International Journal of Reliability, Quality and Safety Engineering
© World Scientific Publishing Company

LOGISTIC REGRESSION MODELING OF SOFTWARE QUALITY

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

and

EDWARD B. ALLEN
Department of Computer Science and Engineering
Florida Atlantic University
Boca Raton, Florida USA
Email: edward.allen@computer.org

Received (received date)
Revised (revised date)

Reliable software is mandatory for complex mission-critical systems. Classifying modules as fault-prone, or not, is a valuable technique for guiding development processes, so that resources can be focused on those parts of a system that are most likely to have faults.

Logistic regression offers advantages over other classification modeling techniques, such as interpretable coefficients. There are few prior applications of logistic regression to software quality models in the literature, and none that we know of account for prior probabilities and costs of misclassification. A contribution of this paper is the application of prior probabilities and costs of misclassification to a logistic regression-based classification rule for a software quality model.

This paper also contributes an integrated method for using logistic regression in software quality modeling, including examples of how to interpret coefficients, how to use prior probabilities, and how to use costs of misclassifications. A case study of a major subsystem of a military, real-time system illustrates the techniques.

Keywords: Software Process; Process Measures; Software Metrics; Fault-prone Modules; Software Reuse; Spiral Life Cycle; Software Quality Modeling; Logistic Regression.

1. Introduction

For mission-critical systems to be highly reliable, embedded software must also be highly reliable. Software faults are not due to wear and tear during operations, but result from mistakes in design and coding decisions during development and enhancement. Early classification of modules as fault-prone or not is a valuable technique for guiding development processes, so that resources can be focused to remove faults before the software becomes operational. Software quality models are a tool for doing this.

Logistic regression offers easier interpretation compared to other classification techniques.[1] There are few prior applications of logistic regression to software quality models in the literature,[2,3] and none that we know of that account for prior probabilities and costs of misclassification. Previous software quality modeling classification studies by other researchers have used uniform prior probabilities and equal costs for all kinds of misclassifications. In a preliminary case study, prior probabilities and costs of misclassification improved software quality models based on nonparametric discriminant analysis.[4] A contribution of this research is confirmation that the same principles apply to logistic regression.

Another contribution of this research is presentation of an integrated method for using logistic regression in software quality modeling, including examples of how to interpret logistic regression coefficients, how to use prior probabilities, and how to use costs of misclassifications.

A case study was based on a large subsystem of a tactical military system, the Joint Surveillance Target Attack Radar System, JSTARS. It is an embedded, real time application. The objective of our software quality model was to predict at the beginning of integration whether each module will be considered *not fault-prone* or *fault-prone* at the end of the current development cycle. With such predictions, one can focus review, integration, and testing resources on high risk parts of the system. The independent variables were measures of development processes prior to integration.[5]

The remainder of this paper presents general principles, a case study, and conclusions. The case study is a step by step example of how to prepare data and how to build, interpret, and validate a model.

2. Applying Logistic Regression

Logistic regression is a statistical modeling technique where the dependent variable has only two possible values.[6] Independent variables may be categorical, discrete, or continuous. In software quality modeling, we usually consider a module as an "observation". Our dependent variable is *Class*, which has only two possible values: a module is a member of one group or the other. In this paper, we will use the groups *not fault-prone* and *fault-prone*; other groups may be used in other circumstances.

A classification model predicts membership in one group or the other. However, since a model is not likely to be perfect, some modules will probably be misclassified by the model, compared to actual group membership. Being consistent with terminology in our previously published work,[7] Type I errors misclassify modules that are actually *not fault-prone* as *fault-prone*. Type II errors misclassify modules that are actually *fault-prone* as *not fault-prone*.

There are several possible strategies for encoding categorical independent variables. For binary categorical variables, we encode the categories as the values zero and one. Discrete and continuous variables may be used directly. Let x_j be the j^{th} independent variable, and let $\mathbf{x}_i$ be the vector of the i^{th} module's independent variable values.

2.1. *Building a Model*

We designate a module being *fault-prone* as an "event". Let p be the probability of an event, and thus, $p/(1-p)$ is the odds of an event. The logistic regression model has the form

$$\log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x_1 + \ldots + \beta_j x_j + \ldots + \beta_m x_m \tag{1}$$

where log means natural logarithm and m is the number of independent variables. Let b_j be the estimated value of β_j. This model can be restated as

$$p = \frac{\exp(\beta_0 + \beta_1 x_1 + \ldots + \beta_m x_m)}{1 + \exp(\beta_0 + \beta_1 x_1 + \ldots + \beta_m x_m)} \tag{2}$$

which implies each x_j is assumed to be monotonically related to p. Fortunately, most software engineering measures do have a monotonic relationship with faults that is inherent in the underlying processes.

Given a list of candidate independent variables and a threshold significance level, α, some of the estimated coefficients may not be significantly different from zero. Such variables should not be included in the final model. The process of choosing significant independent variables is called "model selection". Stepwise logistic regression is one method of model selection which uses the following procedure.

Initially, estimate a model with only the intercept. Evaluate the significance of each variable not in the model. Add to the model the variable with the largest chi-squared p-value which is better than a given threshold significance level. Estimate parameters of the new model. Evaluate the significance of each variable in the model. Remove from the model the variable with the smallest chi-squared p-value whose significance is worse than a given threshold significance level. Repeat until no variables can be added or removed from the model. Tests for adding or removing a variable are based on an adjusted residual chi-squared statistic for each variable, comparing models with and without the variable of interest.[6]

We calculate maximum likelihood estimates of the parameters of the model, b_j, using the iteratively reweighted least squares algorithm. Other algorithms are also available to calculate maximum likelihood estimates. The same algorithm is used both in the stepwise procedure and for the final model. The estimated standard deviation of a parameter can be calculated, based on the log-likelihood function.[8] These calculations are provided by commonly available statistical packages, such as SAS.[9]

The odds ratio, ψ_j, is a statistic that indicates the relative effect on the odds of an event by a one unit change in the j^{th} independent variable.[6] For example, suppose x_j is a binary variable with values zero or one. Let $p(1)$ be the probability of an event when $x_j = 1$, and let $p(0)$ be the probability of an event when $x_j = 0$, other things being equal.

$$\psi_j = \frac{p(1)/(1-p(1))}{p(0)/(1-p(0))} \tag{3}$$

Thus, the odds of an event for an observation with $x_j = 1$ is ψ_j times the odds of an event for $x_j = 0$. The odds ratio is estimated by

$$\hat{\psi}_j = e^{b_j} \tag{4}$$

The odds ratio illustrates how straightforward interpretation is one of logistic regression's advantages.

2.2. *Prior Probabilities*

We often model software development as a process that produces modules which are random samples from a large population of modules that might have been developed. From a Baysian viewpoint, our knowledge of the population is embodied in prior probabilities of class membership, *i.e.*, "prior" to knowing the attributes of any modules in the sample. Logistic regression calculates the probability of being *fault-prone* based on module attributes, but this is not enough. A decision rule that minimizes misclassifications should also take into account the overall proportions of the underlying populations for each group, as well.[10]

Let π_{fp} be the prior probability of membership in the *fault-prone* group, and let π_{nfp} be the prior probability of membership in the *not fault-prone* group. We want to choose prior probabilities that are appropriate for each set of modules that we classify. When a large fit data set is representative of the population, we choose the prior probabilities, π_{fp} and π_{nfp}, to be the proportion of fit modules in each group. Otherwise, we make adjustments according to our knowledge about the data set. If we do not have information about the population of modules, we choose the uniform prior, $\pi_{fp} = \pi_{nfp} = 0.5$. Most software engineering classification models in the literature by other researchers use the uniform prior.[2,11,12,13,14] Choosing prior probabilities for software quality models is a contribution of this research.

There are several module sets of interest: the fit data set, the validation data set, and application data sets. When judging quality of fit, we use priors based on the fit data set. When validating a model with an independent data set, we consider whether or not its proportion of fault-prone modules should be similar to the fit data set's. If so, we use prior probabilities based on the fit data set. When applying the model to other projects or subsequent releases (application data sets), we may adjust prior probabilities based on our knowledge of project attributes and plans.

2.3. *Costs of Misclassifications*

In software engineering, the cost for acting on each type of erroneous prediction will depend on the process improvement technique that uses the prediction.[4] For example, suppose we apply reliability enhancement processes, such as additional reviews, to modules identified as *fault-prone*. The cost of a Type I misclassification is to waste time on additional reviews of a module that is actually *not fault-prone*. The cost of a Type II misclassification is the lost opportunity to review a *fault-prone* module and detect its faults earlier in development. A fault that might have been discovered during a review will end up being discovered later in the project, when the cost of fixing it is much greater.

Let C_I be the cost of a Type I misclassification, and let C_{II} be the cost of a Type II misclassification. Logistic regression calculates the probability of being *fault-prone* based on attributes of the modules. A classification rule that minimizes the number of misclassifications may include prior probabilities, but this is not enough. Not all misclassifications are equivalent. Some types cost more than others. An optimal classification rule should minimize the expected cost of misclassifications (ECM), given by

$$ECM = C_I \pi_{nfp} \Pr(fp|nfp) + C_{II} \pi_{fp} \Pr(nfp|fp) \tag{5}$$

where $\Pr(fp|nfp)$ and $\Pr(nfp|fp)$ are the Type I and Type II misclassification rates, respectively. When costs of misclassification are unknown or unimportant, we choose equal costs, $C_I / C_{II} = 1$.

2.4. *Classification*

Given a logistic regression model, a module can be classified as *fault-prone* or not, by the following procedure:

1. Calculate $\hat{p}/(1-\hat{p})$ using

$$\log\left(\frac{\hat{p}}{1-\hat{p}}\right) = b_0 + b_1 x_1 + \ldots + b_j x_j + \ldots + b_m x_m \tag{6}$$

2. Assign the module by a classification rule that minimizes the expected cost of misclassification.

$$Class(\mathbf{x}_i) = \begin{cases} \textit{fault-prone} & \text{If } \frac{\hat{p}}{1-\hat{p}} \geq \left(\frac{C_I}{C_{II}}\right)\left(\frac{\pi_{nfp}}{\pi_{fp}}\right) \\ \textit{not fault-prone} & \text{Otherwise} \end{cases} \tag{7}$$

This rule minimizes the expected cost of misclassification (Eq. (5)) as shown in Johnson and Wichern,[10] and generalizes when priors or costs are unknown or unimportant. We have found that this rule is valuable in software engineering applications.

3. Case Study

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.[15,5] The system performs ground surveillance, providing real time detection, location, classification, and tracking of moving and fixed objects. The system was developed under the spiral life cycle model.[16] 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.

The following sections illustrate our method for applying logistic regression to software quality modeling. The major steps are prepare data sets, build a model, interpret the model, and validate the model. Then the model is ready for application to similar projects or subsequent spiral life cycle iterations.

We have also successfully applied the same method to other industry projects which used product and/or process attributes as independent variables.

3.1. *Prepare Data*

This major step consists of the following detailed steps: collect configuration management data, retrieving source code if necessary; analyze source code; collect problem reporting system data; calculate variables; and prepare *fit* and *test* data sets.

Collect configuration management data. The subject of this case study was the set of FORTRAN modules from a major subsystem of the final Build, totaling 1,643 modules and accounting for 38% of FORTRAN modules in JSTARS. Each module was a source file with one compilation unit, such as a subroutine. The project's configuration management system identified module versions for the baseline and ending versions of the system. Only modules under study that existed in both of these versions were selected.

Analyze source code. The configuration management system also retrieved archived source code for this case study. 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. Future work will consider software product metrics.

Collect problem reporting system data. The project uses a problem reporting system to track and control modifications to software, documents, and other software development objects. The primary raw data for the case study were Software Trouble Reports (STR). STRs are generated as problems are discovered by reviews, integration, and testing. Whenever source code is changed, the reason for the change and the version of the affected module is recorded in the STR.

One of the STR attributes is its "Activity". We categorized Activities of interest into four general reasons that a module was modified.[5] FAULTS means code changed due to developer errors; REQUIREMENT means code changed due to unplanned requirements changes; PERFORMANCE means code changed due to inadequate speed or capacity; and DOCUMENTATION means code changes were mandated during documentation changes. Our study included only those STRs that resulted in modification to declarations or executable statements of the code.

Calculate variables. 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.

The dependent variable of the model was defined as class membership where

$$Class = \begin{cases} \textit{not fault-prone} & \text{If } \textit{Faults} < \textit{threshold} \\ \textit{fault-prone} & \text{If } \textit{Faults} \geq \textit{threshold} \end{cases} \tag{8}$$

where *threshold* was chosen according to project-specific criteria. After discussions with project engineers, a classification *threshold* of two faults was selected. In other words, the project engineers considered approximately one quarter of the modules to be *fault-prone*. Another threshold might be appropriate for another project. *Class* is known at the time of the ending version.

The cumulative numbers of STRs that affected code prior to the baseline version are attributes of a module's process history as follows. Let *BaseFlts*, *BaseReq*, *BasePerf*, and *BaseDoc* be the number of FAULTS, REQUIREMENT, PERFORMANCE, and DOCUMENTATION STRs, respectively.

Since each selected STR resulted in changes to code, and changes to code are opportunities for faults, a statistical relationship to *Faults* is plausible. 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 involves creating a new design that balances minimal disruption to the existing design, conformance to requirements, and flexibility for the future. Improving performance depends on an analysis of run-time behavior, rather than functionality. Diagnosing and correcting implementation discrepancies discovered during documentation revisions entails a detailed view of the code. Thus, these baseline variables are related to various aspects of software engineering which may affect the occurrence of faults.

We define categorical variables to model reuse from the prior Build. New modules were created during development of the current Build.

$$IsNew = \begin{cases} 1 & \text{If module did not exist in ending version of prior Build} \\ 0 & \text{Otherwise} \end{cases} \tag{9}$$

Preexisting modules with some changed code were reused with modifications. If a module had no code changed between the ending version of the prior Build and the baseline version of the current Build, then it was reused as an object.

$$IsChg = \begin{cases} 0 & \text{If no changed code since prior Build} \\ 1 & \text{Otherwise} \end{cases} \tag{10}$$

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.

$$Age = \begin{cases} 0 & \text{If module is new} \\ 1 & \text{If module was new in the prior Build} \\ 2 & \text{Otherwise} \end{cases} \tag{11}$$

Our data did not include information on whether a module's age was more than two Builds.

This project had data to support calculating the above variables. Other projects might have data for other process variables.

Prepare data sets. Data splitting is a technique for evaluating model accuracy when data on a similar subsequent project is not available. We impartially divided the modules into *fit* and *test* data sets. The *fit* data set was used to build a model, and the *test* data set was used to validate it. The *fit* data set had two thirds of the modules (1,096), and the *test* data set had the remaining third (547). (Other split ratios may be appropriate in other studies.)

3.2. *Build Model*

This major step consists of the following detailed steps: select significant independent variables; and estimate parameters of the final logistic regression model.

Select significant independent variables. The candidate 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*). Independent variables were selected in the order shown in Equation (12) below, using stepwise logistic regression at the $\alpha = 0.15$ significance level. In this study, we did not consider interactions among independent variables. This is a topic for further research.

Estimate parameters of the final logistic regression model. Recall that p is the probability that a module is a member of the *fault-prone* class. Logistic regression estimated the following model based on the *fit* data set.

$$\begin{aligned} \log\left(\frac{\hat{p}}{1-\hat{p}}\right) &= -1.611 + 0.247\, BaseFlts + 0.728\, IsNew + 0.779\, BaseReq \\ &\quad -0.560\, Age + 0.457\, IsChg \end{aligned} \tag{12}$$

Table 1 shows each estimated parameter, b_j, its standard deviation, s_j, and its odds ratio, ψ_j. This model with these coefficients is only applicable to our case study project. The modeling method is generally applicable, but each software development project must build and calibrate its own model.

Table 1. Model

Variable	Coeff b_j	StdDev s_j	OddsRatio ψ_j
Intercept	-1.611	0.337	—
BaseFlts	+0.247	0.032	1.280
BaseReq	+0.779	0.206	2.180
IsNew	+0.728	0.323	2.070
IsChg	+0.457	0.242	1.580
Age	-0.560	0.187	0.571

Other variables were not significant at 15% level.

3.3. *Interpret the Model*

This major step interprets the following aspects of a model: the significance of selected independent variables, the signs of coefficients, and the odds ratios of the independent variables.

Interpret the significance of selected variables. *BaseFlts* was the first variable selected, implying that it was more closely correlated to *Class* than the other variables. This is also indicated by fact that it has the smallest standard deviation relative to the size of the coefficient. Its relationship to *Class* 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.

The significance of *BaseReq* confirms conventional wisdom and earlier research on the relationship between enhancements and faults.[17] Unplanned requirements changes often disrupt the development process, resulting in more faults to be discovered later.

IsNew, *IsChg*, and *Age* were also significant. New modules have not been tested as much as older modules, and changed modules are often changed by someone other than the original designer, risking misunderstandings.

BasePerf and *BaseDoc* were not included in this model at significance level $\alpha = 0.15$, and thus, are not strongly related to faults discovered during integration and testing.

Interpret the sign of coefficients. Since p was defined as the probability of being *fault-prone*, a negative intercept, b_0, means that if we ignore the other variables, we expect a typical module to be *not fault-prone*, which is consistent with the small proportion of *fault-prone* modules in the *fit* data set. A positive coefficient,

$b_1, \ldots, b_m$, means that larger values of that variable were associated with worse reliability. *Age* was the only variable with a negative coefficient. This implies that older modules were more reliable, because the probability of being *fault-prone* was less. This was expected since they have had more testing and operational use.

Interpret the odds ratio of variables. The odds ratio for *IsChg* was 1.580. This indicates that the odds of being *fault-prone* for changed modules was about 58% more than the odds for unchanged modules.

The odds ratios for *BaseFlts* and *BaseReq* indicate the influence of an additional baseline STR on the likelihood of faults during integration and testing. Suppose modules A and B had the same process history except B had one more REQUIREMENT STR than A. The odds ratio of *BaseReq* was 2.180. Therefore, the odds of B being *fault-prone* would be more than twice the odds of A. Since the odds ratio of *BaseReq* was larger than that of *BaseFlts*, we conclude that a requirements change prior to the baseline version had a greater expected impact on the likelihood of faults during integration and testing than fixing a bug.

3.4. *Validate the Model*

This major step consists of these detailed steps: select priors ratio and cost ratio; classify each module in the *test* data set; and compare the predicted values with the actual values.

Select the ratio of prior probabilities and the ratio of costs of misclassification. We considered both uniform and nonuniform priors. For nonuniform priors, we assumed the probability of *fault-prone* modules in the *test* data set was the same as the proportion in the *fit* data set. We considered a wide range of misclassification cost ratios. The *fit* data set had 809 *not fault-prone* modules and 287 *fault-prone modules.*

Classify each module in the *test* data set. Once parameters are estimated, one can input values for the independent variables into Equation (12) and then apply the decision rule in Equations (7) to classify a module. Validation of a model should be done with an independent *test* data set which also contains values for both the dependent variable and independent variables. Validation tells us the level of accuracy to expect when applying the model to a similar data set at the beginning of integration, when the actual class memberships are not known.

Compare the predicted values with the actual values. The model was applied to the *test* data set. Table 2 and Figure 1 compare the actual with the predicted classifications for a range of values of C_I/C_{II}.

Since only about one quarter of the modules in the fit data set were actually *fault-prone*, the case of uniform priors ($\pi_{nfp}/\pi_{fp} = 0.5/0.5$) and equal misclassification costs ($C_I/C_{II} = 1/1$) was similar to the case of ($\pi_{nfp}/\pi_{fp} = 809/287$) and

Table 2. Validation

test data set

Number of modules/percent

			Misclassifications		
π_{nfp}/π_{fp}	C_I/C_{II}	$(\pi_{nfp}/\pi_{fp})(C_I/C_{II})$	Type I	Type II	Overall
0.5/0.5	1/1	1.000	23	106	129
			5.69%	74.13%	23.58%
809/287	1/1	2.829	5	123	128
			1.24%	86.01%	23.40%
809/287	1/2	1.409	13	116	129
			3.22%	81.12%	23.58%
809/287	1/3	0.940	24	104	128
			5.94%	72.73%	23.40%
809/287	1/4	0.705	37	90	127
			9.16%	62.94%	23.22%
809/287	1/5	0.564	94	60	154
			23.27%	41.96%	28.15%
809/287	1/6	0.470	99	55	154
			24.50%	38.46%	28.15%
809/287	1/7	0.403	102	48	150
			25.25%	33.57%	27.42%
809/287	1/8	0.352	119	39	158
			29.46%	27.27%	28.88%
809/287	1/9	0.313	119	39	158
			29.46%	27.27%	28.88%
809/287	1/10	0.282	137	29	166
			33.91%	20.28%	30.35%
809/287	1/15	0.188	194	22	216
			48.02%	15.38%	39.49%
809/287	1/20	0.141	227	9	236
			56.19%	6.20%	43.14%
809/287	1/30	0.094	296	4	300
			73.27%	2.80%	54.84%
809/287	1/50	0.056	404	0	404
			100.00%	0.00%	73.86%

404 *not fault-prone* modules (base of Type I %)

143 *fault-prone* modules (base of Type II %)

547 modules, total (base of Overall %) in the *test* data set

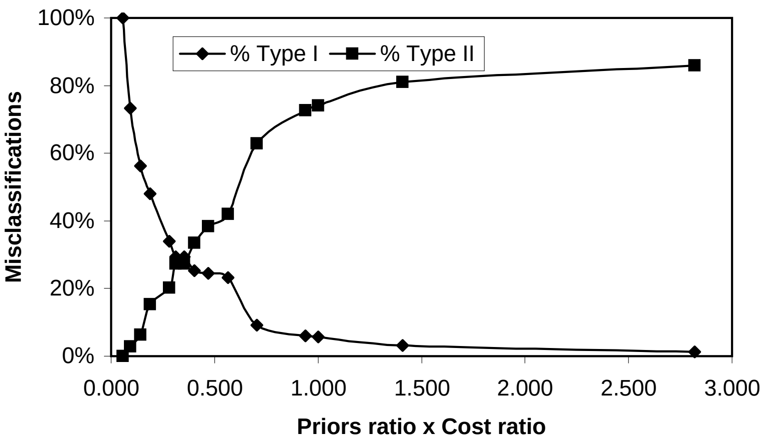

Figure 1: The Effects of Costs on Misclassification Rates

$(C_I/C_{II} = 1/3)$. For this data, uniform priors and equal costs was not a satisfactory classification rule for practical use because the Type II misclassification rate (74.13%) is so large that a reliability enhancement effort would miss most of the *fault-prone* modules.

When we use nonuniform priors and let misclassification costs be equal, our classification rule minimized the overall misclassification rate for the fit data set. This model was also not useful due to a high Type II misclassification rate (86.01%).

The models on each end of the range of cost ratios shown in Table 2 were not useful to the project, because one type of misclassification rate or the other was so extreme. For a cost ratio of 1/2, the Type II misclassification rate was 81.12%, and for a cost ratio of 1/30, the Type I misclassification rate was 73.27%. The cost ratios in the middle are probably often more realistic. The cost of reliability enhancement early in the life cycle is on the order of man-hours, but late in the life cycle, is on the order of man-days per module. For example, a cost ratio of 1/8 has balanced misclassification rates of 29.46% for Type I and 27.27% for Type II, for an overall rate of 28.88%. Future work will seek to improve model performance by adding software product metrics and variable interactions to the list of candidate independent variables.

Another study with this data set found that a nonparametric discriminant analysis model had similar accuracy.[4] Similar to this study, the classification rule with $C_I/C_{II} = 1$ had poor accuracy. The rule with $C_I/C_{II} = 1/10$ had a Type I misclassification rate of 32.4%, a Type II rate of 26.6%, and an overall misclassification rate of 30.9%. In Table 2, the logistic regression model had a Type I misclassification rate of 33.9%, a Type II rate of 20.3%, and an overall misclassification rate of 30.4%. Even though nonparametric discriminant analysis and logistic regression balanced the misclassification rates differently for a given cost ratio, overall accuracy was similar. On the one hand, nonparametric discriminant analysis does not assume a functional form as logistic regression does, but on the other hand, logistic regression is more easily interpreted than nonparametric discriminant analysis.

Apply the model to a current project. Having validated the model, one can then use it to predict whether each module in a current similar project is fault-prone or not. Validation results indicate the level of accuracy one can expect. The predictions, in turn, may be the basis for extra reviews or other reliability enhancement measures.

Consider the accuracy of our model at $C_I/C_{II} = 1/8$. Suppose a current project actually has 738 *not fault-prone* modules and 262 *fault-prone* modules for a total of 1,000 (i.e., similar proportions to π_{nfp} and π_{fp}). The expected cost of enhancement efforts would be $(738)(0.2946) = 217$ cost units wasted on *not fault-prone* modules, and $(262)(1 - 0.2727) = 191$ cost units invested in *fault-prone* modules, for a total investment of 408 cost units. The expected cost avoidance for the enhanced *fault-prone* modules would be $(191)(8) = 1,528$ cost units for a profit of 1,528-408=1,120 cost units. Thus, this level of accuracy could be useful to a similar current project.

4. Conclusions

Reliable software is mandatory for mission-critical software such as tactical military systems. During development, reliability enhancement processes try to find faults before the system becomes operational. Classifying modules as fault-prone, or not, is a valuable technique for guiding such development processes.

This research contributes an integrated method for using logistic regression in software quality modeling to predict whether each module will be *fault-prone* or not. A case study of a major subsystem of JSTARS, a tactical real-time system, illustrates the techniques. We saw that logistic regression parameters can be interpreted, and that using prior probabilities and costs of misclassification can improve model accuracy. This is the first research that we know of that uses prior probabilities and costs of misclassification in a logistic regression model of software quality.

The case study's model validation showed misclassification rates over a range of the ratios. Prior probabilities were set to the proportions of being *fault-prone* or not in the *fit* data set. At one end of the range, the costs of Type I and Type II misclassifications were equal, and the model was not useful due to excessive Type II misclassifications. At the other extreme, Type I misclassifications were excessive. We suspect that the typical relative costs of Type I to Type II misclassifications is on the order of 1:10, which, for this data, gave approximately balanced model accuracy. This case study gives empirical evidence that logistic regression, with thoughtful consideration to prior probabilities and costs of misclassification, can be used for software quality modeling. This case study also illustrates the utility of process measures alone in such models.

Future research will seek to improve model accuracy by combining software product metrics, process measures, and interaction terms in the same logistic regression models. A detailed comparison of classification modeling methods is also a research goal.

Acknowledgments

We thank Robert Halstead for his support and encouragement, and Ronald Flass and Gary P. Trio for helpful discussions regarding collected data and the software development process. We thank Lionel C. Briand for discussion regarding logistic regression. We thank the anonymous reviewers for their thoughtful comments. This work was supported in part by a grant from Northrop Grumman. The findings and opinions in this paper belong solely to the authors, and are not necessarily those of the sponsor. Moreover, our results do not in any way reflect the quality of the sponsor's software products.

References

1. T. M. Khoshgoftaar and E. B. Allen. Classification techniques for predicting software quality: Lessons learned. In *Proceedings of the Annual Oregon Workshop on Software Metrics*, Coeur d'Alene, Idaho USA, May 1997. University of Idaho.
2. V. R. Basili, L. C. Briand, and W. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, October 1996.
3. P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, August 1993.
4. T. M. Khoshgoftaar and E. B. Allen. The impact of costs of misclassification on software quality modeling. In *Proceedings of the Fourth International Software Metrics Symposium*, pages 54–62, Albuquerque, New Mexico USA, November 1997. IEEE Computer Society.
5. T. M. Khoshgoftaar, E. B. Allen, R. Halstead, G. P. Trio, and R. Flass. Process measures for predicting software quality. *Computer*, 31(4):66–72, April 1998.
6. D. W. Hosmer, Jr. and S. Lemeshow. *Applied Logistic Regression*. John Wiley & Sons, New York, 1989.
7. T. M. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan, and N. Goel. Early quality prediction: A case study in telecommunications. *IEEE Software*, 13(1):65–71, January 1996.
8. R. H. Myers. *Classical and Modern Regression with Applications*. Duxbury Series. PWS-KENT Publishing, Boston, 1990.
9. M. E. Stokes, C. S. Davis, and G. G. Koch. *Categorical Data Analysis Using the SAS System*. SAS Institute, Cary, North Carolina USA, 1995.
10. R. A. Johnson and D. W. Wichern. *Applied Multivariate Statistical Analysis*. Prentice Hall, Englewood Cliffs, NJ, 3d edition, 1992.
11. C. Ebert. Classification techniques for metric-based software development. *Software Quality Journal*, 5(4):255–272, December 1996.
12. T. M. Khoshgoftaar and D. L. Lanning. A neural network approach for early detection of program modules having high risk in the maintenance phase. *Journal of Systems and Software*, 29(1):85–91, April 1995.
13. N. F. Schneidewind. Methodology for validating software metrics. *IEEE Transactions on Software Engineering*, 18(5):410–422, May 1992.
14. R. W. Selby and A. A. Porter. Learning from examples: Generation and evaluation of

decision trees for software resource analysis. *IEEE Transactions on Software Engineering*, 14(12):1743–1756, December 1988.
15. T. M. Khoshgoftaar, E. B. Allen, R. Halstead, and G. P. Trio. Detection of fault-prone software modules during a spiral life cycle. In *Proceedings of the International Conference on Software Maintenance*, pages 69–76, Monterey, CA, November 1996. IEEE Computer Society.
16. B. W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, May 1988.
17. D. L. Lanning and T. M. Khoshgoftaar. The impact of software enhancement on software reliability. *IEEE Transactions on Reliability*, 44(4):677–682, December 1995.

About the Authors

Taghi M. Khoshgoftaar is a professor of the Dept. of Computer Science and Engineering, Florida Atlantic University. 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, computational intelligence, computer performance evaluation, multimedia systems, and statistical modeling. He has published more than 150 refereed papers in these areas. He has been a principal investigator and project leader in a number of projects with industry, government, and other research-sponsoring agencies. He is a member of the Association for Computing Machinery, the American Statistical Association, and the IEEE (Computer Society and Reliability Society). He is the general chair of the 1999 International Symposium on Software Reliability Engineering (ISSRE'99). He has served on technical program committees of various international conferences, symposia, and workshops. He has served as North American editor of the *Software Quality Journal*, and is on the editorial board of the *Journal of Multimedia Tools and Applications*.

Edward B. Allen received the B.S. degree in engineering from Brown University in 1971, the M.S. degree in systems engineering from the University of Pennsylvania in 1973, and the Ph.D. degree in computer science from Florida Atlantic University in 1995. He is currently a Research Associate 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, software quality, and computer performance modeling. He has more than 50 refereed publications in these areas. He is a member of the IEEE Computer Society and the Association for Computing Machinery.