Modeling Software Quality: The Software Measurement Analysis and Reliability Toolkit

Taghi M. Khoshgoftaar*
Edward B. Allen
Jason C. Busboom
Florida Atlantic University
Boca Raton, Florida USA

Abstract

This paper presents the Software Measurement Analysis and Reliability Toolkit (SMART) which is a research tool for software quality modeling using case-based reasoning (CBR) and other modeling techniques.

Modern software systems must have high reliability. Software quality models are tools for guiding reliability-enhancement activities to high-risk modules for maximum effectiveness and efficiency. A software quality model predicts a quality factor, such as the number of faults in a module, early in the life cycle in time for effective action. Software product and process metrics can be the basis for such fault predictions. Moreover, classification models can identify fault-prone modules.

CBR is an attractive modeling method based on automated reasoning processes. However, to our knowledge, few CBR systems for software quality modeling have been developed. SMART addresses this area. There are currently three types of models supported by SMART: classification based on CBR, CBR classification extended with cluster analysis, and module-order models, which predict the rank-order of modules according to a quality factor.

An empirical case study of a military command, control, and communications applied SMART at the end of coding. The models built by SMART had a level of accuracy that could be very useful to software developers.

Keywords: software reliability, case-based reasoning, data clustering, module-order model, software quality models, multiple linear regression, analogy models, software tools, fault-prone modules

*Readers may contact the authors through Taghi M. Khoshgoftaar, Empirical Software Engineering Laboratory, Dept. of Computer Science and Engineering, Florida Atlantic University, Boca Raton, FL 33431 USA. Phone: (561)297-3994, Fax: (561)297-2800, Email: taghi@cse.fau.edu, URL: www.cse.fau.edu/esel/.

1 Introduction

Software reliability is essential for mission-critical and high-assurance systems. However, assuring reliability often entails time-consuming costly development processes. Reliability-improvement techniques include more rigorous design and code reviews, automatic test-case generation, more extensive testing, strategic assignment of key personnel, and reengineering of high-risk portions of a system. One cost-effective strategy is to target reliability-enhancement activities to those modules that are most likely to have problems [8]. A software *fault* is a defect in an executable product that causes a software failure. The amount of rework to correct faults is a priority concern to software developers. Rather than focus on the number of failures, they are interested in the number of faults discovered during testing or operations.

A *software quality model* has independent variables that can be measured earlier in the development life cycle than the dependent variable which indicates its quality. Examples variables include lines of code and McCabe cyclomatic complexity as independent variables, and the number of faults as a dependent variable. If one supplies the independent variable values for a module, one can calculate a prediction of the dependent variable's value. Prior research [1, 5, 20] has shown that software product and process metrics, collected early in the software development life cycle, can be the basis for fault predictions. Predicting the number of faults in each module is often not necessary. Much previous research has focused on classification models that identify *fault-prone* and *not fault-prone* modules [13]. In such models, *fault-prone* is usually defined via a threshold on the number of faults expected.

Various classification modeling techniques have been applied to software quality

data, such as discriminant analysis [13], logistic regression [12], decision trees [21], artificial neural networks [11], discriminant power [22], optimal set reduction [2], and fuzzy classification [4].

Case-based reasoning (CBR) is an alternative modeling method based on automated reasoning processes. It has proven useful in a wide variety of domains [17] including software cost estimation, software reuse, software design, and software help desk. CBR is especially useful when there is limited domain knowledge and when an optimal solution process is not known [18]. A recent book [58] presents the state of the art in CBR, the lessons learned from specific applications, and directions for the future. However, to our knowledge, few CBR systems for software quality modeling have been developed [6, 14].

The Software Measurement Analysis and Reliability Toolkit (SMART) is a domain-independent research tool for case-based reasoning and other modeling techniques, whose user interface is tailored for software quality modeling. An "analysis project" is our term for the user's task of building and using a software quality model for the benefit of a software development project. This paper presents SMART and an empirical case study which applied it to software quality modeling of a military software system.

2 Toolkit Architecture

The software architecture of SMART is shown in Figure 1. The tool is implemented using Microsoft Visual C++® and runs in a Windows 95® or Windows/NT® environment. There is a central data manager that handles the *fit* data, which is used to build the model, the *test* data, which is used to validate the model, and analysis-project data. The models are controlled via the graphical user interface and results may be in the form of

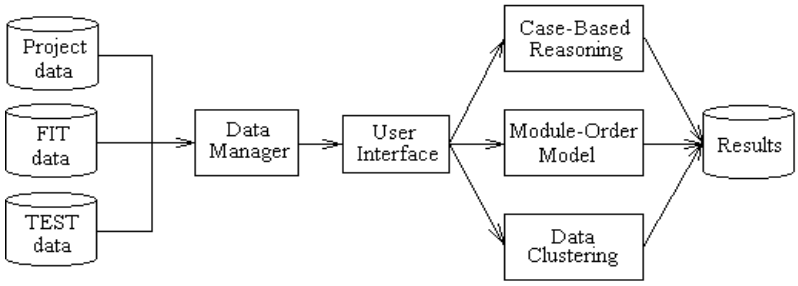

Figure 1: Tool architecture

a display, printed report, or data file.

The user interface is based on a "dialog-based property sheet". Figure 2 shows the General page. Tabs for the various types of models are along the top portion of the property sheet, while the current page is displayed below the tabs. Navigation through the menus and selection of the various available options is done with a mouse. Once satisfied with the options selected, the user may save the analysis project's data, run an experiment, or print out statistics about the current analysis project. There is also an online help document that explains how to use the tool.

There are currently three types of models supported by SMART. Details on each type of model are presented in the following sections. The first type of model is for classification based on case-based reasoning (CBR). This kind of model classifies a software module as *fault-prone* or not by comparing its attributes to a library of similar past modules. The second model extends the CBR classification model with cluster analysis [15]. This technique partitions the case library into clusters. An unclassified module is classified according to the closest cluster. The third type of model is the module-order model [9, 10], which predicts the rank-order of modules according to a quantitative dependent variable. It can also be used for classification.

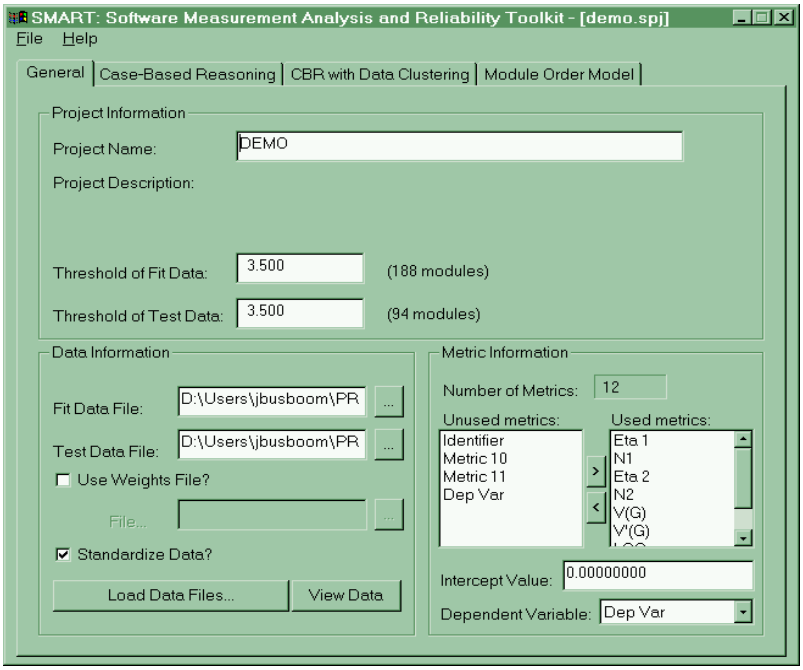

Figure 2: Project data

On the General page, the user specifies the Project Name and Project Description of the analysis project. Modeling parameters and results can be saved under this name. The user specifies the names of space-delimited ASCII files holding the data (Fit Data File and Test Data File). The names of fields are required as the first line of a data file, and each line thereafter is data for one software module. The *fit* data set consists of data from a past software development project including values of both dependent and independent variables. It is used to build a model and estimate model parameters, if any.

The *test* data set is used to simulate use of the model. It also consists of data from a past software development project. The independent variable values are used to make predictions as if the data were from a current software development project, and the actual dependent variable values are used to evaluate the accuracy of the predictions.

After a model is built and evaluated it can be used to predict the classes of a *current* data set. The Test Data File can be data from a current software development project whose independent variable values are known, but whose dependent variable values are not. Let i and j be indices for modules in data sets.

One of the data fields should be a software quality factor that the user identifies as the Dependent Variable, such as the number of faults. Let y_i be the actual value of the dependent variable for module i, and let $\hat{y}_i$ be its predicted value.

In our application, software metrics are the independent variables. The user selects the independent variables, indexed by $k = 1, \ldots, m$. Variables included in the model are listed as Used metrics, and those in the data file but not in the model are listed as Unused metrics.

The user can choose to standardize the independent variables so that all have the same unit of measure. Given a raw measurement, x_{ik}, its estimated mean, $\bar{x}_k$, and its estimated standard deviation, s_k, the standardized measurement is $z_{ik} = (x_{ik} - \bar{x}_k)/s_k$. The unit of measure is a standard deviation. In the discussion below, we use x_{ik} to mean either a raw or a standardized measurement, as selected by the user.

The actual class of a module is defined by whether or not the dependent variable is greater than a Threshold, θ, or not. The user specifies thresholds for the fit and test data sets. In many of our case studies, we use the same threshold for both. However, the tool provides additional flexibility. The actual class of a module is given by

$$Class_i = \begin{cases} \textit{not fault-prone} & \text{if } y_i < \theta \\ \textit{fault-prone} & \text{Otherwise} \end{cases} \tag{1}$$

Not all software metrics are equally related to software faults. It is possible that small changes in one attribute may strongly relate to the number of faults. The user can

specify a Weights File containing weights, w_k, for each independent variable in the model to account for various levels of importance. An Intercept Value, w_0, can also be specified on this page.

3 Case-Based Reasoning

A CBR system finds a solution to a new problem based on past experience, represented by *cases* in a *case library*. Each case is indexed for quick retrieval according to the problem domain. A *solution process* algorithm uses a *similarity function* to measure the relationship between the new problem and each case. The algorithm retrieves relevant cases and determines a solution to the new problem.

A CBR system can function as a software quality classification model. In our application, the problem is to assign a module to the correct class early in development, i.e., whether it is *fault-prone* or not. A good "solution" is a class assignment that turns out to be correct after fault data is known. A *case* consists of all available information on a module. This could include whether it is fault-prone or not, its product attributes, and its process history. The working hypothesis for a software quality model is this: a module currently under development is probably *fault-prone* if a module with similar attributes in an earlier version or similar project was considered to be *fault-prone*. Figure 3 depicts the SMART's Case-Based Reasoning page.

The user specifies the data set to be used as the Case Library, usually the fit data set. A module in the library is a *case*. Let c_{jk} be the value of the k^{th} independent variable for case j, and let $\mathbf{c}_j$ be the vector of independent variable values for case j.

The user also specifies the Target Data Set whose dependent variable values are to be

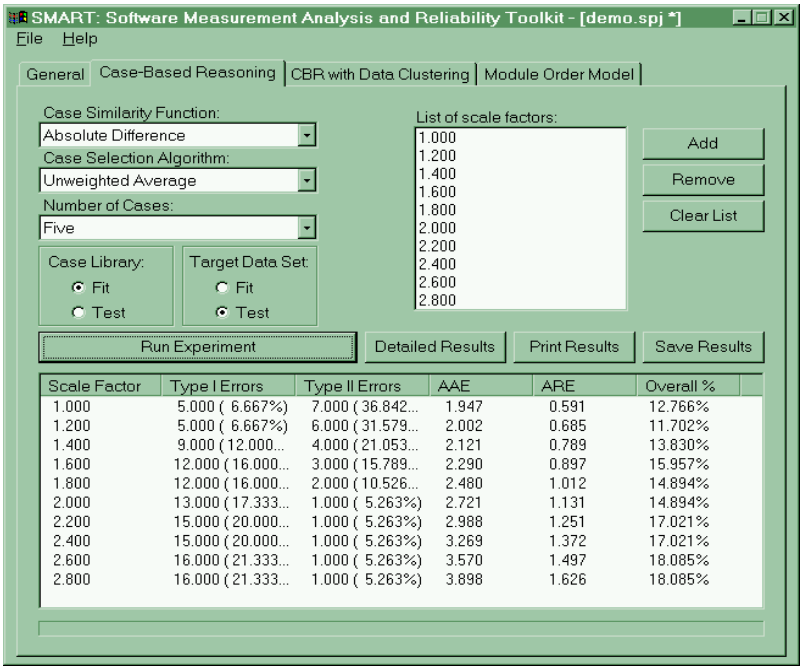

Figure 3: Case-based reasoning model

predicted. Let x_{ik} be the value of the k^{th} independent variable for target module i, and let $\mathbf{x}_i$ be the vector of independent variable values for module i. When using the model, the current set of modules is the target. When evaluating the model, the test data set is the target. When experimenting with parameter values during model development, the fit data set can be designated as the target.

The user selects a Case Similarity Function which calculates a distance, d_{ij}, between the current target module $\mathbf{x}_i$ and every case $\mathbf{c}_j$, where a small distance implies the modules are very similar. Based on the distances, SMART identifies a set of cases who are "nearest neighbors" to the current module, $\mathbf{x}_i$. The user selects the Number of Cases in this set. The user also selects a *solution algorithm* (Case Selection Algorithm) which predicts the dependent variable, $\hat{y}_i$ of the current module, $\mathbf{x}_i$, and predicts its class.

The user can provide a List of scale factors, α, for empirical investigation. Each value of α represents a distinct experiment over all target modules. Summary statistics for the experiments are listed, as shown in Figure 3.

A Type I misclassification is when a model classifies a module as *fault-prone* which is actually *not fault-prone*, and a Type II misclassification is when a model classifies a module as *not fault-prone* which is actually *fault-prone*. SMART provides counts and percentages. SMART summarizes the accuracy of the predicted dependent variable by average absolute error (AAE) and average relative error (ARE) which are given by

$$AAE = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i| \quad (2)$$

$$ARE = \frac{1}{n}\sum_{i=1}^{n}\frac{|\hat{y}_i - y_i|}{y_i + 1} \quad (3)$$

where n is the number of modules in the data set. The denominator of ARE has one added to avoid division by zero. The Overall misclassification rate is also provided.

3.1 Similarity Functions

Several measures of similarity are presented in the literature according to the problem domain, the availability of attribute data, and whether data types are categorical, discrete, real, etc. [16]. For quantitative attributes, the city-block distance and Euclidean distance [14] are commonly used. The Mahalanobis distance [3] has the advantage of explicitly accounting for correlations among attributes. SMART supports the following similarity functions.

Absolute Difference distance is also known as "city-block" distance or "Manhattan" distance. The distance is the weighted sum of the absolute value of the difference in

independent variables between a module and a case. The weights are those provided by the user. The absolute value is taken because distance is computed irrespective of direction.

$$d_{ij} \quad = \quad \sum_{k=1}^{m} w_k \, |c_{jk} - x_{ik}| \tag{4}$$

Euclidean Distance considers each independent variable as a dimension of an m-dimensional space. A module is represented by a point in this space. The Euclidean distance between a module and a case is their weighted distance in this space. The weights are those provided by the user.

$$d_{ij} \quad = \quad \left(\sum_{k=1}^{m} (w_k (c_{jk} - x_{ik}))^2 \right)^{1/2} \tag{5}$$

Linear Regression 1 is intended for when the weights, w_k, supplied by the user were estimated by multiple linear regression of the dependent variable as a function of case attributes, $\mathbf{c}_j$, using some other tool, such as a spreadsheet or a statistical modeling tool. The weights are used in a linear model.

$$\hat{f}(\mathbf{c}_j) \quad = \quad w_0 + w_1 \, c_{j1} + \ldots + w_m \, c_{jm} \tag{6}$$

The distance is the difference in the dependent variable values predicted by the linear model for the attributes of the module and the case.

$$d_{ij} \quad = \quad \hat{f}(\mathbf{c}_j) - \hat{f}(\mathbf{x}_i) = \sum_{k=1}^{m} w_k (c_{jk} - x_{ik}) \tag{7}$$

Linear Regression 2 is also intended for when the weights, w_k, supplied by the user were estimated by multiple linear regression of the dependent variable as a function of case attributes, $\mathbf{c}_j$. The distance is the difference between the actual y_j of the case and

the predicted $\hat{y}_i = \hat{f}(\mathbf{x}_i)$ of the module.

$$d_{ij} = y_j - \hat{f}(\mathbf{x}_i) = y_j - \sum_{k=1}^{m} w_k x_{ik} - w_0 \tag{8}$$

Mahalanobis Distance [3] is an alternative to Euclidean distance in cases where metrics may be poorly scaled or highly correlated. The distance is given by

$$d_{ij} = (\mathbf{c}_j - \mathbf{x}_i)' \, \mathbf{S}^{-1} \, (\mathbf{c}_j - \mathbf{x}_i) \tag{9}$$

where prime ($'$) means transpose, and $\mathbf{S}$ is the covariance matrix of the independent variables over all of the case library, and $\mathbf{S}^{-1}$ is its inverse. In the special case where the independent variables are uncorrelated and the variances are all the same, $\mathbf{S}$ is the identity matrix and the Mahalanobis distance is the Euclidean distance squared.

3.2 Nearest Neighbors

After the tool calculates distances between a module $\mathbf{x}_i$ and all cases using one of the similarity functions, the distances are sorted. The cases, $\mathbf{c}_j$, with the smallest distances, d_{ij}, are of primary interest. The set of nearest neighbors, $\mathcal{N}$, is an input to each solution algorithm below. The user selects the number of nearest neighbors, $n_{\mathcal{N}} \in \{1, 3, 5, 7, 9\}$. Based on a preliminary empirical investigation with software quality data, this range appears to be adequate.

3.3 Solution Algorithms

Each solution algorithm assigns the unclassified module to a class. Predictions of the dependent variable made by the solution algorithm can be scaled in order to improve

the accuracy of class predictions. Most of the solution algorithms below calculate $\hat{y}_i$ and then classify the module by

$$Class(\mathbf{x}_i) = \begin{cases} not\ fault\text{-}prone & \text{If } \alpha\hat{y}_i < \theta \\ fault\text{-}prone & \text{Otherwise} \end{cases} \tag{10}$$

To choose a preferred value of α, the user designates the fit data set as both the case library and the target data set, and supplies a list of candidate scale factors, α. The user can then choose a preferred value of α based on experiment results. When the target is the test data set and when the target is a current data set, the user can then specify the preferred scale factor, α.

SMART supports the following solution algorithms.

Unweighted average. This solution algorithm averages the dependent variable, y_j, of the closest $n_{\mathcal{N}}$ modules from the case library to form a value of $\hat{y}_i$ for the target module.

$$\hat{y}_i = \frac{1}{n_{\mathcal{N}}} \sum_{j \in \mathcal{N}} y_j \tag{11}$$

We are primarily concerned with classification, so the value predicted is not as important as the predicted class, given by Equation (10).

Inverse-distance weighted average. This solution algorithm utilizes the distance measures for the $n_{\mathcal{N}}$ closest cases as weights in a weighted average. Because a smaller distance means a better match, SMART weights each case in the nearest-neighbor set by a normalized inverse distance, δ_{ij}.

$$\delta_{ij} = \frac{1/d_{ij}}{\sum_{j \in \mathcal{N}} 1/d_{ij}} \tag{12}$$

$$\hat{y}_i = \sum_{j \in \mathcal{N}} \delta_{ij}\, y_j \tag{13}$$

The case that is most similar to the target module will naturally have the largest weight, and therefore play a larger role in the classification of the module by Equation (10).

Rank-weighted average. In this solution algorithm, the nearest neighbors are ranked according to their distances from the module. Rank $R_j = 1$ represents the best match while $R_j = n_{\mathcal{N}}$ represents the worst match among the nearest neighbors. The rank-weight, ρ_{ij}, is given by

$$\rho_{ij} = \frac{n_{\mathcal{N}} - R_j + 1}{\sum_{j \in \mathcal{N}} R_j} \tag{14}$$

$$\hat{y}_i = \sum_{j \in \mathcal{N}} \rho_{ij}\, y_j \tag{15}$$

The module is classified by Equation (10).

Majority vote. This solution algorithm assigns the unclassified module to the class of the majority of the cases in the nearest-neighbor set.

$$Class(\mathbf{x}_i) = \begin{cases} \textit{not fault-prone} & \text{if majority of } y_j < \theta \text{ for } j \in \mathcal{N} \\ \textit{fault-prone} & \text{Otherwise} \end{cases} \tag{16}$$

4 Data Clustering

In addition to the basic CBR model, SMART provides an extension by cluster analysis to enhance classification. The case library is partitioned into clusters according to the actual class of each case. SMART compares a module to the *fault-prone* cluster and the *not fault-prone* cluster and determines the closest group. SMART supports the same similarity functions and solution algorithms here as in the CBR model.

The Data Clustering page (DC) in Figure 4 is similar to the CBR page. Instead of scale factors, this page provides for a List of cost ratios, C_I/C_{II}. The user can calculate a cost ratio as the ratio of the cost of a Type I misclassification, C_I, to the cost of a Type II misclassification, C_{II}. However, this is often difficult to estimate, and therefore, SMART provides for experiments with a list of values. In software quality modeling, a

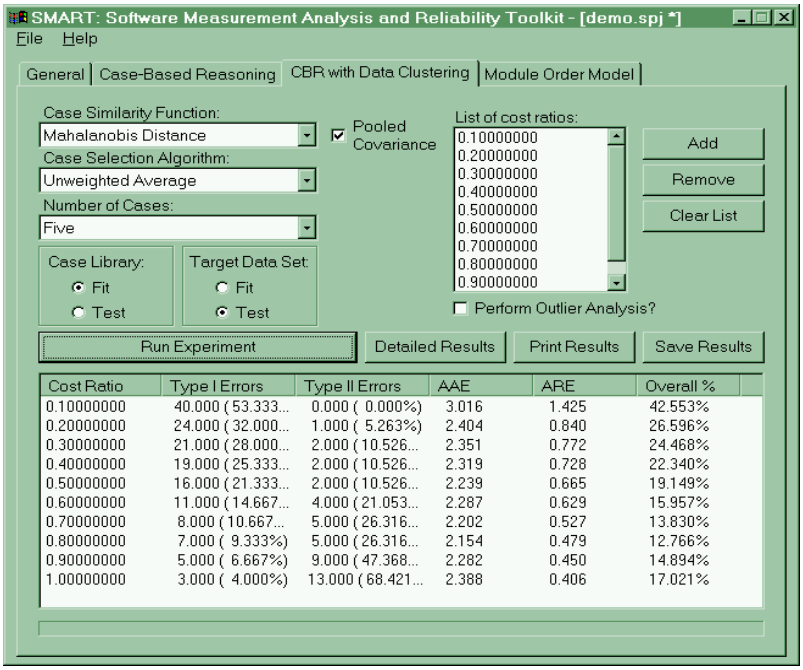

Figure 4: Data clustering model

Type II misclassification can be much more serious than a Type I, because the cost of releasing fault-prone modules to users is usually much more expensive than wasting effort enhancing low-risk modules. The user can experiment to choose a preferred C_I/C_{II} value.

The user has the option of using a Pooled Covariance matrix, $\mathbf{S}$, for the Mahalanobis distance measure. "Pooled" means that all data points from both clusters are used in determining the $\mathbf{S}$ matrix, and "non-pooled" means that two individual $\mathbf{S}$ matrices are calculated for the *fault-prone* cluster and the *not fault-prone* cluster.

The classification rule used here is based on our recent work with statistical classification techniques [12]. For an unclassified module, $\mathbf{x}_i$, let $d_{nfp}(\mathbf{x}_i)$ be the average distance to *not fault-prone* nearest-neighbor cases, and let $d_{fp}(\mathbf{x}_i)$ be the average distance

to *fault-prone* nearest-neighbor cases. The module is classified by

$$Class(\mathbf{x}_i) \quad = \quad \begin{cases} \textit{not fault-prone} & \text{If } \frac{d_{fp}(\mathbf{x}_i)}{d_{nfp}(\mathbf{x}_i)} > \frac{C_I}{C_{II}} \\ \textit{fault-prone} & \text{Otherwise} \end{cases} \tag{17}$$

where C_I/C_{II} is experimentally chosen. The user can choose C_I/C_{II} by a similar method as α, described above.

The Data Clustering page also offers Outlier Analysis. An outlier is a case that has abnormal attributes. One module at a time is removed from the case library and its class is predicted. If both the actual and predicted classes are the same, then the module is not an outlier, otherwise, it is marked as an outlier and is not used as part of the case library. This technique is repeatedly applied to all modules from the case library until all outliers are removed.

5 Module-Order Model

The objective of module-order models [9, 10] is robust predictions of the relative quality of each module, especially those with the highest risk. In particular, we are interested in the order of modules according to a dependent variable. This type of model focuses primarily on the degree to which a module is *fault-prone*, rather than membership in a particular class. A module-order model consists of the following components.

1. An underlying conventional quantitative software quality model. SMART currently uses a linear model, with weights, w_k, supplied by the user. The user can estimate the weights by multiple linear regression of the dependent variable as a function of fit data set attributes, $\mathbf{x}_i$, using some other tool, such as a spreadsheet or a

statistical modeling tool.

$$\hat{y}_i \quad = \quad w_0 + w_1\, x_{i1} + \ldots + w_m\, x_{im} \tag{18}$$

Other quantitative modeling techniques, such as nonlinear regression, regression trees, or computational intelligence techniques will be investigated in the future.

2. A ranking of modules according to the predicted dependent variable. Let R_i be the percentile rank of module i in a perfect ranking of modules according to y_i. Let $\widehat{R}(\mathbf{x}_i)$ be the predicted percentile rank of module i according to $\hat{y}_i$.

3. A procedure for evaluating the accuracy of a model's ranking over a target data set. (See the statistics described below.)

Figure 5 depicts the Module-Order Model page (MOM).

In our application, management will choose to enhance some percentage of 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. The user determines a range of percentiles that covers management's options for the last module, and chooses a set of representative cutoff percentiles (Extraction thresholds), c, from that range.

Statistics on this page are labeled C1 through C8. C1 is the sum of the actual dependent variable values in modules above the cutoff under perfect ranking, $G(c) = \sum_{i:R_i \geq c} y_i$. C2 is is the sum of the actual dependent variable values in modules above the cutoff under the predicted ranking, $\widehat{G}(c) = \sum_{i:\widehat{R}(\mathbf{x}_i) \geq c} y_i$. C3 is a measure of the model's accuracy at the given cutoff, $\phi(c) = \widehat{G}(c)/G(c)$. The variation in $\phi(c)$ over a range of c indicates the robustness of the model; small variation implies a robust model.

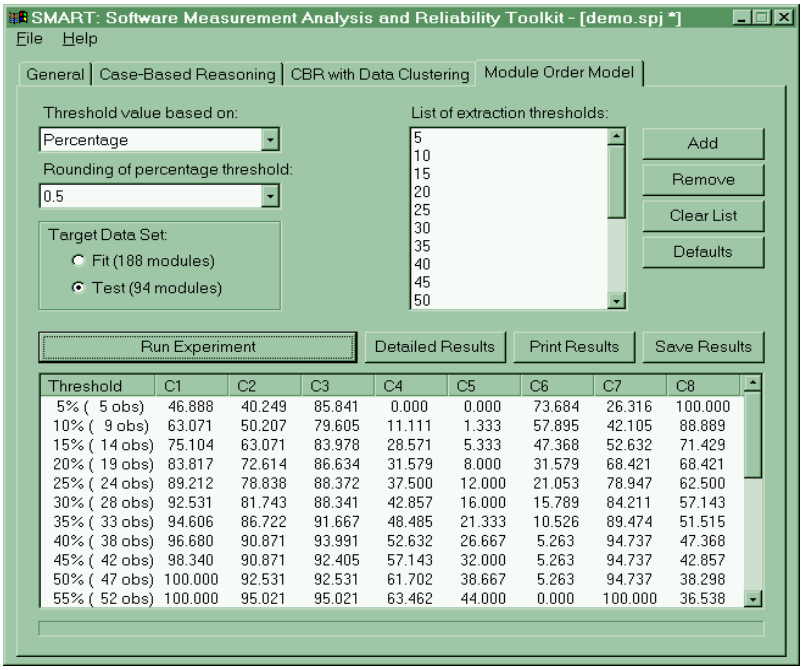

Figure 5: Module-order model

C5 is the Type I misclassification rate at the given cutoff. C6 is the Type II misclassification rate at the given cutoff. C7 is the model's *effectiveness* [10], defined as the proportion of actually *fault-prone* modules that the model correctly classifies. C8 is the model's *efficiency* [10], defined as the proportion of modules that the model classifies as *fault-prone* which are actually *fault-prone*. C4 is the model's "inefficiency", $1 - efficiency$.

6 Case Study

We studied a large system for command, control, and communications, written in Ada, which we call CCCS. Generally, a module was an Ada package, consisting of one or more procedures. A problem reporting system collected data on faults during the system

Table 1: Software product metrics for CCCS

Symbol	Description
η_1	Number of unique operators [7]
N_1	Total number of operators [7]
η_2	Number of unique operands [7]
N_2	Total number of operands [7]
$V(G)$	McCabe's cyclomatic complexity [19]
$V'(G)$	Extended cyclomatic complexity $V'(G) = V(G) +$ logical operators
LOC	Lines of code
$ELOC$	Executable lines of code

integration and test phase and during the first year of deployment. Each fault was attributed to a module. The top 20% of the modules contained 82.2% of the faults. 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 of each module. The number and selection of metrics was determined by available data collection tools.

We had measurements on 282 modules. Applying data splitting, we impartially partitioned this data into two subsets, two thirds of the modules (188) in the fit data set for building the models, and the remaining third (94 modules) in the test data set for evaluating their accuracy. This yielded adequate sample sizes for statistical purposes.

Table 1 describes the eight product metrics that we used as independent variables. The number of faults, y, was also collected for each module. A module was considered *fault-prone* if it had four or more faults during testing and operations. Another project might choose a different threshold.

We preferred values of α, C_I/C_{II}, and c, such that Type I and Type II misclassifica-

Table 2: Results

Model	Similarity	Solution	$n_{\mathcal{N}}$	Parameter	Type I	Type II
CBR	Absolute Diff.	Unweighted Avg.	5	$\alpha = 1.6$	16.0%	15.8%
DC	Mahalanobis	Unweighted Avg.	5	$C_I/C_{II} = 0.6$	14.7%	21.1%
MOM				$c = 0.3$	16.0%	15.8%

tion rates were approximately equal for the fit data set. Another project might prefer a different balance. Table 2 lists the accuracy of selected models based on the test data set. This level of accuracy at the end of coding could be very useful to software developers.

7 Conclusions

High software reliability is essential for modern software systems. However, assuring reliability is often time-consuming and costly. One cost-effective strategy is to target reliability-enhancement activities to high-risk modules. A software quality model predicts quality, such as the number of faults, early in the life cycle in time for effective action. Software product and process metrics, collected early in the software development life cycle, can be the basis for such fault predictions. Moreover, classification models can identify *fault-prone* and *not fault-prone* modules.

Case-based reasoning (CBR) is an attractive modeling method based on automated reasoning processes. However, to our knowledge, few CBR systems for software quality modeling have been developed. This paper presents the Software Measurement Analysis and Reliability Toolkit (SMART) which is a research tool for case-based reasoning and other modeling techniques, whose user interface is tailored for software quality modeling. The software architecture of SMART includes a central data manager and models

controlled via the graphical user interface. There are currently three types of models supported by SMART: classification based on case-based reasoning (CBR), the CBR classification model extended with cluster analysis, and the module-order model, which predicts the rank-order of modules according to a quantitative dependent variable.

An empirical case study applied SMART to software quality modeling of a military software system. The models built by SMART of a large system for command, control, and communications had a level of accuracy at the end of coding that could be very useful to software developers.

Future research will expand the selection of models underlying the module-order model, to include CBR models, and other techniques. Future research will also extend the options for CBR models, and will validate the techniques with larger empirical data sets.

References

[1] J. D. Arthur and S. M. Henry, editors. *Software Process and Product Measurement*, volume 1 of *Annals of Software Engineering*. J. C. Baltzer, 1995.

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

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

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

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

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

[7] M. H. Halstead. *Elements of Software Science*. Elsevier, New York, 1977.

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

[9] T. M. Khoshgoftaar and E. B. Allen. Predicting the order of fault-prone modules in legacy software. In *Proceedings of the Ninth International Symposium on Software Reliability Engineering*, pages 344–353, Paderborn, Germany, Nov. 1998. IEEE Computer Society.

[10] T. M. Khoshgoftaar and E. B. Allen. A comparative study of ordering and classification of fault-prone software modules. *Empirical Software Engineering: An International Journal*, 1999. Forthcoming.

[11] T. M. Khoshgoftaar, E. B. Allen, J. P. Hudepohl, and S. J. Aud. Applications of neural networks to software quality modeling of a very large telecommunications system. *IEEE Transactions on Neural Networks*, 8(4):902–909, July 1997.

[12] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl. Which software modules have faults that will be discovered by customers? *Journal of Software Maintenance: Research and Practice*, 11(1):1–18, Jan. 1999.

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

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

[15] T. M. Khoshgoftaar and D. L. Lanning. An alternative modeling approach for predicting program changes. *Computer Science and Informatics: CSI Journal*, 25(3):25–38, Sept. 1995.

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

[17] R. Kowalski. AI and software engineering. In *Artificial Intelligence and Software Engineering*, pages 339–352. Ablex Publishing, Norwood, NJ USA, 1991.

[18] D. B. Leake, editor. *Case-Based Reasoning: Experiences, Lessons, and Future Directions*. MIT Press, Cambridge, MA USA, 1996.

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

[20] P. Oman and S. L. Pfleeger, editors. *Applying Software Metrics*. IEEE Computer Society Press, Los Alamitos, CA, 1997.

[21] A. A. Porter and R. W. Selby. Empirically guided software development using metric-based classification trees. *IEEE Software*, 7(2):46–54, Mar. 1990.

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