A Practical Classification Rule for Software Quality Models

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

Keywords — **software reliability, classification, prior probabilities, costs of misclassification, software metrics**

Summary & Conclusions — **A "practical" classification rule for a software quality model is one that considers the needs of the project to use a model to guide targeting software reliability-enhancement efforts, such as extra reviews early in development. Such a rule will often prove more useful than alternative rules. The contribution of this paper is discussion of several classification rules for software quality models, and recommendation of a generalized classification rule, where the effectiveness and efficiency of the model for guiding software reliability-enhancement efforts can be explicitly considered. This is the first application of this rule to software quality modeling that we know of.**

Two case studies illustrate application of the generalized classification rule. A case study of a telecommunications system models membership in the class of fault-prone modules as a function of the number of interfaces to other modules. A case study of a military system models membership in the class of fault-prone modules as a function of a set of process metrics that depict the development history of a module. These case studies are examples where balanced misclassification rates resulted in more useful and practical software quality models than the other classification rules.

1 Introduction

Correcting software faults early in development is much more effective and less expensive than waiting until they become evident later. Consequently, software developers of mission-critical systems often apply various techniques throughout the life cycle to improve reliability. Reliability improvement techniques include design and code reviews, automatic test-case generation, more extensive testing, and strategic assignment of tasks to personnel. These are often guided by reliability modeling and prediction.

However, it may not be practical to apply some techniques to every software module. For example, the lead architect may not have time to participate in a detailed design review of every module. In such cases, our goal is to target reliability-enhancement efforts to those modules considered *fault-prone* by the project. Unfortunately, one often does not know whether a module is *fault-prone* until faults are discovered late in development, or even after release.

We use the term "reliability" in a broad sense, rather than as the probability of failure-free execution. Thus, "reliability-enhancement techniques" are simply processes that developers use to find faults early. This paper does not recommend any particular practices over others. According to standard terminology, a "fault" is a defect in a program that may cause incorrect execution [1]. Frankl et al. [2] discuss "delivered reliability" as the probability of failure under operational conditions. In contrast, we focus on the absence of faults recorded by a problem reporting system during the period

of interest as an indicator of reliability, in a broad sense, irrespective of whether a failure resulting from a fault would be frequent or rare during operations. Even rare failures can be very important in mission-critical systems.

Software quality classification models can be a valuable guide for cost-effective quality improvement efforts by early identification of the modules which have high risk of faults. A model is built based on a historical project's data, i.e., a *fit* data set. Ideally, it is validated using an *s*-independent *test* data set, to indicate its expected accuracy. Thereafter, it can be applied to each module of a similar current project. The model predicts the class of each module based on its known attributes. With predictions in hand, one can then target reliability-enhancement efforts on those predicted to be *fault-prone*.

We [3] and others [4] have observed that published modeling methods may not always produce models with useful accuracy. Consistent with our previously published work, a *Type I* misclassification is when a model classifies a software module as *fault-prone* which is actually *not fault-prone*, and a *Type II* misclassification is when a model classifies a software module as *not fault-prone* which is actually *fault-prone*. Because practical interpretation of Type I and Type II misclassification rates may be difficult, we propose in this paper to translate these into measures of the *effectiveness* and *efficiency* of a classification model, which are more closely related to project management concerns [5]. We define *effectiveness* of a model as the proportion of *fault-prone* modules correctly identified. We define *efficiency* of a model as the proportion of modules predicted to be *fault-prone* that actually are. If either type of misclassification rate is large, the model

is generally not useful to guide reliability improvement efforts. This paper argues that a "practical" classification rule is one that considers both the *effectiveness* and *efficiency* of using model predictions according to the needs of the project. To this end, this research is the first that we know of to apply the proposed generalized classification rule to software quality models.

Prior software quality classification models have used several other classification rules. Many studies, using a variety of modeling techniques, classify a module into its most likely class [6, 7, 8, 9, 10, 11, 12]. Our research group and some others have published case studies using a rule that maximizes the number of correct classifications [13, 14]. More recently, we have advocated a rule that minimizes the expected cost of misclassifications [15, 16]. In this paper, we discuss these classification rules and then present a generalized rule designed to give practical results for targeting reliability-enhancement efforts. Two industrial case studies illustrate the effect of the new rule.

Future research may investigate the application of quantitative models to identification of *fault-prone* modules.

Acronyms[1]

ECM	Expected Cost of Misclassification
JSTARS	Joint Surveillance Target Attack Radar System
STR	Software Trouble Report

[1]The singular and plural of an acronym are always spelled the same.

Notation

G_1	the *not fault-prone* class (group) of modules
G_2	the *fault-prone* class (group) of modules
k	index of classes
i	index of modules
$\mathbf{x}_i$	the vector of independent variables of the i^{th} module
$Class(\mathbf{x}_i)$	the i^{th} module's class, predicted by a model's classification rule
$f_k(\mathbf{x}_i)$	a likelihood function for the i^{th} module's membership in the k^{th} class
λ	a smoothing parameter used in nonparametric density estimation
$\hat{f}_k(\mathbf{x}_i\|\lambda)$	an estimate of $f_k(\mathbf{x}_i)$
π_k	the prior probability of membership in G_k
$q_k(\mathbf{x}_i)$	the posterior probability of membership in G_k
C_I	the cost of a Type I misclassification
C_{II}	the cost of a Type II misclassification
$\Pr\{1\|1\}$	the probability that a model correctly classifies a module as in G_1
$\Pr\{2\|2\}$	the probability that a model correctly classifies a module as in G_2
$\Pr\{2\|1\}$	the probability that a model misclassifies a module as in G_2 which is actually in G_1, i.e., the Type I misclassification rate.
$\Pr\{1\|2\}$	the probability that a model misclassifies a module as in G_1 which is actually in G_2, i.e., the Type II misclassification rate.

Other standard notation is given in "Information for Readers and Authors" in the rear of each issue.

2 Classification Rules

This section presents several alternative classification rules for use with software quality models, to predict the class of the i^{th} module, $Class(\mathbf{x}_i)$.

2.1 Most likely class

Suppose a software quality modeling technique produces a likelihood function, f_k, for each class, $k = 1, 2$. For example, in nonparametric discriminant analysis, each likelihood function is a probability density function. In logistic regression, each likelihood function is a probability of an event, or non-event. A decision rule that chooses the most likely class is the following [17, 18].

$$Class(\mathbf{x}_i) = \begin{cases} G_1 & \text{if } f_1(\mathbf{x}_i) \geq f_2(\mathbf{x}_i) \\ G_2 & \text{otherwise} \end{cases} \tag{1}$$

In our case studies, G_1 is the class *not fault-prone* and G_2 is the class *fault-prone*.

2.2 Maximum correct classifications

The most-likely-class rule does not consider the characteristics of the population of modules from which the data set is drawn. If module attributes, $\mathbf{x}_i$, imperfectly separate the

classes, and if the population is predominantly *not fault-prone*, then the most-likely-class rule will probably result in a Type I misclassification rate, $\Pr\{2|1\}$, that is too high. For example, suppose the population is 80% *not fault-prone* and 20% *fault-prone*. If a module's attributes, $\mathbf{x}_i$, are in the region where both classes have likelihood functions greater than zero, $f_k(\mathbf{x}_i) > 0, k = 1, 2$, then the most-likely-class rule will give equal consideration to the two classes, even though 80% are *not fault-prone*. Thus, too many *not fault-prone* modules will be classified as *fault-prone*. An improved classification rule should take into account the overall class proportions of the underlying population.

From a Bayesian viewpoint, the class proportions of the population are information which is known prior to applying a model, i.e., the prior probabilities of class membership. When prior probabilities, $\pi_k, k = 1, 2$, are known, a classification rule should assign an observation, $\mathbf{x}_i$, to the class with the greater posterior probability of membership [18], which is given by

$$q_k(\mathbf{x}_i) = \frac{f_k(\mathbf{x}_i)\pi_k}{f_1(\mathbf{x}_i)\pi_1 + f_2(\mathbf{x}_i)\pi_2} \tag{2}$$

for $k = 1, 2$. The corresponding classification rule is

$$Class(\mathbf{x}_i) = \begin{cases} G_1 & \text{if } q_1(\mathbf{x}_i) \geq q_2(\mathbf{x}_i) \\ G_2 & \text{otherwise} \end{cases} \tag{3}$$

By Equation (2), this is equivalent to

$$Class(\mathbf{x}_i) = \begin{cases} G_1 & \text{if } \frac{f_1(\mathbf{x}_i)}{f_2(\mathbf{x}_i)} \geq \frac{\pi_2}{\pi_1} \\ G_2 & \text{otherwise} \end{cases} \tag{4}$$

This classification rule maximizes the expected number of correct classifications [17, 18].

The maximum-correct-classification rule may result in a large Type II misclassification rate, $\Pr\{1|2\}$, when *fault-prone* modules are rare, $\pi_2 \ll \pi_1$, for example $\pi_2 = 0.20$ and $\pi_1 = 0.80$. This means there are few misclassifications, but a large Type II rate means that a large percentage of the *fault-prone* modules are misclassified. In this case, such a rule is not practical, because it does not achieve the goal of early detection of the *fault-prone* modules.

2.3 Minimum expected cost of misclassifications

The maximum-correct-classifications rule assumes that the penalties for all kinds of misclassifications are the same. In software engineering practice, the penalty for acting on a Type II misclassification is often much more severe than for a Type I. Thus, an improved classification rule should take into account the costs of each kind of misclassification.

The cost of a Type I misclassification, C_I, is the direct cost wasted trying to enhance a module that is already *not fault-prone*. A reliability-enhancement technique, such as extra reviews, typically has modest direct cost per module. On the other hand, the cost of a Type II misclassification, C_{II}, is the lost opportunity to correct faults early. In other words, let C_{II} be the benefit (cost avoidance) of using the model to guide reliability-enhancement of a *fault-prone* module. Calculation of C_{II} may consider the costs of distributing and installing fixes at customer sites after release, and perhaps even the cost of consequential damages due to software faults. In this paper we model C_I and

C_{II} as constants, and we assume the cost of a correct classification is zero. The cost of modeling is also assumed to be already budgeted, and thus, is not considered here. A more sophisticated cost model is a topic for future research.

The expected cost of misclassification of one module is

$$ECM \;=\; C_I \Pr(2|1)\,\pi_1 + C_{II} \Pr(1|2)\,\pi_2 \tag{5}$$

A classification rule that minimizes the expected cost of misclassification [17, 18] is

$$Class(\mathbf{x}_i) = \begin{cases} G_1 & \text{if } \frac{f_1(\mathbf{x}_i)}{f_2(\mathbf{x}_i)} \geq \left(\frac{C_{II}}{C_I}\right)\left(\frac{\pi_2}{\pi_1}\right) \\ G_2 & \text{otherwise} \end{cases} \tag{6}$$

Note that Equation (4) is a special case of Equation (6), where costs are equal ($C_{II}/C_I = 1$). Equation (1) is also a special case where prior probabilities are uniform ($\pi_2/\pi_1 = 1$) and costs are equal.

This rule is appealing because costs are always very interesting to management. However, if C_{II}/C_I is very large, the rule may result in such a large Type I misclassification rate, $\Pr\{2|1\}$, that the model would recommend applying reliability-enhancement processes to almost all the modules in the system. In other words, the model would say that discovering a *fault-prone* module is so valuable that wasting effort on many *not fault-prone* modules is worthwhile for the sake of finding a few *fault-prone* modules that a less extreme rule would miss. In this paper, we assume that reliability enhancements are given to some modules, but not all. Some enhancement processes may be so expensive or time-consuming that they must be applied judiciously. Reliability-enhancement treat-

ments for practically all modules may not be affordable, no matter how large C_{II}/C_I is. Similarly, schedule constraints may limit the amount of reliability-enhancement activity.

2.4 Generalized classification

In many projects, costs of misclassification are difficult to estimate. Similarly, prior probabilities may also be unknown or difficult to estimate. The minimum-expected-cost rule may be impractical in such cases. Thus, a more general rule that does not require these parameters is needed.

In a standard development process, the expected proportion of *fault-prone* modules is π_2. Following the model's recommendation, the proportion of modules that receive reliability enhancement that are actually *fault-prone* is $\Pr\{2|2\}\pi_2$. Let us define *effectiveness* as the proportion of *fault-prone* modules that received reliability-enhancement treatment out of all the *fault-prone* modules.

$$\textit{effectiveness} \quad = \quad \frac{\Pr\{2|2\}\pi_2}{\pi_2} = 1 - \Pr\{1|2\} \tag{7}$$

This means that we can maximize *effectiveness* by minimizing $\Pr\{1|2\}$.

When we apply a reliability-enhancement process to a *not fault-prone* module, it will probably be a waste of time, because the reliability is already satisfactory. Let us define *efficiency* as the proportion of reliability-enhancement effort that is not wasted, namely, the proportion of *fault-prone* modules among those recommended for enhancement.

$$\textit{efficiency} \quad = \quad \frac{\Pr\{2|2\}\pi_2}{\Pr\{2|1\}\pi_1 + \Pr\{2|2\}\pi_2} \tag{8}$$

This means that we can maximize *efficiency* by minimizing Pr{2|1}, for given Pr{2|2}, π_1, and π_2.

There is a tradeoff between Pr{2|1} and Pr{1|2}. We have observed that as one goes down, the other goes up. Our goal is to design a practical, flexible classification rule that allows appropriate emphasis on *effectiveness* and *efficiency* according to the needs of the project. The following rule enables a project to select the best balance between the misclassification rates, and consequently, between *effectiveness* and *efficiency*.

$$Class(\mathbf{x}_i) = \begin{cases} G_1 & \text{if } \frac{f_1(\mathbf{x}_i)}{f_2(\mathbf{x}_i)} \geq c \\ G_2 & \text{otherwise} \end{cases} \tag{9}$$

where c is a constant which we can choose. We have observed in several empirical studies that the misclassification rates, which range in value from zero to one, are approximately monotonic functions of c. Accordingly, we have observed a monotonic empirical relationship between c and both *effectiveness* and *efficiency* as well. We estimate these functions by repeated calculations with a *fit* data set. Given a candidate value of c, we estimate Pr{2|1} and Pr{1|2}. We repeat for various values of c. Determining the c that corresponds to a project's preferred balance between *effectiveness* and *efficiency* is straightforward when they are monotonic functions of c.

The generalized classification rule does not depend on our knowledge of π_1 and π_2, nor of C_I and C_{II}, and thus, is useful when these quantities are difficult to estimate. However, if information is available, having selected a preferred c, one can interpret the value of c in terms of the priors ratio times the cost ratio, as in the minimum-expected-

cost rule.

If one chooses c such that $\Pr\{2|1\} = \Pr\{1|2\}$, then both misclassification rates are minimized [18]. It is especially appropriate when $\pi_2 \ll \pi_1$, e.g., LeGall et al. [19] identified only 4% to 6% of modules as high-risk. In such cases, the maximum-correct-classification rule is often unsatisfactory. Equal misclassification rates may also be appropriate when C_{II}/C_I is so large that the minimum-expected-cost rule is unaffordable. In practice, we can achieve only approximate equality due to finite discrete data sets.

3 Evaluation of Models

In the case studies, we used three methods for estimating $\Pr\{2|1\}$ and $\Pr\{1|2\}$ [20]. In each method, a *fit* data set is used to estimate model parameters, and a *test* data set is used to evaluate model accuracy.

Resubstitution uses the same data set for *fit* and *test*. The results may be biased, because the evaluation is based on the same data as parameter estimates.

Cross-validation works in the following way [21, 22]: Suppose there are n observations available in a single data set. Let one observation be the *test* data set and all the others be the *fit* data set. Build a model, and evaluate it for the current observation. Repeat for each observation, resulting in n models. Let the misclassification rates summarize the n evaluations of the models. This is not biased like resubstitution, but requires more computation. For the generalized-classification rule, we prefer cross-validation rather

than resubstitution as an evaluation method when choosing c, whenever practical [20, 16].

Data splitting derives *fit* and *test* data sets from a single data set by impartially sampling from available observations.

4 Case Studies

4.1 Methodology

The following steps describe the modeling methodology used in our case studies.

1. Collect configuration management data and problem reporting system data from a past project.

2. Determine the class of each module.

$$Class = \begin{cases} not\ fault\text{-}prone & \text{If } Faults < threshold \\ fault\text{-}prone & \text{If } Faults \geq threshold \end{cases} \tag{10}$$

   where *threshold* is chosen according to project-specific criteria. For example, project management may consider how model predictions will be used, and whether there are enough modules in each class for adequate sample sizes. Other projects may choose different *thresholds*.

3. Prepare *fit* and *test* data sets by data splitting.

   Derive *fit* and *test* data sets from the data by impartially sampling from the set of modules studied. In the case studies, the *fit* data set had two thirds of the

modules, and the *test* data set had the remaining third. These proportions were chosen, considering sample sizes. Other proportions may be appropriate for other studies.

4. Select s-significant independent variables [23] using stepwise discriminant analysis on the *fit* data set [18]. See Appendix A for details.

5. Estimate likelihood functions of the final model using nonparametric discriminant analysis on the *fit* data set.

   Nonparametric discriminant analysis estimates each within-class probability density as a function of the independent variables for each class. We used the normal kernel method of nonparametric density estimation with smoothing parameter λ [14]. The smoothing parameter, λ, was chosen to optimize the results of cross-validation with the *fit* data set. See Appendix B for details.

6. Predict the class of each module to evaluate model accuracy.

   Each model was evaluated by the generalized-classification rule in Equation (9) with various values of c. Evaluation was based on resubstitution of the *fit* data set, cross-validation with the *fit* data set, and the *test* data set generated by data splitting. The *test* data set result tells us the level of accuracy to expect when applying the model to a similar project or subsequent release when the actual class membership is not known.

Table 1: System Profile

Application	Telecommunications
Language	Pascal-like
Lines of Code	1.3 million
Executable Statements	1.0 million
CFG Edges	364 thousand
Source Files	25 thousand
Functional Modules	2 thousand

4.2 Telecommunications System

This case study examined a very large telecommunications system written by professional programmers in a large organization [3, 13, 24]. This embedded computer application included numerous finite state machine algorithms and interfaces to various kinds of equipment. A random sample of 1,980 modules was taken for analysis. As shown in Table 1, the sample represented about 1.3 million lines of code. It was written in a proprietary procedural high level language similar to Pascal.

In this study, we focused on the number of faults found in the current version, *Faults*, during the period from entry of a module into the source code control system in the Coding phase until the end of data collection in the Operational phase. Table 2 shows the distribution of faults among modules.

Table 2: Distribution of Faults

	Quantity			Faults	
Modules	Total	Zeros	Fault-Prone	Mean	Std Dev
All	1980	1103	239	1.93	4.52
New	194	53	54	3.43	4.35
Changed	917	274	177	3.22	5.86
Unchanged	869	776	8	0.22	0.99

Project engineers defined the *not fault-prone* group, G_1, as those modules with *Faults* < 5, and the *fault-prone* group, G_2, as those with *Faults* ≥ 5. G_2 had about 12% of the modules. Another criterion for *fault-prone* group membership might be appropriate in other situations. Statistical techniques can evaluate the sensitivity of results to the choice of threshold [5]. Class membership was the dependent variable of the model.

Various product metrics were collected from source code at the module level. A module may consist of multiple files. We built a nonparametric discriminant model, with the single best-correlated design metric, the number of Modules Used, MU, as the independent variable [3]. MU is related to the number of interfaces to other modules.

Using the generalized-classification rule, Table 3 shows the effect of various values of c on the accuracy of the single variable model. The *fit* data set was used for resubstitution and for cross-validation. The *test* data set generated by data splitting was used for

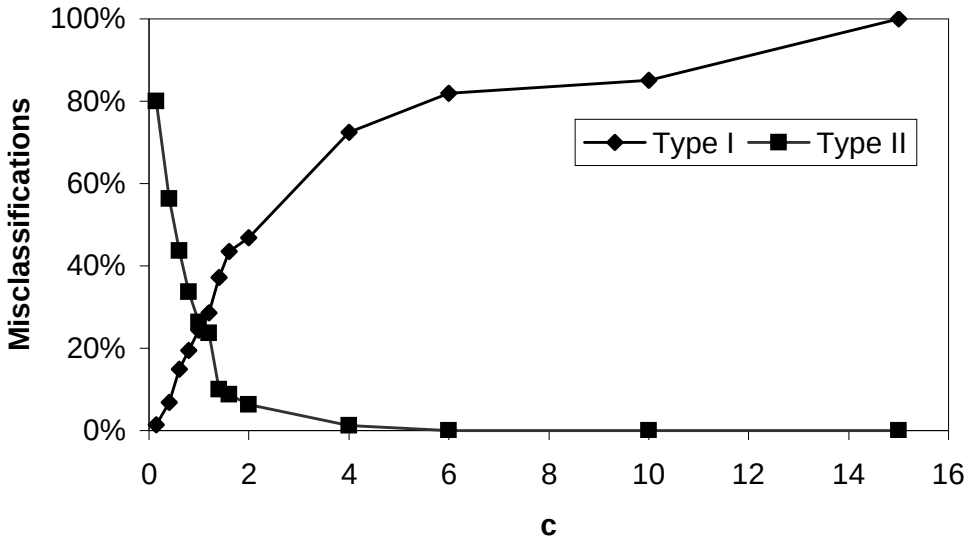

Figure 1: Misclassification Rates for Single Variable Model (data splitting)

evaluation. Figure 1 depicts the effect of c on misclassification rates for the data-splitting columns in Table 3. The *fit* data set had 1,320 observations, consisting of 1,161 *not fault-prone* modules (base of Type I per cent) and 159 *fault-prone* modules (base of Type II per cent). The *test* data set had 660 observations, consisting of 580 *not fault-prone* modules (base of Type I per cent) and 80 *fault-prone* modules (base of Type II per cent).

Figure 2 depicts the effect of c on the associated *effectiveness* and *efficiency* shown in Table 4.

A value of $c = 1$ corresponded to the most-likely-class rule according to Equations (1) and (9). This rule would probably be satisfactory in this case, having data-splitting column results of $\Pr\{2|1\} = 24.31\%$ and $\Pr\{1|2\} = 26.25\%$. This translated to an effectiveness of 73.75% and an efficiency of 29.5%.

Table 3: Misclassification Rates for Single Variable Model

Telecommunications System

MU was the single independent variable

Prior probabilities: proportions of fit data

Misclassification rates (%)

		Resubstitution		Cross-validation		Data Splitting	
		Pr{2|1}	Pr{1|2}	Pr{2|1}	Pr{1|2}	Pr{2|1}	Pr{1|2}
c	λ	Type I	Type II	Type I	Type II	Type I	Type II
0.14	0.005	0.34	82.39	1.03	86.16	1.38	80.00
0.40	0.50	6.72	62.26	6.72	62.26	6.90	56.25
0.60	0.40	12.75	47.80	12.75	47.80	14.83	43.75
0.80	0.40	16.97	39.62	16.97	39.62	19.48	33.75
1.00	0.40	21.53	35.22	21.53	35.22	24.31	26.25
1.20	0.40	25.15	32.70	25.15	32.70	28.62	23.75
1.40	0.30	33.16	21.38	33.16	21.38	37.24	10.00
1.60	0.15	40.83	13.84	40.91	13.84	43.45	8.75
2.00	0.10	43.67	12.58	43.76	12.58	46.90	6.25
4.00	0.15	72.78	3.14	72.87	3.14	72.41	1.25
6.00	0.15	81.22	1.26	81.31	1.26	81.90	0.00
10.00	0.10	85.01	1.26	85.10	1.26	85.17	0.00
15.00	0.20	100.00	0.00	100.00	0.00	100.00	0.00

Table 4: Effectiveness and Efficiency for Single Variable Model

Telecommunications System

Test data set		
c	*effectiveness* (%)	*efficiency* (%)
0.14	20.00	66.66
0.40	43.75	46.65
0.60	56.25	34.35
0.80	66.25	31.93
1.00	73.75	29.50
1.20	76.25	26.87
1.40	90.00	25.00
1.60	91.25	22.46
2.00	93.75	21.61
4.00	98.75	15.83
6.00	100.00	14.41
10.00	100.00	13.94
15.00	100.00	12.12

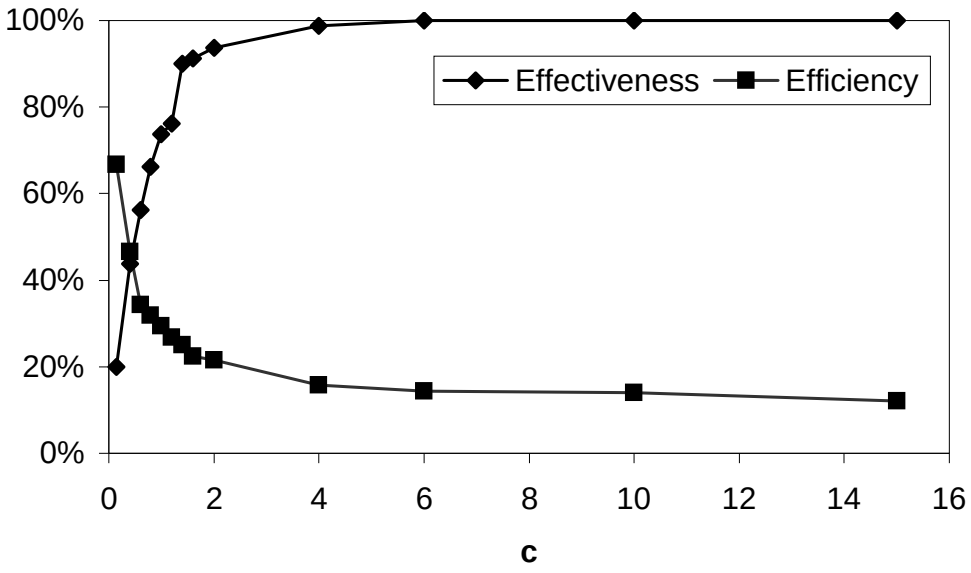

Figure 2: Effectiveness and Efficiency for Single Variable Model

A value of $c = 0.14$ corresponded to the maximum-correct-classifications rule according to Equations (4) and (9). This rule was not practical, because $\Pr\{1|2\} = 80\%$ in the data-splitting column. In other words, it was ineffective in finding *fault-prone* modules. For the *test* data set, it had an effectiveness of only 20.00% and an efficiency of 66.66%.

A value of $1.2 < c < 1.4$ had approximately equal cross-validation misclassification rates from $\Pr\{2|1\} = 25.15\%$ and $\Pr\{1|2\} = 32.70\%$ to $\Pr\{2|1\} = 33.16\%$ and $\Pr\{1|2\} = 21.38\%$. Exactly equal misclassification rates were not empirically feasible. The corresponding data-splitting column results ranged from $\Pr\{2|1\} = 28.62\%$ and $\Pr\{1|2\} = 23.75\%$ to $\Pr\{2|1\} = 37.24\%$ and $\Pr\{1|2\} = 10.00\%$. When equality is not achievable in practice, we prefer that $\Pr\{1|2\} < \Pr\{2|1\}$, because usually $C_{II} > C_I$. A value of $c = 1.4 = (0.137)(10.2)$ corresponded to the minimum-expected-cost rule at a

cost ratio of $C_{II}/C_I = 10.2$ according to Equations (6) and (9). For the *test* data set, this translated to an effectiveness of 90% and an efficiency of 25%.

Even though this model was not necessarily the most accurate possible for this system, and even though we do not generally recommend single variable models [3], this case study is a simple example of how the value of c can influence model accuracy. In this case, if one chose an appropriate value of c, then the model could be of practical value to target extra reviews of module designs early in development.

4.3 Military System

The Joint Surveillance Target Attack Radar System, JSTARS, was developed by Northrop Grumman for the U.S. Air Force in support of the U.S. Army [25, 26]. The system consists of an E-8 aircraft with a multimode radar system and mobile ground stations. Computer systems are both in the aircraft and on the ground. The system performs ground surveillance, providing real-time detection, location, classification, and tracking of moving and fixed objects for a real-time tactical view of the battlefield. The system was developed under the spiral life cycle model [27].

We call each prototype of a spiral life cycle a "Build". Successive versions of modules are created as development progresses. The "baseline version" of a Build has all planned functionality implemented but not necessarily integrated and tested. The "ending version" is the one released for operational testing or the one accepted by the customer. Development of planned enhancements is done between the ending version of the prior

Build and the baseline version of the current Build.

For this case study, we selected the FORTRAN modules from a major subsystem of the final Build, accounting for 38% of FORTRAN modules in JSTARS (1,643 modules). Each module was a source file with one compilation unit, such as a subroutine.

The project uses Software Trouble Reports (STR) to track and control modifications to software. One of the attributes of an STR is its "Activity" code, which describes the reason a module was modified. Detailed activity codes were aggregated into the following reasons: FAULTS means the STR was due to developer errors; REQUIREMENT means the STR was due to unplanned requirements changes; PERFORMANCE means the STR was due to inadequate speed or capacity; and DOCUMENTATION means the STR was mandated during documentation changes. Other Activity codes were not related to software reliability.

Let the variable *Faults* be the number of FAULTS STRs that caused updates to a module's source code between the baseline version and the ending version of the Build. This is essentially the number of faults discovered during integration and testing. Table 5 summarizes the distribution of *Faults*. More than half the modules had no faults. The module at the 75^{th} percentile had two faults, and 16 faults was the maximum in a module.

After discussions with project engineers, a classification threshold of two faults was selected. Approximately one quarter of the modules were considered *fault-prone*. Another threshold might be appropriate for another project.

The cumulative numbers of STRs that affected code prior to the baseline version are

Table 5: Distributions of Process Variables

Percentile	*Faults*	*BaseFlts*	*BaseReq*	*BasePerf*	*BaseDoc*
100	16	36	3	5	3
99	9	14	2	2	2
95	6	8	1	1	1
90	4	6	0	1	1
75	2	3	0	0	0
50	0	1	0	0	0

attributes of a module's process history as follows.

$$BaseFlts = \text{Number of FAULTS STRs} \tag{11}$$

$$BaseReq = \text{Number of REQUIREMENT STRs} \tag{12}$$

$$BasePerf = \text{Number of PERFORMANCE STRs} \tag{13}$$

$$BaseDoc = \text{Number of DOCUMENTATION STRs} \tag{14}$$

Table 5 also summarizes the distributions of baseline STRs.

We defined categorical variables to model reuse from the prior Build.

$$IsNew = \begin{cases} 1 & \text{If module did not exist in ending version of prior Build} \\ 0 & \text{Otherwise} \end{cases} \tag{15}$$

Table 6: Distribution of Reuse Variables

	Number of Modules			
Age:	0	1	2	
IsNew:	1	0	0	Total
IsChg:				
1	311	418	354	1083
0	—	201	359	560
Total:	311	619	713	1643

$$IsChg = \begin{cases} 0 & \text{If no changed code since prior Build} \\ 1 & \text{Otherwise} \end{cases} \tag{16}$$

Since modules with a long history may be more reliable, we define the age of a module in terms of the number of Builds it has existed.

$$Age = \begin{cases} 0 & \text{If module is new} \\ 1 & \text{If module was new in the prior Build} \\ 2 & \text{Otherwise} \end{cases} \tag{17}$$

Our data did not include information on whether a module's age was more than two Builds. Table 6 shows the distribution of reuse variables.

The independent variables represent the history of each module, known at the time of the baseline version, namely, the cumulative number of STRs for each reason, and its reuse from previous Builds (*BaseFlts*, *BaseReq*, *BasePerf*, *BaseDoc*, *IsNew*, *IsChg*, *Age*).

In this study, we did not consider interactions among independent variables. This is a topic for further research.

BaseFlts, *IsNew*, *BaseReq*, *Age*, *BasePerf* were selected as s-significant independent variables using stepwise discriminant analysis at the $\alpha = 0.15$ s-significance level. Non-parametric discriminant analysis estimated the model based on the *fit* data set.

Using the generalized-classification rule, Table 7 shows the effect of various values of c on the accuracy of the multivariate model for each evaluation method. Figure 3 depicts the misclassification rates for the data-splitting columns in Table 7. The *fit* data set had 1,096 observations, consisting of 809 *not fault-prone* modules (base of Type I per cent) and 287 *fault-prone* modules (base of Type II per cent). The *test* data set had 547 observations, consisting of 404 *not fault-prone* modules (base of Type I per cent) and 143 *fault-prone* modules (base of Type II per cent).

Figure 4 depicts the associated effectiveness and efficiency in Table 8.

A value of $c = 1$ corresponded to the most-likely-class rule. This was not a practical rule, because it had $\Pr\{1|2\} = 72.73\%$ in the data-splitting column. This translated to an effectiveness of only 27.27% and an efficiency of 58.21%.

A value of $c = 0.35 = (287/809) = (\pi_2/\pi_1)$ corresponded to the maximum-correct-classifications rule. This was not a practical rule either, because it had $\Pr\{1|2\} = 79.72\%$ in the data-splitting column. This translated to an effectiveness of only 20.28% and an efficiency of 59.19%. In other words, neither the most-likely-class rule nor the maximum-correct-classifications rule was effective in identifying *fault-prone* modules.

Table 7: Misclassification Rates for Multivariate Model

Military System

Prior probabilities: proportions of fit data

		Misclassification rates (%)					
		Resubstitution		Cross-validation		Data Splitting	
		Pr{2|1}	Pr{1|2}	Pr{2|1}	Pr{1|2}	Pr{2|1}	Pr{1|2}
c	λ	Type I	Type II	Type I	Type II	Type I	Type II
0.35	0.005	0.00	79.44	4.45	81.88	4.95	79.72
1.00	0.20	1.98	66.20	5.32	73.17	6.93	72.73
2.00	0.50	6.06	59.23	8.65	61.67	8.91	65.73
3.00	0.60	21.76	30.31	23.61	33.10	25.74	39.86
4.00	0.80	25.09	27.18	26.82	28.92	27.48	33.57
5.00	1.00	30.66	25.09	31.03	25.09	31.19	27.97
6.00	1.20	31.03	25.09	31.03	25.09	31.44	27.97
7.00	1.00	35.72	20.21	35.72	20.21	33.91	21.68
8.00	2.30	64.03	5.23	64.03	5.23	64.36	8.39
10.00	2.00	79.73	1.05	79.73	1.05	78.96	1.40
15.00	2.00	100.00	0.00	100.00	0.00	100.00	0.00

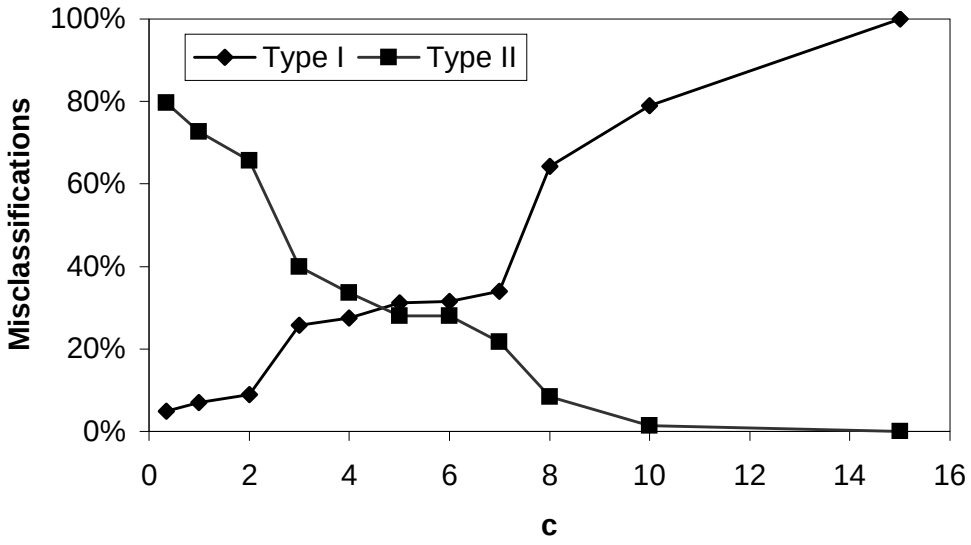

Figure 3: Misclassification Rates for Multivariate Model (data splitting)

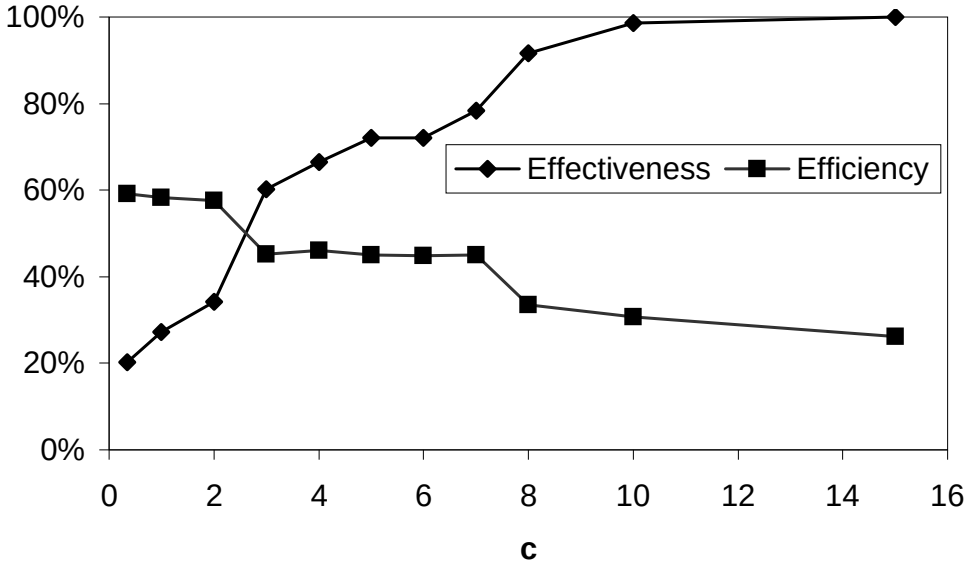

Figure 4: Effectiveness and Efficiency for Multivariate Model

Table 8: Effectiveness and Efficiency for Multivariate Model

Military System

	Test data set	
c	*effectiveness* (%)	*efficiency* (%)
0.35	20.28	59.19
1.00	27.27	58.21
2.00	34.27	57.65
3.00	60.14	45.27
4.00	66.43	46.11
5.00	72.03	44.98
6.00	72.03	44.78
7.00	78.32	44.98
8.00	91.61	33.50
10.00	98.60	30.65
15.00	100.00	26.14

A value of $c = 4.0$ had approximately equal cross-validation misclassification rates of $\Pr\{2|1\} = 26.82\%$ and $\Pr\{1|2\} = 28.92\%$, and corresponding data-splitting column rates of $\Pr\{2|1\} = 27.48\%$ and $\Pr\{1|2\} = 33.57\%$. This value of $c = 4.0 = (11.4)(0.35) = (C_{II}/C_I)(\pi_2/\pi_1)$ corresponded to the minimum-expected-cost rule at a cost ratio of

$C_{II}/C_I = 11.4$. For the *test* data set, this translated to an effectiveness of 66.43% and an efficiency of 46.11%.

Like the first case study, if the value of c was extreme, the model was not useful. Although this model can be improved by additional variables, it is sufficient to illustrate how the appropriate value of c can make the most of the available data. In this case study, if one chose an appropriate value of c, then the model could be practical for targeting additional reviews and testing during integration.

Acknowledgments

We thank Rama Munikoti, Kalai Kalaichelvan, and Robert Halstead for their encouragement and support. We thank Nishith Goel, John P. Hudepohl, Jean Mayrand, Gary Trio, and Ronald Flass for providing the data for the case studies and for helpful discussions regarding the case study systems. We thank the anonymous reviewers for their thoughtful comments. This work was supported in part by a grants from Nortel and Northrop Grumman. The findings and opinions in this paper belong solely to the authors, and are not necessarily those of the sponsors. Moreover, our results do not in any way reflect the quality of the sponsors' software products.

References

[1] M. R. Lyu. Introduction. In M. R. Lyu, editor, *Handbook of Software Reliability*

Engineering, chapter 1, pages 3–25. McGraw-Hill, New York, 1996.

[2] P. G. Frankl, D. Hamlet, B. Littlewood, and L. Strigini. Evaluating testing methods by delivered reliability. *IEEE Transactions on Software Engineering*, 24(8):586–601, August 1998. See [28].

[3] T. M. Khoshgoftaar and E. B. Allen. Multivariate assessment of complex software systems: A comparative study. In *Proceedings of the First International Conference on Engineering of Complex Computer Systems*, pages 389–396, Fort Lauderdale, FL, November 1995. IEEE Computer Society.

[4] F. Lanubile. Why software reliability predictions fail. *IEEE Software*, 13(4):131–132,137, July 1996.

[5] N. F. Schneidewind. Minimizing risk in applying metrics on multiple projects. In *Proceedings Third International Symposium on Software Reliability Engineering*, pages 173–182, Research Triangle Park, NC USA, October 1992. IEEE Computer Society.

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

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

[8] S. S. Gokhale and M. 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.

[9] J. C. Munson and T. M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Transactions on Software Engineering*, 18(5):423–433, May 1992.

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

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

[12] M. A. Vouk and K. C. Tai. Some issues in multi-phase software reliability modeling. In *Proceedings of the CASCON*, pages 513–523, Toronto, ON Canada, October 1993. National Research Council of Canada.

[13] 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, January 1996.

[14] D. L. Lanning and T. M. Khoshgoftaar. Fault severity in models of fault-correction activity. *IEEE Transactions on Reliability*, 44(4):666–671, September 1995.

[15] T. M. Khoshgoftaar and E. B. Allen. The impact of costs of misclassification on software quality modeling. In *Proceedings of the Fourth International Software Metrics Symposium*, pages 54–62, Albuquerque, NM USA, November 1997. IEEE Computer Society.

[16] T. M. Khoshgoftaar and E. B. Allen. Classification of fault-prone software modules: Prior probabilities, costs, and model evaluation. *Empirical Software Engineering: An International Journal*, 3(3):275–298, September 1998.

[17] R. A. Johnson and D. W. Wichern. *Applied Multivariate Statistical Analysis*. Prentice Hall, Englewood Cliffs, NJ, 3d edition, 1992.

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

[19] G. Le Gall, M. F. Adam, H. Derriennic, B. Moreau, and N. Valette. Studies on measuring software. *IEEE Journal of Selected Areas in Communications*, 8(2):234–245, February 1990.

[20] W. R. Dillon and M. Goldstein. *Multivariate Analysis: Methods and Applications*. John Wiley & Sons, New York, 1984.

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

[22] P. A. Lachenbruch and M. R. Mickey. Estimation of error rates in discriminant analysis. *Technometrics*, 10(1):1–11, February 1968.

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

[24] T. M. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan, and N. Goel. The impact of software evolution and reuse on software quality. *Empirical Software Engineering: An International Journal*, 1(1):31–44, 1996.

[25] T. M. Khoshgoftaar, E. B. Allen, R. Halstead, and G. P. Trio. Detection of fault-prone software modules during a spiral life cycle. In *Proceedings of the International Conference on Software Maintenance*, pages 69–76, Monterey, CA, November 1996. IEEE Computer Society.

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

[27] B. W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, May 1988.

[28] P. Frankl, D. Hamlet, B. Littlewood, and L. Strigini. Correction to: Evaluating testing methods by delivered reliability. *IEEE Transactions on Software Engineering*, 25(2):286, March 1999. See [2].

Appendices

A Model Selection

We use *stepwise* model selection at a s-significance level, α, to choose the independent variables in the nonparametric discriminant model [18]. The candidate variables are entered into the model in an incremental manner, based on an F test from analysis of variance which is recomputed for each change in the current model. Beginning with no variables in the model, the variable not already in the model with the best s-significance level is added to the model, as long as its s-significance is better than the threshold (α). Then the variable already in the model with the worst s-significance level is removed from the model as long as its s-significance is worse than the threshold (α). These steps are repeated until no variable can be added to the model.

B Nonparametric Density Estimation

We estimate a probability density function based on the *fit* data set. In addition to the notation at the beginning of this paper, let $\mathbf{S}_k$ be the covariance matrix for all samples

in G_k, and let $|\mathbf{S}_k|$ be its determinant. Let $K_k(\mathbf{u}|\mathbf{v},\lambda)$ be a multivariate kernel function on vector $\mathbf{u}$ with modes at $\mathbf{v}$. We select the normal kernel.

$$K_k(\mathbf{u}|\mathbf{v},\lambda) = (2\,\pi\lambda^2)^{-n_k/2}|\mathbf{S}_k|^{-1/2}\exp(\,(-1/2\lambda^2)(\mathbf{u}-\mathbf{v})'\mathbf{S}_k^{-1}(\mathbf{u}-\mathbf{v})\,) \tag{18}$$

Let $\mathbf{x}_{kl}, l = 1,\ldots,n_k$ represent a module in group G_k.

The estimated density function is given by the multivariate kernel density estimation technique [18].

$$\hat{f}_k(\mathbf{x}_i|\lambda) = \frac{1}{n_k}\sum_{l=1}^{n_k} K_k(\mathbf{x}_i|\mathbf{x}_{kl},\lambda) \tag{19}$$

This technique does not make assumptions about the functional form of the density function.

Dr. Taghi M. Khoshgoftaar: Dept. of Computer Science and Engineering; Florida Atlantic University; Boca Raton, Florida 33431 USA. Internet (e-mail): taghi@cse.fau.edu

Taghi M. Khoshgoftaar is a professor of the Dept. of Computer Science and Engineering, Florida Atlantic University, and is also the Director of the Empirical Software Engineering Laboratory. His research interests are in software engineering, software complexity metrics and measurements, software reliability and quality engineering, computational intelligence, computer performance evaluation, multimedia systems, and statistical modeling. He has published more than 150 refereed papers in these areas. He is a member of the Association for Computing Machinery, the American Statistical Association, and the IEEE (Computer Society and Reliability Society). He has served as North American editor of the *Software Quality Journal*.

Dr. Edward B. Allen: Dept. of Computer Science and Engineering; Florida Atlantic University; Boca Raton, Florida 33431 USA. Internet (e-mail): edward.allen@computer.org

Edward B. Allen received a B.S. from Brown University in 1971, an M.S. from the University of Pennsylvania in 1973, and a Ph.D. from Florida Atlantic University in 1995. He is currently a Research Associate in the Department of Computer Science and Engineering at Florida Atlantic University. His research interests include software measurement, software process modeling, software quality, and computer performance modeling. He has more than 40 refereed publications in these areas. He is a member of the IEEE Computer Society and the Association for Computing Machinery.