Predicting Fault-Prone Modules In Embedded Systems Using Analogy Based Classification Models*

Taghi M. Khoshgoftaar†
Bojan Cukic
Naeem Seliya

Abstract

Embedded systems have become ubiquitous and essential entities in our ever growing high-tech world. The backbone of today's information-highway infrastructure are embedded systems such as telecommunication systems. They demand high reliability, so as to prevent severe consequences of failures including costly repairs at remote sites. Technology changes mandate that embedded systems evolve, resulting in a demand for techniques for improving reliability of their future system releases. Reliability models based on software metrics can be effective tools for software engineering of embedded systems, because quality improvements are so resource consuming that it is not feasible to apply them to all modules. Identification of the likely fault-prone modules before system testing, can be effective in reducing the likelihood of faults discovered during operations.

A software quality classification model is calibrated using software metrics from a past release, and is then applied to modules currently under development to estimate which modules are likely to be fault-prone. This paper presents and demonstrates an effective case-based reasoning approach for calibrating such classification models. It is attractive for software engineering of embedded systems, because it can be used to develop software reliability models using a faster, cheaper, and easier method. We illustrate our approach with two large-scale case studies obtained from embedded systems. They involve data collected from telecommunication systems including wireless systems. It is indicated that the level of classification accuracy observed in both case studies would be beneficial in achieving high software reliability of subsequent releases of the embedded systems.

Keywords: high assurance, embedded systems, software reliability, case-based reasoning, classification models, software metrics

---

*For review only. Do not distribute.

†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.

1 Introduction

Embedded software systems have become ubiquitous and essential entities in today's technology-based world. The backbone of the modern world's information-highway infrastructure are embedded systems, such as telecommunication systems. Many mission-critical systems [22] such as military tactical systems, have powerful embedded computers. Success of such high-assurance embedded systems is very critical and depends on high reliability, i.e., failure free operations of these systems, so as to prevent severe consequences of failures including costly repairs at remote sites and expensive down-time periods. High security and reliability mandates for embedded systems often result in constant updates and multiple releases of the underlying embedded software. Logic would dictate that subsequent releases of embedded systems must be more reliable than their respective predecessors.

Effective techniques and methods for improving reliability of software systems are being developed to aid in achieving high system reliability. This is because quality enhancement activities are so expensive that it is practically not feasible to apply them to all modules. For example, specialized modeling, verification and validation, extra reviews, extra testing, and re-engineering can be expensive, and yet, may be beneficial only to high-risk modules. Early (pre-testing phases) identification [11] of *fault-prone* modules during the development life-cycle is effective in reducing the likelihood of faults being discovered during system operations.

Reliability models based on software metrics [23] can be efficient tools for software engineering of embedded systems, because they can assist in reducing development and maintenance costs [2]. Software product metrics can quantify the size and complexity of software artifacts in many ways. Software process metrics, such as inspection metrics, can capture significant features of a module's history as it evolves. Software execution metrics quantify opportunities for faults to be detected during operations. A software reliability model [19] can be developed using measurements and fault data from a past release. A fault is a defect in an executable product that causes a software failure. The calibrated model can then be applied to modules currently under development, yielding a quality prediction on a module-by-module basis.

Software quality classification models can be used as reliability models for embedded systems, by identifying fault-prone (or not fault-prone) modules early in the life-cycle. Various classification techniques have been proposed and investigated, such as discriminant analysis [11, 20], optimized set reduction [1], regression and classification trees [9], artificial neural networks [14], fuzzy logic [3, 4], and logistic regression [7]. This paper examines case-based reasoning (CBR) [16, 21], an automated reasoning approach which attempts to resolve problems of a current case based on instances of past cases, which are stored in a "case library". CBR systems have demonstrated important applications in numerous fields, including software cost estimation, software reuse, software quality estimation [5, 17], and software design. In comparison to other software reliability modeling techniques, CBR has several advantages, including:

- CBR systems can be designed to alert users when a new case is outside the bounds of current experience. This is attractive when a solution of "I don't know" is better than a guess. In contrast, a typical prediction model always gives some kind of decisive solution, even in extreme situations.

- Cases can be added or deleted as new information becomes available, without the hassle of model re-estimation to track new information.

- CBR is scalable to very large case libraries with fast retrieval, even as the case library scales up.

- Users of CBR systems can be easily convinced that the solution was derived in a reasonable way, i.e., they are not "black boxes", and hence, the CBR system lends itself to user acceptance.

- CBR systems are attractive, because they are modeled with the premise of human intuition in mind. It relies on learned experiences of the past to analyze new issues.

We present and demonstrate an effective CBR software quality classification modeling technique [17, 21]. A high assurance embedded systems such as telecommunication systems, often have very few fault-prone modules. Due to this low proportion of fault-prone modules, software quality classification modeling methods in literature, such as [13] are inappropriate for such systems.

The software quality classification method illustrated in this paper, is attractive for software engineering of embedded systems, because it can be used to develop useful software reliability models using a faster, cheaper, and easier method. A CBR classification model provides a fast prediction retrieval, without having to re-calibrate every time new modules are added to the case library. This feature makes the CBR software quality modeling process faster, and unlike other classification techniques, extra resources needed for re-calibration, if any, are minimal.

Useful and effective software quality classification models can be built with CBR using few primitive metrics as well as many complex metrics. Data collection efforts can be minimized by obtaining only few relevant metrics data for modeling purposes. CBR classification models are easy to understand and interpret, because the model is based on the premise of human intuition in mind, and hence does not need high technical expertise for model interpretation and application.

Classification models using the CBR approach were designed for two large-scale case studies of software measurement and fault data obtained from embedded software systems. The first case study, denoted as LLTS, involves software metrics collected over four historical releases of a very large legacy telecommunications system. The first release, i.e., Release 1 is used to *train* and *calibrate* the classification model, whereas the subsequent releases, i.e., Releases 2, 3, and 4, are used to *evaluate* the *classification accuracy* of the calibrated model.

The second case study, denoted as WLTS, was performed to further validate the usefulness of CBR as a software quality modeling technique, especially in embedded systems. It involves software metrics data obtained from initial releases of two large Windows-based embedded system applications used primarily for customizing the configuration of wireless telecommunications products. An impartial data splitting technique [17] is employed to obtain the *training* and *evaluation* data sets.

The working hypothesis of our modeling approach, when applied to software quality estimation, is that a module currently under development will probably be fault-prone if previously developed modules with similar *attributes* were fault-prone. Classification accuracy of a model is measured in terms of its *Type I* and *Type II* misclassification error rates. A Type I error occurs when a not fault-prone module is classified as fault-prone, whereas a Type II error occurs when a fault-prone module is classified as not fault-prone.

An analyst seeks a classification model that has the minimum expected cost of misclassifications. The actual costs of rectifying the two misclassifications are not known until quite late in the life-cycle. A Type II error is relatively more expensive than a Type I error, because it may involve repairs to embedded systems at remote sites, and costly down-time periods during operations. In contrast, Type I errors may involve costs of extra reviews of the falsely identified not fault-prone modules. Consequently, a classification model must provide an appropriate simulation that reflects an effort of minimizing the two costs.

The costs of the two misclassifications are generally not the same, hence the accuracy and usefulness of a classification model is affected by the *cost ratio*, $\frac{C_I}{C_{II}}$, where C_I is the cost of rectifying a Type I misclassification, and C_{II} is the cost of rectifying a Type II misclassification. Factors besides cost ratio may determine the preferred balance between the misclassification rates. For example, in high assurance embedded systems, where the proportion of fault-prone modules is very small and Type II errors have more severe consequences, one may prefer equal misclassification rates. In this paper, we present a practical generalized classification rule [21] in the context of CBR software quality classification modeling that allows appropriate emphasis on each type of misclassification according to the needs of the project. A similar classification rule was proposed by our research team [8] in the context of classification tree based software quality estimation models [9].

The results of our case studies indicate that models designed using the illustrated case-based reasoning approach, yielded useful classification accuracy that can be used for software reliability control of their respective embedded systems. The calibrated models can be applied to modules currently under development, thereby assisting in obtaining high software reliability of the subsequent system releases. A similar approach of model calibration and application, can be universally adopted for software engineering of other embedded systems.

The layout of the rest of the paper is as follows. In Section 2 we present the theoretical details of the classification technique adopted for our case-based reasoning approach. Section 3 discusses the details and procedures of the experiments performed. The system descriptions and results of the two case studies examined in this paper are presented in Sections 4 and 5, respectively. Finally, we conclude with inferences based on our case studies with suggestions for related future work in Section 6.

2 Analogy-Based Classification Models

The past experience(s) of a CBR classification system is represented by "cases" in a "case library". The case library and the associated retrieval and decision rules constitute a CBR model [16]. The past instances or cases are well-known project data from previously developed systems or projects, and contain all relevant information pertaining to each case. When associating with software quality modeling based on software metrics, a case in the case library is composed of a set of predictors or *independent* variables ($\mathbf{x}_i$), and a response or *dependent* variable (y_i).

In our study the response variable is a *class membership*, i.e., either *fault-prone* or *not fault-prone*. The definition of whether a module is *fault-prone* or *not fault-prone* depends on the pre-set threshold value of the quality factor, which in our studies is the number of faults. Consequently, if the number of faults is fewer than the threshold value, the module is deemed as not fault-prone, and fault-prone otherwise. Suppose each case in the library has known attributes and class membership. Then, given a case with unknown class, we predict its class to be the same as the class of the most *similar* case(s) in the case library, where similarity is defined in terms of the case attributes.

A CBR classification model uses a *similarity function* to determine the most similar cases to the current case, from the case library (*fit* or *training* data). The function computes the distance d_{ij}, between the current case $\mathbf{x}_i$, and every other case $\mathbf{c}_j$ in the case library. The cases with the smallest possible distances are of primary interest, and the set of similar cases forms the set of *nearest neighbors*, N. Model parameter $\mathbf{n}_N$, represents the number of the best (most similar to current case) cases selected from N for case analysis and class estimation. $\mathbf{n}_N$ can be varied during model calibration to obtain different classification models. Once $\mathbf{n}_N$ is selected, a classification technique is used to classify the current case as either fault-prone or not fault-prone.

There are several types of similarity functions available including, *City Block distance*, *Euclidean distance*, and *Mahalonobis distance.* Our previous research [21], indicated that the *Mahalonobis* distance similarity function yielded better classification accuracy than the other two functions. Consequently, we use this distance function in our case studies. The *Mahalonobis* distance similarity function is given:

$$d_{ij} = (\mathbf{c}_j - \mathbf{x}_i)' S^{-1} (\mathbf{c}_j - \mathbf{x}_i) \tag{1}$$

where, S is the variance-covariance matrix of the independent variables for the case library and S^{-1} is its inverse. Prime ($'$) indicates the transpose. Unlike the *Euclidean* and *City Block* distance functions, the *Mahalonobis* distance function explicitly accounts for the correlation [12] among the attributes, and does not require *standardization* or *normalization* of the independent variables [17, 21].

A classification model may be sensitive to the ratio of the costs of the Type I and

Type II misclassifications. This is because in practice actual costs of the two misclassifications (C_I and C_{II}) are not known during the modeling period. Practically speaking, the costs of the two misclassifications are not the same. Hence, the use of equal costs for C_I and C_{II} during model calibration is not realistic. Moreover, other factors besides cost ratio may determine the best balance between the Type I and Type II error rates. For example, when the proportion of fault-prone modules (as compared to not fault-prone modules) is very small and Type II misclassifications have much more severe consequences than Type I, one may prefer equal misclassification rates. The cost ratio, $\frac{C_I}{C_{II}}$, can be varied during modeling to obtain a preferred balance [15] between the misclassification error rates.

In the context of CBR software classification models, we proposed [21] two methods of classification, i.e., *majority voting* and *data clustering*. It was indicated that the *data clustering* technique yielded better classification accuracy. Consequently, we use it as the classification technique for our case studies. In the *data clustering* method, the case library is partitioned into two clusters, fault-prone (fp) and not fault-prone (nfp), according to the class of each case. For a currently unclassified case $\mathbf{x}_i$, let $d_{nfp}(\mathbf{x}_i)$ be the average distance to the not fault-prone nearest neighbor cases, and $d_{fp}(\mathbf{x}_i)$ be the average distance to the fault-prone nearest neighbor cases. The number of nearest neighbor cases to be used for analysis, can be varied (as a model parameter) during modeling.

Once the average distances to the nearest neighbor cases are computed, our *proposed* generalized classification rule for *data clustering* is then used to estimate the class,

$Class(\mathbf{x}_i)$, of the unclassified case. The classification rule is given by:

$$Class(\mathbf{x}_i) = \begin{cases} nfp & \text{If } \frac{d_{fp}(\mathbf{x}_i)}{d_{nfp}(\mathbf{x}_i)} > \frac{C_I}{C_{II}} \\ fp & \text{otherwise} \end{cases} \tag{2}$$

A CBR software quality classification model therefore, consists of the following: $\frac{C_I}{C_{II}}$, the modeling cost ratio; $\mathbf{n}_N$, the number of nearest neighbor cases for analysis and class estimation; a *similarity function*, such as Mahalonobis distance function and, a *classification method*, such as data clustering.

3 Methodology

The generalized modeling methodology adopted for both the LLTS and WLTS case studies is presented briefly in this section. Further details and discussions of the experiments are presented in [17, 21]. Classification modeling studies using CBR were performed with SMART, the Software Measurement Analysis and Reliability Toolkit [10]. SMART is an empirical software quality modeling tool that has been developed at the Empirical Software Engineering Research Laboratory, Florida Atlantic University.

The algorithm for model calibration and selection used during our empirical investigations is summarized in Figure 1. The modeling methodology used for the two case studies is presented below.

1. Case library: A fit (or training) data set is selected as the case library. For the LLTS case study, Release 1 is used as the training data set. In the case of the WLTS study, an impartial data splitting is performed to obtain the fit and test data sets.

2. Target data: Release 2, 3, and 4 are used as test or evaluation data sets for the

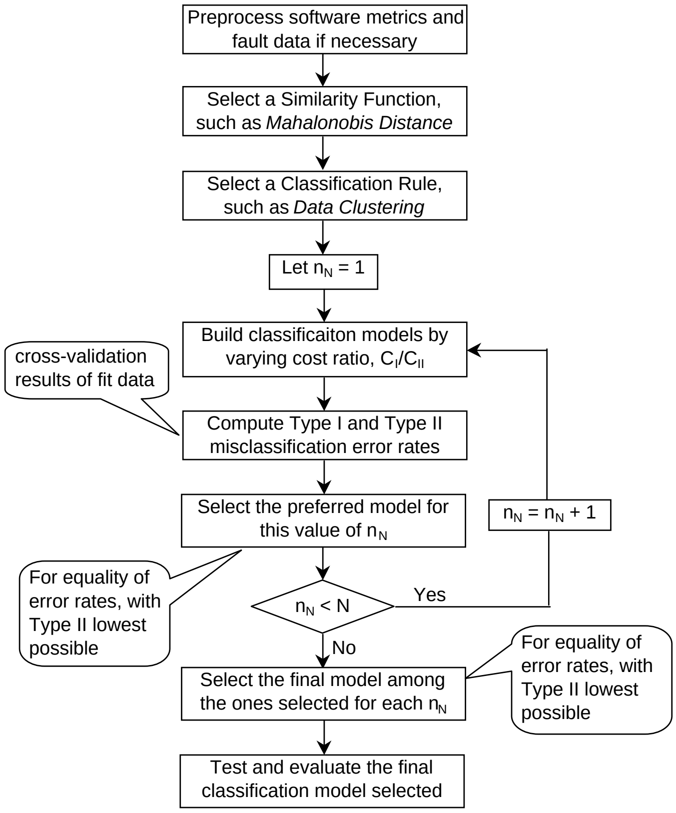

Figure 1: Model Calibration and Selection Flowchart

LLTS case study. For the WLTS case study the appropriate test data set obtained in the above step is used.

3. Classification technique: The Mahalonobis distance similarity function together with the data clustering classification method is used for both case studies.

4. Parameter $\mathbf{n}_N$: Values, with increments of 1, starting from 1 and as high as 100 were considered for both case studies. Other higher values were considered, but did not yield better results.

5. Parameter $\frac{C_I}{C_{II}}$: Cost ratios ranging from as low as 0.0001 to as high as 5.0 were

considered. Other values were also used during modeling, but did not yield better results. These cost ratios are used for modeling purposes, and should not be confused with the actual cost ratios.

6. Build models: For each $\mathbf{n}_N$ value, the cost ratios were varied to build different classification models. The Type I and Type II misclassification rates for the training data sets are computed using a *v-fold cross-validation* technique, where v is the number of iterations used for model building and evaluation. For our case studies, v is the number of cases in the case library (fit data). In our approach, at each iteration, one observation is removed from the training data set and is used for evaluation of the model built using the $(v - 1)$ observations. The Type I and Type II misclassification rates for the fit data set are then summarized across all v iterations.

7. Select final model: A preferred balance of equality between the misclassification rates is desired as per the requirements of the two case studies. Among the models built using the different combinations of $\mathbf{n}_N$ and $\frac{C_I}{C_{II}}$, a model with a good balance between the Type I and Type II errors with Type II as low as possible, is selected as the final model. These error values are based on the cross-validation results of the fit data set.

8. Evaluate model: The test data sets are used to validate the accuracy of the models calibrated using the fit data set. Type I and Type II errors are computed for the test data sets, and are then compared with those of the fit data set. A stable classification model exhibits a small error-rate variation (absolute value) across its training and evaluation data sets.

4 A Large Legacy Telecommunications System

4.1 System Description

The data for this case study (denoted as LLTS) was collected over four historical releases from a very large legacy embedded software system written in a high-level language, using the procedural paradigm, and maintained by professional programmers in a large organization. We label the releases 1 through 4. This telecommunications system had over ten million lines of code and included numerous finite-state machines and interfaces to various kinds of equipment.

A software module was considered as a set of related source-code files. A module was considered *fault-prone* if any faults were discovered during operations, and *not fault-prone* otherwise. Faults in deployed embedded systems are extremely expensive because, in addition to down-time due to failures, visits to remote sites are usually necessary to repair them. Fault data, collected at the module-level by the problem reporting system, comprised of faults discovered during post unit testing phases. Post unit testing phases recorded faults that were discovered before and after the product was released to customers.

There were too few faults in unchanged modules for effective statistical reliability modeling. Approximately 99% of the unchanged modules had no faults. Consequently, this case study considered modules that were new or had at least one source code update since the prior release. Configuration management data analysis identified software modules that were unchanged from the prior release.

The system had several million lines of code in a few thousand modules per release. Each release has approximately 3500 to 4500 updated software modules. The number of

Table 1: LLTS: Software Product Metrics

Symbol	Description
	Call Graph Metrics
CALUNQ	Number of distinct procedure calls to others.
CAL2	Number of second and following calls to others. $CAL2 = CAL - CALUNQ$ where CAL is the total number of calls.
	Control Flow Graph Metrics
CNDNOT	Number of arcs that are not conditional arcs.
IFTH	Number of 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.

modules considered in Release 1, 2, 3, and 4 were 3649, 3981, 3541, and 3978 respectively. The proportion of modules with no faults among the updated modules of the first release was $\pi_{nfp} = 0.937$, and the proportion with at least one fault was $\pi_{fp} = 0.063$. Such a small set is often difficult for a reliability modeling technique to identify.

The set of available software metrics is usually determined by pragmatic considerations. A data mining approach is preferred in exploiting software metrics data [4], by which a broad set of metrics are analyzed rather than limiting data collection according to a predetermined set of research questions.

Data collection for this case study involved extracting source code from the configu-

Table 2: LLTS: Software Process Metrics

Symbol	Description
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.
UNQ_DES	Number of different designers making changes.
VLO_UPD	Number of updates to this module by designers who had 10 or less total updates in entire company career.
LO_UPD	Number of updates to this module by designers who had between 11 and 20 total updates in entire company career.
UPD_CAR	Number of updates that designers had in their company careers.

Table 3: LLTS: Software Execution Metrics

Symbol	Description
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.

ration management system. Measurements were recorded using the EMERALD (Enhanced Measurement for Early Risk Assessment of Latent Defects) software metrics analysis tool, which includes software-measurement facilities and software quality models [6]. Preliminary data analysis selected metrics (aggregated to the module level) that were appropriate for our modeling purposes. The software metrics considered included 24 product metrics, 14 process metrics, and 4 execution metrics. Consequently, this case study consists of 42 independent variables that are used to predict the response variable, i.e., *Class*: *fp* or *nfp*.

The software product metrics in Table 1 are based on call graph, control flow graph, and statement metrics. The number of procedure calls by each module (*CALUNQ* and

CAL2) are derived from a call graph depicting calling relationships among procedures. A module's control flow graph, consists of nodes and arcs depicting the flow of control of the program. Statement metrics are measurements of the program statements without expressing the meaning or logic of the statements.

Process metrics in Table 2 may be associated with either the likelihood of inserting a fault during development, or the likelihood of discovering and fixing a fault prior to product release. The configuration management systems tracked each change to source code files, including identity of the designer and the reason of the change, e.g., a change to fix a problem or to implement a new requirement. The problem reporting system maintained records on past problems. Execution metrics listed in Table 3 are associated with the likelihood of executing a module, i.e., operational use. The proportion of installations that had a module, *USAGE*, was approximated by deployment data on a prior release. Execution times were measured in a laboratory setting with different simulated workloads.

4.2 Empirical Results

The case library for this case study consisted of modules from the Release 1 data set. If the modules in the training and evaluation data sets had more than one fault they were classified as fault-prone, and not fault-prone otherwise. This threshold value was determined as per the modeling requirements of the embedded system. The faults associated with modules, were those discovered by customers during system operations.

The aim of calibrating classification models was to obtain a model with a good balance (for equality) between the Type I and Type II errors, with Type II being as low

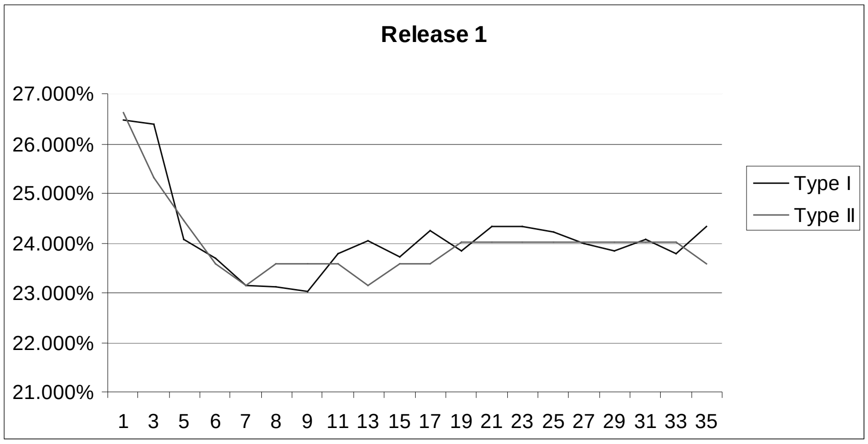

Figure 2: LLTS: Misclassification Rates vs. $\mathbf{n}_N$

as possible. It was observed that for a given cost ratio, the Type I and Type II error rates showed lower variation as $\mathbf{n}_N$ increased. Figure 2 demonstrates this observation. The figure represents the variation for our selected classification model. Other values of $\mathbf{n}_N$ than those shown in Figure 2 were considered. However, they did not show any improvements in the results and are hence, not presented.

The variation of misclassification rates (for a given $\mathbf{n}_N$) with respect to the cost ratio, is demonstrated in Table 4. For a given $\mathbf{n}_N$, as the cost ratios are increased, the Type I error decreases, whereas the Type II error increases. Subsequently, at a particular cost ratio the errors are the most balanced, and that model is selected as the preferred model for the given $\mathbf{n}_N$.

The values shown in Table 4 are for $\mathbf{n}_N = 7$, and the row highlighted in bold is the model with the preferred balance between the Type I and Type II errors. The model with $\frac{C_I}{C_{II}} = 0.95$ represents the value for our selected classification model. Other cost ratio values besides those shown in Table 4 were considered. However, they did not yield better results and hence, are not presented. It should be noted that these cost ratios

Table 4: LLTS: Misclassification Rates vs. $\frac{C_I}{C_{II}}$

Cost Ratio	Type I Error	Type II Error
0.10	90.175%	0.000%
0.20	81.287%	0.873%
0.30	73.655%	1.310%
0.40	66.257%	2.620%
0.50	58.041%	4.803%
0.60	50.760%	8.734%
0.70	42.310%	12.227%
0.80	34.181%	17.904%
0.90	26.754%	21.834%
0.91	26.053%	22.271%
0.92	25.322%	22.271%
0.93	24.561%	22.271%
0.94	23.860%	22.271%
0.95	**23.158%**	**23.144%**
0.96	22.427%	24.454%
0.97	21.988%	24.891%
0.98	21.579%	25.328%
0.99	20.936%	26.201%
1.00	20.526%	27.948%

Table 5: LLTS Model: $\mathbf{n}_N = 7$, $\frac{C_I}{C_{II}} = 0.95$

Data Set	Type I Error	Type II Error
Release 1	23.158%	23.144%
Release 2	25.132%	26.984%
Release 3	28.792%	27.660%
Release 4	28.667%	25.000%

are modeling cost ratios, and not the actual cost ratios for the embedded system. As mentioned earlier, the actual costs of misclassifications are unknown until very late in the system life-cycle.

The selected model and its classification accuracy across the multiple releases of the

LLTS case study are presented in Table 5. This model demonstrated the best balance of the two error rates. Classification models are susceptible to over-fitting [18], in which the model performs significantly better for the training data set as compared to the evaluation data sets. The LLTS model does not show excessive over-fitting, because the maximum difference in the Type I and Type II releases across the different system releases is only about 5% and 4%, respectively.

The model also reflects good stability, in terms of maintaining the preferred balance between the Type I and Type II error rates, across the different releases. Thus, the classification model calibrated in this study using the illustrated CBR approach, can successfully be used for software engineering and reliability control of subsequent releases of the telecommunication system.

5 Wireless Configurations Systems

5.1 System Description

This case study (denoted as WLTS) involved data collection efforts [17] from initial releases of two large Windows-based embedded system applications used primarily for customizing the configuration of wireless telecommunications products. The two embedded applications (written in C++) provide similar functionalities, and contain common source code. The main difference between them is the type of wireless product that each supports. Both systems comprised of over 1400 source code files and contained more than 27 million lines of code each.

Software inspection metrics were obtained by observing the configuration manage-

Table 6: WLTS: Product and Process Metrics

Product Metrics	Description
BASE_LOC	Number of lines of code for the source file version prior to the coding phase, i.e, auto-generated code.
SYST_LOC	Number of lines of code for the source file version delivered to system tests.
BASE_COM	Number of lines of commented code for source file version prior to coding phase, i.e, auto-generated code.
SYST_COM	Number of lines of commented code for source file version delivered to system tests.
Process Metrics	Description
INSP	Number of times the source file was inspected prior to system tests.

ment systems of the applications. The problem reporting system tracked and recorded problem statuses. Information such as, how many times a source file was inspected prior to system tests, were logged in its database. Software metrics obtained reflected aspects of source files, and consequently, a module in this case study comprised of a source file. Many software metrics were collected to record information such as, fault severity, inspection time, and major/minor errors. However, only a few primitive metrics were selected for modeling purposes since they provided the most relevant and concise information pertaining to the project. Further details of the data collection and metrics selection processes are presented in [17].

The fault data collected for this case study, represent the faults discovered during system tests. Upon preprocessing and cleaning the collected data, i.e., removal of outliers and illogical data points, 1211 modules remained. Over 66% of modules (809) were observed to have no faults, and the remaining 402 modules had at least 1 or more faults. The *five* software metrics used for reliability modeling for this case study are presented in Table 6. The product metrics used are statement metrics for the source files. They primarily indicated the number of lines of source code prior to the coding phase (i.e., auto-generated code) and just before system tests. The inspection metric, *INSP*, was obtained from problem reporting systems of the two embedded applications.

Classification models are dependent on the chosen threshold value that identifies modules as fault-prone or not fault-prone. The main stimulus for selecting the appropriate threshold value is to build the most useful and system-relevant software quality model possible. If the threshold of number of faults is set too low, all modules may be classified as fault-prone, whereas if it is set too high, all modules may be classified as not fault-prone. The consideration of different thresholds was part of a CBR classification model-sensitivity study performed for the WLTS case study.

Since data from subsequent releases was not available, an impartial data splitting was performed on the data set in order to obtain the *fit* and *test* data sets. Consequently, the *fit* and *test* data sets had 807 and 404 observations, respectively. In order to avoid biased results due to a lucky split, the original data set was randomly split 50 times, i.e., fifty pairs of the *fit* and *test* data sets were obtained. Empirical studies were then performed on all 50 split combinations.

5.2 Empirical Results

Prior to adopting a particular software quality classification technique, it would be greatly beneficial to an embedded system development team, if they knew how sensitive a particular classification technique is with respect to the chosen threshold value. It should be noted that the best threshold value may be dependent on the requirements of the

Table 7: WLTS Classification Models: Threshold 1

Split	n_N	$\frac{C_I}{C_{II}}$	Fit Data Set		Test Data Set	
			Type I	Type II	Type I	Type II
1	4	0.6	16.14%	16.79%	14.44%	13.43%
2	1	0.5	14.47%	14.93%	18.52%	15.67%
3	1	0.6	16.51%	16.79%	17.41%	14.93%
4	1	0.6	16.51%	16.79%	17.41%	14.93%
5	2	0.6	13.57%	13.38%	14.39%	22.56%
6	6	0.6	16.33%	16.79%	18.52%	20.15%
7	3	0.5	15.77%	14.18%	14.44%	14.18%
8	9	0.6	17.22%	16.11%	17.47%	18.52%
9	2	0.6	15.58%	13.81%	18.89%	14.93%
10	2	0.55	14.60%	15.04%	15.30%	18.38%
11	3	0.6	15.86%	15.87%	19.41%	12.98%
12	5	0.6	16.24%	15.85%	15.36%	21.17%
13	1	0.55	17.07%	16.05%	9.63%	13.43%
14	1	0.55	15.80%	15.24%	17.71%	18.05%
15	2	0.6	15.86%	16.24%	16.48%	14.50%
16	2	0.55	16.27%	16.54%	12.31%	12.50%
17	1	0.55	16.36%	16.35%	14.34%	20.14%
18	1	0.55	17.25%	16.05%	11.48%	14.18%
19	2	0.55	14.31%	14.13%	18.08%	18.05%
20	1	0.55	16.79%	15.87%	16.12%	18.32%
21	1	0.55	16.57%	15.56%	13.97%	14.39%
22	3	0.6	16.57%	15.56%	11.11%	20.98%
23	1	0.5	15.71%	16.17%	15.67%	11.77%
24	1	0.55	15.46%	14.07%	15.44%	21.97%
25	1	0.6	17.07%	17.15%	13.41%	13.28%
26	2	0.5	13.92%	13.41%	16.73%	15.60%
27	5	0.65	16.30%	15.33%	12.93%	24.82%
28	2	0.55	15.77%	14.93%	15.56%	13.43%
29	3	0.55	16.36%	14.13%	20.30%	12.78%
30	1	0.5	16.24%	15.47%	17.23%	14.60%
31	1	0.5	16.24%	15.47%	21.72%	18.25%
32	2	0.6	14.60%	15.04%	12.31%	22.06%
33	2	0.55	16.61%	15.47%	16.61%	15.47%
34	2	0.6	15.95%	14.60%	12.68%	19.53%
35	1	0.55	16.24%	16.60%	13.70%	12.69%
36	1	0.55	14.63%	15.73%	13.38%	16.30%
37	4	0.55	15.75%	15.71%	20.91%	19.15%
38	2	0.6	15.95%	15.69%	13.77%	14.84%
39	1	0.6	16.57%	17.41%	18.38%	13.64%
40	3	0.55	17.64%	16.79%	19.12%	12.88%
41	3	0.6	18.11%	16.25%	14.34%	11.20%
42	5	0.55	16.06%	16.22%	18.39%	18.88%
43	2	0.55	15.57%	14.96%	15.22%	17.97%
44	1	0.5	15.31%	15.09%	14.23%	15.33%
45	5	0.6	14.95%	15.09%	17.23%	16.79%
46	2	0.6	15.61%	15.24%	15.87%	19.55%
47	2	0.6	16.36%	15.59%	13.21%	17.27%
48	2	0.6	15.95%	15.69%	13.77%	14.84%
49	2	0.55	14.89%	15.21%	17.33%	10.24%
50	1	0.55	14.15%	15.21%	15.09%	17.27%
Average	2.28	0.57	15.91%	15.55%	15.75%	16.37%
STD	1.629	0.037	0.95%	0.94%	2.62%	3.30%

Table 8: WLTS Classification Models: Threshold 2

Split	n_N	$\frac{C_I}{C_{II}}$	Fit Data Set		Test Data Set	
			Type I	Type II	Type I	Type II
1	1	0.4	15.19%	14.86%	15.77%	13.79%
2	2	0.4	13.74%	14.37%	16.46%	22.73%
3	3	0.45	12.95%	12.64%	13.29%	19.32%
4	2	0.45	13.90%	14.94%	13.29%	20.46%
5	2	0.35	15.82%	16.00%	14.51%	14.94%
6	2	0.45	13.63%	14.21%	16.04%	16.28%
7	1	0.35	15.48%	15.52%	14.56%	14.77%
8	2	0.4	14.40%	14.29%	13.57%	19.46%
9	1	0.4	14.53%	14.94%	14.87%	11.36%
10	1	0.4	14.29%	14.12%	17.24%	16.47%
11	3	0.4	16.43%	14.94%	15.19%	13.64%
12	1	0.35	14.49%	13.37%	14.65%	13.33%
13	7	0.45	17.35%	16.19%	14.92%	13.48%
14	1	0.4	14.01%	13.97%	18.38%	15.66%
15	1	0.35	15.37%	14.77%	15.72%	16.28%
16	1	0.35	15.64%	15.52%	16.46%	18.18%
17	1	0.4	14.69%	12.64%	14.87%	18.18%
18	3	0.4	17.09%	16.57%	12.93%	14.94%
19	2	0.45	11.57%	10.80%	16.35%	24.42%
20	2	0.35	16.35%	14.62%	16.61%	15.39%
21	3	0.45	13.97%	15.29%	15.39%	13.04%
22	1	0.3	14.96%	15.12%	16.88%	13.33%
23	2	0.4	13.61%	14.29%	14.83%	19.54%
24	2	0.4	14.44%	14.77%	16.04%	18.61%
25	3	0.4	15.24%	13.56%	12.23%	17.65%
26	2	0.4	13.52%	13.45%	16.29%	23.08%
27	2	0.4	15.13%	14.46%	11.04%	18.75%
28	3	0.4	16.93%	16.57%	14.51%	9.20%
29	4	0.45	15.56%	15.82%	16.30%	15.29%
30	1	0.3	16.77%	15.39%	18.97%	7.53%
31	3	0.45	14.83%	14.44%	13.98%	15.85%
32	2	0.4	15.14%	16.19%	13.02%	22.47%
33	2	0.4	13.90%	14.94%	18.04%	14.77%
34	1	0.4	16.48%	17.05%	12.89%	17.44%
35	3	0.4	16.09%	15.03%	11.43%	11.24%
36	1	0.4	15.24%	14.69%	13.79%	15.29%
37	1	0.35	14.38%	13.77%	15.21%	18.95%
38	2	0.4	15.31%	14.44%	14.91%	14.63%
39	1	0.35	15.32%	15.52%	16.46%	14.77%
40	1	0.4	15.39%	13.53%	13.78%	18.48%
41	2	0.35	16.19%	14.12%	14.42%	12.94%
42	6	0.45	14.51%	14.45%	17.46%	21.35%
43	2	0.4	16.22%	14.61%	15.31%	11.91%
44	1	0.4	13.92%	14.29%	15.14%	20.69%
45	2	0.4	16.19%	16.95%	14.42%	14.12%
46	3	0.4	15.53%	14.47%	14.47%	19.77%
47	1	0.4	15.19%	15.43%	14.20%	14.94%
48	2	0.35	14.51%	14.45%	16.83%	12.36%
49	3	0.4	16.69%	15.70%	15.61%	12.22%
50	4	0.4	16.30%	13.50%	14.57%	16.16%
Average	2.1	0.4	15.09%	14.71%	15.08%	16.19%
STD	1.249	0.037	1.17%	1.15%	1.66%	3.62%

Table 9: WLTS Classification Models: Threshold 3

Split	n_N	$\frac{C_I}{C_{II}}$	Fit Data Set		Test Data Set	
			Type I	Type II	Type I	Type II
1	2	0.3	15.13%	14.29%	15.13%	14.93%
2	3	0.4	14.86%	15.67%	18.34%	19.70%
3	3	0.35	14.96%	14.39%	13.99%	16.18%
4	2	0.35	14.67%	15.91%	13.39%	20.59%
5	4	0.45	14.71%	13.43%	13.02%	24.24%
6	5	0.4	16.02%	17.29%	14.54%	13.43%
7	3	0.35	16.02%	14.29%	14.84%	19.40%
8	3	0.35	16.62%	14.29%	14.84%	17.91%
9	3	0.35	16.00%	15.91%	15.77%	10.29%
10	2	0.35	12.84%	13.87%	16.13%	17.46%
11	3	0.35	17.75%	17.56%	15.22%	14.49%
12	3	0.45	11.72%	12.03%	14.84%	22.39%
13	3	0.35	15.90%	17.91%	13.91%	9.09%
14	3	0.4	14.78%	16.06%	16.42%	12.70%
15	2	0.3	15.24%	15.27%	15.82%	17.39%
16	1	0.25	17.06%	16.54%	17.22%	17.81%
17	3	0.4	14.18%	13.87%	15.25%	19.05%
18	5	0.45	14.24%	14.29%	12.46%	20.90%
19	5	0.4	16.87%	13.87%	13.78%	12.70%
20	1	0.3	14.92%	16.15%	15.27%	20.00%
21	2	0.35	17.67%	16.41%	15.36%	11.11%
22	3	0.4	15.88%	14.29%	15.43%	11.94%
23	3	0.35	16.79%	14.93%	17.46%	18.18%
24	4	0.4	14.52%	14.39%	16.96%	16.18%
25	4	0.4	14.67%	15.91%	13.69%	20.59%
26	4	0.4	15.73%	13.53%	15.73%	20.90%
27	1	0.3	15.17%	14.84%	13.25%	20.83%
28	5	0.4	18.69%	17.29%	16.02%	7.46%
29	4	0.4	17.26%	16.30%	15.63%	9.23%
30	5	0.4	15.42%	14.29%	14.24%	17.57%
31	3	0.4	15.44%	14.29%	15.12%	11.67%
32	3	0.35	15.13%	15.79%	14.54%	19.40%
33	1	0.25	15.73%	16.54%	18.99%	13.43%
34	5	0.45	15.90%	15.67%	10.65%	19.70%
35	2	0.3	17.16%	16.03%	13.73%	14.49%
36	4	0.45	14.44%	14.82%	11.21%	16.29%
37	2	0.35	15.68%	13.74%	15.82%	18.84%
38	5	0.45	15.74%	15.71%	14.54%	18.33%
39	3	0.3	17.83%	15.67%	18.94%	9.09%
40	4	0.45	16.02%	15.04%	12.76%	13.43%
41	3	0.35	17.39%	18.66%	14.79%	21.21%
42	2	0.35	13.08%	14.93%	15.39%	25.76%
43	4	0.4	16.62%	14.29%	15.73%	14.93%
44	2	0.35	13.76%	14.50%	14.03%	15.94%
45	4	0.45	14.88%	15.56%	12.98%	16.92%
46	4	0.35	18.48%	16.91%	15.88%	14.06%
47	3	0.4	13.13%	15.33%	15.25%	11.11%
48	1	0.25	17.66%	16.54%	20.48%	17.91%
49	4	0.4	15.46%	14.84%	13.55%	22.22%
50	5	0.4	16.64%	14.06%	15.66%	18.06%
Average	3.16	0.37	15.65%	15.28%	15.08%	16.55%
STD	1.201	0.054	1.46%	1.30%	1.84%	4.21%

Table 10: WLTS Classification Models

Threshold	Fit Data Set		Test Data Set		Average	
	Type I	Type II	Type I	Type II	n_N	$\frac{C_I}{C_{II}}$
1	15.91%	15.55%	15.75%	16.37%	2.28	0.57
2	15.09%	14.71%	15.08%	16.19%	2.10	0.40
3	15.65%	15.28%	15.08%	16.55%	3.16	0.37

respective system. The WLTS case study comprised of three classification experiments, each with a different threshold value for module classification. If a module had number of faults, discovered during system tests, greater than or equal to the threshold value, it was classified as fault-prone. The threshold values considered for the WLTS case study are – Threshold 1: number of faults = 1; Threshold 2: number of faults = 2; and Threshold 3: number of faults = 3.

Similar to the LLTS case study, for each $\mathbf{n}_N$ value, the cost ratio, $\frac{C_I}{C_{II}}$, was varied with different values. Such classification models were built for all three threshold values listed above. Moreover, for each threshold value, experiments with different $\mathbf{n}_N$ and $\frac{C_I}{C_{II}}$ were performed for all of the 50 data splits. For each pair of the fit and test data sets, a model with a good balance between the Type I and Type II errors is selected, and its $\mathbf{n}_N$ and $\frac{C_I}{C_{II}}$ are recorded.

Tables 7, 8, and 9 display the model parameters and misclassification errors of the selected model for each data split. The misclassification error rates for the fit data set are based on cross-validation results obtained during the model building process. It can be observed that many models with $\mathbf{n}_N = 1$ gave best results, i.e., good balance of error rates. This indicates that within the case library, there were several groups of cases that contained the same exact values of all five of the independent variables used during modeling. As a result, a classification match is perfect in this case. Such a perfect classification is unique to CBR, and further validates its attractiveness as a modeling technique for embedded systems.

Selecting the best model (based on balance and error rates) out of these 50 as the final classification model may not be appropriate, because the associated data split may be biased. Computing the average values across all splits gives a more realistic picture of the performance of the modeling technique. The average values of $\mathbf{n}_N$ and $\frac{C_I}{C_{II}}$ and their standard deviation (abbreviated as STD) over the 50 data splits is presented at the bottom of Tables 7, 8, and 9. These values along with the average Type I and Type II errors for the fit and test data sets are summarized in Table 10.

For each of the respective threshold values, our empirical findings suggest that the Type I and Type II errors of the fit and test data sets were quite similar, i.e., very low overfitting. Also, a relatively small number of cases were needed to classify modules. For example, an average of 2.10 cases were needed when the threshold value was 2 faults. Thus, for the WLTS case study about 2 to 4 cases (out of 807 cases in the library) may be required to identify an unclassified module.

Similar to the LLTS case study, the CBR classification models for the WLTS case study demonstrated good model stability with almost no over-fitting. For example, the difference between the Type I and Type II errors of the fit and test data sets (Threshold 2) were approximately 0.2% and 0.7%, respectively. Further more, despite the different threshold values considered, the classification accuracy of the WLTS CBR model were very similar. This indicates that a CBR classification model may not be too sensitive to the threshold value selected during model calibration.

6 Conclusion

Reliability of embedded systems such as telecommunications systems demands utmost scrutiny, because system failures may have severe and sometimes even catastrophic consequences. Rectifying problems at remote sites can be exhaustive on both monetary and human resources of the embedded system's organization. Software engineering techniques for embedded systems must reflect the needs of the operational environment, and desirably be compatible with the evolution of the software over subsequent system releases.

Software quality modeling tools that aid in focussing improvement efforts on the high-risk system modules, can be valuable in reliability estimation and quality control of embedded systems. This is because, some quality improvements are so expensive that it is not practically feasible to apply them to all modules. Targeting such improvement techniques is an effective way to reduce the likelihood of faults discovered during operations. A software quality classification model can be used to detect the fault-prone modules early in their life-cycle, thereby facilitating focused cost-effective quality enhancement activities to the high-risk areas. Embedded systems can benefit greatly from such classification models, if an accurate and stable modeling technique is used for the purpose.

In this paper, we examined the effectiveness of a *case-based reasoning* approach for calibrating software quality classification models. To illustrate this, two large-scale case studies of embedded software systems were used. The first study, a legacy telecommunications system, involved measurement and fault data over multiple system releases. The second study consisted of data from two wireless telecommunications systems.

The first case study illustrated that the CBR model designed in this paper indicated good model stability across the system releases. The classification model may be useful for software engineering of its subsequent system releases as the embedded system evolves. Despite the use of 42 software metrics as independent variables, model calibration using the illustrated CBR classification technique was relatively faster and easier than other classification methods [9, 14].

The second case study further validated the effectiveness of building classification models using the CBR technique illustrated. Further more, it was demonstrated that the CBR classification model was robust with respect to the chosen fault-prone threshold value. The good classification accuracy obtained for this case study, suggests that with CBR one can obtain useful software quality models by using a few (inexpensive to obtain) software metrics. The subsequent system release of the embedded software for the WLTS case study is currently under development. Our current and future work will involve data collection efforts for software measurement and fault data of the newly developed/updated modules. The CBR classification model calibrated in our WLTS case study will be validated using the data collected for the second release. The effect of system evolution on the classification model will be investigated further.

The achieved accuracy and stability of the CBR classification models for the two case studies would be beneficial to developers of an embedded system, especially those of a telecommunications system. Fault-prone modules of the system would be identified prior to system testing, and developers could then focus cost-effective efforts on those modules. Consequently, almost all software faults of the embedded system may be detected and rectified prior to system operations. Future research may concentrate on expanding the use of the illustrated CBR approach for effective software engineering of other embedded systems.

Acknowledgments

We thank John P. Hudepohl, Wendell D. Jones and the EMERALD team for collecting the necessary case-study data (LLTS). We also thank Michelle Lim and Linda Lim for their data collection efforts (WLTS), and Fletcher Ross for his assistance with experiments and data analysis. We also thank Kenneth McGill for his helpful suggestions. This work was supported in part by Cooperative Agreement NCC 2-1141 from NASA Ames Research Center, Software Technology Division, and Center Software Initiative for the NASA software Independent Verification and Validation Facility at Fairmont, West Virginia.

References

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

[2] L. C. Briand, T. Langley, and I. Wieczorek. A replicated assessment and comparison of common software cost modeling techniques. *International Conference of Software Engineering*, pages 377–386, June 2000.

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

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

[5] K. Ganesan, T. M. Khoshgoftaar, and E. B. Allen. Case-based software quality prediction. *International Journal of Software Engineering and Knowlegde Engineering*, 10(2):139–152, 2000.

[6] 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.

[7] T. M. Khoshgoftaar and E. B. Allen. Logistic regression modeling of software quality. *International Journal of Reliability, Quality and Safety Engineering*, 6(4):303–317, Dec. 1999.

[8] T. M. Khoshgoftaar and E. B. Allen. A practical classification rule for software quality models. *IEEE Transactions on Reliability*, 49(2), June 2000.

[9] T. M. Khoshgoftaar and E. B. Allen. Modeling software quality with classification trees. In H. Pham, editor, *Recent Advances in Reliability and Quality Engineering*, chapter 15, pages 247–270. World Scientific Publishing, Singapore, 2001.

[10] T. M. Khoshgoftaar, E. B. Allen, and J. C. Busboom. Modeling software quality: The software measurement analysis and reliability toolkit. In *Proceedings: International Conference on Tools with Artificial Intelligence*, pages 54–61, Nov 2000.

[11] 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.

[12] T. M. Khoshgoftaar, E. B. Allen, and R. Shan. Improving tree-based models of software quality with principal components analysis. In *Proceedings or the Eleventh International Symposium on Software Reliability Engineering*, pages 198–209, San Jose, California USA, Oct. 2000. IEEE Computer Society.

[13] T. M. Khoshgoftaar, K. Ganesan, E. B. Allen, F. D. Ross, R. Munikoti, N. Goel, and A. Nandi. Predicting fault-prone modules with case-based reasoning. In *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, pages 27–35, Albuquerque, NM USA, Nov. 1997. IEEE Computer Society.

[14] 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.

[15] T. M. Khoshgoftaar, X. Yuan, and D. L. Lanning. Balancing misclassification rates in classification tree models of software quality. *Empirical Software Engineering*, 5:313–330, 2000.

[16] J. Kolodner. *Case-Based Reasoning*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.

[17] L. Lim. Developing accurate software quality models using a faster, easier, and cheaper method. Master's thesis, Florida Atlantic University, Boca Raton, FL USA, May 2001. Advised by T. M. Khoshgoftaar.

[18] R. S. Michalski, I. Brato, and M. Kubat. *Machine Learning and Data Mining*. John Wiley and Sons, 1998.

[19] J. C. Munson and T. M. Khoshgoftaar. Software metrics for reliability assessment. In M. Lyu, editor, *Handbook of Software Reliability Engineering*, chapter 12, pages 493–529. McGraw-Hill, New York, 1996.

[20] N. Ohlsson, M. Zhao, and M. Helander. Application of multivariate analysis for software fault prediction. *Software Quality Journal*, 5:51–66, 1998.

[21] F. D. Ross. An empirical study of analogy based software quality classification models. Master's thesis, Florida Atlantic University, Boca Raton, FL USA, Aug. 2001. Advised by T. M. Khoshgoftaar.

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

[23] N. F. Schneidewind. Software metrics model for integrating quality control and prediction. In *Proceedings of the Eighth ISSRE*, pages 402–415, Albuquerque, NM USA, Nov. 1997. IEEE Computer Society.