Empirical Software Engineering, 7, 297–318, 2002.
© 2002 Kluwer Academic Publishers. Manufactured in The Netherlands.

Uncertain Classification of Fault-Prone Software Modules

TAGHI M. KHOSHGOFTAAR taghi@cse.fau.edu
XIAOJING YUAN xyuan@cse.fau.edu
Empirical Software Engineering Laboratory, Department of Computer Science and Engineering, Florida Atlantic University, Boca Raton, FL 33431, USA

EDWARD B. ALLEN* edward.allen@computer.org
Mississippi State University, Mississippi, USA

WENDELL D. JONES wjones@asciences.com
JOHN P. HUDEPOHL hudepohl@nortelnetworks.com
Nortel Networks, Research Triangle Park, North Carolina, USA

Abstract. Many development organizations try to minimize faults in software as a means for improving customer satisfaction. Assuring high software quality often entails time-consuming and costly development processes. A software quality model based on software metrics can be used to guide enhancement efforts by predicting which modules are fault-prone. This paper presents statistical techniques to determine which predictions by a classification tree should be considered uncertain.

We conducted a case study of a large legacy telecommunications system. One release was the basis for the training dataset, and the subsequent release was the basis for the evaluation dataset. We built a classification tree using the TREEDISC algorithm, which is based on χ^2 tests of contingency tables. The model predicted whether a module was likely to have faults discovered by customers, or not, based on software product, process, and execution metrics. We simulated practical use of the model by classifying the modules in the evaluation dataset. The model achieved useful accuracy, in spite of the very small proportion of fault-prone modules in the system. We assessed whether the classes assigned to the leaves were appropriate by statistical tests, and found sizable subsets of modules with uncertain classification. Discovering which modules have uncertain classifications allows sophisticated enhancement strategies to resolve uncertainties. Moreover, TREEDISC is especially well suited to identifying uncertain classifications.

Keywords: Software metrics, software quality, fault-prone modules, classification trees, CHAID, TREEDISC, telecommunications.

1. Introduction

High software quality is essential for modern computer systems. However, assuring high quality often entails time-consuming and costly development processes, such as more rigorous design and code reviews, automatic test-case generation, more

* The work was performed while Edward B. Allen was at Florida Atlantic University.

extensive testing, and reengineering of high-risk portions of a system. One cost-effective strategy is to target enhancement activities to those software modules that are most likely to have problems (Hudepohl et al., 1996). In particular, many development organizations try to minimize faults in the software as a means for improving customer satisfaction. A software *fault* is a defect in an executable product that causes a software failure.

A software quality model based on empirical data can be used to guide enhancement efforts. Software-metrics research has shown that software product and process metrics can be the basis for quality predictions, such as whether a software module is likely to have faults discovered during operations, in other words, whether it is *fault-prone*.

Of course, no model can make perfect predictions, and thus, an analyst should assess the expected accuracy and certainty of the predictions. To this end, this paper presents statistical techniques to determine which predictions by a two-class classification-tree model should be considered uncertain (Khoshgoftaar et al., 1996). One approach to this problem might be to define a third class between fault-prone and not fault-prone called *uncertain*. However, the criterion for the three classes must be defined a priori, and every training module must be classified into one of the three groups prior to modeling. Only a few studies have applied classification techniques to three or more software quality classes. Ohlsson and Wohlin (1998) classified modules as red, yellow, or green based on a pattern of faultiness over two releases. Szabo and Khoshgoftaar (1995) classified modules as high, medium, and low-risk using thresholds on the number of faults found in one release and then used three-class discriminant analysis. Rather than view an assessment of uncertain as an attribute of the software module alone, we view it as a characteristic of the module in relation to the classification model. A third class is inappropriate for analysis of uncertainty; after all, classification into three classes can still be uncertain. Consequently, this paper presents statistical techniques for assessment of uncertain classifications with a two-class tree modeling method.

In this paper, we employ classification trees to model software quality. A variety of other classification techniques have also been used to model software quality, such as discriminant analysis (Khoshgoftaar et al., 1996b) discriminant coordinates (Ohlsson et al., 1998) discriminant power (Schneidewind, 1995, 1997) logistic regression (Basili et al., 1996) optimal set reduction (Briand et al., 1993) neural networks (Khoshgoftaar and Lanning, 1995) and fuzzy classification (Ebert, 1996). However, the certainty of predictions has rarely been utilized. Classification-tree models are attractive, because they readily model nonlinear and nonmonotonic relationships among variables, and they are especially amenable to assessing uncertainty.

Published studies have used various classification-tree algorithms to model software quality. Selby and Porter (1988) used the ID3 algorithm [23] which maximizes homogeneity of leaves by an entropy-based criterion. Takahashi et al. (1997) refined the ID3 algorithm by pruning the resulting tree according to Akaike Information Criterion procedures. Case studies by Gokhale and Lyu (1997), Troster and Tian

(1995), and Khoshgoftaar et al. (2000a) used a regression-tree algorithm based on deviance. A case study by Khoshgoftaar et al. (1998) employed the CART algorithm which is based on the Gini index of diversity derived from probabilities of class membership.

The TREEDISC algorithm[1] is the focus of this paper (SAS Institute Staff, 1995). It is a refinement of the CHAID algorithm (Kass, 1980), emphasizing statistical significance based on χ^2 tests of contingency tables, rather than heuristic criteria. TREEDISC also allows multiway branching, possibly resulting in a more parsimonious model than binary branching. Preliminary empirical work with a small software subsystem showed that TREEDISC has promise for modeling software quality, but did not address uncertain classifications (Khoshgoftaar et al., 1996a).

In the remainder of this paper we present background on classification trees and methods for assessing which predicted classifications should be considered uncertain. We then present a case study of a large legacy telecommunications system (Khoshgoftaar et al., 1999b; Yuan, 1999) to illustrate the methods. Finally, we summarize lessons learned.

In the case study, one release was the basis for the *training* dataset and the subsequent release was the basis for the *evaluation* dataset. A classification-tree model predicted whether a module was likely to have faults discovered by customers, or not, based on software product, process, and execution metrics. Faults discovered by customers in telecommunications systems usually have extremely expensive consequences, and thus, prediction of fault-prone modules is very important to the developers. The case study illustrated how TREEDISC models can be used not only to classify software modules, but also to identify modules with uncertain classifications.

2. Uncertainty in Classification Trees

A classification tree is an algorithm of decision rules to classify an object, represented by an abstract tree. In our application, an object is a software module. We follow terminology in the statistics literature on classification trees, calling independent variables *predictors*, and the dependent variable the *response variable*. Each internal node represents a decision, and each outgoing edge represents a possible result of that decision. Each leaf node is labeled with a class of the response variable: fault-prone (fp) or not fault-prone (nfp) in our application. The *root* of the tree is the node at the top.

Given a classification tree and a module's predictor values, beginning at the root, the algorithm traverses a downward path in the tree. When the algorithm reaches a decision node, the value of the associated predictor is compared to the range of values associated with each outgoing edge, and the algorithm proceeds along the proper edge to the next node. This process is repeated for each node along the path. When a leaf is reached, the module is classified according to the label of the leaf, and the path is complete. Table 1 lists notation for convenient reference.

Table 1. Notation.

Symbol	Definition
fp	Fault-prone
nfp	Not fault-prone
Pr(fp|nfp)	Type I misclassification rate
Pr(nfp|fp)	Type II misclassification rate
c	Number of predictor categories
$c_{\max}$	Maximum number of predictor categories
i	Index of a module
k	Number of merged predictor categories
l	Index of a leaf
$L(\mathbf{x}_i)$	Index of the leaf that module i falls into
n	Number of modules in training dataset
n_l	Number of training modules in leaf l
n_{fp}	Number of fault-prone training modules in current leaf
n_{nfp}	Number of not fault-prone training modules in current leaf
p	p-Value of a χ^2 test
$q(l)$	Probability that a module in leaf l is fault-prone
$\hat{q}(l)$	Estimated $q(l)$
q_{lower}	Lower confidence limit of $q(l)$
q_{upper}	Upper confidence limit of $q(l)$
r	Number of response categories, $r = 2$
x_{ij}	Value of the jth predictor of module i
$\mathbf{x}_i$	Vector of predictor values for module i
α	Significance level
ϵ_l	A probability near zero for leaf l
θ	A threshold parameter, $\theta = \zeta/(1+\zeta)$
θ_{low}	Low end point of a threshold interval
θ_{high}	High end point of a threshold interval
ζ	A classification-rule parameter

2.1. Preprocessing Metrics

In this paper, predictors are based on software metrics. Most primitive software metrics are positive integers with no upper bound, in principle. A few are real numbers. TREEDISC is not compatible with numeric predictors, but it is suitable for ordinal predictors with many categories, within computational constraints. Therefore, we transform all raw measurements into discrete ordinal predictors.

In this paper, we used *grouping* to transform numeric metrics into discrete ordinal predictors (Khoshgoftaar et al., 1999c). Let $c_{\max}$ be the maximum number of predictor categories. Let rank_{ij} be an ith module's rank within the dataset according to the jth raw metric and let n be the number of modules in the dataset. Ties are assigned a common rank. Predictor values are calculated by the following.

$$x_{ij} = \left\lfloor \left(\frac{c_{\max}}{n+1} \right) \mathrm{rank}_{ij} \right\rfloor \tag{1}$$

The jth predictor value of module i is an integer, $x_{ij} \in \{0, \ldots, c_{\max} - 1\}$. In the absence of ties, the groups have equal or nearly equal number of modules. For example, a value of $c_{\max} = 4$ would produce four predictor categories, each with about one-quarter of the modules. A large number of ties may reduce the actual number of categories, $c \leq c_{\max}$.

In this study, we applied this transformation to each metric in each dataset using the same value of $c_{\max} = 60$. Future research will investigate other transformations.

2.2. Building a Classification Tree

We build a classification tree using the TREEDISC algorithm (SAS Institute Staff, 1995) which is a refinement of the CHAID algorithm (Kass, 1980). TREEDISC is implemented as a macropackage for the SAS® System. Improvements to the CHAID algorithm include adjusting the χ^2 statistic for better accuracy (Hawkins and Kass, 1982), specifying a method for finding the most significant branching criterion, and avoiding the possibility of infinite loops.

The algorithm recursively partitions ("splits") the training dataset's modules into new leaves of the classification tree until a stopping criterion applies to every leaf. Partitioning of a leaf stops when there is no significant partition or when there are too few modules in the leaf for a useful significance test. Let α be the significance level. The size of a tree is controlled by the specified significance level, rather than by a pruning criterion such as with CART (Khoshgoftaar et al., 1998).

The range of predictor values associated with each edge consists of adjacent predictor categories merged together. For a given predictor, let c be the number of original predictor categories, and let k be the number of merged predictor categories.

For the predictor's $k \times r$ merged contingency table, the χ^2 statistic, X in this context, is given by the following:

$$X = \sum_i \sum_j \frac{(f_{ij} - \hat{f}_{ij})^2}{\hat{f}_{ij}} \tag{2}$$

where f_{ij} is a frequency in cell ij of the contingency table, and $\hat{f}_{ij}$ is the expected frequency under the null hypothesis that the response variable is independent of the predictor (Zar, 1984) X has a chi-squared (χ^2) distribution. Significance is evaluated using an adjusted p-value.

$$p = \min\left(\binom{c-1}{k-1} \Pr(\chi^2_{[(r-1)(k-1)]} > X),\ \Pr(\chi^2_{[(r-1)(c-1)]} > X) \right) \tag{3}$$

where $r = 2$ is the number of response-variable classes. A significance test rejects the null hypothesis if $p < \alpha$. The TREEDISC algorithm finds the most significant

partitioning of a predictor's original categories with respect to the response variable, creating merged predictor categories. TREEDISC allows multiway splits and minimizes the adjusted p-value of χ^2 tests to determine the merged predictor categories. Multiway splits allow more accurate modeling than binary splits for nonmonotonic relationships between predictors and the response variable.

2.3. A Classification Rule

After the tree is built, each leaf, l, must be labeled with a class. This, in effect, determines a rule for classifying modules. Recall that the TREEDISC algorithm partitions all the modules in the training dataset among the leaves according to the structure of the tree. Moreover, the actual class of each module in the training dataset is known. Let $q(l)$ be the probability that a module in leaf l is fault-prone, and let the estimated probability, $\hat{q}(l)$, be the proportion of training modules in leaf l that are actually fault-prone.

Let $\mathbf{x}_i$ be the ith module's vector of predictor values. Let $L(\mathbf{x}_i)$ be the leaf that the ith module falls into according to the structure of the tree. TREEDISC's default rule for classifying a module is the following:

$$\text{Class}(\mathbf{x}_i) = \begin{cases} \text{nfp} & \text{if } 1 - \hat{q}(L(\mathbf{x}_i)) \geq \hat{q}(L(\mathbf{x}_i)) \\ \text{fp} & \text{otherwise} \end{cases} \tag{4}$$

This rule did not yield satisfactory accuracy in our empirical investigation.

Accuracy is measured by misclassification rates. The *Type I* misclassification rate, Pr(fp|nfp), is estimated by the proportion of not fault-prone modules that are misclassified. The *Type II* misclassification rate, Pr(nfp|fp), is estimated by the proportion of fault-prone modules that are misclassified. When the proportion of fault-prone modules in the system is small, the default classification rule in Equation (4) is not useful for software-quality improvement. To address this phenomenon, we propose the following general classification rule for a classification tree (Khoshgoftaar et al., 2000b):

$$\text{Class}(\mathbf{x}_i) = \begin{cases} \text{nfp} & \text{if } \frac{1-\hat{q}(L(\mathbf{x}_i))}{\hat{q}(L(\mathbf{x}_i))} \geq \zeta \\ \text{fp} & \text{otherwise} \end{cases} \tag{5}$$

With various classification techniques, we have observed a tradeoff between Type I and Type II misclassification rates as a function of ζ (Khoshgoftaar and Allen, 2000; Khoshgoftaar et al., 1998, 1999a). As Pr(nfp|fp) goes down, Pr(fp|nfp) goes up, and conversely. The value of ζ can be chosen according to the preference of the project, to achieve a preferred balance between the Type I and Type II misclassification rates.

An equivalent formulation is the following:

$$\text{Class}(\mathbf{x}_i) = \begin{cases} \text{nfp} & \text{if } 1 - \hat{q}(L(\mathbf{x}_i)) \geq \theta \\ \text{fp} & \text{otherwise} \end{cases} \tag{6}$$

where the threshold $\theta = \zeta/(1+\zeta)$. The TREEDISC default rule in Equation (4) is equivalent to $\zeta = 1$ and $\theta = 0.50$. We choose a preferred value of θ empirically. Given a candidate value of θ, we estimate misclassification rates Pr(fp|nfp) and Pr(nfp|fp) by resubstitution of the training dataset into the model. If the balance is not satisfactory, we select another candidate value of θ and estimate again, until we arrive at the preferred θ for the project. This procedure is straightforward in practice, because the misclassification rates are monotonic functions of θ. For example, if one chooses θ such that Pr(fp|nfp) $\approx$ Pr(nfp|fp), then the rule approximately minimizes the larger of the misclassification rates (min–max rule) (Seber, 1984). In practice, we can achieve only approximate equality due to finite datasets and discrete predictors. Another balance between misclassification rates may be preferred by the project in other situations.

To ease interpretation, we simplify each tree when adjacent leaves from the same decision node have the same preferred classification, yielding an equivalent tree.

2.4. Confidence Interval Analysis

Given one preferred value of θ, a more sophisticated classification rule than Equation (6) should include uncertain classifications. A detailed examination of each leaf l in the full classification tree built by TREEDISC can give useful insights.

In general, the number of fault-prone modules in a sample is a random variable, X in this context, which is distributed according to the binomial distribution with the following cumulative distribution function (Zar, 1984):

$$\Pr(X \leq x | N, q) = \sum_{j=0}^{x} \binom{N}{j} q^j (1-q)^{N-j} \tag{7}$$

where N is the size of the sample in this context, $0 \leq x \leq N$, and q is the probability of being fault-prone. Similarly, the number of not fault-prone modules also has a corresponding binomial distribution.

Let n_{fp} and n_{nfp} be the number of fault-prone and not fault-prone training modules that fall into leaf l, and let $n_l = n_{\text{fp}} + n_{\text{nfp}}$ be the total. Recall that the probability that a module in a leaf is fault-prone is estimated by $\hat{q}(l) = n_{\text{fp}}/n_l$. A confidence interval at significance α for $q(l)$ is given by lower and upper confidence limits, $q_{\text{lower}}(l) < q(l) < q_{\text{upper}}(l)$. The preferred significance level α is chosen by the analyst. TREEDISC's significance level for building the tree is not necessarily the same as the significance level of the confidence interval. The F distribution is related to the binomial distribution so that the confidence interval of $q(l)$ can be calculated by the following (Zar, 1984):

$$q_{\text{lower}}(l) = \frac{n_{\text{fp}}}{n_{\text{fp}} + (n_l - n_{\text{fp}} + 1)F_{\alpha/2,v_n,v_d}} \tag{8}$$

where the degrees of freedom are given by $v_n = 2(n_l - n_{fp} + 1)$ and $v_d = 2n_{fp}$.

$$q_{\text{upper}}(l) = \frac{(n_{\text{fp}} + 1)F_{\alpha/2,v'_n,v'_d}}{n_l - n_{\text{fp}} + (n_{\text{fp}} + 1)F_{\alpha/2,v'_n,v'_d}} \tag{9}$$

where the degrees of freedom are given by $v'_n = 2(n_{\text{fp}} + 1)$ and $v'_d = 2(n_l - n_{\text{fp}})$.

Recall that training modules are partitioned among the leaves; some leaves have fewer modules than others. One should evaluate whether the number of fault-prone modules in each leaf is sufficient for its $\hat{q}(l)$ to be a credible estimate.[2] When $q(l) = 0$, leaf l should certainly be labeled not fault-prone. When there are only a few fault-prone observations, we have little confidence that $q(l) > 0$, and thus, a label of fault-prone for leaf l is not very credible, no matter what the preferred value of θ is. In the case study, we considered $\epsilon_l = 2/n_l$ to be "near zero". (Another small probability could be used in another study.) The confidence interval analysis above gives us a tool for assessing the credibility of $\hat{q}(l)$. When the lower confidence limit is smaller than "near zero", we label the leaf not fault-prone, because we cannot distinguish $q(l)$ from zero, irrespective of θ.

Because the choice of threshold θ is simply a preference, one should assess the sensitivity of results to nearby alternatives. If a leaf's value of $1 - \hat{q}(l)$ is "near" the threshold θ, then its appropriate label is uncertain.

$$\text{Class}(\mathbf{x}_i) = \begin{cases} \text{nfp} & \text{if } q_{\text{lower}}(L(\mathbf{x}_i)) \leq \epsilon_{L(\mathbf{x}_i)} \\ \text{nfp} & \text{if } 1 - q_{\text{upper}}(L(\mathbf{x}_i)) \geq \theta \\ \text{fp} & \text{if } 1 - q_{\text{lower}}(L(\mathbf{x}_i)) \leq \theta \\ \text{uncertain} & \text{otherwise} \end{cases} \tag{10}$$

at significance α. Note that if the confidence limits of $q(l)$ are $q_{\text{lower}} < q(l) < q_{\text{upper}}$, then the confidence limits for $1 - q(l)$ are $1 - q_{\text{upper}} < 1 - q(l) < 1 - q_{\text{lower}}$. Equation (10) means that if θ falls within the confidence interval of $1 - q(l)$, the leaf's label is uncertain.

All modules that fall into uncertain leaves have uncertain predicted classifications. Knowing which modules have uncertain predicted classifications is valuable information that classification alone does not provide. A project could employ sophisticated enhancement strategies to maximize effectiveness. For example, further risk assessment of uncertain modules could examine information not included in the classification model. Those uncertain modules evaluated as high risk could then be enhanced in the same manner as other modules classified as fault-prone. Further research will investigate the range of possible enhancement options. Moreover, when

diagnosis or enhancement options span a range of costs, extending our approach to multiple levels of uncertainty may help one choose an appropriate strategy.

2.5. Ambivalent Threshold

In practice, one is usually able to select one preferred value of θ. However, in some cases, a range of values for θ may be satisfactory. Suppose we prefer an interval of values, $[\theta_{\text{low}}, \theta_{\text{high}}]$, instead of a single value. The previous section analyzes uncertainty due to random variation. This section considers uncertain classification due to ambivalence concerning the threshold, θ, and analyzes random variation near the endpoints of the interval.

If the estimated probability of a module being not fault-prone, $1 - \hat{q}(l)$, is within the interval $[\theta_{\text{low}}, \theta_{\text{high}}]$, then the class of the module is uncertain (Khoshgoftaar et al., 1999b):

$$\text{Class}(\mathbf{x}_i) = \begin{cases} \text{nfp} & \text{if } 1 - \hat{q}(L(\mathbf{x}_i)) \geq \theta_{\text{high}} \\ \text{fp} & \text{if } 1 - \hat{q}(L(\mathbf{x}_i)) \leq \theta_{\text{low}} \\ \text{uncertain} & \text{otherwise} \end{cases} \tag{11}$$

However, this rule does not consider the uncertainty in the estimated values of $q(l)$. Recall that each $q(l)$ is estimated by the proportion of training modules in leaf l that are actually fault-prone. When there are fewer training modules in leaf l, the estimate is less precise. For each leaf l that has a credible estimate of $q(l)$, rather than a confidence interval analysis, consider the following classification criteria (Zar, 1984).

1. For the low end of the interval, let the null hypothesis and alternative hypothesis be

$$\begin{aligned} H_0 &: \quad 1 - q(l) \geq \theta_{\text{low}} \\ H_A &: \quad 1 - q(l) < \theta_{\text{low}} \end{aligned} \tag{12}$$

   In this context, let X be a random variable for the number of fault-prone modules. If $\Pr(X \leq \text{n}_{\text{nfp}} | n_l, \theta_{\text{low}}) < \alpha/2$ then reject H_0 at the $\alpha/2$ level of significance, and label leaf l as fault-prone.

2. For the high end of the interval, let the null hypothesis and alternative hypothesis be

$$\begin{aligned} H_0 &: \quad q(l) \geq 1 - \theta_{\text{high}} \\ H_A &: \quad q(l) < 1 - \theta_{\text{high}} \end{aligned} \tag{13}$$

   If $\Pr(X \leq \text{n}_{\text{fp}} | n_l, 1 - \theta_{\text{high}}) < \alpha/2$ then reject H_0 at the $\alpha/2$ level of significance, and label leaf l as not fault-prone.

3. If we reject neither null hypothesis above, then label leaf l as uncertain at significance α.

The following classification rule based on the binomial distribution in Equation (7) incorporates these criteria with Equation (10).

$$\text{Class}(\mathbf{x}_i) = \begin{cases} \text{nfp} & \text{if } q_{\text{lower}}(L(\mathbf{x}_i)) \leq \epsilon_{L(\mathbf{x}_i)} \\ \text{nfp} & \text{if } \Pr(X \leq n_{\text{fp}} | n_{L(\mathbf{x}_i)}, 1 - \theta_{\text{high}}) < \alpha/2 \\ \text{fp} & \text{if } \Pr(X \leq n_{\text{nfp}} | n_{L(\mathbf{x}_i)}, \theta_{\text{low}}) < \alpha/2 \\ \text{uncertain} & \text{otherwise} \end{cases} \tag{14}$$

at significance α, where n_{fp} and n_{nfp} correspond to leaf $L(\mathbf{x}_i)$.

Equation (10) is an appropriate classification rule when we have a single preferred value of the threshold parameter θ. Equation (14) is an alternative that is appropriate when there is an interval of acceptable thresholds, $[\theta_{\text{low}}, \theta_{\text{high}}]$.

3. Case Study

Table 2 lists the steps of our case study, incorporating the above analysis of uncertain classifications.

3.1. System Description

We conducted a case study (Yuan, 1999) of a very large legacy telecommunications system maintained by professional programmers in a large organization using the procedural-development paradigm, and written in a high-level language (Protel) similar to Pascal. This embedded-computer application included numerous finite-

Table 2. Case study methodology.

1. Collect measurements (see Tables 3–5)
2. Perform preliminary statistical analysis of fault data
3. Select modules for study
4. Transform data into response variable and predictors
5. Empirically select TREEDISC parameters
6. Build classification tree model
7. Choose classification rule's parameter
8. Evaluate accuracy of model
9. Analyze confidence intervals
10. Analyze ambivalent range of thresholds for classification rule
11. Simplify classification tree

state machines. The entire system had significantly more than 10 million lines of code. We studied two consecutive releases. The earlier release was the basis for the training dataset, and the subsequent release was the basis for the evaluation data.

A *module* consisted of a set of functionally related source-code files. An average module had about four files. Fault data was collected at the module level by the problem reporting system. A module was considered fault-prone if any problems discovered by customers resulted in changes to source code in the module, and not fault-prone otherwise. Faults discovered in deployed telecommunications systems are typically extremely expensive, because, in addition to down-time due to failures, visits to remote customer sites are usually required to repair them.

We found that more than 99% of the modules that were unchanged from the prior release had no faults. Consequently, there were too few fault-prone modules in the unchanged set for effective modeling. This case study considered only those modules that were new or had at least one update to source code since the prior release. For modeling, we selected updated modules with no missing data in relevant variables.[3] These modules had several million lines of code in a few thousand modules in each release. The training data set had 3,649 modules. The proportion of modules with no faults among the updated modules of the training dataset was 0.937, and the proportion with at least one fault was 0.063. Such a small set of modules is often difficult for statistical models to identify.

This project used Enhanced Measurement for Early Risk Assessment of Latent Defects (EMERALD), which is a decision-support system that includes software-measurement facilities and software-quality models (Hudepohl et al., 1996). We do not advocate a particular set of software metrics to the exclusion of others recommended in the literature. Pragmatic considerations usually determine the set of available metrics. Because marginal data collection costs were modest, EMERALD provided over 50 source-code metrics (Mayrand and Coallier, 1996) Preliminary data analysis selected product metrics aggregated to the module level that were appropriate for modeling purposes, as listed in Table 3. Counts of procedure calls were derived from call graphs. Some product metrics were measures of a module's control flow graph, which consists of nodes and arcs depicting the flow of control. Other product metrics quantified attributes of statements.

Process metrics listed in Table 4 were tabulated from the configuration management system which maintained records regarding updates by each designer, and from the problem reporting system which maintained records on past problems.

Execution metrics listed in Table 5 were forecast from deployment records (Jones et al., 1999) and laboratory measurements of an earlier release. Future research will refine these metrics.

EMERALD helps software designers and managers to assess risks of embedded software and thereby to improve software quality (Hudephl et al., 1996). It was developed by Nortel Networks in partnership with Bell Canada and others. At various points in the development process, EMERALD's software quality models predict module risk based on available measurements. The models developed here are under evaluation for inclusion in EMERALD.

Table 3. Software product metrics

Symbol	Description
Call graph metrics	
CALUNQ	Number of distinct procedure calls to others
CAL2	Number of second and following calls to others $CAL2 = CAL - CALUNQ$ where *CAL* is the total number of calls
Control flow graph metrics	
CNDNOT	Number of arcs that are not conditional arcs
IFTH	Number of nonloop conditional arcs, i.e., if-then constructs
LOP	Number of loop constructs
CNDSPNSM	Total span of branches of conditional arcs. The unit of measure is arcs
CNDSPNMX	Maximum span of branches of conditional arcs
CTRNSTMX	Maximum control structure nesting
KNT	Number of knots. A "knot" in a control flow graph is where arcs cross due to a violation of structured programming principles
NDSINT	Number of internal nodes (i.e., not an entry, exit, or pending node)
NDSENT	Number of entry nodes
NDSEXT	Number of exit nodes
NDSPND	Number of pending nodes, i.e., dead code segments
LGPATH	Base 2 logarithm of the number of independent paths
Statement metrics	
FILINCUQ	Number of distinct include files
LOC	Number of lines of code
STMCTL	Number of control statements
STMDEC	Number of declarative statements
STMEXE	Number of executable statements
VARGLBUS	Number of global variables used
VARSPNSM	Total span of variables
VARSPNMX	Maximum span of variables
VARUSDUQ	Number of distinct variables used
VARUSD2	Number of second and following uses of variables $VARUSD2 = VARUSD - VARUSDUQ$ where *VARUSD* is the total number of variable uses

3.2. Empirical Results

Candidate predictors were derived from the product, process, and execution metrics. We grouped measurement data for each metric in each dataset, transforming the raw metric values into discrete ordinal predictor values; for this case study, $c_{\max} = 60$ was determined empirically (Khoshgoftaar et al., 1999c). TREEDISC built the tree shown in Figure 1, using $\alpha = 0.01$. Empirical investigation also established other parameters for this case study (Khoshgoftaar and Allen, 2001): partitioning of a leaf stopped when it had 80 or fewer modules; and the minimum number of modules in a new leaf was 40. In Figure 1, each diamond node represents a decision based on one predictor, and each edge represents a merged category of that predictor. Even though

Table 4. Software process metrics

Symbol	Description
DES_PR	Number of problems found by designers during development
BETA_PR	Number of problems found during β testing
DES_FIX	Number of problems fixed that were found by designers in the prior release
BETA_FIX	Number of problems fixed that were found by β testing in the prior release
CUST_FIX	Number of problems fixed that were found by customers in the prior release
REQ_UPD	Number of changes to the code due to new requirements
TOT_UPD	Total number of changes to the code for any reason
REQ	Number of distinct requirements that caused changes to the module
SRC_GRO	Net increase in lines of code
SRC_MOD	Net new and changed lines of code (deleted lines are not counted)
UNQ_DES	Number of different designers making changes
VLO_UPD	Number of updates to this module by designers who had 10 or less total updates in entire company career
LO_UPD	Number of updates to this module by designers who had between 11 and 20 total updates in entire company career
UPD_CAR	Number of updates that designers had in their company careers

Table 5. Software execution metrics

Symbol	Description
USAGE	Deployment percentage of the module
RESCPU	Execution time of an average transaction on a system serving consumers
BUSCPU	Execution time of an average transaction on a system serving businesses
TANCPU	Execution time of an average transaction on a tandem system

the tree was constructed using predictor categories, the labels on edges in the figure have been transformed back to equivalent ranges of raw metric values for easier interpretation. Each circular node is a leaf whose label is noted near the bottom. The number of training modules in each class in each leaf is shown at the bottom, and the value of $1 - \hat{q}(l)$.

Figure 1 includes decision nodes based on product, process, and execution metrics. TREEDISC determined that the range represented by each branch was significantly distinct. For example, from Node 1 four ranges of *VARUSD2* were distinct. In later TREEDISC iterations, it so happened that the most significant predictor of both Node 4 and Node 5 was *UNQ_DES*. A leaf represents the combination of all the predictors in the path from root to leaf. For example, Leaf L5 represents the combination of values of *VARUSD2*, *BETA_PR*, *RESCPU*, and *DES_PR*.

Figure 2 depicts the tradeoff between Type I and Type II misclassification rates, Pr(fp|nfp) and Pr(fp|nfp) respectively, as functions of θ, based on resubstitution of training data into the model. For this case study, we preferred $\theta = 0.92$, which yields approximately equal misclassification rates for the training data. Another study might prefer a different value.

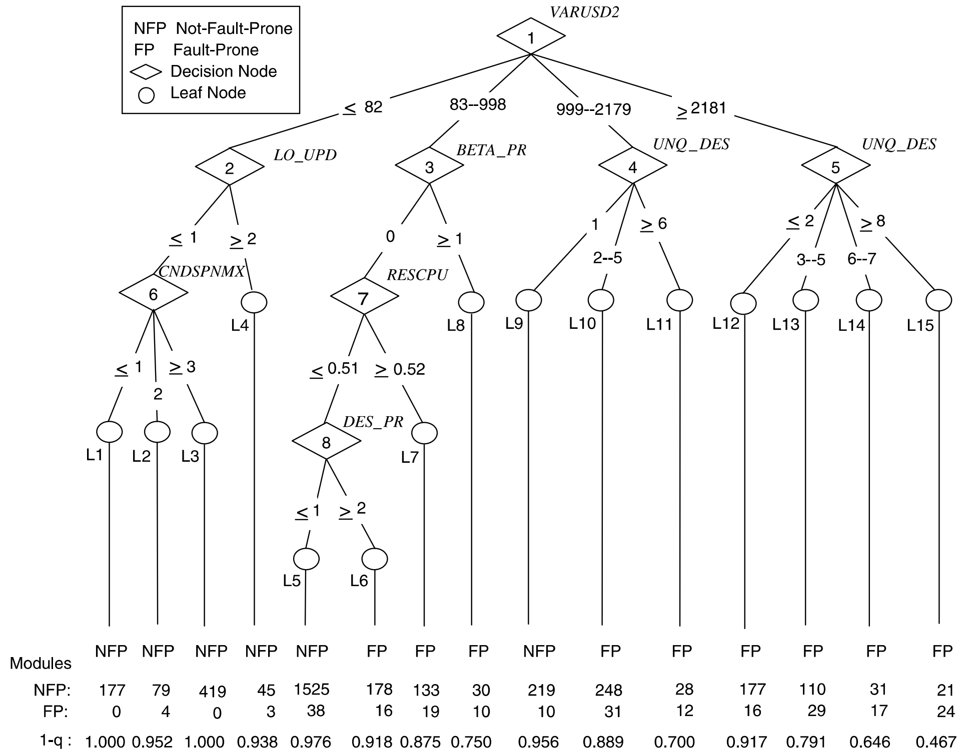

Figure 1. Full classification tree.

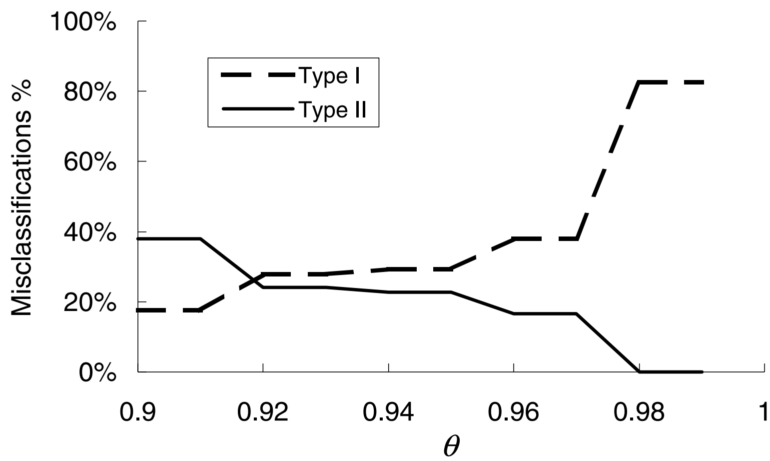

Figure 2. Balancing misclassification rates.

Table 6. Evaluating accuracy.

Actual	Model		Total
	nfp (%)	fp (%)	
nfp	73.1	26.9	100
fp	25.9	74.1	100

Overall misclassification rate: 26.9%
Preferred $\theta = 0.92$.

We simulated practical use of the model by classifying the modules in the evaluation dataset. Table 6 shows the accuracy of the preferred tree (Figure 1) on the evaluation dataset. The rows are actual classifications, and the columns are predicted classifications. Each table entry represents a group of modules in the evaluation dataset that have a certain combination of actual class and predicted class. The table entries show fractions of the row total.

Correct classifications are in the upper left ($\Pr(\text{nfp}|\text{nfp}) = 73.1\%$) and lower right ($\Pr(\text{fp}|\text{fp}) = 74.1\%$). The model achieved a Type I misclassification rate of $\Pr(\text{fp}|\text{nfp}) = 26.9\%$ and a Type II rate of $\Pr(\text{nfp}|\text{fp}) = 25.9\%$ based on evaluation data, in spite of the very small proportion of fault-prone modules in the evaluation dataset. This level of accuracy could be useful to the developers to target reengineering efforts when applying the model to another release. Predictions would be available near the end of β testing, because measurements could be collected at that point in the life cycle.

3.3. Confidence Interval Analysis

The leaf labels in Figure 1 and the results in Table 6 were based on a point estimate of the probability of being not fault-prone, $1 - \hat{q}(l)$, for each leaf, in relation to a preferred threshold, $\theta = 0.92$. Table 7 gives a confidence interval of $q(l)$ for each leaf l, and an analysis of the labels assigned to the leaves. We chose a significance level $\alpha = 0.10$ for a confidence interval.

In this study, we chose $\epsilon_l = 2/n_l$ as a probability that is near zero relative to the number of modules in leaf l. Leaves L1 through L4 were labeled nfp irrespective of the preferred value of θ, because $q(l)$ was indistinguishable from zero at significance $\alpha = 0.10$. In contrast to Figure 1, leaves L6 and L12 (shown bold) were labeled uncertain in Table 7, because the preferred threshold $\theta = 0.92$ is too close to $1 - \hat{q}(l)$ to determine whether the leaf should be labeled fault-prone or not, according to Equation (10). In particular, θ falls within the confidence intervals of $1 - q(l)$ for these leaves at significance $\alpha = 0.10$.

Leaves L1, L2, and L3 were merged categories of the maximum span of conditional structures (*CNDSPNMX*). TREEDISC determined that the category $CNDSPNMX = 2$ was significantly different from the other values. Figure 1 shows

Table 7. Confidence interval analysis.

Leaf	n_{nfp}	n_{fp}	n_l	ϵ_l^{a}	q_{lower}	$\hat{q}(l)$	q_{upper}	$\hat{q}(l)$ ok?	*Class*($\mathbf{x}_i$)[b]
L1	177	0	177	0.011	0.000	0.000	0.017	near zero	nfp
L2	79	4	83	0.024	0.017	0.048	0.107	near zero	nfp
L3	419	0	419	0.005	0.000	0.000	0.007	near zero	nfp
L4	45	3	48	0.042	0.017	0.063	0.154	near zero	nfp
L5	1525	38	1563	0.001	0.018	0.024	0.032	ok	nfp
L6	**178**	**16**	**194**	**0.010**	**0.052**	**0.082**	**0.123**	**ok**	**uncertain**
L7	133	19	152	0.013	0.083	0.125	0.178	ok	fp
L8	30	10	40	0.050	0.142	0.250	0.387	ok	fp
L9	219	10	229	0.009	0.024	0.044	0.073	ok	nfp
L10	248	31	279	0.007	0.082	0.111	0.147	ok	fp
L11	28	12	40	0.050	0.183	0.300	0.440	ok	fp
L12	**177**	**16**	**193**	**0.010**	**0.053**	**0.083**	**0.123**	**ok**	**Uncertain**
L13	110	29	139	0.014	0.153	0.209	0.273	ok	fp
L14	31	17	48	0.042	0.240	0.354	0.483	ok	fp
L15	21	24	45	0.044	0.401	0.533	0.662	ok	fp

[a] $\epsilon_l = 2/n_l$.
[b] $\theta = 0.92$.

that leaf L2 had four fault-prone training modules, but the other two leaves had none. This distinction resulted in a statistically significant χ^2 test in the course of the TREEDISC algorithm. However, the lower confidence limit of all three leaves was less than the corresponding ϵ_l, and thus, we could not distinguish these $q(l)$ from zero. Therefore, we labeled them not fault-prone.

Similarly, even though leaf L4 had only three fault-prone modules, this was enough to yield a significant split on updates by designers who had between 11 and 20 total updates in their entire company careers (*LO_UPD*). However, its lower confidence limit was less than its ϵ_{L4} and thus, irrespective of θ, we labeled it not fault-prone.

Leaves L5 and L6 were merged categories of the number of problems found by designers during development of the current release (*DES_PR*). TREEDISC determined that they were significantly different from each other. Each $q(l)$ was definitely greater than zero. In Figure 1, leaf L6 was labeled fault-prone, however, it was close to the threshold, $1 - \hat{q}(\text{L6}) = 0.918 < 0.92 = \theta$. Because θ falls within the confidence interval of $1 - q(\text{L6})$, we believe that the classification of modules in leaf L6 should be considered uncertain, whereas we have no doubt that those in L5 were properly classified as not fault-prone.

We see in Figure 1 that decision nodes 4 and 5 were both based on the number of different designers making changes (*UNQ_DES*) under different ranges of the number of second and following uses of variables (*VARUSD*2). Data analysis revealed that *VARUSD*2 was strongly correlated with module size. Decision nodes 4 and 5 represent large and very large modules. TREEDISC determined that leaves L9 through L15 were all significantly different from each other based on combinations

of *VARUSD2* and *UNQ_DES*. Based on Table 7, Leaf L9 was properly labeled not fault-prone ($1 - \hat{q}(\text{L9}) = 0.956 > 0.92 = \theta$). Similarly, leaves L10, L11, L13, L14, and L15 were properly labeled fault-prone, because θ was not near $1 - \hat{q}(l)$ in each case. However, leaf L12 was near the preferred threshold, $1 - \hat{q}(\text{L12}) = 0.917 < 0.92 = \theta$, and therefore, the modules in leaf L12 had an uncertain classification. In this study, a confidence interval analysis affected the classification of about 10.6% of the training modules (i.e., those in L6 and L12).

3.4. Ambivalent Threshold Analysis

Recall that the choice of $\theta = 0.92$ above was a preference. Because 2 shows that the misclassification rates were somewhat balanced over the range of approximately $\theta_{\text{low}} = 0.92 \leq \theta \leq 0.95 = \theta_{\text{high}}$, other values of θ in this range might also be satisfactory. Table 8 gives the results for each leaf when the threshold is an interval.

In Figure 1, we see that $1 - \hat{q}(\text{L4}) = 0.937 < 0.95 = \theta_{\text{high}}$, which might imply that its label should be uncertain. However, the confidence interval analysis in Table 7, according to Equation (14), showed that $q(\text{L4})$ is not distinguishable from zero, and thus, we labeled it not fault-prone irrespective of θ.

In Table 8, the label of each leaf is the same as Table 7 except for leaf L9 (shown bold). L9's label is uncertain, because its value of $1 - \Pr(\text{fp})$ is close to the upper end of the preferred interval, ($1 - \hat{q}(\text{L9}) = 0.956 > 0.95 = \theta_{\text{high}}$), according to Equation (14).

Table 8. Ambivalent threshold analysis.

Leaf	n_{nfp}	n_{fp}	n_l	Pr(fp)[a]	Pr(nfp)[b]	Class($\mathbf{x}_i$)[c]
L1	177	0	177	1.14E-04	1.000	nfp[d]
L2	79	4	83	0.599	0.907	nfp[d]
L3	419	0	419	4.46E-10	1.000	nfp[d]
L4	45	3	48	0.782	0.750	nfp[d]
L5	1525	38	1563	2.06E-07	1.000	nfp
L6	178	16	194	0.982	0.487	uncertain
L7	133	19	152	1.000	0.035	fp
L8	30	10	40	1.000	0.001	fp
L9	**219**	**10**	**229**	**0.403**	**0.990**	**uncertain**
L10	248	31	279	1.000	0.040	fp
L11	28	12	40	1.000	4.44E-05	fp
L12	177	16	193	0.983	0.479	uncertain
L13	110	29	139	1.000	1.64E-06	fp
L14	31	17	48	1.000	8.46E-08	fp
L15	21	24	45	1.000	3.34E-15	fp

[a] $\Pr(\text{fp}) = \Pr(X \leq n_{\text{fp}} | n_l, 1 - \theta_{\text{high}})$.
[b] $\Pr(\text{nfp}) = \Pr(X \leq n_{\text{nfp}} | n_l, \theta_{\text{low}})$.
[c] Preferred $\theta \in [0.92, 0.95]$.
[d] $q(l)$ near zero (see Table 7).

In this study, using an interval instead of one preferred threshold value affected the classification of the 229 modules in L9, about 6.3% of the training modules.

3.5. Discussion

This case study illustrated the possibility of subsets of modules with significantly uncertain classification. Leaves L6 and L12 accounted for 387 modules in the training data set (10.6%). Figure 3 depicts the simplified tree for $\theta = 0.92$, but shows the uncertain leaves (L6 and L12).

Leaves L1 and L3 had no fault-prone training modules and so we labeled them not fault-prone. A confidence interval analysis at significance $\alpha = 0.10$ of leaves L2 and L4 revealed we could not distinguish their $q(l)$ from zero, and so we labeled them not fault-prone. If we had preferred the interval $\theta \in [0.92, 0.95]$, leaf L9 would also be labeled uncertain.

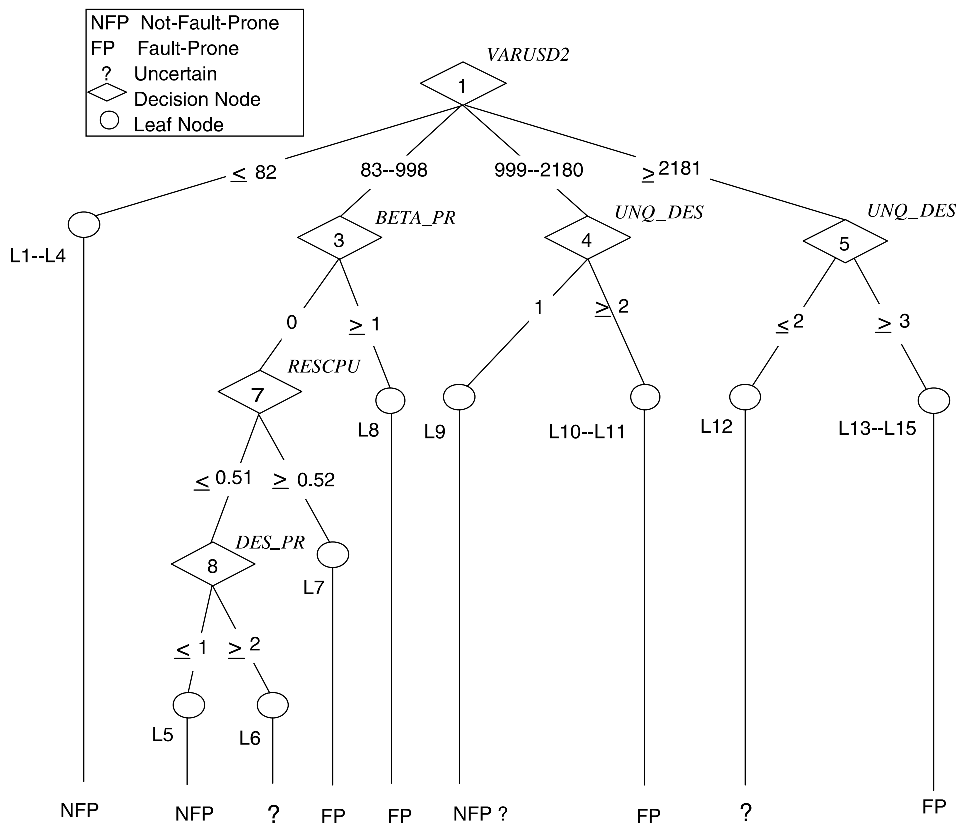

Figure 3. Simplified classification tree.

TREEDISC is especially amenable to identifying uncertain classifications, because the algorithm for building a tree is based on a well-known statistical test which identifies subsets of modules with distinctive attributes, but does not mandate the classification rule.

Rather than just two options, *enhance a module* or not, a project could devise an intermediate improvement strategy that would be appropriate for modules with uncertain classification. For example, preliminary reviews of these modules might ascertain which have high risk and which do not. Those evaluated as high risk could then be enhanced in the same manner as other modules classified as fault-prone. Thus, discovering modules with uncertain classifications allows one to employ sophisticated enhancement strategies to resolve uncertainties, which in turn, could improve the ultimate effectiveness and efficiency of enhancements.

4. Conclusions

Developing high-quality software often entails time-consuming and costly development processes. One way to improve quality is to target enhancement activities to those software modules that a software-quality model predicts are most likely to have problems (Hudepohl et al., 1996). This paper presents statistical techniques for assessing which predictions by a classification-tree model should be considered uncertain. We employ the TREEDISC algorithm (SAS Institute Staff, 1995) to model software quality based on χ^2 tests. TREEDISC also allows multiway branching, possibly resulting in a more parsimonious model than binary branching.

A case study of a large legacy telecommunications system illustrated how TREEDISC models can be used not only to classify software modules, but also to identify modules with uncertain quality. A model predicted whether a module was likely to have faults discovered by customers, or not, based on software product, process, and execution metrics. We simulated practical use of the model near the end of β testing by classifying the modules in the evaluation dataset. This would be valuable for mission-critical systems, such as telecommunications equipment. The model achieved a Type I misclassification rate of $\Pr(\text{fp}|\text{nfp}) = 26.9\%$ and a Type II rate of $\Pr(\text{nfp}|\text{fp}) = 25.9\%$ on evaluation data, in spite of the very small proportion of fault-prone modules. Prediction of fault-prone modules is very important to the developers of telecommunications systems, because faults discovered by customers are usually extremely expensive.

We assessed whether leaf labels were appropriately assigned by Equation (6) by examining the details of the full tree, and through confidence-interval analysis (Equation (10)), we found subsets of modules with significantly uncertain classification (about 10.6% of the training modules). If one were ambivalent about the preferred threshold for fault-prone modules, because the accuracy of the model was about the same over a range of thresholds, then we found that ambivalent-threshold analysis (Equation (14)) indicated an additional 6.3% of the training modules have significantly uncertain classification. For the case study's model, we

concluded that a total of 16.9% of the training modules (three leaves) have uncertain classification. When applying the model to a current project, all modules that fall in the leaves labeled "uncertain" would have an uncertain classification. Discovering such modules allows one to employ sophisticated enhancement strategies to resolve uncertainties. TREEDISC is especially well suited to identifying uncertain classifications.

Future research will apply uncertain classification to other modeling techniques, and will investigate various software enhancement strategies. Future research will also investigate the interaction between transformations, such as principal components analysis, and methods for preprocessing metrics into ordinal predictors.

Acknowledgements

We thank the EMERALD team for collecting the case-study data. This work was supported in part by a grant from Nortel through the Software Reliablity Engineering Department, Research Triangle Park, North Carolina. The findings and opinions in this paper belong solely to the authors, and are not necessarily those of the sponsor. Moreover, our results do not in any way reflect the quality of the sponsor's software products.

Notes

1. The latest version of TREEDISC is available as part of SAS Institute's Enterprise Data Miner product. CHAID has also been implemented as part of the SPSS package.
2. In our application, because the overall proportion of not fault-prone modules is usually large, θ is usually large also. Thus, there is no practical interest in whether or not $1 - \hat{q}(l)$ is near zero.
3. The fraction of modules with missing data was very small. Missing data was due to practical problems with tools, or incomplete recordkeeping.

References

Basili, V. R., Briand, L. C., and Melo, W. 1996. A Validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering* 22(10): 751–761.

Briand, L. C., Basili, V. R., and Hetmanski, C. J. 1993. Developing interpretable models with optimized set reduction for identifying high-risk software components. *IEEE Transactions on Software Engineering* 19(11): 1028–1044.

Ebert, C. 1996. Classification techniques for metric-based software development. *Software Quality Journal* 5(4): 255–272.

Gokhale, S. S., and Lyu, M. R. 1997. Regression tree modeling for the prediction of software quality. In: H. Pham (ed.): *Proceedings of the Third ISSAT International Conference on Reliability and Quality in Design*. Anaheim, CA: International Society of Science and Applied Technologies, pp. 31–36.

Hawkins, D. M., and G. V. Kass: 1982. Automatic interaction detection. In: D. M. Hawkins (ed.): *Topics in Applied Multivariate Analysis*. Cambridge: Cambridge University Press, Chap 5. pp. 269–302.

Hudepohl, J. P., Aud, S. J., Khoshgoftaar, T. M., Allen, E. B., and Mayrand, J. 1996. EMERALD: Software metrics and models on the desktop. *IEEE Software* 13(5): 56–60.

Jones, W. D., Hudepohl, J. P., Khoshgoftaar, T. M., and Allen, E. B. 1999. Application of a usage profile in software quality models. In: *Proceedings of the Third European Conference on Software Maintenance and Reengineering*. Amsterdam, Netherlands: IEEE Computer Society, pp. 148–157.

Kass, G. V. 1980. An exploratory technique for investigating large quantities of categorical data. *Applied Statistics* 29: 119–127.

Khoshgoftaar, T. M., and Allen, E. B. 2000. A practical classification rule for software quality models. *IEEE Transactions on Reliability* 49(2): 209–216.

Khoshgoftaar, T. M., and Allen, E. B. 2001. Controlling overfitting in classification-tree models of software quality. *Empirical Software Engineering: An International Journal* 6(1): 59–79.

Khoshgoftaar, T. M., and Lanning, D. L. 1995. A neural network approach for early detection of program modules having high risk in the maintenance phase. *Journal of Systems and Software* 29(1): 85–91.

Khoshgoftaar, T. M., Allen, E. B., Bullard, L. A., Halstead, R., and Trio, G. P. 1996a. A tree-based classification model for analysis of a military software system. In: *Proceedings of the IEEE High-Assurance Systems Engineering Workshop*. Niagara on the Lake, Ontario, Canada: IEEE Computer Society, pp. 244–251.

Khoshgoftaar, T. M., Allen, E. B., Jones, W. D., and Hudepohl, J. P. 1999a. Which software modules have faults that will be discovered by customers? *Journal of Software Maintenance: Research and Practice* 11(1): 1–18.

Khoshgoftaar, T. M., Allen, E. B., Kalaichelvan, K. S., and Goel, N. 1996b. Early quality prediction: A case study in telecommunications. *IEEE Software* 13(1): 65–71.

Khoshgoftaar, T. M., Allen, E. B., Naik, A., Jones, W. D., and Hudepohl, J. P. 1998. Using classification trees for software quality models: Lessons Learned. In: *Proceedings of the Third IEEE International High-Assurance Systems Engineering Symposium*. Bethesda, Maryland, USA: IEEE Computer Society, pp. 82–89.

Khoshgoftaar, T. M., Allen, E. B., Yuan, X., Jones, W. D., and Hudepohl, J. P. 1999b. Assessing Uncertain Predictions of Software Quality. In: *Proceedings: Sixth International Software Metrics Symposium*. Boca Raton, Florida, USA: IEEE Computer Society, pp. 159–168.

Khoshgoftaar, T. M., Allen, E. B., Yuan, X., Jones, W. D., and Hudepohl, J. P. 1999c. Preparing measurements of legacy software for predicting operational faults. In: *Proceedings: International Conference on Software Maintenance*. Oxford, England: IEEE Computer Society, pp. 359–368.

Khoshgoftaar, T. M., Shan, R., and Allen, E. B. 2000a. Improving Tree-Based Models of Software Quality with Principal Components Analysis. In: *Proceedings: Eleventh International Symposium on Software Reliability Engineering*. San Jose, California, USA: IEEE Computer Society, pp. 198–209.

Khoshgoftaar, T. M., Yuan, X., and Allen, E. B. 2000b. Balancing misclassification rates in classification-tree models of software quality. *Empirical Software Engineering: An International Journal* 5(4): 313–330.

Mayrand, J., and Coallier, F. 1996. System acquisition based on software product assessment. In: *Proceedings of the Eighteenth International Conference on Software Engineering*. Berlin: IEEE Computer Society, pp. 210–219.

Ohlsson, M. C., and Wohlin, C. 1998. Identification of green, yellow and red legacy components. In: *Proceedings of the International Conference on Software Maintenance*. Bethesda, MD, USA: IEEE Computer Society, pp. 6–15.

Ohlsson, N., Zhao, M., and Helander, M. 1998. Application of multivariate analysis for software fault prediction. *Software Quality Journal* 7: 51–66.

Quinlan, J. R. 1986. Induction of decision trees. *Machine Learning* 1: 81–106.

SAS Institute Staff. 1995. TREEDISC macro (beta version). Technical report, SAS Institute, Inc., Cary, NC: Documentation with macros.

Schneidewind, N. F. 1995. Software metrics validation: Space shuttle flight software example. *Annals of Software Engineering* 1: 287–309.

Schneidewind, N. F. 1997. Software metrics model for integrating quality control and prediction. In: *Proceedings of the Eighth International Symposium on Software Reliability Engineering*. Albuquerque, NM, USA: IEEE Computer Society.

Seber, G. A. F. 1984. *Multivariate Observations*. New York: John Wiley and Sons.

Selby, R. W., and Porter, A. A. 1988. Learning from examples: Generation and evaluation of decision trees for software resource analysis. *IEEE Transactions on Software Engineering* 14(12): 1743–1756.

Szabo, R. M., and Khoshgoftaar, T. M. 1995. An assessment of software quality in a C++ environment. In: *Proceedings of the Sixth International Symposium on Software Reliability Engineering*. Toulouse, France: IEEE Computer Society, pp. 240–249.

Takahashi, R., Muraoka, Y., and Nakamura, Y. 1997. Building software quality classification trees: Approach, Experimentation, Evaluation. In: *Proceedings of the Eighth International Symposium on Software Reliability Engineering*. Albuquerque, NM, USA: IEEE Computer Society, pp. 222–233.

Troster, J., and J. Tian 1995. Measurement and defect modeling for a legacy software system. *Annals of Software Engineering* 1: 95–118.

Yuan, X. 1999. Modeling software quality with TREEDISC. Master's thesis, Florida Atlantic University, Boca Raton, Florida. Advised by Taghi M. Khoshgoftaar.

Zar, J. H. 1984. *Biostatistical Analysis*. 2nd Ed. Englewood Cliffs, NJ: Prentice-Hall.