Empirical Software Engineering, 4, 159–186 (1999)

A Comparative Study of Ordering and Classification of Fault-Prone Software Modules

TAGHI M. KHOSHGOFTAAR taghi@cse.fau.edu
Dept. of Computer Science and Engineering, Florida Atlantic University, Boca Raton, Florida 33431 USA

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

Abstract. Software quality models can predict the quality of modules early enough for cost-effective prevention of problems. For example, software product and process metrics can be the basis for predicting reliability. Predicting the exact number of faults is often not necessary; classification models can identify fault-prone modules. However, such models require that "fault-prone" be defined before modeling, usually via a threshold. This may not be practical due to uncertain limits on the amount of reliability-improvement effort. In such cases, predicting the rank-order of modules is more useful.

A module-order model predicts the rank-order of modules according to a quantitative quality factor, such as the number of faults. This paper demonstrates how module-order models can be used for classification, and compares them with statistical classification models.

Two case studies of full-scale industrial software systems compared nonparametric discriminant analysis with module-order models. One case study examined a military command, control, and communications system. The other studied a large legacy telecommunications system. We found that module-order models give management more flexible reliability enhancement strategies than classification models, and in these case studies, yielded more accurate results than corresponding discriminant models.

Keywords: Software reliability, fault-prone modules, software quality models, module-order models, multiple linear regression, nonparametric discriminant analysis, principal components analysis

1. Introduction

Various special techniques can be used in addition to normal development processes to improve the reliability of software modules. Examples include more rigorous design and code reviews, automatic test case generation to support more extensive testing, and strategic assignment of key personnel. Our goal is to target reliability enhancement activities to those modules that are most likely to have problems (Hudepohl et al., 1996).

Software product and process metrics can be measured substantially before reliability problems become evident. Prior research has shown that software product and process metrics (Fenton and Pfleeger, 1997) collected prior to the test phase can be the basis for reliability predictions (Khoshgoftaar et al., 1996c; Khoshgoftaar et al., 1997; Khoshgoftaar and Munson, 1990; Munson and Khoshgoftaar, 1996; Schneidewind, 1995). We want to enhance the quality of modules recommended by a model early enough to prevent problems from poor quality later in the life cycle.

Predicting the exact number of faults in each module is often not necessary; previous research has focused on classification models to identify *fault-prone* and *not fault-prone* modules (Briand et al., 1993; Ebert, 1996; Khoshgoftaar et al., 1996a; Khoshgoftaar et al., 1996b; Munson and Khoshgoftaar, 1992). Some prior research has considered criticality of faults as well (Ebert, 1997). Such models require that *fault-prone* be defined before modeling, usually via a threshold on the number of faults expected. However, due to uncertain resource constraints that limit the amount of reliability-improvement effort, software development managers often cannot choose an appropriate threshold at the time of modeling. In such cases, a prediction of the rank-order of modules, from the least to the most fault-prone, is more useful (Khoshgoftaar et al., 1994b; Ohlsson and Alberg, 1996). With a predicted rank-order in hand, one can select as many from the top of the list for reliability enhancement as resources will allow.

A *module-order model* predicts the rank-order of modules according to a quantitative quality factor, such as the number of faults. A module-order model has an underlying quantitative software quality model as the basis for predictions, without specifying *a priori* a threshold to define *fault-prone*. We evaluate module-order models from a project management perspective.

This paper summarizes the concept of module-order models for software reliability enhancement (Khoshgoftaar and Allen, 1998b), and demonstrates how module-order models can be used for classification, and compares them with software quality classification models. This is the first such comparative study that we know of.

Two case studies of full-scale industrial software systems compared module-order models with a representative classification technique, nonparametric discriminant analysis. The subject of one case study was a military command, control, and communications system (Khoshgoftaar et al., 1994a). The subject of the other was a large legacy telecommunications system (Khoshgoftaar et al., 1996a). We found that module-order models give management more flexible reliability-enhancement strategies than classification models, and in these case studies, yielded more accurate results than corresponding discriminant models. The two case studies had similar results, in spite of differing organizations, programming languages, software metrics, and life cycle phases.

The remainder of this paper summarizes module-order modeling, classification modeling, our case study methodology, results of two full-scale industrial case studies, and conclusions.

2. Module-Order Modeling

Most quality factors are directly measurable only after software has become operational, for example, the number of faults. In contrast, most software product and process metrics can be measured during development. A suitable software quality model can make predictions when it is not too late to take compensatory actions, for example, through reliability enhancement activities. Our case studies define software quality either in terms of the number of faults, or in terms of debug code churn, i.e., the amount of code changed due to repair of faults. In the following discussion, we use the number of faults as an example quality measure. The same method applies to other quality measures, such as debug code churn.

Our objective is to develop a model that will predict the relative quality of each module, especially those which are most fault-prone. In particular, we are interested in the order of modules according to the number of faults. Even though the number of faults is absolute scale when directly measured, our recommended use of a predicted value is only ordinal scale (Briand et al., 1996). Classification models, on the other hand, treat dependent variables as nominal scale, such as whether a module is *fault-prone* or not. A module-order model consists of the following components.

1. An underlying quantitative software quality model.
2. A ranking of modules according to a quality measure predicted by the underlying model.
3. A procedure for evaluating the accuracy of a model's ranking.

Suppose we have a quantitative model, $F_i = f(\mathbf{x}_i)$, where the number of faults, F_i, in module i is a function of its software measurements, the vector $\mathbf{x}_i$. Let $\widehat{F}(\mathbf{x}_i)$ be the estimate of F_i by a fitted model, $\hat{f}(\mathbf{x}_i)$. In this paper, we use a simple quantitative modeling technique, multiple linear regression, without resorting to sophisticated estimation techniques (Khoshgoftaar et al., 1992b). Other quantitative modeling techniques could be used, such as nonlinear regression (Khoshgoftaar et al., 1992a), regression trees (Gokhale and Lyu, 1997), or computational intelligence techniques (Briand et al., 1992; Ganesan et al., 1999; Khoshgoftaar et al., 1995; Leake, 1996). Future research will investigate the effects of such refinements.

A simple underlying quantitative modeling technique is an advantage for module-order modeling. Users of module-order models with familiar underlying modeling techniques can easily see that the results were derived in a reasonable manner. Consequently, the module-order modeling technique lends itself to user acceptance. Module-order models are not "black boxes".

Let R_i be the percentile rank of observation i in a perfect ranking of modules according to F_i. Let $\widehat{R}(\mathbf{x}_i)$ be the percentile rank of observation i in the predicted ranking according to $\widehat{F}(\mathbf{x}_i)$.

Spearman correlation between actual and predicted rankings is a conventional approach to evaluating the accuracy of a module-order model (Ohlsson et al., 1996; Shepperd and Ince, 1991). Spearman correlation evaluates the exact order over the entire set of modules. It is not appropriate for our application, because we do not care whether the rankings match exactly. Within the group that is enhanced, the order they are enhanced does not matter. They get equal treatment. Similarly, for those modules not enhanced, order does not matter. What does matter is where a module falls in relationship to a cutoff percentile that marks the end of the sequence of enhanced modules. However, at the time of modeling, we are uncertain what the cutoff will be. We want the reliability enhancement processes to find as many faults as possible. A module-order model enables management to enhance modules in the predicted order, confident that the total number of faults found will be close to expectations, even though the order of enhancement may not be perfect. Based on these considerations, the following is our evaluation procedure for a module-order model. Given a model, and a validation data set indexed by i:

1. Management will choose to enhance modules in priority order, beginning with the most fault-prone. However, the rank of the last module enhanced is uncertain at the time of modeling. Determine a range of percentiles that covers management's options for the last module, based on the schedule and resources allocated to reliability enhancement and associated uncertainties. Choose a set of representative cutoff percentiles, c, from that range.

2. For each cutoff percentile value of interest, c, define the number of faults accounted for by modules above the percentile c:

$$G(c) = \sum_{i:\ R_i \geq c} F_i \tag{1}$$

$$\widehat{G}(c) = \sum_{i:\ \widehat{R}(\mathbf{x}_i) \geq c} F_i \tag{2}$$

   where higher c corresponds to the more fault-prone modules.

3. Let G_{tot} be the total number of actual faults in the validation data set's software modules. Calculate the percentage of faults accounted for by each ranking, namely, $G(c)/G_{tot}$ and $\widehat{G}(c)/G_{tot}$, and depict the accuracy of the model with an Alberg diagram (Ohlsson and Alberg, 1996).

4. Calculate a function measuring model performance, $\phi(c)$, which indicates how closely the faults accounted for by the model ranking match those of the perfect ranking.

$$\phi(c) = \frac{\widehat{G}(c)}{G(c)} \tag{3}$$

   Plot performance as a function of c.

We want to identify only the worst modules, because resources are usually limited for reliability enhancement. In this paper, we chose 50 through 95 percentiles, in 5 percent increments as a hypothetical preferred set of cutoff percentiles. If resources for reliability enhancement are indeed limited, it is unlikely that more than 50% of the modules will be enhanced. Another project might choose different percentiles, but this set illustrates our methodology. Cutoff percentiles can be calculated easily, if management's objective is in terms of the number of modules enhanced.

Ohlsson and Alberg introduce a variation of Pareto diagrams which they call "Alberg diagrams" (Ohlsson and Alberg, 1996). Curves are plotted for an ordering based on the actual number of faults, and an ordering based on the number of faults predicted by a model. We employ Alberg diagrams (Ohlsson and Alberg, 1996) in this paper as an informal depiction of model accuracy.

In our context, an Alberg diagram consists of two curves, $G(c)/G_{tot}$ and $\widehat{G}(c)/G_{tot}$ plotted as functions of the percentage of modules selected for reliability enhancement, i.e., $1 - c$. Modules are selected for reliability enhancement in descending order, beginning with the most fault-prone. The percentages of all faults, $G(c)/G_{tot}$ and $\widehat{G}(c)/G_{tot}$, give us insight into the importance of each cutoff percentile. If the two curves are close together, then the model is considered accurate.

When model predictions are intended only for ordinal purposes, Ohlsson, *et al.* (Ohlsson and Alberg, 1996; Ohlsson et al., 1996), consider the model with the smallest area between curves to be the most useful. In case studies, they also compared models by the distance between the curves at selected percentages of modules.

The percentage of faults in a perfect ranking, $\phi(c)$, indicates the accuracy of the model at a given c. The variation in $\phi(c)$ over a range of c indicates the robustness of the model; small variation implies a robust model. A robust model is important in the face of uncertain resources for reliability enhancement. Because the number of cutoff percentiles is small, a statistical measure of robustness should be a simple measure of variation, such as the range of $\phi(c)$. We prefer a graphical presentation.

3. Classification

Our case studies compare module-order models to typical software quality classification models (Khoshgoftaar et al., 1996b). A classification model defines *fault-prone* by a threshold on the number of faults or some other software quality factor. Based on this definition, the classification model recommends a certain set of modules for reliability enhancement.

A module-order model can be used for classification: those modules ranked above the cutoff percentile, c, are predicted to be *fault-prone*, and those below are considered *not fault-prone*. The fraction recommended by a conventional classification model corresponds to a specific cutoff value, c, in a module-order model.

Classification accuracy can be evaluated when a threshold defining *fault-prone* is available. Such a threshold is usually based on project-specific factors, independent of any model. After fault data is known, such a threshold answers the question, "Is a module actually *fault-prone* in the eyes of the project?" Accuracy is evaluated in terms of misclassification rates. The same methods apply to statistical classification and module-order model classification. A Type I misclassification is when a classification model identifies a module as *fault-prone* which is actually *not fault-prone*. A Type II misclassification is when the model identifies a module as *not fault-prone* which is actually *fault-prone*. We have observed that it is often difficult to apply Type I and Type II misclassification rates to management issues. Therefore, we translate these rates into measures of the *effectiveness* and *efficiency* of a classification model, which are more closely related to project management concerns (Khoshgoftaar and Allen, 1997). Details are shown in Table I.

We want to identify all *fault-prone* modules. We define *effectiveness* as the proportion of *fault-prone* modules correctly identified. When we apply a reliability enhancement process to a *not fault-prone* module, it will be a waste of time, because the reliability is already satisfactory. We define *efficiency* as the proportion of reliability enhancement effort that is not wasted. We can maximize *effectiveness* by minimizing Type II misclassifications, and we can maximize *efficiency* by minimizing Type I misclassifications. There is a tradeoff between the Type I misclassification rate and the Type II misclassification rate. As one goes down, the other goes up. There is a similar tradeoff between *effectiveness* and *efficiency*.

4. Case Study Methodology

We take a case-study approach to illustrate the usefulness of a software quality modeling technique in a real-world setting, rather than controlled experiments (Basili et al., 1986;

Table I. Effectiveness and efficiency.

G_1	*Not fault-prone* group (class)
G_2	*Fault-prone* group (class)
$Class_i$	Actual class of module i
$Class(\mathbf{x}_i)$	Predicted class of module i based on vector of independent variables, $\mathbf{x}_i$
π_1	Expected proportion of *not fault-prone* modules
π_2	Expected proportion of *fault-prone* modules
$\Pr\{1\|1\}$	Rate of correct classifications of *not fault-prone* modules. $$\Pr\{1\|1\} = \Pr\{Class(\mathbf{x}_i) = G_1 \| Class_i = G_1\} \quad (4)$$
$\Pr\{2\|2\}$	Rate of correct classifications of *fault-prone* modules
$\Pr\{2\|1\}$	Type I misclassification rate
$\Pr\{1\|2\}$	Type II misclassification rate
effectiveness	Proportion of *fault-prone* modules that received reliability enhancement treatment out of all the *fault-prone* modules $$effectiveness = \Pr\{2\|2\} = 1 - \Pr\{1\|2\} \quad (5)$$
efficiency	Proportion of *fault-prone* modules that received reliability enhancement treatment out of all modules that received it. $$efficiency = \frac{\Pr\{2\|2\}\pi_2}{\Pr\{2\|1\}\pi_1 + \Pr\{2\|2\}\pi_2} \quad (6)$$

Pfleeger, 1995). A case study is based on data from one or more past development projects. To be credible, the software engineering community demands that the subject of an empirical study be a system with the following characteristics (Votta and Porter, 1995): (1) developed by a group, rather than an individual programmer; (2) developed by professionals, rather than students; (3) developed in an industrial environment, rather than an artificial setting; and (4) large enough to be comparable to real industry projects. Our case studies fulfill these criteria.

A retrospective case study builds software quality models that could have been built at an early point in the life cycle and calculates predictions. The predictions are compared to the actual quality that occurred in a later phase of the life cycle. Thus, a case study simulates what could have been predicted during the development project. The case study results indicate the accuracy that one can expect from the model when applied to a similar current project.

The following is a summary of our methodology for building and validating an underlying quantitative model in the context of a case study (Khoshgoftaar et al., 1996d). Our case studies used multiple linear regression, as described in Appendix B. The same approach is applicable to a wide variety of quantitative modeling techniques.

1. Preprocess measurements, if necessary, to improve the model. The second case study used principal components analysis, which is described in Appendix A, to reduce the number of independent variables and to improve model stability.

2. Choose a model validation strategy, such as resubstitution, data splitting, cross-validation, or multiple projects (Dillon and Goldstein, 1984; Khoshgoftaar and Allen, 1998a). Model validation evaluates the accuracy of a model.

3. Prepare *fit* and *test* data sets according to the model validation strategy.

   A quality factor is the dependent variable and software product and/or process metrics are the independent variables.

4. Select significant independent variables from a set of candidates, based on the *fit* data set. The case studies used stepwise regression, which is described in Appendix B.

5. Estimate parameters of the model on the selected independent variables, based on the *fit* data set. The case studies used multiple linear regression, which is summarized in Appendix B.

6. Use the model to predict the quality factor of each module in the *test* data set, and compare predictions to actual values for the *test* data set. Appendix B explains conventional error measures used by our case studies.

The *fit* data set is used to estimate parameters of a model. The *test* data set is used to evaluate its accuracy (Schneidewind, 1992). 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. The first case study used the *data splitting* technique (Geisser, 1975), which derives *fit* and *test* data sets from a single data set by impartially partitioning available observations. The second case study used one release to build the model (*fit*), and a subsequent release to evaluate it (*test*). This simulated actual use more closely than data splitting.

After an underlying quantitative model has been built, one can evaluate the associated module-order model, as described in Sections 2 and 3.

The case studies used nonparametric discriminant analysis, which is described in Appendix C, as a representative classification technique. Other classification techniques could have been used (Basili et al., 1996; Briand et al., 1993; Ebert, 1996; Khoshgoftaar et al., 1994a; Munson and Khoshgoftaar, 1992; Schneidewind, 1995). Discriminant analysis used the same *fit* and *test* data sets as the regression modeling (see Steps 1 through 3 above) and then performed the following steps.

4. Select significant independent variables from the same set of candidates as used by regression modeling, based on the *fit* data set. The case studies used stepwise discriminant analysis, which is described in Appendix C.

5. Estimate parameters of the model with the selected independent variables, based on the *fit* data set. The case studies used nonparametric discriminant analysis which is summarized in Appendix C.

6. Use the model to predict the class of each module in the *test* data set, and compare predictions to actual values. Calculate misclassification rates, *effectiveness*, and *efficiency*.

Table II. Software product metrics for CCCS.

Symbol	Description
η_1	Number of unique operators (Halstead, 1977)
N_1	Total number of operators (Halstead, 1977)
η_2	Number of unique operands (Halstead, 1977)
N_2	Total number of operands (Halstead, 1977)
$V(G)$	McCabe's cyclomatic complexity (McCabe, 1976)
N_L	Number of logical operators
LOC	Lines of code
ELOC	Executable lines of code

5. A Military System

5.1. System Description

We studied a large military command, control, and communications system, written in Ada, which we call "CCCS". The system was developed in a large organization by professionals using the procedural programming paradigm. Generally, a module was an Ada package, consisting of one or more procedures. Faults were attributed to each module during the system integration and test phase and during the first year of deployment. The top 20% of the modules contained 82.2% of the faults, and the top 5% of the modules contained 40.2% of the faults. Consequently, identification of modules with the most faults before the testing phase could pay off handsomely. 52% of the modules had no faults, and over three quarters of the modules had two or fewer faults. The maximum number of faults in one module was 42. The developers collected software product metrics from the source code. The number and selection of metrics was determined by available data collection tools. Table II lists the software product metrics used in this study. Another project might collect a different set (Khoshgoftaar et al., 1998a).

We analyzed all 282 modules measured by the developers. Applying data splitting, we impartially partitioned this data into two subsets, two thirds of the modules (188) for fitting models, and the remaining third (94 modules) for evaluating their predictive accuracy. This yielded adequate sample sizes for statistical purposes. Other proportions might be appropriate in other studies. Table III shows descriptive statistics on variables in the *fit* data set. Table IV show correlations among independent variables in the *fit* data set.

5.2. Module-Order Model

We applied multiple linear regression to CCCS data. The predicted dependent variable, $\widehat{F}(\mathbf{x}_i)$, was the number of faults. Candidate independent variables were the eight product metrics listed in Table II. Based on the *fit* data set, stepwise regression selected η_2, $V(G)$, N_L, and η_1 at the 5% significance level. The intercept was not significantly different from zero. We identified two outliers at $\alpha = 0.01$ (see Appendix B for details). However, we

Table III. Descriptive statistics of CCCS data.

Metric	Median	Mean	Std Dev	Min	Max
η_1	21	22.8	15.6	4	85
η_2	61	110.9	152.8	2	1124
N_1	187	589.3	1069.5	6	8606
N_2	164	499.4	941.9	2	7736
$V(G)$	8	29.7	68.4	1	614
N_L	0	3.2	8.0	0	58
LOC	389.5	808.9	1142.4	19	9163
ELOC	42.5	108.4	182.9	3	1412

188 modules in *fit* data set

Table IV. Correlations of CCCS data.

	η_1	η_2	N_1	N_2	$V(G)$	N_L	*LOC*	*ELOC*
η_1	1.00	0.81	0.75	0.74	0.61	0.66	0.73	0.76
η_2		1.00	0.93	0.92	0.83	0.64	0.88	0.93
N_1			1.00	0.99	0.79	0.71	0.96	0.98
N_2				1.00	0.77	0.70	0.96	0.97
$V(G)$					1.00	0.47	0.76	0.87
N_L						1.00	0.73	0.67
LOC							1.00	0.96
ELOC								1.00
Faults	0.65	0.76	0.73	0.74	0.47	0.73	0.74	0.69

188 modules in *fit* data set

did not have detailed information on these two modules to justify removing them from the *fit* data set. The following model was estimated using the least-squares technique.

$$\widehat{F} = 0.0305\,\eta_2 - 0.0318\,V(G) + 0.2300\,N_L - 0.0401\,\eta_1 \qquad (7)$$

Each variable was significant at $\alpha < 0.04$. Table V lists details about the model. The quality of fit was indicated by an $R^2 = 0.738$. Application of the model to the *test* data set yielded average absolute error of $AAE = 2.2$ faults. (The average relative error of $ARE = 0.70$ is not an appropriate measure of model accuracy because over 75% of the modules had two or fewer faults.) As a conventional quantitative model, the quality of fit was satisfactory and the model was accurate enough to be useful to a project. A more sophisticated model

Table V. Details of CCCS model.

Variable	Parameter	Std Dev
η_2	0.0305	0.0029
$V(G)$	−0.0318	0.0047
N_L	0.2300	0.0289
η_1	−0.0401	0.0130

Table VI. Results of CCCS model.

c	$G(c)/G_{tot}$	$\widehat{G}(c)/G_{tot}$	$\phi(c)$	
0.950	0.419	0.361	0.861	
0.900	0.631	0.544	0.862	
0.850	0.751	0.697	0.928	
0.800	0.822	0.772	0.939	
0.750	0.884	0.830	0.939	
0.700	0.925	0.846	0.915	
0.650	0.942	0.871	0.925	
0.600	0.963	0.884	0.918	
0.550	0.983	0.909	0.924	
0.500	1.000	0.913	0.913	
c	Type I	Type II	*effectiveness*	*efficiency*
0.950	0.000	0.789	0.211	1.000
0.900	0.013	0.579	0.421	0.889
0.850	0.027	0.368	0.632	0.857
0.800	0.040	0.211	0.789	0.833
0.750	0.067	0.053	0.947	0.783
0.700	0.133	0.053	0.947	0.643
0.650	0.173	0.000	1.000	0.594
0.600	0.240	0.000	1.000	0.514
0.550	0.307	0.000	1.000	0.452
0.500	0.373	0.000	1.000	0.404
		fault-prone: $F_i \geq 4$		

might use principal components analysis to deal with multicollinearity. For this paper, we preferred to employ a simple, well-known, quantitative-modeling technique.

Table VI lists results for the module-order model of CCCS. Figure 1 is an Alberg diagram (Ohlsson and Alberg, 1996) of $G(c)/G_{tot}$ and $\widehat{G}(c)/G_{tot}$ as listed in Table VI. Figure 2 shows the ratio of the two lines in Figure 1, which is our measure of model performance, $\phi(c)$. It shows how close the model comes to a perfect ordering of the modules. The ordering specified by the model accounts for nearly the same number of faults as a perfect ordering.

To evaluate a module-order model as a classification model, we must have an *a priori* definition for the *fault-prone* class. In this case study, *fault-prone* modules had four or more faults, corresponding to the 81st percentile of modules. This threshold illustrates project-specific criteria, such as practical constraints on the amount of reliability enhancement effort. Other thresholds might be appropriate in other circumstances.

Those modules with $\widehat{R}(\mathbf{x}_i) \geq c$ were predicted to be *fault-prone*. Table VI also lists misclassification rates, *effectiveness*, and *efficiency*, based on this threshold. Figure 3 depicts the tradeoff between Type I and Type II misclassification rates for various values of the cutoff parameter, c. Figure 4 depicts the same tradeoff in terms of *effectiveness* and *efficiency* as a function of c.

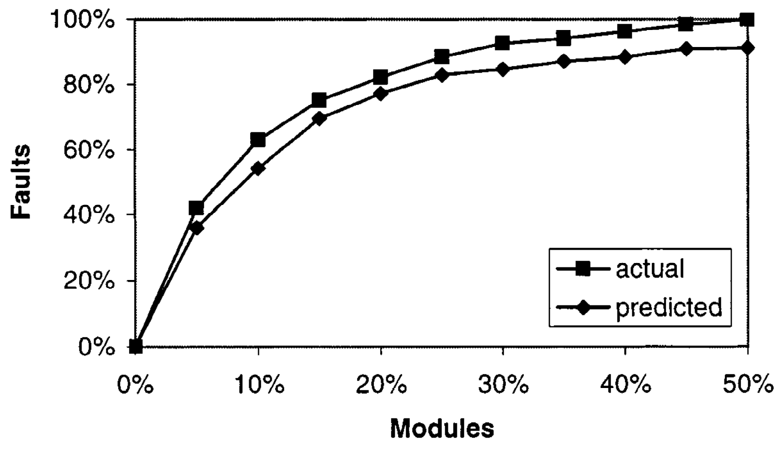

Figure 1. Alberg diagram for CCCS model.

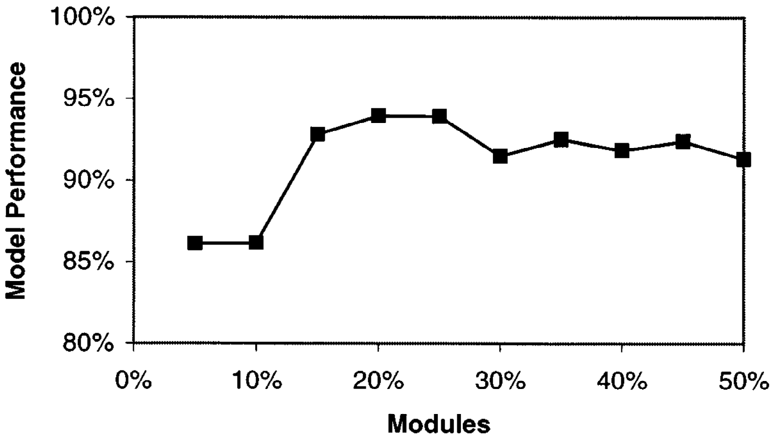

Figure 2. Performance of CCCS model, $\phi(c)$.

5.3. Nonparametric Discriminant Analysis

We applied nonparametric discriminant analysis to CCCS data (Seber, 1984). The dependent variable was whether a module was *fault-prone* or not. We used the same threshold as above, i.e., *fault-prone* modules had four or more faults. The candidate independent variables were the same product metrics, as listed in Table II. Stepwise discriminant analysis selected η_2, $V(G)$, η_1, *ELOC*, and N_2 at the 15% significance level. The significance of each parameter was $\alpha \leq 0.042$. We empirically selected a smoothing parameter, $\lambda = 0.20$, as discussed

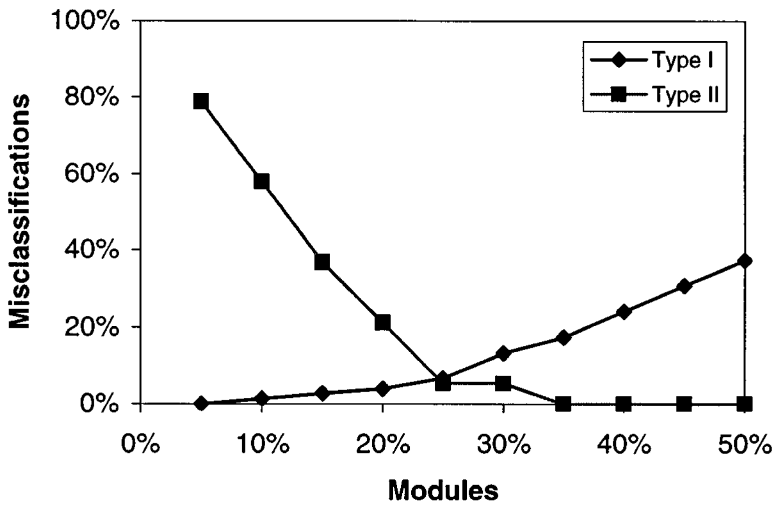

Figure 3. Misclassifications by CCCS model.

Table VII. CCCS discriminant analysis.

test data set
Number of Observations/Percent

	Model		
Class	*not f-p*	*f-p*	Total
Actual			
not f-p	60	15	75
	80.0%	20.0%	100.0%
f-p	3	16	19
	15.8%	84.2%	100.0%
Total	63	31	94
Percent	67.0%	33.0%	100.0%
Prior	80.8%	19.2%	

Overall misclassification: 19.2%

in Appendix C. Table VII shows the results when the model was applied to the *test* data set. The proportion of modules recommended for reliability enhancement was 33.0%. This model had useful accuracy: the Type I misclassification rate was 20.0%, the Type II rate was 15.8%, and the overall rate was 19.2%. The misclassification rates translated into an *effectiveness* of 84.2% and *efficiency* of 50.0%. In other words, 84.2% of the *fault-prone* modules were recommended for reliability enhancement, and half of the modules given reliability enhancement would, in fact, be *fault-prone*.

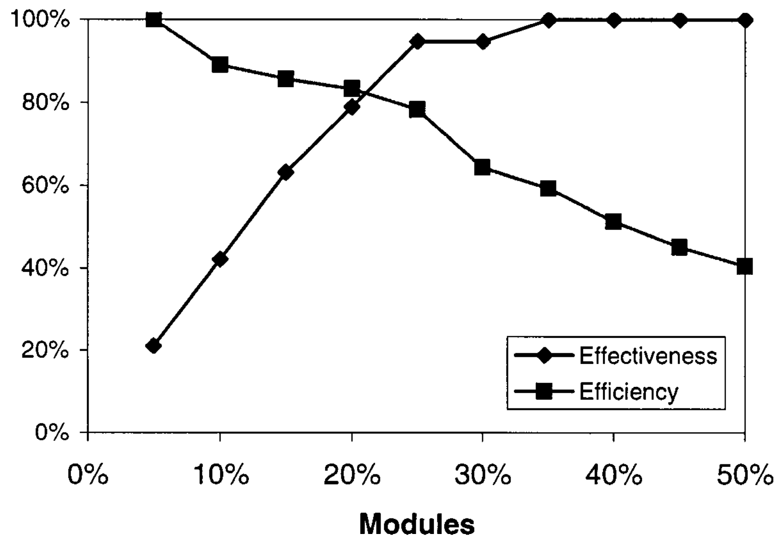

Figure 4. Effectiveness and efficiency of CCCS model.

5.4. Discussion

If project management selects a percentage of modules for reliability enhancement, i.e., $1 - c$, and a threshold to define *fault-prone,* we can use the module-order model as a classification model, and directly compare it to the nonparametric discriminant model. In this case study, the module-order model had the following advantages compared to the discriminant model.

The module-order model did not require a threshold to define *fault-prone* at the time the model was built. This means reliability enhancement can be given to as many modules as resource constraints allow. The small variation in $\phi(c)$ in Figure 2 indicates that the module-order model was robust for different cutoff percentiles, c, with good accuracy.

The threshold defining *fault-prone* was relevant only for evaluation of module-order models as a classification technique and for comparison to nonparametric discriminant analysis. The module-order model was more accurate than the discriminant model. Recall that the discriminant model recommended 33.0% for reliability enhancement, i.e., predicted to be *fault-prone*. Therefore, we compare it to $1 - c = 0.30$ and $1 - c = 0.35$, which were the neighboring data points in Table VI. Table VIII compares the accuracy of the models; both $1 - c = 0.30$ and $1 - c = 0.35$ were more accurate than nonparametric discriminant analysis in this case study. For $1 - c = 0.35$, the module-order model correctly classified all *fault-prone* modules and classified *not fault-prone* modules more accurately than the discriminant model. This translated to perfect *effectiveness* and better *efficiency*.

Table VIII. Classification comparison for CCCS.

	Model		
	Discrim	Module-Order	
% Recommended $(1 - c)$	33.0%	30.0%	35.0%
Type I errors	20.0%	13.3%	17.3%
Type II errors	15.8%	5.3%	0.0%
Effectiveness	84.2%	94.7%	100.0%
Efficiency	50.0%	64.3%	59.4%

Table IX. System profile of LTS.

	Release	
	0	1
Procedures	22004	38194
Modules	97	171
Code Churn (Lines)		
Development	18308	136792
Debug	15009	56711

New or changed modules
Code Churn = lines added or changed
Deleted lines were not measured.

6. A Legacy Telecommunication System

6.1. System Description

We studied a large legacy telecommunications system (LTS) written by professional programmers in a large organization (Khoshgoftaar et al., 1996a). This embedded computer application included numerous finite state machines and interfaces to other kinds of equipment. It was written in a proprietary, procedural, high-level language similar to Pascal. Table IX summarizes the size of the system. The entire system (Release 1) had over 50 thousand procedures; the portion we studied had over 38 thousand procedures in the 171 modules that were new or changed since the prior release.

Prior research has shown that the reuse history of a module can be a significant input to software quality models (Khoshgoftaar et al., 1996b; Khoshgoftaar et al., 1996c). The unchanged modules were extremely reliable, i.e., they had no changes due to debugging. The amount of code that was new or changed due to development, namely, *development code churn*, *DEV*, quantified the amount of code that was not reused from the previous release. *DEV* can be measured at the same point in the life cycle as software metrics.

In this study, we focused on the number of lines new or changed due to bug fixes, namely, *debug code churn*, *FIX*. Bug fixes generally occurred after development coding was complete. Table X gives summary statistics on debug code churn for both releases. This data was available at the module level.

Table X. Debug code churn statistics for LTS.

	Release	
Statistic	0	1
Obs	97	171
Mean	154.7	331.6
Std Dev	300.0	734.9
Min	0	0
Q1	11	12
Median	77	68
Q3	199	294
Max	2439	5066

Lines new/changed due to bug fixes, *FIX*

Software product metrics were collected by a Logiscope metric analyzer at the procedure level, and were aggregated to the module level. Many of the metrics are defined on a control flow graph of the module, consisting of nodes and edges which depict the algorithmic flow of control. Table XI lists the product metrics used in this study. Other metrics might be used in another project.

Data for each module consisted of product metrics, development churn (*DEV*), and debug churn (*FIX*). New or changed modules from Release 0 were used as a *fit* data set (97 observations), and new or changed modules from Release 1 were used as a *test* data set (171 observations). This simulated how a model can be used in practice.

6.2. Preprocess Data

In this case study, we preprocessed the data to reduce the large number of product metrics to a few principal component variables, which we call "domain metrics". Appendix A describes principal components analysis.

Principal components analysis of the standardized software product metrics from the combined *fit* and *test* data sets retained five components under the stopping rule that we retain components accounting for at least 90% of the overall variance. We combined the data sets for this step to improve the sample size. The resulting domain metrics were not correlated with each other, had a mean of zero, and a standard deviation of one. Table XII shows the correlation between each original metric and each domain metric. The largest in each row is shown bold, indicating the product metrics most closely related to each domain metric. The variance of each domain metric is also shown. In general, D_1 was strongly correlated to various measures of module size; D_2 was strongly correlated to a other measures of size, such as the number of procedures, *N_IN*; D_3 was correlated to the size of the largest control structure; D_4 was correlated to the maximum nesting level; and D_5 was correlated to the number of independent paths.

The principal components analysis resulted in a standardized transformation matrix which was separately applied to the *fit* and *test* standardized product metrics to calculate domain metrics, $D_1, \ldots, D_5$. Table XIII lists descriptive statistics for the *fit* data set of the domain

Table XI. Software product metrics for LTS.

N_STMTS	Number of statements.
N_COM	Number of comments.
TOT_OPTR	Total operators (Halstead, 1977).
N_EDGES	Number of edges in control flow graph.
N_NODES	Number of nodes in control flow graph.
P_NODES	Number of pending nodes. The number of nodes that begin a sequence of unreachable code (dead code).
MAX_DEG	Maximum degree of node. The maximum number of edges going in or out of a node.
N_PATHS	Number of independent paths (Nejmeh, 1988).
N_IN	Number of entries. Since each procedure has only one entry point, this is equivalent to the number of procedures in the module.
N_STRUC	Number of control structures. (e.g., `IF`, `WHILE`, or `DO`)
MAX_LVLS	Maximum nesting levels in a procedure.
N_SEQ	Number of sequential nodes.
MAX_NODES	Maximum nodes in a control structure.
MAX_STMTS	Maximum statements in control structure.
DRCT_CALLS	Direct calls. This is the total number of procedure or function calls.
ESS_CPX	McCabe essential complexity. This measures the degree a procedure has constructs due to branching in/out of control structures. A well structured procedure has $ESS\_CPX = 1$.
DES_CPX	McCabe design complexity (McCabe and Butler, 1989). This measures the cyclomatic complexity (McCabe, 1976) related to procedure calls.

metrics and development code churn, *DEV*. The mean of each domain metric is about zero and its standard deviation is about one.

6.3. Module-Order Model

We applied multiple linear regression to LTS data. The predicted dependent variable, $\widehat{F}(\mathbf{x}_i)$, was debug code churn, *FIX*. Candidate independent variables were *DEV* and domain metrics D_1 through D_5. Based on the *fit* data set, stepwise regression selected *DEV* and D_1 at the 5% significance level. We identified one outlier at $\alpha = 0.01$ (see Appendix B for details). However, we did not have detailed information on this module to justify removing it from the *fit* data set. The following model was estimated using the least-squares technique.

$$\widehat{F} = 95.654 + 0.289\,DEV + 63.274\,D_1 \tag{8}$$

Each variable was significant at $\alpha < 0.033$. Table XIV gives details about the model. The quality of fit was indicated by an $R^2 = 0.138$. Application of the model to the *test* data

Table XII. Domain pattern for LTS product metrics.

	Domain Metric				
	D_1	D_2	D_3	D_4	D_5
P_NODES	**0.933**	0.099	0.203	0.054	0.124
ESS_CPX	**0.881**	0.356	0.236	0.119	0.106
DES_CPX	**0.880**	0.353	0.238	0.112	0.113
N_STRUC	**0.813**	0.306	0.255	0.244	0.306
N_NODES	**0.752**	0.512	0.246	0.212	0.230
N_EDGES	**0.733**	0.508	0.261	0.226	0.250
N_STMTS	**0.699**	0.590	0.200	0.193	0.222
N_COM	**0.686**	0.533	0.188	0.169	0.201
TOT_OPTR	0.213	**0.921**	0.146	0.040	0.003
DRCT_CALLS	0.266	**0.918**	0.104	0.047	0.010
N_IN	0.460	**0.831**	0.136	0.102	−0.005
N_SEQ	0.566	**0.724**	0.231	0.189	0.168
MAX_DEG	0.118	**0.586**	0.569	−0.229	−0.184
MAX_NODES	0.298	0.154	**0.908**	0.125	0.126
MAX_STMTS	0.286	0.163	**0.887**	0.150	0.142
MAX_LVLS	0.272	0.091	0.130	**0.922**	0.057
N_PATHS	0.408	−0.017	0.145	0.060	**0.876**
Variance	6.224	4.821	2.506	1.248	1.210
% Variance	36.6%	28.4%	14.7%	7.3%	7.1%
Cumulative	36.6%	65.0%	79.7%	87.0%	94.1%

Table XIII. Descriptive statistics of LTS data.

Metric	Median	Mean	Std Dev	Min	Max
DEV	65.00	188.74	290.70	10.00	1470.00
D_1	−0.20	0.07	0.99	−1.48	3.68
D_2	−0.14	−0.03	1.02	−1.76	2.64
D_3	−0.22	−0.01	0.94	−1.62	3.23
D_4	−0.13	0.07	0.94	−1.96	2.51
D_5	−0.24	0.04	1.08	−1.32	6.44

97 modules in *fit* data set

set yielded average absolute error $AAE = 188.5$ lines of code and average relative error $ARE = 35.3$. The regression model was statistically significant but the quality of fit was low. From the *AAE* and *ARE*, we concluded that the model was not very accurate.

Table XV lists results for the module-order model of LTS. Figure 5 is an Alberg diagram (Ohlsson and Alberg, 1996) of $G(c)/G_{tot}$ and $\widehat{G}(c)/G_{tot}$ as listed in Table XV. Figure 6 graphs our model-performance measure, $\phi(c)$. Like the CCCS case study, the model accounts for nearly the same amount of debug code churn as a perfect ordering. This was surprising in view of its poor accuracy as a conventional quantitative model.

The definition of *fault-prone* should be based on historical statistics and the opinions of project experts for each release. In some projects, the appropriate threshold is obvious. For example, another telecommunications case study based the threshold on customer

Table XIV. Details of LTS model.

Variable	Parameter	Std Dev
Intercept	95.654	34.141
DEV	0.289	0.100
D_1	63.274	29.202

Table XV. Results of LTS model.

c	$G(c)/G_{tot}$	$\widehat{G}(c)/G_{tot}$	$\phi(c)$	
0.950	0.434	0.374	0.861	
0.900	0.639	0.568	0.889	
0.850	0.738	0.684	0.928	
0.800	0.817	0.757	0.927	
0.750	0.864	0.791	0.915	
0.700	0.905	0.834	0.922	
0.650	0.929	0.858	0.924	
0.600	0.948	0.910	0.960	
0.550	0.962	0.922	0.959	
0.500	0.974	0.940	0.965	
c	Type I	Type II	*effectiveness*	*efficiency*
0.950	0.000	0.810	0.190	1.000
0.900	0.008	0.619	0.381	0.941
0.850	0.023	0.476	0.524	0.880
0.800	0.054	0.357	0.643	0.794
0.750	0.101	0.310	0.690	0.690
0.700	0.140	0.214	0.786	0.647
0.650	0.186	0.167	0.833	0.593
0.600	0.217	0.048	0.952	0.588
0.550	0.271	0.024	0.976	0.539
0.500	0.333	0.000	1.000	0.494
fault-prone: $R_i \geq 75^{th}$ percentile				

satisfaction (Khoshgoftaar et al., 1998b). A module with any customer-discovered faults was considered *fault-prone*: $F_i > 0$. This project, in contrast, considered practical limits on reliability enhancement in terms of proportions. *Fault-prone* modules were those in the top quartile of modules with respect to debug code churn. As we saw in Table X, the distribution of debug code churn was different in each release. Therefore, the top quartile represented different amounts of code churn. For Release 0, modules were *fault-prone* when $FIX \geq 199$. For Release 1, modules with $FIX \geq 300$ were considered *fault-prone*. Both thresholds correspond to approximately the same proportion of modules. Another criterion for *fault-prone* group membership might be appropriate in other situations.

Those modules with $\widehat{R}(\mathbf{x}_i) \geq c$ were predicted to be *fault-prone*. Table XV also lists classification results for various values of c. Figure 7 depicts the tradeoff between Type I and Type II misclassification rates as a function of the cutoff parameter, c. Figure 8 depicts the same tradeoff in terms of *effectiveness* and *efficiency* as a function of c.

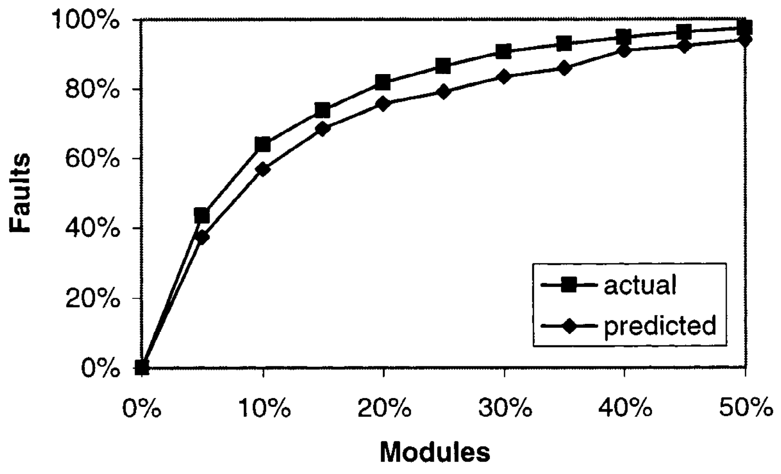

Figure 5. Alberg diagram for LTS model.

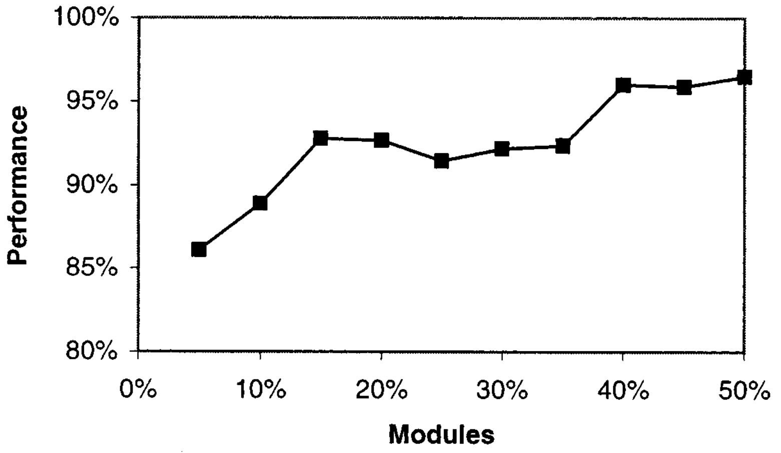

Figure 6. Performance of LTS model, $\phi(c)$.

6.4. Nonparametric Discriminant Analysis

We applied nonparametric discriminant analysis to LTS data. The dependent variable was whether the module was *fault-prone* or not. We used the same thresholds on debug code churn as above to define *fault-prone*. Candidate independent variables were the same as above, i.e., *DEV*, and domain metrics D_1 through D_5. Stepwise discriminant analysis selected *DEV*, D_1, and D_4 at the 15% significance level. Each parameter's significance

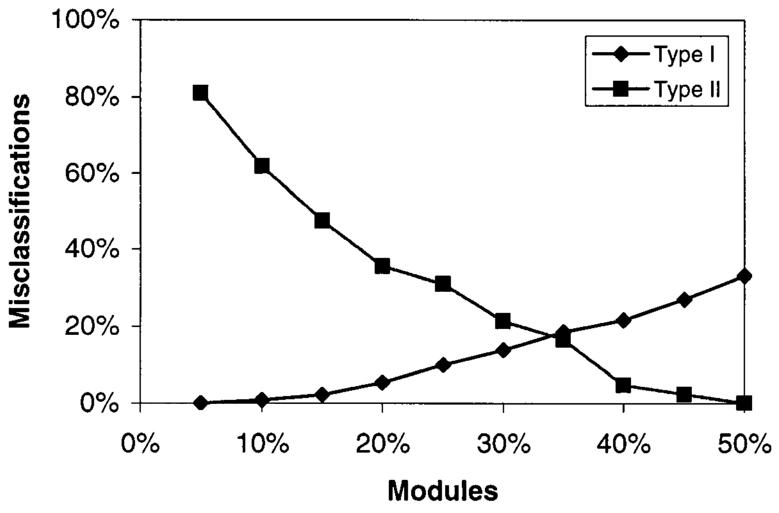

Figure 7. Misclassifications by LTS model.

Table XVI. LTS discriminant analysis.

test data set
Number of Observations/Percent

	Model		
Class	*not f-p*	*f-p*	Total
Actual			
not f-p	99	30	129
	76.7%	23.3%	100.0%
f-p	8	34	42
	19.1%	80.9%	100.0%
Total	107	64	171
Percent	62.6%	37.4%	100.0%
Prior	74.2%	25.8%	

Overall misclassification: 22.2%

was $\alpha < 0.070$. We empirically chose a smoothing parameter, $\lambda = 0.25$, as discussed in Appendix C. Table XVI shows the results when the model was applied to the *test* data set. The model identified 37.4% of the modules as *fault-prone*, and thus, they were recommended for reliability enhancement. The model had useful accuracy: the Type I misclassification rate was 23.3%, the Type II misclassification rate was 19.1%, and the overall misclassification rate of 22.2%. This corresponded to an *effectiveness* of 80.9% and *efficiency* of 54.5%. In other words, 80.9% of the *fault-prone* modules were recommended for reliability enhancement, and of those enhanced, 54.5% would, in fact, be *fault-prone*.

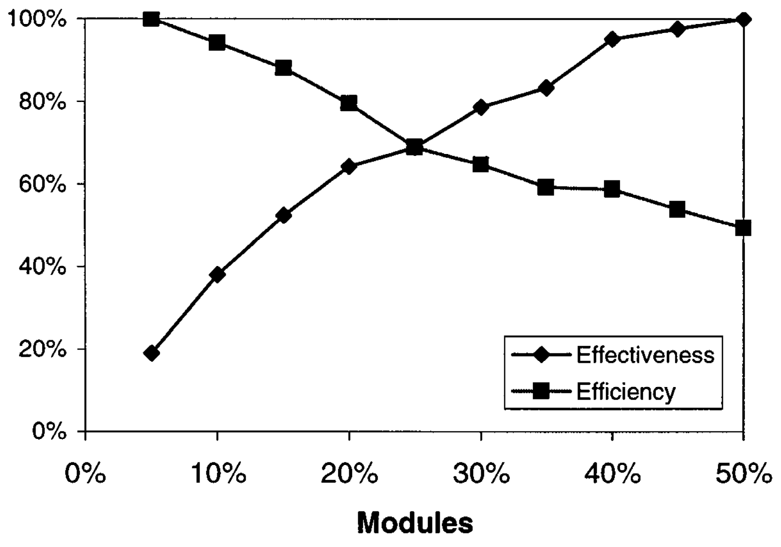

Figure 8. Effectiveness and efficiency of LTS model.

Table XVII. Classification comparison for LTS.

	Model		
	Discrim	Module-Order	
% Recommended (1 − *c*)	37.4%	35.0%	40.0%
Type I errors	23.3%	18.6%	21.7%
Type II errors	19.1%	16.7%	4.8%
Effectiveness	80.9%	83.3%	95.2%
Efficiency	54.5%	59.3%	58.8%

6.5. Discussion

A module-order model can be used as a classification model, and thus, can be directly compared to a nonparametric discriminant model. The module-order model had similar advantages as in the CCCS case study compared to the nonparametric discriminant model. Figure 6 shows that $\phi(c)$ had small variation, indicating that the module-order model was similarly robust for different cutoff percentiles, c, with good accuracy. The module-order model was also more accurate than the discriminant model. Recall that the discriminant model recommended 37.4% for reliability enhancement. Therefore, we compare $1 - c = 0.35$ and $1 - c = 0.40$, which were the neighboring data points in Table XV. Table XVII compares the accuracy of the models; both values of $1 - c$ were more accurate than nonparametric discriminant analysis in this case study. For $1 - c = 0.40$, the module-order model misclassified only 4.8% of the *fault-prone* modules, rather than the 19.1% that the discriminant model misclassified. The module-order model also classified *not fault-prone* modules a little more accurately than the discriminant model. This resulted in better *effectiveness* and *efficiency*.

7. Conclusions

Software product and process metrics can be the basis for predicting reliability. Predicting the exact number of faults is often not necessary; classification models can identify fault-prone modules. However, such models require that *fault-prone* be defined before modeling, usually via a threshold. This is not practical when one is uncertain of resource constraints that limit the amount of reliability-improvement effort. For example, if reliability enhancement consists of extra design reviews and correcting mistakes, it is difficult to know ahead of time how much effort will be needed to fix the mistakes. In such cases, predicting the rank-order of modules is more useful than just classification.

A module-order model predicts the rank-order of modules according to a quantitative quality factor, such as the number of faults. This paper demonstrates how module-order models can be used for classification, and compares them with software quality classification models, in particular, nonparametric discriminant analysis.

Two case studies of full-scale industrial software systems compared nonparametric discriminant analysis with module-order models. The subject of one case study was a military command, control, and communications system. The subject of the other was a large legacy telecommunications system. Lessons learned are the following.

– Module-order models are most appropriate when a threshold to define *fault-prone* modules is not practical at the time of modeling. They facilitate enhancing the reliability of modules in priority order, most fault-prone first. All modules above a cutoff percentile are enhanced.

– User acceptance is facilitated, because users can easily be shown the rationale behind module-order model results, especially when the underlying quantitative modeling technique is familiar. Such models are not “black boxes”.

– In the case studies, the module-order model was robust for different cutoff percentiles with good accuracy, even though the accuracy of one study’s underlying quantitative model was poor.

– The module-order model was more accurate than the nonparametric discriminant analysis in the case studies. This translated into better *effectiveness* and better *efficiency* as well.

Overall, we found that module-order models give management more flexible reliability enhancement strategies than classification models.

Future work will investigate additional modeling techniques underlying the module-order model concept, such as evolutionary computation (Evett et al., 1998).

Acknowledgments

We thank Nishith Goel, Amit Nandi, and John McMullen for providing the data on the legacy telecommuniations system. We thank Jason Busboom for data analysis calculations. We thank Lionel Briand, editor, and anonymous reviewers for their helpful comments. The findings and opinions in this paper belong solely to the authors, and are not necessarily those of collaborators. Moreover, our results do not in any way reflect the quality of our collaborators' software products. Logiscope is a trademark of Verilog, S. A.

Appendix

A. Principal Components Analysis

Software product 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 common unit of measure becomes one standard deviation.

Principal components analysis is a statistical technique for transforming multivariate data into orthogonal variables, and for reducing the number of variables without losing significant variation. 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 the 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, $\mathbf{e}_j$, $j = 1, \ldots, m$. Each eigenvalue is the variance of the corresponding principal component. 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

$$\mathbf{t}_j = \frac{\mathbf{e}_j}{\sqrt{\lambda_j}} \text{ for } j = 1, \ldots, p \tag{9}$$

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, $j = 1, \ldots, p$.

$$D_j = \mathbf{Z}\mathbf{t}_j \tag{10}$$

$$\mathbf{D} = \mathbf{Z}\mathbf{T} \tag{11}$$

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

B. Multiple Linear Regression

Even though we may have a long list of independent variables, it is possible that some do not significantly influence the dependent variable. If an insignificant variable is included in the model, it may add noise to the results and may cloud interpretation of the model. In particular, if a coefficient, a_j for the j^{th} variable in a linear model is not significantly different from zero, then it is best to omit that term from the model. The process of determining which variables are significant is called *model selection*. Of several model selection techniques available for multiple linear regression, we use the *stepwise regression* method (Myers, 1990). Stepwise model selection is an iterative procedure. Given a list of candidate independent variables, each iteration may add a significant variable to the model and may remove an insignificant variables from the model, based on an F test. The significance test is recomputed on each iteration, until no variable can be added to or removed from the model.

We use the R-student statistic for each observation to test the hypothesis that the observation is an outlier at a significance level, α (Myers, 1990). Many models have a general mathematical form with parameters that must be chosen so that the *fit* data set matches the model as closely as possible. This step consists of estimating the values of such parameters. Suppose there are n observations in the *fit* data set, and the subscript i indicates data for the i^{th} observation. In general, a multivariate linear model has the following form.

$$\hat{y}_i = a_0 + a_1 x_{i1} + \cdots + a_p x_{ip} \tag{12}$$

$$y_i = a_0 + a_1 x_{i1} + \cdots + a_p x_{ip} + e_i \tag{13}$$

where $x_{i1}, \ldots, x_{ip}$ are the independent variables' values, $a_0, \ldots, a_p$ are parameters to be estimated, $\hat{y}_i$ is the predicted value of the dependent variable, y_i is the dependent variable's actual value, and $e_i = y_i - \hat{y}_i$ is the error for the i^{th} observation. We estimate the parameters, $a_0, \ldots, a_p$, using the *least squares* method. This method chooses a set of parameter values that minimizes $\sum_{i=1}^{n} e_i^2$ (Myers, 1990).

When the parameters have been estimated, and given a set of independent variable values, a model can calculate a value of the dependent variable. Since the independent variables are known earlier than the actual value of the dependent variable, the calculated value is a *prediction*.

When we know the actual dependent variable value, y_i, for the i^{th} module, we validate that the predictions, $\hat{y}_i$, are sufficiently accurate for the needs of the current project. Two common statistics for evaluating predictions are average absolute error, *AAE*, and average relative error, *ARE*.

$$AAE = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i| \tag{14}$$

$$ARE = \frac{1}{n} \sum_{i=1}^{n} \left| \frac{y_i - \hat{y}_i}{y_i + 1} \right| \tag{15}$$

where the denominator of *ARE* has one added to avoid dividing by zero (Khoshgoftaar et al., 1992b). A lower average error is better. Results based on the *fit* data set indicate quality of fit. Results based on the *test* data set indicate the accuracy of predictions one can expect for a similar project or a subsequent release. Once a model has been validated to make acceptable predictions for this development environment, the model is ready to apply to a current similar project.

C. Nonparametric Discriminant Analysis

Nonparametric discriminant analysis is a statistical technique for predicting the class (group), G_1 or G_2, of an observation, such as the i^{th} software module, represented by its vector of independent variables, $\mathbf{x}_i$.

We use *stepwise discriminant analysis* at a significance level, α, to choose the independent variables in the nonparametric discriminant model (Seber, 1984). An F test is used to test significance. Given a list of candidate variables, and 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.

We estimate a discriminant function based on the *fit* data set. Let $f_k(\mathbf{x}_i)$ be the multivariate probability density giving the likelihood that a module represented by $\mathbf{x}_i$ is in G_k. We predict class membership of a module by comparing density functions f_1 and f_2 at $\mathbf{x}_i$. A large likelihood means the module is probably in that group. Since the density functions, f_k, are often not normal distributions, we use "nonparametric" density estimation, and the multivariate kernel density estimation technique (Seber, 1984). Let $\hat{f}_k(\mathbf{x}_i|\lambda)$ be an approximation of $f_k(\mathbf{x}_i)$, where λ is a smoothing parameter. The estimated density function is given by

$$\hat{f}_k(\mathbf{x}_i|\lambda) = \frac{1}{n_k}\sum_{l=1}^{n_k} K_k(\mathbf{x}_i|\mathbf{x}_{kl}, \lambda) \tag{16}$$

where n_k is the number of observations in group G_k, $k = 1, 2$, and the vector $\mathbf{x}_{kl}$, $l = 1, \ldots, n_k$ represents a module in group G_k, and $K_k(\mathbf{u}|\mathbf{v}, \lambda)$ is a multivariate kernel function on vector $\mathbf{u}$ with modes at $\mathbf{v}$. We use the normal kernel.

$$K_k(\mathbf{u}|\mathbf{v}, \lambda) = (2\pi\lambda^2)^{-n_k/2}|\mathbf{S}_k|^{-1/2}\exp\left((-1/2\lambda^2)(\mathbf{u}-\mathbf{v})'\mathbf{S}_k^{-1}(\mathbf{u}-\mathbf{v})\right) \tag{17}$$

$\mathbf{S}_k$ is the covariance matrix for all observations in G_k, and $|\mathbf{S}_k|$ is its determinant.

We have observed that Type I and Type II misclassification rates vary in opposite directions, within limits, as λ varies. We empirically choose a preferred value for λ based on misclassification rates of cross-validation using the *fit* data set. We prefer approximately equal misclassification rates to the extent feasible.

Let π_k be the prior probability of membership in G_k, which we usually choose to be the proportion of *fit* observations in G_k. A classification rule that minimizes the expected number of misclassification is the following.

$$Class(\mathbf{x}_i) = \begin{cases} G_1 & \text{if } \frac{\hat{f}_1(\mathbf{x}_i|\lambda)}{\hat{f}_2(\mathbf{x}_i|\lambda)} \geq \frac{\pi_2}{\pi_1} \\ G_2 & \text{otherwise} \end{cases} \tag{18}$$

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.

Basili, V. R., Selby, R. W., and Hutchens, D. H. 1986. Experimentation in software engineering. *IEEE Transactions on Software Engineering* SE–12(7): 733–743.

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.

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

Briand, L. C., El Emam, K., and Morasca, S. 1996. On the application of measurement theory in software engineering. *Empirical Software Engineering: An International Journal* 1(1): 61–88.

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

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

Ebert, C. 1997. Experiences with criticality predictions in software development. In *Software Engineering—ESEC/FSE '97: Proceedings of the Sixth European Software Engineering Conference and the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1301 of *Lecture Notes in Computer Science*, pages 278–293. Zurich, Switzerland: Springer-Verlag. Also published as *ACM SIGSOFT Software Engineering Notes* 22(6), November 1997.

Evett, M. P., Khoshgoftar, T. M., Chien, P.-D., and Allen, E. B. 1998. GP-based software quality prediction. In Koza, J. R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M. H., Goldberg, D. E., Iba, H., and Riolo, R., editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 60–65. Madison, WI: AAAI, Morgan Kaufmann.

Fenton, N. E., and Pfleeger, S. L. 1997. *Software Metrics: A Rigorous and Practical Approach*, 2d edition. London: PWS Publishing.

Ganesan, K., Khoshgoftaar, T. M., and Allen, E. B. 1999. Case-based software quality prediction. *International Journal of Software Engineering and Knowledge Engineering*, forthcoming.

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.

Halstead, M. H. 1977. *Elements of Software Science*. New York: Elsevier.

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

Khoshgoftaar, T. M., and Allen, E. B. 1997. A practical classification rule for software quality models. Technical Report TR-CSE-97-56, Florida Atlantic University, Boca Raton, FL.

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

Khoshgoftaar, T. M., and Allen, E. B. 1998b. Predicting the order of fault-prone modules in legacy software. In *Proceedings of the Ninth International Symposium on Software Reliability Engineering*. Paderborn, Germany: IEEE Computer Society.

Khoshgoftaar, T. M., Allen, E. B., Goel, N., Nandi, A., and McMullan, J. 1996a. Detection of software modules with high debug code churn in a very large legacy system. In *Proceedings of the Seventh International Symposium on Software Reliability Engineering*, pages 364–371. White Plains, NY: IEEE Computer Society.

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

Khoshgoftaar, T. M., Allen, E. B., Jones, W. D., and Hudepohl, J. P. 1998b. Return on investment of software quality models. In *Proceedings 1998 IEEE Workshop on Application-Specific Software Engineering and Technology*, pages 145–150. Richardson, TX: IEEE Computer Society.

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., Allen, E. B., Kalaichelvan, K. S., and Goel, N. 1996d. Predictive modeling of software quality for very large telecommunications systems. In *Proceedings of the International Communications Conference*, volume 1, pages 214–219. Dallas, TX: IEEE Communications Society.

Khoshgoftaar, T. M., Bhattacharyya, B. B., and Richardson, G. D. 1992a. Predicting software errors during development using nonlinear regression models: A comparative study. *IEEE Transactions on Reliability* 41(3): 390–395.

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

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

Khoshgoftaar, T. M., and Munson, J. C. 1990. Predicting software development errors using software complexity metrics. *IEEE Journal on Selected Areas in Communications* 8(2): 253–261.

Khoshgoftaar, T. M., Munson, J. C., Bhattacharya, B. B., and Richardson, G. D. 1992b. Predictive modeling techniques of software quality from software measures. *IEEE Transactions on Software Engineering* 18(11): 979–987.

Khoshgoftaar, T. M., Munson, J. C., and Lanning, D. L. 1994b. Alternative approaches for the use of metrics to order programs by complexity. *Journal of Systems and Software* 24(3): 211–221.

Khoshgoftaar, T. M., Pandya, A. S., and Lanning, D. L. 1995. Application of neural networks for predicting faults. *Annals of Software Engineering* 1: 141–154.

Leake, D. B. 1996. CBR in context: The present and future. In Leake, D. B., editor, *Case-Based Reasoning: Experiences, Lessons, and Future Directions*, chapter 1, pages 3–30. Cambridge, MA: MIT Press.

McCabe, T. J. 1976. A complexity measure. *IEEE Transactions on Software Engineering* SE–2(4): 308–320.

McCabe, T. J., and Butler, C. W. 1989. Design complexity measurement and testing. *Communications of the ACM* 32(12): 1415–1425.

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

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

Myers, R. H. 1990. *Classical and Modern Regression with Applications*. Duxbury Series. Boston: PWS-KENT Publishing.

Nejmeh, B. A. 1988. NPATH: A measure of execution path complexity and its applications. *Communications of the ACM* 31(2): 188–200.

Ohlsson, N., and Alberg, H. 1996. Predicting fault-prone software modules in telephone switches. *IEEE Transactions on Software Engineering* 22(12): 886–894.

Ohlsson, N., Helander, M., and Wohlin, C. 1996. Quality improvement by identification of fault-prone modules using software design metrics. In *Proceedings of the Sixth International Conference on Software Quality*, pages 2–13, Ottawa, Ontario, Canada. Sponsored by ASQC.

Pfleeger, S. L. 1995. Experimental design and analysis in software engineering. *Annals of Software Engineering* 1: 219–253.

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*. New York: John Wiley and Sons.

Shepperd, M., and Ince, D. 1991. Design metrics and software maintainability: An experimental investigation. *Journal of Software Maintenance: Research and Practice* 3(4): 215–232.

Votta, L. G., and Porter, A. A. 1995. Experimental software engineering: A report on the state of the art. In *Proceedings of the Seventeenth International Conference on Software Engineering*, pages 277–279. Seattle, WA: IEEE Computer Society.

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.