Controlling Overfitting in Classification-Tree Models of Software Quality

Taghi M. Khoshgoftaar (taghi@cse.fau.edu)
Florida Atlantic University, Boca Raton, Florida USA

Edward B. Allen (edward.allen@computer.org)*
Mississippi State University, Mississippi USA

Abstract. Predicting which modules are likely to have faults during operations is important to software developers, so that software enhancement efforts can be focused on those modules that need improvement the most. Modeling software quality with classification trees is attractive because they readily model nonmonotonic relationships. In this paper, we apply the TREEDISC algorithm which is a refinement of the CHAID algorithm to build classification-tree models. CHAID-based algorithms differ from other classification-tree algorithms in their reliance on chi-squared tests when building the tree.

Classification-tree models are vulnerable to overfitting, where the model reflects the structure of the training data set too closely. Even though a model appears to be accurate on training data, if overfitted, it may be much less accurate when applied to a current data set. To account for the severe consequences of misclassifying fault-prone modules, our measure of overfitting is based on expected costs of misclassification, rather than the total number of misclassifications.

We conducted a case study of a very large telecommunications system. A two-way analysis of variance with repetitions found that TREEDISC's significance level was highly related to overfitting, and can be used to control it. Moreover, the minimum number of modules in a leaf also influenced the degree of overfitting.

Keywords: empirical studies, software metrics, software quality, fault-prone modules, classification trees, TREEDISC, CHAID

1. Introduction

Software quality models are tools for focusing software enhancement efforts. Such models yield timely predictions of which modules are likely to have faults on a module-by-module basis, enabling one to target high-risk modules. Mission-critical embedded software, such as telecommunications systems, can especially benefit from such predictions due to the high cost of correcting problems discovered by customers. A software quality model is developed using measurements and fault data from a past release. The resulting model is then applied to modules currently under development (Schneidewind, 1992).

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

The field of software metrics posits that characteristics of software products and their development processes strongly influence the quality of the released product (Evanco and Agresti, 1994; Henry et al., 1994), and its residual faults, in particular. The more complex the product is, the more likely that developers will make mistakes. Research has found that process attributes and forecasts of execution metrics (Jones et al., 1999) can also be significantly associated with faults. Because product and process characteristics can be measured earlier than faults during operations, software metrics can guide improvements to quality before the product is released and during development of the next release (Schneidewind, 1998).

One's expectation for model accuracy is generally formed by experience with the training data set. *Overfitting* is characterized by an adverse deviation from that base of experience. An overfitted model reflects the structure of the training data set too closely. Even though a model appears to be accurate on training data, if overfitted, it may be much less accurate when applied to a current data set. Moreover, a software engineering interpretation of an overfitted software quality model's structure may not be true for similar systems or subsequent releases.

The focus of this paper is classification-tree models of software quality based on software product, process, and execution metrics. In particular, we apply the TREEDISC algorithm (SAS Institute staff, 1995),[1] which is a refinement of the CHAID algorithm (Kass, 1980), to build classification-tree models. CHAID-based algorithms, and TREEDISC in particular, differ from other classification-tree algorithms in their reliance on chi-squared tests when building the tree.

Classification-tree models are often vulnerable to overfitting. Our goal is to control overfitting of TREEDISC models by choosing appropriate parameter values. This paper presents an analysis of variance that assessed which TREEDISC parameters are significantly related to overfitting, and can be used to control it. The empirical experiments were based on a case study of a very large telecommunications system. A module was considered fault-prone if any faults were discovered by customers. Data on updated modules from one release formed the training data set, and data on three subsequent releases formed evaluation data sets.

The remainder of this paper summarizes related work in applications of classification trees to software quality modeling, gives details on

[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.

modeling with TREEDISC, specifies our experimental design, describes our case study, and draws conclusions.

2. Classification Trees

A classification tree is an algorithm of decision rules represented by an abstract tree, to classify an object. 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 of the tree 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, the algorithm traverses a downward path in the tree, beginning at the root. 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.

Alternative classification techniques used in software quality modeling include discriminant analysis (Khoshgoftaar et al., 1996b), the discriminative power technique (Schneidewind, 1995), logistic regression (Basili et al., 1996), pattern recognition (Briand et al., 1992), artificial neural networks (Khoshgoftaar and Lanning, 1995), and fuzzy classification (Ebert, 1996). A classification tree differs from these in the way it models nonmonotonic relationships between class membership and combinations of variables. Moreover, a classification tree is often readily interpreted.

Several algorithms for building classification trees have appeared in the software engineering literature. Selby and Porter (1988) used the ID3 algorithm (Quinlan, 1986) which recursively partitions the training set of modules using an entropy-based criterion. In this application, preprocessing transforms measurements into nominal predictors. Each decision node consists of a selected predictor and each outgoing edge represents a possible value (category) of that predictor. The algorithm selects the predictor that minimizes the weighted average of the entropy of each possible child subtree. The entropy calculation is based on the probability of a module being fault-prone or not, and the weighting factor is the fraction of modules in each child subtree. The entropy criterion is a mechanism to avoid overfitting.

Takahashi, Muraoka, and Nakamura(1997) extend the ID3 algorithm by applying Akaike Information Criterion procedures (Akaike, 1987) to prune the tree. The pruned classification tree can be more stable than that built by the original algorithm alone, without any significant decrease in the proportion of correct classifications. Pruning is a strategy for controlling overfitting.

Case studies by Gokhale and Lyu (1997) and by Troster and Tian (1995) used regression trees with continuous response variables and continuous predictors. Gokhale and Lyu suggest a way to classify modules. This algorithm is implemented in the S-Plus system. The algorithm recursively partitions the training set of modules. When creating a decision node, this algorithm makes a binary partition of the modules based on a selected predictor and a cutoff threshold. The algorithm chooses the predictor-threshold combination that minimizes the total deviance of the partitioned subsets (Clark and Pregibon, 1992). Gokhale and Lyu (1997) use cost-complexity based on deviance to select the proper size for the treeTroster and Tian (1995) suggest that after building a tree, the analyst perform "intelligent pruning" when a decision node yields little insight, in the view of the analystChoosing a tree's size is a surrogate for controlling overfitting.

Our research team has applied the Classification And Regression Trees (CART) algorithm (Breiman et al., 1984) to software quality modeling (Khoshgoftaar et al., 1999a; Khoshgoftaar et al., 1998). Kitchenham briefly reports using CART to model software project productivity (Kitchenham, 1998). It is implemented as a supplementary module for the SYSTAT package (Steinberg and Colla, 1995). CART builds a tree from continuous ordinal predictors. The algorithm recursively partitions ("splits") the training set into two leaves until a stopping criterion applies to every leaf node. A goodness-of-split criterion minimizes the heterogeneity ("node impurity") of each leaf at each stage of the algorithm. CART's default goodness-of-split criterion is the "Gini index of diversity" which is based on probabilities of class membership (Breiman et al., 1984). CART first builds a maximal tree and then prunes it based on a cost function to avoid overfitting. As mentioned above, pruning is a way to control overfitting.

The CHAID algorithm (Hawkins and Kass, 1982; Kass, 1980) builds classification-tree models based on discrete ordinal predictors (Khoshgoftaar et al., 1996a; Khoshgoftaar et al., 1999d; Khoshgoftaar et al., 1999e). It recursively partitions the training set of modules by finding the most significant partitioning of predictors' original categories with respect to the response variable based on chi-squared tests. TREEDISC, which is the focus of this paper, is a refinement of CHAID by SAS Institute(1995) Like ID3, TREEDISC allows multiway splits, but it minimizes the adjusted p-value of chi-squared tests to determine the ranges. TREEDISC has several parameters that may control overfitting.

In summary, each algorithm has its own characteristics in terms of (1) the scale type of the predictors (nominal, ordinal, etc.) and whether continuous or discrete, (2) the criteria for partitioning the training data and thus, creation of branches, (3) the stopping criteria, and (4) the pruning method, if any. These in turn influence preprocessing of data, the parsimony of the overall tree in terms of nodes and leaves, and methods for controlling overfitting.

3. Modeling with TREEDISC

3.1. 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. (Although we need only ordinal predictors, the full TREEDISC algorithm allows nominal predictors also (SAS Institute staff, 1995).)

We used *grouping* to transform numeric metrics into discrete ordinal predictors (Khoshgoftaar et al., 1999e). Let $c_{\max}$ be the maximum number of predictor categories. Let $rank_{ij}$ be an i^{th} module's rank within the data set according to the j^{th} raw metric and let n be the number of modules in the data set. Ties are assigned a common rank. Predictor values are calculated by the following.

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

The j^{th} 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 numbers 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 data set using the same empirically determined value of $c_{\max} = 60$ (Khoshgoftaar et al., 1999e). Future research will investigate other transformations.

Because arbitrary ordinal categories do not convey software engineering concepts, we apply the reverse transformation after building a model to facilitate interpretation of its structure.

3.2. 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 macro package for the SAS System. Improvements to the CHAID algorithm include adjusting the chi-squared 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 complete classification tree is built by invoking TREEDISC with all the training data-set objects assigned to a single leaf node. This single node becomes the root of the classification tree. The algorithm recursively partitions ("splits") the training data set'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. The significance threshold, α, determines when to stop splitting. The minimum number of objects in a node to qualify for further subdivision, *BranchLimit*, also constrains splitting. The size of a tree is primarily controlled by the stopping criteria, rather than by pruning as with CART (Khoshgoftaar et al., 1998), or the approach of Takahashi et al. (1997).

The range of predictor values associated with each edge consists of adjacent predictor categories merged together. The TREEDISC algorithm finds the most significant partitioning of a predictor's original categories with respect to the response variable, creating merged predictor categories. The minimum number of objects in a leaf, *LeafMin*, constrains the merging of categories. For a given predictor, let c be the number of original predictor categories, let k be the number of merged predictor categories, and in this context, let $r = 2$ be the number of response-variable classes.

Significance is evaluated using an adjusted p-value of the chi-squared statistic, X, of the predictor's $k \times r$ merged contingency table, under the null hypothesis, H_0, that the predictor is not related to the response variable. Let f_{ij} be the frequency of the ij cell and let $\hat{f}_{ij}$ be the expected frequency under H_0. The chi-squared statistic is calculated by the following (Berenson et al., 1983).

$$X = \sum_{i=1}^{k}\sum_{j=1}^{r} \frac{(f_{ij} - \hat{f}_{ij})^2}{\hat{f}_{ij}} \qquad (2)$$

The adjusted p-value is calculated by the following.

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

A significance test rejects the null hypothesis that the response variable is independent of the predictor, if $p < \alpha$. In the CHAID algorithm, Kass (1980) uses a Bonferroni adjustment to a basic chi-squared test, represented by the combination in the first term of the minimization. The minimization is due to Gabriel's adjustment (Hawkins and Kass, 1982).

TREEDISC allows multiway splits and minimizes the adjusted p-value of chi-squared tests to determine the merged predictor categories. Multiway splits allow more parsimonious modeling than binary splits for nonmonotonic relationships between predictors and the response variable.

3.3. 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 all the modules in the training data set are partitioned among the leaves according to the structure of the tree. Moreover, the actual class of each module in the training data set 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 i^{th} module's vector of predictor values. Let $L(\mathbf{x}_i)$ be the leaf that the i^{th} module falls into according to the structure of the tree. TREEDISC's default rule for classifying a module is the following.

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

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 have proposed a general classification rule in the context of discriminant analysis (Khoshgoftaar and Allen, 2000), and the following applies it to classification trees (Khoshgoftaar et al., 1999c; Khoshgoftaar et al., 1999d).

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

where ζ is chosen to achieve a preferred balance between the Type I and Type II misclassification rates. The TREEDISC default rule in Equation (4) is equivalent to $\zeta = 1$. With various classification techniques, we have observed a tradeoff between Type I and Type II misclassification rates as functions of ζ (Khoshgoftaar and Allen, 2000; Khoshgoftaar et al., 1999b; Khoshgoftaar et al., 1998). As $\Pr(nfp|fp)$ goes down, $\Pr(fp|nfp)$ goes up, and conversely. For example, if one chooses ζ such that $\Pr(fp|nfp) \approx \Pr(nfp|fp)$, then the rule approximately minimizes the larger of the misclassification rates (the min-max rule). In practice, we can achieve only approximate equality due to finite data sets and discrete predictors. Another balance between misclassification rates may be preferred by a project in another situation.

We choose a preferred value of ζ empirically. Given a candidate value of ζ, we estimate misclassification rates $\Pr(fp|nfp)$ and $\Pr(nfp|fp)$ by resubstitution of the training data set into the model. If the balance is not satisfactory, we select another candidate value of ζ and estimate again, until we arrive at the preferred ζ for the project. This procedure is straightforward in practice, because the misclassification rates are monotonic functions of ζ.

From a Bayesian viewpoint, the class proportions of the population are information which is known prior to applying a model, i.e., the prior probabilities of class membership, π_{nfp} and π_{fp}. We estimate prior probabilities, π_{nfp} and π_{fp}, from proportions in the training data set.

In software engineering practice, the penalty for a Type II misclassification is often much more severe than for a Type I. A software enhancement technique, such as extra reviews, typically has modest direct cost per module. The cost of a Type I misclassification, C_I is the effort wasted on a not fault-prone module. On the other hand, the cost of a Type II misclassification, C_{II}, is the lost opportunity to correct faults early. The consequences of letting a fault go undetected until after release can be very expensive indeed. In this paper, we model the costs of misclassifying a module, C_I and C_{II}, as constants (Khoshgoftaar and Allen, 1998). Future research will consider more sophisticated cost functions.

The expected cost of misclassification of one module, *ECM*, takes prior probabilities and costs of misclassification into account.

$$ECM \;=\; C_I \,\Pr(fp|nfp)\,\pi_{nfp} + C_{II}\,\Pr(nfp|fp)\,\pi_{fp} \tag{6}$$

Hence, a classification rule that minimizes the expected cost of misclassification (Seber, 1984) is

$$Class(\mathbf{x}_i) = \begin{cases} nfp & \text{if } \frac{1-\hat{q}(L(\mathbf{x}_i))}{\hat{q}(L(\mathbf{x}_i))} \geq \left(\frac{C_{II}}{C_I}\right)\left(\frac{\pi_{fp}}{\pi_{nfp}}\right) \\ fp & \text{otherwise} \end{cases} \tag{7}$$

Note that when applying a model to a current data set, one should use a priors ratio, π_{fp}/π_{nfp}, that is true for the data set being classified. This is an issue when one expects the current data set to have a different priors ratio than the training data set (El Emam et al., 1999). In this paper, we expect the priors ratio of the training data set to hold for the evaluation data sets as well.

When actual costs can be estimated, then Equation (7) would be a reasonable preferred classification rule. However, in many organizations, it is not practical to quantify the cost ratio. In such cases, choosing a preferred value of ζ according to Equation (5) is a subjective judgment. When ζ is chosen subjectively, one can interpret it in the context of this minimum-expected-cost rule (Equation (7)) to imply a subjective assessment of the (unknown) cost ratio.

$$\frac{C_{II}}{C_I} \;=\; \zeta\left(\frac{\pi_{nfp}}{\pi_{fp}}\right) \tag{8}$$

3.4. A Measure of Overfitting

In the statistics literature, overfitting is often defined as *bias*, namely, the difference between a model's actual misclassification rate and its purported rate (Stone and Rasp, 1993). Overfitting can be detected by a statistically unbiased estimate of model accuracy compared to its possibly biased accuracy on training data. The difference is a measure of the degree of *overfitting*. The total number of misclassifications can be a satisfactory measure of a model's accuracy if the proportions of each class are approximately equal and the two kinds of costs of misclassification are about equal. However, in software engineering applications, the proportion of fault-prone modules is often small and the practical penalty for misclassifying a fault-prone module as not fault-prone can be quite serious after release. Thus, we prefer the expected cost of misclassification as a measure of a model's accuracy. As a measure of overfitting, we propose the difference between expected costs of misclassification estimated by an independent evaluation data set and a training data set, normalized by the cost of a Type I misclassification. By Equations (6) and (8),

$$\Delta ECM = \frac{1}{C_I}(ECM_{\text{eval}} - ECM_{\text{train}}) \tag{9}$$

$$\begin{aligned} \Delta ECM = {} & \pi_{nfp}(\Pr(fp|nfp)_{\text{eval}} - \Pr(fp|nfp)_{\text{train}}) \\ & + \zeta\, \pi_{nfp}(\Pr(nfp|fp)_{\text{eval}} - \Pr(nfp|fp)_{\text{train}}) \end{aligned} \tag{10}$$

where subscripts indicate the training data set ($_{\text{train}}$) and an evaluation data set ($_{\text{eval}}$). In our application, the unit of measure is the cost of misclassifying a not fault-prone module. If ΔECM is positive, then the accuracy of the model on the training data set is somewhat misleading. When ΔECM is zero, we have avoided overfitting. If ΔECM is negative, then the accuracy of the model on the evaluation data set is better than on the training data set, a pleasant surprise.

4. Experimental Design

When a classification tree grows very large using many predictors, it can easily overfit the training data set, and fail to yield good predictions when applied to current data sets. Our working hypothesis is that choosing appropriate parameter values will control the growth of the tree to avoid overfitting.

An alternative approach to controlling overfitting is exemplified by CART (Breiman et al., 1984), which grows a maximal tree and then uses a heuristic to prune the tree in such a way that overfitting is controlled. CART's heuristic appears to be somewhat inappropriate in our application, because it is based on the overall number of misclassifications (Breiman et al., 1984), without consideration of prior probabilities or costs of misclassification which are far from equal in software quality data. Anecdotal evidence in our research (Mao, 2000) indicates that CART tends to prune the tree too much. A comparison of various pruning algorithms and parameters for controlling overfitting is a topic for future research.

The goal of our experiments is to assess TREEDISC parameters for controlling overfitting. Our measure of overfitting is ΔECM, as defined by Equation (10), and our experimental factors are TREEDISC's significance level, α, and the minimum number of objects in a leaf, *LeafMin*.

In our case study, we chose to use TREEDISC's default, $BranchLimit = 2 \times LeafMin$. Thus, it is not an independent factor here.

Each TREEDISC model with its parameter settings is an *experiment*. Because we used a small number of values (*levels*) for the parameters in all combinations, we model *LeafMin* and α as nominal factors, A and B, respectively. We modeled overfitting as a real response variable, $y = \Delta ECM$. We use a two-way analysis of variance (ANOVA) with replications (Berenson et al., 1983), which models the response variable as

$$y_{ijk} \;=\; \mu + A_j + B_i + AB_{ij} + e_{ijk} \tag{11}$$

where y_{ijk} is the response for the k^{th} replication of the experiment with the j^{th} level of *LeafMin* and the i^{th} significance level, μ is the mean response, A_j is the effect of the j^{th} level of *LeafMin*, B_i is the effect of the i^{th} significance level, AB_{ij} is the effect of the interaction between *LeafMin* and the significance level, and e_{ijk} is the experimental error. All experiments use the same training data set to build the tree and in assessing overfitting, but repetitions are based on evaluation data sets.

We perform statistical tests of the null hypotheses (H_0) that each factor is independent of overfitting, for example, that $A_j = 0$ for all j (Berenson et al., 1983). If we can show there is a very small probability that an H_0 is true, then we have succeeded in identifying a parameter for controlling overfitting.

The ratios of mean squares of factors (e.g., *MSA*) to mean squared error (*MSE*) are distributed according to F distributions (Berenson et al., 1983). We reject the corresponding null hypothesis, H_0, at a given significance level, if the computed ratio is higher than the selected value from the F distribution, concluding that the factor is statistically significant at that level. Alternatively, one can find the minimum level where the factor is significant. This provides a way to assess which parameters, if any, are significantly related to overfitting.

5. Case Study

5.1. System Description

We conducted a case study 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-state machines. The entire system had significantly more than ten million lines of code. We studied four

Table I. Distribution of faults discovered by customers.
Percentage of updated modules

	Release			
Faults	1	2	3	4
0	93.7%	95.3%	98.7%	97.7%
1	5.1%	3.9%	1.0%	2.1%
2	0.7%	0.7%	0.2%	0.2%
3	0.3%	0.1%	0.1%	0.1%
4	0.1%	*		
6	*			
9	*			

* one module

consecutive releases which we labeled 1 through 4. Data on Release 1 formed our training data set, and data on the subsequent releases were evaluation data sets.

A *module* consisted of a set of related source-code 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. This means that detecting fault-prone modules early in development has very high priority.

We found that more than 99% of the modules 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. (The fraction of modules with missing data was small. Missing data was due to practical problems with tools, or incomplete recordkeeping.) The set of updated modules had several million lines of code in each release. Release 1 had 3,649 modules; Release 2 had 3,981 modules; Release 3 had 3,541 modules; and Release 4 had 3,978 modules. Table I summarizes the distribution of faults discovered by customers. The proportion of modules with no faults in Release 1 was $\pi_{nfp} = 0.937$, and the proportion with at least one fault was $\pi_{fp} = 0.063$.

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 fifty 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 II. Product metrics were derived from call graphs, control flow graphs, and statements. Process metrics were tabulated from the configuration management system which maintained records regarding updates by each developer, and from the problem reporting system which maintained records on past problems. Execution metrics indicated how much the software was expected to be used during operations. $USAGE$ was the fraction of installations that had the given module installed; it was forecast from deployment records (Jones et al., 1999). The execution time measurements, $RESCPU$, $BUSCPU$, and $TANCPU$, were derived from laboratory measurements of an earlier release using simulated workloads on a standard hardware configuration. Each metric represents the average execution time in the given module of a transaction in the specified workload. Future research will refine these metrics.

EMERALD helps software developers and managers to assess risks of embedded software and thereby to improve software quality (Hudepohl 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.

We built two sets of models: coding-phase models had candidate predictors derived from measurements available at the end of coding; beta testing-phase models included additional candidate predictors derived from measurements at the end of beta testing. Predictions by a coding-phase model could guide enhancement activities during the integration phase. Predictions by a beta testing-phase model could guide reengineering of the next release.

5.2. Empirical Results of Coding Phase Models

Candidate predictors of the coding phase models were derived from the 28 product metrics and execution metrics in Table II. We built TREEDISC models using all combinations of *LeafMin* $\in \{5, 10, 20, 40\}$

Table II. Software metrics

Symbol	Description
Software Product Metrics	
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 non-loop 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.

continued

Table II. Software metrics (continued)

Symbol	Description
Software Process Metrics	
DES_PR	Number of problems found by designers in the current release
BETA_PR	Number of problems found during beta testing in the current release
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 beta 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
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	Sum of the number of updates that designers had in their company careers at the time of each update
Execution Metrics	
USAGE	Deployment percentage of the module.
RESCPU	Execution time of this module on an average transaction in a system serving consumers.
BUSCPU	Execution time of this module on an average transaction in a system serving businesses.
TANCPU	Execution time of this module on an average transaction in a tandem system.

and $\alpha \in \{0.01, 0.05, 0.10\}$. This set of levels was sufficient for experimental purposes, even though additional values are possible. Table III characterizes the trees resulting from these experiments.

Each project may have its own criteria for choosing a preferred parameter value ζ, considering enhancement-activity details and constraints. In this case study, we preferred approximately equal misclassification rates and chose ζ accordingly in each experiment. The equal-misclassification-rates rule minimizes the larger misclassification rate (Seber, 1984), and in the absence of other considerations, is es-

Table III. Sizes of trees — coding phase models

α	*LeafMin*	Nodes	Leaves	Predictors
0.01	5	44	30	12
0.05	5	44	29	12
0.10	5	87	62	19
0.01	**10**	**26**	**18**	**5**
0.05	10	50	34	12
0.10	10	62	42	14
0.01	20	24	15	7
0.05	20	32	19	9
0.10	20	40	26	10
0.01	**40**	**19**	**12**	**6**
0.05	40	23	14	7
0.10	40	25	15	7

Candidate predictors: product and execution metrics
Examples are bold.

pecially appropriate when one class is much more rare than the other. Another project might have a different preference.

Table III shows three alternative measures of a tree's size: the number of nodes, the number of leaves, and the number of distinct predictors used. In each case, only a small number of predictors were significant. A small tree is desirable, if its accuracy is acceptable. The pruning approach (Breiman et al., 1984) is based on the phenomenon that smaller trees tend to have less overfitting. For a given level of *LeafMin*, Table III shows that the smallest trees were obtained by our most restrictive stopping rule, $\alpha = 0.01$. For a given level of α, Table III shows that larger values of *LeafMin* usually yielded a smaller tree.

Let us consider two trees as examples (bold in Table III). Figure 1 shows the tree generated by TREEDISC for $\alpha = 0.01$, and *LeafMin* $= 10$, and Figure 2 shows the tree for $\alpha = 0.01$, and *LeafMin* $= 40$. Each diamond node represents a decision based on one predictor, and each edge represents a merged category of that predictor. Even though the tree was constructed using predictor categories, the labels on edges in each 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. Leaves are labeled for $\zeta = 24.0$, which was preferred in both instances. This corresponds to $C_{II}/C_I = 356.9$

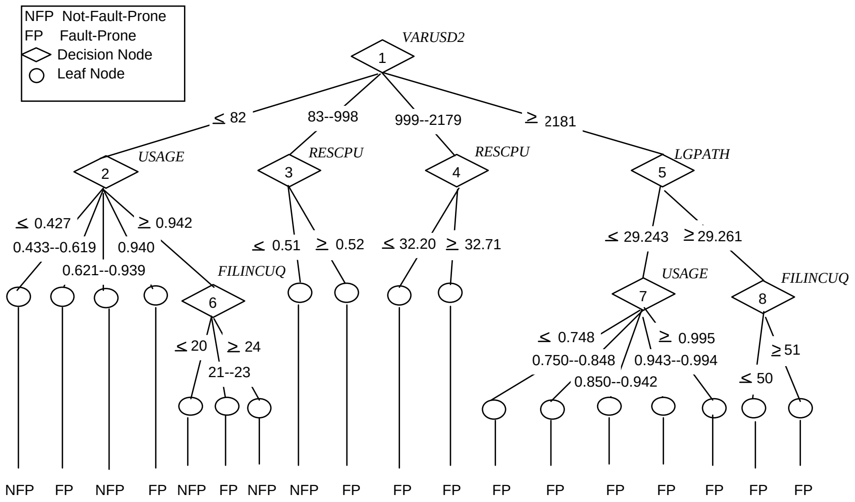

Figure 1. Coding phase tree — *LeafMin* = 10

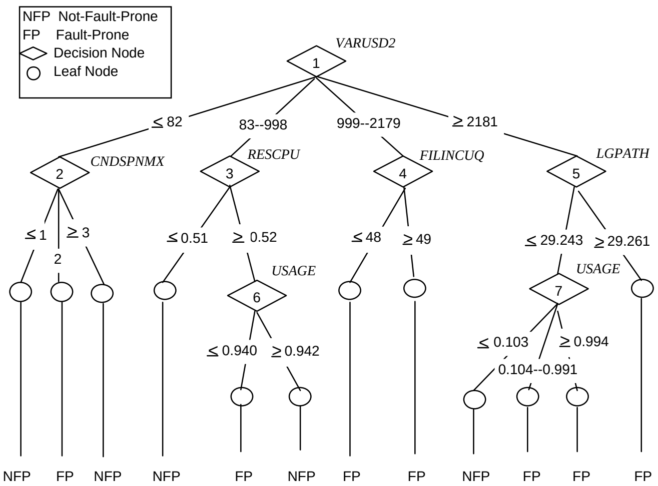

Figure 2. Coding phase tree — *LeafMin* = 40

by Equation (8). TREEDISC identifies significantly different merged categories irrespective of our preferred classification rule, ζ. Consequently, in some cases, all the leaves descended from a decision node have the same class label due to our choice of ζ.

The significant predictors in the two figures are similar. Data analysis revealed that *VARUSD2* was strongly correlated with module size. The execution metrics, *USAGE* and *RESCPU*, were related to the amount of use a module received during operations. *FILINCUQ* was associated with the extent of a module's interfaces. *LGPATH* and *CNDSPNMX* were attributes of the control flow. A module is classified by its combination of predictors along a path to a leaf.

Even though Figures 1 and 2 have some identical nodes, Figure 2 is not simply a pruned version of Figure 1. Figure 2 has fewer branches due to the effect of *LeafMin* in the merging of categories. The influence of α in these examples is evidenced by the large number of leaves where α was the stopping criterion. In Figure 1, 9 leaves stopped splitting because there was no significant split at $\alpha = 0.01$; 6 leaves stopped splitting because there were too few modules in the node ($BranchLimit = 20$); and 3 leaves stopped splitting because all modules were in the same class. In Figure 2, 8 leaves stopped splitting because there was no significant split at $\alpha = 0.01$; 2 leaves stopped splitting because there were too few modules in the node ($BranchLimit = 80$); and 2 leaves stopped splitting because all modules were in the same class.

Table IV gives the accuracy of each experiment's tree. The lowest evaluation misclassification rates were obtained at $\alpha = 0.01$. Higher values of α resulted in overfitting, as indicated by good accuracy on the training data set, but poor accuracy on the evaluation data sets. When $\alpha = 0.01$, *LeafMin* had some impact on the selection of predictors, the training accuracy and evaluation accuracy.

Table V shows the data for the experimental design, derived from Table IV. Elements in the table are the differences in the expected cost of misclassifications, ΔECM. Recall that we built twelve trees based on the training data set and each combination of factor levels. Misclassification rates for each tree were estimated on each evaluation data set for a total of three repetitions.

We estimated the effects of the factors (A_j, B_i, and AB_{ij} for all i and j) (Berenson et al., 1983). Table VI summarizes an analysis of variance of the factors, including the significance of an F-test for each factor (Berenson et al., 1983). Most of the variation in overfitting was accounted for by TREEDISC's significance, α. In the F-test for factor A, we see that *LeafMin* was related to overfitting at the 1.3% significance level. In other words, the probability that *LeafMin* is independent of ΔECM is 0.013, which we consider very small. Thus, we conclude that *LeafMin* is useful for controlling overfitting. The extremely small significance level in factor B's F-test shows that TREEDISC's α was very significantly related to overfitting. The interaction AB F-test shows that interactions between *LeafMin* and TREEDISC's α were also significant at the 1.2% level.

Model errors as a function of overfitting, y, did not exhibit a strong trend and had a generally uniform spread. A normal-quantile plot was

Table IV. Accuracy of coding phase models
Misclassification rates (%)

		training		evaluation					
		Release 1		Release 2		Release 3		Release 4	
α	ζ	I	II	I	II	I	II	I	II
LeafMin = 5									
0.01	13.3	27.07	28.32	27.63	28.49	29.41	25.53	29.00	27.47
0.05	13.3	28.42	21.24	36.26	28.49	30.64	31.91	53.49	25.27
0.10	13.3	28.91	13.90	34.93	30.11	30.25	31.91	53.03	25.27
LeafMin = 10									
0.01	**24.0**	**29.77**	**28.38**	**30.30**	**23.81**	**31.40**	**23.40**	**43.62**	**23.91**
0.05	13.3	25.91	25.33	32.54	36.51	27.79	42.55	61.25	34.78
0.10	11.5	28.83	17.90	36.39	35.98	31.08	40.43	64.49	38.04
LeafMin = 20									
0.01	24.0	28.86	27.95	29.93	25.40	30.62	23.40	30.98	26.09
0.05	24.0	27.11	27.95	28.51	28.04	29.31	25.53	30.98	26.09
0.10	13.3	15.85	33.19	17.48	41.80	16.49	42.55	18.91	48.91
LeafMin = 40									
0.01	**24.0**	**27.13**	**29.69**	**25.08**	**29.10**	**28.91**	**25.53**	**28.33**	**28.26**
0.05	24.0	20.61	31.88	18.38	36.51	20.03	31.91	19.38	43.48
0.10	19.0	20.61	31.88	18.38	36.51	20.03	31.91	19.38	43.48

I and II stand for type I and type II misclassification rate (%)
Candidate predictors: product and execution metrics
Examples are bold.

linear, indicating that errors were normally distributed. Thus, it appears that the errors were generally well behaved statistically (Berenson et al., 1983).

We conclude that adjusting TREEDISC's α is a major way to control overfitting and that adjusting *LeafMin* (and *BranchLimit* accordingly) provides further control.

5.3. Empirical Results of Beta-Testing Phase Models

We conducted additional experiments with an expanded set of candidate predictors to confirm the importance of *LeafMin*. Candidate predictors of the beta testing phase models were derived from all 42 product, process, and execution metrics in Table II. We built TREEDISC models using $\alpha = 0.01$ and various values of *LeafMin* $\in \{5, 10, 20, 40\}$.

Table V. Experiment data
$y_{ijk} = \Delta ECM$

α	Eval Release	*LeafMin* 5	10	20	40
0.01	2	0.0264	-1.0227	-0.5634	-0.1519
	3	-0.3254	-1.1046	-1.0067	-0.9188
	4	-0.0877	-0.8754	-0.3984	-0.3103
0.05	2	0.9760	1.4539	0.0334	1.0203
	3	1.3491	2.1613	-0.5236	0.0013
	4	0.7366	1.5075	-0.3820	2.5971
0.10	2	2.0743	2.0190	1.0871	0.8034
	3	2.2546	2.4488	1.1712	-0.0001
	4	1.6414	2.5043	1.9856	2.0536

Table VI. Analysis of variance

	SS	% of variation	*df*	*MS*	*MS/MSE*	*p*-value
A	4.28	8.62%	3	1.43	4.46	0.013
B	30.90	62.24%	2	15.45	48.28	3.87E-09
AB	6.79	13.67%	6	1.13	3.53	0.012
E	7.68	15.47%	24	0.32		
Total	49.65		35			

Table VII shows the sizes of resulting trees. This confirmed that a larger *LeafMin* results in a smaller tree. For example, the tree generated for *LeafMin* = 40 had significant predictors (1) that were also part of the corresponding coding phase model, (*VARUSD2*, *CNDSPNMX*, and *RESCPU*), (2) that were attributes of the developers (*LO_UPD* and *UNQ_DES*), and (3) that were counts of past (fixed) problems (*BETA_PR* and *DES_PR*).

We again chose the preferred ζ so that misclassification rates of the training data set were approximately equal. For example, $\zeta = 11.5$ implies $C_{II}/C_I = 171.0$ by Equation (8). Table VIII gives the accuracy of each tree. The impact of *LeafMin* on overfitting was similar to the results with coding phase models.

Table VII. Sizes of trees — beta-testing phase models

LeafMin	Nodes	Leaves	Predictors
5	44	29	12
10	28	18	9
20	26	17	8
40	**23**	**15**	**7**

Candidate predictors: product, process, and execution metrics
$\alpha = 0.01$
Example is bold.

Table VIII. Accuracy of beta-testing phase models
Misclassification rates (%)

		training				evaluation			
		Release 1		Release 2		Release 3		Release 4	
LeafMin	ζ	I	II	I	II	I	II	I	II
5	11.5	23.98	21.83	24.47	28.57	26.27	21.28	23.98	28.26
10	10.1	23.54	28.38	23.55	32.80	24.33	19.15	21.13	30.43
20	11.5	25.03	24.89	24.76	29.10	26.36	19.15	26.15	30.43
40	**11.5**	**27.95**	**24.02**	**26.93**	**25.93**	**28.68**	**14.98**	**27.77**	**26.09**

I and II stand for type I and type II misclassification rate (%)
Candidate predictors: product, process, and execution metrics
$\alpha = 0.01$
Example is bold.

6. Conclusions

Predicting which modules are likely to have faults during operations is important to software developers. Software quality models which yield timely predictions on a module-by-module basis are tools for focusing software enhancement efforts. Modeling software quality with classification trees is attractive because they readily model nonmonotonic relationships.

This paper presents classification-tree modeling of software quality using the TREEDISC algorithm (SAS Institute staff, 1995) and evaluates its parameters for controlling overfitting. The results offer lessons learned for other tree-building algorithms with analogous parameters, especially for implementations of the CHAID algorithm (Hawkins and Kass, 1982; Kass, 1980). An overfitted model reflects the training data set too closely. An interpretation of an overfitted model's structure may be misleading. Even though a model appears to be accurate on training data, if overfitted, it may be much less accurate when applied in practice. As a measure of overfitting, we use a difference between expected costs of misclassification, ΔECM.

We found that when the proportion of fault-prone modules in the system is small, TREEDISC's default classification rule does not yield useful accuracy. Therefore, we choose a parameter value, ζ, to yield a preferred balance between misclassification rates.

We conducted a case study of a very large telecommunications system. A two-way analysis of variance with repetitions found that TREEDISC's significance level, α, is closely related to overfitting and can be used to control it. The minimum number of modules in a leaf, *LeafMin*, also has a statistical relationship with overfitting. (Our experiments fixed $BranchLimit = 2 \times LeafMin$, the minimum number of modules in a node to qualify for further splitting.) Because overfitting depends on the amount of data and the data set's characteristics, we recommend that practitioners conduct experiments, as we did, over ranges of values for α and *LeafMin* to determine controls for overfitting empirically.

Future work will apply similar principles to other classification techniques, and will compare various pruning algorithms and parameters for controlling overfitting.

Acknowledgements

We thank Wendell Jones and the EMERALD team for collecting the case-study data. We thank John Hudepohl for his encouragement and support. We thank Xiaojing Yuan for conducting the experiments. This work was supported in part by a grant from Nortel Networks 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.

References

Akaike, H.: 1987, 'Factor Analysis and AIC'. *Psychometrika* **52**(3), 317–332.

Basili, V. R., L. C. Briand, and W. Melo: 1996, 'A Validation of Object-Oriented Design Metrics as Quality Indicators'. *IEEE Transactions on Software Engineering* **22**(10), 751–761.

Berenson, M. L., D. M. Levine, and M. Goldstein: 1983, *Intermediate Statistical Methods and Applications: A Computer Package Approach*. Englewood Cliffs, New Jersey USA: Prentice-Hall.

Breiman, L., J. H. Friedman, R. A. Olshen, and C. J. Stone: 1984, *Classification and Regression Trees*. London: Chapman & Hall.

Briand, L. C., V. R. Basili, and W. M. Thomas: 1992, 'A Pattern Recognition Approach for Software Engineering Data Analysis'. *IEEE Transactions on Software Engineering* **18**(11), 931–942.

Clark, L. A. and D. Pregibon: 1992, 'Tree-Based Models'. In: J. M. Chambers and T. J. Hastie (eds.): *Statistical Models in S*. Pacific Grove, California: Wadsworth, pp. 377–419.

Ebert, C.: 1996, 'Classification Techniques for Metric-Based Software Development'. *Software Quality Journal* **5**(4), 255–272.

El Emam, K., S. Benlarbi, and N. Goel: 1999, 'Comparing Case-Based Reasoning Classifiers for Predicting High Risk Software Components'. Technical Report NRC/ERB-1058, National Research Council Canada, Ottawa, Canada. NRC 43602.

Evanco, W. M. and W. W. Agresti: 1994, 'A Composite Complexity Approach for Software Defect Modeling'. *Software Quality Journal* **3**(1), 27–44.

Gokhale, S. S. and M. R. Lyu: 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, 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, Chapt. 5, pp. 269–302.

Henry, J., S. Henry, D. Kafura, and L. Matheson: 1994, 'Improving Software Maintenance at Martin Marietta'. *IEEE Software* **11**(4), 67–75.

Hudepohl, J. P., S. J. Aud, T. M. Khoshgoftaar, E. B. Allen, and J. Mayrand: 1996, 'EMERALD: Software Metrics and Models on the Desktop'. *IEEE Software* **13**(5), 56–60.

Jones, W. D., J. P. Hudepohl, T. M. Khoshgoftaar, and E. B. Allen: 1999, 'Application of a Usage Profile in Software Quality Models'. In: *Proceedings of the Third European Conference on Software Maintenance and Reengineering*. Amsterdam, Netherlands, 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 E. B. Allen: 1998, 'Classification of Fault-Prone Software Modules: Prior Probabilities, Costs, and Model Evaluation'. *Empirical Software Engineering: An International Journal* **3**(3), 275–298.

Khoshgoftaar, T. M. and E. B. Allen: 2000, 'A Practical Classification Rule for Software Quality Models'. *IEEE Transactions on Reliability* **49**(2). Forthcoming.

Khoshgoftaar, T. M., E. B. Allen, L. A. Bullard, R. Halstead, and G. P. Trio: 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.

Khoshgoftaar, T. M., E. B. Allen, W. D. Jones, and J. P. Hudepohl: 1999a, 'Classification Tree Models of Software Quality over Multiple Releases'. In: *Proceedings:*

The Tenth International Symposium on Software Reliability Engineering. Boca Raton, Florida USA, pp. 116–125.

Khoshgoftaar, T. M., E. B. Allen, W. D. Jones, and J. P. Hudepohl: 1999b, '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., E. B. Allen, K. S. Kalaichelvan, and N. Goel: 1996b, 'Early Quality Prediction: A Case Study in Telecommunications'. *IEEE Software* **13**(1), 65–71.

Khoshgoftaar, T. M., E. B. Allen, A. Naik, W. D. Jones, and J. P. Hudepohl: 1998, 'Using Classification Trees for Software Quality Models: Lessons Learned'. In: *Proceedings of the Third IEEE International High-Assurance Systems Engineering Symposium*. Bethesda, MD USA, pp. 82–89.

Khoshgoftaar, T. M., E. B. Allen, and X. Yuan: 1999c, 'Balancing Misclassification Rates in Classification-Tree Models of Software Quality'. Technical report, Florida Atlantic University, Boca Raton, FL USA.

Khoshgoftaar, T. M., E. B. Allen, X. Yuan, W. D. Jones, and J. P. Hudepohl: 1999d, 'Assessing Uncertain Predictions of Software Quality'. In: *Proceedings of the Sixth International Software Metrics Symposium*. Boca Raton, Florida USA, pp. 159–168.

Khoshgoftaar, T. M., E. B. Allen, X. Yuan, W. D. Jones, and J. P. Hudepohl: 1999e, 'Preparing Measurements of Legacy Software for Predicting Operational Faults'. In: *Proceedings: International Conference on Software Maintenance*. Oxford, England, pp. 359–368.

Khoshgoftaar, T. M. and D. L. Lanning: 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.

Kitchenham, B. A.: 1998, 'A Procedure for Analyzing Unbalanced Datasets'. *IEEE Transactions on Software Engineering* **24**(4), 278–301.

Mao, W.: 2000, 'Classification of Software Quality Using Tree Modeling with the SPRINT/SLIQ Algorithm'. Master's thesis, Florida Atlantic University, Boca Raton, Florida USA. Advised by Taghi M. Khoshgoftaar.

Mayrand, J. and F. Coallier: 1996, 'System Acquisition Based on Software Product Assessment'. In: *Proceedings of the Eighteenth International Conference on Software Engineering*. Berlin, pp. 210–219.

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.: 1992, 'Methodology for Validating Software Metrics'. *IEEE Transactions on Software Engineering* **18**(5), 410–422.

Schneidewind, N. F.: 1995, 'Software Metrics Validation: Space Shuttle Flight Software Example'. *Annals of Software Engineering* **1**, 287–309.

Schneidewind, N. F.: 1998, 'An Integrated Process and Product Model'. In: *Proceedings Fifth International Software Metrics Symposium*. Bethesda, MD USA, pp. 224–234.

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

Selby, R. W. and A. A. Porter: 1988, 'Learning from Examples: Generation and Evaluation of Decision Trees for Software Resource Analysis'. *IEEE Transactions on Software Engineering* **14**(12), 1743–1756.

Steinberg, D. and P. Colla: 1995, 'CART: A supplementary modules for SYSTAT'. Salford Systems, San Diego, CA.

Stone, M. and J. Rasp: 1993, 'The Assessment of Predictive Accuracy and Model Overfitting: An Alternative Approach'. *Journal of Business Finance and Accounting* **20**(1), 125–131.

Takahashi, R., Y. Muraoka, and Y. Nakamura: 1997, 'Building Software Quality Classification Trees: Approach, Experimentation, Evaluation'. In: *Proceedings of the Eighth International Symposium on Software Reliability Engineering.* Albuquerque, NM USA, 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.

Authors' Vitae

Taghi M. Khoshgoftaar
is a professor of the Department of Computer Science and Engineering, Florida Atlantic University and the Director of the Empirical Software Engineering Laboratory. His research interests are in software engineering, software complexity metrics and measurements, software reliability and quality engineering, computational intelligence, computer performance evaluation, multimedia systems, and statistical modeling. He has published more than 150 refereed papers in these areas. He has been a principal investigator and project leader in a number of projects with industry, government, and other research-sponsoring agencies. He is a member of the Association for Computing Machinery, the American Statistical Association, the IEEE Computer Society, and IEEE Reliability Society. He served as the general chair of the 1999 International Symposium on Software Reliability Engineering (ISSRE'99), and the general chair of the 2001 International Conference on Engineering of Computer Based Systems. He has served on technical program committees of various international conferences, symposia, and workshops. He has served as North American editor of the *Software Quality Journal*, and is on the editorial board of the *Journal of Multimedia Tools and Applications.*

Edward B. Allen
received the B.S. degree in engineering from Brown University, Providence, Rhode Island USA, in 1971, the M.S. degree in systems engineering from the University of Pennsylvania, Philadelphia, Pennsylvania USA, in 1973, and the Ph.D. degree in computer science from Florida Atlantic University, Boca Raton, Florida USA, in 1995. He is currently an assistant professor in the Department of Computer Science at Mississippi State University. He began his career as a programmer with the U.S. Army. From 1974 to 1983, he performed systems engineering and

software engineering on military systems, first for Planning Research Corp. and then for Sperry Corp. From 1983 to 1992, he developed corporate data processing systems for Glenbeigh, Inc., a specialty health care company. His research interests include software measurement, software process, software quality, and computer performance modeling. He has more than 60 refereed publications in these areas. He is a member of the IEEE Computer Society and the Association for Computing Machinery.

Address for Offprints: Readers may contact the authors through Taghi M. Khoshgoftaar, Empirical Software Engineering Laboratory, Dept. of Computer Science and Engineering, Florida Atlantic University, Boca Raton, FL 33431 USA. Phone: (561)297-3994, Fax: (561)297-2800, Email: taghi@cse.fau.edu, URL: www.cse.fau.edu/esel/.