast, the *Selective Naive Bayes* classifier tries to improve the Naive Bayes classifier by omitting certain variables to deal with strong correlation among attributes [67].

In this study, the Augmented Naive Bayes classifiers developed by Sacha [83] are used. Building upon the ideas introduced by Friedman et al., this family of Bayesian classifiers provides a further relaxation on the TAN approach: Not all attributes need to be dependent on the class node and there does not necessarily need to be an undirected path between two attributes that does not pass through the class node. The Augmented Naive Bayes algorithms consist of a combination of two dependency discovery operators and two augmenting operators, summarized in Table 1. The measure of dependency between

TABLE 1
Augmented Naive Bayes Approach: Different Operators

Dependency Discovery Operators	**Description**
Selective Augmented Naive Bayes (SAN)	Operator which connects the class node with the attributes it depends on. Starting with an empty set, SAN greedily searches for possible arcs from the class variable y to other variables $x_{(j)}$ optimizing a certain quality measure, see Table II. The selected variable together with an associated arc are added to the network which is then passed to one of the *augmenting* operators. The latter establishes the dependencies among *all* variables $x_{(j)}$, irrespective of their connection with the class node.
Selective Augmented Naive Bayes with Discarding (SAND)	Operator which connects the class node with the attributes it depends on, as the SAN operator does. The difference, however, lies in that SAND will discard all variables which are not dependent on the class node before passing the network to one of the *augmenting* operators. As a result, the discarded variables are not part of the network; the difference between a network resulting from the SAN operator and SAND operator is illustrated in Fig. 3c and 3d respectively. Dashed lines indicate absent arcs or nodes in a network.
Augmenting Operators	**Description**
Tree-Augmenter	Operator which builds the maximum spanning tree among a given set of attributes. The algorithm is based on a method developed by Chow and Liu [22], but differs in the way how the mutual information is calculated. Sacha uses the conditional or unconditional probability of $x_{(j)}$ and $x_{(j')}$ depending on whether there exists an arc between the class node and the attribute (see formula 5). The resulting network can be regarded as a generalization of the network obtained using a TAN classifier, *not* requiring all variables to be connected with the class variable.
Forest-Augmenter	Operator which is also used to establish dependencies between attributes, but allowing for more flexibility. The forest-augmenter can create dependencies between variables in the form of a number of disjoint trees not requiring the existence of an undirected path between two attributes that does not pass through the class node. The difference between both augmenting operators is shown in Fig. 3b and 3c.

TABLE 2
Augmented Naive Bayes Approach: Different Quality Measures

Quality Measures	Description
Standard Bayesian measure (SB)	This *global* quality measure was first proposed by [13] and is proportional to the posterior probability distribution $p(G, \Theta \mid D_{trn})$ with an added penalty term for network size. The network size or dimensionality is defined as the number of free parameters required to fully specify the joint probability distribution, $P_B(x_{(1)}, ..., x_{(n)})$.
Local Leave-One-Out-Cross Validation (LOO)	This *local* quality measure calculates iteratively the class probability conditional on the data, $P(y \mid x_{(1)}, \ldots, x_{(n)})$, using all observations minus one to estimate all parameters. The remaining observation is then used to assess the network quality in the class node [83].

two attributes, the conditional mutual information $I(x_{(j)}, x_{(j')})$, which is used by augmenting both operators, is defined as follows:

$$\begin{cases} \sum_{x_{(j)}, x_{(j')}} p(x_{(j)}, x_{(j')} \mid y) \log\left(\frac{p(x_{(j)}, x_{(j')} \mid y)}{p(x_{(j)} \mid y) p(x_{(j')})}\right) & \text{if } y \prec x_{(j)} \\ \sum_{x_{(j)}, x_{(j')}} p(x_{(j)}, x_{(j')} \mid y) \log\left(\frac{p(x_{(j)}, x_{(j')} \mid y)}{p(x_{(j)}) p(x_{(j')} \mid y)}\right) & \text{if } y \prec x_{(j')} \\ \sum_{x_{(j)}, x_{(j')}} p(x_{(j)}, x_{(j')} \mid y) \log\left(\frac{p(x_{(j)}, x_{(j')} \mid y)}{p(x_{(j)} \mid y) p(x_{(j')} \mid y)}\right) & \text{if } y \prec x_{(j)}, x_{(j')} \\ \sum_{x_{(j)}, x_{(j')}} p(x_{(j)}, x_{(j')}) \log\left(\frac{p(x_{(j)}, x_{(j')})}{p(x_{(j)}) p(x_{(j')})}\right) & \text{if } y \perp x_{(j)}, x_{(j')}. \end{cases} \quad (5)$$

Combining the dependency discovery operators with different augmenting operators from Table 1 yields four possible combinations. Note that both augmenting operators can also be applied directly on a Naive Bayes network, providing the following six Bayesian network classifiers:

- TAN: Tree Augmented Naive Bayes,
- FAN: Forest Augmented Naive Bayes,
- STAN: Selective Tree Augmented Naive Bayes,
- STAND: Selective Tree Augmented Naive Bayes with Discarding,
- SFAN: Selective Forest Augmented Naive Bayes,
- SFAND: Selective Forest Augmented Naive Bayes with Discarding.

The aim of these classifiers is to find a tradeoff between the simplicity of the Naive Bayes classifiers (with a limited number of parameters) and the more realistic and complex case of full dependency between the attributes.

Except for TAN, all of the above procedures adopt a quality measure to assess the fitness of a network given the data. Commonly, a distinction is made between global and local quality measures. The former evaluate the complete network, while the latter only evaluate the network at the class node. As the task of software fault prediction is in fact a classification task requiring the prediction of a single class attribute, local quality measures would seem the most preferable. In this study, a representative from both categories is used, see Table 2. Both quality measures were combined with the five algorithms defined above, resulting in 10 different BN learners.

The implementation of Sacha has been used for both the Naive Bayes and the Augmented Naive Bayes classifiers [83]. This implementation is available as a set of Weka bindings, which allows the execution of the software directly from within the Weka environment.[4]

3.4 General Bayesian Network Classifiers

All previously discussed methods restrain the network structure G in order to limit the search space of allowed DAGs. Omitting these restrictions, *General Bayesian Networks (GBN)* can adopt any DAG as G. Selecting the optimal structure is, however, known to be an NP-hard problem since the possible sets of parents for each variable grow exponentially with the number of candidate parents [20]. Several algorithms have been proposed to limit the computational expense of finding a suitable network structure. Commonly, these algorithms can be subdivided into two broad categories, i.e., those using a heuristic search procedure ("*Search-and-Score algorithms*") and algorithms which employ statistical tests to infer the conditional independence relationships among variables ("*Constraint-Based algorithms*") [66].

3.4.1 K2

Several Search-and-Score algorithms have been proposed in the literature, e.g., the Greedy Equivalence Search (GES) algorithm [19] and algorithms based on the use of genetic operators [68]. In this study, the K2 algorithm of Cooper and Herskovits, which employs a greedy search procedure, is investigated [23]. While greedy search can become trapped in local minima, it has been shown that K2 yields comparable results to other Search-and-Score algorithms [105].

The K2 algorithm adopts a bottom-up search strategy, assuming equal prior probabilities for all possible network structures, and considers all variables one by one, assuming some ordering in the variables. For each variable $x_{(i)}$, the posterior probability of the network structure where $x_{(i)}$ is conditionally independent of all other variables is evaluated. Next, the parents whose addition increases the posterior probability of the resulting network structure the most are sequentially added. When no further parents that increase the posterior probability of the network can be added, the algorithm traces back until all variables have been considered. The K2 algorithm, available in the Weka workbench, is used in this study [103].

3.4.2 MMHC

Opposite to the Search-and-Score paradigm is the use of statistical tests to verify whether certain conditional independencies between variables hold. Examples are the PC algorithm [89] and the Three Phase Dependency Analysis (TPDA) algorithm [18]. In our analysis, a hybrid combining the advantages of Search-and-Score and Constraint-Based algorithms is considered, i.e., the Max-Min Hill-Climbing (MMHC) algorithm proposed in [95]. Comparison with, among others, GES, TPDA, and PC empirically illustrated the strength of this algorithm [95].

The MMHC algorithm first constructs the skeleton of a Bayesian network (i.e., a network structure containing only undirected edges) by adopting a local discovery algorithm

4. www.jbnc.sourceforge.net.

called Max-Min Parents and Children (MMPC) to determine the parent-children set (**PC**) of every node. MMPC proceeds by sequentially adding nodes to a candidate **PC** set (**CPC**) as selected by a heuristic procedure. The set may contain false positives, which are removed in a second step. The algorithm tests whether any variable in **CPC** is conditionally independent of the target variable, given a blocking set $\mathbf{S} \subseteq \mathbf{CPC}$. If such variables are found, they are removed from **CPC**. As a measure of conditional (in)dependence, the G^2 measure, as described by Spirtes et al. [89], is used. This measure is asymptotically following a χ^2 distribution with appropriate degrees of freedom under the null hypothesis of conditional independence, which allows calculation of a p-value indicating the probability of falsely rejecting this null hypothesis. Conditional independence is assumed when the p-value is more than the significance level α (α equals 0.15 in this study). Once the skeleton is determined, the final network structure is learned using a greedy hill-climbing search which is constrained to add only an edge if it was discovered by MMPC. The *BDeu* score [51] is used to guide this greedy search. The network structure is induced using the Causal Explorer package for Matlab,[5] while the Bayesian Net Toolbox[6] is used for inference afterward.

3.5 Benchmark Classifiers

As illustrated by Fig. 1, numerous techniques other than BN classifiers have been used to construct software fault prediction models. As a reference, two such techniques are included, i.e., random forests and logistic regression, which are both implemented in the Weka toolbox [103]. These techniques are selected on the basis of illustrated performance in software fault prediction and other domains [7], [70].

Logistic regression is a well-known statistical technique that fits the data to the following expression:

$$P(y_i = 0|x_i) = \frac{1}{1 + e^{-x_i'\boldsymbol{\beta}}}, \tag{6}$$

where β is a vector of unknown parameters. Note that it is possible to reformulate a logistic regression model as a general BN model. However, instead of using, e.g., the empirical frequencies as parameter estimates, logistic regression typically uses an iterative parameter estimation procedure [104]. Further explanation on software fault prediction models using logistic regression can be found in, e.g., [70], [94].

Random forests is an ensemble learning schema that has been successfully applied in software fault prediction [45], [70]. It can be regarded as a classifier which consists of a collection of independently induced base classifiers which are then combined using a voting procedure. As originally proposed by Breiman, CART decision trees are adopted as base classifier [12]. Key in this approach is the dissimilarity among the base classifiers, which is obtained by adopting a bagging procedure to select the training samples of individual base classifiers and the selection of a random subset of attributes at each node. The latter is a parameter which, in line with Lessmann et al., is tuned using a five-fold stratified cross-validation approach.

5. www.dsl-lab.org/causal_explorer.
6. www.code.google.com/p/bnt.

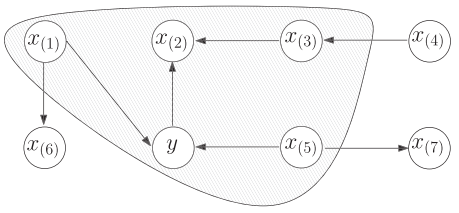


Fig. 4. The Markov blanket of a classification node $\mathbf{y}$.

3.6 Markov Blanket Feature Selection

Learning from high-dimensional data often poses considerable difficulties to machine learning techniques due to the presence of irrelevant or redundant features. Moreover, when considering more features, typically comparatively more parameters need to be estimated, which in turn induces additional uncertainty in these estimations [90].

Previous work on mining static code features indicated that a single best set of features does not exist, but instead the set of best features is highly dependent on the specific dataset [74]. As a result, several software fault prediction studies pass all features to the machine learning technique and let the technique decide which features should be selected [70], [99]. Such an approach is often feasible as most techniques include some sort of embedded feature selection or can be adjusted to this goal by, e.g., including a penalty on the size of the parameters [2]. This is, however, not the case for the Naive Bayes classifier and some of the Augmented Naive Bayes Classifiers, which thus also include uninformative variables.

The use of a Markov blanket-based feature selection approach provides a natural solution to this issue. The Markov blanket (MB) of a node y is the union of y's parents, y's children, and the parents of y's children and is the minimal variable subset conditioned on which all other variables are independent of y. In other words, no other variables than those contained in the MB of y need to be observed to predict the value of y. The concept is illustrated in Fig. 4, where the MB of y is indicated by the shaded area. For instance, the value of $x_{(6)}$ can be ignored when predicting the value of y as it is the child of a parent of y and thus is no part of the MB of y.

The HITON algorithm is used for the Markov blanket feature selection, which adopts the same test of conditional (in)dependence as the MMHC algorithm, the G^2 measure, and has been applied to the datasets at a significance level of 5 and 15 percent, referred to as MB.05 and MB.15, respectively [3]. Note that if attribute selection is performed, it is applied prior to training and testing the classifiers. Hence, every classifier is applied three times to each dataset. The feature selection algorithm has been implemented in the *Causal Explorer* package.

4 Empirical Layout

4.1 Datasets

The data considered in this study stems from two independent sources, i.e., from the NASA IV&V facility and the open source Eclipse Foundation. Both data sources are in the public domain, enabling researchers to validate

TABLE 3
Overview of Datasets: Origin

NASA	*Origin data set*	*Project size*	*# faulty modules / # modules*
JM1	Real time project in C with eight years of defect data associated	315 KSLOC	2,102 / 10,878 (19.32 %)
KC1	Storage management system for ground data in C++ with five years of defect data associated	43 KSLOC	325 / 2,107 (15.42 %)
MC1	Software developed in C & C++ for a combustion experiment on the space shuttle	63 KLOC	68 / 4,625 (1.47 %)
PC1	Flight software from an earth orbiting satellite in C which is no longer operational	40 KLOC	76 / 1,059 (7.18 %)
PC2	Software of a dynamic simulator for attitude control systems in C	26 KLOC	23 / 4,505 (0.51 %)
PC3	Flight software from an earth orbiting satellite in C	40 KLOC	160 / 1,511 (10.59 %)
PC4	Flight software from an earth orbiting satellite in C	36 KLOC	178 / 1,347 (13.21 %)
PC5	Software of safety enhancements of a cockpit upgrade in C++	164 KLOC	503 / 15,414 (3.26 %)
Eclipse	*Origin data set*	*Project size*	*# faulty files / # files*
Ecl2.0a	The Eclipse platform 2.0 was released on 27^{th} of June 2002	796.9 KLOC	975 / 6,729 (14.49 %)
Ecl2.1a	The Eclipse platform 2.1 was released on 27^{th} of March 2003	987.6 KLOC	854 / 7,888 (10.83 %)
Ecl3.0a	The Eclipse platform 3.0 was released on 25^{th} of June 2004	1,305.9 KLOC	1,568 / 10,593 (14.80 %)

our findings. Note that the set of static code features is not homogenous, including McCabe complexity, Halstead, OO, and LOC metrics, depending on the origin of the dataset. Notwithstanding this dissimilarity, the purpose of both data collection efforts is to investigate the relationship between static code features and software faults.

It should be noted that static code features are known to be correlated; previous work examining the different static code features, e.g., indicated that these could be grouped into four categories [71]. A first category related to metrics derived from flowgraphs (i.e., McCabe metrics), while a second category contained metrics related to the size and item count of a program. The two other categories represented different types of Halstead metrics. This again motivates the use of a feature selection procedure, especially when applying techniques that do not include some sort of embedded feature selection.

4.1.1 NASA IV&V Facility

The NASA datasets can be freely obtained directly from the NASA Metrics Data Program (MDP) repository which is hosted at the NASA IV&V facility website[7] or from the Promise repository.[8] Recently, it was pointed out that differences exist between the data from both sources. In this study, eight datasets taken from the NASA MDP repository have been preprocessed as detailed in Section 4.2 and studied. As machine learning typically benefits from more data, only datasets with more than 1,000 observations have been selected, see Table 3, top panel.

Table 4 provides an overview of all available features for each of the NASA datasets included in this study and indicates how they relate to each other. As noted earlier, such data can be mined directly from the source code using several purpose-built tools. The tool selected for this task was McCabe IQ 7.1 and it has been used for all datasets, providing a common measurement framework. The set of available static code features include LOC, Halstead, and McCabe complexity metrics. The first is arguably one of the widest used proxies for software complexity in fault prediction studies and has been used as an approximation of software size since the late 1960s [35]. As LOC counts have been recognized to be dependent on the selected programming language, a number of alternative measures were introduced in the 1970s to quantify software complexity. Two such sets of metrics are McCabe complexity metrics and Halstead software science metrics. The first maps a program or module to a flowchart where each node corresponds to a block of code where the flow is sequential and the arcs correspond to branches in the program. Software complexity is then related to the number of linearly independent paths through a program. Halstead metrics take a different perspective by considering a program or module as a sequence of tokens, i.e., a sequence of operators and operands. Based on the counts of these tokens, a number of derivative measures have been defined which are sometimes referred to as "software science" metrics [47]. Note that the projects stemming from the NASA IV&V facility were mainly developed using procedural programming, and typically only contain LOC, Halstead, and McCabe complexity metrics. For some projects, requirement metrics (projects "PC1," "CM1," and "JM1") [57] and class level metrics (project "KC1") [15] have also been collected. These additional metrics have not been considered in this study as these have not been collected on the same granularity level as the rest of the data.

4.1.2 Eclipse Foundation

The Eclipse platform project was founded in 2001 by IBM with the support of a consortium of software vendors. In early

7. www.mdp.ivv.nasa.gov.
8. www.promisedata.org.

TABLE 4
Overview of NASA Datasets: Selection of Attributes and Their Calculation

Data sets Metrics Data Program (MDP) repository			JM1	KC1	MC1	PC1	PC2	PC3	PC4	PC5
LOC based metrics	*Handle*	*Calculation*								
LOC_Total	LOC		✓	✓	✓	✓	✓	✓	✓	✓
LOC_Blank	BLOC		✓	✓	✓	✓		✓	✓	✓
LOC_Executable	SLOC		✓	✓	✓	✓	✓	✓	✓	✓
LOC_Comments	CLOC		✓	✓	✓	✓	✓	✓	✓	✓
LOC_Code_and_Comment	C&SLOC		✓	✓	✓	✓	✓	✓	✓	✓
Number_of_Lines	nl				✓	✓	✓	✓	✓	✓
Percent_Comments	% Comments	$\frac{\text{CLOC+C\&SLOC}}{\text{SLOC+CLOC+C\&SLOC}}$			✓	✓	✓	✓	✓	✓
McCabe metrics	*Handle*	*Calculation*								
Cyclomatic_Complexity	$v(G)$		✓	✓	✓	✓	✓	✓	✓	✓
Cyclomatic_Density	$vd(G)$				✓	✓	✓	✓	✓	✓
Decision_Density	$dd(G)$	$\frac{\text{Cond\_C}}{\text{Dec\_C}}$				✓	✓	✓	✓	
Design_Complexity	$iv(G)$		✓	✓	✓	✓	✓	✓	✓	✓
Design_Density	$id(G)$	$\frac{iv(G)}{v(G)}$			✓	✓	✓	✓	✓	✓
Essential_Complexity	$ev(G)$		✓	✓	✓	✓	✓	✓	✓	✓
Essential_Density	$ed(G)$	$\frac{ev(G)-1}{v(G)-1}$			✓	✓	✓	✓	✓	✓
Global_Data_Complexity	$gdv(G)$				✓					✓
Global_Data_Density	$gd(G)$	$\frac{gdv(G)}{v(G)}$			✓					✓
Norm_Cyclomatic_Compl	*Norm* $v(G)$	$\frac{v(G)}{nl}$			✓	✓	✓	✓	✓	✓
Maintenance Severity	Maint_Sev	$\frac{ev(G)}{v(G)}$			✓	✓	✓	✓	✓	✓
Halstead metrics	*Handle*	*Calculation*								
Num_Operators	N_1		✓	✓	✓	✓	✓	✓	✓	✓
Num_Operands	N_2		✓	✓	✓	✓	✓	✓	✓	✓
Num_Uniq_Operators	n_1		✓	✓	✓	✓	✓	✓	✓	✓
Num_Uniq_Operands	n_2		✓	✓	✓	✓	✓	✓	✓	✓
Length	N	$N_1 + N_2$	✓	✓	✓	✓	✓	✓	✓	✓
Difficulty	D	$\frac{n_1 \times N_2}{2 \times n_2}$	✓	✓	✓	✓	✓	✓	✓	✓
Level	L	$\frac{1}{\text{D}}$	✓	✓	✓	✓	✓	✓	✓	✓
Volume	V	$\text{N} \times \log_2(n_1 + n_2)$	✓	✓	✓	✓	✓	✓	✓	✓
Programming_Effort	E	$\text{D} \times \text{V}$	✓	✓	✓	✓	✓	✓	✓	✓
Programming_Time	T	$\frac{\text{E}}{18}$	✓	✓	✓	✓	✓	✓	✓	✓
Error_Estimate	B	$\frac{\text{V}}{3000}$	✓	✓	✓	✓	✓	✓	✓	✓
Content	I	$\frac{\text{V}}{\text{D}}$	✓	✓	✓	✓	✓	✓	✓	✓
Miscellaneous metrics	*Handle*	*Calculation*								
Branch_Count	Branch_C		✓	✓	✓	✓	✓	✓	✓	✓
Call_Pairs	Call_C				✓	✓	✓	✓	✓	✓
Condition_Count	Cond_C				✓	✓	✓	✓	✓	✓
Decision_Count	Dec_C				✓	✓	✓	✓	✓	✓
Edge_Count	Edge_C				✓	✓	✓	✓	✓	✓
Node_Count	Node_C				✓	✓	✓	✓	✓	✓
Parameter_Count	Parameter_C				✓	✓	✓	✓	✓	✓
Multiple_Condition_Count	Mul_Cond_C				✓	✓	✓	✓	✓	✓
Modified_Condition_Count	Mod_Cond_C				✓	✓	✓	✓	✓	✓

2004, the Eclipse Foundation was instated to support the growing Eclipse community which constitutes both individuals and companies.[9] Data from three major releases (release 2.0, 2.1, and 3.0) have been collected by the University of Saarland on the granularity of files and packages [109]. As fault prediction models built on a finer granularity provide more information to developers, only file level datasets are considered in this study. More information on the origin of the Eclipse datasets can be found in Table 3, bottom panel.

Static code features have been collected using the built-in Java parser of Eclipse; some features were only collected at a finer granularity (i.e., at the granularity of methods or classes) and were thus aggregated taking the average, total, and maximum value of the metrics. Table 5 provides an overview of all available features. These include LOC and McCabe complexity metrics as well as counts on the use of object-oriented constructs. The source code is matched with six months of postrelease failure data from the Eclipse bug repository [109].

9. www.eclipse.org.

4.2 Data Preprocessing

A first important step in each data mining exercise is preprocessing the data. In order to correctly assess the techniques discussed in Section 3, the same preprocessing steps are applied to each of the eleven datasets. Each observation (software module or file) in the datasets consists of a unique ID, several static code features, and an error count. First, the data used to learn and validate the models

TABLE 5
Overview of Eclipse Datasets: Selection of Attributes

Data sets Eclipse Foundation		
Method level metrics	*Handle*	*Available aggregators*
Fan out	FOUT	avg, max, total
Method lines of code	MLOC	avg, max, total
Nested block depth	NBD	avg, max, total
Number of parameters	PAR	avg, max, total
Cyclomatic complexity	VG	avg, max, total
Classes level metrics	*Handle*	*Available aggregators*
Number of fields	NOF	avg, max, total
Number of methods	NOM	avg, max, total
Number of static fields	NSF	avg, max, total
Number of static methods	NSM	avg, max, total
File level metrics	*Handle*	*Available aggregators*
Number of anonymous type declarations	ACD	value
Number of interfaces	NOI	value
Number of classes	NOT	value
Total lines of code	TLOC	value

are selected and thus the *ID* as well as attributes exhibiting zero variance are discarded. Moreover, observations with a total line count of zero are deemed logically incorrect and are removed. In the case of the NASA datasets, the *error density* is also removed. The *error count* 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., [70], [74], [97], [99], [100].

As some of the Bayesian learners are unable to cope with continuous features, a discretized version of each dataset was constructed using the algorithm of Fayyad and Irani [34]. This supervised discretization algorithm uses entropy to select subintervals that are as pure as possible with respect to the target attribute. Most techniques use the discretized datasets; if a technique employs the continuous data instead, it is labeled accordingly.

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 3, the smallest dataset contains 1,059 observations, while the largest contains up to 15,414 observations. Each of the datasets is randomly partitioned into two disjoint sets, i.e., a training and test set consisting of, respectively, 2/3 and 1/3 of the observations, using stratified sampling in order to preserve the class distribution. To account for a potential sampling bias, this partitioning procedure is repeated 10 times. Please note that as a side benefit of the automated collection of the static code features, the datasets are complete, i.e., there is no need for missing value handling.

After performing these steps, the datasets are passed to the learners described in Section 3 with and without first applying the Markov blanket feature selection procedure.

4.3 Classifier Evaluation

The induced models are compared to each other in terms of classification performance and comprehensibility. Note that the latter is often neglected during model selection, but is of critical importance when building software fault prediction models in practice [36].

4.3.1 Classifier Performance

A variety of performance measures has been used to gauge the strength of different classifiers and to select the appropriate model [1]. A very commonly used tool in the performance measurement of classifiers is receiver operating characteristic (ROC) analysis [33]. Typically, a classifier assigns a score $s(\mathbf{x}_i)$ to each instance i (i.e., each software module or file), based on its characteristics which are captured by $\mathbf{x}_i$.[10] Classification is then based on this score by defining a threshold t, whereby instances with scores lower (higher) than t are classified as (not) fault prone. An ROC curve shows the fraction of the identified faulty instances (the sensitivity) versus one minus the fraction of the identified fault free instances (one minus the specificity) for a varying threshold. A classifier whose ROC curve lies above the ROC curve of a second classifier is superior, and the point $(0, 1)$ corresponds to perfect classification.

Although ROC curves are a powerful tool for comparing classifiers, practitioners prefer a simple numeric measure indicating the performance over the visual comparison of ROC curves. Therefore, single point metrics such as the area under the ROC curve (AUC) were proposed. Let $f_l(s)$ be the probability density function of the scores s for the classes $l \in \{0, 1\}$, and $F_l(s)$ the corresponding cumulative distribution function. Then, it can be shown that AUC is defined as follows:

$$AUC = \int_{-\infty}^{\infty} F_0(s) f_1(s) ds. \tag{7}$$

The AUC can be regarded as a measure of aggregated classification performance as it in some sense averages performance over all possible thresholds [39]. Moreover, the AUC has an interesting statistical interpretation in the sense that it is the probability that a randomly chosen positive instance will be ranked higher than a randomly chosen negative instance.

Although the AUC has been extensively used, it was pointed out by Hand that the AUC is flawed as a measure of aggregated classification performance [48]. He developed a performance measure based on the expected minimum misclassification loss, whereby the misclassification costs are not exactly known but follow a probability distribution. Assume that misclassifying a faulty instance as not fault prone has a misclassification cost c_0, whereas a fault-free instance classified as fault-prone costs c_1. Then, the expected minimum misclassification loss is defined as

$$L = \mathrm{E}[b] \int_0^1 [c\pi_0(1 - F_0(T)) + (1 - c)\pi_1 F_1(T)] u(c) \mathrm{dc}, \tag{8}$$

with π_0 and π_1 the prior probabilities for fault-prone and not fault-prone instances, respectively, and T the optimal threshold t for a given value of c. Moreover, a variable transformation for the cost parameters has been applied, implying $b = c_0 + c_1$ and $c = c_0/(c_0 + c_1)$. The probability

10. Note that for notational convenience, it will be assumed that higher scores correspond to fault-free instances; if this is not the case, the scores need to be multiplied by minus one.

distribution of c (the ratio of the costs) is given by $u(c)$, and is assumed independent from the distribution of b (the level of the costs), leading to the expected value of b, $\mathrm{E}[b]$, outside the integral. Hand has shown that an AUC-based ranking is equivalent to a ranking based on the expected minimum misclassification loss for an appropriate choice of $u(c)$. The problem is that the probability distribution implied by the AUC measure varies with the empirical score distribution and thus with the classifiers. However, beliefs on the likely values of c should depend on contextual information, not on the classification tools used. Therefore, Hand proposes the H-measure, which takes a beta distribution with parameters α and β for $u(c)$, and is defined by

$$H = 1 - \frac{\int_0^1 [c\pi_0(1 - F_0(T)) + (1 - c)\pi_1 F_1(T)]u_{\alpha,\beta}(c)\mathrm{d}c}{\pi_0 \int_0^{\pi_1} c \cdot u_{\alpha,\beta}(c)\mathrm{d}c + \pi_1 \int_{\pi_1}^1 (1 - c) \cdot u_{\alpha,\beta}(c)\mathrm{d}c}. \quad (9)$$

This is a normalized measure based on the expected minimum misclassification loss, ranging from zero for a random classifier to one for a perfect classifier. Hand gives a number of examples which clearly illustrate how the AUC implies cost distributions which vary between classifiers [48].

Recently, an alternative and coherent interpretation of the AUC as a measure of aggregated classification performance was put forward by Flach et al. [39], relaxing the assumption made by Hand of selecting the optimal threshold T for a given value of c. More specifically, they showed that the AUC can be reformulated as being linearly related to expected misclassification loss by, instead of selecting this optimal threshold T, considering as many thresholds as there are examples, taking a uniform distribution over the data points and setting the threshold equal to the score of the selected instance, $t = s(\mathbf{x}_i)$. The H-measure, on the other hand, has the benefit of explicitly balancing the losses arising from classifying fault-prone as not fault-prone instances against the opposite type of misclassification, an aspect the AUC does not allow for.

In this study, the performance of the classification algorithms will be quantified using both measures. The AUC will be reported to verify the results of other studies and is shown to be less discriminative as the H-measure, supporting earlier findings in the literature [70], [75]. We also report the H-measure and the impact of varying the parameters for the beta distribution underlying the H-measure.

H-measure parameters. When no additional knowledge of the likely values of c is available, Hand proposes using a symmetric beta distribution with $\alpha = \beta = 2$. As no specific costs have been specified in the fault prediction literature, the H-measure will be calculated with these default values. However, depending on the context, it can be argued that misclassifying a faulty instance as non-fault-prone is more serious than the opposite, e.g., when considering high-risk software. On the other hand, e.g., in open source software development, the opposite is true: In order to keep participants motivated, it is advised to release early and often and thus the cost of missing defects is perhaps lower than the cost of delays due to unnecessary testing [81]. It can be seen from Table 3 that the data obtained from NASA relate to high risk projects, while the Eclipse project is an example of the latter.

In [55], the impact of different cost ratios using the MetaCost framework of Domingos was investigated. Misclassification costs ranging from $(c_0, c_1) = (75, 1)$, i.e., risk averse, to $(c_0, c_1) = (1, 75)$, i.e., delay averse, were selected. A similar approach is followed in this study, allowing c_0 (respectively, c_1) to take discrete values from 1 to 75 while keeping the opposite misclassification cost equal to one. As such, the robustness of the H-measure with respect to changes in the software development context is investigated. The findings of this analysis are reported in Section 5.

4.3.2 Classifier Comprehensibility

As opposed to classifier performance, no single point comprehensibility measure exists that is applicable to all types of classification models. Instead, comprehensibility can be explored from different perspectives, depending on the application context, and, likewise, a universal definition of comprehensibility is difficult to formulate. Typically, it is regarded as the extent to which there exists a mental fit between the end user and the classification model which justifies the subjective nature of this concept. Previous work found *representation type* and *model complexity* to be the main drivers of this mental fit [5], [54].

All BN learners considered in this study, with the exception of the Naive Bayes learner with kernel density estimation, result in a model which can be represented in a similar way, i.e., by a DAG augmented with a conditional probability table at each node, see Fig. 2. The discriminating aspect of the different Bayesian models lies in the complexity of the network structure and the number of entries in the probability table. It is argued by, e.g., Domingos that smaller, less complex models are to be preferred [27]. The complexity of the network structure of each algorithm with and without Markov blanket feature selection is quantified by the number of nodes and arcs in the DAG. Note that Naive Bayes and certain Augmented Naive Bayesian learners are unable to exclude variables from the network structure and thus invariably contain as many nodes as there are variables in the dataset. The aggregated size of the probability table is measured by the network dimensionality. This is defined as the number of parameters required to fully specify the joint probability distribution encoded by the network and is calculated as

$$DIM = \sum_{j=1}^{n} \big(r_j - 1\big) \cdot q_j, \quad (10)$$

with r_j being the cardinality of variable $x_{(j)}$ and

$$q_j = \prod_{x_{(j')} \in \Pi_{x_{(j)}}} r_{j'}, \quad (11)$$

with $\Pi_{x_{(j)}}$ the direct parent set for node $x_{(j)}$. Note that for Naive Bayes using a kernel density estimate, the network dimensionality cannot be calculated in a meaningful way and thus is excluded from this comparison.

4.4 Statistical Testing

The statistical testing framework described by Demšar is adopted in analyzing the results of this study [26]. In a first step of this procedure, the Friedman test is used to investigate whether classification performance is influenced

by specific factors. Two factors are of interest here, i.e., the type of (Bayesian Network) classifier and the use of feature selection prior to model construction. The Friedman test can be regarded as a nonparametric alternative to a repeated measures ANOVA to detect differences in treatment across multiple test attempts. The Friedman test statistic is defined as

$$\chi_F^2 = \frac{12P}{k(k+1)}\left[\sum_m R_m^2 - \frac{k(k+1)^2}{4}\right], \quad (12)$$

with R_m the average rank (AR) of treatment $m = 1, 2, \ldots, k$ over P test attempts. Under the null hypothesis of no significant differences between treatments, the Friedman test statistic is χ_F^2 distributed with $k-1$ degrees of freedom, at least when P and k are large enough (e.g., Lehman and D'Abrera specify $k \times P > 30$ as a guideline [69]). When comparing the results without feature selection with those after applying MB.15 and MB.05, k equals 3 and P equals $11 \times 17 = 187$. As the MB.05 feature selection procedure turned out to negatively impact predictive performance, $k = 17$ and $P = 11 \times 2 = 22$ when analyzing the impact of classifiers.

If the null hypothesis is rejected by the Friedman test, a posthoc Nemenyi test [76] is applied to compare all treatments to each other. The posthoc Nemenyi test is a nonparametric alternative to the Tukey test and is defined as:

$$CD = q_\alpha \sqrt{\frac{k(k+1)}{6P}}, \quad (13)$$

with critical value q_α based on the Studentized range statistic divided by $\sqrt{2}$. The performance difference between treatments is significant if their average ranks differ by at least the Critical Difference (CD). When assessing classifiers, an additional Bonferroni-Dunn test [30] is applied which compares all classifiers with the single best performing classifier, which is similar to posthoc Nemenyi but adjusts the confidence level to control for family-wise testing.

Also, the network dimensionality discussed in the previous section has been assessed using the Demšar framework. As explained, only for a selection of BN learners is it possible to calculate the network dimension in a meaningful way and thus k equals 14 and P equals $11 \times 2 = 22$ in this situation.

5 Results and Discussion

This section reports the results of the techniques discussed in Section 3. The average performance and standard deviation (indicated by the $\pm$ symbol) of the 10 independent iterations in terms of AUC and H-measure taking $\beta(2, 2)$ as distribution parameters are presented in Tables 6 and 7, respectively. The upper panel displays the results prior to Markov blanket feature selection, while the bottom panel shows the performance after MB feature selection at a significance level of 15 percent. The last column of each table displays the AR for each technique. The AR is calculated by ranking all techniques according to their performance on each dataset, rank 1 indicating the best performance and rank 17 the worst. The ARs are then obtained by averaging the ranks across all eight datasets. The best performing technique is reported in bold and underlined. The AR of a technique that is not significantly different from the best performing technique at 5 percent is tabulated in boldface font, while results significantly different at 1 percent are displayed in italic script. Classifiers differing at the 5 percent level but not at the 1 percent level are displayed in normal script. The Bonferroni-Dunn test is used for these assessments.

5.1 Empirical Results

The results without MB feature selection and with MB.15 and MB.05 are first compared to each other using a Friedman test. The outcome of this test indicated that feature selection did have a significant impact on the results (p-value 1.025×10^{-4} in the case of AUC and 3.331×10^{-16} in the case of the H-measure). Using the posthoc Bonferroni-Dunn test to compare the results without input selection with those obtained by performing the MB feature selection procedure prior to model construction, it was found that MB.15 did not result in significantly lower performance; MB.05 did, however, result in significantly worse performing models. Hence, the results of MB.05 are omitted in the remainder of this discussion.

5.1.1 Software Fault Prediction Techniques

All software fault prediction techniques are compared by first applying a Friedman test, followed by a posthoc Nemenyi test, as explained in Section 4.4. The Friedman test resulted in a p-value close to zero in both cases; the p-value was 2.622×10^{-21} for AUC and 2.507×10^{-23} for the H-measure. The null hypothesis of equal performance among all techniques is thus strongly rejected and, in a next step, the posthoc Nemenyi test assessing all pairwise differences between techniques is performed. The outcome of this test is given in Fig. 5. The horizontal axis in these figures corresponds to the AR of a technique across all datasets. The techniques are represented by a horizontal line; the more this line is situated to the left, the better performing a technique is. The left end of this line depicts the AR, while the length of the line corresponds to the critical distance for a difference between any two techniques to be significant at the 1 percent significance level. In the case of 17 techniques and 11 datasets, this critical distance equals 5.959. The first set of dotted and full vertical lines in the figure indicates the critical difference at, respectively, the 5 and 1 percent significance level with the overall best performing technique. The second set of vertical lines, displayed in bold, represents the differences with the best performing Bayesian Network learner. A technique is significantly outperformed if located on the right side of the vertical line.

Recently, an alternative to the AUC as a measure of aggregated classification performance was proposed, allowing specification of a probability distribution over the misclassification losses: the H-measure. The results of both metrics exhibit a similar pattern as random forests is found to be the overall best performing technique, both in terms of the AUC and H-measure, confirming a.o. the work of Lessmann et al. [70] and Guo et al. [45]. Furthermore, one can observe a similar ranking across techniques, indicating the same techniques as worst performing. One notable exception is logistic regression (Log. Reg.); in terms of AUC, this technique is found to be outperformed by RndFor at the

TABLE 6
Comparison of Classifier Performance: Out-of-Sample AUC Performance

Technique / **Data set**	**JM1**	**KC1**	**MC1**	**PC1**	**PC2**	**PC3**	**PC4**	**PC5**	**Ecl2.0a**	**Ecl2.1a**	**Ecl3.0a**	**AR**	
Log. Regr.	0.71	0.79	0.81	0.78	0.70	0.79	0.89	0.95	0.80	0.74	0.76	9.00	
using continuous data	± *0.010*	± *0.018*	± *0.073*	± *0.046*	± *0.126*	± *0.037*	± *0.020*	± *0.011*	± *0.007*	± *0.010*	± *0.007*		
RndFor	0.74	0.82	0.91	0.84	0.73	0.82	0.93	0.97	0.82	0.75	0.77	**2.18**	
using continuous data	± *0.006*	± *0.015*	± *0.041*	± *0.022*	± *0.129*	± *0.024*	± *0.010*	± *0.005*	± *0.008*	± *0.011*	± *0.004*		
Naive Bayes	0.70	0.80	0.86	0.77	0.84	0.76	0.82	0.95	0.79	0.73	0.77	*10.91*	
	± *0.009*	± *0.021*	± *0.030*	± *0.044*	± *0.097*	± *0.027*	± *0.018*	± *0.006*	± *0.006*	± *0.012*	± *0.006*		
Naive Bayes kernel	0.69	0.80	0.81	0.77	0.81	0.77	0.79	0.95	0.80	0.74	0.76	*12.18*	
using continuous data	± *0.010*	± *0.020*	± *0.057*	± *0.038*	± *0.129*	± *0.044*	± *0.018*	± *0.008*	± *0.006*	± *0.012*	± *0.006*		
TAN	0.71	0.79	0.88	0.81	0.83	0.79	0.89	0.97	0.80	0.74	0.75	**7.18**	
	± *0.009*	± *0.014*	± *0.035*	± *0.041*	± *0.099*	± *0.028*	± *0.007*	± *0.005*	± *0.012*	± *0.015*	± *0.006*		
FAN-SB	0.71	0.79	0.88	0.81	0.83	0.79	0.89	0.97	0.80	0.74	0.75	**7.68**	
	± *0.009*	± *0.014*	± *0.035*	± *0.041*	± *0.099*	± *0.028*	± *0.007*	± *0.005*	± *0.012*	± *0.015*	± *0.006*		
FAN-LCV_LO	0.71	0.79	0.88	0.81	0.83	0.79	0.89	0.97	0.80	0.74	0.75	**7.64**	
	± *0.009*	± *0.014*	± *0.043*	± *0.041*	± *0.099*	± *0.028*	± *0.007*	± *0.005*	± *0.012*	± *0.015*	± *0.006*		
SFAN-SB	0.71	0.79	0.88	0.81	0.83	0.79	0.89	0.97	0.80	0.74	0.75	**6.95**	
	± *0.009*	± *0.014*	± *0.036*	± *0.041*	± *0.099*	± *0.028*	± *0.007*	± *0.005*	± *0.012*	± *0.015*	± *0.006*		
SFAN-LCV_LO	0.72	0.78	0.88	0.80	0.84	0.78	0.90	0.96	0.79	0.74	0.76	**7.82**	
	± *0.010*	± *0.018*	± *0.030*	± *0.034*	± *0.104*	± *0.023*	± *0.016*	± *0.007*	± *0.009*	± *0.011*	± *0.007*		
SFAND-SB	0.71	0.79	0.88	0.81	0.83	0.79	0.89	0.97	0.80	0.74	0.75	**6.86**	
	± *0.009*	± *0.014*	± *0.036*	± *0.041*	± *0.099*	± *0.028*	± *0.007*	± *0.005*	± *0.012*	± *0.015*	± *0.006*		
SFAND-LCV_LO	0.72	0.78	0.88	0.80	0.84	0.78	0.90	0.96	0.79	0.74	0.76	**7.50**	
	± *0.010*	± *0.018*	± *0.030*	± *0.034*	± *0.104*	± *0.023*	± *0.016*	± *0.007*	± *0.009*	± *0.011*	± *0.007*		
STAN-SB	0.70	0.75	0.84	0.74	0.62	0.79	0.89	0.95	0.77	0.72	0.75	*14.27*	
	± *0.008*	± *0.019*	± *0.048*	± *0.106*	± *0.201*	± *0.025*	± *0.010*	± *0.011*	± *0.010*	± *0.015*	± *0.009*		
STAN-LCV_LO	0.71	0.77	0.87	0.78	0.79	0.79	0.90	0.96	0.79	0.73	0.76	*10.36*	
	± *0.011*	± *0.015*	± *0.048*	± *0.056*	± *0.095*	± *0.023*	± *0.014*	± *0.008*	± *0.007*	± *0.013*	± *0.009*		
STAND-SB	0.71	0.79	0.88	0.81	0.83	0.79	0.89	0.97	0.80	0.74	0.75	**6.59**	
	± *0.009*	± *0.014*	± *0.036*	± *0.041*	± *0.099*	± *0.028*	± *0.007*	± *0.005*	± *0.012*	± *0.015*	± *0.006*		
STAND-LCV_LO	0.72	0.78	0.88	0.80	0.83	0.78	0.90	0.96	0.79	0.74	0.76	**8.41**	
	± *0.010*	± *0.018*	± *0.030*	± *0.034*	± *0.101*	± *0.023*	± *0.016*	± *0.007*	± *0.009*	± *0.011*	± *0.007*		
K2	0.70	0.78	0.88	0.82	0.83	0.78	0.89	0.97	0.79	0.72	0.74	*11.36*	
	± *0.011*	± *0.018*	± *0.041*	± *0.055*	± *0.127*	± *0.025*	± *0.010*	± *0.006*	± *0.007*	± *0.011*	± *0.008*		
MMHC15	0.71	0.72	0.69	0.72	0.61	0.77	0.88	0.95	0.71	0.69	0.74	*16.09*	
	± *0.012*	± *0.092*	± *0.150*	± *0.064*	± *0.119*	± *0.022*	± *0.021*	± *0.010*	± *0.086*	± *0.063*	± *0.046*		
Log. Regr.	0.71	0.79	0.82	0.77	0.80	0.79	0.87	0.95	0.80	0.73	0.76	**11.09**	
using continuous data	± *0.009*	± *0.021*	± *0.051*	± *0.052*	± *0.123*	± *0.035*	± *0.019*	± *0.011*	± *0.007*	± *0.009*	± *0.008*		
RndFor	0.74	0.80	0.92	0.81	0.66	0.78	0.89	0.97	0.82	0.73	0.77	**5.82**	
using continuous data	± *0.007*	± *0.021*	± *0.047*	± *0.037*	± *0.129*	± *0.053*	± *0.036*	± *0.003*	± *0.007*	± *0.016*	± *0.006*		
Naive Bayes	0.70	0.79	0.87	0.77	0.82	0.79	0.85	0.95	0.79	0.73	0.76	**9.91**	
	± *0.008*	± *0.020*	± *0.035*	± *0.045*	± *0.096*	± *0.022*	± *0.014*	± *0.006*	± *0.006*	± *0.014*	± *0.005*		
Naive Bayes kernel	0.69	0.80	0.81	0.75	0.79	0.78	0.80	0.95	0.79	0.74	0.76	12.91	
using continuous data	± *0.011*	± *0.021*	± *0.051*	± *0.058*	± *0.158*	± *0.037*	± *0.012*	± *0.008*	± *0.006*	± *0.012*	± *0.007*		
TAN	0.71	0.80	0.88	0.78	0.81	0.79	0.90	0.97	0.80	0.74	0.76	**6.36**	
	± *0.009*	± *0.018*	± *0.039*	± *0.039*	± *0.105*	± *0.030*	± *0.016*	± *0.005*	± *0.010*	± *0.013*	± *0.008*		
FAN-SB	0.71	0.80	0.88	0.78	0.81	0.79	0.90	0.97	0.80	0.74	0.76	**6.14**	
	± *0.009*	± *0.018*	± *0.039*	± *0.039*	± *0.105*	± *0.030*	± *0.016*	± *0.005*	± *0.010*	± *0.013*	± *0.008*		
FAN-LCV_LO	0.71	0.80	0.88	0.78	0.81	0.79	0.90	0.97	0.80	0.74	0.76	**6.27**	
	± *0.009*	± *0.018*	± *0.039*	± *0.038*	± *0.105*	± *0.030*	± *0.016*	± *0.005*	± *0.010*	± *0.013*	± *0.008*		
SFAN-SB	0.71	0.80	0.88	0.78	0.81	0.79	0.90	0.97	0.80	0.74	0.76	**7.14**	
	± *0.009*	± *0.018*	± *0.040*	± *0.040*	± *0.105*	± *0.030*	± *0.016*	± *0.005*	± *0.010*	± *0.013*	± *0.008*		
SFAN-LCV_LO	0.72	0.79	0.87	0.78	0.81	0.79	0.90	0.96	0.79	0.74	0.76	**7.55**	
	± *0.010*	± *0.017*	± *0.038*	± *0.041*	± *0.100*	± *0.029*	± *0.014*	± *0.006*	± *0.011*	± *0.012*	± *0.005*		
SFAND-SB	0.71	0.80	0.88	0.78	0.81	0.79	0.90	0.97	0.80	0.74	0.76	**7.14**	
	± *0.009*	± *0.018*	± *0.040*	± *0.040*	± *0.105*	± *0.030*	± *0.016*	± *0.005*	± *0.010*	± *0.013*	± *0.008*		
SFAND-LCV_LO	0.72	0.79	0.87	0.78	0.81	0.79	0.90	0.96	0.79	0.74	0.76	**7.73**	
	± *0.010*	± *0.017*	± *0.038*	± *0.041*	± *0.100*	± *0.029*	± *0.014*	± *0.006*	± *0.011*	± *0.012*	± *0.005*		
STAN-SB	0.70	0.75	0.83	0.76	0.75	0.78	0.89	0.95	0.77	0.71	0.75	*15.36*	
	± *0.008*	± *0.032*	± *0.079*	± *0.044*	± *0.109*	± *0.021*	± *0.014*	± *0.010*	± *0.009*	± *0.017*	± *0.009*		
STAN-LCV_LO	0.71	0.78	0.87	0.78	0.82	0.79	0.89	0.96	0.79	0.74	0.76	**8.73**	
	± *0.011*	± *0.027*	± *0.034*	± *0.040*	± *0.093*	± *0.027*	± *0.017*	± *0.008*	± *0.008*	± *0.015*	± *0.008*		
STAND-SB	0.71	0.80	0.88	0.78	0.81	0.79	0.90	0.97	0.80	0.74	0.76	**7.32**	
	± *0.009*	± *0.018*	± *0.040*	± *0.040*	± *0.105*	± *0.030*	± *0.016*	± *0.005*	± *0.010*	± *0.013*	± *0.008*		
STAND-LCV_LO	0.72	0.79	0.87	0.78	0.81	0.79	0.90	0.96	0.79	0.74	0.76	**7.91**	
	± *0.010*	± *0.017*	± *0.038*	± *0.041*	± *0.098*	± *0.029*	± *0.014*	± *0.006*	± *0.011*	± *0.012*	± *0.005*		
K2	0.70	0.79	0.88	0.77	0.81	0.78	0.89	0.97	0.80	0.73	0.74	**10.00**	
	± *0.012*	± *0.019*	± *0.041*	± *0.038*	± *0.103*	± *0.028*	± *0.016*	± *0.006*	± *0.006*	± *0.015*	± *0.009*		
MMHC15	0.71	0.74	0.84	0.77	0.77	0.77	0.88	0.95	0.75	0.71	0.75	*15.64*	
	± *0.012*	± *0.060*	± *0.067*	± *0.040*	± *0.115*	± *0.024*	± *0.013*	± *0.011*	± *0.050*	± *0.042*	± *0.010*		

Without Markov Blanket feature selection

With Markov Blanket feature selection at 15%

1 percent significance level, while for the H-measure, Log. Reg. ranks second. This can be partially explained by the leveling out effect observed for AUC, i.e., several Augmented Naive Bayesian learners perform similarly in terms of AUC and are thus attributed a similar ranking. Log. Reg., which performs slightly worse in terms of AUC than these learners, is thus ranked much lower. The fact that the rankings are similar when no additional information on misclassification costs is included in the H-measure is interesting.

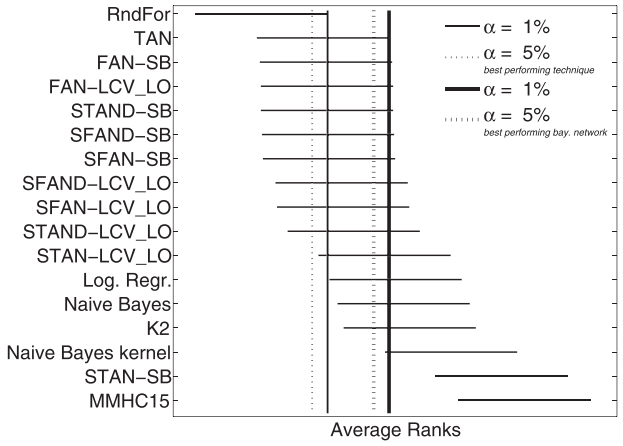
Interestingly, when considering the AUC metric, most of the BN learners are not significantly outperformed at the 1 percent significance level by RndFor, see Fig. 5a. However, unlike the conclusions of Lessmann [70], who only considered AUC, it is found that the Naive Bayes learner, which is often used in software fault prediction research, is outperformed at the 1 percent significance level. Similar results can be found when focusing on the H-measure; giving more discriminative results, the Naive Bayes learner as well as a number of augmented Naive Bayes classifiers are found to be significantly outperformed at the 1 percent level. As such, other BN learners which provide a more informative network structure can indeed be regarded as a valid alternative to Naive Bayes. Considering BN learners only, Tree Augmented Naive Bayes was found to be the best performing classifier, while STAN-SB and MMHC15 are found to perform significantly worse than TAN at the 1 percent level, in the case of both the AUC and the H-measure. Especially, the fact that MMHC15 scores last is noteworthy, as this BN

TABLE 7
Comparison of Classifier Performance: Out-of-Sample H-Measure Performance

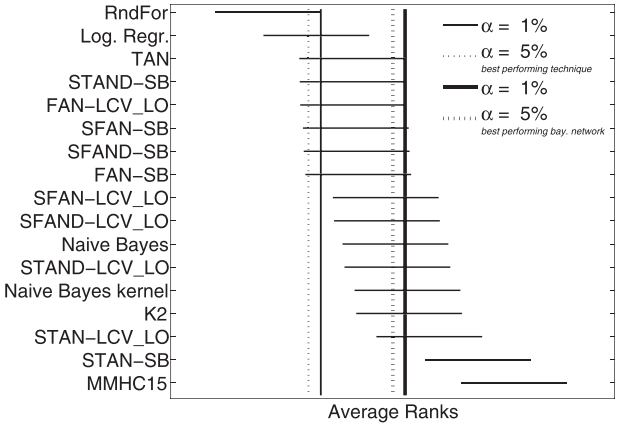
Technique / **Data set**	**JM1**	**KC1**	**MC1**	**PC1**	**PC2**	**PC3**	**PC4**	**PC5**	**Ecl2.0a**	**Ecl2.1a**	**Ecl3.0a**	**AR**	
Log. Regr.	0.113	0.193	0.197	0.115	0.022	0.124	0.367	0.270	0.183	0.094	0.126	**5.09**	*Without* Markov Blanket feature selection
using continuous data	± *0.0129*	± *0.0273*	± *0.1079*	± *0.0555*	± *0.0452*	± *0.0468*	± *0.0453*	± *0.0235*	± *0.0076*	± *0.0129*	± *0.0095*		
RndFor	0.136	0.202	0.379	0.233	0.007	0.171	0.430	0.380	0.220	0.088	0.154	**2.45**	
using continuous data	± *0.0099*	± *0.0365*	± *0.0881*	± *0.0609*	± *0.0079*	± *0.0334*	± *0.0431*	± *0.0344*	± *0.0141*	± *0.0076*	± *0.0109*		
Naive Bayes	0.093	0.159	0.198	0.108	0.010	0.082	0.205	0.157	0.162	0.093	0.134	*10.73*	
	± *0.0090*	± *0.0315*	± *0.0980*	± *0.0573*	± *0.0087*	± *0.0246*	± *0.0224*	± *0.0111*	± *0.0142*	± *0.0118*	± *0.0087*		
Naive Bayes kernel	0.087	0.163	0.010	0.098	0.023	0.104	0.149	0.217	0.164	0.087	0.120	*10.36*	
using continuous data	± *0.0089*	± *0.0323*	± *0.0050*	± *0.0544*	± *0.0228*	± *0.0363*	± *0.0416*	± *0.0249*	± *0.0132*	± *0.0115*	± *0.0067*		
TAN	0.098	0.151	0.235	0.124	0.010	0.097	0.289	0.280	0.170	0.087	0.119	**7.45**	
	± *0.0072*	± *0.0203*	± *0.0789*	± *0.0467*	± *0.0145*	± *0.0317*	± *0.0178*	± *0.0170*	± *0.0230*	± *0.0113*	± *0.0087*		
FAN-SB	0.098	0.151	0.235	0.124	0.010	0.096	0.289	0.280	0.170	0.087	0.119	**8.23**	
	± *0.0072*	± *0.0203*	± *0.0789*	± *0.0466*	± *0.0143*	± *0.0314*	± *0.0178*	± *0.0170*	± *0.0230*	± *0.0113*	± *0.0087*		
FAN-LCV_LO	0.098	0.151	0.235	0.124	0.010	0.097	0.289	0.280	0.170	0.087	0.119	**7.82**	
	± *0.0072*	± *0.0203*	± *0.0788*	± *0.0467*	± *0.0145*	± *0.0317*	± *0.0178*	± *0.0170*	± *0.0230*	± *0.0113*	± *0.0087*		
SFAN-SB	0.098	0.151	0.235	0.124	0.010	0.097	0.289	0.280	0.170	0.087	0.119	**7.95**	
	± *0.0072*	± *0.0205*	± *0.0790*	± *0.0467*	± *0.0143*	± *0.0315*	± *0.0178*	± *0.0177*	± *0.0228*	± *0.0115*	± *0.0086*		
SFAN-LCV_LO	0.104	0.144	0.247	0.117	0.015	0.081	0.306	0.277	0.161	0.088	0.117	**8.45**	
	± *0.0102*	± *0.0286*	± *0.0862*	± *0.0545*	± *0.0225*	± *0.0188*	± *0.0435*	± *0.0306*	± *0.0130*	± *0.0167*	± *0.0103*		
SFAND-SB	0.098	0.151	0.235	0.125	0.010	0.096	0.289	0.280	0.170	0.087	0.119	**8.05**	
	± *0.0072*	± *0.0205*	± *0.0790*	± *0.0474*	± *0.0143*	± *0.0312*	± *0.0178*	± *0.0177*	± *0.0228*	± *0.0115*	± *0.0086*		
SFAND-LCV_LO	0.104	0.144	0.247	0.117	0.015	0.081	0.306	0.277	0.161	0.088	0.117	**8.50**	
	± *0.0102*	± *0.0286*	± *0.0862*	± *0.0545*	± *0.0225*	± *0.0188*	± *0.0435*	± *0.0306*	± *0.0130*	± *0.0167*	± *0.0103*		
STAN-SB	0.092	0.127	0.238	0.064	0.010	0.093	0.282	0.222	0.136	0.071	0.108	*13.36*	
	± *0.0067*	± *0.0243*	± *0.0869*	± *0.0353*	± *0.0204*	± *0.0282*	± *0.0259*	± *0.0270*	± *0.0153*	± *0.0104*	± *0.0104*		
STAN-LCV_LO	0.101	0.153	0.231	0.099	0.005	0.092	0.301	0.271	0.157	0.080	0.115	*11.45*	
	± *0.0076*	± *0.0267*	± *0.0838*	± *0.0390*	± *0.0045*	± *0.0207*	± *0.0326*	± *0.0323*	± *0.0192*	± *0.0108*	± *0.0073*		
STAND-SB	0.098	0.151	0.235	0.125	0.010	0.097	0.289	0.280	0.170	0.087	0.119	**7.50**	
	± *0.0072*	± *0.0205*	± *0.0790*	± *0.0474*	± *0.0145*	± *0.0315*	± *0.0178*	± *0.0177*	± *0.0228*	± *0.0115*	± *0.0086*		
STAND-LCV_LO	0.104	0.144	0.247	0.118	0.010	0.081	0.305	0.277	0.161	0.088	0.117	9.32	
	± *0.0102*	± *0.0286*	± *0.0862*	± *0.0536*	± *0.0120*	± *0.0188*	± *0.0406*	± *0.0306*	± *0.0130*	± *0.0167*	± *0.0103*		
K2	0.093	0.150	0.264	0.125	0.013	0.084	0.275	0.308	0.158	0.072	0.104	9.82	
	± *0.0090*	± *0.0320*	± *0.0832*	± *0.0590*	± *0.0188*	± *0.0346*	± *0.0314*	± *0.0145*	± *0.0139*	± *0.0105*	± *0.0057*		
MMHC15	0.090	0.086	0.049	0.081	0.000	0.066	0.270	0.217	0.093	0.057	0.097	*16.45*	
	± *0.0100*	± *0.0441*	± *0.0807*	± *0.0515*	± *0.0003*	± *0.0119*	± *0.0449*	± *0.0275*	± *0.0546*	± *0.0267*	± *0.0259*		
Log. Regr.	0.111	0.180	0.114	0.118	0.023	0.112	0.282	0.269	0.180	0.093	0.127	**5.36**	*With* Markov Blanket feature selection at 15%
using continuous data	± *0.0121*	± *0.0425*	± *0.1300*	± *0.0604*	± *0.0490*	± *0.0294*	± *0.0689*	± *0.0238*	± *0.0123*	± *0.0110*	± *0.0091*		
RndFor	0.133	0.181	0.397	0.220	0.016	0.113	0.304	0.380	0.220	0.070	0.147	**2.55**	
using continuous data	± *0.0102*	± *0.0439*	± *0.0879*	± *0.0577*	± *0.0386*	± *0.0450*	± *0.0840*	± *0.0377*	± *0.0173*	± *0.0112*	± *0.0101*		
Naive Bayes	0.093	0.160	0.236	0.087	0.007	0.084	0.247	0.160	0.167	0.090	0.131	**8.64**	
	± *0.0088*	± *0.0354*	± *0.0730*	± *0.0427*	± *0.0056*	± *0.0162*	± *0.0321*	± *0.0123*	± *0.0161*	± *0.0105*	± *0.0067*		
Naive Bayes kernel	0.087	0.162	0.029	0.103	0.029	0.106	0.167	0.217	0.170	0.080	0.121	*10.36*	
using continuous data	± *0.0090*	± *0.0304*	± *0.0207*	± *0.0576*	± *0.0397*	± *0.0318*	± *0.0449*	± *0.0248*	± *0.0165*	± *0.0117*	± *0.0050*		
TAN	0.097	0.164	0.235	0.096	0.006	0.079	0.290	0.277	0.172	0.089	0.122	**7.05**	
	± *0.0098*	± *0.0289*	± *0.0875*	± *0.0431*	± *0.0048*	± *0.0213*	± *0.0317*	± *0.0202*	± *0.0210*	± *0.0134*	± *0.0100*		
FAN-SB	0.097	0.164	0.235	0.096	0.006	0.079	0.290	0.277	0.172	0.089	0.122	**6.95**	
	± *0.0098*	± *0.0289*	± *0.0875*	± *0.0431*	± *0.0048*	± *0.0213*	± *0.0318*	± *0.0202*	± *0.0210*	± *0.0134*	± *0.0100*		
FAN-LCV_LO	0.097	0.164	0.235	0.096	0.006	0.079	0.291	0.277	0.172	0.089	0.122	**6.77**	
	± *0.0098*	± *0.0289*	± *0.0875*	± *0.0431*	± *0.0048*	± *0.0213*	± *0.0312*	± *0.0202*	± *0.0210*	± *0.0134*	± *0.0100*		
SFAN-SB	0.097	0.164	0.231	0.096	0.006	0.079	0.290	0.278	0.172	0.089	0.122	**6.95**	
	± *0.0098*	± *0.0290*	± *0.0929*	± *0.0435*	± *0.0048*	± *0.0213*	± *0.0318*	± *0.0200*	± *0.0209*	± *0.0136*	± *0.0099*		
SFAN-LCV_LO	0.104	0.149	0.218	0.090	0.006	0.081	0.289	0.277	0.159	0.085	0.118	9.82	
	± *0.0104*	± *0.0337*	± *0.0806*	± *0.0425*	± *0.0049*	± *0.0208*	± *0.0317*	± *0.0302*	± *0.0162*	± *0.0154*	± *0.0106*		
SFAND-SB	0.097	0.164	0.231	0.096	0.006	0.079	0.290	0.278	0.172	0.089	0.122	**6.95**	
	± *0.0098*	± *0.0290*	± *0.0929*	± *0.0435*	± *0.0048*	± *0.0213*	± *0.0318*	± *0.0200*	± *0.0209*	± *0.0136*	± *0.0099*		
SFAND-LCV_LO	0.104	0.149	0.218	0.090	0.006	0.081	0.289	0.277	0.159	0.085	0.118	9.91	
	± *0.0104*	± *0.0337*	± *0.0806*	± *0.0425*	± *0.0049*	± *0.0208*	± *0.0317*	± *0.0302*	± *0.0162*	± *0.0154*	± *0.0106*		
STAN-SB	0.092	0.128	0.201	0.086	0.004	0.070	0.282	0.222	0.138	0.072	0.109	*15.27*	
	± *0.0077*	± *0.0251*	± *0.0956*	± *0.0478*	± *0.0054*	± *0.0157*	± *0.0336*	± *0.0274*	± *0.0168*	± *0.0134*	± *0.0100*		
STAN-LCV_LO	0.101	0.149	0.221	0.095	0.006	0.080	0.287	0.273	0.154	0.084	0.117	*11.73*	
	± *0.0082*	± *0.0370*	± *0.0645*	± *0.0430*	± *0.0048*	± *0.0192*	± *0.0297*	± *0.0306*	± *0.0202*	± *0.0142*	± *0.0069*		
STAND-SB	0.097	0.164	0.231	0.096	0.006	0.079	0.290	0.278	0.172	0.089	0.122	**7.05**	
	± *0.0098*	± *0.0290*	± *0.0929*	± *0.0435*	± *0.0048*	± *0.0213*	± *0.0317*	± *0.0200*	± *0.0209*	± *0.0136*	± *0.0099*		
STAND-LCV_LO	0.104	0.149	0.218	0.088	0.006	0.081	0.289	0.277	0.159	0.085	0.118	*10.27*	
	± *0.0104*	± *0.0337*	± *0.0806*	± *0.0445*	± *0.0049*	± *0.0208*	± *0.0320*	± *0.0302*	± *0.0163*	± *0.0154*	± *0.0106*		
K2	0.093	0.160	0.225	0.093	0.006	0.074	0.288	0.308	0.158	0.077	0.108	*11.09*	
	± *0.0101*	± *0.0321*	± *0.0726*	± *0.0445*	± *0.0051*	± *0.0173*	± *0.0293*	± *0.0143*	± *0.0145*	± *0.0109*	± *0.0107*		
MMHC15	0.090	0.100	0.189	0.087	0.005	0.070	0.267	0.206	0.118	0.063	0.105	*16.27*	
	± *0.0100*	± *0.0252*	± *0.0984*	± *0.0355*	± *0.0051*	± *0.0132*	± *0.0206*	± *0.0266*	± *0.0373*	± *0.0211*	± *0.0090*		

learner allows construction of any possible DAG as network structure. This can be explained by the fact that MMHC15 uses conditional independence tests to determine the network structure; even small amounts of noise in the dataset can lead to incorrect conclusions reached by such tests [84].

As explained in Section 4.3, the H-measure relies on a beta distribution characterized by two parameters which determine the likelihood of different cost ratios. It can be argued that this cost ratio is in fact context specific, and distribution parameters reflecting different cost ratios should be considered [55]. Parameter settings reflecting a different development context have thus been adopted, investigating the robustness of the H-measure in the context of software fault prediction. The outcome is presented in Fig. 6. The horizontal axis of this figure represents the expected value of the cost ratio, while the vertical axis corresponds to the AR. Techniques are represented by a line; if a technique does not perform statistically worse than the best performing technique at the 5 percent significance level, a full line is used and a dotted line otherwise. Bonferroni-Dunn tests are used in assessing the techniques at each cost ratio. Note that to improve readability, only a selection of techniques is shown, combining the best techniques both from a comprehensibility and performance point of view. It can be seen that RndFor remains the overall best performing technique when the cost ratio is larger than one, which corresponds to a risk averse development context. When considering a delay averse context, however,



(a) AUC



(b) H-measure with $\beta(2, 2)$

Fig. 5. Ranking of software fault prediction models for (a) the AUC and (b) H-measure with $\beta(2, 2)$ using the posthoc Nemenyi test.

very different conclusions can be reached. In such a context, Augmented Bayesian Network classifiers are found to be best performing. A cost ratio of one seems to be pivotal in this respect. One possible explanation for these findings lies in the fact that BN learners are known to be biased, exhibiting a tendency toward overconfidence in their predictions [49]. This reaffirms earlier conclusions concerning the importance of taking development context into account in software fault prediction [55], [56].

5.1.2 Markov Blanket Feature Selection

It is known that several static code features are correlated and, e.g., principal component analysis or factor analysis has previously been used to reduce the number of features [61], [71], [97]. A possible downside of such approach is a decrease in comprehensibility as several static code features are aggregated into a single feature. An alternative explored by, e.g., Menzies et al. in the context of the NASA datasets is the use of a filter approach to select the most informative subset of features prior to model construction [74]. Catal and Diri also considered a filter approach and compared it to directly discarding aggregated features such as derived Halstead measures [15]. Both confirmed the possibility of

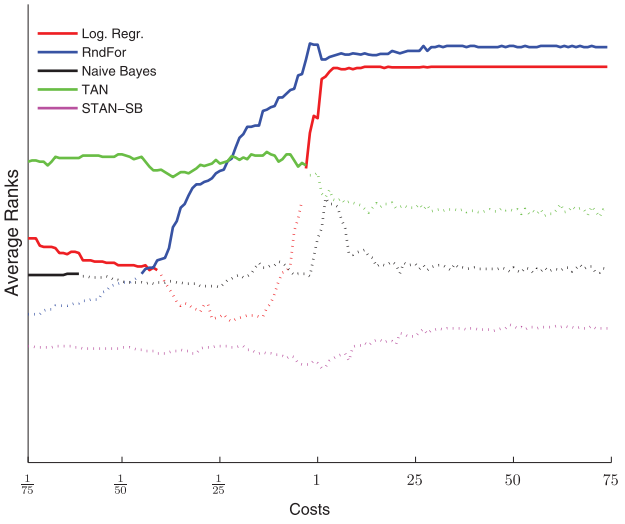


Fig. 6. Robustness of the H-measure.

selecting a set of most informative features from the data without incurring a performance penalty. In the first study, the authors were able to build fault prediction models based on three features, while the filter employed in the more recent study of Catal and Diri selected between three and eight metrics prior to model construction. The MB feature selection procedure of this study can be regarded as an example of a filter approach. It was in some cases able to select as little as five attributes; however, on some datasets the MB included up to 23 features. The MB.05 filter effectively further reduced the number of selected features, but resulted in lowered performance. Menzies et al. reported Halstead and LOC-based metrics to be the most often selected features.[11] Fig. 7 reports our findings hereon; the bar chart depicts the average number of attributes selected by the MB.15 procedure per dataset and per group of static code features. It can be seen that in the case of the NASA datasets, Halstead and LOC-based metrics are most often selected by the MB.15 filter. A notable exception is the PC5 dataset, for which McCabe complexity metrics were found to be the second most important group. Note that this last dataset was not included in their study. The Eclipse datasets, containing an alternative set of static code features, provide another picture as method level attributes are prevalent. In all three Eclipse datasets, metrics collected at different granularity were selected. Investigation at the level of individual attributes reveals significant differences in selected features between datasets. This supports the findings of Menzies et al., who concluded that "The best attributes to use for defect prediction vary from dataset to dataset." Finally, it can be argued that depending on the data, other feature selection techniques seem more effective than the MB procedure. This can in part be explained by the requirement to discretize the data when considering BN learners. Note that RndFor and some of the BN learners investigated in this study also include embedded feature selection, the impact of which is further discussed in the next section.

11. Note that our selection of datasets is not identical to Menzies' study and that minor differences exist in the grouping of static code features, see Table 4. For example, "Percent_comments" was regarded as an LOC-based static code feature, in line with the documentation of the NASA MDP.

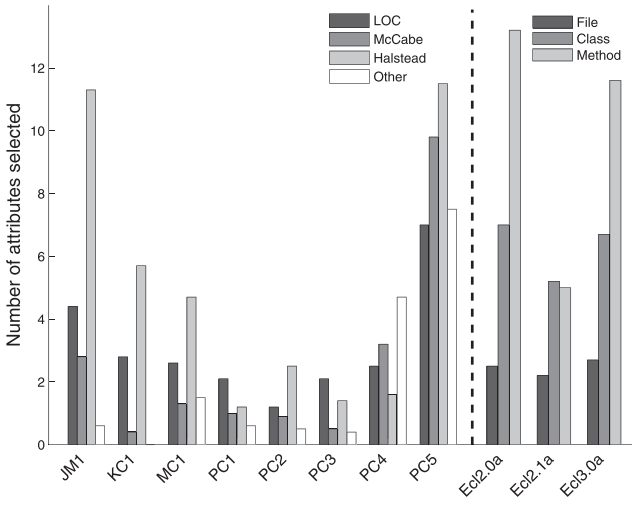
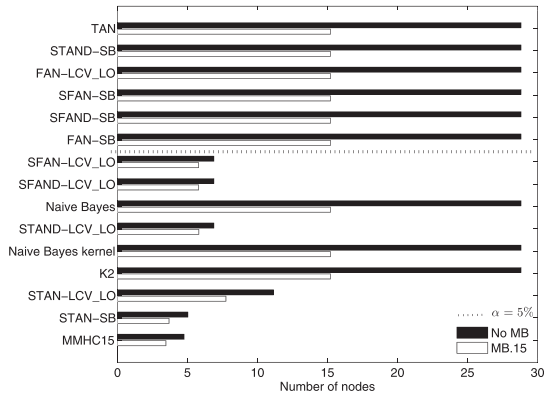


Fig. 7. Bar chart of the average number of selected attributes per dataset and per attribute group.

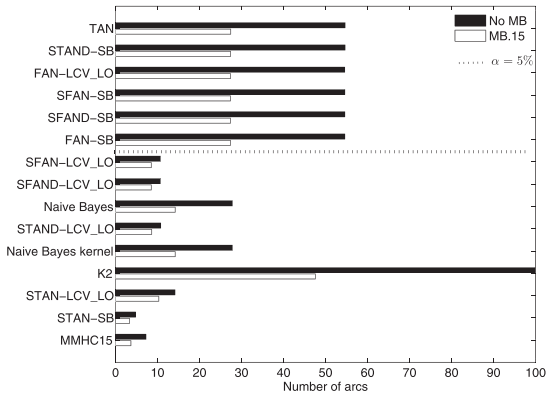
5.2 Comprehensibility of the Bayesian Networks

Several characteristics constitute a good software fault prediction model, of which performance is only one element. Model comprehensibility is also important, especially if such a model would be deployed in a real life setting [36]. As argued by, e.g., Kotsiantis et al. [66], BN classifiers are among the most comprehensible classifiers, but their comprehensibility can be hampered by the complexity of the network structure. Fig. 8 reports on this aspect by plotting the number of nodes, arcs, and the network dimensionality of each BN learner, both with and without prior application of the MB feature selection procedure. Techniques are ordered according to their classification performance using the H-measure; a technique situated above the dotted line was not found to be significantly outperformed at the 5 percent level by RndFor, the best performing learner.

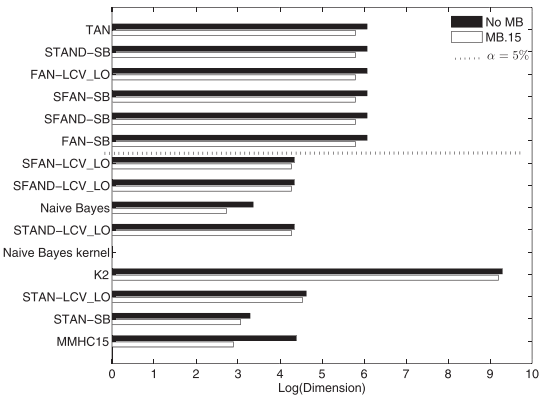
The graphs illustrate the impact of MB feature selection on network complexity by reducing the number of nodes and arcs in the network and lowering the number of parameters to be estimated, or network dimension. Similarly to the performance assessment, a Friedman test is first carried out to establish whether differences observed in the network dimension are significant. As the null hypothesis of no significant differences is strongly rejected with a p-value of 3.473×10^{-49}, a Bonferroni-Dunn test is performed comparing the best BN learner to all others. The results are depicted in Fig. 9 and indicate that Selective Tree Augmented Naive Bayes using the Standard Bayesian quality measure, STAN-SB, is the BN learner associated with the lowest network dimension. As such, one can argue that models induced by this learner are the simplest and most comprehensible. Naive Bayes, MMHC15, and several Augmented Naive Bayes learners using the Local Leave-One-out Cross Validation



(a) Number of nodes in the network



(b) Number of arcs in the network



(c) Logarithm of network dimension

Fig. 8. Comparison of Bayesian networks: comprehensibility.

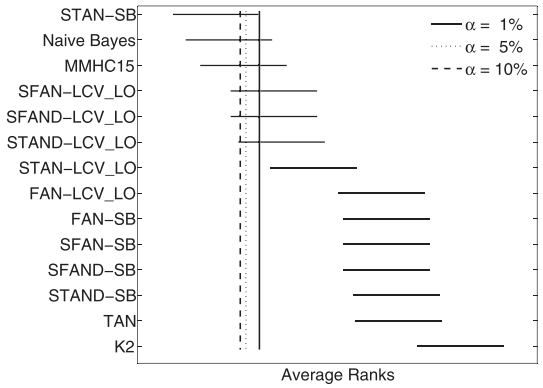


Fig. 9. Ranking of software fault prediction models for the network dimension using the Bonferroni-Dunn test.

quality measure to assess network fitness are found to be not significantly more complex. Taking into account, however, the fact that STAN-SB is outperformed by the best performing Bayesian learner, TAN, it can be argued that the use of a local quality measure is arguably better than using the Standard Bayesian quality measure as it results in similar performance to TAN while resulting in networks which are not significantly more complex than STAN-SB. More generally, one can observe that the Augmented Naive Bayes classifiers, which provide a relaxation to the TAN assumption, are able to reduce the number of nodes and arcs compared to TAN, without a loss in predictive power.

General Bayesian Networks, which are able to adopt any DAG as network structure, are found to be less appealing. MMHC15, which performed significantly worse than other BN learners, typically constructs very simple networks. These networks are often even overly simple, containing only a very limited set of features (nodes). The other General Bayesian Network learner, K2, performed much better but at the expense of very complex network structures.[12]

When selecting the optimal learner to construct software fault prediction models, a tradeoff is typically made between model comprehensibility and classification performance. It should be noted that the techniques found to result in the most comprehensible models are also found to be outperformed by random forests. Hence, it can be argued that when gaining insight into what drives software faults is of key importance, BN classifiers offer considerable advantages over other more opaque models. More specifically, the NB learner as well as several Augmented Bayesian Learners using the LOO-CV quality criterion during network construction are to be recommended. An important note in this respect is that Naive Bayes is typically easy to implement and can be written as a sum of logs to obtain a linear model [49], [97]. However, this learner is unable to discard uninformative attributes, which can prove important in gaining further insight into fault prediction. On the other hand, when classification performance is crucial, other techniques such as random forests would seem more appropriate. As discussed in the previous section, however, the question of which technique results in the best predictive performance depends on the development context.

As an example, Fig. 10 shows the network for the PC1 dataset learned by the STAND LCV_LO classifier (without prior input selection), a technique not found to be outperformed by the best Bayesian learner while typically not resulting in significantly more complex networks than STAN-SB. As one can observe, the algorithm was able to make accurate predictions retaining only five features. When interpreting the network, it is important to realize that the existence of an arc does not necessarily imply causality, but rather should be seen as (conditional) dependence between the variables. In this network, the presence of software faults is directly governed by CLOC (number of commentary lines), I (Halstead content), and the normalized cyclomatic complexity. Further correlations between, e.g., I and the number of unique operands can be discerned, which is plausible when considering that the latter serves as input to calculate the first. The relations present in the network can be helpful when for instance issuing guidelines on software complexity to programmers.

6 Conclusion

Time and cost effective software development are decisive for today's developers and since the pioneering work from the 1970s, several avenues to tackle problems related hereto have been investigated. Software fault prediction can be regarded as one piece of the solution to these issues. It is argued by Lessmann et al. that fault prediction techniques should not be judged on their predictive performance alone, but that other aspects such as computational efficiency, ease of use, and especially comprehensibility should also be paid attention to [70].

This paper tries to answer this call by comparing 15 Bayesian network learners both in terms of the Area Under the ROC Curve and the recently introduced H-measure. The results of the experiments show that Augmented Naive Bayes classifiers can yield similar or better performance than the commonly used Naive Bayes classifier. This additional performance, however, comes at the expense of more complex models. Considering comprehensible models only, Augmented Naive Bayes classifiers using the Local Leave-One-out Cross Validation quality measure are to be recommended. The Naive Bayes classifier, which can be turned into a linear model, is also a valid alternative, despite its simple network structure. General Bayesian Networks were found to be either outperformed by other Bayesian learners or to result in overly complex network structures. It can be argued that networks which focus on a smaller set of highly predictive features provide practitioners with the means to gain insights more easily into the drivers of software faults and, to further capitalize hereon, the use of MB feature selection was also tested. The outcome indicates that MB is able to reduce the number of variables while not negatively impacting performance. However, other feature selection approaches are possibly able to select an even smaller set of highly predictive features.

Depending on the development context and the associated costs of misclassifying a (non)faulty instance, other more opaque models are found to be more discriminative. Our findings support earlier results indicating the random

12. The K2 algorithm allows limiting the number of parents for each node, but as the objective was to test this algorithm as a GBN, this restriction was not imposed.

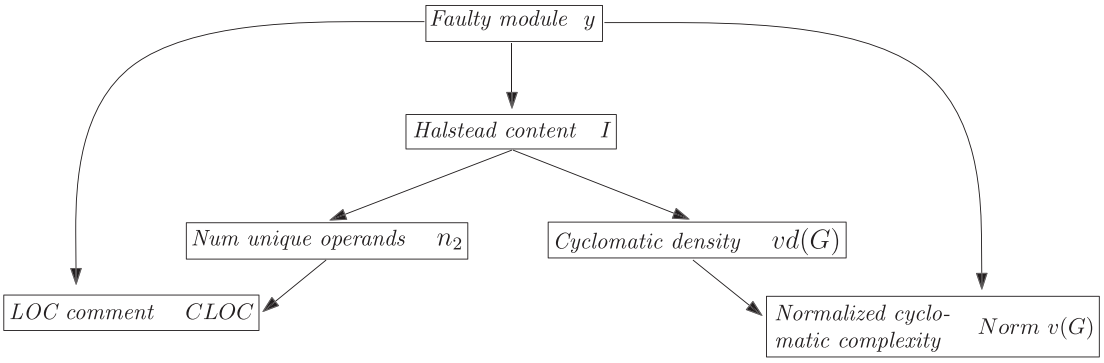


Fig. 10. Bayesian network learned by the Augmented Naive Bayes classifier "STAND LCV_LO" without MB feature selection on the PC1 dataset.

forest learner to be the most appropriate to model the presence of faults if the cost of not detecting faults outweighs the additional testing effort. In the opposite situation, Augmented Bayesian Network classifiers are found to be the better choice. The question of how other techniques such as support vector machines or neural networks perform under these circumstances remains to be explored.

Recently, several researchers turned their attention to another topic of interest, i.e., the inclusion of information other than static code features into fault prediction models such as information on intermodule relations [98] and requirement metrics [57]. The relation to the more commonly used static code features remains however unclear. Using, e.g., Bayesian network learners, important insights into these different information sources could be gained which is left as a topic for future research.

ACKNOWLEDGMENTS

This research was supported by the Odysseus program (Flemish Government, FWO) under grant G.0915.09.

REFERENCES

[1] S. Ali and K. Smith, "On Learning Algorithm Selection for Classification," *Applied Soft Computing,* vol. 6, no. 2, pp. 119-138, 2006.

[2] C. Aliferis, A. Statnikov, I. Tsamardinos, S. Mani, and X. Koutsoukos, "Local Causal and Markov Blanket Induction for Causal Discovery and Feature Selection for Classification Part I: Algorithms and Empirical Evaluation," *The J. Machine Learning Research,* vol. 11, pp. 171-234, 2010.

[3] C. Aliferis, I. Tsamardinos, and A. Statnikov, "HITON: A Novel Markov Blanket Algorithm for Optimal Variable Selection," *Proc. AMIA Ann. Symp.,* 2003.

[4] E. Arisholm and L. Briand, "Predicting Fault-Prone Components in a Java Legacy System," *Proc. ACM/IEEE Int'l Symp. Empirical Software Eng.,* 2006.

[5] I. Askira-Gelman, "Knowledge Discovery: Comprehensibility of the Results," *Proc. 31st Ann. Hawaii Int'l Conf. System Sciences,* vol. 5, pp. 247-256, 1998.

[6] D. Azar and J. Vybihal, "An Ant Colony Optimization Algorithm to Improve Software Quality Prediction Models: Case of Class Stability," *Information and Software Technology,* vol. 53, pp. 388-393, 2011.

[7] B. Baesens, T. Van Gestel, S. Viaene, M. Stepanova, J. Suykens, and J. Vanthienen, "Benchmarking State-of-the-Art Classification Algorithms for Credit Scoring," *J. Operational Research Soc.,* vol. 54, no. 6, pp. 627-635, 2003.

[8] E. Baisch and T. Liedtke, "Comparison of Conventional Approaches and Soft-Computing Approaches for Software Quality Prediction," *Proc. IEEE Int'l Conf. Systems, Man, and Cybernetics,* vol. 2, pp. 1045-1049, 1997.

[9] M. Baojun, K. Dejaeger, J. Vanthienen, and B. Baesens, "Software Defect Prediction Based on Association Rule Classification," *Proc. Int'l Conf. Electronic-Business Intelligence,* pp. 396-402, 2010.

[10] B. Boehm, "A View of 20th and 21st Century Software Engineering," *Proc. 28th Int'l Conf. Software Eng.,* pp. 12-29, 2006.

[11] B. Boehm and P. Papaccio, "Understanding and Controlling Software Costs," *IEEE Trans. Software Eng.,* vol. 14, no. 10, pp. 1462-1477, Oct. 1988.

[12] L. Breiman, "Random Forests," *Machine Learning,* vol. 45, no. 1, pp. 5-32, 2001.

[13] E. Castillo, J. Gutiérrez, and A. Hadi, *Expert Systems and Probabilistic Network Models.* Springer Verlag, 1997.

[14] C. Catal, "Software Fault Prediction: A Literature Review and Current Trends," *Expert Systems with Applications,* vol. 38, pp. 4626-4636, 2011.

[15] C. Catal and B. Diri, "Investigating the Effect of Dataset Size, Metrics Sets, and Feature Selection Techniques on Software Fault Prediction Problem," *Information Sciences,* vol. 179, no. 8, pp. 1040-1058, 2009.

[16] C. Catal and B. Diri, "A Systematic Review of Software Fault Prediction Studies," *Expert Systems with Applications,* vol. 36, no. 4, pp. 7346-7354, 2009.

[17] C. Catal, U. Sevim, and B. Diri, "Practical Development of an Eclipse-Based Software Fault Prediction Tool Using Naive Bayes Algorithm," *Expert Systems with Applications,* vol. 38, pp. 2347-2353, 2011.

[18] J. Cheng, R. Greiner, J. Kelly, D. Bell, and W. Liu, "Learning Bayesian Networks from Data: An Information-Theory Based Approach," *Artificial Intelligence,* vol. 137, pp. 43-90, 2002.

[19] D. Chickering, "Optimal Structure Identification with Greedy Search," *J. Machine Learning Research,* vol. 3, pp. 507-554, 2002.

[20] D. Chickering, C. Meek, and D. Heckerman, "Large-Sample Learning of Bayesian Networks Is NP-Hard," *J. Machine Learning Research,* vol. 5, pp. 1287-1330, 2004.

[21] S. Chidamber and C. Kemerer, "A Metrics Suite for Object-Oriented Design," *IEEE Trans. Software Eng.,* vol. 20, no. 6, pp. 476-493, June 1994.

[22] C. Chow and C. Liu, "Approximating Discrete Probability Distributions with Dependence Trees," *IEEE Trans. Information Theory,* vol. 14, no. 3, pp. 462-467, May 1968.

[23] G. Cooper and E. Herskovits, "A Bayesian Method for the Induction of Probabilistic Networks from Data," *Machine Learning,* vol. 9, pp. 309-347, 1992.

[24] V. Dallmeier and T. Zimmermann, "Extraction of Bug Localization Benchmarks from History," *Proc. IEEE/ACM 22nd Int'l Conf. Automated Software Eng.,* pp. 433-436, 2007.

[25] K. Dejaeger, W. Verbeke, D. Martens, and B. Baesens, "Data Mining Techniques for Software Effort Estimation: A Comparative Study," *IEEE Trans. Software Eng.,* vol. 38, no. 2, pp. 375-397, Mar./Apr. 2011.

[26] J. Demšar, "Statistical Comparison of Classifiers over Multiple Data Sets," *J. Machine Learning Research,* vol. 7, pp. 1-30, 2006.

[27] P. Domingos, "The Role of Occam's Razor in Knowledge Discovery," *Data Mining and Knowledge Discovery,* vol. 3, no. 4, pp. 409-425, 1999.

[28] P. Domingos and M. Pazzani, "On the Optimality of the Simple Bayesian Classifier under Zero-One Loss," *Machine Learning,* vol. 29, pp. 103-130, 1997.

[29] R. Duda and P. Hart, *Pattern Classification and Scene Analysis.* John Wiley, 1973.

[30] O.J. Dunn, "Multiple Comparisons among Means," *J. Am. Statistical Assoc.,* vol. 56, pp. 52-64, 1961.

[31] K. Elish and M. Elish, "Predicting Defect-Prone Software Modules Using Support Vector Machines," *J. Systems and Software,* vol. 81, no. 5, pp. 649-660, 2008.

[32] M. Evett, T. Khoshgoftaar, P. Chien, and E. Allen, "GP-Based Software Quality Prediction," *Proc. Third Ann. Conf. Genetic Programming,* pp. 60-65, 1999.

[33] T. Fawcett, "An Introduction to ROC Analysis," *Pattern Recognition Letters,* vol. 27, no. 8, pp. 861-874, 2006.

[34] U. Fayyad and K. Irani, "Multi-Interval Discretization of Continuous-Valued Attributes for Classification Learning," *Proc. Int'l Joint Conf. Uncertainty in Artificial Intelligence,* pp. 1022-1027, 1993.

[35] N. Fenton and M. Neil, "Software Metrics: Successes, Failures and New Directions," *J. Systems and Software,* vol. 47, nos. 2/3, pp. 149-157, 1999.

[36] N. Fenton and M. Neil, "A Critique of Software Defect Prediction Models," *IEEE Trans. Software Eng.,* vol. 25, no. 5, pp. 675-689, Sept./Oct. 1999.

[37] N. Fenton and S. Pfleeger, *Software Metrics: A Rigorous & Practical Approach.* PWS Publishing Company, 1998.

[38] M. Fischer, M. Pinzger, and H. Gall, "Populating a Release History Database from Version Control and Bug Tracking Systems," *Proc. Int'l Conf. Software Maintenance,* 2003.

[39] P. Flach, J. Hernández-Orallo, and C. Ferri, "A Coherent Interpretation of AUC as a Measure of Aggregated Classification Performance," *Proc. 28th Int'l Conf. Machine Learning,* 2011.

[40] N. Friedman, D. Geiger, and M. Goldszmidt, "Bayesian Network Classifiers," *Machine Learning,* vol. 29, pp. 131-163, 1997.

[41] S. Goedertier, J. De Weerdt, D. Martens, J. Vanthienen, and B. Baesens, "Process Discovery in Event Logs: An Application in the Telecom Industry," *Applied Soft Computing,* vol. 11, no. 2, pp. 1697-1710, 2011.

[42] S. Gokhale, "Architecture-Based Software Reliability Analysis: Overview and Limitations," *IEEE Trans. Dependable and Secure Computing,* vol. 4, no. 1, pp. 32-40, Jan.-Mar. 2007.

[43] I. Gondra, "Applying Machine Learning to Software Fault-Proneness Prediction," *J. Systems and Software,* vol. 81, pp. 186-195, 2008.

[44] J. Goodenough and S. Gerhart, "Toward a Theory of Test Data Selection," *IEEE Trans. Software Eng.,* vol. 1, no. 2, pp. 156-173, Mar. 1975.

[45] L. Guo, Y. Ma, B. Cukic, and H. Singh, "Robust Prediction of Fault-Proness by Random Forests," *Proc. 15th Int'l Symp. Software Reliability Eng.,* 2004.

[46] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction," *IEEE Trans. Software Eng.,* vol. 31, no. 10, pp. 897-910, Oct. 2005.

[47] M. Halstead, *Elements of Software Science.* Elsevier, 1977.

[48] D. Hand, "Measuring Classifier Performance: A Coherent Alternative to the Area under the ROC Curve," *Machine Learning,* vol. 77, no. 1, pp. 103-123, 2009.

[49] D. Hand and K. Yu, "Idiot's Bayes Not So Stupid After All?" *Int'l Statistical Rev.,* vol. 69, no. 3, pp. 385-398, 2001.

[50] M. Harrold, "Testing: A Roadmap," *Proc. Conf. Future of Software Eng.,* pp. 61-72, 2000.

[51] D. Heckerman, D. Geiger, and D. Chickering, "Learning Bayesian Networks: The Combination of Kknowledge and Statistical Data," *Machine Learning,* vol. 20, pp. 194-243, 1995.

[52] R. Holte, "Very simple Classification Rules Perform Well on Most Commonly Used Datasets," *Machine Learning,* vol. 11, no. 1, pp. 63-90, 1993.

[53] J. Hudepohl, S. Aud, T. Khoshgoftaar, E. Allen, and J. Mayrand, "EMERALD: Software Metrics on the Desktop," *IEEE Software,* vol. 13, no. 5, pp. 56-60, Sept. 1996.

[54] 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 Systems," *Decision Support Systems,* vol. 51, no. 1, pp. 141-154, 2011.

[55] Y. Jiang and B. Cukic, "Misclassification Cost-Sensitive Fault Prediction Models," *Proc. Fifth Int'l Conf. Predictor Models in Software Eng.,* 2009.

[56] Y. Jiang, B. Cukic, and Y. Ma, "Techniques for Evaluating Fault Prediction Models," *Empirical Software Eng.,* vol. 13, pp. 561-595, 2008.

[57] Y. Jiang, B. Cukic, and T. Menzies, "Fault Prediction Using Early Lifecycle Data," *Proc. 18th IEEE Int'l Symp. Software Reliability,* pp. 237-246, 2007.

[58] G. John and P. Langley, "Estimating Continuous Distributions in Bayesian Classifiers," *Proc. 11th Conf. Uncertainty in Artificial Intelligence,* pp. 338-345, 1995.

[59] M. Jørgensen and K. Moløkken-Østvold, "How Large Are Software Cost Overruns? A Review of the 1994 CHAOS Report," *Information and Software Technology,* vol. 48, pp. 297-301, 2006.

[60] T. Kamiya, S. Kusumoto, and K. Inoue, "Prediction of Fault-Proneness at Early Phase in Object-Oriented Development," *Proc. Second IEEE Int'l Symp. Object-Oriented Real-Time Distributed Computing,* 1999.

[61] S. Kanmani, V. Uthariaraj, V. Sankaranarayanan, and P. Thambidurai, "Object-Oriented Software Fault Prediction Using Neural Networks," *Information and Software Technology,* vol. 49, pp. 483-492, 2007.

[62] T. Khoshgoftaar, E. Allen, and J. Deng, "Using Regression Trees to Classify Fault-Prone Software Modules," *IEEE Trans. Reliability,* vol. 51, no. 4, pp. 455-462, Dec. 2002.

[63] T. Khoshgoftaar and N. Seliya, "Tree-Based Software Quality Estimation Models for Fault Prediction," *Proc. IEEE Eighth Symp. Software Metrics,* pp. 203-214, 2002.

[64] E. Kocaguneli, A. Tosun, B. Turhan, and B. Caglayan, "Prest: An Intelligent Software Metrics Extraction, Analysis and Defect Prediction Tool," *Proc. Int'l Conf. Software Eng. and Knowledge Eng.,* 2009.

[65] I. Kononenko, "Semi-Naive Bayesian Classifier," *Proc. Sixth European Working Session on Learning,* pp. 206-219, 1991.

[66] S. Kotsiantis, S. Zaharakis, and P. Pintelas, "Supervised Machine Learning: A Review of Classification Techniques," *Informatica,* vol. 31, pp. 249-268, 2007.

[67] P. Langley and S. Sage, "Induction of Selective Bayesian Classifiers," *Proc. 10th Conf. Uncertainty in Artificial Intelligence,* D.P.R. Lopez de Mantaras, ed., pp. 399-406, 1994.

[68] P. Larranaga, C. Kuijpers, R. Murga, and Y. Yurramendi, "Learning Bayesian Network Structures by Searching for the Best Ordering with Genetic Algorithms," *IEEE Trans. Systems, Man, and Cybernetics, Part A: Systems and Humans,* vol. 26, no. 4, pp. 487-493, July 1996.

[69] E. Lehman and H. D'Abrera, *Nonparametrics-Statistical Methods Based on Ranks.* Holden-Day, 1975.

[70] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings," *IEEE Trans. Software Eng.,* vol. 34, no. 4, pp. 485-496, July/Aug. 2008.

[71] H. Li and W. Cheung, "An Empirical Study of Software Metrics," *IEEE Trans. Software Eng.,* vol. 13, no. 6, pp. 697-708, June 1987.

[72] S. Mahmood, R. Lai, Y. Soo Kim, J. Hong Kim, S. Cheon Park, and H. Suk Oh, "A Survey of Component Based System Quality Assurance and Assessment," *Information and Software Technology,* vol. 47, no. 10, pp. 693-707, 2005.

[73] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the Conceptual Cohesion of Classes for Fault Prediction in Object-Oriented Systems," *IEEE Trans. Software Eng.,* vol. 34, no. 2, pp. 287-300, Mar./Apr. 2008.

[74] T. Menzies, J. Greenwald, and A. Frank, "Data Mining Static Code Attributes to Learn Defect Predictors," *IEEE Trans. Software Eng.,* vol. 32, no. 11, pp. 2-13, Nov. 2007.

[75] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect Prediction From Static Code Features: Current Results, Limitations, New Approaches," *Automated Software Eng.,* pp. 1-33, 2010.

[76] P.B. Nemenyi, "Distribution-Free Multiple Comparisons," PhD dissertation, Princeton Univ., 1963.

[77] T. Ostrand, E. Weyuker, and R. Bell, "Where the Bugs Are," *ACM SIGSOFT Software Eng. Notes,* vol. 29, no. 4, pp. 86-96, 2004.

[78] T. Ostrand, E. Weyuker, and R. Bell, "Predicting the Location and Number of Faults in Large Software Systems," *IEEE Trans. Software Eng.,* vol. 31, no. 4, pp. 340-355, Apr. 2005.

[79] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks for Plausible Inference.* Morgan Kaufmann, 1988.

[80] T.-S. Quah and M. Thwin, "Application of Neural Networks for Software Quality Prediction Using Object-Oriented Metrics," *Proc. Int'l Conf. Software Maintenance,* 2003.
[81] E. Raymond, "The Cathedral and the Bazaar," *Knowledge, Technology and Policy,* vol. 12, no. 3, pp. 23-49, 1999.
[82] W. Royce, "Managing the Development of Large Software Systems," *Proc. IEEE WESCON,* pp. 1-9, 1970.
[83] J. Sacha, "New Synthesis of Bayesian Network Classifiers and Cardiac SPECT Image Interpretation," PhD. dissertation, Univ. Toledo, 1999.
[84] V. Sessions and M. Valtorta, "Towards a Method for Data Accuracy Assessment Utilizing a Bayesian Network Learning Algorithm," *J. Data and Information Quality,* vol. 1, no. 3, pp. 1-34, 2009.
[85] M. Shepperd, "A Critique of Cyclomatic Complexity as a Software Metric," *Software Eng. J.,* vol. 3, no. 2, pp. 30-36, 1988.
[86] M. Shepperd and D. Ince, "A Critique of Three Metrics," *J. Systems and Software,* vol. 26, no. 3, pp. 197-210, 1994.
[87] S. Sherer, "Software Fault Prediction," *J. Systems and Software,* vol. 29, no. 2, pp. 97-105, 1995.
[88] F. Shull, V. Basili, B.B.A. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz, "What We Have Learned about Fighting Defects," *Proc. Eighth IEEE Symp. Software Metrics,* pp. 249-258, 2002.
[89] P. Spirtes, C. Glymour, and R. Scheines, *Causation, Prediction, and Search,* second ed. The MIT Press, 2000.
[90] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining.* Pearson Addison Wesley, 2006.
[91] B. Todd and R. Stamper, "The Relative Accuracy of a Variety of Medical Diagnostic Programs," *Methods of Information in Medicine,* vol. 33, pp. 402-416, 1994.
[92] P. Tomaszewski, J. Hakansson, H. Grahn, and L. Lundberg, "Statistical Models vs. Expert Estimation for Fault Prediction in Modified Code-An Industrial Case Study," *J. Systems and Software,* vol. 80, no. 8, pp. 1227-1238, 2007.
[93] A. Tosun, A. Bener, and B. Turhan, "An Industrial Case Study of Classifier Ensembles for Locating Software Defects," *Software Quality J.,* pp. 1-22, 2011.
[94] A. Tosun, B. Turhan, and A. Bener, "Validation of Network Measures as Indicators of Defective Modules in Software Systems," *Proc. Fifth Int'l Conf. Predictor Models in Software Eng.,* 2009.
[95] I. Tsamardinos, L. Brown, and C. Aliferis, "The Max-Min Hill-Climbing Bayesian Network Structure Learning Algorithm," *Machine Learning,* vol. 65, no. 1, pp. 31-78, 2006.
[96] B. Turhan and A. Bener, "Software Defect Prediction: Heuristics for Weighted Naive Bayes," *Proc. Second Int'l Conf. Software and Data Technologies,* pp.244-249, 2007.
[97] B. Turhan and A. Bener, "Analysis of Naive Bayes' Assumptions on Software Fault Data: An Empirical Study," *Data & Knowledge Eng.,* vol. 68, no. 2, pp. 278-290, 2009.
[98] B. Turhan, G. Kocak, and A. Bener, "Software Defect Prediction Using Call Graph Based Ranking (CGBR) Framework," *Proc. 34th Euromicro Conf. Software Eng. and Advanced Applications,* 2008.
[99] B. Turhan, T. Menzies, A. Bener, and J. Di Stefano, "On the Relative Value of Cross-Company and Within-Company Data for Defect Prediction," *Empirical Software Eng.,* vol. 14, no. 5, pp. 540-578, 2009.
[100] O. Vandecruys, D. Martens, B. Baesens, C. Mues, M. De Backer, and R. Haesen, "Mining Software Repositories for Comprehensible Software Fault Prediction Models," *J. Systems and Software,* vol. 81, no. 5, pp. 823-839, 2008.
[101] W. Verbeke, D. Martens, C. Mues, and B. Baesens, "Building Comprehensible Customer Churn Prediction Models with Advanced Rule Induction Techniques," *Expert Systems with Applications,* vol. 38, pp. 2354-2364, 2011.
[102] T. Verbraken, W. Verbeke, and B. Baesens, "Profit Optimizing Customer Churn Prediction with Bayesian Network Classifiers," *Intelligent Data Analysis,* vol. In press, 2011.
[103] I. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques.* Morgan Kaufmann Publishers, 2005.
[104] X. Wu, V. Kumar, R. Quinlan, J. Gosh, Q. Yang, H. Motoda, G. McLachlan, A. Ng, B. Liu, P. Yu, Z.-H. Zhou, M. Steinbach, D. Hand, and D. Steinbach, "Top 10 Algorithms in Data Mining," *Knowledge and Information Systems,* vol. 14, pp. 1-37, 2008.
[105] J. Yu, V. Smith, P. Wang, A. Hartemink, and E. Jarvis, "Using Bayesian Network Inference Algorithms to Recover Molecular Genetic Regulatory Networks," *Proc. Third Int'l Conf. Systems Biology,* 2002.
[106] X. Yuan, T. Khoshgoftaar, E. Allen, and K. Ganesan, "An Application of Fuzzy Clustering to Software Quality Prediction," *Proc. Third IEEE Symp. Application-Specific Systems and Software Eng. Technology,* pp. 85-90, 2000.
[107] H. Zhang, "On the Distribution of Software Faults," *IEEE Trans. Software Eng.,* vol. 34, no. 2, pp. 301-302, Mar./Apr. 2008.
[108] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-Project Defect Prediction," *Proc. Symp. Foundations of Software Eng.,* 2009.
[109] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting Defects for Eclipse," *Proc. Third Int'l Workshop Predictor Models in Software Eng.,* 2007.

Karel Dejaeger received the graduate degree in 2009 as a business engineer from the Department of Decision Sciences and Information Management at the Katholieke Universiteit Leuven (KU Leuven), Belgium, where he is a doctoral researcher. His research interests include data mining, fraud detection, and software engineering. His work has been published in the *IEEE Transactions on Software Engineering*, *European Journal of Operational Research*, and *Decision Support Systems.*

Thomas Verbraken received the MSc degree in civil engineering from the Katholieke Univsiteit Leuven (KU Leuven), Belgium, in 2007, and in 2011, he completed all three levels of the CFA program. Since 2010, he has been working toward the PhD degree from the Faculty of Business and Economics, Department of Decision Sciences and Information Management at KU Leuven. Being a member of the Research Center for Management Informatics (LIRIS), his main research focuses on data mining for business applications. More specifically, he has been working on customer churn prediction, Bayesian network classifiers, profit-based classification performance measurement, and classification in network environments.

Bart Baesens is an associate professor at the Katholieke Universiteit Leuven, Belgium, and a lecturer at the University of Southampton, United Kingdom. He has done extensive research on predictive analytics, data mining, customer relationship management, fraud detection, and credit risk management. His findings have been published in well-known international journals and presented at top international conferences. He is also coauthor of the book *Credit Risk Management: Basic Concepts*, published in 2008.