Classification Tree Models of Software Quality Over Multiple Releases

Taghi M. Khoshgoftaar*
Edward B. Allen
Florida Atlantic University
Boca Raton, Florida USA

Wendell D. Jones
John P. Hudepohl
EMERALD, a Business Unit of Nortel Networks
Research Triangle Park, North Carolina USA

Abstract

Software quality models are tools for focusing software enhancement efforts. Such efforts are essential for mission-critical embedded software, such as telecommunications systems, because customer-discovered faults have very serious consequences and are very expensive to repair. This paper presents an empirical study that evaluated software quality models over several releases to address the question, "How long will a model yield useful predictions?"

*This paper also introduces the Classification And Regression Trees (*CART*) algorithm to software reliability engineering practitioners. We present our method for exploiting* CART *features to achieve a preferred balance between the two types of misclassification rates. This is desirable because misclassifications of fault-prone modules often have much more severe consequences than misclassifications of those that are not fault-prone.*

We developed two classification-tree models based on four consecutive releases of a very large legacy telecommunications system. Forty-two software product, process, and execution metrics were candidate predictors. The first software quality model used measurements of the first release as the training data set and measurements of the subsequent three releases as evaluation data sets. The second model used measurements of the second release as the training data set and measurements of the subsequent two releases as evaluation data sets. Both models had accuracy that would be useful to developers. One might suppose that software quality models lose their value very quickly over successive releases due to evolution of the product and the underlying development processes. We found the models remained useful over all the releases studied.

Much of the software metrics literature has shown that software product metrics can be significant predictors of fault-prone *modules. This case study showed that process and execution metrics can also be significant predictors.*

Keywords*: software reliability, software metrics, fault-prone modules, classification trees, CART*

1. Introduction

Predicting which modules are likely to have faults during operations is important to software developers. 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. Software quality models are tools for focusing software enhancement efforts. Such models yield timely predictions on a module-by-module basis, enabling one to target high-risk modules. 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 [29]. This paper presents an empirical case study that evaluated software quality models over several releases to address the question, "How long will a model yield useful predictions?" [3]

The field of software metrics posits that characteristics of software products and their development processes strongly influence the quality of the released product, and its residual faults, in particular [9, 12]. The more complex the product is, the more likely that developers will make mistakes causing faults and consequent failures. Research has also found that process attributes, such as reuse [1, 20], the history of corrected faults [18], and the experience of programmers [19] can be significantly associated with faults. Because prod-

*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/.

uct 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 [31]. Recent research [15] has also found that forecasts of execution metrics can be significantly related to operational faults. Our case study considered forty-two metrics.

The multitude of software product, process, and execution metrics can seem overwhelming. Some try to identify one metric that is the key to software quality. However, due to the variety and complexity of software products, processes, and operating conditions, no single metric can consistently give accurate predictions. Predicting software faults is a multivariate problem. Instead of relying on programmer intuition, software quality models use computational intelligence and statistical modeling techniques to predict risk indicators that developers and managers are familiar with.

Lyu et al. [26] report on a prototype system to support software developers with software quality models. Similarly, Enhanced Measurement for Early Risk Assessment of Latent Defects (EMERALD) [13] is a sophisticated system of decision support tools used by software designers and managers to assess risk and improve software quality on a routine basis [14]. It was developed by Nortel (Northern Telecom) in partnership with Bell Canada and others. At various points in the development process, EMERALD's software quality models predict which modules are likely to be fault-prone, based on available measurements. Systems such as EMERALD are the key to improved software quality. For example, risk predictions at the time of release can be used to target reengineering efforts to those modules in the next release that need improvement the most, before problems are reported by customers on the latest release.

This paper introduces the Classification And Regression Trees (CART) algorithm [4] to software reliability engineering practitioners, and presents a case study that evaluated the usefulness of classification tree models over several releases. The case study developed two classification-tree models based on four consecutive releases of a very large legacy telecommunications system. The first software quality model used measurements of the first release as the training data set and measurements of the subsequent three releases as evaluation data sets. The second model used measurements of the second release as the training data set and measurements of the subsequent two releases as evaluation data sets. We interpreted the models and evaluated their accuracy.

2. Related Work

A classification tree is an algorithm, depicted as a tree graph, that classifies an input object, such as a software module. Alternative classification techniques used in software quality modeling include discriminant analysis [20], the discriminative power technique [30], logistic regression [2], pattern recognition [5], artificial neural networks [24], and fuzzy classification [7]. 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. We follow terminology in the classification-tree literature, calling independent variables "predictors", and the dependent variable the "response" variable.

Several algorithms for building classification trees have appeared in the software engineering literature. Selby and Porter [32] first transformed measurements into nominal predictors. Their algorithm recursively partitions the training set of modules using an entropy-based criterion. Each decision node consists of a selected predictor and each child node 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.

Takahashi, Muraoka, and Nakamura [34] extend the algorithm used by Selby and Porter by applying Akaike Information Criterion procedures 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.

Case studies by Gokhale and Lyu [11] and by Troster and Tian [35] used regression trees with real response variables and real 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 [6]. Gokhale and Lyu use cost-complexity based on deviance to select the proper size for the tree. Troster and Tian suggest that after building a tree, the analyst perform "intelligent pruning" when a decision node yields little insight, in the view of the analyst.

We have used the TREEDISC algorithm to build classification-tree models based on discrete ordinal predictors [17, 23, 22]. It is a refinement of the CHAID algorithm, which automatically determines the most significant split criteria based on chi-squared tests, and is implemented as a macro package for the SAS® System. It recursively partitions the training set of modules. It finds the most significant partitioning of a predictor's original categories with respect to the dependent variable. Like the entropy-based algorithm, TREEDISC allows multiway splits, but it minimizes the adjusted p-value of chi-squared tests to determine the ranges.

CART [4], which is the focus of this paper, automatically builds a parsimonious tree from continuous ordinal predictors by first building a maximal tree and then pruning it to an appropriate level of detail. CART is attractive because it emphasizes pruning to achieve robust models. Although Kitchenham briefly reports using CART to model software project productivity [25], to our knowledge, CART has seldom been used to model software quality [21, 28].

3. System Description

We conducted a case study 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. The entire system had significantly more than ten million lines of code. This embedded-computer application included numerous finite-state machines and interfaces to other kinds of equipment. We studied four consecutive releases which we labeled 1 through 4.

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 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.

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. Consequently, 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. These modules had several million lines of code in a few thousand modules in each release.[1] The proportion of modules with no faults among the updated modules of Release 1 was 0.937, and the proportion with at least one fault was 0.063. Such a small set of modules is often difficult for a software quality model to identify.

We used forty-two software product, process, and execution metrics as candidate predictors. Pragmatic considerations usually determine the set of available software metrics. We do not advocate collecting a particular set of metrics for software quality models to the exclusion of others recommended in the literature. We prefer a data-mining approach to exploiting metric data [10], analyzing a broad set of metrics, rather than limiting data collection according to predetermined research questions. Collection of product-metric data involved extracting source code from the configuration management system and measuring it using EMERALD's software metrics analysis tool. Because marginal data collection costs were modest, EMERALD provided over fifty source-code metrics [27]. Preliminary data analysis selected metrics aggregated to the module level that were appropriate for modeling purposes. The software product metrics used in this study are listed below. which were based on three abstractions of software: call graphs, control flow graphs, and statements. A call graph depicts calling relationships among procedures. A control flow graph consists of nodes and arcs depicting the flow of control. Statement metrics quantify statement attributes.

Process metrics listed below were tabulated from configuration management and problem reporting data. The configuration management system identified the designer that made each update. The problem reporting system maintained records on past problems.

Execution metrics listed below were forecast from deployment records [15] and laboratory measurements of execution times of an earlier release. Future research will refine these metrics.

Symbol: Description

Call Graph Metrics

$CALUNQ$: Number of distinct procedure calls to others.

$CAL2$: Number of second and following calls to others. $CAL2 = CAL - CALUNQ$ where CAL is the total number of calls.

Control Flow Graph Metrics

$CNDNOT$: Number of arcs that are not conditional arcs.

[1]Exact counts are proprietary, so that the system and its releases remain anonymous.

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.

Software Process Metrics
DES_PR: Number of problems found by designers
BETA_PR: Number of problems found during beta testing
DES_FIX: Number of problems fixed that were found by designers
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 (deleted lines are not counted)
UNQ_DES: Number of different designers making changes
VLO_UPD: Number of updates to this module by designers who had 10 or less total updates in entire company career.
LO_UPD: Number of updates to this module by designers who had between 11 and 20 total updates in entire company career
UPD_CAR: Number of updates that designers had made in their company careers

---

Software Execution Metrics
USAGE: Deployment percentage of the module.
RESCPU: Execution time (microseconds) of an average transaction on a system serving consumers.
BUSCPU: Execution time (microseconds) of an average transaction on a system serving businesses.
TANCPU: Execution time (microseconds) of an average transaction on a tandem system.

---

4. Classification Tree Modeling

A classification tree represents an algorithm as an abstract tree of decision rules to classify a module. 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.

The Classification And Regression Trees (CART) algorithm [4] builds a classification tree. It is implemented as a supplementary module for the SYSTAT package [33]. We model each module with a set of continuous ordinal predictors, namely, software product, process, and execution metrics, and a nominal response variable with two categories, *not fault-prone* and *fault-prone*.

Beginning with all training modules in the root node, the CART algorithm recursively partitions ("splits") the 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 [4]. Further splitting is impossible if only one module is in a leaf, or if all modules have exactly the same measurements. CART also stops splitting if a node has too few modules (e.g., less than 10 modules). The result of this process is typically a large tree. Usually, such a maximal tree overfits the data set, and consequently, is not robust. CART then generates a series of trees by progressively pruning branches from the maximal tree. The accuracy of each size of tree in the series is estimated and the most accurate tree in the series is selected as the final classification tree.

CART uses *ν-fold cross-validation* to estimate the accuracy of each pruned tree. This method uses the training data set to make an unbiased estimate of model accuracy [8, 11]. The algorithm has these steps: Randomly divide the sample into ν approximately equal subsets (e.g., $\nu = 10$). Set aside one subset as a test sample, and build a tree with the modules in the remaining $\nu - 1$ subsets. Classify the modules in the test subset and note the accuracy of each prediction. A *Type I* misclassification is when the model classifies a module as *fault-prone* which is actually *not fault-prone*. A *Type II* misclassification is when the model classifies a module as *not fault-prone* which is actually *fault-prone*. Repeat this process, setting aside each subset in turn. Calculate the overall accuracy in terms of the Type I and Type II misclassification rates. This is an estimate of the accuracy of the tree built using all the modules. The number of subsets, ν, should not be small; Breiman et al. [4] found that ten or more worked well.

Due to different costs associated with each type of misclassification, we have devised a way to provide appropriate emphasis on Type I and Type II misclassification rates according to the needs of the project. Let π_{fp} and π_{nfp} be prior probabilities of membership in the *fault-prone* and *not fault-prone* classes, respectively, and let C_I and C_{II} be the costs of Type I and Type II misclassifications, respectively. We experimentally choose a preferred value of the parameter ζ, which can be interpreted as a priors ratio times a cost ratio.

$$\zeta = \left(\frac{\pi_{fp}}{\pi_{nfp}}\right)\left(\frac{C_{II}}{C_I}\right) \qquad (1)$$

CART allows one to specify prior probabilities, and costs of misclassifications as parameters to account for the underlying distribution of faults and disparate consequences of misclassifications. These four parameters are used when evaluating goodness-of-split of a node as a tree is recursively generated. We observed that for a given value of ζ, CART generates the same tree, irrespective of the four components of ζ. There is a tradeoff between the Type I and the Type II misclassification rates, as functions of ζ. Generally, as one goes down, the other goes up. We empirically estimate the relationships between ζ and the misclassification rates by repeated calculations with the training data set. Given a candidate value of ζ, we build a tree and estimate the Type I and Type II rates using ν-fold cross-validation. We repeat for various values of ζ, until we arrive at the preferred ζ for the project. This is straightforward in practice, because the misclassification rates are approximately monotonic functions of ζ. (We have proposed a generalized classification rule applied to discriminant analysis, which has a similar parameter [16].)

5. Empirical Results

We acquired fault data and software product, process, and execution measurements of four consecutive releases of the system under study. Release 1's data set was a training data set and data on the subsequent releases were evaluation data sets, and similarly, Release 2's data set was another training data set. Candidate predictors were all the metrics listed above. We found that CART was not able to build a tree when π_{fp} and π_{nfp} had extreme values (e.g. $\pi_{fp} < 0.1$). Therefore, we set $\pi_{fp} = 0.2$, $\pi_{nfp} = 0.8$, and $C_I = 1$, and then set C_{II} to achieve various values of ζ. In this case study, we preferred the value of ζ that most nearly balanced Type I and Type II misclassification rates of 10-fold cross-validation of the training data set. Another balance might be preferred in other circumstances.

CART built Model 1, shown in Figure 1, based on measurements of Release 1 as the training data set, and $\zeta = 1.05$. Let us examine details of the resulting classification tree (Figure 1). Given a module's measurements, the algorithm first considers at Node 1 the number of file-includes (*FILINCUQ*). Because include-files are used for declaring shared data structures and procedure prototypes, *FILINCUQ* is associated with the variety of procedure interfaces. If there were few interfaces ($FILINCUQ \leq 34$) then we proceed to Node 2. Otherwise, we proceed to Node 8. At Node 2, if the number of designers that modified the module was large ($UNQ\_DES > 5$), then the module is classified as *fault-prone*. Otherwise, we proceed to Node 3. In other words, requiring increased coordination among designers increases risk. At Node 3, if the number of new or changed lines of code was zero

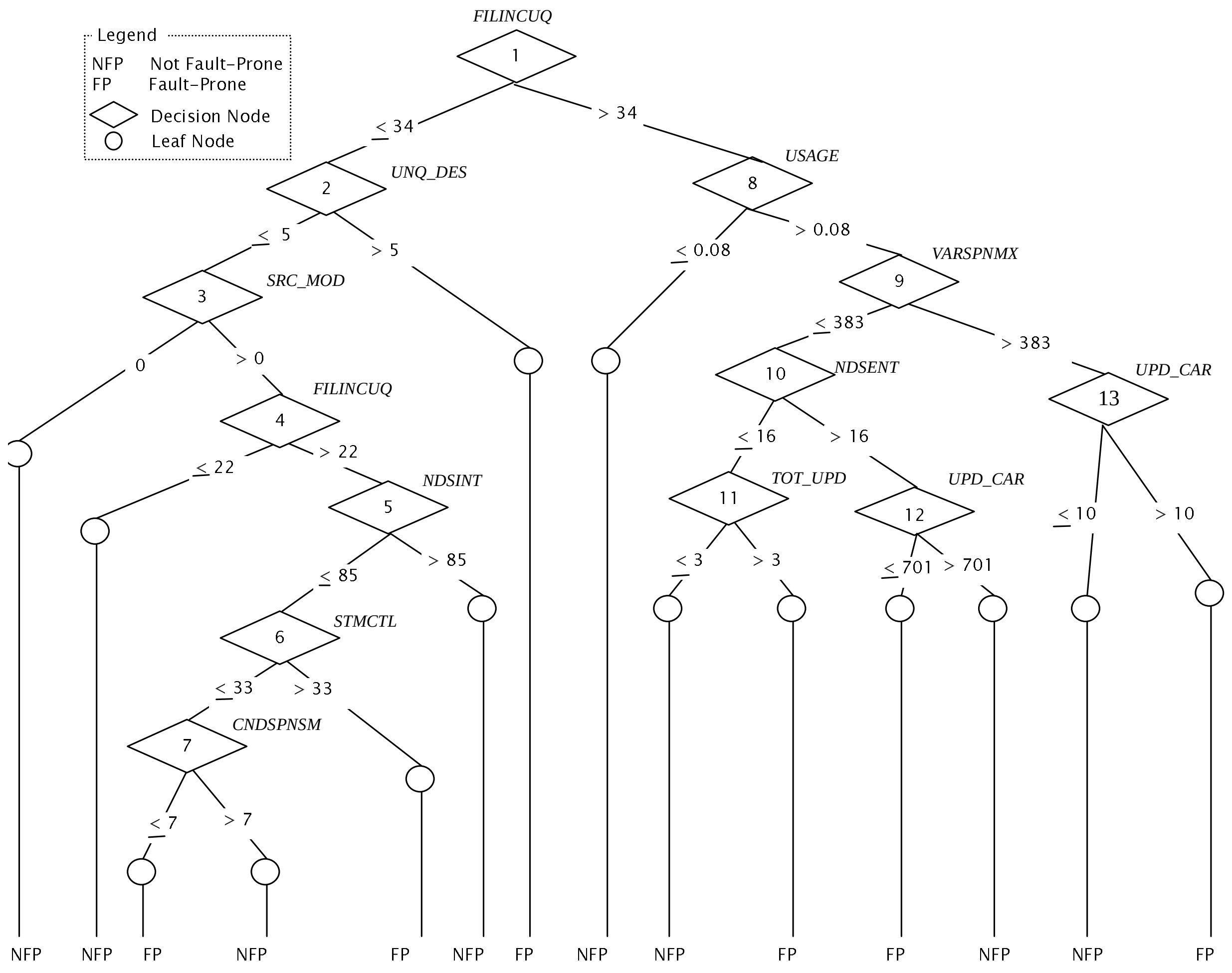

Figure 1. Classification Tree Based on Release 1

(*SRC_MOD* $= 0$, i.e., no additions or changes to code; all updates consisted of deleting lines of code only), then the module is classified as *not fault-prone*. Otherwise, we proceed to Node 4. Similarly, Nodes 4, 5, 6, and 7 classify modules according to combinations of the following product metrics: file-includes (*FILINCUQ*), internal nodes (*NDSINT*), control statements (*STMCTL*), and span of conditional structures (*CNDSPNSM*). At Node 8, if the usage level was very small (*USAGE* ≤ 0.08) then the module is classified as *not fault-prone*. Otherwise, we proceed to Node 9. In other words, if the module was installed at few sites then customers are unlikely to discover faults. However, defects in a little-used module may remain undiscovered. At Node 9, if the maximum span of variables was not large (*VARSPNMX* ≤ 383), then we proceed to Node 10. Otherwise, we proceed to Node 13. The maximum span of variables (*VARSPNMX*) is the largest number of lines of code between first and last use within a procedure of any variable in the module. A large maximum span may make overlooking a variable in the code more likely. At Node 10, if the module was not large (*NDSENT* ≤ 16), then we proceed to Node 11. Otherwise, we proceed to Node 12. Because all procedures in this system have only one entry point, *NDSENT* is equivalent to the number of procedures in the module which is a rough measure of size. At Node 11, if there were few updates (*TOT_UPD* ≤ 3) then the module is classified as *not fault-prone*. Otherwise, it is considered *fault-prone*. Each update is an opportunity for a designer to make a mistake. Nodes 12 and 13 examine total updates in designers' careers (*UPD_CAR*), in combination with the nodes above them. The threshold at Node 12 isolates large numbers of career updates (*UPD_CAR* > 701) as *not fault-prone*. Perhaps a module with a large number of career updates was modified by experienced designers, and thus, there were few mistakes. The threshold at Node 13 isolates small numbers of career updates (*UPD_CAR* ≤ 10) also as *not fault-prone*. There was a strong mentoring program for new employees on this project. Perhaps a module with a small number of career updates was modified by a new designer who had a mentor reviewing the change, and thus, there were few mistakes. Further in-depth analysis of configuration management data may substantiate the reasons for *UPD_CAR*'s significance. In summary, this model found eleven significant predictors out of forty-two candidates.

Suppose one used Model 1 results until problems reports for Release 2 became available. Would a new model based on Release 2 be more accurate for future releases? Using CART, we built Model 2 based on measurements of Release 2 as the training data set, and $\zeta = 0.95$, which was the value of ζ that most nearly balanced the Type I and Type II misclassification rates with its training data set. We evaluated both models by cross-validation of the training data sets and by classifying subsequent releases, the evaluation data sets. This simulated using the models in practice. The Type I and Type II misclassification rates are listed in Table 1. The results in Table 1 represent accuracy that would be useful to developers. Recall that customer-discovered faults have very serious consequences and are very expensive to repair. Given Model 1, for example, correctly predicting 71.4% of the *fault-prone* modules in Release 2 would give designers the opportunity to reengineer many of those *fault-prone* modules in the early phases of Release 3's development, before fault data on Release 2 became available.

Skeptics of software quality modeling often assume that models lose their value very quickly. In this case study, project personnel felt the development process was reasonably stable. Although the accuracy of each model did vary somewhat from release to release, we found the models still could be useful over all the releases studied. Apparently, the process metrics in the model captured the important aspects of the project's slowly evolving development process. Moreover, Model 1 was still competitive with Model 2 at Release 4. Evidently, even though the product and process had evolved, the relationship between software metrics and faults, as captured by Model 1, was reasonably stable. We conclude that software quality models in practice can be useful for a longer time than some might have supposed.

6. Conclusions

Software quality models are tools for focusing software enhancement efforts. Such efforts are essential for mission-critical embedded software, such as telecommunications systems, because customer-discovered faults have very serious consequences and are very expensive to repair. For example, risk predictions at the time of release can be used to target reengineering efforts to those modules in the next release that need improvement the most, before problems are reported by customers on the latest release.

This paper introduces the Classification And Regression Trees (CART) algorithm [4] to software reliability engineering practitioners. CART is attractive because it prunes large classification trees to yield robust models. We found that CART is amenable to achieving a preferred balance between the two types of misclassification rates. This is desirable because Type II misclassifications often have much more severe conse-

Table 1. Accuracy of Classification Trees

Misclassification rate (%)

	Model 1			Model 2		
Release	ζ	Type I	Type II	ζ	Type I	Type II
1	*1.05*	*27.7*	*26.6*			
2		27.9	28.6	*0.95*	*20.1*	*34.4*
3		30.4	34.0		27.4	27.6
4		33.7	27.2		31.5	35.9

Cross-validation of training data set is *italic*.

quences than Type I. We defined a parameter ζ as the product of the ratio of CART's prior probabilities (π_{fp}/π_{nfp}) times the ratio of CART's costs of misclassification (C_{II}/C_I). We observed that CART generates the same tree for given ζ irrespective of the four components, and that each type of misclassification rate is approximately a monotonic function of ζ. Consequently, one can select the value of ζ that achieves the balance between the types of misclassifications that the project prefers.

We developed two classification-tree models based on four consecutive releases of a very large legacy telecommunications system. The first software quality model used measurements of the first release as the training data set and measurements of the subsequent three releases as evaluation data sets. The second model used measurements of the second release as the training data set and measurements of the subsequent two releases as evaluation data sets. This simulated using a model in practice.

Much of the software metrics literature has shown that software product metrics can be significant predictors of *fault-prone* modules. This case study showed that process and execution metrics can also be significant predictors [15, 18]. Model 1 had eleven significant predictors out of forty-two candidates. Analysis of these predictors yielded insights into various software development practices. We can infer from the first model that the following were significant features of Release 1 (Figure 1). The number of interfaces (*FILINCUQ*) was the most significant predictor (Node 1). Too many different designers updating a module ($UNQ\_DES > 5$) increases risk (Node 2). Deletion of lines of code alone ($SRC\_MOD = 0$) has low risk (Node 3). Low usage ($USAGE \leq 0.08$) implies little opportunity for customers to discover faults (Node 8). Few updates ($TOT\_UPD \leq 3$) implies few opportunities for mistakes (Node 11). Judicious use of experienced personnel, and a strong mentoring program can reduce risk (Nodes 12 and 13). Each of these insights were significant in contexts limited by other predictors, as shown by the classification tree's structure.

Both models had accuracy that would be useful to developers. One might suppose that software quality models lose their value very quickly over successive releases due to evolution of the product and the underlying development processes. We found that the models still could be useful over all the releases studied. In fact, Model 1 was still competitive with Model 2 at Release 4, in spite of its age.

Future research will compare CART to other classification-tree techniques for software quality modeling.

Acknowledgments

We thank the EMERALD team for collecting the data, and we thank Archana Naik for performing the experiments. We thank the anonymous reviewers for their thoughtful comments. This work was supported in part by a grant from Nortel through the Software Reliability Engineering Department in 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

[1] V. R. Basili, L. C. Briand, and W. Melo. How reuse influences productivity in object-oriented systems. *Communications of the ACM*, 39(10):104–116, Oct. 1996.

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

[3] S. Biyani and P. Santhanam. Exploring defect data from development and customer usage on software modules over multiple releases. In *Proceedings the Ninth International Symposium on Software Reliability Engineering*, pages 316–320, Paderborn, Germany, Nov. 1998. IEEE Computer Society.

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

[5] L. C. Briand, V. R. Basili, and W. M. Thomas. A pattern recognition approach for software engineering data analysis. *IEEE Transactions on Software Engineering*, 18(11):931–942, Nov. 1992.

[6] L. A. Clark and D. Pregibon. Tree-based models. In J. M. Chambers and T. J. Hastie, editors, *Statistical Models in S*, pages 377–419. Wadsworth, Pacific Grove, California, 1992.

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

[8] B. Efron. Estimating the error rate of a prediction rule: Improvement on cross-validation. *Journal of the American Statistical Association*, 78(382):316–331, June 1983.

[9] W. M. Evanco and W. W. Agresti. A composite complexity approach for software defect modeling. *Software Quality Journal*, 3(1):27–44, Mar. 1994.

[10] U. M. Fayyad. Data mining and knowledge discovery: Making sense out of data. *IEEE Expert*, 11(4):20–25, Oct. 1996.

[11] S. S. Gokhale and M. R. Lyu. Regression tree modeling for the prediction of software quality. In H. Pham, editor, *Proceedings of the Third ISSAT International Conference on Reliability and Quality in Design*, pages 31–36, Anaheim, CA, Mar. 1997. International Society of Science and Applied Technologies.

[12] J. Henry, S. Henry, D. Kafura, and L. Matheson. Improving software maintenance at Martin Marietta. *IEEE Software*, 11(4):67–75, July 1994.

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

[14] J. P. Hudepohl, S. J. Aud, T. M. Khoshgoftaar, E. B. Allen, and J. Mayrand. Integrating metrics and models for software risk assessment. In *Proceedings of the Seventh International Symposium on Software Reliability Engineering*, pages 93–98, White Plains, NY, Oct. 1996. IEEE Computer Society.

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

[16] T. M. Khoshgoftaar and E. B. Allen. A practical classification rule for software quality models. *IEEE Transactions on Reliability*, 48, 1999. Forthcoming.

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

[18] T. M. Khoshgoftaar, E. B. Allen, R. Halstead, G. P. Trio, and R. Flass. Process measures for predicting software quality. *Computer*, 31(4):66–72, Apr. 1998.

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

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

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

[22] T. M. Khoshgoftaar, E. B. Allen, X. Yuan, W. D. Jones, and J. P. Hudepohl. Assessing uncertain predictions of software quality. In *Proceedings of the Sixth International Software Metrics Symposium*, Boca Raton, Florida USA, Nov. 1999. IEEE Computer Society. Forthcoming.

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

[24] T. M. Khoshgoftaar and D. L. Lanning. 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, Apr. 1995.

[25] B. A. Kitchenham. A procedure for analyzing unbalanced datasets. *IEEE Transactions on Software Engineering*, 24(4):278–301, Apr. 1998.

[26] M. R. Lyu, J. S. Yu, E. Keramidas, and S. R. Dalal. ARMOR: Analyzer for reducing module operational risk. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, pages 137–142, Pasadena, CA, June 1995. IEEE Computer Society.

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

[28] A. Naik. Prediction of software quality using classification tree modeling. Master's thesis, Florida Atlantic University, Boca Raton, FL USA, Dec. 1998. Advised by Taghi M. Khoshgoftaar.

[29] N. F. Schneidewind. Methodology for validating software metrics. *IEEE Transactions on Software Engineering*, 18(5):410–422, May 1992.

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

[31] N. F. Schneidewind. An integrated process and product model. In *Proceedings Fifth International Software Metrics Symposium*, pages 224–234, Bethesda, MD USA, Nov. 1998. IEEE Computer Society.

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

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

[34] R. Takahashi, Y. Muraoka, and Y. Nakamura. Building software quality classification trees: Approach, experimentation, evaluation. In *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, pages 222–233, Albuquerque, NM USA, Nov. 1997. IEEE Computer Society.

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