Classification of Fault-Prone Software Modules

Prior Probabilities, Costs, and Model Evaluation

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

Abstract. Software quality models can give timely predictions of reliability indicators, for targeting software improvement efforts. In some cases, classification techniques are sufficient for useful software quality models.

The software engineering community has not applied informed prior probabilities widely to software quality classification modeling studies. Moreover, even though costs are of paramount concern to software managers, costs of misclassification have received little attention in the software engineering literature. This paper applies informed prior probabilities and costs of misclassification to software quality classification. We also discuss the advantages and limitations of several statistical methods for evaluating the accuracy of software quality classification models.

We conducted two full-scale industrial case studies which integrated these concepts with nonparametric discriminant analysis to illustrate how they can be used by a classification technique. The case studies supported our hypothesis that classification models of software quality can benefit by considering informed prior probabilities and by minimizing the expected cost of misclassifications. The case studies also illustrated the advantages and limitations of resubstitution, cross-validation, and data splitting for model evaluation.

Key words: software quality modeling, fault-prone, nonparametric discriminant analysis, cost, misclassification, resubstitution, cross-validation, data splitting

1. Introduction

Detecting faults early is a high priority during software development. Software quality models are quantitative tools for improving fault detection processes. Such models give timely predictions of reliability indicators, on a module by module basis, which enable one to target reliability improvement techniques more effectively and efficiently. Some reliability improvement strategies only need an assessment of whether each module has a high risk of faults. Predicting the number of faults is not really needed. If a module is predicted to be in the fault-prone class, then one expects an excessive number of faults to be discovered subsequently. The definition of "excessive" is project-specific, according to the project's reliability improvement strategy. In such situations, a classification model is appropriate. A variety of techniques are applicable to classification models, such as discriminant analysis (Munson and

Khoshgoftaar, 1992), logistic regression (Basili et al., 1996; Khoshgoftaar and Allen, 1997a), classification trees (Khoshgoftaar et al., 1996a; Selby and Porter, 1988), discriminant power (Schneidewind, 1995), Optimal Set Reduction (Briand et al., 1993), neural networks (Khoshgoftaar and Lanning, 1995), and fuzzy classification (Ebert, 1996).

A classification model includes a "classification rule". When it is applied to a module, the rule predicts the class of the module based on the independent variables. A Bayesian approach to classification modeling (Johnson and Wichern, 1992; Seber, 1984) estimates a "posterior probability" of class membership which may depend on "prior probabilities" of class membership. Prior probabilities may be "informed" by beliefs or data that the analyst has prior to building a model, or may be "uniform" when all classes are assumed to be equally likely.

The software engineering community has not applied informed prior probabilities widely to software quality modeling studies. Many studies in the literature ignore prior probabilities or do not have enough information to estimate them (Basili et al., 1996; Munson and Khoshgoftaar, 1992; Schneidewind, 1992; Selby and Porter, 1988). This is equivalent to assuming all prior probabilities are equal. This paper presents how to use informed priors.

We consider two kinds of misclassifications: misclassification of *not fault-prone* modules, and misclassification of *fault-prone* modules. The practical impact of acting on incorrect predictions is quite different for each type, and therefore, a classification rule that just minimizes the probability of misclassification may not be satisfactory. Costs of misclassification can be incorporated into the same statistical framework as prior probabilities, resulting in a more sophisticated classification rule that minimizes the expected cost (Johnson and Wichern, 1992; Seber, 1984). Ignoring costs is equivalent to assuming equal cost for each kind of misclassification.

Even though costs are of paramount concern to software managers, costs of misclassification have received little attention in the software engineering literature. Khoshgoftaar, Lanning and Pandya (Khoshgoftaar et al., 1994) gave a rationale for considering costs of misclassification, but did not include them in the modeling. Schneidewind empirically analyzed the cost-sensitivity of the discriminative power technique for one software metric in an example (Schneidewind, 1992). This paper expands on preliminary results reported elsewhere (Khoshgoftaar and Allen, 1997b), incorporating costs into a software quality model and giving examples of its effects.

We have observed that a wide variety of methods are used in the software engineering literature to evaluate the accuracy of software quality models. Moreover, terminology for these methods is not standard. We distinguish between the data set that is used to estimate parameters of a model, the *fit* data set, and the data set used to evaluate its accuracy, the *test* data set. Some methods use the same data for both purposes. In this paper, we discuss the advantages and limitations of a range of statistical methods, and apply three major techniques in each of two case studies.

Our hypothesis is that classification models of software quality can benefit by considering prior probabilities and by minimizing the cost of misclassifications. We conducted two full-scale industrial case studies for empirical evidence. In the case studies, we integrate these concepts with nonparametric discriminant analysis to illustrate how they can be used by a classification technique. Each model predicted whether each module would be considered fault-prone. Such predictions could be used to adjust the development process dynamically. Review, integration, and testing resources could be focused on high risk parts of the system, compared to a standard process model.

The first case study examined a large sample of modules from a very large telecommunications system which is a mature system with formal releases enhancing the system from time to time (Khoshgoftaar et al., 1996b; Khoshgoftaar et al., 1996c). *Fault-prone* was defined by a threshold on the total number of faults discovered for the current release. The independent variables were indicators of reuse from the prior release and software design metrics.

The second case study was based on data from a large subsystem of a real-time tactical military system, the Joint Surveillance Target Attack Radar System, JSTARS (Khoshgoftaar and Allen, 1997b; Khoshgoftaar et al., 1998). *Fault-prone* was defined by a threshold on the number of faults discovered during integration and testing. The independent variables were indicators of reuse from prior prototypes and metrics of development process prior to integration.

The contribution of this paper is empirical evidence, supported by theoretical considerations, that software quality classification models can significantly benefit from including informed prior probabilities and costs of misclassifications. We also present advantages and limitations of several model evaluation methods. In particular, this paper is one of the first to apply classic cross-validation to software quality modeling. We illustrate the principles with two case studies which used nonparametric discriminant analysis. We observed similar phenomena in two different kinds of projects in different companies. Other classification techniques may be the subject of future research. The remainder of this paper consists of details on prior probabilities, on costs of misclassification, and on model evaluation methods, followed by presentation of case studies and conclusions.

2. Prior Probabilities

In classification models, the dependent variable is categorical. Here, we use a variable that has only two possible values: the observation is a member of one group or the other, G_1 or G_2. For example, let G_1 be *not fault-prone* modules and G_2 be *fault-prone* modules. The principles discussed here can be extended to more than two groups, such as risk levels, "red", "yellow", and "green" (Szabo and Khoshgoftaar, 1995). In software quality modeling, we usually define an "observation" to be a module. Let n_k be the number of observations in group $G_k, k = 1, 2$. Let $\mathbf{x}_i$ be the vector of the i^{th} observation's independent variables, and let $Class(\mathbf{x}_i)$ be the group to which it belongs.

A good classification rule should result in few misclassifications. In other words, the probabilities of misclassification should be small. Let $f_k(\mathbf{x}_i)$ be a likelihood function, such as the multivariate probability density, indicating whether an observation, $\mathbf{x}_i$, is in G_k. A good decision rule for this situation is the following (Johnson and Wichern, 1992).

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

In general, these likelihood functions will overlap, and a classification rule cannot perfectly assign every $\mathbf{x}_i$ to its group.

In addition to the likelihood of class membership due to attributes represented by $\mathbf{x}_i$, an improved classification rule should also take into account the overall class proportions of the underlying population. Let π_k be the prior probability of membership in $G_k, k = 1, 2$.

Knowing prior probabilities, a classification rule should assign an observation, $\mathbf{x}_i$, to the group with the greater posterior probability of group membership (Seber, 1984), which is given by

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

The classification rule is

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

This is equivalent to a classification rule that minimizes the expected number of misclassifications (Johnson and Wichern, 1992), given by

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

When the fit data set is representative of the population, we choose the prior probability, π_k, to be the proportion of fit observations in G_k. Otherwise we choose the uniform prior, $\pi_k = 0.5$.

3. Costs of Misclassifications

For a dependent variable consisting of two classes, two kinds of misclassifications are possible. Type I errors misclassify modules that are actually in G_1 as in G_2. Type II errors misclassify modules that are actually in G_2 as in G_1. If a model classifies a module incorrectly, then processes that utilize the erroneous result will have costs. Alternatively, the cost of misclassification may be viewed as foregoing the benefits of a desirable correct classification. An optimal classification rule for software engineering should account for the costs of misclassification, because the costs are often quite different.

In software engineering, the cost for acting on each type of erroneous prediction will depend on the reliability improvement technique that uses model results. For example, suppose we apply additional reviews to modules identified as *fault-prone*. The cost of a Type I misclassification is to invest in additional reviews of a module that is actually *not fault-prone* and thus, one would expect to find few additional faults during the review, *i.e.*, a small return on investment. The cost of a Type II misclassification is the lost opportunity to review a *fault-prone* module and detect its faults earlier in development. We have observed that most of the faults are in a small portion of the modules, namely, the *fault-prone* group (Selby, 1990). Faults that might have been discovered during a review will end up being discovered later in the project. It is well known that the cost of discovering, diagnosing, and correcting faults late in development is much greater than when done early. Faults discovered during the operations phase are extremely expensive in many environments, such as telecommunications systems, due to operational consequences multiplied over many users at remote locations. In mission-critical applications, such as military systems, faults during operations are unacceptable.

Let C_I be the cost of a Type I misclassification, and let C_{II} be the cost of a Type II misclassification. Each cost is determined by the software engineering processes that utilize model results. When costs of misclassification are unknown or unimportant, we choose equal costs, $C_{II}/C_I = 1$. Let $\Pr(2|1)$ be the probability that the model classifies a module as G_2, given that it is actually in G_1. Define $\Pr(1|2)$ as the converse. 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 (Johnson and Wichern, 1992) is given by

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

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. Equation (6) implies that neither prior probabilities alone nor costs of misclassification alone are the key to a good classification rule; it is the product of the ratios that is important. This is fortunate, since it is sometimes easier to estimate the ratios' product than to estimate the quantities individually.

4. Evaluation of Models

The fit data set is used to estimate parameters of a model. The test data set is used to evaluate its accuracy. Both data sets consist of the actual class and values of the independent variables for each observation. Ideally, the test data set is an independent sample of observations.

Each of the methods described below (Dillon and Goldstein, 1984, p.392) evaluates the accuracy of a classification model by predicting the class of each observation in the test data set, and then calculating the Type I and Type II misclassification rates. However, each method uses a different test data set.

4.1. Resubstitution

The resubstitution method uses the fit data set as a test data set. Since the fit and test data sets are not at all independent, this is the least realistic of the methods discussed here.

After model parameters are estimated, the class of each observation in the fit data set is predicted. The misclassification rates characterize "model fit". This assessment of model accuracy is often overly optimistic.

4.2. Subsequent Project

The subsequent-project method uses one project as a fit data set and a subsequent similar project as a test data set. This is a realistic simulation of applying a software quality model in practice.

This method is sensitive to the degree of similarity between the projects. One must address the question, "Is it appropriate to model the observations in both projects as coming from the same population?" This same question must be answered when applying the model in practice, as well.

4.3. Data Splitting

The data-splitting method is sometimes called the "Holdout" method (Geisser, 1975). Fit and test data sets are derived from a single data set by impartially sampling from available observations. This is also a simulation of applying a model in practice.

Data splitting may be appropriate when data on a similar subsequent project is not available. The proportion of observations in each data set is chosen according to sample sizes needed for the various statistical techniques to be employed. Consequently, this method often requires large samples of available data.

Statistical similarity of the fit and test data set is assured by the partitioning method. If statistical variation in the partitioning is a concern, then a number of data set pairs can be generated by random resampling, and then analyzed (Khoshgoftaar and Allen, 1997b).

A variation of the resampling approach partitions the available data into disjoint test data sets. For example, if each test data set has one tenth of the observations, then we generate ten models, using the remaining nine tenths as a fit data set. Gokhale and Lyu call the technique "10-fold cross-validation" (Gokhale and Lyu, 1997). Since the test data sets do not have observations in common, they are statistically independent.

4.4. Cross-Validation

Cross-validation is sometimes called the "U-Method" (Efron, 1983; Lachenbruch and Mickey, 1968). Suppose there are n observations available. 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.

In contrast to resubstitution, this does not have serious bias. This method is appropriate for smaller data sets than data splitting, but involves more computation per observation.

5. Case Study: Methodology

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

1. Collect configuration management data and problem reporting system data.

2. Select modules for analysis.

3. Measure or calculate *Faults* and independent variables, $\mathbf{x}_i$.

4. Determine the class of each module.

$$Class = \begin{cases} \textit{not fault-prone} & \text{If } Faults < threshold \\ \textit{fault-prone} & \text{If } Faults \geq threshold \end{cases} \tag{7}$$

   where *threshold* is chosen according to project-specific criteria.

5. Prepare *Fit* and *Test* data sets.

   For evaluation 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.

   Evaluation by resubstitution used the *Fit* data set as a test data set.

   Cross-validation used the *Fit* data set as the basis for evaluation.

6. Select significant independent variables using the *Fit* data set. Appendix B describes the stepwise discriminant analysis algorithm.

7. Estimate likelihood functions of the final model using a *Fit* data set. Appendix C gives mathematical details on nonparametric discriminant analysis. The smoothing parameter, λ, was chosen to optimize the cross-validation results.

   Nonparametric discriminant analysis estimates each within-class probability density as a function of the independent variables for each possible value of the dependent variable. We used the normal kernel method of nonparametric density estimation.

8. Predict dependent variable values for evaluation.

   Each model was evaluated by resubstitution and cross-validation using the *Fit* data set, and by data splitting using the *Test* data set. Validation tells us the level of accuracy to expect when applying the model to a similar project when the actual class membership is not known.

Table I. 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

Table II. Distribution of Faults

Modules	Total	Modules with Zero Faults	Fault-Prone	Faults Mean	Faults Std Dev
All modules	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

6. Case Study: A Telecommunications System

6.1. System Description

This case study examined a very large telecommunications system written by professional programmers in a large organization (Khoshgoftaar and Allen, 1995; Khoshgoftaar et al., 1996b; Khoshgoftaar et al., 1996c). This embedded computer application included numerous finite state machines and interfaces to other kinds of equipment. A random sample of 1,980 modules was taken for analysis. Resource constraints on data collection limited the number of modules in the sample. As shown in Table I, the sample represented about 1.3 million lines of code. It was written in a proprietary procedural high level language similar to Pascal. No macros were used.

In this study, we focused on the number of faults found in the current version, *Faults*, during the period from entry into the source code control system during the Coding phase until the end of data collection during the Operational phase. Table II shows the distribution of faults among modules. It also shows the distribution by reuse class of all modules, modules with zero faults, and *fault-prone* modules.

Due to limited resources for reliability enhancement, 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 $Faults \geq 5$. G_2

had about 12% of the modules. Another criterion for *fault-prone* group membership might be appropriate in other situations. We did not have cost information on this project, and thus, the actual cost ratio was not known.

We modeled reuse from the prior version with two binary variables. New modules were created during development of the current version.

$$IsNew = \begin{cases} 1 & \text{If module did not exist in the prior version} \\ 0 & \text{Otherwise} \end{cases} \tag{8}$$

Preexisting modules with some changed code were reused with modifications. If a module had no code changed during development of the current version, then it was reused as an object.

$$IsChg = \begin{cases} 0 & \text{If not changed since prior version} \\ 1 & \text{Otherwise} \end{cases} \tag{9}$$

Various product metrics were collected from source code at the module level. A module may consist of many files, and may have multiple entry and exit points. The average size of a module was 12 files; the median was 4 files. Since we were interested in predicting whether a module is fault-prone as early as possible in the development life cycle, for this study, we selected design product metrics derived from a call graph or a control flow graph that could be collected from design documentation at an early stage, and then reconfirmed from code later. Table III lists the metrics used in this study. Like other software metrics studies, we measured attributes of the detailed design that are related to size, complexity, and interrelationships.

6.2. Single Metric Models

6.2.1. *Informed Priors and Various Costs*

We built a nonparametric discriminant model, with the best-correlated single metric, MU, as the independent variable (Khoshgoftaar and Allen, 1995). The prior probabilities of group membership were the proportions of each group in the *Fit* data set, and misclassification costs were assumed to be equal. We then evaluated the model with the resubstitution, cross-validation and data splitting methods. Tables IV and V give the evaluation results. From a practical point of view, this model was clearly not satisfactory. The Type II misclassification rate was so high that one would not detect most of the fault-prone modules. For example, for data splitting, the Type II error rate was 80%.

Table VI shows the effect of various cost ratios (C_{II}/C_I) on the accuracy of the single variable model. Recall that the cost ratio is determined by software engineering processes; it is not a modeling parameter. In each case, the smoothing parameter, λ, was chosen to optimize

Table III. Design Product Metrics

Symbol	Title/Description
Call Graph Metrics	
MU	Modules Used. The number of modules that this module uses, including itself.
TC	Total Calls. The number of calls to entry points.
UC	Unique Calls. The number of unique entry points called by this module.
Control Flow Graph Metrics	
$IFTH$	If-Then Conditional Arcs. The number of arcs that contain a predicate of a control structure, but are not loops.
LP	Loops. The number of arcs that contain a predicate of a loop construct.
NL	Nesting Level. The total nesting level of all arcs.
SPC	Span of Conditional Arcs. The total number of arcs located within the span of conditional arcs.
SPL	Span of Loops. The number of vertices plus the number of arcs within loop control structure spans.
VG	McCabe cyclomatic complexity. $$VG = \text{Arcs} - \text{Vertices} + \text{ entry points } + \text{ exit points}$$

the cross-validation results. In this series of models, resubstitution gave almost identical results as cross-validation, which were also similar to data splitting results. Figure 1 depicts the effect for data splitting in Table VI.

The *Fit* data set had 1320 observations used in resubstitution and cross-validation, consisting of 1161 *not fault-prone* observations (base of Type I per cent) and 159 *fault-prone* observations (base of Type II per cent). The *Test* data set had 660 observations used in data splitting validation, consisting of 580 *not fault-prone* observations (base of Type I per cent) and 80 *fault-prone* observations (base of Type II per cent).

We saw that a cost ratio of one did not give a useful model because the Type II misclassification rate was too high. Cost ratios over 15 also were not useful because the Type I misclassification rate was too high. In other words, most modules were considered fault-prone. We are not recommending single variable models in general, but in this case, if one chose a reliability improvement strategy with a cost ratio (C_{II}/C_I) between five and ten, then the model could be useful.

For example, suppose we adopted a reliability improvement strategy where we give extra reviews to each module suspected of being fault-prone, at a cost of one unit. If a module is actually not fault-prone the reviews are a waste of time. Also suppose that if a module is indeed fault-prone, that correcting problems uncovered by reviews will avoid

Table IV. Evaluation of Single Variable Model (*Fit* Data Set)

Telecommunications System
MU was independent variable
Prior probabilities: proportions of Fit data
Costs of misclassification: equal
Smoothing parameter: $\lambda = 0.005$

Resubstitution
Number of Observations/Percent

	Model		
Actual	G_1	G_2	Total
G_1	1157	4	1161
	99.7%	0.3%	100.0%
G_2	131	28	159
	82.4%	17.6%	100.0%
Total	1288	32	1320
Percent	97.6%	2.4%	100.0%
Prior	87.9%	12.1%	

Overall misclassification rate: 10.2%

Cross-validation
Number of Observations/Percent

	Model		
Actual	G_1	G_2	Total
G_1	1149	12	1161
	99.0%	1.0%	100.0%
G_2	137	22	159
	86.2%	13.8%	100.0%
Total	1286	34	1320
Percent	96.4%	3.6%	100.0%
Prior	87.9%	12.1%	

Overall misclassification rate: 11.3%

about nine units of debugging cost later in the life cycle for a net benefit of eight units. Thus, cost of a Type I misclassification is one unit, the net cost of a Type II misclassification is eight units of forfeited benefit, and the cost ratio is eight. In this situation, according to data splitting in Table VI, the Type I misclassification rate would be 24.31% and the Type II misclassification rate would be 26.25%.

Given a sample of 660 modules, a perfect model would cost 80 units for reviews, yielding 720 units in avoided debugging costs. When we compare the single variable model to a perfect model, the expected cost

Table V. Evaluation of Single Variable Model (*Test* Data Set)

Telecommunications System
MU was independent variable
Prior probabilities: proportions of Fit data
Costs of misclassification: equal
Smoothing parameter: $\lambda = 0.005$

Data Splitting
Number of Observations/Percent

	Model		
Actual	G_1	G_2	Total
G_1	572	8	580
	98.6%	1.4%	100.0%
G_2	64	16	80
	80.0%	20.0%	100.0%
Total	636	24	660
Percent	96.4%	3.6%	100.0%
Prior	87.9%	12.1%	

Overall misclassification rate: 10.9%

Table VI. Effect of Cost Ratio on Single Variable Model

Telecommunications System
MU was independent variable
Prior probabilities: proportions of Fit data

Misclassification rates

		Resubstitution		Cross-validation		Data Splitting	
C_{II}/C_I	λ	Type I	Type II	Type I	Type II	Type I	Type II
1	0.005	0.34%	82.39%	1.03%	86.16%	1.38%	80.00%
2	0.50	3.45%	74.84%	3.45%	76.10%	3.45%	67.50%
3	0.50	6.72%	62.26%	6.72%	62.26%	6.90%	56.25%
4	0.45	11.11%	50.94%	11.11%	50.94%	12.76%	46.25%
5	0.40	14.64%	44.65%	14.64%	44.65%	16.72%	37.50%
6	0.45	16.97%	39.62%	16.97%	39.62%	19.48%	33.75%
7	0.35	21.53%	35.22%	21.53%	35.22%	24.31%	26.25%
8	0.45	21.53%	35.22%	21.53%	35.22%	24.31%	26.25%
9	0.40	25.15%	32.70%	25.15%	32.70%	28.62%	23.75%
10	0.20	35.14%	18.87%	35.23%	18.87%	39.48%	8.75%
15	0.10	43.67%	12.58%	43.67%	12.58%	46.90%	6.25%
20	0.30	62.19%	6.29%	62.19%	6.29%	63.10%	1.25%
30	0.30	78.12%	1.26%	78.12%	1.26%	78.97%	0.00%
50	0.10	81.22%	1.26%	81.31%	1.26%	81.90%	0.00%
75	0.10	85.01%	1.26%	85.01%	1.26%	85.17%	0.00%
100	0.20	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%

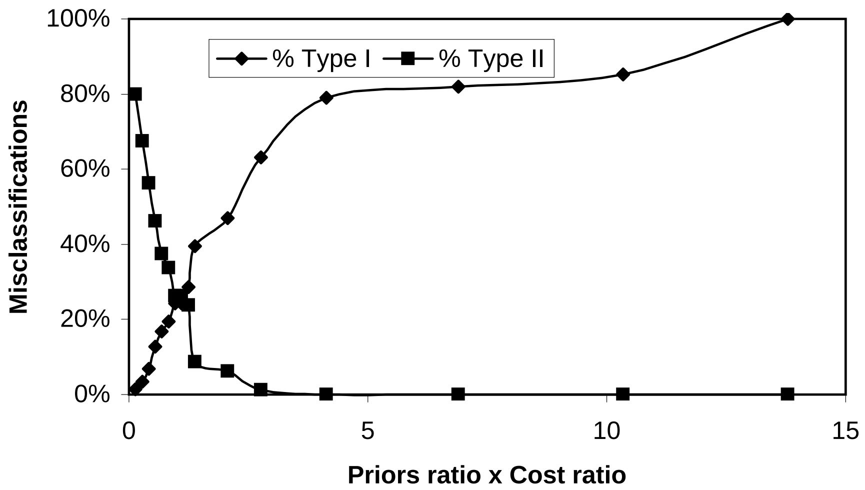

Figure 1. Effect of Cost Ratio on Single Variable Model

of misclassifications would be $309 = (580\times.2431)+(80\times0.2625\times(9-1))$. The cost of reviews would be $200 = (580 \times .2431) + (80 \times 0.7375)$. The benefit would be reliability improvement for $59 = 80 \times 0.7375$ fault-prone modules, which might avoid $531 = 59 \times 9$ units in debugging cost later. Most would consider this good return on investment $ROI = 531 : 200$.

6.2.2. *Uniform Priors and Equal Costs*

Since the priors ratio was $0.137 = 159/1161$, a net cost ratio of 7.3 happened to be equivalent to a classification rule with uniform priors and equal costs. Thus, for this system, a model with uniform priors and equal costs would give similar results as discussed above, with Type I error of 24.31% and Type II error of 26.25%. The second case study below illustrates that such good results are not guaranteed for uniform priors and equal costs.

6.3. Multivariate Model

Even though a single metric model may be useful, it is not necessarily the preferred choice. We found that a multivariate model based on the same data had better accuracy than the best single metric model above (Khoshgoftaar et al., 1996b).

Since some of the software product metrics were highly correlated, we applied principal components analysis. Principal components anal-

Table VII. Domain Pattern

Metric	Domain 1	Domain 2	Domain 3
SPL	**0.901**	0.359	0.137
LP	**0.880**	0.370	0.134
SPC	**0.719**	0.545	0.316
NL	**0.683**	0.593	0.334
TC	0.359	**0.864**	0.216
UC	0.426	**0.830**	0.245
VG	0.597	**0.724**	0.309
IFTH	0.599	**0.681**	0.357
MU	0.177	0.265	**0.939**
Eigenvalues	3.63	3.41	1.46
% Variance	40.3%	37.9%	16.2%
Cumulative	40.3%	78.1%	94.4%

ysis of the standardized product metrics of all observations retained three components that accounted for 94.4% of the variance of the standardized metric data. Appendix A gives details on the calculations which transformed the nine product metrics into three "domain metrics". Table VII shows the relationship between the original metrics and the domain metrics. Each table entry is the correlation between the column and row metrics; the largest in each row is shown bold. This indicates the degree that each product metric contributed to each domain metric.

The sample data set was split into a *Fit* data set of two thirds of the modules (1,320 observations), and a *Test* data set of one third of the modules (660 observations). The standardized transformation matrix that resulted from principal components analysis was separately applied to the *Fit* and *Test* standardized design product metrics, producing independent *Fit* and *Test* data sets of domain metrics.

The stepwise discriminant model selection process selected both reuse covariates, $ISNEW$ and $ISCHG$, and all three domain metrics, at the 5% significance level.

For this model, the prior probabilities of group membership were the proportions of each group in the *Fit* data set. Misclassification costs were assumed to be equal. We empirically chose a smoothing parameter of $\lambda = 0.05$ to fit the nonparametric discriminant model. Tables VIII and IX give the evaluation results of resubstitution, cross-validation, and data splitting.

Table VIII. Evaluation of Multivariate Model (*Fit* Data Set)

Telecommunications System
Prior probabilities: proportions of Fit data
Costs of misclassification: equal
Smoothing parameter: $\lambda = 0.05$

Resubstitution
Number of Observations/Percent

	Model		
Actual	G_1	G_2	Total
G_1	1161	0	1161
	100.0%	0.0%	100.0
G_2	15	144	159
	9.4%	90.6%	100.0
Total	1176	144	1320
Percent	89.1%	10.9%	100.0
Prior	87.9%	12.1%	

Overall misclassification rate: 1.1%

Cross-validation
Number of Observations/Percent

	Model		
Actual	G_1	G_2	Total
G_1	910	251	1161
	78.4%	21.6%	100.0
G_2	39	120	159
	24.5%	75.5%	100.0
Total	949	371	1320
Percent	71.9%	28.1%	100.0
Prior	87.9%	12.1%	

Overall misclassification rate: 22.0%

Resubstitution gave a much too optimistic evaluation. This illustrates the danger of relying on resubstitution alone to evaluate software quality models.

The cross-validation and data splitting evaluations showed that this model had better accuracy than the best single metric model above. For example, based on data splitting, we expect the model to predict that 31.4% of the modules are *fault-prone*. If extra effort is invested in these modules to improve their reliability, then the fraction of fault-prone modules could be reduced from about 12.1% to about 1.66% $(0.0166 = 0.121 \times 0.1375)$ of all the modules.

Table IX. Evaluation of Multivariate Model (*Test* Data Set)

Telecommunications System
Prior probabilities: proportions of Fit data
Costs of misclassification: equal
Smoothing parameter: $\lambda = 0.05$

Data Splitting
Number of Observations/Percent

	Model		
Actual	G_1	G_2	Total
G_1	442	138	580
	76.2%	23.8%	100.0%
G_2	11	69	80
	13.75%	86.25%	100.0%
Total	453	207	660
Percent	68.6%	31.4%	100.0%
Prior	87.9%	12.1%	

Overall misclassification rate: 22.6%

7. Case Study: A Military System

A second case study gives perspective on general principles, so that we do not draw overly specific conclusions from just one example project.

7.1. System Description

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 (Khoshgoftaar and Allen, 1997b; Khoshgoftaar et al., 1998). 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. During Operation Desert Storm in 1991, two prototypes gave air and ground commanders a real-time tactical view of the battlefield every day of the war. The system was developed under the spiral life cycle model (Boehm, 1988). Enhancements are currently being implemented.

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.

Only modules that existed in both the baseline and ending versions were selected. A source code analysis tool determined whether declarations or executable statements were changed from one version to the next. Changes to comments were not considered.

The project uses Software Trouble Reports (STR) to track and control modifications to software, documents, and other software development objects. STRs are generated after unit testing as problems are discovered by reviews, integration, and testing. When a problem is fixed, the module version of the corrected source file is recorded in the STR.

One of the attributes of an STR is its "Activity" code. This describes the reason a module was modified. Detailed activity codes were aggregated into the following reasons. FAULTS means code changes due to developer errors; REQUIREMENT means code changes due to unplanned requirements changes; PERFORMANCE means code changes due to inadequate speed or capacity; and DOCUMENTATION means code changes mandated during documentation changes. Other Activity codes were not related to software reliability. Our study included only those STRs that resulted in modification to declarations or executable statements of the code.

Let *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 X summarizes the distribution of *Faults*.

After discussions with project engineers, a classification threshold of two faults was selected, to limit the *fault-prone* group to practical proportions. In this case, they considered approximately one quarter of the modules to be *fault-prone*. Another threshold might be appropriate for another project. We did not have cost data on this project, and thus, the actual cost ratio was not known.

The cumulative numbers of STRs that affected code prior to the baseline version are attributes of a module's process history as follows.

$$BaseFlts = \text{Number of FAULTS STRs} \tag{10}$$

$$BaseReq = \text{Number of REQUIREMENT STRs} \tag{11}$$

Table X. 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

$$BasePerf = \text{Number of PERFORMANCE STRs} \tag{12}$$

$$BaseDoc = \text{Number of DOCUMENTATION STRs} \tag{13}$$

Table X also summarizes the distributions of baseline STRs.

A fault may be caused by various kinds of human mistakes. Each STR's reason entails different kinds of mental processes to implement a change to code. Diagnosis and correction of a fault occurs in the context of familiar requirements, specifications and designs. Analysis of new or changed requirements entails creating a new design that balances minimal disruption to the existing design, conformance to requirements, and flexibility for the future. Improving performance involves an analysis of run-time behavior, rather than functionality. Diagnosing and correcting implementation discrepancies discovered during documentation revisions requires attention to details. Thus, these baseline variables are related to various aspects of software engineering which may affect *Faults*.

Similar to the telecommunication system case study, we defined categorical variables to model reuse from the prior Build.

$$IsNew = \begin{cases} 1 & \text{If module did not exist in prior Build} \\ 0 & \text{Otherwise} \end{cases} \tag{14}$$

$$IsChg = \begin{cases} 0 & \text{If no changed code since prior Build} \\ 1 & \text{Otherwise} \end{cases} \tag{15}$$

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.

$$Age = \begin{cases} 0 & \text{If module is new} \\ 1 & \text{If module was new in the prior Build} \\ 2 & \text{Otherwise} \end{cases} \tag{16}$$

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

Table XI. 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

7.2. Multivariate Model

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.

Nonparametric discriminant analysis estimated the models in this section with the same independent variables based on the *Fit* data set. *BaseFlts*, *IsNew*, *BaseReq*, *Age*, *BasePerf* were selected as independent variables using stepwise discriminant analysis at the $\alpha = 0.15$ significance level. Variables not selected were not considered in estimating the density functions.

BaseFlts was the first variable selected, implying that it was the independent variable most closely related to *Class*. This may be due to several factors. For example, code changed in fixing faults are opportunities for more mistakes. Moreover, the underlying software attributes that cause human errors are probably not removed by fixing bugs. *BaseDoc* was not included in the model. In other words, changes to code resulting from documentation reviews were not related to faults discovered during integration and testing.

7.2.1. *Informed Priors and Various Costs*

Once the model has been estimated, one can input values for the independent variables of a module to estimate its density values, $\hat{f}_k(\mathbf{x}_i), k = 1, 2$, and then apply the decision rule in Equations (6) to classify it. In this case study, we let the proportion of each class in the *Fit* data set be the prior probability for that class, π_k, and we investigated a range of cost ratios.

Table XII. Effect of Cost Ratio on a Multivariate Model

Military System
Prior probabilities: proportions of Fit data

Misclassification rates

		Resubstitution		Cross-validation		Data Splitting	
C_{II}/C_I	λ	Type I	Type II	Type I	Type II	Type I	Type II
1	0.005	0.00%	79.44%	4.45%	81.88%	4.95%	79.72%
2	0.001	0.25%	77.00%	4.70%	81.88%	5.20%	79.02%
3	0.20	1.98%	66.20%	5.93%	72.47%	6.93%	72.73%
4	0.30	4.08%	62.02%	8.28%	67.94%	7.92%	68.53%
5	0.40	5.93%	59.23%	8.16%	62.37%	8.91%	66.43%
6	0.50	6.30%	58.54%	8.78%	61.67%	8.91%	65.73%
7	0.40	11.50%	47.74%	15.33%	52.26%	12.62%	58.04%
8	0.55	20.64%	31.01%	22.87%	33.45%	24.01%	41.26%
9	0.55	22.50%	29.27%	23.86%	32.40%	25.99%	37.06%
10	1.00	26.45%	29.97%	26.45%	31.01%	28.22%	33.57%
15	1.30	31.03%	25.09%	31.03%	25.09%	31.44%	27.97%
20	1.00	35.72%	20.21%	35.72%	20.21%	33.91%	21.68%
30	1.90	79.73%	1.05%	79.73%	1.05%	78.96%	1.40%
50	1.40	100.00%	0.00%	100.00%	0.00%	100.00%	0.00%

Table XII shows the effect of various cost ratios (C_{II}/C_I) on the accuracy of the multivariate model for each validation method. In each case, the smoothing parameter, λ, was chosen to optimize the cross-validation results. In this case study, resubstitution gave similar results as the other two methods. Figure 2 depicts the misclassification rates for data splitting in Table XII. The *Fit* data set had 1096 observations used in resubstitution and cross-validation, consisting of 809 *not fault-prone* observations (base of Type I per cent) and 287 *fault-prone* observations (base of Type II per cent). The *Test* data set had 547 observations used in data splitting validation, consisting of 404 *not fault-prone* observations (base of Type I per cent) and 143 *fault-prone* observations (base of Type II per cent).

Like the first case study, if the cost ratio was near one or extremely high, the model was not useful. In this case study, given informed priors, if one chose a reliability improvement strategy with a cost ratio between 10 and 20 then the model could be useful. Recall that software engineering processes determine the cost ratio.

7.2.2. *Uniform Priors and Equal Costs*

Since the informed priors ratio was $0.354 = 287/809$, uniform priors and equal costs happened to be equivalent to the informed priors ratio times

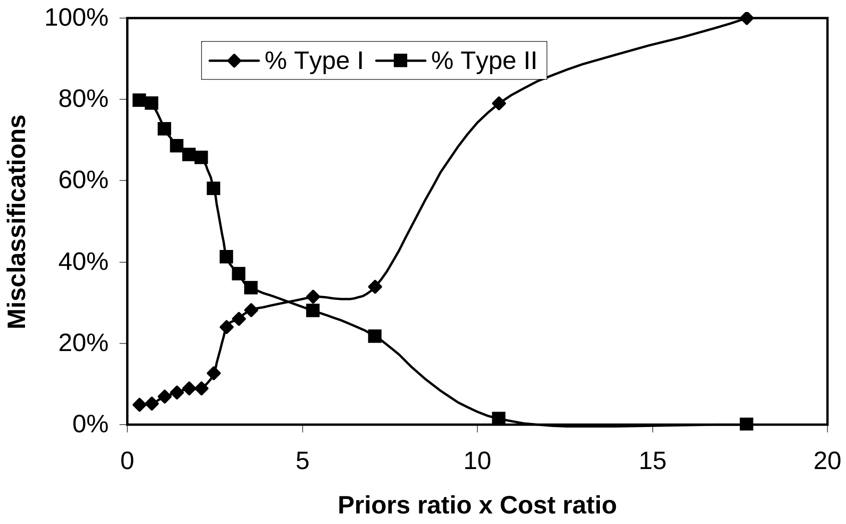

Figure 2. Effect of Cost Ratio on a Multivariate Model

a net cost ratio of 2.82. The data splitting columns in Table XII, show that a cost ratio of three had a Type I error rate of 6.93% and a Type II error rate of 72.73%. Thus, for this system, a model with uniform priors and equal costs would similarly have unacceptable results. In contrast to the first case study (Table VI), this illustrates that for some projects, the simple approach of uniform priors and equal costs may not be satisfactory.

8. Conclusions

Software quality models can give timely predictions of reliability indicators, on a module by module basis, which enable one to target reliability improvement efforts. Sometimes it is not necessary to predict the number of faults; predicting whether a module will be considered fault-prone may be enough. In such cases, classification techniques are appropriate for software quality models.

Under a Bayesian approach, a "posterior probability" of class membership may depend on "prior probabilities" of class membership. Prior probabilities may be "informed" by beliefs or data that the analyst has when building a model, or may be "uniform" when all classes are assumed to be equally likely. We use the proportion of observations in each class in the fit data set as an estimate of the prior probability of class membership. This paper presents how to use informed priors in a software quality classification model.

Costs of misclassification can be incorporated into the same statistical framework, resulting in a classification rule that minimizes the expected cost of misclassifications. This paper incorporates costs into a classification rule and gives examples of its effect on software quality models. Prior probabilities and costs of misclassification do not individually determine model accuracy; the product of the priors ratio and the costs ratio determine the classification rule. Each development environment/software application is likely to have its own functional relationship between model accuracy and this product.

In this paper, we discuss the advantages and limitations of various statistical methods for evaluating the accuracy of software quality classification models. We used resubstitution, cross-validation, and data splitting in each of two case studies. Resubstitution is often overly optimistic. Cross-validation is useful for small data sets, but involves more computation per observation. This paper is one of the first to apply classic cross-validation to software quality modeling. Data splitting is a convincing simulation of using a software quality model when a large data set is available, but data on multiple projects is not.

We conducted two full-scale industrial case studies for empirical evidence. In the case studies, we integrated these concepts with nonparametric discriminant analysis to illustrate how they can be used by a classification technique. Each model predicted whether each module would be considered fault-prone. Such predictions could be used to target reliability improvement efforts on high risk parts of the system, compared to a standard process model.

The first case study examined a large sample of modules from a very large telecommunications system. The second case study was based on data from a large subsystem of a real-time tactical military system, the Joint Surveillance Target Attack Radar System, JSTARS. The case studies supported our hypothesis that classification models of software quality can benefit by considering prior probabilities and by minimizing the cost of misclassifications. The case studies also illustrated the advantages and limitations of resubstitution, cross-validation, and data splitting. We observed similar phenomena in two different kinds of projects in different companies. We have observed the importance of prior probabilities and costs of misclassification in other projects as well.

We suggest that future work investigate ways to integrate informed priors and costs of misclassification into other software quality classification modeling techniques.

Acknowledgements

We thank Rama Munikoti, Kalai Kalaichelvan, and Robert Halstead for their encouragement and support. We thank Nishith Goel, Jean Mayrand, Gary Trio, and Ronald Flass 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 Technology 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

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

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

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

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

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

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

Geisser, S. (1975). The predictive sample reuse method with applications. *Journal of the American Statistical Association*, 70(350):320–328.

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

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

Khoshgoftaar, T. M. and Allen, E. B. (1995). 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. IEEE Computer Society.

Khoshgoftaar, T. M. and Allen, E. B. (1997a). Classification techniques for predicting software quality: Lessons learned. In *Proceedings of the Annual Oregon Workshop on Software Metrics*, Coeur d'Alene, ID, USA. University of Idaho.

Khoshgoftaar, T. M. and Allen, E. B. (1997b). 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. IEEE Computer Society.

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

Khoshgoftaar, T. M., Allen, E. B., Halstead, R., Trio, G. P., and Flass, R. (1998). Process measures for predicting software quality. *Computer*, 31(4).

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

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

Khoshgoftaar, T. M. and Lanning, D. L. (1995). A neural network approach for early detection of program modules having high risk in the maintenance phase. *Journal of Systems and Software*, 29(1):85–91.

Khoshgoftaar, T. M., Lanning, D. L., and Pandya, A. S. (1994). A comparative study of pattern recognition techniques for quality evaluation of telecommunications software. *IEEE Journal on Selected Areas in Communications*, 12(2):279–291.

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

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

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

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

Seber, G. A. F. (1984). *Multivariate Observations*. John Wiley and Sons, New York.

Selby, R. W. (1990). Empirically based analysis of failures in software systems. *IEEE Transactions on Reliability*, 39(4):444–454.

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

Szabo, R. M. and Khoshgoftaar, T. M. (1995). An assessment of software quality in a C++ environment. In *Proceedings of the Sixth International Symposium on Software Reliability Engineering*, pages 240–249, Toulouse, France. IEEE Computer Society.

Appendix

A. Principal Components Analysis

Software metrics have a variety of units of measure, which are not readily combined in a multivariate model. We transform all product metric variables, so that each standardized variable has a mean of zero and a variance of one. Thus, the unit of measure becomes one standard deviation.

Suppose we have m measurements on n modules. Let $\mathbf{Z}$ be the $n \times m$ matrix of standardized measurements where each row corresponds to a module and each column is a standardized variable. Our principal components are linear combinations of m standardized random variables, $Z_1, \ldots, Z_m$. The principal components represent the same data in a new coordinate system, where the variability is maximized in each direction and the principal components are uncorrelated (Seber, 1984). If the covariance matrix of $\mathbf{Z}$ is a real symmetric matrix with distinct roots, then one can calculate its eigenvalues, λ_j, and its eigenvectors, $\mathbf{e}_j, j = 1, \ldots, m$. Since the eigenvalues form a nonincreasing series, $\lambda_1 \geq \ldots \geq \lambda_m$, one can reduce the dimensionality of the data without significant loss of explained variance by considering only the first p components, $p \ll m$, according to some stopping rule, such as achieving a threshold of explained variance. For example, choose the minimum p such that $\sum_{j=1}^{p} \lambda_j / m \geq 0.90$ to achieve at least 90% of explained variance.

Let $\mathbf{T}$ be the $m \times p$ standardized transformation matrix whose columns, $\mathbf{t}_j$, are defined as

$$\mathbf{t}_j = \frac{\mathbf{e}_j}{\sqrt{\lambda_j}} \text{ for } j = 1, \ldots, p \qquad (17)$$

Let D_j be a principal component random variable, and let $\mathbf{D}$ be an $n \times p$ matrix with D_j values for each column, $j = 1, \ldots, p$.

$$D_j = \mathbf{Z}\mathbf{t}_j \qquad (18)$$

$$\mathbf{D} = \mathbf{Z}\mathbf{T} \qquad (19)$$

When the underlying data is software metric data, we call each D_j a *domain metric*.

B. Model Selection

We use *stepwise* model selection at a significance level, α, to choose the independent variables in the nonparametric discriminant model (Seber, 1984). 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 significance level is added to the model, as long as its significance is better than the threshold (α). Then the variable already in the model with the worst significance level is removed from the model as long as its significance is worse than the threshold (α). These steps are repeated until no variable can be added to the model.

C. Nonparametric Discriminant Analysis

We estimate a discriminant function based on the fit data set. Consider the following notation.

- $\mathbf{x}_i$ is the vector of the i^{th} module's independent variables.
- G_1 and G_2 are mutually exclusive groups.
- n_k is the number of observations in group $G_k, k = 1, 2$.
- π_k is the prior probability of membership in G_k, which we usually choose to be the proportion of *fit* observations in G_k.
- $f_k(\mathbf{x}_i)$ is the multivariate probability density giving the likelihood that a module represented by $\mathbf{x}_i$ is in G_k. Since the density functions, f_k, are not likely to conform to the normal distribution, we use nonparametric density estimation.
- $\hat{f}_k(\mathbf{x}_i|\lambda)$ is an approximation of $f_k(\mathbf{x}_i)$, where λ is a smoothing parameter in this context.
- $\mathbf{S}_k$ is the covariance matrix for all samples in G_k, and $|\mathbf{S}_k|$ is its determinant.
- $K_k(\mathbf{u}|\mathbf{v}, \lambda)$ is a multivariate kernel function on vector $\mathbf{u}$ with modes at $\mathbf{v}$. We select the normal kernel.

$$\begin{aligned} K_k(\mathbf{u}|\mathbf{v}, \lambda) = \; & (2\,\pi\lambda^2)^{-n_k/2}|\mathbf{S}_k|^{-1/2} \\ & \exp(\,(-1/2\lambda^2)(\mathbf{u}-\mathbf{v})'\mathbf{S}_k^{-1}(\mathbf{u}-\mathbf{v})\,) \end{aligned} \tag{20}$$

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 (Seber, 1984).

$$\hat{f}_k(\mathbf{x}_i|\lambda) = \frac{1}{n_k} \sum_{l=1}^{n_k} K_k(\mathbf{x}_i|\mathbf{x}_{kl}, \lambda) \tag{21}$$

Let C_I and C_{II} be the costs of Type I and Type II misclassifications, respectively. A classification rule that minimizes the expected cost of misclassification is given by

$$Class(\mathbf{x}_i) = \begin{cases} G_1 & \text{if } \frac{\hat{f}_1(\mathbf{x}_i|\lambda)}{\hat{f}_2(\mathbf{x}_i|\lambda)} \geq \left(\frac{C_{II}}{C_I}\right)\left(\frac{\pi_2}{\pi_1}\right) \\ G_2 & \text{otherwise} \end{cases} \tag{22}$$

Authors' Vitae

Taghi M. Khoshgoftaar
is a professor of the Dept. of Computer Science and Engineering, Florida Atlantic University, Boca Raton, Florida, USA. He is also the

Director of the Empirical Software Engineering Laboratory, established through a grant from the National Science Foundation. His research interests are in software engineering, software complexity metrics and measurements, software reliability and quality engineering, software testing, software maintenance, computational intelligence applications, computer performance evaluation, multimedia systems, and statistical modeling. He has published more than 100 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 on technical program committees of various international conferences, symposia, and workshops. He is Program co-Chair of the IEEE International Conference on Software Maintenance, 1998, and the General Chair of the IEEE International Symposium on Software Reliability Engineering, 1999. He was a Guest Editor of the IEEE *Computer* special issue on Metrics in Software, September 1994. He is on the editorial boards of the *Software Quality Journal* and the *Journal of Multimedia Tools and Applications*.

Edward B. Allen
received the B.S. degree in engineering from Brown University, Providence, RI, USA, in 1971, the M.S. degree in systems engineering from the University of Pennsylvania, Philadelphia, PA, USA, in 1973, and the Ph.D. degree in computer science from Florida Atlanatic University, Boca Raton, FL, USA, in 1995. He is currently a Research Associate and an adjunct professor in the Department of Computer Science and Engineering at Florida Atlantic University. He began his career as a programmer with the U.S. Army. From 1974 to 1983, he performed systems engineering and software engineering on military systems, first for Planning Research Corp. and then for Sperry Corp. From 1983 to 1992, he developed corporate data processing systems for Glenbeigh, Inc., a specialty health care company. His research interests include software measurement, software process modeling, and software quality. He is a member of the IEEE Computer Society and the Association for Computing Machinery.

Address for correspondence: Taghi M. Khoshgoftaar, Dept. of Computer Science and Engineering, Florida Atlantic University, Boca Raton, FL 33431 USA, (561)297-3994, taghi@cse.fau.edu, http://www.cse.fau.edu/esel.html