International Journal of Software Engineering and Knowledge Engineering
Vol. 0, No. 0 (1993) 000–000

USING CLASSIFICATION TREES FOR SOFTWARE QUALITY MODELS: LESSONS LEARNED

TAGHI M. KHOSHGOFTAAR
EDWARD B. ALLEN
ARCHANA NAIK
Department of Computer Science and Engineering
Florida Atlantic University
Boca Raton, Florida 33431 USA

and

WENDELL D. JONES
JOHN P. HUDEPOHL
Software Reliability Engineering Dept.
Nortel
Research Triangle Park, NC 27709 USA

Received (received date)
Revised (revised date)
Accepted (accepted date)

High software reliability is an important attribute of high-assurance systems. Software quality models yield timely predictions of quality indicators on a module-by-module basis, enabling one to focus on finding faults early in development. This paper introduces the Classification And Regression Trees (CART) algorithm to practitioners in high-assurance systems engineering. This paper presents practical lessons-learned on building classification trees for software quality modeling, including an innovative way to control the balance between misclassification rates. A case study of a very large telecommunications system used CART to build software quality models. The models predicted whether or not modules would have faults discovered by customers, based on various sets of software product and process metrics as independent variables. We found that a model based on two software product metrics had comparable accuracy to a model based on forty product and process metrics.

Keywords: software quality, fault-prone modules, classification trees, CART, software product metrics, software process metrics

1. Introduction

High-assurance systems often must have high software reliability. Because the risk of poor reliability varies among software modules, it is often difficult to guide software enhancement efforts effectively and efficiently. According to standard terminology, a "fault" is a defect in a program that may cause incorrect execution [1]. "Enhancement techniques" is our term for any set of processes for finding faults early. We focus on faults discovered by users, irrespective of whether a failure resulting from a fault would be frequent or rare. Even rare failures can be very important in high-assurance systems.

Due to the high cost of correcting problems discovered by customers, the goal of our modeling is identification of fault-prone modules early in development. Software quality models are tools for focusing efforts to find faults. Such models yield timely predictions of quality on a module-by-module basis, enabling one to target enhancement techniques.

Enhanced Measurement for Early Risk Assessment of Latent Defects (EMERALD) is an example of a sophisticated system of decision support tools used by software designers and managers to assess risk and improve software quality [2]. It was developed by Nortel (Northern Telecom) in partnership with Bell Canada and others. EMERALD provides access to software product metrics, deployment usage, fault histories, other software process metrics, and related software quality models. 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 like EMERALD are the key to improved software quality. For example, suppose EMERALD's models indicate that certain modules are at-risk when changed. After a review, the design team may choose to reengineer these modules first to reduce risk before continuing their implementation of the current release. Moreover, any changes proposed for high-risk modules may require more stringent justification and inspection.

This paper introduces the Classification And Regression Trees (CART) algorithm [3] to software engineering practitioners. A "classification tree" is an algorithm, depicted as a tree graph, that classifies an input object. This study confirms prior empirical work [4, 5, 6, 7] showing that classification trees can be useful to identify fault-prone modules based on the pattern of software metrics. Alternative classification techniques include discriminant analysis [8], the discriminative power technique [9], logistic regression [10], pattern recognition [11], artificial neural networks [12], and fuzzy classification [13]. A classification tree differs from these in the way it models complex relationships between class membership and combinations of variables. CART automatically builds a parsimonious tree 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.

This paper gives practical lessons-learned on building classification trees for software quality modeling [14]. In particular, we present an innovative way to control the balance between misclassification rates. This paper extends preliminary results indicating that CART can be useful for software quality modeling [15]. A case study of a very large telecommunications system used CART to build software quality models. This study focused on problems discovered in the field by customers. If any problems discovered by customers resulted in changes to a module's source code, then the module was considered *fault-prone*. Rework of *fault-prone* modules after release is a major concern of many software development organizations. The models predicted whether or not modules were fault-prone, based on various sets

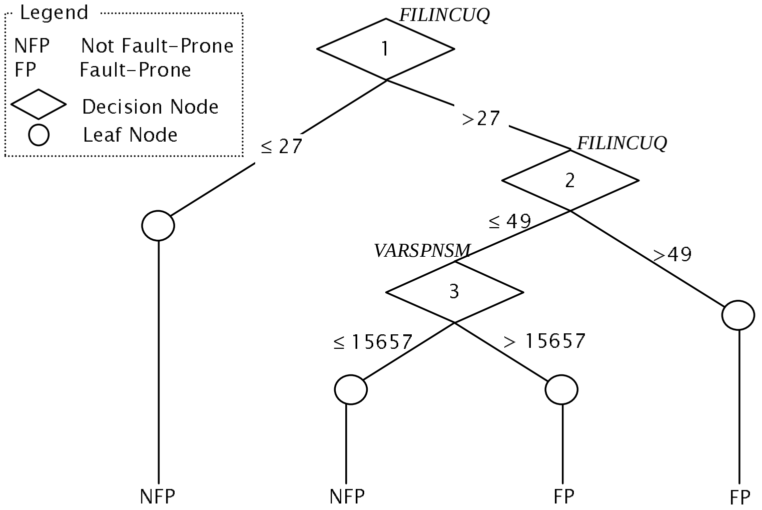

Fig. 1. Tree Based on Product Metrics

of software product and process metrics as independent variables. The results of this paper are being refined and further evaluated for the EMERALD system. The remainder of this paper presents how to use classification trees, a summary of the CART algorithm, details on our case study, and conclusions.

2. Classification Trees Using CART

2.1. *Using a Classification Tree: An Example*

Suppose we have measurements on one software module. For example, one metric could be the number of unique file-include directives in the code, *FILINCUQ*. Because include files ("header files") are often used to implement interfaces and data abstractions, this is an attribute of the high-level design. Another could be the total span of variables, *VARSPNSM*, where the span of one variable is the number of lines of code between the first and last use of the variable in a procedure. This is a measure of locality of reference. It is an attribute of the coding.

In this application, a classification tree represents an algorithm as an abstract tree of decision rules to classify a module as a member of the *not fault-prone* group or the *fault-prone* group. Figure 1 depicts a classification tree created in our case study, described below. Each diamond node represents a decision, and each edge represents a possible result of that decision. Each circular node is a leaf that classifies a module into the group noted at the bottom. The root of the tree is the node at the top.

A module is represented by its measurements. Beginning at the root node, the algorithm traverses a downward path in the tree, one node after another, until it reaches a leaf node. The current decision node is applied to one measurement. For example, in Figure 1, Node 1 examines the module's measurement of *FILINCUQ*.

If *FILINCUQ* $\leq$ 27 then the algorithm chooses the left edge to a leaf where the module is classified as *not fault-prone*. In other words, few interfaces implies mistakes discovered by customers are not likely. Otherwise, the algorithm proceeds to Node 2 which is another decision. At Node 2, if *FILINCUQ* $\leq$ 49 then the algorithm proceeds to Node 3. Otherwise, the algorithm proceeds to the right. In other words, many interfaces implies customers are more likely to discover mistakes. Node 3 examines *VARSPNSM*. If *VARSPNSM* $\leq$ 15657, then the module is classified as *not fault-prone*, and otherwise, *fault-prone*. In other words, when there is a medium number of interfaces, locality of reference (*VARSPNSM*) indicates a module's class. The process is repeated for each node along the path. When a decision node is reached, it is applied to the module. 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 such an algorithm. Designers can readily interpret the decision nodes. Some other classification techniques, such as discriminant analysis [8] or neural networks [12], can be difficult to interpret in software engineering terms.

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*.

2.2. *Building a Classification Tree*

The Classification And Regression Trees (CART) algorithm [3] builds a classification tree. It is implemented as a supplementary module for the SYSTAT package [16]. We follow terminology in the classification tree statistics literature, calling independent variables "predictors". We model each module with a set of continuous ordinal-scaled predictors, namely, software product and process metrics, and a nominal-scaled dependent variable (a "response" variable) with two categories, *not fault-prone* or *fault-prone*.

Beginning with all modules in the root node, the algorithm recursively partitions ("splits") the set into two leaves until a stopping criterion applies to every leaf node. A goodness-of-split criterion is used to minimize the heterogeneity ("node impurity") of each leaf at each stage of the algorithm. 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.

Early work with classification trees in software engineering selected branches by a measure of homogeneity until a stopping rule was satisfied [5, 6]. More recent work has used an algorithm based on deviance [7], and an algorithm based on statistical significance [4]. CART's default goodness-of-split criterion is the "Gini index of diversity" which is based on probabilities of class membership [3].

Resubstitution is a method for estimating model accuracy by using the model to classify the same modules that were the basis for building the model, and then calculating misclassification rates. The estimated accuracy can be overly optimistic.

ν-fold cross-validation is an alternative method that also uses the same modules as were the basis for building the model, but the estimated accuracy is not biased [17, 18, 19]. 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 remaining modules. Classify the modules in the test subset and note the accuracy of each prediction. Repeat this process, setting aside each subset in turn. Calculate the overall accuracy. 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. found that ten or more worked well [3].

CART allows one to specify prior probabilities, and costs of misclassifications. These parameters are used when evaluating goodness-of-split of a node as a tree is recursively generated. 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} are the costs of Type I and Type II misclassifications, respectively.

Due to different costs associated with each type of misclassification, we need a way to provide appropriate emphasis on Type I and Type II misclassification rates according to the needs of the project. (We have proposed a generalized classification rule applied to discriminant analysis, which has a similar constant c [20].) We experimentally choose a constant c, which can be interpreted as a priors ratio times a cost ratio.

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

We have observed a tradeoff between the Type I and the Type II misclassification rates, as functions of c. Generally, as one goes down, the other goes up. We observed that for a given value of c, CART generates the same tree, irrespective of the components of c. We estimate these functions by repeated calculations with the *fit* data set. Given a candidate value of c, we build a tree and estimate the Type I and Type II rates using resubstitution and ν-fold cross-validation. We repeat for various values of c, until we arrive at the preferred c for the project.

3. Case Study

3.1. *System Description*

We conducted a case study of a very large legacy telecommunications system. Such systems require very high reliability. It was written in a high level language, 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. A *module* consisted of a set of related source code files.

A module was considered *fault-prone* if any faults were discovered by customers, and *not fault-prone* otherwise. Faults discovered in deployed systems are typically extremely expensive, because, in addition to down-time due to failures, visits to customer sites are usually required to repair them. Fault data was collected at the module-level by the problem reporting system.

Analysis of configuration management data identified modules that were unchanged from the prior release. Approximately 99% of the unchanged modules had no faults. This case study considered only "updated" modules, i.e., those that were new or had at least one update to source code since the prior release. These modules had several million lines of code in a few thousand modules. The proportion of modules with no faults among the updated modules was $\pi_{nfp} = 0.9260$, and the proportion with at least one fault was $\pi_{fp} = 0.0740$. Such a small set of modules is often difficult to identify early in development.

Tables 1 and 2 list the product and process metrics used in this study. We do not advocate a particular set of metrics for software quality models to the exclusion of others recommended in the literature. Pragmatic considerations usually determine the set of available metrics. We prefer a data mining approach, using analytical techniques to identify metrics that are significantly related to class membership out of a large set of candidate metrics.

USAGE was calculated using data derived from installation records of an earlier release. Deployment plans could also be considered. Considering that this is a legacy system, *USAGE* was a forecast of deployment usage in the current release. The EMERALD system collected the other software product metrics from source code at the procedure level, and then aggregated them to the module level. Attributes of a call graph, control flow graphs, and source code statements were measured [21]. A call graph depicts invocation relationships among procedures. A control flow graph shows the flow of control where each arc represents a series of in-line statements and each node represents a decision statement, such as an IF, FOR, or WHILE statement, or a branch destination.

Metrics of problems found and fixed were derived from problem reporting system data. Personnel attributes were derived from configuration management data. Metrics related to updates by designers, such as *UPD_CAR*, capture different attributes of the personnel that maintained a module, without requiring access to a personnel data base.

3.2. *Empirical Results*

We modeled all the software modules from one release as samples from a population. Using a portion of the modules, we built a classification tree model and then evaluated it by making predictions for the remainder. Our case study's methodology is summarized by the following.

Table 1. Software Product Metrics

Symbol	Description
USAGE	Deployment percentage of module.
Call Graph Metrics	
CALUNQ	Distinct calls to others.
CAL2	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. Unit of measure is arcs.
CNDSPNMX	Maximum span of branches of conditional arcs.
CTRNSTSM	Total control structure nesting.
CTRNSTMX	Maximum 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.

(i) Measure static software product metrics from source code for each module, and derive software process metrics from configuration management data and problem reporting data. Forecast usage from deployment records and installation plans.
(ii) Prepare *fit* and *test* data sets for the set of modules, using data splitting.
(iii) Choose the parameter c to achieve a preferred balance between the Type I and Type II misclassification rates using the *fit* data set. (This involves building a series of trees using the CART algorithm.)
(iv) Build the final classification tree based on the preferred value of c, and the *fit* data set, using the CART algorithm. Calculate the final resubstitution and ν-fold cross-validation misclassification rates.
(v) Classify each module in the *test* data set using the final tree, and calculate misclassification rates.
(vi) Evaluate the model's accuracy estimated by misclassification rates for resubstitution (based on the *fit* data set), cross-validation (based on the *fit* data

Table 2. Software Process Metrics

Symbol	Description
DES_PR	Number of problems found by designers
BETA_PR	Number of problems found during beta testing
TOT_FIX	Total number of problems fixed
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
UNQ_DES	Number of different designers making changes
VLO_UPD	Number of updates to this module by designers who had 10 or less total updates in entire company career.
LO_UPD	Number of updates to this module by designers who had between 11 and 20 total updates in entire company career
UPD_CAR	Number of updates that designers had in their company careers

set), and data splitting (based on the *test* data set).

The model is ready to use on a similar project or subsequent release where measurements are available.

We impartially divided the available data on updated modules into approximately equal *fit* and *test* data sets. The *fit* data set was used to build the model, and the *test* data set was used to evaluate its accuracy. This assured that the *test* data set's observations were independent of those in the *fit* data set, facilitating an unbiased estimate of model accuracy. These proportions were chosen to provide an adequate statistical sample in each data set. Other proportions might be appropriate in another study, according to the needs of the modeling technique.

We applied the CART algorithm to the *fit* data set to build classification tree models that identify *fault-prone* modules. We chose to use 10-fold cross-validation, which is the CART default.

For the first model, candidate predictors were all 40 product and process metrics in Tables 1 and 2. We generated a series of trees using various values of c. The value $c = 5.0$ was too extreme for CART to generate a useful model. The CART parameters for prior probabilities were fixed at artificial values, $\pi'_{nfp} = 0.8$, and $\pi'_{fp} = 0.2$. CART's cost of a Type I error, C'_I, was set to one, and CART's cost of a Type II error, C'_{II}, was varied to achieve the desired values of c, according to Eq. (1). Table 3 lists the accuracy of models as a function of c. Type I and Type II misclassification rates for the *fit* data set are shown based on resubstitution and 10-fold cross-validation. The proportions of modules predicted to be *fault-prone* (Pred f-p) for cross-validation are also listed. Table 3 also shows corresponding Type I, Type II, and overall misclassification rates when applying the model to the

Table 3. Accuracy of Models

Candidate predictors: 40 product and process metrics

	fit data set Misclassification Rates					*test* data set Misclassification Rates		
	Resubstitution		Cross-validation					
c	Type I	Type II	Type I	Type II	Pred f-p	Type I	Type II	Overall
0.50	9.2%	23.0%	10.8%	59.3%	13.0%	11.0%	57.8%	14.5%
0.60	15.0%	24.4%	15.0%	51.1%	17.5%	15.9%	43.0%	17.9%
0.70	21.1%	27.4%	18.3%	44.4%	21.1%	21.3%	34.1%	22.2%
0.80	21.1%	27.4%	21.2%	39.3%	24.1%	21.3%	34.1%	22.2%
0.90	27.0%	17.8%	25.7%	28.9%	29.1%	27.7%	28.9%	27.8%
0.95	**25.8%**	**15.6%**	**26.4%**	**25.9%**	**29.9%**	**26.2%**	**28.9%**	**26.4%**
1.00	28.2%	10.4%	28.2%	24.1%	31.7%	28.7%	25.9%	28.5%
1.10	27.4%	12.6%	30.4%	21.5%	34.0%	28.0%	25.9%	27.8%
1.20	28.9%	7.4%	30.9%	23.7%	34.3%	29.2%	25.2%	28.9%
1.30	29.3%	5.9%	30.9%	23.7%	34.3%	29.6%	25.2%	29.3%
1.40	36.3%	15.6%	42.5%	14.8%	45.7%	36.4%	24.4%	35.5%
1.50	32.5%	5.9%	37.4%	17.0%	40.8%	32.9%	24.4%	32.3%
2.00	51.5%	4.4%	48.3%	13.3%	51.1%	52.0%	11.8%	49.0%
3.00	52.2%	1.5%	52.4%	11.9%	55.0%	53.7%	9.6%	50.4%
4.00	52.2%	1.5%	54.6%	10.4%	57.2%	53.7%	9.6%	50.4%
5.00	No tree generated							

test data set. Figure 2 depicts the cross-validation accuracy of this series of trees as a function of c, and Figure 3 shows the corresponding proportion of modules predicted to be *fault-prone*. We preferred $c = 0.95$, which is bold in Table 3.

In this case study, because the proportion of *fault-prone* modules was very small, we preferred the tree with approximately equal Type I and Type II misclassification rates using cross-validation. When a range of c values had essentially the same cross-validation accuracy, we preferred that resubstitution accuracy also be as balanced as possible for improved robustness. Other criteria might be appropriate for other situations.

Figure 4 depicts the preferred classification tree model. Note that only six metrics out of forty were used in the final tree: the number of distinct include files (*FILINCUQ*), the net number of new and changed lines of code (*SRC_MOD*), the maximum nesting level (*CTRNSTMX*), the number of updates in designers' company careers (*UPD_CAR*), the number of control statements (*STMCTL*), and deployment usage (*USAGE*). CART determined that other variables would not significantly improve the model.

For the second model, we similarly applied the CART algorithm to the *fit* data set to build a model based on the 25 product metrics listed in Table 1. Using the same criteria, we selected $c = 0.90$. Figure 1, in Section above, depicts the resulting classification tree. Note that only two metrics out of twenty-five were used in the final tree: the number of distinct include files (*FILINCUQ*) and the total span of variables (*VARSPNSM*).

We applied each model to the *test* data set to predict class membership of each

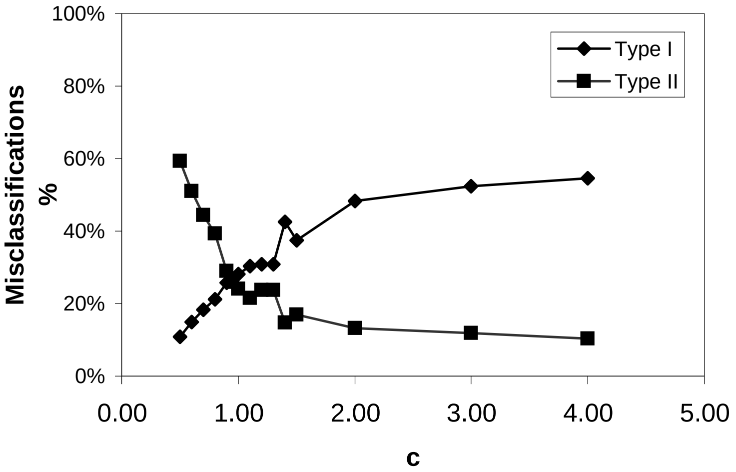

Fig. 2. 10-Fold Cross-Validation Accuracy

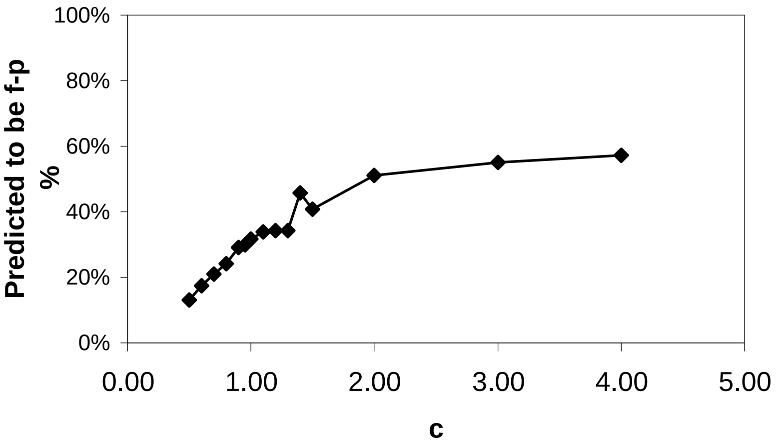

Fig. 3. Modules Predicted to be Fault-Prone

module. Table 4 shows the resulting estimated accuracy. The table indicates the data set on which the evaluation is based. We prefer the data-splitting estimate, because it simulated practical use of the model.

3.3. *Discussion*

Figure 2 illustrates the impact of the value of c on estimated model accuracy. The CART default is uniform priors ($\pi_{fp} = \pi_{nfp} = 0.5$) and equal costs ($C_I = C_{II}$), $c = 1$, and in this study, this was close to our preferred value. However, the preferred value of c is project-dependent and data-dependent. Therefore, one should carefully estimate misclassification rates as functions of c and choose a preferred value.

Our experiments found that extreme values of CART's prior probabilities, π'_{fp}

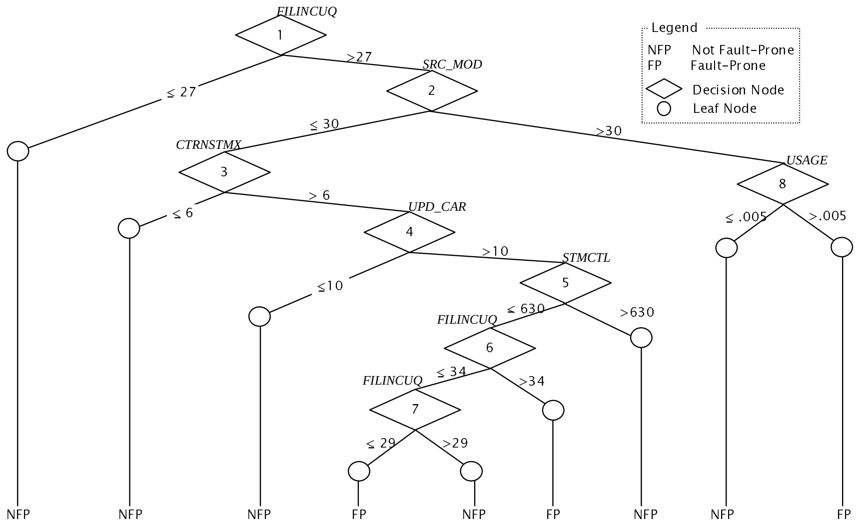

Fig. 4. Tree Based on Product and Process Metrics

Table 4. Accuracy of Models

		Misclass. rates	
	c	Type I	Type II
Product Metrics			
	0.90		
Resubstitution (*fit*)		26.7%	28.1%
Cross-validation (*fit*)		34.6%	25.2%
Data Splitting (*test*)		27.0%	27.4%
Product & Process Metrics			
	0.95		
Resubstitution (*fit*)		25.8%	15.6%
Cross-validation (*fit*)		26.4%	25.9%
Data Splitting (*test*)		26.2%	28.9%

and π'_{nfp}, (e.g., 0.1) prevented the algorithm from generating a tree. Therefore, we could not use the actual proportions of *fault-prone* and *not fault-prone* modules, because $\pi_{fp} = 0.0740$ was too small.

We found that the same tree is generated for constant c, irrespective of the values of the component ratios, as long as the ratios are not extreme. For example, based on Eq. (1), π'_{fp}/π'_{nfp}, can be a convenient constant and C'_{II}/C'_{I} can be varied to achieve a range of values for c. We used $\pi'_{fp}/\pi'_{nfp} = 0.2/0.8$ and $C'_{I} = 1$, varying C'_{II}.

There are several strategies for choosing c. One approach is to choose c so that the Type II misclassification rate is no more than a maximum acceptable level. Another approach sets a maximum on the number of modules predicted to be *fault-prone* as shown in Figure 3 above. In a third approach, if one chooses c such that Type I and Type II misclassification rates are equal, then both misclassification rates are effectively minimized [22]. This criterion for choosing c is especially appropriate when one class is a much smaller proportion of the population than the other. In practice, we can achieve only approximate equality due to finite discrete data sets. Here, we preferred approximately equal misclassification rates to illustrate our methodology. Many other strategies are also possible.

As shown in Table 4, we were surprised that a model based on only two product metrics out of twenty-five candidates had similar data-splitting accuracy as the model based on six significant metrics out of forty candidates. This was probably due to correlation among the variables. However, the cross-validation accuracy of the smaller model was worse, suggesting it may be less robust. The smaller model was easier to interpret, data collection was streamlined, and its two variables (*FILINCUQ* and *VARSPNSM*) can be collected earlier in the life cycle than some variables in the larger model (*SRC_MOD* and *UPD_CAR*). This illustrates that experimentation with the list of candidate metrics is advisable to achieve the preferred balance among parsimony (few variables), accuracy, robustness, and timeliness.

The number of distinct include files (*FILINCUQ*) was the most significant individual variable. It was the root decision node in all the tree models we generated. We also generated the classification tree where *FILINCUQ* was the only predictor. The model had Type I misclassification rate of 27.3% and a Type II rate of 39.2%, using data splitting. This was somewhat worse than the other two models. In general, we do not recommend single variable models, because they are often not robust.

The data splitting results in Table 4 indicated the level of accuracy to expect on a similar project or subsequent release. The tree models had sufficient accuracy to be useful to a software development project for targeting enhancement techniques. If model predictions are used to guide extra reviews or extra testing, the modules recommended for extra treatment would consist of those that are actually *not fault-prone* but are misclassified (Type I), plus those correctly classified as *fault-prone*. For example, the product-metrics model would recommend that $30.4\% = (0.270)(0.926) + (1 - 0.274)(0.074)$ of the modules be given special treatment to discover faults early. Extra reviews would be wasted on the *not fault-prone* modules that were mistakenly recommended. However, effective reviews would discover $72.6\% = 1 - 0.274$ of the *fault-prone* modules early, so that faults could be corrected before release. If we selected the same number of modules at random for extra reviews, then only 30.4% of the *fault-prone* modules would be covered. The product and process metrics-based model was similar. Using such models to select modules for review would have a substantial benefit [23], because avoiding customer-discovered faults is extremely valuable.

4. Conclusions

High software reliability is required of high-assurance systems. We focus on software quality models that make timely predictions of quality indicators on a module-by-module basis. Our objective is to target enhancement techniques early in development to those modules that will benefit the most. For example, Enhanced Measurement for Early Risk Assessment of Latent Defects (EMERALD) is a decision-support system at Nortel, which provides software quality models that predict which modules are likely to be fault-prone, based on available software measurements.

This paper introduces the Classification And Regression Trees, CART, to practitioners in high-assurance systems engineering. This paper discusses the following practical lessons-learned on building classification trees for software quality modeling:

- Extreme prior probabilities can prevent CART from building a tree.
- A project should control the balance between misclassification rates. In this paper, we implemented this control by choosing a constant c, defined as the the ratio of prior probabilities times the ratio of misclassification costs. These parameters control the goodness-of-split criterion when CART generates a tree.
- Minimize the number of independent variables without significantly sacrificing accuracy. However, we do not advocate models with a single independent variable.
- Building a tree is vulnerable to overfitting, and consequently, the accuracy of resubstitution of the *fit* data set into the model can be misleading.

A case study of a very large telecommunications system used CART to build software quality models based on various sets of software product and process metrics. The models predicted whether or not modules were fault-prone. We found that a model based on two significant software product metrics out of twenty-five candidates had comparable accuracy to a model based on six significant product and process metrics selected out of a set of forty metrics. This illustrates the way that tree algorithms can build successful parsimonious models. The results of this paper are being refined and further evaluated for the EMERALD system.

Future research will compare several classification tree algorithms with each other and with other classification techniques. In particular, methods for balancing misclassification rates will be explored further. Future research will also evaluate the sensitivity of resulting tree structures to various conditions. When data on additional releases becomes available, further research will seek to confirm the results presented here.

Acknowledgements

We thank the EMERALD team at Nortel for collecting the data. 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, Research Triangle Park, NC, 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. Michael R. Lyu. Introduction. In Michael R. Lyu, editor, *Handbook of Software Reliability Engineering*, chapter 1, pages 3–25. McGraw-Hill, New York, 1996.
2. John P. Hudepohl, Stephen J. Aud, Taghi M. Khoshgoftaar, Edward B. Allen, and Jean Mayrand. EMERALD: Software metrics and models on the desktop. *IEEE Software*, 13(5):56–60, September 1996.
3. Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. *Classification and Regression Trees*. Chapman & Hall, London, 1984.
4. Taghi M. Khoshgoftaar, Edward B. Allen, Lofton A. Bullard, Robert Halstead, and Gary 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, October 1996. IEEE Computer Society.
5. Adam A. Porter and Richard W. Selby. Empirically guided software development using metric-based classification trees. *IEEE Software*, 7(2):46–54, March 1990.
6. Ryouei Takahashi, Yoichi Muraoka, and Yukihiro 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, November 1997. IEEE Computer Society.
7. Joel Troster and Jeff Tian. Measurement and defect modeling for a legacy software system. *Annals of Software Engineering*, 1:95–118, 1995.
8. Taghi M. Khoshgoftaar, Edward B. Allen, Kalai S. Kalaichelvan, and Nishith Goel. Early quality prediction: A case study in telecommunications. *IEEE Software*, 13(1):65–71, January 1996.
9. Normal F. Schneidewind. Software metrics validation: Space Shuttle flight software example. *Annals of Software Engineering*, 1:287–309, 1995.
10. Victor R. Basili, Lionel C. Briand, and Walcélio Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, October 1996.
11. Lionel C. Briand, Victor R. Basili, and William M. Thomas. A pattern recognition approach for software engineering data analysis. *IEEE Transactions on Software Engineering*, 18(11):931–942, November 1992.
12. Taghi M. Khoshgoftaar and David 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, April 1995.
13. Christof Ebert. Classification techniques for metric-based software development. *Software Quality Journal*, 5(4):255–272, December 1996.
14. Taghi M. Khoshgoftaar, Edward B. Allen, Archana Naik, Wendell D. Jones, and John 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, November 1998. IEEE Computer Society.
15. Taghi M. Khoshgoftaar, Edward B. Allen, Archana Naik, Wendell D. Jones, and John P. Hudepohl. Modeling software quality with classification trees. In Hoang Pham and Ming-Wei Lu, editors, *Proceedings of the Fourth ISSAT International Conference on Reliability and Quality in Design*, pages 178–182, Seattle WA, August 1998. International Society of Science and Applied Technologies.
16. Dan Steinberg and Phillip Colla. *CART: A supplementary modules for SYSTAT*. Salford Systems, San Diego, CA, 1995.
17. Bradley 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.
18. Swapna S. Gokhale and Michael R. Lyu. Regression tree modeling for the prediction of

software quality. In Hoang Pham, editor, *Proceedings of the Third ISSAT International Conference on Reliability and Quality in Design*, pages 31–36, Anaheim, CA, March 1997. International Society of Science and Applied Technologies.

19. Peter A. Lachenbruch and M. Ray Mickey. Estimation of error rates in discriminant analysis. *Technometrics*, 10(1):1–11, February 1968.
20. Taghi M. Khoshgoftaar and Edward B. Allen. A practical classification rule for software quality models. Technical Report TR-CSE-97-56, Florida Atlantic University, Boca Raton, FL, August 1997.
21. Jean Mayrand and François Coallier. System acquisition based on software product assessment. In *Proceedings of the Eighteenth International Conference on Software Engineering*, pages 210–219, Berlin, March 1996. IEEE Computer Society.
22. G. A. F. Seber. *Multivariate Observations.* John Wiley and Sons, New York, 1984.
23. Taghi M. Khoshgoftaar, Edward B. Allen, Wendell D. Jones, and John P. Hudepohl. Return on investment of software quality models. In *Proceedings 1998 IEEE Workshop on Application-Specific Software Engineering and Technology*, pages 145–150, Richardson, TX USA, March 1998. IEEE Computer Society.