Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 81 (2008) 823–839
The Journal of Systems and Software
www.elsevier.com/locate/jss

Mining software repositories for comprehensible software fault prediction models

Olivier Vandecruys [a], David Martens [a,*], Bart Baesens [a,b,c], Christophe Mues [b], Manu De Backer [a], Raf Haesen [a]

[a] Department of Decision Sciences and Information Management, Naamsestraat 69, B-3000 Leuven, Belgium
[b] School of Management, University of Southampton, Highfield Southampton, SO17 1BJ, United Kingdom
[c] Vlerick Management School, Reep 1, B-9000 Gent, Belgium

Received 6 June 2007; received in revised form 30 July 2007; accepted 30 July 2007
Available online 28 August 2007

Abstract

Software managers are routinely confronted with software projects that contain errors or inconsistencies and exceed budget and time limits. By mining software repositories with comprehensible data mining techniques, predictive models can be induced that offer software managers the insights they need to tackle these quality and budgeting problems in an efficient way. This paper deals with the role that the Ant Colony Optimization (ACO)-based classification technique AntMiner+ can play as a comprehensible data mining technique to predict erroneous software modules. In an empirical comparison on three real-world public datasets, the rule-based models produced by AntMiner+ are shown to achieve a predictive accuracy that is competitive to that of the models induced by several other included classification techniques, such as C4.5, logistic regression and support vector machines. In addition, we will argue that the intuitiveness and comprehensibility of the AntMiner+ models can be considered superior to the latter models.

Keywords: Classification; Software mining; Fault prediction; Comprehensibility; Ant Colony Optimization

1. Introduction

In recent decades we have witnessed an explosion of data. Although much information is available in this data, it is hidden in the vast collection of raw data. Data mining entails the overall process of extracting knowledge from this data. Data mining techniques have been successfully applied in many different domains. Well-known examples are breast-cancer detection in the biomedical sector, market basket analysis in the retail sector (Berry and Linoff, 2004), and credit scoring (Baesens et al., 2003b; Higgins, 2003) in the financial sector. This paper however focuses on the rapidly evolving domain of *software mining*, i.e. the use of data mining to support and improve the development of software.

A common assumption in the software engineering literature is that the development of software is a process that is subject to inherent laws. However, since software is not a tangible product, the nature and size of the factors that influence this process are hard to establish. Therefore, software managers are routinely confronted with software projects that contain errors or inconsistencies and exceed budget and time limits. Software mining aims to tackle some of these issues by extracting knowledge from past project data that may be applied to future projects or development stages.

This paper mainly focuses on the prediction of errors in software modules by the use of data mining techniques. This enables software managers to focus their testing activities on those software modules that are classified as

* Corresponding author.
E-mail addresses: David.Martens@econ.kuleuven.be (D. Martens), Bart.Baesens@econ.kuleuven.be, B.M.M.Baesens@soton.ac.uk (B. Baesens), C.Mues@soton.ac.uk (C. Mues), Manu.Debacker@econ.kuleuven.be (M. De Backer), Raf.Haesen@econ.kuleuven.be (R. Haesen).

0164-1212/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2007.07.034

fault-prone. Clearly, the predictive accuracy of the induced classification models is important. However, in order to reveal the how and why of faults in software projects, and to boost confidence in the reliability of the models produced, the data mining technique's ability to produce a comprehensible model is at least of equal importance.

Our approach, AntMiner+, which uses Ant Colony Optimization (ACO) to infer rules from the data, takes into account the importance of both accuracy and comprehensibility, and has so far been successfully applied to the credit scoring domain (Martens et al., 2007). A further advantage of AntMiner+ is the possibility to incorporate domain knowledge (Martens et al., 2006b), ensuring intuitive decision support models. To summarize, AntMiner+ induces accurate, comprehensible and intuitive classification models. In this paper, we introduce its novel application to software mining, more specifically, to the problem of predicting faults in software modules.

The remainder of this paper is structured as follows. First, in Section 2, the domain of software mining is introduced by means of a literature study of recent software mining papers. Then in Section 3, as a preamble to Section 4, we explain the workings of our AntMiner+ algorithm. In Section 4 we apply AntMiner+ for software mining purposes and discuss the setup and results of our experiments. The final section concludes the paper and sets out some interesting issues for future research.

2. Software mining

As software mining entails the use of data mining to support and improve the development of software, we will first shortly discuss the principles of data mining.

2.1. Data mining

Data mining entails the overall process of extracting knowledge from large amounts of data. Different types of data mining are discussed in the literature (see a.o. Baesens, 2003), such as regression, classification and associations. Regression and classification are predictive data mining tasks, where the target variable to predict is continuous and discrete respectively. Association rule mining is a descriptive data mining task and aims at inferring frequently accruing patterns. The main task of interest in this paper is classification, which is the task of assigning a data point to a predefined class or group according to its predictive characteristics. The result of a classification technique is a model

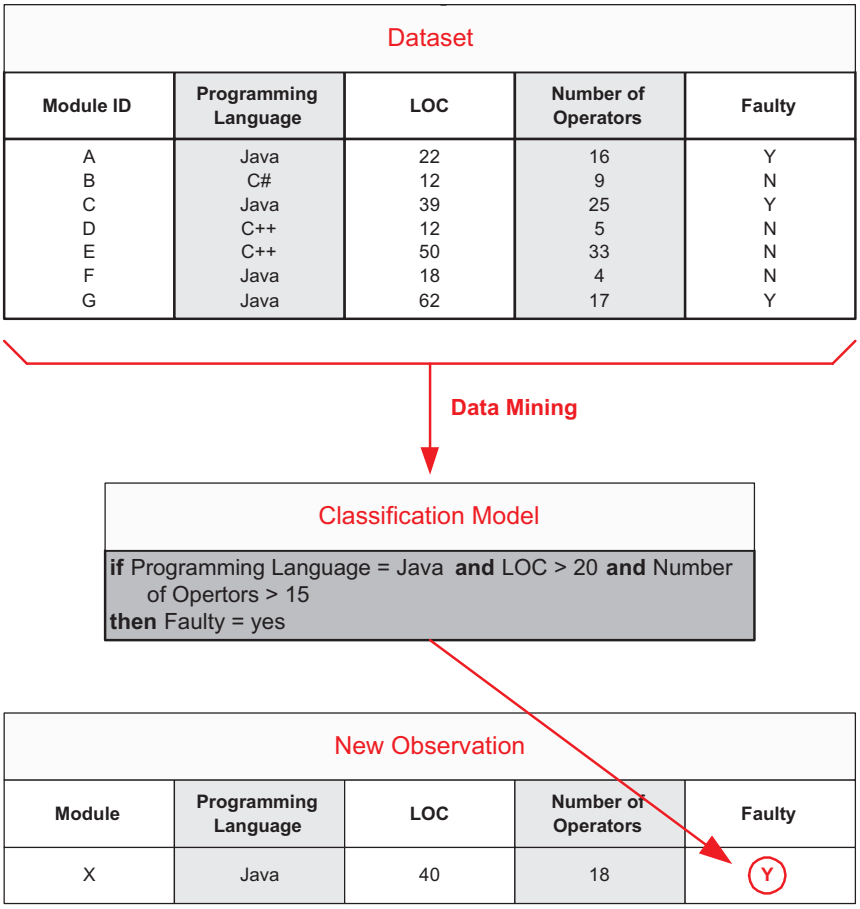

Fig. 1. Building classification models from data with data mining.

which makes it possible to classify future data points based on a set of specific characteristics in an automated way, as described in a simplified software fault prediction example in Fig. 1. In the literature, there is a myriad of different techniques proposed for this classification task, some of the most commonly used being C4.5, logistic regression, linear and quadratic discriminant analysis, *k*-nearest neighbor, Artificial Neural Networks (ANN) and Support Vector Machines (SVM) (Baesens, 2003; Hand, 2002).

Classification techniques are often applied for credit scoring (Baesens et al., 2003a; Baesens et al., 2003b; Thomas et al., 2002), medical diagnostic, such as for the prediction of dementia (Pazzani et al., 2001), classifying a breast mass as benign or malignant and selecting the best in-vitro fertilized embryo (Passmore et al., 2003). Many other data mining applications have been put forward recently, such as the use of data mining for bio-informatics (Huysmans et al., 2005), marketing and election campaigns (Hand, 2006) and counter-terrorism (Seifert, 2006).

The generated classification model has to fulfill several requirements in order to be acceptable for implementation. Accuracy is the most straightforward performance requirement for classification models, but comprehensibility of the generated model is of key importance as well in domains where the predictive model needs to be validated by an expert, such as for credit scoring and medical diagnosis. Justifiability concerns the extent to which the induced model is in line with existing domain knowledge, and is crucial as well. As the ant-based classification technique, AntMiner+, is able to generate such accurate, comprehensible and justifiable classification models (Martens et al., 2007; Martens et al., 2006a; Martens et al., 2006b), this technique is used to induce a model predicting faulty modules.

2.2. Software mining

In recent papers dedicated to the subject of software mining we were able to distinguish three common areas of interest, i.e. fault prediction, the use of change histories to detect incomplete changes and effort prediction. We summarized these papers in Table 1, categorizing them into these three areas of interest, each of which refers to some specific problem encountered in the software engineering domain.

2.3. Problems encountered in software development

2.3.1. Faults in software projects

Companies nowadays are mainly focusing on delivering quality to their customers. Quality leads towards customer satisfaction, which in turn leads towards customer binding and greater revenues in the long term. Software companies are no exception. One of the key elements that determine the end user's perception of software quality is the degree to which the software system is free of bugs (Dick et al., 2003).

Therefore, delivering error-free software products to the end user has become a priority for each software manager. In practice it seems difficult to guarantee this kind of quality. The US Department of Defense for example loses over four billion dollars on an annual basis due to software failures (Dick et al., 2003). Moreover, a study conducted by Cusumano et al. (2004) on 104 software projects developed in different continents reveals that on average the end user reports 0.15 faults per 1000 LOC (lines of code) during the first year after the software project was delivered. For a 100000 LOC software project, this comes down to 15 faults that can not be neglected.

Higher quality software can be obtained by the introduction of a structured software development process and an intensive testing process. However, in most of the cases, testing software in an exhaustive way is not a feasible option. The number of states a software system can be in is so large that even the current generation of processors is not able to simulate them one after one. Finally, as the testing environment might differ from the environment in which the system will eventually run, the results of the testing process are not always representative for the quality the end user will experience.

2.3.2. Incomplete changes in software projects

Typically a software project is implemented by many programmers. In the specific case of open source software, these programmers are not even geographically on the same location. For example, a case study by Mockus et al. (2000), dealing with the open source Apache Server, reveals that the core of this software project, including new functionalities, is developed by approximately 15 programmers. However, much more volunteers cooperate in completing changes to the source code as a result of problem reports. A programmer that changes one module, possibly does not foresee the impact on other modules that should also be changed, and as such makes an incomplete change.

The typical structure of open source software is not the only possible cause of incomplete changes to the source code. Often the source code of a software project is written in different programming languages. This makes it rather difficult to discover interdependencies between parts written in different programming languages (Ying et al., 2004), which also increases the risk of incomplete changes to the source code.

2.3.3. Inaccurate estimates of software projects

At the start of a software project, the software manager has to decide on the amount of time, people and money he or she wants to allocate to the development process. In practice this decision seems far from straightforward as the CHAOS Report by The Standish Group (2004) reveals that 53% of software projects are either late, too expensive or delivered with less than the required functionality. This proves there is a need for structural models to increase the accuracy of the estimates. A

Table 1
Literature table summarizing related software mining research in recent years

Title	Authors	Year	What?	Technique	Data collection (1) = Own data (2) = Public data	Size
Fault prediction						
Quantitative Analysis of Faults and Failures Fenton and Ohlsson (1999) in a Complex Software System *IEEE Transactions on Software Engineering*	Fenton N.E., Ohlsson N.	1999	Empirical verification of generally accepted software engineering hypotheses	Visualisation	Data from Ericsson Telecom AB, obtained through design documents by means of the ERIMET tool (1)	Release n: 140 modules, Release n+1: 246 modules
A Critique of Software Defect Prediction Fenton and Neil (1999) Models *IEEE Transactions on Software Engineering*	Fenton N.E., Neil M.	1999	Critique on common methods for fault prediction: size and complexity metrics, multivariate analysis, multicollinearity, Goldilock's Conjecture etc.	Bayesian belief networks	N.a. (Not available)	N.a.
Predicting Fault Incidence Using Software Change History Graves et al. (1999) *IEEE Transactions on Software Engineering*	Graves T.L., Karr A.F., Marron J.S., Siy H.	1999	Comparison of statistical models to decide on which aspects of the change history of a module predict faults	Linear regression, prediction with the non-linear weighted time damp model	Data of telephone switching subsystem, obtained through registration of attributes of changes by means of the change control system SCCS (1)	80 modules, 1.5 million LOC
Data Mining in Software Metrics Databases Dick et al. (2003) *Fuzzy Sets and Systems*	Dick S., Meeks A., Last M., Bunke H., Kandel A.	2003	The use of unsupervised learning techniques for data mining, i.e. the fuzzy clustering technique	Fuzzy c-means clustering in combination with principal components analysis	Master thesis of Randy Lind and Warren DeVilbiss (University of Wisconsin-Milwaukee) (1)	MIS: 390 records, ProcSoft: 422 records, OOSoft: 562 records
Analogy-Based Practical Classification Rules for Software Quality Estimation Khoshgoftaar and Seliya (2003) *Empirical Software Engineering*	Khoshgoftaar T.M., Seliya N.	2003	The use of CBR for classifying modules	CBR, majority vote, data clustering	Data of telecommunication system, obtained through SCM system (1)	Release 1: 3649 modules, Release 2: 3981 modules, Release 3: 3541 modules, Release 4: 3978 modules
Mining Repositories to Assist in Project Planning and Resource Allocation Menzies et al. (2004) *International Workshop on Mining Software Repositories*	Menzies T., Di Stefano J.S., Cunanan C., Chapman R.M.	2004	Advocates the use of public metrics repositories	Linear regression, Rule induction techniques, J48 and ROCKY	NASA's MetricsData Program (2)	N.a.
Analyzing Software Measurement Data with Clustering Techniques Zhong et al. (2004) *IEEE Intelligent Systems*	Zhong S., Khoshgoftaar T.M., Seliya N.	2004	Use of clustering techniques to increase the quality of the collected data and to efficiently classify software modules	k-means clustering, neural gas clustering	Datasets of 2 NASA software projects (1)	JM1: 8850 modules: KC2: 520 modules
Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques Williams and Hollingsworth (2005) *IEEE Transactions on Software Engineering*	Williams C.C., Hollingsworth J.K.	2005	Improving techniques for statistical analysis to predict faults by means of historical data	N.a.	Historical data of Apache Web Server en Wine, obtained through CVS (2)	Apache Web Server:6944 CVS commits, Wine: 21.671 CVS commits
Software Field Failure Rate Prediction before Software Deployment Zhang and Pham (2006) *The Journal of Systems and Software*	Zhang Z., Pham H.	2006	Prediction of field failure rate based on system test data and previous releases	Statistical prediction model	Data of a telecommunication system (1)	N.a.

(*continued on next page*)

Table 1 (*continued*)

Title	Authors	Year	What?	Technique	Data collection (1) = Own data (2) = Public data	Size
Detecting Incomplete Changes						
Predicting Source Code Changes by Mining Change History Ying et al. (2004) *IEEE Transactions on Software Engineering*	Ying A.T.T., Murphy G.C., Ng R., Chu-Carroll M.C.	2004	The use of change histories to find unexpected associated files when changing the source code	FP-growth association analysis	Data of Mozilla and Eclipse project, obtained through CVS (2)	N.a.
Mining Version Histories to Guide Software Changes Zimmerman et al. (2005)*IEEE Transactions on Software Engineering*	Zimmermann T., Weissgerber P., Diel S., Zeller A.	2005	The use of version histories to make predictions by means of the ROSE prototype	Apriori association analysis	Data of 8 open source programs, obtained through CVS (2)	Total of 10761 transactions
Software Defect Association Mining and Defect Correction Effort Prediction Song et al. (2006) *IEEE Transactions on Software Engineering*	Song Q., Shepperd M., Cartwright M., Mair C.	2006	The use of association rule mining to predict defect associations and defect correction efforts	Apriori association analysis	Through online database SEL Data. Extraction with SQL and definition of transactions by means of sliding window (2)	More than 200 projects
Effort prediction						
On Building Prediction Systems for Software Engineers Shepperd et al. (2000) *Empirical Software Engineering*	Shepperd M., Cartwright M., Kadoda G.	2000	The use of different accuracy indicators impede useful comparisons of prediction systems. Simulation is a possible solution	Linear regression	Desharnais (2)	77 projects
Predicting with Sparse Data Shepperd and Cartwright (2001) *IEEE Transactions on Software Engineering*	Shepperd M., Cartwright M.	2001	Offering a solution to the lack of accurate, consistent and complete data for prediction purposes	Mathematical decision process	Data of 2 large telecommunication (1)	Dataset 1: 21 projects, Dataset 2: 18 projects
Combining Techniques to Optimize Effort Predictions in Software Project Management MacDonell and Shepperd (2003) *The Journal of Systems and Software*	MacDonell S.G., Shepperd M.J.	2003	Combining prediction techniques	Linear regression, CBR, Opinion of experts	Data of a medical database system(1)	77 modules
An Empirical Analysis of Software Productivity Over Time Premraj et al. (2005) *11th IEEE International Software Metrics Symposium (METRICS'05)*	Premraj R., Kitchenham B., Shepperd M., Forselius P.	2005	Research on trends in software productivity throughout time	Linear regression	Data of Software Technology Transfer Finland, obtained through Experience Pro tool (2)	602 projects
Using Grey Relational Analysis to Predict Effort with Small Data Sets Song et al. (2005) *11th IEEE International Software Metrics Symposium (METRICS'05)*	Song Q., Shepperd M., Mair C.	2005	The use of grey relational analysis for feature subset selection and software effort prediction in an early stage of software development	The machine learning technique grey relational analysis, neural networks, linear regression, decision trees	Albrecht, COCOMONASA, COCOMO81, Desharnais, Kemerer (2)	Albrecht: 19 projects, COCOMONASA: 60 projects, COCOMO81: 63 projects, Desharnais: 77 projects, Kemerer: 15 projects

well-known example is COCOMO II, a cost model that can serve as a basis for planning and executing software projects (Boehm et al., 2000). The basic formula of COCOMO II is

$$PM_{NS} = A \cdot Size^{E} \cdot \prod_{i=1}^{n} EM_i \tag{1}$$

with

$$E = B + 0.01 \cdot \sum_{i=1}^{5} SF_j \tag{2}$$

which expresses the total effort in person-months (PM). A and B are constants, $EM_i\,(i = 1,2,\ldots,n)$ are effort multipliers and $SF_j\,(j = 1,2,\ldots,5)$ are scale factors. *Size* stands

for the size of the software project (measured in or converted to thousands of source code lines). The effort multiplicators are used to incorporate the characteristics of the specific software product in the model. The scale factors take into account returns to scale related to precedentedness, team cohesion, etc.

Using the COCOMO II model is definitely a step in the right direction. The model is well structured, easy to use and offers the possibility of calibration to incorporate company-specific characteristics in the model. However, in practice there are certain pitfalls in this process of calibration one has to avoid (De Rore et al., 2006). First of all, the values of the constants A and B are often determined using trial-and-error, whereas regression techniques offer a better alternative. Moreover, the values of the scaling factors and effort multiplicators are determined in a slightly arbitrary way. Usually for this purpose question lists are filled out by the responsible persons. The quality of the resulting model then depends on the subjective answers of these persons. Finally, the question lists only offer about five possible answers for each question; each answer corresponds to a predefined value for the parameter involved. This results in a discrete spectrum for the effort multiplicators and the scale factors, which can never be as accurate as a continuous spectrum.

2.4. The function of software mining

We speak of software mining when metrics for software projects/products (such as LOC, complexity and number of source code changes) or software artifacts themselves (e.g. the source code) are used as input for data mining techniques. Software mining can be a powerful instrument for both software managers as programmers, and can play a role of importance in addressing the problems discussed in the previous paragraph.

2.4.1. An instrument for fault prediction

The end user of a software system associates software quality among other things with the software system being free of bugs. However, we already mentioned that testing software systems in an exhaustive way is infeasible. Software mining can offer a solution by modeling software quality predictions or software quality classifications (Khoshgoftaar and Seliya, 2003). In the case of software quality predictions we make use of software metrics and regression techniques to predict the number of bugs in the software module. In the case of software quality classifications, we use software metrics to classify software modules as fault-prone or not fault-prone.

Both software mining techniques allow software managers to allocate their testing budget to those modules that are most likely to contain errors, based on the results of the software mining algorithm. Clearly this approach is far more efficient than equally allocating the testing budget to the whole software system. Indeed, the latter approach would waste a considerable amount of effort on extensive testing of modules that were not likely to contain errors. Moreover, too little attention would be paid to the modules that do contain errors.

It should be clear that software mining can play a role of importance in detecting bugs, and thus in improving the quality of software systems. However, the results of software mining should be interpreted with caution. First of all, one has to realize that detecting many faults does not imply a higher reliability of the software system. The Pareto principle indeed states that a small percentage of the bugs in a software system usually causes the majority of operational errors (Fenton and Ohlsson, 1999; Juran, 1964). Secondly, software mining can not replace expert judgment as a prediction technique. Expert judgment and software mining are complementary techniques where the prediction of software quality is concerned. Therefore, Menzies et al. (2004) distinguish between primary and secondary methods for fault prediction, where expert judgment constitutes the primary method and software mining the secondary. More specifically, the secondary methods are used to efficiently verify those modules that were already excluded by the primary methods. Finally, one has to be aware that software mining to detect faults in software systems is quite different from the quality control in other industries (Dick et al., 2003). Software mining inevitably deals with non-tangible products. Therefore, the exact relationship between the measured software metrics and quality of the end product is very difficult to determine.

2.4.2. Detecting incomplete changes in software projects

By keeping change histories of software projects, one can prevent faults caused by incomplete changes to the source code. Applying an algorithm for association analysis on these change histories could generate warnings like *"Programmers who changed this function, also made changes to the functions X, Y and Z."* (Zimmerman et al., 2005). These warnings can be displayed directly to the programmers by using an association rule mining tool as a plug-in for the programming environment, that will detect frequently co-occurring changes. Such a tool can predict and suggest probable changes to the source code that can not be predicted by means of simple methods for program analysis: interdependencies between parts of the source code written in different programming languages and even between parts that do not belong to the source code itself (e.g. documentation) can be discovered.

Note that the data mining techniques, as well as the targeted user, differ for the fault prediction and the detecting of incomplete changes. Fault prediction models are induced by classification techniques and aimed at the software manager, while association rule mining is used for detecting incomplete changes and is aimed at programmers. This paper focuses on the fault prediction aspect of software mining.

2.4.3. An instrument for effort prediction

Software mining for effort prediction offers the advantages of COCOMO II without its limitations. Applying soft-

ware mining for effort prediction quickly leads towards a company-specific predictive model. However, software mining is based on mathematically supported data mining techniques. This eliminates the undesired consequences of arbitrarily defined parameters and part of the subjectivity of the COCOMO II model. Moreover, one has the continuous spectrum of real numbers at one's disposal in determining the different parameters of the predictive model. Hence it seems fair to state that using software mining to predict effort leads towards more accurate predictions and a more efficient allocation of time and money.

A remark similar to the remark on software mining for fault prediction can be made. Software mining should not replace expert judgment as a prediction technique. Expert judgment and software mining should be seen as complementary techniques. In general, MacDonell and Shepperd (2003) conclude that one should make use of different prediction techniques when estimating the effort needed for a software project because there exists no technique that always performs best. Unfortunately, MacDonell and Shepperd have not yet been able to find patterns to determine which technique performs best under which circumstances. To facilitate this kind of research, it is important that research on software mining is subject to certain customs and agreements. This is clarified in the next paragraph.

2.5. Critique on software mining experiments

When taking a detailed look at Table 1, we observe that within each problem domain of software management, except for detecting incomplete changes, different data mining techniques are used. However, making useful comparisons between these techniques becomes difficult, since many authors apply different methodological frameworks, which makes it hard to directly compare results.

In approximately half of the papers summarized in Table 1, own datasets are used, i.e. datasets that are not publicly accessible. It should be clear however that making use of public datasets offers important advantages. Research such as that of MacDonell and Shepperd (2003) on the best performing data mining technique under certain circumstances would be facilitated if everyone would make use of the same public datasets, of which the characteristics are known. This way benchmarks are created to which existing and new data mining techniques can be compared. As the experimental part of this paper will demonstrate, we attempt to accommodate these issues by including the most commonly used state-of-the-art classification techniques, and apply them to publicly available datasets.

Finally, the comprehensibility aspect of the predictive classification models is commonly overlooked. If the model is to be incorporated as a decision support model in everyday practices of the software manager, validation of the model to ensure the model is intuitive, is of great importance.

3. AntMiner+: classification based on Ant Colony Optimization

As software fault prediction models should be both accurate and comprehensible, we will focus on the use of rule-based classification techniques. More specifically, we induce sets of rules from the available software repositories with AntMiner+, a classification technique that employs artificial ants to induce rules[1]. Our previous benchmarking study reveals that the models generated by AntMiner+ fulfill both the accuracy and comprehensibility requirements (Martens et al., 2007), and are also intuitively correct (Martens et al., 2006b). In this section the main workings of this technique are explained, starting with a short introduction to the basis of AntMiner+: Ant Colony Optimization.

3.1. Ant Colony Optimization

Ant Colony Optimization (ACO) is a metaheuristic inspired on the foraging behavior of real ant colonies (Dorigo and Stützle, 2004). A biological ant by itself is a simple insect with limited capabilities, and is guided by straightforward decision rules. However, these simple rules are sufficient for the overall ant colony to find short paths from the nest to the food source. By dropping a chemical substance called pheromone that attracts other ants, an ant indirectly communicates with its fellow ants from the colony. How this indirect communication leads to shortest path finding capabilities is shown in Fig. 2. Suppose two ants start from their nest (left) and look for the shortest path to a food source (right). Initially no pheromone is present on either trails, so there is a 50–50 chance of choosing either of the two possible paths (see Fig. 2a). Suppose one ant chooses the lower trail, and the other one the upper trail. The ant that has chosen the lower (shorter) trail will have returned faster to the nest, resulting in twice as much pheromone on the lower trail as on the upper one, as illustrated in Fig. 2b. As a result, the probability that the next ant will choose the lower, shorter trail will be twice as high, resulting in more pheromone and thus more ants choosing this trail, until eventually (almost) all ants will follow the shorter path. Note that the pheromone on the longer trail will finally disappear through evaporation.

Ant Colony Optimization employs artificial ants that cooperate in a similar manner as their biological counterparts, in order to find good solutions for discrete optimization problems (Dorigo and Stützle, 2004). The first ACO algorithm is Ant System (Dorigo et al., 1991; Dorigo et al., 1996), where ants iteratively construct solutions and add pheromone to the paths corresponding to these solutions. Path selection is a stochastic procedure based

[1] http://www.econ.kuleuven.be/public/ndbaf65/antminer.html.

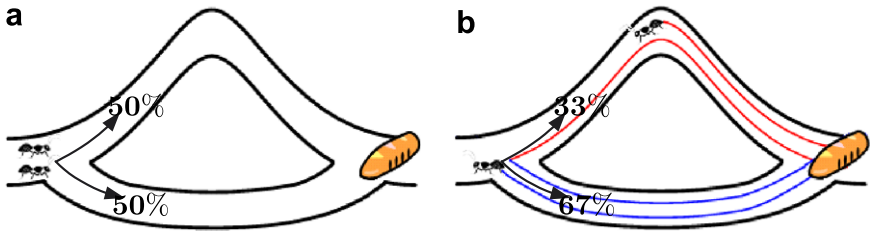

Fig. 2. Path selection directed by pheromone: the more pheromone on a path, the more likely an ant will follow the path. This simple mechanism of indirect communication is sufficient for the overall ant colony to find short paths from the nest to the food source.

on not only a history-dependent pheromone value, but also a problem-dependent heuristic value. The pheromone value gives an indication of the number of ants that chose the trail recently, while the heuristic value is a problem dependent quality measure. When an ant reaches a decision point, it is more likely to choose the trail with the higher pheromone and heuristic values. Once the ant arrives at its destination, the solution corresponding to the ant's followed path is evaluated and the pheromone value of the path is increased accordingly. Additionally, evaporation causes the pheromone level of all trails to diminish gradually. Hence, trails that are not reinforced gradually lose pheromone and will in turn have a lower probability of being chosen by subsequent ants.

The performance of traditional ACO algorithms however is rather poor on larger-scale problems (Stützle and Hoos, 1996). To overcome this issue, other ACO algorithms have been proposed, such as Ant Colony System (Dorigo and Gambardella, 1997), rank-based Ant System (Bullnheimer et al., 1999b), Elitist Ant System (Dorigo et al., 1996) and $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System (Stützle and Hoos, 2000). As the latter is the one employed in the AntMiner+ classification technique, the main features of $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System are discussed next.

Stützle and Hoos (2000) advocate that a better exploitation of the best solutions can be obtained by only adding pheromone to the path of the best ant. To avoid early search stagnation, which is the situation where all ants take the same path and thus describe the same solution, possible pheromone values are limited to the interval $[\tau_{min}, \tau_{max}]$. Finally, initializing the pheromone values to τ_{max} entails a higher exploration at the beginning of the algorithm.

ACO has been applied to a wide variety of problems (Dorigo and Stützle, 2004), such as the vehicle routing problem (Wade and Salhi, 2004; Bullnheimer et al., 1999a; Montemanni et al., 2005), scheduling (Colorni et al., 1994; Blum, 2005), timetabling (Socha et al., 2002), the traveling salesman problem (Stützle and Hoos, 2000; Dorigo et al., 1996; Gambardella and Dorigo, 1995) and routing in packet-switched networks (DiCaro and Dorigo, 1998). Recently, ACO has also entered the data mining domain, addressing both the clustering (Abraham and Ramos, 2003; Handl et al., 2006) and classification task (Parpinelli et al., 2001; Liu et al., 2002; Liu et al., 2003), which is the topic of interest in this paper. The first application of ACO to the classification task is reported by Parpinelli et al. (2001) and was named AntMiner. Extensions were put forward by Liu et al. in AntMiner2 (Liu et al., 2002) and AntMiner3 (Liu et al., 2003). Our approach, AntMiner+, differs from these previous AntMiner versions in several ways, resulting in an improved performance, as described in Martens et al. (2007). Next follows a brief discussion of the principles and workings of AntMiner+.

3.2. AntMiner+ algorithm

ACO can be used to induce comprehensible and accurate rule-based classification models from data, as done in the AntMiner+ classification technique.

First of all, an environment needs to be defined in which the ants operate. When an ant moves through the environment from *Start* to *Stop* vertex, it should incrementally construct a solution to the problem at hand, in this case the classification problem. In order to build a set of classification rules, we define the construction graph in such a way that each ant's path will implicitly describe a classification rule. For each variable V_i a vertex $v_{i,j}$ is created for each of its values $Value_{i,j}$. The set of vertices for one variable is defined as a vertex group. To allow for rules where not all variables are involved, hence shorter rules, an extra *dummy vertex* is added to each variable whose value is unspecified, meaning it can take any of the values available. Although only discrete variables are allowed, we make a distinction between nominal (no apparent ordering in its values, e.g. sex and purpose of loan) and ordinal variables (a clear ordering of the values, e.g. amount on savings or checking account and income). Each nominal variable has one vertex group (with the inclusion of the mentioned dummy vertex), but for the ordinal variables however, we build two vertex groups to allow for intervals to be chosen by the ants. The first vertex group corresponds to the lower bound of the interval and should thus be interpreted as $V_i \geqslant Value_{i,\ k}$, the second vertex group determines the upper bound, giving $V_{i+1} \leqslant Value_{i+1,l}$ (of course, the choice of the upper bound is constrained by the lower bound). This allows to have less, shorter and actually better rules. To extract a rule set that is exhaustive, such that all future data points can be classified, the majority class is not included in the vertex group of the class variable, and will be the predicted class for the final **else** clause.

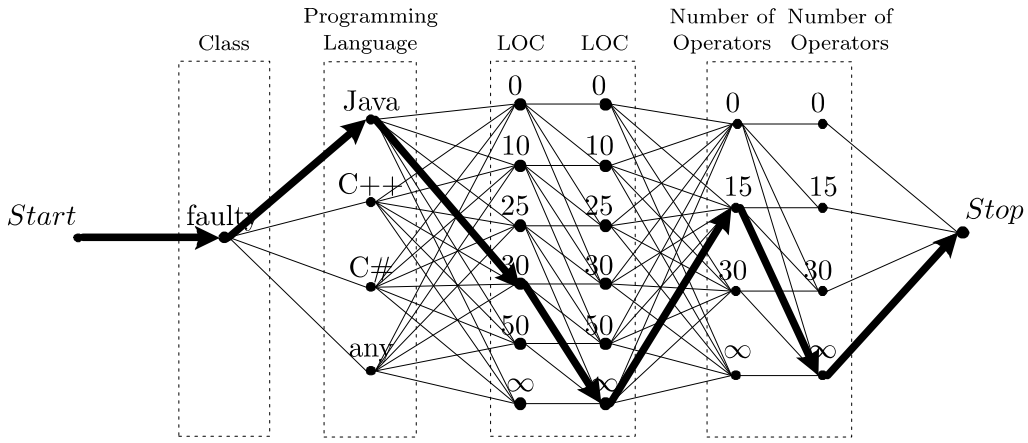

Fig. 3. Example of a path described by an ant for a software construction graph defined by AntMiner+. The rule corresponding to the chosen path is "**if** Programming Language = Java **and** LOC ∈ [30,∞) **and** Number of Operators ∈ [15,∞) **then** class = faulty".

An example AntMiner+ construction graph for a software mining dataset with only three variables (programming language, lines of code (LOC) and number of operators) is shown in Fig. 3. The path denoted in bold describes the rule "**if** Programming Language = Java **and** LOC $\geqslant$ 30 **and** Number of Operators $\geqslant$ 15 **then** class = faulty". A formal illustration of the construction graph is provided in Fig. 4, for a data set with d classes, n variables, of which the first and last variable are nominal and V_2 is ordinal (hence the two vertex groups). The weight parameters α and β determine the relative importance of the pheromone and heuristic values, and its notion is described by Eq. (3).

Now that the environment is defined, we can explain the workings of the technique, which is described in pseudo-code in Algorithm 1. All ants begin in the *Start* vertex and walk through their environment to the *Stop* vertex, gradually constructing a rule. Only the ant that describes the best rule will update the pheromone of its path, as imposed by the $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System approach. Evaporation decreases the pheromone of all edges by multiplication with ρ (a real number typically in the range of [0.8,0.99]), while the pheromone levels are constrained to lie within the given interval $[\tau_{min},\tau_{max}]$. Then another iteration occurs with ants walking from *Start* to *Stop*. Convergence occurs when all the edges of one path have a pheromone level τ_{max} and all other edges have pheromone level τ_{min}. Next, the rule corresponding to the path with τ_{max} is extracted and added to the rule set. Finally, training data covered by this rule is removed from the training set. This iterative process will be repeated until an early stopping criterion is met. Next we will have a closer look at the algorithm specifics, such as the edge probabilities and rule quality measure.

Algorithm 1. Pseudo-code of AntMiner+ algorithm

```
construct graph
while not early stopping or minimum percentage data covered do
    initialize heuristics, pheromones and probabilities of edges
    while not converged do
        create ants
        let ants run from source to sink
        evaporate pheromone on edges
        prune rule of best ant
        update path of best ant
        adjust pheromone levels if outside boundaries
        kill ants
        update probabilities of edges
    end while
    extract rule corresponding to converged path
    flag data points covered by the extracted rule
end while
evaluate performance on test set
```

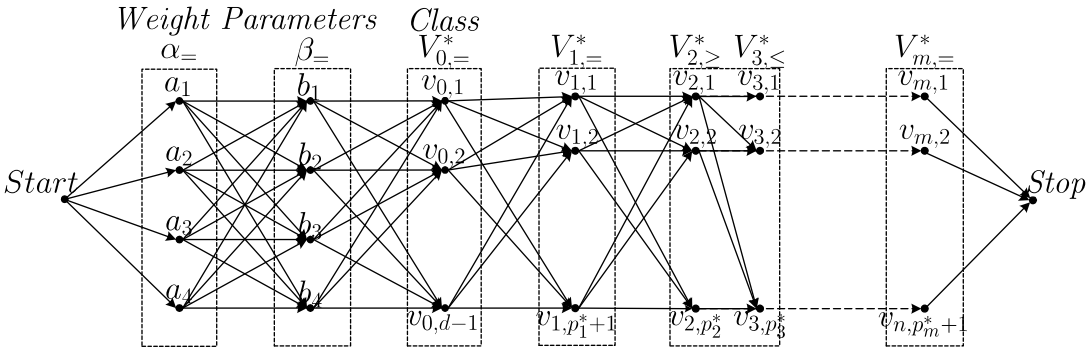

Fig. 4. Multiclass construction graph of AntMiner+, with the inclusion of weight parameters.

$$P_{ij}(t) = \frac{[\tau_{(v_{i-1,k},v_{i,j})}(t)]^{\alpha}.[\eta_{v_{i,j}}(t)]^{\beta}}{\sum_{l=1}^{p_i}[\tau_{(v_{i-1,k},v_{i,l})}(t)]^{\alpha}.[\eta_{v_{i,l}}(t)]^{\beta}} \quad (3)$$

$$\eta_{ij} = \frac{|T_{ij}\&CLASS = class_{ant}|}{|T_{ij}|} \quad (4)$$

$$\tau_{(v_{i-1,k},v_{i,j})}(0) = \tau_{max} \quad (5)$$

$$\tau_{(v_{i-1,k},v_{i,j})}(t+1) = \rho \cdot \tau_{(v_{i-1,k},v_{i,j})}(t) + \frac{Q^{+}_{best}}{10} \quad (6)$$

The edge to choose when an ant arrives at a vertex $v_{i-1,k}$, and thus the term to add next, is dependent on the pheromone value of the edge between vertices $v_{i-1,k}$ and $v_{i,j}$ ($\tau_{(v_{i-1,k},v_{i,j})}$) and the heuristic value of the vertex $v_{i,j}$ ($\eta_{i,j}$), and normalized over all possible vertices, providing a probability P_{ij} for each of the possible vertices, according to (3). As the heuristic function η is problem-dependent, we have defined the heuristic value η_{ij} of vertex $v_{i,j}$, corresponding to the term $V_i = Value_{i,j}$[2], as the fraction of training cases that are correctly covered (described) by this term, as defined by (4).

The initial pheromone value is by definition τ_{max}, as imposed by $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System. The pheromone to add to the path of the best ant should be proportional to the quality of the path, which we define as the sum of the *confidence* and the *coverage* of the corresponding rule. Confidence measures the fraction of the number of correctly classified remaining (not yet covered by any of the extracted rules) data points by a rule compared to the total number of remaining data points covered by that rule. The coverage gives an indication of the overall importance of the specific rule by measuring the number of correctly classified remaining data points over the total number of remaining data points. More formally, the pheromone amount to add to the path of the iteration best ant is given by the benefit of the path of the iteration best ant, as indicated by (7), with $rule_{ant}$ the rule antecedent (**if** part) comprising of a conjunction of terms corresponding to the path chosen by the ant, $rule^{c}_{ant}$ the conjunction of $rule_{ant}$ with the class chosen by the ant, and Cov a binary variable expressing whether a data point is already covered by one of the extracted rules ($Cov = 1$) or not ($Cov = 0$). The number of remaining data points can therefore be expressed as $|Cov = 0|$. This means that, taking into account the evaporation factor as well, the update rule for the best ant's path is described by (6), where the division by ten is a scaling factor that is needed such that both the pheromone and heuristic values lie within the range [0,1].

$$Q^{+} = \underbrace{\frac{|rule^{c}_{ant}|}{|rule_{ant}|}}_{\text{confidence}} + \underbrace{\frac{|rule^{c}_{ant}|}{|Cov = 0|}}_{\text{coverage}} \quad (7)$$

Advantages of AntMiner+ are not only the accuracy and comprehensibility of the generated models, but also the possibility to demand intuitive predictive models, which is crucial whenever comprehensibility is required. For example, when a classification rule is induced to predict whether or not a software module is faulty, the rule "**if** $LOC > 1000$ **then** class = not faulty", is an unintuitive rule, as we would expect that larger modules will be more subject to faults, making the expected sign for this example "$<$". The rule "**if** $LOC > 1000$ **then** class = faulty" on the other hand, is intuitive. By stating constraints on these inequality signs, such domain knowledge can be incorporated, resulting in intuitive, justifiable classification models. Note that the constraints are imposed by the domain expert, who might disagree with such constraints and choose to impose no or other inequality constraints.

4. Software mining with AntMiner+

4.1. Datasets

We applied AntMiner+ on three publicly available datasets of NASA software projects[3]: PC1, PC4 and KC1. Both PC1 and PC4 describe a flight software project written in C and used for an earth orbiting satellite. PC1 contains 40000 lines of code (LOC), PC4 counts 36000 LOC. KC1 describes a 43000 LOC software project written in C++ and is a subsystem of a large ground control system. All software measurements were conducted at the level of program functions, subroutines or methods. So in the remainder of this chapter, a software module refers to such a program function, subroutine or method. The number of software modules for each of these datasets, and thus the number of data instances, as well as the number of faulty and non-faulty modules are summarized in Table 2. This table clearly indicates the uneven distribution of the number of faulty versus non-faulty modules.

4.2. Data preprocessing

Data preprocessing was conducted for discretization, input selection and oversampling.

4.2.1. Discretization

Since AntMiner+ can only deal with discrete variables, a discretization step is needed to turn all continuous variables into discrete ones. This was done using SAS Enterprise Miner, with an entropy-based procedure. Note that this and other input selection procedures are also available in the open source data mining workbench Weka[4] (Witten and Frank, 2000).

4.2.2. Input selection

To make sure that only relevant variables are included in the datasets, we went through an input selection procedure

[2] Shortened as T_{ij}.

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

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

Table 2
Dataset properties, in terms of number of modules (Size), number of faulty modules (Nb faulty), and number of non-faulty modules (Nb non-faulty), for each of the datasets

	PC1	PC4	KC1
Size	1059	1347	1571
Nb faulty	76	178	319
Nb non-faulty	983	1169	1252

using a χ^2-based filter (Martens et al., 2007; Thomas et al., 2002). First, the observed frequencies of all possible combinations of values for class and variable are measured. Based on this, the theoretical frequencies, assuming complete independence between the variable and the class, are calculated. The hypothesis of equal odds provides a χ^2 test statistic; higher values allow one to more confidently reject the zero hypothesis of equal odds; hence, these values allow one to rank the variables according to predictive power. In this manner, 11 metrics were retained for PC1, 13 for PC4 and 10 for KC1, which include LOC, Halstead[5] and complexity measures, and are listed in Table 3.

Table 3
The metrics retained after input selection, for the different datasets

	PC1	PC4	KC1
LOC total	x	x	x
LOC blank	x		
LOC code and comment	x		
LOC comments	x	x	
LOC executable	x		
Halstead Error Est	x	x	x
Halstead volume	x	x	x
Halstead content	x		
Halstead length		x	x
Halstead level		x	
Halstead difficulty			x
Halstead prog time			x
Halstead effort			x
Num operators			x
Num operands		x	x
Num unique operands	x		x
Num lines	x		
Condition count		x	
Modified condition count		x	
Multiple condition count		x	
Cyclomatic complexity		x	
Normalized cylomatic complex	x	x	
Call pairs		x	

4.2.3. Oversampling

Finally, the target variable in the different datasets was heavily skewed: in all three datasets the number of erroneous software modules is much smaller than the number of correct software modules. This leads to the different classification techniques experiencing difficulties in learning when software modules are erroneous. Since software managers are mainly interested in predicting erroneous software modules, we decided to make use of oversampling to solve this problem. With oversampling, (training) observations representing erroneous software modules in the datasets are repeated several times, as shown in Fig. 5.

Depending on the number of times these observations are repeated, the resulting accuracy, specificity (percentage of faulty software projects that is correctly classified) and sensitivity (percentage of non-faulty software projects that is correctly classified) varies. Table 4 illustrates this problem, showing the results for AntMiner+ on the different datasets for different degrees of oversampling. The results for the other reviewed classification techniques are similar. For the original dataset (0 oversampling) we observe that AntMiner+ reaches a reasonable accuracy and a high sensitivity, but a fairly low specificity. This reflects the difficulties encountered in learning when software modules are erroneous. A higher degree of oversampling results in a higher specificity, but implies a declining sensitivity. Because of the larger share of correct modules in the datasets, the importance of sensitivity relative to specificity is greater in the calculation of accuracy. This means that, as the degree of oversampling increases, the accuracy decreases. However, since the cost of not detecting an error is likely to be higher than the cost of testing a correct module, this is a trade-off that one is willing to make. A reasonable trade-off between specificity on the one hand and sensitivity and accuracy on the other hand, is reached at five oversampling. At this oversampling rate, the distribution of faulty versus non-faulty modules is almost even, as there are about as much faulty as non-faulty data modules. This is reflected in the results, which show good performances both in terms of specificity and sensitivity. Therefore we decided to repeat each observation corresponding to an erroneous software module five times in the datasets.

4.3. Experimental setup

For our experiments, the classification task of AntMiner+ is to distinguish between erroneous and correct software modules. In order to do so, we have made a 70%/30% stratified split up of the original datasets into training and test datasets. Since the datasets used are relatively large, early stopping can be applied. Validation datasets are then needed and therefore one third of the training datasets is set apart for validation purposes. As for the parameters of AntMiner+, the only parameters that need to be set are the total number of ants and the evaporation factor ρ, which are initialised to 1000 and 0.85 respectively, as suggested by Martens et al. (2007).

To compare the results of AntMiner+, a benchmarking study is performed that includes commonly used state-of-the-art classification techniques C4.5, RIPPER, logistic regression, 1-nearest neighbor, support vector machine

[5] http://mdp.ivv.nasa.gov/halstead_metrics.html.

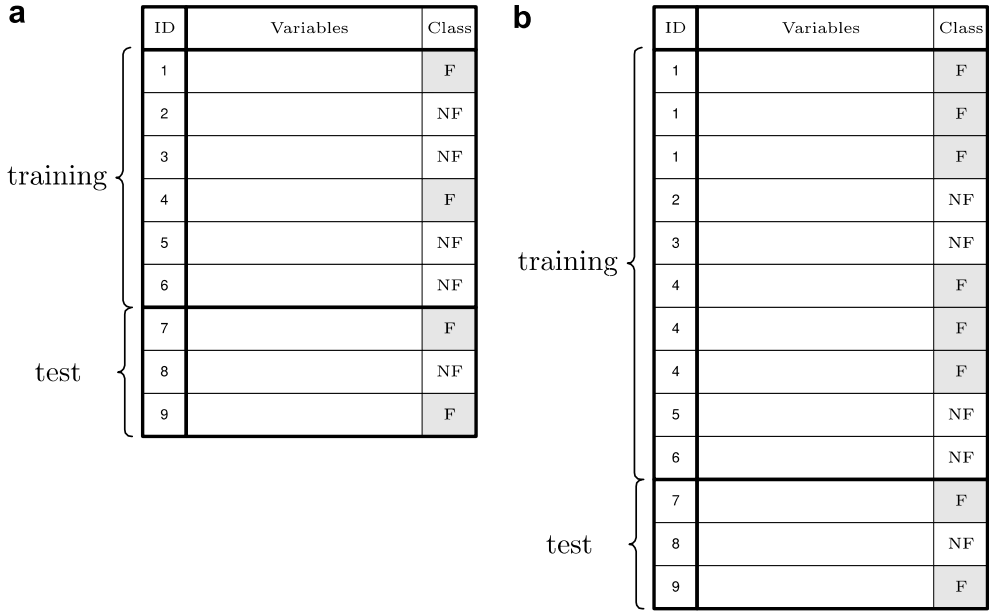

Fig. 5. A simplified example, where the original data with 9 modules (a) is oversampled twice in (b), meaning each training instance that is classified as faulty, is repeated twice.

Table 4
Out-of-sample performance gain for AntMiner+ using oversampling

	PC1			PC4			KC1		
	Acc	Spec	Sens	Acc	Spec	Sens	Acc	Spec	Sens
0 Oversampling	88.12	20.75	98.29	88.09	20.75	98.29	80.51	23.96	94.95
1 Oversampling	92.16	0.00	99.66	88.09	20.75	98.29	80.51	26.04	94.42
2 Oversampling	93.10	25.00	98.64	88.09	20.75	98.29	80.51	26.04	94.41
5 Oversampling	93.10	25.00	98.64	75.00	96.23	71.79	47.88	91.67	36.70
10 Oversampling	89.66	33.33	94.24	74.26	96.23	70.94	46.19	91.67	34.57
20 Oversampling	70.53	70.83	70.51	70.79	96.23	66.95	46.19	91.67	34.57

and majority vote. C4.5 is a popular decision tree builder (Quinlan, 1993) where each leaf assigns a class label to observations. Each of these leaves can be represented by a rule and therefore C4.5 also builds comprehensible classifiers. For our experiments, standard pruning factors were used, i.e. a confidence factor of 0.25. While C4.5 extracts an unordered rule set, AntMiner+ extracts an ordered rule set. As it also extracts an ordered rule set, the rule induction technique RIPPER is additionally included in our experiments. Again standard pruning factors were used. *K*-nearest neighbor (*k*-NN) classification classifies a data instance by only considering the *k* most similar data points in the training dataset. Logistic regression (logit) and support vector machines (SVM) (Vapnik, 1995) are also included. The parameters of SVM were set using the gridsearch method (Van Gestel et al., 2004). RIPPER, *1*-NN, C4.5 and logit are all evaluated using the Weka workbench (Witten and Frank, 2000). Finally, the very simplistic majority vote is also included.

The provided measures are out-of-sample performances, meaning the performance is measured on an independent test set. Only the training data was oversampled, the test set was not, as to provide a completely unbiased indication of the accuracy of future predictions.

4.4. Results and discussion

The results of our software mining experiments are summarized in Table 6 with the best performances in terms of accuracy, specificity and sensitivity, shown in bold. Although the focus is on the datasets with oversampling, the results on the datasets without oversampling are also given for comparison. Also included is the number of

Table 5
Ranking of the different classification techniques

	Acc	Spec	Sens
AntMiner+	4.67	2.67	5.00
RIPPER	3.83	4.17	4.67
C4.5	**3.00**	4.33	3.33
Logit	4.67	**2.50**	5.33
1-NN	3.50	5.67	3.33
SVM	5.00	3.67	3.33
Majority vote	3.33	5.00	**3.00**

Table 6
Out-of-sample results of the software mining experiments

	KC1							PC1							PC4						
	Original data				5 Oversampling			Original data				5 Oversampling			Original data				5 Oversampling		
	Acc	#R	Spec	Sens	Acc	#R	Spec	Sens	Acc	#R	Spec	Sens	Acc	#R	Spec	Sens	Acc	#R	Spec	Sens	Acc	#R	Spec	Sens
AntMiner+	80.51	2	23.96	94.95	47.88	2	91.67	36.70	91.85	2	0.00	99.32	**93.10**	5	25.00	98.64	88.12	4	20.75	98.29	75.00	2	**96.23**	71.79
RIPPER	**81.14**	3	26.04	95.21	57.84	5	8.33	51.33	92.19	2	0.00	99.66	85.94	18	**62.50**	87.84	85.64	4	**54.72**	90.31	82.67	21	79.25	83.19
C4.5	79.87	19	18.75	95.48	**59.11**	94	77.08	**54.52**	**92.50**	1	0.00	**100.00**	85.94	97	58.33	88.18	86.39	25	35.85	94.02	84.65	114	71.70	86.61
logit	79.45		18.75	94.95	51.69		86.46	42.82	92.19		0.00	99.66	81.25		**62.50**	82.77	**88.86**		32.08	97.44	86.14		86.79	86.04
1-NN	55.93		**75.00**	51.06	55.93		75.00	51.06	91.56		**8.33**	98.31	91.56		8.33	98.31	84.65		49.06	89.74	84.65		49.06	89.74
SVM	79.24		26.04	92.82	57.57		90.59	39.31	92.45		0.00	**100.00**	85.35		40.48	90.68	88.15		16.98	98.86	79.69		93.33	76.82
Majority vote	79.66		0.00	**100.00**	20.34		**100.00**	0.00	**92.50**		0.00	**100.00**	92.50		0.00	**100.00**	86.88		0.00	**100.00**	86.88		0.00	**100.00**

extracted rules (#R). Finally, Table 5 shows the average ranking of the different classification techniques for accuracy, specificity and sensitivity. In this table, a lower value stands for a better result. The computational duration for one run of the AntMiner+ algorithm is in the order of minutes. Although this is higher than for the other included rule-based classification techniques, it is certainly acceptable within this data mining context where computational time is not a critical factor (Martens et al., 2007).

Considering the average accuracy of the different classification techniques, C4.5 performs best, closely followed by majority vote, *1*-NN and RIPPER. AntMiner+, logit and SVM follow at a distance. The remarkably high accuracy for the very simplistic majority vote is another consequence of the very heavily skewed data distribution of the datasets. Only if the target variable has the same value in a clear majority of the observations, can majority vote obtain such a high ranking.

Accuracy is not an adequate performance measure for our experiments, as it implicitly assumes a relatively balanced class distribution among the observations and equal misclassification costs (Baesens et al., 2003b; Provost et al., 1998). We already mentioned the heavily skewed distribution of the datasets, which is typical for fault prediction in software mining. But also the assumption of equal misclassification costs can not be sustained. Typically, a software manager who applies software mining techniques for fault prediction purposes will be mainly interested in the correct detection of erroneous software modules. Probably even to that extent that he or she would prefer testing one module too many rather than one less. Indeed, the costs of testing one extra software module do not weigh up to the costs the software company incurs by delivering faulty software modules. Not only will the software company have to correct the faulty software modules anyway, the reputation of the software company will also be damaged as it provided a faulty system. As the costs associated with the incorrect classification of an erroneous software module are clearly higher than the costs associated with the incorrect classification of a correct software module, it seems fair to us to assume unequal misclassification costs. Consequently, a high specificity is of more importance to a software manager than a high sensitivity. However, this does not mean that sensitivity can be neglected. A classification technique that classifies all software modules as erroneous might well result in high quality software, the testing costs however will be unjustifiably high. A trade-off has to be made in order to obtain a high specificity combined with a reasonable sensitivity. This allows the software manager to efficiently allocate his testing budget to fault-prone software modules and increases the quality of the delivered software modules.

Keeping this in mind, we can again consult Table 5. We observe that AntMiner+ performs very well, with an average ranking for specificity of 2.67. Only logit performs slightly better, with an average ranking of 2.50. SVM, RIPPER and C4.5 follow at a distance. Majority vote and

Table 7
Comparison of the extracted rules

	PC1		PC4		KC1		Average	
	#R	#T/R	#R	#T/R	#R	#T/R	#R	#T/R
AntMiner+	6	3.5	2	1.5	2	1	3.3	2.0
RIPPER	18	3.3	21	3.5	5	2	14.7	2.9
C4.5	97		114		94		101.7	

1-NN end up last, with an average ranking of 5.00 and 5.67 respectively. As for sensitivity, the results are more or less the opposite. *1*-NN, C4.5 and SVM rank best, whereas RIPPER, AntMiner+ and logit follow at a distance.

We tested the statistical significance of these results, using the non-parametric Friedman test, as is advised by Demar (2006). We found the differences for accuracy, specificity and sensitivity between the classification techniques to be not statistically significant at the 5% level. This means that AntMiner+ does not perform significantly better, but also, and more importantly, not significantly worse than the other classification techniques. Performance-wise the classification techniques are thus very evenly matched.

However, a high accuracy, specificity and sensitivity does not say it all. A software manager might be interested in the reasons why a certain software module is classified as erroneous. This allows him to undertake correcting actions and prevent faults in the future. For example, if the software mining process would reveal that software modules written by more than five programmers are more fault-prone, the manager could proceed towards a restructuring of the programming activities. Or the software manager might want to switch towards a more modular programming style, if most of the developed software modules are rather large and the software mining process would reveal that large software modules are more fault-prone. Therefore, comprehensibility of the classification model is also an important requirement in software mining.

There is not really much to comprehend about a majority vote model. The only principle behind this technique is ''majority wins''. Therefore, majority vote adds almost no value for a software manager. The comprehensibility of *1*-NN, based on similarity with training data, is limited since there is no actual classifier. Logistic regression performs well, but its model structure is arguably more opaque than a rule-based representation. Finally, the non-linear SVM model is almost completely incomprehensible for a human user.

The only classification techniques that take comprehensibility into account, are C4.5, RIPPER and AntMiner+, as they extract comprehensible rules from the datasets. Comprehensibility increases as the number of rules (#R) and the number of terms per rule (#T/R) decreases. Table 7 compares the comprehensibility of these classification techniques.

AntMiner+ extracts the smallest number of rules, with an average of 3.3 rules as opposed to 14.7 rules for RIPPER and even 101.7 rules for C4.5. The issue faced by C4.5 here is its greedy character, since every split made in the decision tree is irreversibly present in all leaves underneath. AntMiner+ also has the smallest number of terms per rule. Therefore, it is obvious that AntMiner+ is the most comprehensible classification technique tested in our experiments.

Note that the performance differences between the datasets originate from the fact that these datasets come from different software projects. Therefore these datasets inherently (1) have different variables, and (2) have different noise levels present in the data.

Tables 8–10 show the rules extracted by AntMiner+ from the various datasets. Only 9 metrics from a total of 23 are actually used in the antecedents of these rules. It is worth noting that the rule antecedents of the extracted classification rules are never built from LOC metrics alone. In all cases a combination of complexity, Halstead and LOC metrics is used. For example, having a closer look at the rules extracted from the PC1 dataset (cf. Table 8), one can see that most rules include several LOC measures (such as number of lines and comment lines) as well as complexity measures (such as normalized cyclomatic complexity), thereby reflecting that the likelihood of encountering errors

Table 8
AntMiner+ rules for PC1

if LOC_Blank $\geqslant$ 16 **and** LOC_Code_And_Comment $\geqslant$ 2
and Normalized_Cyclomatic_Complexity $\geqslant$ 0.17
then class = Erroneous module

else if LOC_Code_And_Comment $\geqslant$ 1 **and**
LOC_Comments $\geqslant$ 5 **and**
Normalized_Cyclomatic_Complexity $\geqslant$ 0.23 **and**
Num_Unique_Operands $\geqslant$ 38
then class = Erroneous module

else if Halstead_Content $\geqslant$ 50.37 **and** Halstead_Error_Est $\geqslant$ 0.6
and LOC_Blank $\geqslant$ 16 **and** LOC_Comments $\geqslant$ 14 **and**
LOC_Executable $\geqslant$ 53
then class = Erroneous module

else if Halstead_Content $\geqslant$ 50.37 **and**
LOC_Blank $\geqslant$ 16 **and** LOC_Code_And_Comment $\geqslant$ 2
and LOC_Comments $\geqslant$ 14 **and**
Normalized_Cyclomatic_Complexity $\geqslant$ 0.08
then class = Erroneous module

else if Halstead_Content $\geqslant$ 50.37 **and**
LOC_Code_And_Comment $\geqslant$ 2 **and**
LOC_Comments $\geqslant$ 5 **and**
Normalized_Cyclomatic_Complexity $\geqslant$ 0.17
then class = Erroneous module
else class = Correct module

Table 9
AntMiner+ rule for PC4

if Halstead_Error_Est $\geqslant$ 0.04
and Halstead_Length $\geqslant$ 35
and LOC_Total $\geqslant$ 9
then class = Erroneous module

else class = Correct module

Table 10
AntMiner+ rule for KC1

if Halstead_Error_Est $\geqslant$ 0.02
and LOC_Total $\geqslant$ 7
then class = Erroneous module

else class = Correct module

increases with the size of the model and its complexity. This is completely in line with the findings of Fenton and Ohlsson (1999) and Fenton and Neil (1999), who remarked that metrics solely based on LOC are not capable of making good predictions. Furthermore, the intuitive character of the rules extracted by AntMiner+ makes them easy to accept for software managers.

As a final remark, we would like to draw the attention to the fact that the concrete rules extracted in this experiment can give an indication to software managers, but are by no means generally applicable. If a software manager wishes to use the described techniques to improve the quality of his software, he or she is advised to repeat the experiment on any available in-house datasets, containing data of prior software projects within the company (Menzies et al., 2004). Indeed, every software company has its own characteristics and habits, that are reflected in the specific properties of the source code. By repeating the experiment, one obtains personalized rules, that offer a better fit for the specific software company. This can only lead towards an increase in software quality.

5. Conclusion

The possibility to predict faults in software modules can be very valuable for software managers. The quality of the software improves and moreover, this quality is obtained in a more efficient way, as during the testing phase, efforts can be focused more on those software modules that were classified as fault-prone by the classification technique. If this technique provides a comprehensible classification model, as is the case in our AntMiner+ approach, this model can be used to discover critical points in the software development process. The software manager can use these new insights to take concrete actions to improve the software development process. This way the quality of future software projects is also improved.

We have shown the induced classification models offer both accuracy and comprehensibility, which corroborates our initial motivation for the use of this technique.

Acknowledgments

We extend our gratitude to the editor and the anonymous reviewers, as their constructive remarks certainly contributed much to the quality of this paper. Further, we would like to thank the Flemish Research Council (FWO, Grant G.0615.05), and the Microsoft and KBC-Vlekho-K.U.Leuven Research Chairs for financial support to the authors.

References

Abraham, A., Ramos, V., 2003. Web usage mining using artificial ant colony clustering. Proceedings of IEEE Congress on Evolutionary Computation (CEC 2003). IEEE Press, Canberra, Australia, pp. 1384–1391.

Baesens, B., 2003. Developing intelligent systems for credit scoring using machine learning techniques. Ph.D. thesis, K.U.Leuven.

Baesens, B., Setiono, R., Mues, C., Vanthienen, J., 2003a. Using neural network rule extraction and decision tables for credit-risk evaluation. Management Science 49 (3), 312–329.

Baesens, B., Van Gestel, T., Viaene, S., Stepanova, M., Suykens, J.A.K., Vanthienen, J., 2003b. Benchmarking state-of-the-art classification algorithms for credit scoring. Journal of the Operational Research Society 54 (6), 627–635.

Berry, M., Linoff, G., 2004. Data Mining Techniques: for Marketing, Sales and Customer Relationship Management. John Wiley & Sons, New York, NY.

Blum, C., 2005. Beam-ACO – hybridizing ant colony optimization with beam search: An application to open shop scheduling. Computers & Operations Research 32 (6), 1565–1591.

Boehm, B., Abts, C., Brown, A., Chulani, S., Clark, B.K., Horowitz, E., D.K., R.M.D.R., Steece, B., 2000. Software Cost Estimation with COCOMO II. Prentice-Hall PTR, New Jersey.

Bullnheimer, B., Hartl, R., Strauss, C., 1999a. Applying the ant system to the vehicle routing problem. In: Voss, S., Martello, S., Osman, I., Roucairol, C. (Eds.), Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization. Kluwer Academic Publishers., Dordrecht, Netherlands, pp. 285–296.

Bullnheimer, B., Hartl, R.F., Strauss, C., 1999b. A new rank based version of the ant system: a computational study. Central European Journal for Operations Research and Economics 7 (1), 25–38.

Colorni, A., Dorigo, M., Maniezzo, V., Trubian, M., 1994. Ant system for jobûshop scheduling. Journal of Operations Research, Statistics and Computer Science 34 (1), 39–53.

Cusumano, M., MacCormack, A., Kemerer, C., Crandall, W., 2004. Software development worldwide: the state of practice. IEEE Computer Society 20 (6), 28–34.

Demar, J., 2006. Statistical comparisons of classifiers over multiple data sets. Journal of Machine Learning Research 7, 1–30.

De Rore, L., Snoeck, M., Dedene, G., 2006. COCOMO II applied in a banking and insurance environment: experience report. In: Proceedings of the 3rd Software Measurement European Forum. pp. 247–257.

Di Caro, G., Dorigo, M., 1998. Antnet: distributed stigmergetic control for communications networks. Journal of Artificial Intelligence Research 9, 317–365.

Dick, S., Meeks, A., Last, M., Bunke, H., Kandel, A., 2003. Data mining in software metrics databases. Fuzzy Sets and Systems, 81–110.

Dorigo, M., Gambardella, L.M., 1997. Ant colony system: a cooperative learning approach to the traveling salesman problem. IEEE Transactions on Evolutionary Computation 1 (1), 53–66.

Dorigo, M., Stützle, T., 2004. Ant Colony Optimization. MIT Press, Cambridge, MA.

Dorigo, M., Maniezzo, V., Colorni, A., 1991. Positive feedback as a search strategy. Tech. Rep. 91016, Dipartimento di Elettronica e Informatica, Politecnico di Milano, IT.

Dorigo, M., Maniezzo, V., Colorni, A., 1996. The ant system: optimization by a colony of cooperating agents. IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics 26 (1), 29–41.

Fenton, N., Neil, M., 1999. A critique of software defect prediction models. IEEE Transactions on Software Engineering 25 (5), 675–689.

Fenton, N., Ohlsson, N., 1999. Quantitative analysis of faults and failures in a complex software system. IEEE Transactions on Software Engineering 26 (8), 797–814.

Gambardella, L., Dorigo, M., 1995. Ant-q: a reinforcement learning approach to the traveling salesman problem. In: Proceedings of the Eleventh International Conference on Machine Learning. pp. 252–260.

Graves, T., Karr, A., Marron, J., Siy, H., 1999. Predicting fault incidence using software change history. IEEE Transactions on Software Engineering 26 (7), 653–661.

Hand, D.J., 2002. Pattern detection and discovery. In: Hand, D.J., Adams, N., Bolton, R. (Eds.), Pattern detection and discovery, . In: Lecture Notes in Computer Science, vol. 2447. Springer-Verslag, Berlin, Germany, pp. 1–12.

Hand, D., 2006. Protection or privacy? Data mining and personal data. Proceedings of the 10th Pacific-Asia Conference, PAKDD. In: Lecture Notes in Computer Science, vol. 3918. Springer, pp. 1–10.

Handl, J., Knowles, J., Dorigo, M., 2006. Ant-based clustering and topographic mapping. Artificial Life 12 (1), 35–61.

Higgins, R., 2003. Analysis for Financial Management. McGraw-Hill/Irwin, New York, NY.

Huysmans, J., Baesens, B., Martens, D., Denys, K., Vanthienen, J., 2005. New trends in data mining. In: Tijdschrift voor economie en Management, vol. L. pp. 697–711.

Juran, J., 1964. Managerial Breakthrough. McGraw-Hill, New York, NY.

Khoshgoftaar, T., Seliya, N., 2003. Analogy-based practical classification rules for software quality estimation. Empirical Software Engineering 8 (4), 325–350.

Liu, B., Abbass, H.A., McKay, B., 2002. Density-based heuristic for rule discovery with ant-miner. In: 6th Australasia-Japan Joint Workshop on Intelligent and Evolutionary Systems (AJWIS2002). Canberra, Australia, pp. 180–184.

Liu, B., Abbass, H.A., McKay, B., 2003. Classification rule discovery with ant colony optimization. Proceedings of the IEEE/WIC International Conference on Intelligent Agent Technology. IEEE Computer Society, Los Alamitos, CA, pp. 83–88.

MacDonell, S., Shepperd, M., 2003. Combining techniques to optimize effort predictions in software project management. Journal of Systems and Software 66, 91–98.

Martens, D., De Backer, M., Haesen, R., Baesens, B., Holvoet, T., 2006a. Ants constructing rule-based classifiers. Swarm Intelligence and Data Mining. Studies in Computational Intelligence. Springer-Verslag, Ch. 2.

Martens, D., De Backer, M., Haesen, R., Baesens, B., Mues, C., Vanthienen, J., 2006b. Ant-based approach to the knowledge fusion problem. In: Dorigo, M., Gambardella, L., Birattari, M., Martinoli, A., Poli, R., Stützle, T. (Eds.), Ant Colony Optimization and Swarm Intelligence, 5th International Workshop, ANTS 2006, vol. 4150. Springer-Verslag, Berlin, Germany, pp. 84–95.

Martens, D., De Backer, M., Haesen, R., Snoeck, M., Vanthienen, J., Baesens, B., 2007. Classification with ant colony optimization. IEEE Transaction on Evolutionary Computation 11(5), pp. 651–665.

Menzies, T., Di Stefano, J., Cunanan, C., Chapman, R., 2004. Mining repositories to assist in project planning and resource allocation. In: Proceedings of the International Workshop on Mining Software Repositories.

Mockus, A., Fielding, R., Herbsleb, J., 2000. A case study of open source software development: The apache server. In: Proceedings of the ICSE 2000 conference. pp. 263–272.

Montemanni, R., Gambardella, L.M., Rizzoli, A.E., Donati, A., 2005. Ant colony system for a dynamic vehicle routing problem. Journal of Combinatorial Optimization 10 (4), 327–343.

Parpinelli, R.S., Lopes, H.S., Freitas, A.A., 2001. An ant colony based system for data mining: applications to medical data. Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001). Morgan Kaufmann Publishers Inc., San Francisco, CA, pp. 791–797.

Passmore, L., Goodside, J., Hamel, L., Gonzales, L., Silberstein, T., Trimarchi, J., 2003. Assessing decision tree models for clinical in-vitro fertilization data. Technical Report TR03-296, Department of Computer Science and Statistics, University of Rhode Island.

Pazzani, M., Mani, S., Shankle, W., 2001. Acceptance by medical experts of rules generated by machine learning. Methods of Information in Medicine 40 (5), 380–385.

Premraj, R., Shepperd, M.J., Kitchenham, B., Forselius, P., 2005. An empirical analysis of software productivity over time. 11th IEEE International Symposium on Software Metrics. IEEE Computer Society, p. 37.

Provost, F., Fawcett, T., Kohavi, R., 1998. The case against accuracy estimation for comparing classifiers. In: 15th International Conference on Machine Learning.

Quinlan, J.R., 1993. C4.5: programs for Machine Learning. Morgan Kaufmann Publishers Inc., San Francisco, CA.

Seifert, J., 2006. Data mining and homeland security: an overview. CRS Report for Congress.

Shepperd, M., Cartwright, M., 2001. Predicting with sparse data. IEEE Transactions on Software Engineering 27 (11), 987–998.

Shepperd, M., Cartwright, M., Kadoda, G., 2000. On building prediction systems for software engineers. Empirical Software Engineering 5, 175–182.

Socha, K., Knowles, J., Sampels, M., September 2002. A $\mathcal{MAX}$–$\mathcal{MIN}$ ant system for the university timetabling problem. In: Dorigo, M., Di Caro, G., Sampels, M. (Eds.), Proceedings of ANTS 2002 – Third International Workshop on Ant Algorithms, Vol. 2463 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany, pp. 1–13.

Song, Q., Shepperd, S., Mair, C., 2005. Using grey relational analysis to predict software effort with small data sets. In: Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05). p. 35.

Song, Q., Shepperd, S., Cartwright, M., Mair, C., 2006. Software defect association mining and defect correction effort prediction. IEEE Transactions on Software Engineering 32 (2), 69–82.

Stützle, T., Hoos, H.H., 1996. Improving the ant-system: a detailed report on the $\mathcal{MAX}$–$\mathcal{MIN}$ ant system. Technical Report AIDA 96-12, FG Intellektik, TU Darmstadt, Germany.

Stützle, T., Hoos, H.H., 2000. $\mathcal{MAX}$–$\mathcal{MIN}$ ant system. Future Generation Computer Systems 16 (8), 889–914.

The Standish Group, 2004. Chaos report. Tech. rep.

Thomas, L., Edelman, D., Crook, J. (Eds.), 2002. Credit Scoring and its Applications. SIAM, Philadelphia, PA.

Van Gestel, T., Suykens, J., Baesens, B., Viaene, S., Vanthienen, J., Dedene, G., De Moor, B., Vandewalle, J., 2004. Benchmarking least squares support vector machine classifiers. Machine Learning 54 (1), 5–32.

Vapnik, V.N., 1995. The nature of statistical learning theory. Springer-Verlag, New York, NY.

Wade, A., Salhi, S., 2004. An ant system algorithm for the mixed vehicle routing problem with backhauls. Metaheuristics: computer decision-making. Kluwer Academic Publishers., Norwell, MA, pp. 699–719.

Williams, C., Hollingsworth, J., 2005. Automatic mining of software repositories to improve bug finding techniques. IEEE Transactions on Software Engineering 31 (6), 466–480.

Witten, I.H., Frank, E., 2000. Data mining: practical machine learning tools and techniques with Java implementations. Morgan Kaufmann Publishers Inc., San Francisco, CA.

Ying, A., Murphy, G., Ng, R., Chu-Carroll, M., 2004. Predicting source code changes by mining change history. IEEE Transactions on Software Engineering 30 (9), 574–586.

Zhang, X., Pham, H., 2006. Software field failure rate prediction before software deployment. Journal of Systems and Software 79, 291–300.

Zhong, S., Khoshgoftaar, T., Naeem, S., 2004. Analyzing software measurement data with clustering techniques. IEEE Intelligent Systems 19 (2), 20–27.

Zimmerman, T., Weissgerber, P., Diehl, S., Zeller, A., 2005. Mining version histories to guide software changes. IEEE Transactions on Software Engineering 31 (6), 429–445.