Balancing Misclassification Rates in Classification-Tree Models of Software Quality

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

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

Abstract. Software product and process metrics can be useful predictors of which modules are likely to have faults during operations. Developers and managers can use such predictions by software quality models to focus enhancement efforts before release. However, in practice, software quality modeling methods in the literature may not produce a useful balance between the two kinds of misclassification rates, especially when there are few faulty modules.

This paper presents a practical classification rule in the context of classification tree models that allows appropriate emphasis on each type of misclassification according to the needs of the project. This is especially important when the faulty modules are rare.

An industrial case study using classification trees, illustrates the tradeoffs. The trees were built using the TREEDISC algorithm which is a refinement of the CHAID algorithm. We examined two releases of a very large telecommunications system, and built models suited to two points in the development life cycle: the end of coding and the end of beta testing. Both trees had only five significant predictors, out of 28 and 42 candidates, respectively. We interpreted the structure of the classification trees, and we found the models had useful accuracy.

Keywords: classification trees, CHAID, TREEDISC, telecommunications, software quality, fault-prone modules, software metrics, knowledge discovery in data bases

1. Introduction

Research in the field of software metrics has shown that software product and process metrics can be useful predictors of which modules are most likely to have customer-discovered faults (Ebert, 1997; Henry and Wake, 1991; Khoshgoftaar et al., 1999a; Schneidewind, 1995a). Nortel's Enhanced Measurement for Early Risk Assessment of Latent Defects, EMERALD (Hudepohl et al., 1996), is an example of a decision-support system that routinely assesses the risk of software defects through software quality models. Predictions can be used by software devel-

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

opers and managers to focus improvement efforts before release or to prioritize reengineering efforts for the next release.

A variety of classification techniques have been used to model software quality, such as discriminant analysis (Khoshgoftaar et al., 1996b; Munson and Khoshgoftaar, 1992), logistic regression (Basili et al., 1996), discriminant power (Schneidewind, 1995b; Schneidewind, 1999), optimal set reduction (Briand et al., 1993), neural networks (Khoshgoftaar and Lanning, 1995), fuzzy classification (Ebert, 1996), and classification trees (Selby and Porter, 1988). The accuracy of a classification model is measured by its misclassification rates. A *Type I* misclassification is when the model identifies a module as fault-prone which is actually not fault-prone. A *Type II* misclassification is when the model identifies a module as not fault-prone which is actually fault-prone. We (Khoshgoftaar and Allen, 2000) and others (Lanubile, 1996) have observed that published modeling methods may not consistently produce a useful balance between the Type I and Type II rates. If either type of misclassification rate is large, the model is generally not useful to guide software improvement efforts. If the Type I misclassification rate is high, then targeting of enhancement efforts will be inefficient. If the Type II misclassification rate is high, then most fault-prone modules will not be enhanced.

This paper presents a practical approach to achieving a preferred balance between misclassification rates in the context of a classification-tree model, and presents an industrial case study. This paper extends to classification trees our earlier work based on discriminant analysis and logistic regression (Khoshgoftaar and Allen, 2000; Khoshgoftaar et al., 1999a). Classification-tree models are attractive, because they are easily understood by software engineers. Moreover, because their decision rules are based on thresholds, they readily model nonlinear and nonmonotonic relationships among variables.

Selby and Porter pioneered the use of classification trees as software quality models, using the ID3 algorithm (Quinlan, 1986; Selby and Porter, 1988; Takahashi et al., 1997). Recent software quality studies have also applied a regression-tree algorithm (Gokhale and Lyu, 1997; Troster and Tian, 1995), and the Classification And Regression Trees (CART) algorithm (Khoshgoftaar et al., 1998b; Kitchenham, 1998). Preliminary studies of the TREEDISC algorithm indicated its potential value for software quality modeling (Khoshgoftaar et al., 1996a; Khoshgoftaar et al., 1999b). TREEDISC[1] (SAS Institute staff, 1995)is a refinement of the CHAID algorithm (Kass, 1980), and is attrac-

[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.

tive because it finds statistically significant branching criteria, rather than using heuristic criteria to build a tree. TREEDISC has been little used for software quality modeling.

The case study examined two releases of a very large telecommunications system. Each release consisted of over ten million lines of code, and had more than thirteen thousand configuration management transactions which were due to normal development and resolution of more than six thousand problem reports (reported both before and after release). Updated modules had more than five million lines of code in more than three thousand modules. This mission-critical system was required to have high reliability.

We built two classification-tree models using different sets of candidate predictors to simulate using software quality models at two points in the development life cycle. One model used only predictors available at the end of coding, and the other added predictors available at the end of beta testing.

The case study adapted Fayyad's framework for knowledge discovery in data bases (Fayyad, 1996), which is appropriate when one seeks valuable information in large amounts of data, collected for some other purpose (Hand, 1998). This aptly describes our approach to software quality modeling. Software faults result from mistakes by developers. Because relevant human behavior is very difficult to measure directly in the workplace, we take a more pragmatic approach, leveraging existing data bases, collected for other purposes, to capture relevant variation of module attributes. Many software development organizations maintain large data bases supporting configuration management, problem reporting, and installation support. The case study consisted of the following steps: measurement, selection and cleaning of target data, transformation of data, modeling, and evaluation of knowledge.

2. Measurement

We studied two consecutive releases of a very large legacy telecommunications system, written in a high-level language (Protel) similar to Pascal, using the procedural development paradigm, and maintained by professional programmers in a large organization. This embedded-computer application included numerous finite-state machines. It was required to have high reliability, because it provided essential infrastructure to many mission-critical systems. Data on one release was the *training* data set and data on the subsequent release was the *evaluation* data set. This simulated the use of a software quality model, in practice. The entire system had significantly more than ten million lines of code.

A *module* consisted of a set of functionally related source-code files according to the system's architecture. An average module had about four source files.

Fault data was collected at the module-level by a problem reporting system. A module was considered *fault-prone* if any faults 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 customer sites are usually required to repair them. Preventing customer-discovered faults was a high priority for the developers of this system, and thus, they were very interested in timely software quality predictions.

Software product, process, and execution metrics were collected from source code, the configuration management system, the problem reporting system, deployment records, and laboratory measurements of execution times under various workload profiles. Pragmatic considerations usually determine the set of available software metrics. We do not advocate collecting a particular set of metrics to the exclusion of others recommended in the literature. We prefer to analyze a broad set of metrics, rather than limiting data collection according to predetermined research questions (Hand, 1998).

Collection of software product metrics involved extracting source code from the configuration management system and then measuring the source code with EMERALD's software-metrics analysis tool. Because marginal data-collection costs were modest, EMERALD provided over fifty source-code metrics (Mayrand and Coallier, 1996). Preliminary data analysis eliminated metrics that were inappropriate for modeling purposes, such as those with the same value for all modules (Kitchenham et al., 1995). Table I lists the software product metrics used in this study. Counts of procedure calls are included. Some of the metrics are measures of a module's control flow graph, which consists of nodes and arcs depicting the flow of control. Other metrics quantify attributes of statements. For example, the span of a variable is the number of lines between its first and last use within a procedure.

Process metrics listed in Table I were tabulated from configuration management and problem reporting data. The configuration management system maintained records regarding updates to source code by each developer (designer). The problem reporting system maintained records on past problems.

Execution metrics listed in Table I were forecast from deployment records (*USAGE*) (Jones et al., 1999) and laboratory measurements (*RESCPU*, *BUSCPU*, and *TANCPU*) of a prior release.

Table I. 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 I. 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.

3. Selection and Cleaning of Target Data

Analysis of configuration management data identified modules that were unchanged from the prior release. More than 99% of the unchanged modules had no faults. There were too few fault-prone modules in the unchanged set for effective modeling. This case study considered only updated modules, that is, those that were new or had at least one update to source code since the prior release. These modules had more

Table II. Summary statistics

	Release		
	training	both	evaluation
Updated modules*	3649	1761	3981
Fault-prone modules	229	47	189
π_{fp}	0.063		
Faults	306		229

* Updated means new or changed

than five million lines of code in more than three thousand modules in each release. Table II presents some details about the data sets.

The proportion of modules with no faults among the updated modules was $\pi_{nfp} = 0.937$, and the proportion with at least one fault was $\pi_{fp} = 0.063$. Such a small set of modules is often difficult to predict effectively. This case study did not need to perform extensive cleaning of data. We omitted from the cleaned data sets those modules missing relevant data due to practical data collection issues. For example in the evaluation data set, usage data was not available for 1,101 updated modules and metrics data was missing on 57 updated modules due to parsing problems, resulting in 3,981 modules in the cleaned evaluation data set.

4. Transformation

We transformed selected individual software metrics for improved modeling, and then transformed all metrics into discrete ordinal predictors that were suitable for TREEDISC.

In particular, we sought to remove correlations due to definitions of related metrics. Correlation among variables risks a less robust model. The measurement data included the total number of nodes (*NDS*) in the control flow graph, as well as the number of entry nodes (*NDSENT*), exit nodes (*NDSEXT*), and inaccessible nodes (dead code) (*NDSPND*). The latter three types of nodes are somewhat correlated with the total, because they are components. Therefore, we transformed the metrics, and as shown in Table I, used internal nodes, $NDSINT = NDS - NDSENT - NDSEXT - NDSPND$, in the modeling, rather than the total (*NDS*). The total number of procedure calls (*CAL*) is inherently somewhat correlated with the number of distinct procedure calls (*CALUNQ*). Similarly, the total number of variable uses

(*VARUSD*) is correlated with the number of distinct variables used (*VARUSDUQ*). As shown in Table I, in both cases, we defined "second and following ..." metrics, *CAL2* and *VARUSD2* to avoid inherently correlated metrics.

The measurement data included the total number of independent paths (*PTHIND*) in the control flow graph. The range was greater than 10^{32} which was too large for statistical analysis. We made a monotonic transformation to a more practical range; as shown in Table I, we used the base 2 logarithm of independent paths, $LGPATH = \log_2 PTHIND$.

The measurement data also included several averages, such as the average number of updates in designers' company careers (*UPDAV*), and the average span of conditional structures (*CNDSPNAV*). Because we measured customer-discovered faults over the entire module and not as a density, we transformed averages to the totals, for example, $UPD\_CAR = (UPDAV)(TOT\_UPD)$ and $CNDSPNSM = (CNDSPNAV)(CND)$, where *CND* is the number of conditional structures. Among these transformed individual metrics, we discovered that only *VARUSD2* and *LGPATH* were significant with respect to customer-discovered faults.

Most of the software metrics listed in Table I are positive integers with no upper bound, in principle. A few are real numbers, such as *LGPATH* and *USAGE*. The execution-time metrics (*RESCPU*, *BUSCPU*, and *TANCPU*) are based on laboratory measurements, and thus, have real values within measurement error. However, numeric data are not suitable for TREEDISC (Kass, 1980), and therefore, it is necessary to transform numeric data, such as ours, into discrete ordinal predictors. After some empirical investigation, we preferred a maximum of sixty predictor categories, $c_{\max} = 60$, for this case study.

We transformed each metric into a predictor by ranking and then grouping (Khoshgoftaar et al., 1999b). 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. Predictor values are given by the following.

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

where x_{ij} is the i^{th} module's value of the j^{th} predictor. After transformation, the predictor values are integers, $x_{ij} \in \{0, \ldots, c_{\max} - 1\}$. 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, using the same value of

$c_{\max}$, to each metric in each data set. Future research will investigate customizing the transformation for each software metric.

5. Modeling

5.1. Using a Classification Tree

In our application, a classification tree is an algorithm to classify a module, represented by an abstract tree of decision rules. Each internal node represents a decision, and each edge represents a possible result of that decision. Each leaf is labeled with a class of the response variable: not fault-prone or fault-prone in our application. The *root* of the tree is the node at the top.

Given a module's predictor values, beginning at the root, the algorithm traverses a downward path in the tree, one node after another, until reaching a leaf node. Each decision node is associated with one predictor. When the algorithm reaches a decision node, the value of its predictor is compared to the range 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. In our application, when a leaf node is reached, the module is classified as not fault-prone or fault-prone, and the path is complete. Each module in a data set can be classified using this algorithm.

5.2. Building a Classification Tree with TREEDISC

The TREEDISC algorithm (SAS Institute staff, 1995) builds classification trees. It is a refinement of the CHAID algorithm (Kass, 1980), and 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 algorithm recursively partitions the training data set modules into leaves of the classification tree until a stopping criterion applies to every leaf node. Thus, each module in the training data set is considered assigned to a leaf as the algorithm progresses. A leaf is not partitioned if there is no significant branching criterion. Empirical investigation found that a minimum leaf size $\mathit{LeafMin} = 40$ yielded the most robust family of models in this case study (i.e., little overfitting) (Yuan, 1999). The default branching limit was twice the minimum leaf size, $\mathit{BranchLimit} = 2\,\mathit{LeafMin}$. Thus, a leaf was not partitioned when the number of modules in the leaf was less than or equal to $\mathit{BranchLimit} = 80$.

The algorithm chooses a predictor from a list of candidates, and forms the most significant merging of its categories, constrained by the value of *LeafMin*. Let $r = 2$ be the number of response variable categories: not fault-prone and fault-prone. For a given predictor, let c be the number of original predictor categories, $c \leq c_{\max}$, and let k be the number of merged predictor categories, $1 \leq k \leq c$. Significance is evaluated using the chi-squared test of the predictor's $k \times r$ merged contingency table. The chi-squared statistic, X, 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 (Zar, 1984). The adjusted p-value is given by the following.

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

A significance test rejects the null hypothesis that the response variable is independent of the predictor, if $p < \alpha$. Our case study used $\alpha = 0.01$ which was empirically chosen to control overfitting (Yuan, 1999). The TREEDISC algorithm finds the most significant partitioning of each candidate predictor's original categories with respect to the response variable, and chooses the most significant overall where $p < \alpha$. TREEDISC allows multiway splits and minimizes the adjusted p-value of chi-squared tests to determine the ranges. Multiway splits allow more parsimonious modeling than binary splits when there are non-monotonic relationships between the predictors and the response variable.

The complete classification tree is built by invoking TREEDISC with all the training data set modules assigned to a single leaf node. This single node becomes the root of the classification tree. The recursive algorithm then builds the tree.

We built two classification trees using TREEDISC with alternative sets of candidate predictors. Candidate predictors for the first tree were derived using Equation (1) from product and execution metrics in Table I, which can be measured at the end of coding. Candidate predictors for the second tree were also derived using Equation (1) from product, process, and execution metrics in Table I, which can be measured at the end of beta testing.

5.3. 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 data set 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.

$$Class(\mathbf{x}_i) = \begin{cases} \textit{not fault-prone} & \text{if } 1 - \hat{q}(L(\mathbf{x}_i)) \geq \hat{q}(L(\mathbf{x}_i)) \\ \textit{fault-prone} & \text{otherwise} \end{cases} \tag{4}$$

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

Accuracy is measured by misclassification rates. Let the Type I misclassification rate, $\Pr(fp|nfp)$, be estimated by the proportion of not fault-prone modules that are misclassified. Let the Type II misclassification rate, $\Pr(nfp|fp)$, be estimated by the proportion of fault-prone modules that are misclassified. With various classification techniques, we have observed a tradeoff between Type I and Type II misclassification rates (Khoshgoftaar and Allen, 1998; Khoshgoftaar and Allen, 2000; Khoshgoftaar et al., 1999a; Khoshgoftaar et al., 1998b). As $\Pr(nfp|fp)$ goes down, $\Pr(fp|nfp)$ goes up, and conversely. We propose the following classification rule which allows for appropriate emphasis on each type of misclassification according to the needs of the project. Suppose a software quality modeling technique produces a likelihood function for each class, f_{nfp} and f_{fp}, such as probability functions. The following rule enables a project to select its preferred balance between the misclassification rates by choosing a parameter ζ (Khoshgoftaar and Allen, 2000).

$$Class(\mathbf{x}_i) = \begin{cases} \textit{not fault-prone} & \text{if } \frac{f_{nfp}(\mathbf{x}_i)}{f_{fp}(\mathbf{x}_i)} \geq \zeta \\ \textit{fault-prone} & \text{otherwise} \end{cases} \tag{5}$$

When applied to a classification tree, the likelihood functions, f_{nfp} and f_{fp}, are probability mass functions of class membership over the leaves. Thus, our general classification rule for a classification tree is the following.

$$Class(\mathbf{x}_i) = \begin{cases} \textit{not fault-prone} & \text{if } \frac{1-\hat{q}(L(\mathbf{x}_i))}{\hat{q}(L(\mathbf{x}_i))} \geq \zeta \\ \textit{fault-prone} & \text{otherwise} \end{cases} \tag{6}$$

An alternative formulation is the following.

$$Class(\mathbf{x}_i) = \begin{cases} \textit{not fault-prone} & \text{if } 1 - \hat{q}(L(\mathbf{x}_i)) \geq \theta \\ \textit{fault-prone} & \text{otherwise} \end{cases} \tag{7}$$

where the threshold $\theta = \zeta/(1+\zeta)$. The TREEDISC default rule in Equation (4) is equivalent to $\zeta = 1$ and $\theta = 0.5$.

We choose a preferred value of the classification parameter ζ (or equivalently θ) 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 best ζ for the project. This procedure is straightforward in practice, because the misclassification rates are monotonic functions of ζ and θ. For example, if one chooses ζ such that $\Pr(fp|nfp) = \Pr(nfp|fp)$, then the greater of the two misclassification rates is minimized (Seber, 1984). In practice, we can achieve only approximate equality due to finite discrete data sets and discrete predictors.

We classified modules in the training data set and the evaluation data set using $\theta = 0.5$ and θ in the range 0.90 to 0.99 for both trees in the case study. As mentioned above, $\theta = 0.5$ yielded unsatisfactory accuracy. Table III shows the accuracy of the trees over the range of classification parameter values when the candidate predictors were derived from product and execution metrics. Figure 1 shows the misclassification rates in the training data set columns as a function of the classification rule parameter (θ) for this model. One could use such tables and graphs as analysis tools when choosing the preferred value of the classification rule parameter. Our preference of $\theta = 0.96$ for the product and execution metrics-based model is bold in Table III.

Table III also shows the accuracy of the tree when the candidate predictors were derived from all of the product, process, and execution metrics. Our preferred classification rule of $\theta = 0.92$ is bold. Figure 2 shows the misclassification rates in the training data set columns as a function of the classification rule parameter (θ).

We simplified each tree when adjacent leaves from the same decision node had the same preferred classification, yielding an equivalent tree. The tree produced by TREEDISC based on product and execution metrics was simplified for $\theta = 0.96$, resulting in the tree in Figure 3. Each diamond represents a decision node, and each circular node is a

Table III. Experiments balancing misclassification rates

Misclassification rates (%)

	Product and Execution Metrics				All Metrics			
	training		evaluation		training		evaluation	
θ	Type I	Type II	Type I	Type II	Type I	Type II	Type I	Type II
0.50	0.00	100.00	0.00	100.00	0.61	89.52	0.42	92.06
0.90	16.23	37.55	16.40	37.57	17.57	37.99	17.72	37.04
0.91	16.23	37.55	16.40	37.57	17.57	37.99	17.72	37.04
0.92	16.23	37.55	16.40	37.57	**27.95**	**24.02**	**26.93**	**25.93**
0.93	16.23	37.55	16.40	37.57	27.95	24.02	26.93	25.93
0.94	16.23	37.55	16.40	37.57	29.27	22.71	29.27	24.87
0.95	18.71	35.37	16.40	37.57	29.27	22.71	29.27	24.87
0.96	**27.13**	**29.69**	**25.08**	**29.10**	37.98	16.59	35.55	18.52
0.97	77.63	1.75	75.42	6.35	37.98	16.59	35.55	18.52
0.98	79.80	0.87	77.43	2.65	82.57	0.00	81.36	0.00
0.99	79.80	0.87	77.43	2.65	82.57	0.00	81.36	0.00

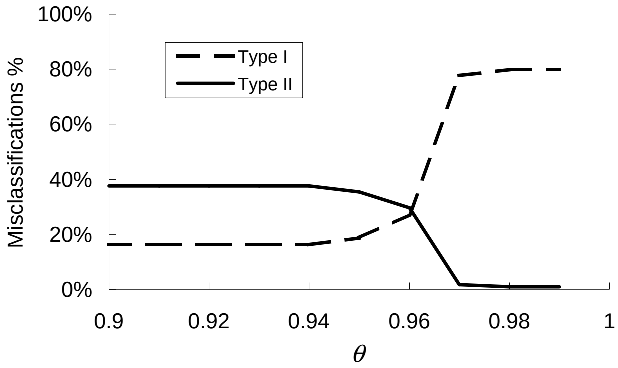

Figure 1. Experiments for product and execution metrics

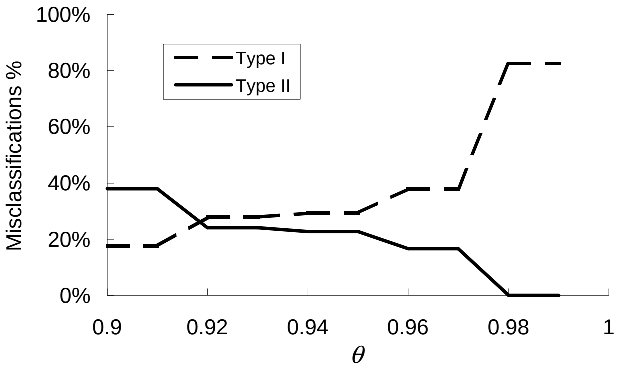

Figure 2. Experiments for product, process, and execution metrics

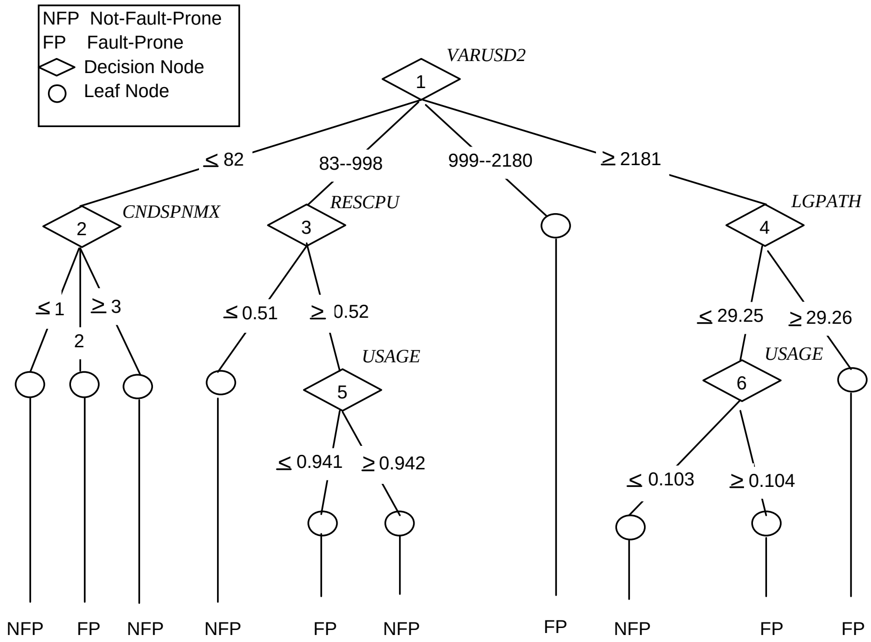

Figure 3. Classification tree based on product and execution metrics

leaf. Even though the tree was constructed using data groupings, the labels on edges in the figure have been transformed back to equivalent raw metric values for easier interpretation. Similarly, the tree produced by TREEDISC based on product, process, and execution metrics was simplified for $\theta = 0.92$ resulting in the tree in Figure 4.

6. Evaluation of Knowledge

Knowledge derived from modeling consisted of the structure of the classification trees and an evaluation of their accuracy when used for prediction. The tree in Figure 3 had only five significant predictors out of 28 candidates. Let us explore its structure by walking through the classification algorithm. Suppose we had data on a module. Node 1 compared the module's number of second and following uses of variables, *VARUSD2*, with four ranges. Like many other software metrics, *VARUSD2* was correlated with the size of a module. If a module was small ($VARUSD2 \leq 82$), then we proceeded to Node 2 which was another decision. If the maximum span of conditional arcs was exactly two arcs ($CNDSPNMX = 2$), then the module was classified as fault-prone. Other values of *CNDSPNMX* resulted in a not fault-prone classification. The value $CNDSPNMX = 2$ in small mod-

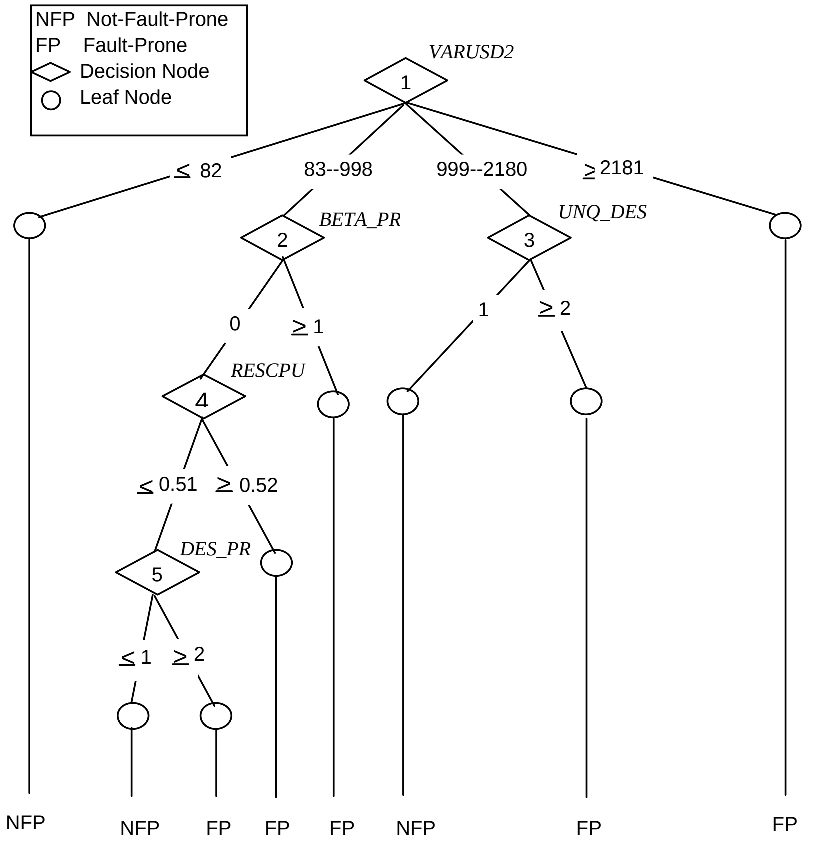

Figure 4. Classification tree based on product, process, and execution metrics

ules was probably a surrogate for a significant attribute that was not measured. This warrants further investigation by the project team. If a module was medium-sized ($83 \leq$ *VARUSD2* ≤ 998), then we proceeded from Node 1 to Node 3. Its predicted class depended on its exposure to customer activity represented by the combination of its execution metrics (*RESCPU* and *USAGE*). If a module was large ($999 \leq$ *VARUSD2* ≤ 2180), it was classified as fault-prone. If a module was very large (*VARUSD2* ≥ 2181), it was classified as fault-prone, unless it also had the combination of not-large logic (*LGPATH* ≤ 29.25) and very small deployment (*USAGE* ≤ 0.103) which was classified as not fault-prone. In other words, if a very large module was installed in very few locations, and the logic was not extremely large, then discovery of faults by customers was unlikely.

We assess the accuracy of a classification tree by its predictions for the evaluation data set. (Results on the training data set may be overly optimistic.) Consider the following naive model as a baseline for evaluating accuracy.

> Classification rule: If a module was fault-prone in the training data set, then it is predicted to be fault-prone, otherwise (i.e., not fault-prone in the training data set or new) it is predicted to be not fault-prone.

For the evaluation data set, the Type I misclassification rate of this model was $\Pr(fp|nfp) = 2.5\%$, but the Type II misclassification rate was $\Pr(nfp|fp) = 75.1\%$. In other words, the naive model was largely unsuccessful in identifying fault-prone modules.

As shown in Table III, the tree based on product and execution metrics for our preferred value of $\theta = 0.96$ had a Type I misclassification rate of $\Pr(fp|nfp) = 25.08\%$ and a Type II rate of $\Pr(nfp|fp) = 29.10\%$. We would expect this level of accuracy for predictions on a current release at the end of coding when all the significant predictors can be measured. Predictions at this level of accuracy could be very useful in guiding software enhancement efforts. Suppose one gives extra reviews or extra testing to all modules predicted to be fault-prone, i.e., $27.97\% = (0.063)(1 - 0.2910) + (0.937)(0.2508)$ of the modules where $\pi_{nfp} = 0.937$ and $\pi_{fp} = 0.063$. One would expect this set of modules to contain 70.90% of the modules that are actually fault-prone, and thus, the model's predictions would productively target the extra reviews or extra testing.

The tree in Figure 4 had only five significant predictors out of 42 candidates based on product, process and execution metrics. The root node here was equivalent to the root node in Figure 3; Node 1 compared the module's size with four ranges, as measured by the number of second and following uses of variables, *VARUSD2*. Small modules ($VARUSD2 \leq 82$) were classified as not fault-prone, and very large modules ($VARUSD2 \geq 2181$) were classified as fault-prone. Medium-sized modules ($83 \leq VARUSD2 \leq 998$) were generally classified as fault-prone, except for a particular combination of other predictors which was classified as not fault-prone: modules with no faults discovered during beta test ($BETA\_PR = 0$), low execution time in consumer-oriented systems ($RESCPU \leq 0.51$ microseconds), and zero or one fault discovered by designers during development ($DES\_PR \leq 1$). We have observed in several studies that past faults which have been fixed (*BETA_PR* and *DES_PR*) indicate higher likelihood of future faults in the same modules (Khoshgoftaar et al., 1998a; Khoshgoftaar et al., 1999a). Large modules ($999 \leq VARUSD2 \leq 2180$) that were updated by only one person ($UNQ\_DES = 1$) were classified as not fault-prone, but those updated by multiple persons were fault-prone. In this case study, low *UNQ_DES* was correlated to few updates (*TOT_UPD*), and hence, less risk of an error; moreover, a single designer also implied less risk of inconsistent updates.

As shown in Table III, the tree based on product, process, and execution metrics for our preferred value of $\theta = 0.92$ had a Type I misclassification rate of $\Pr(fp|nfp) = 26.93\%$ and a Type II rate of $\Pr(nfp|fp) = 25.93\%$. We expect predictions on a current release to have similar accuracy at the end of beta testing when all the significant predictors can be measured. Predictions at this level of accuracy could be very useful to a project during development of the next release. Suppose one considers reengineering all modules predicted to be fault-prone, i.e., 29.90% of the modules. One would expect this set of modules to contain 74.07% of the modules that are actually fault-prone, and thus, the model's predictions would productively target the reengineering evaluation effort. This second model could be used by the project to refine predictions made by the first model. The improvement in Type II misclassification rates could be valuable, because Type II misclassifications are often more important than Type I.

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, C_I. 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. Thus, a classification rule should take into account the costs of each kind of misclassification, if they are known (Khoshgoftaar and Allen, 1998). From a Bayesian viewpoint, the class proportions of the population are information which is known prior to applying a model, that is, the prior probabilities of class membership, π_{nfp} and π_{fp}. The expected cost of misclassification (ECM) of one module (Seber, 1984) is

$$ECM = C_I \Pr(fp|nfp)\, \pi_{nfp} + C_{II} \Pr(nfp|fp)\, \pi_{fp} \tag{8}$$

A general classification rule that minimizes the expected cost of misclassification (Khoshgoftaar and Allen, 1998; Seber, 1984) is

$$Class(\mathbf{x}_i) = \begin{cases} \textit{not fault-prone} & \text{if } \frac{1-\hat{q}(L(\mathbf{x}_i))}{\hat{q}(L(\mathbf{x}_i))} \geq \left(\frac{C_{II}}{C_I}\right)\left(\frac{\pi_{fp}}{\pi_{nfp}}\right) \\ \textit{fault-prone} & \text{otherwise} \end{cases} \tag{9}$$

One can interpret the preferred value of ζ in the context of a classification rule that minimizes the expected cost of misclassifications, even though the generalized-classification rule in Equation (6) does not depend on our knowledge of π_{nfp} and π_{fp}, nor of C_I and C_{II}.

$$\zeta = \left(\frac{C_{II}}{C_I}\right)\left(\frac{\pi_{fp}}{\pi_{nfp}}\right) \tag{10}$$

When the priors ratio is known but the cost ratio is not, a preferred value of ζ implies a subjective assessment of the cost ratio, C_{II}/C_I, according to the minimum-expected-cost rule. In our case study, $\pi_{fp}/\pi_{nfp} = 0.067$. A preferred $\theta = 0.96$ implied $\zeta = \theta/(1-\theta) = 24$, and thus, a subjective assessment of the cost ratio at the end of coding was $C_{II}/C_I = 358$. Similarly, at the end of beta testing, $\theta = 0.92$ implied $C_{II}/C_I = 172$. Considering the extremely high cost of faults discovered by telecommunications-equipment customers, these cost ratios are plausible.

7. Conclusions

Software product and process metrics derived from project data bases, such as configuration management, problem reporting, and installation support, can be useful predictors of which modules are most likely to have customer-discovered faults, and such predictions can be used by software developers and managers to focus improvement efforts before release or reengineering efforts for the next release.

However, in practice, many classification modeling methods in the software-quality literature do not consistently produce a useful balance between the Type I and Type II misclassification rates, especially when the proportion of fault-prone modules is small (Lanubile, 1996). We saw a similar phenomenon with TREEDISC's default classification rule. This paper presents a practical approach to balancing misclassification rates to achieve useful accuracy in the context of a classification-tree model. We present a flexible classification rule that allows appropriate emphasis on each type of misclassification according to the needs of the project. An industrial case study using the TREEDISC algorithm illustrated the tradeoff of misclassification rates. The TREEDISC algorithm is attractive for software quality modeling because it uses chi-squared tests to find the most significant branching criteria, rather than heuristic criteria.

We examined data on two consecutive releases of a very large legacy telecommunications system. This mission-critical system had high reliability requirements. One data set was used for training, and the other was used for evaluation. We built two classification trees using TREEDISC with alternative sets of candidate predictors, which can be measured at the end of coding, and at the end of beta testing, respectively. Both trees had only five significant predictors, out of 28 and 42 candidates, respectively. We interpreted the structure of the classification trees, and we found the models had useful levels of accuracy. The models could be used by the project to assess the risk of faults at the end of coding and to refine those predictions at the end of beta testing. In the case study, the latter model had a lower Type II misclassification rate which could prove valuable, because Type II misclassifications are often more important than Type I.

Future research will investigate ways to build more robust and accurate models with the TREEDISC algorithm, and will explore customized transformation of software metrics. Future research will also compare TREEDISC to other tree-based classification models.

Acknowledgements

We thank Wendell D. Jones and the EMERALD team for collecting the case-study data, and we thank John P. Hudepohl for his encouragement and support. We also thank the anonymous reviewers for their thoughtful comments. This work was supported in part by a grant from Nortel Networks through the Software Reliablity Engineering Department, Research Triangle Park, North Carolina USA. 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

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.

Briand, L. C., V. R. Basili, and C. J. Hetmanski: 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.

Ebert, C.: 1997, 'Experiences with Criticality Predictions in Software Development'. In: *Software Engineering — ESEC/FSE '97: Proceedings of the Sixth European Software Engineering Conference and the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Vol. 1301 of *Lecture Notes in Computer Science*. Zurich, Switzerland, pp. 278–293, Springer-Verlag. Also published as *ACM SIGSOFT Software Engineering Notes*. Vol.22 No.6, November 1997.

Fayyad, U. M.: 1996, 'Data Mining and Knowledge Discovery: Making Sense Out of Data'. *IEEE Expert* **11**(4), 20–25.

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.

Hand, D. J.: 1998, 'Data Mining: Statistics and More?'. *The American Statistician* **52**(2), 112–118.

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, S. and S. Wake: 1991, 'Predicting Maintainability with Software Quality Metrics'. *Journal of Software Maintenance: Research and Practice* **3**, 129–143.

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). In press.

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, pp. 244–251.

Khoshgoftaar, T. M., E. B. Allen, R. Halstead, G. P. Trio, and R. Flass: 1998a, 'Process Measures for Predicting Software Quality'. *Computer* **31**(4), 66–72.

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

Khoshgoftaar, T. M., E. B. Allen, X. Yuan, W. D. Jones, and J. P. Hudepohl: 1999b, '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.

Kitchenham, B. A., S. L. Pfleeger, and N. E. Fenton: 1995, 'Towards a Framework for Software Measurement Validation'. *IEEE Transactions on Software Engineering* **21**(12), 929–944. See comments in (Kitchenham et al., 1997; Morasca et al., 1997).

Kitchenham, B. A., S. L. Pfleeger, and N. E. Fenton: 1997, 'Reply to: Comments on "Towards a Framework for Software Measurement Validation"'. *IEEE Transactions on Software Engineering* **23**(3), 189. See (Kitchenham et al., 1995; Morasca et al., 1997; Weyuker, 1988).

Lanubile, F.: 1996, 'Why Software Reliability Predictions Fail'. *IEEE Software* **13**(4), 131–132,137.

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.

Morasca, S., L. C. Briand, V. R. Basili, E. J. Weyuker, and M. V. Zelkowitz: 1997, 'Comments on "Towards a Framework for Software Measurement Validation"'. *IEEE Transactions on Software Engineering* **23**(3), 187–188. See (Kitchenham et al., 1995; Weyuker, 1988).

Munson, J. C. and T. M. Khoshgoftaar: 1992, 'The Detection of Fault-Prone Programs'. *IEEE Transactions on Software Engineering* **18**(5), 423–433.

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.: 1995a, 'Controlling and Predicting the Quality of Space-Shuttle Software Using Metrics'. *Software Quality Journal* **4**(1), 49–68.

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

Schneidewind, N. F.: 1999, 'Software Quality Maintenance Model'. In: *Proceedings: IEEE International Conference on Software Maintenance.* Oxford, England, pp. 277–286.

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.

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.

Weyuker, E. J.: 1988, 'Evaluating Software Complexity Measures'. *IEEE Transactions on Software Engineering* **14**(9), 1357–1365.

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.* Englewood Cliffs, NJ: Prentice-Hall, 2d edition.

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*.

Xiaojing Yuan
received the M.S. degree in computer science from Florida Atlantic University, Boca Raton, Florida USA, in 1999. She is currently an application developer with Interactive Response Technologies, Inc. Her research interests include software engineering and empirical studies.

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/.